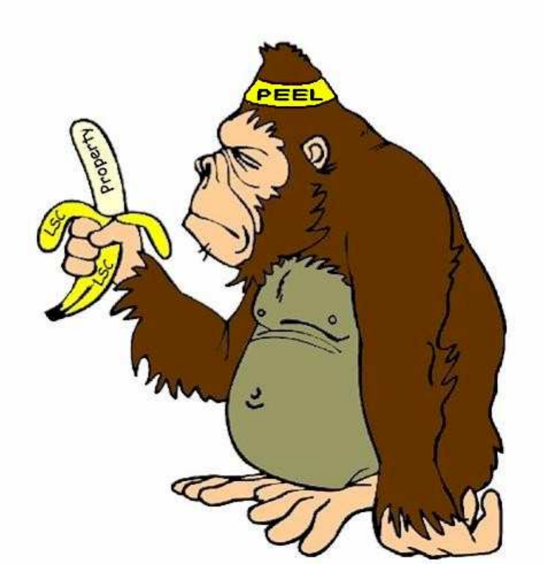
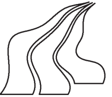
PEEL

Property
Extraction
Engine
for LSCs

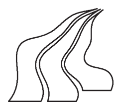
DAT-6 · Master's thesis
1st of June, 2004



Aalborg University
Department of Computer Science



Aalborg University
Department of Computer Science



Title:

Property **E**xtraction **E**ngine for **L**SCs

Theme:
Modelling and Verification Tools for Embedded Programming

Project period:
DAT-6: Master's thesis
1st of Feb., 2004 - 1st of June, 2004

Authors:
Jens Gorm Rye-Andersen
Mads W. Jensen
René Gøttler
Michael Jakobsen

Supervisor:
Anders Peter Ravn

Number of copies: 9

Number of pages: viii + 104

Appendix: 6

Synopsis:

This report presents Live Sequence Charts (LSCs) as a diagrammatic requirements specification language for the formal verification tool UPPAAL, supplementing the already used tree logic language of TCTL.

UPPAAL models are formally described to identify a subset of LSC constructs relevant for UPPAAL model verification. The LSC subset is also formally specified and used in a prototype verification engine PEEL.

Experiments are conducted in order to evaluate LSCs as a requirement specification language for UPPAAL and demonstrate the PEEL prototype.

The experiments show that LSCs are an intuitive way for specifying interobject communication usable for UPPAAL requirements specification. Furthermore, PEEL demonstrates the support of the selected LSC subset.

Preface

This report was written during the spring term of the year 2004. The project was created in collaboration with CISS - Center for Embedded Software Systems, at AAU (Aalborg University). The project report is the authors' Master's thesis in Computer Science.

The report documents an extension of UPPAAL with a diagrammatic approach for requirements specification in the form of Live Sequence Charts, LSCs. A UPPAAL compliant subset of LSC features are identified and formally specified. A prototype verification engine, PEEL, implements support for verifying a UPPAAL model through LSCs.

All first person statements such as "Our opinion..." refer to the authors. All references occur as alphanumeric text in square brackets referring to bibliography entries.

We would like to thank the OFFIS group at the Carl von Ossietzky Universität Oldenburg, especially Bernd Westphal and Bernhard Josko, for providing the LSCEditor with support for use in this project.

We also thank Gerd Behrmann from CISS at Aalborg University for help regarding UPPAAL and for providing the UPPAAL verifyta with support for FSM computation graph output.

If interested in the software implemented and used in this project, please contact Anders Peter Ravn from CISS Aalborg University at <apr@cs.auc.dk>.

Jens Gorm Rye-Andersen   Mads W. Jensen

René Gøttler   Michael Jakobsen

Contents

1 Introduction **1**
- 1.1 Diagrammatic requirements specification . . . 3
- 1.2 Project goals . . . 4
- 1.3 Related work . . . 6
- 1.4 Overview . . . 7

2 UPPAAL **9**
- 2.1 Informal description . . . 9
- 2.2 Formal description . . . 13
- 2.3 Requirements specification language (TCTL) . . . 19
- 2.4 Example . . . 21
- 2.5 Summary . . . 23

3 Live Sequence Charts **25**
- 3.1 LSC constructs . . . 26
- 3.2 Extensions to the basic LSC constructs . . . 38
- 3.3 LSC behaviour . . . 43
- 3.4 Formal description . . . 46
- 3.5 Summary . . . 53

4 PEEL - Property Extraction Engine for LSCs **55**
- 4.1 LSCEditor . . . 56
- 4.2 UPPAAL computation graph . . . 61
- 4.3 Verification algorithm . . . 63
- 4.4 The running time of the algorithms . . . 71
- 4.5 Optimisation . . . 74
- 4.6 Test . . . 76
- 4.7 Summary . . . 79

5 Experiments **81**
- 5.1 Broadcast . . . 81
- 5.2 Train-gate . . . 84
- 5.3 Distributed control . . . 89
- 5.4 Summary . . . 94

6 Evaluation and Conclusion **95**
6.1 Project summary 95
6.2 Evaluation 98
6.3 Future work 101
6.4 Conclusion 103

A UPPAAL Expressions **105**

B PEEL Diagrams **108**

Bibliography **111**

List of Figures

1.1 Sequence diagram from Rhapsody modelling a scenario from a set-top box. 2
1.2 An existential LSC. 4
1.3 Overview of PEEL. 6

2.1 The sender and receiver templates of a Broadcast example. 10
2.2 UPPAAL model of a coffee vending machine. 21
2.3 FSM computation graph for the vending machine example. 22

3.1 LSC with prechart and main chart. 27
3.2 The overlap of prechart and main chart. 28
3.3 An LSC with hot and cold instance lines. 31
3.4 Synchronous and asynchronous messages in an LSC. 32
3.5 Two messages exchanged at the same time. 33
3.6 The types of conditions in LSCs. 34
3.7 An LSC with a coregion. 36
3.8 How coregions are interpreted. 37
3.9 LSC modelling an if-then-else construct. 39
3.10 LSC two simultaneous regions. 40
3.11 LSC illustrating Klose's timer constructs. 42
3.12 LSC with a local invariant. 43
3.13 An LSC divided into snapshots. 44
3.14 The order of the charts to be satisfied by a trace. 48

4.1 Data-flow diagram for PEEL. 56
4.2 Screenshot displaying the LSCEditor. 57
4.3 LSC for the coffee vending machine example. 59
4.4 Map file for the vending machine example. 59
4.5 LSC file for the vending machine example. 60
4.6 FSM graph for the vending machine example 62
4.7 FSM computation graph for the vending machine example. 62
4.8 FSM computation graph traversion. 64
4.9 PEEL sequence output 77
4.10 PEEL LSC structure output 78
4.11 PEEL help output 79

5.1 The sender and receiver templates. 82
5.2 LSC for the broadcast experiment. 82
5.3 LSC prechart and body for the broadcast experiment. 83
5.4 The train template. 84
5.5 The gate template. 85
5.6 The queue template. 86
5.7 LSC prechart and body for Train-gate. 87
5.8 The subcharts for the Train-gate LSC 87
5.9 An existential LSC for the Train-gate experiment. 88
5.10 LSC prechart and body for the queue in Train-gate. 88
5.11 The template of the controller. 89
5.12 The bus template. 90
5.13 The template of the sensor. 91
5.14 The templates of the actuators. 91
5.15 The template of the plant. 92
5.16 LSC prechart for the read-write scenario. 93
5.17 LSC chart specifying the read-write cycle of the Distributed control experiment. 93

6.1 Screenshot of UPPAAL. 102

B.1 Class diagram of the PEEL LSC Parser output. 108
B.2 Class diagram for the sequence of LSC elements. 109
B.3 Class diagram of the PEEL FSM Parser output. 110

1 Introduction

Contents

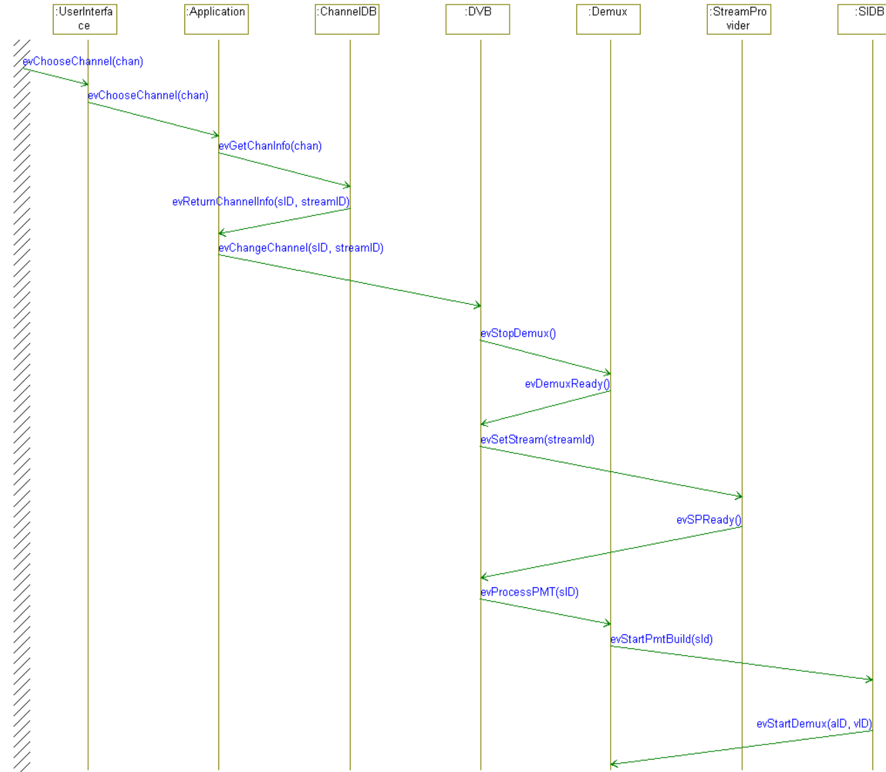
1.1 Diagrammatic requirements specification 3
1.2 Project goals 4
1.3 Related work 6
1.4 Overview 7

The goal of software development is to develop software. But how can we know that the software behaves correct? This can be established by validating and verifying the software.

The first part of this Master's thesis was in collaboration with FUTARQUE, which is a company that develops software and hardware for set-top boxes. The goal was to integrate the CASE (computer-aided software engineering) tool Rhapsody into the existing software development department at FUTARQUE. Rhapsody allows for a model-driven development process, which means that the models themselves (UML models) provide means for analysing problem domains and designing the software. Rhapsody provides means for compiling, testing, validating, executing, and debugging the designed models [RAJGJ03].

An advantage of Rhapsody is the use of UML diagrams, e.g., sequence diagrams (SDs), which is a version of message sequence charts (MSCs). The SD in Figure 1.1 shows a scenario from a set-top box modelled in Rhapsody. The vertical lines are instances of the system, and the messages between the instance lines describe a message trace through the system. SDs can be used for both analysis and testing in Rhapsody.

Rhapsody's *TestConductor*, which uses SDs as its specification method, was found to perform well for system validation. The SDs provide a good visual formalism for specifying scenario requirements, but the system testing is a validation testing only. Validation is the process of running simulations of the system being built, which gives the benefits of exercising the actual system, but at the cost of test coverage, i.e., it is very difficult, if not impossible, to simulate every possible trace through a system. Verification is the process of building a model of the system and performing formal checking of the model, i.e., model-checking. The technique enables testing of all possible traces through a system, but the downside to this



The scenario shows a channel change in the set-top box. First the channel data information is looked up in the `ChannelDB`, then the information is passed to the `DVB` layer. The `DVB` layer first stops the `Demux`, sets channel information in the `StreamProvider`, starts the de-multiplexing and starts parsing DVB packets in the service information database, `SIDB`.

Figure 1.1: Sequence diagram from Rhapsody modelling a scenario from a set-top box.

approach is that it is only a model of the system that is being tested and not the actual system.

Tools for verification of systems are used in the industry today, an example is Statemate MAGNUM from I-LOGIX [i-l04]. Statemate provides an environment for specifying, analysing, designing, documenting, and verifying complex reactive systems [HLN+88]. Spin is another verification system, used for the formal verification of distributed software systems [spi04]. visualSTATE [vis04] from IAR systems is a tool used for generating state machines, which can be documented, simulated, verified, and finally, code-generated through visualSTATE. UPPAAL, which is used in this project, is developed between Uppsala and Aalborg Universities [upp04]. UPPAAL is an integrated tool environment for modelling, validation,

and verification of real-time systems in the form of networks of timed automata. Another verification tool for real-time systems is the KRONOS tool [DOTY95].

Most of the above tools support the developer during all phases of the software development, but the testing phase must be emphasised as this is the final step before releasing the product. When testing a product, a good requirements specification is important as this is the exit criteria for the final product. It therefore seems natural to use the requirements specification to automate the validation and verification of the product. Requirements specification can be done in many ways and different representations exist, e.g., use cases, state machines, MSCs, and in particular LSCs (live sequence charts) seem to be a powerful way to express property specifications with respect to validation and verification. The goal of the requirement specification is to develop a clear and unambiguous understanding of the software to be developed and the requirements specification is the basis for the software testing.

One of the problems with the domain of formal specification is that the methods developed demand that practitioners need to be experts within the property specification language used in order to gain full potential of the tools. It has long been known that computer programming languages are meant for humans to understand and not machines, thus the popularity of high level programming languages, but in formal specification, logic languages are still being used instead of newer diagrammatic requirements specification languages, which can be used as a high level specification language compared to logic languages. [vL00, BS03].

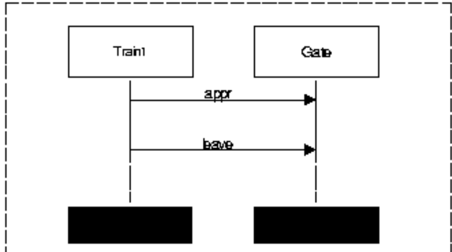
1.1 Diagrammatic requirements specification

A high-level modelling language, which may be used for expressing property specifications, is the Unified Modelling Language (UML) [HT03]. UML is a language for specifying object-oriented software, and it is widely used in the industry. Among other things, UML contains a diagrammatic requirements specification formalism, the MSC.

Unfortunately, MSCs have several deficits. They only show possible and safe behaviour, i.e., they only specify that nothing bad will ever happen, and they cannot express liveness properties that something may happen nor enforce progress through a chart. In addition, the MSC constructs of conditions do not have a formal semantics, i.e., they are merely comments in the chart, whereas conditions are first-class citizens in LSCs and affect the run of a model. A formal semantics is also lacking for the timer durations, and so they are ignored in MSCs as well. Fi-

nally, it is not possible to specify simultaneity in MSCs, i.e., that more than one event may happen at the same time [Klo03, BDW+02, DH01].

LSCs are a graphical notation for specifying temporal relationships between signals and events, and they are an extension of MSCs that provide stronger expressive power in order to make up for the deficits described above, especially concerning liveness, i.e., representation of provisional versus mandatory behaviour in systems [DH01]. Work is being done in relating specifications in LSCs to UML's diagrams [KW02], [DW03], and [dBBGdR03], which indicates that CASE-tools will soon adopt variants of this specification language. Statemate MAGNUM from I-LOGIX has already been extended with support for LSCs [BDW00], and work is in progress for extending Rhapsody as well [BJK+01]. See Figure 1.2 for an example of an LSC.



This example shows a simple LSC which specifies that at least one scenario must exist with the specified messages.

Figure 1.2: An existential LSC.

LSCs enable the developer to construct requirements specifications in a visual fashion and does not require background knowledge in formal validation and verification.

1.2 Project goals

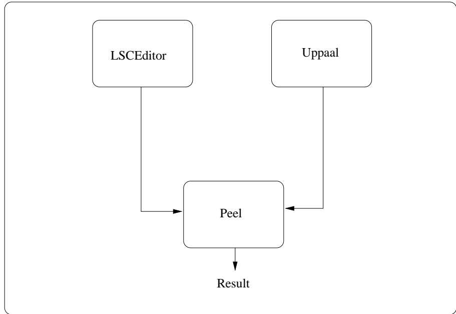
The purpose of this project is to introduce LSCs as a diagrammatic specification language in the formal verification tool UPPAAL. In order to allow LSCs to be used in combination with UPPAAL, the semantics of both feature sets are described informally and additionally the semantics of UPPAAL is formally described in order to determine what properties from UPPAAL models are suitable for specification in LSCs. Based on the feature selection, a formal semantics of the LSC subset is described.

UPPAAL has been chosen because it is a verification tool for timed automata and the behaviour of these automata may be specified by LSCs, which are suitable for interobject communication in the form of messages. Another reason is that UPPAAL uses Timed Computation Tree Logic (TCTL) to specify properties for verification, and TCTL cannot specify message sequences, which is one of LSCs strong features. Furthermore, as UPPAAL is developed in-house, i.e., at Aalborg University, it is an obvious candidate because it is easier to acquire detailed knowledge about the tool.

If the analysis determines that some elements in the languages are not possible to translate such elements are disregarded in the final result. This means that the formal understanding of the subset of LSCs adopted in this report is defined in relation to UPPAAL models, and the LSC elements not in this subset must be prohibited from being used or just ignored when traversing an LSC. Correspondingly, elements in the UPPAAL language, which cannot be captured by LSCs, limit what kind of properties LSCs can be used to specify.

The requirements specification is verified by extracting properties from LSCs and these properties are used to test UPPAAL models. More specifically, the computation tree from UPPAAL is traversed and the LSC properties are used to query this tree. Since LSCs focus on interobject communication, messages and message sequences are the most important properties to test in the UPPAAL models. This is one of the problems with the specification language used in UPPAAL, TCTL, as it has no means of referring to messages. Thus, LSCs have the potential to be a good supplement compared to TCTL with respect to semantic expressibility as well as general user-friendliness.

A prototype engine, PEEL, is implemented to perform the testing of UPPAAL models with LSC specifications. This prototype is based on a graphical LSC editor, and the properties from this editor are extracted and tested against the UPPAAL models, see Figure 1.3. PEEL is applied to a number of experimental cases in order to evaluate PEEL and the diagrammatic approach for specifying properties. The evaluation includes a discussion of advantages and disadvantages of visual specification as opposed to logic languages like TCTL. Furthermore, the application of LSCs as a specification language in combination with the verification tool UPPAAL are evaluated.



This is an overview of the PEEL prototype. PEEL uses an LSC to verify that a UPPAAL model is correct.

Figure 1.3: Overview of PEEL.

1.3 Related work

Several studies regarding usage of LSCs as a specification language have been performed. In [KKR02], LSCs are used as a specification language which introduces a visual environment for designing and validating software systems early in the development process. The authors propose LSCs as a visual means of specifying properties of a system design, so the environment can be used by practitioners unknowing of formal methods. LSCs are translated to AR-automata, which are run in parallel to the design being validated. The automata simulate the design and therefore some features of LSCs, e.g., liveness requirements are not used. The environment helped the authors find several design flaws in a train system they had built. Also, they found that the effort in specifying the requirements had been reduced, which combined with the visual counter-examples shortened their validation cycles significantly. We use LSCs as a visual means of specifying properties as well, but [KKR02] does unlike the work presented in this report not adopt LSCs liveness features. This is because they simulate the LSCs using finite automata for validating the design, whereas we verify behaviour of models and thus need to test for infinite traces. The authors conclude that less effort is required when specifying properties in LSCs. We agree, it is more intuitive to use than TCTL.

A case study used an extension of LSCs to specify parts of an air traffic control system which modelled all scenarios of the system [BHK03]. In fact they found that LSCs are so straightforward that they believe non-technical stake holders can understand and help capture LSCs. We agree with [BHK03] on their remark of how easy it is to construct LSCs, and this is one of the motivations for our project - to ease specification of verification properties.

[BDW+02] presents a methodology for developing train system applications based on Statemate, which is extended to include verification and testing facilities in the form of a model-checker and an automatic generation of test vectors. Also, LSCs as a specification language is included in the extension. The difference from our work is that instead of constructing a verification engine from bottom, we use a small verification engine on top of the existing UPPAAL engine. This requires us to explore the symbolic state space of the UPPAAL model through the computation tree when testing the message sequence. UPPAAL's TCTL is still used for non-message related property checking.

In [BG01] the authors apply LSCs on hardware protocols. They found that LSCs have a significant potential for use when formally specifying hardware standard protocols if ignoring a weakness with the timing model. Instead of having a full timing model, MSCs and LSCs rely on partial order imposed by the order in which events occur along a life line in which messages are passed between processes. They determined that LSCs are not strong enough to formally specify protocol standards, but as LSCs were developed for system level and software design the authors were not surprised of this. The authors want to extend LSCs with a full timing model, thus eliminating the weakness. This weakness is corrected in Klose's dissertation [Klo03], but Because it has been chosen not to use Klose's extension in this report, the LSCs adopted in our LSC subset do not have a full timing model either. The total order of messages is sufficient for specifying the order of the messages and thus message traces.

1.4 Overview

First, an introduction of UPPAAL is presented in Chapter 2. It includes a formal semantics of UPPAAL models and the subset of TCTL used.

Next, LSCs are described informally in Chapter 3 and a relevant subset is described formally to be used in the PEEL engine.

Chapter 4 describes the property extraction engine. Chapter 5 provides experiments of how LSC diagrams can be used to verify UPPAAL automata with the PEEL engine, and finally an evaluation and conclusion of the project is given in Chapter 6 together with directions of future work.

2 UPPAAL

Contents

2.1 **Informal description** **9**
2.1.1 Model constructs 10
2.1.2 Model behaviour 12
2.2 **Formal description** **13**
2.2.1 Network 15
2.2.2 Maximal delay 17
2.3 **Requirements specification language (TCTL)** **19**
2.3.1 Local properties 19
2.3.2 Temporal properties 20
2.4 **Example** **21**
2.5 **Summary** **23**

UPPAAL is a tool for modelling, validating, and verifying real-time systems. It models systems as a collection of non-deterministic processes with finite control structures and real-valued clocks, i.e., network of timed automata. The processes synchronise through channels and may exchange values through shared, finite data structures.

This chapter gives an informal as well as a formal description of UPPAAL and the requirements specification language, TCTL, used. An example of a UPPAAL model is presented, and use of UPPAAL as a model checking tool is introduced.

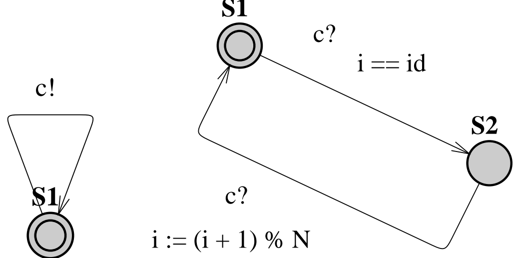
2.1 Informal description

This section introduces the system description language of UPPAAL. The presentation is based mainly on [DMY03] and the documentation accompanying the UPPAAL application. The static and the dynamic elements of a UPPAAL model are described. The static elements are the language constructs, and the dynamic elements are the behaviours of the constructs, i.e., what a configuration is and how a model can progress from one configuration to another.

2.1.1 Model constructs

A UPPAAL model is a network of timed automata. The automata have constructs for holding data, and the automata can synchronise over globally declared channels, either through *binary synchronisation* in the form of a hand-shake or through *broadcast synchronisation*.

Each automaton is represented by a finite, directed graph, where edges are labelled with guards, synchronisation labels, and updates in the form of variable assignments. The nodes in the graph are referred to as *locations*, and the edges as *transitions*. The *initial location* is marked with a double circle. Figure 2.1 shows an example UPPAAL model.



The model shows an example of a UPPAAL network with broadcast. The left is a sender and the right is a receiver model template.

Figure 2.1: The sender and receiver templates of a Broadcast example.

Data constructs

Data in the model are stored in clocks and variables. Clocks contain non-negative real values, i.e., values in the set $\mathbb{R}_{\geq 0}$. Variables can be of type boolean or finite subrange of integer, and be ordered in multi dimensional arrays. Constant values can also be declared.

Integers must be declared with a finite bound, e.g., $[-5, 5]$ in order to restrict the number of configurations, the maximum bound is $[-32768, 32767]$, which is used if no bound is explicitly defined. Boolean variables take the values *true* or *false*, and they are type consistent with integers as in C++, i.e., 0 evaluate to *false* and all other integer values evaluates to *true*. Likewise, *false* evaluates to 0 and *true* evaluates to 1.

Both clocks and variables can be declared locally or globally. Normal scope rules apply, meaning that locally declared names override globally declared names.

Channels

As already mentioned, automata synchronise over channels, but no information is sent over the channels, instead, information may be exchanged through global variables.

A channel can be declared *urgent*, which means that if transitions labelled with this channel are enabled, then no time delay must occur before the enabled transitions are carried out. The type of the channel used for synchronisation decides whether the synchronisation is binary, or if marked with *broadcast*, one-to-many. A channel can be marked as both urgent and broadcast. A binary synchronisation channel is a blocking synchronisation, whereas a broadcast synchronisation is non-blocking, a sending transition does not wait for a recipient, but can always be enabled.

Locations

Locations are the nodes of the directed graph defining an automaton. There are two special types of locations: *urgent* locations and *committed* locations. An urgent location is a location in which time may not progress, until the location is left again. A committed location is an urgent location with an additional constraint; transitions from committed locations have precedence over all other enabled transitions, thus committed locations can be used to model atomicity. The special location types are marked with a U and C for urgent and committed locations, respectively.

It is possible to attach a boolean expression over clocks and variables to a location, an *invariant* that needs to be fulfilled while the location is active. It is not permitted to set lower bounds on clock values in invariants and the *or* and *not* operators are restricted to expressions over integers. Expressions in invariants must be side effect free. Appendix A contains a syntax for invariant expressions.

Transitions

Transitions, being the edges of the directed graph, describe the possible steps from a given location. The steps may be restricted by *guards*, which are expressions that must evaluate to *true* before a transition is enabled, and *synchronisation labels*,

which are labels indicating synchronisation between automata. A transition may furthermore contain *updates*, which are variable and clock assignments.

A guard is a set of side effect free boolean expressions over variables, clocks, and integers, but unlike location invariants they are not restricted from setting lower bounds on clocks. A guard on an urgent channel or on a broadcast receiving channel may not contain any clocks. Appendix A contains a syntax for guards.

Synchronisation labels are channels marked with a direction. The synchronisation scheme in UPPAAL is similar to the synchronisation scheme used in the CCS calculus [LPY97]. Outgoing transitions with $e!$ labels are enabled when there is another outgoing transition with the corresponding label $e?$ at another active location, and neither guards nor location invariants restrict the progress. Because broadcast is non-blocking, a receiver is not necessary in order to enable a transition. The transition marked with $e!$ is evaluated first and the set of receivers last. The evaluation order is the order in which the automata instances are declared in the model. e is a side effect free expression evaluating to a channel, see a BNF with the syntax in Appendix A.

2.1.2 Model behaviour

The behaviour of a model can be seen as a sequence of computation steps from an initial configuration. For an automaton a configuration is a location and a value of clocks and variables. The *initial configuration* of the automaton is the configuration with the active location being the start location and all variables and clocks having the initial values. The combined configurations of all the automata and the values of variables and clocks constitute the configuration of a network of automata.

Transitions may change the active location and the variable values as well as reset clocks. If there is no synchronisation label, a transition is referred to as an *internal transition step*, and when there is a synchronisation label it is either a *binary synchronisation step* or a *broadcast synchronisation step*. When the automaton is in a location time can progress, but some constructs restrict time from progressing:

- Urgent locations,
- Committed locations,
- Urgent channels, and
- Location invariants on clocks.

Thus, a computation step on a model is either an internal transition step or a synchronisation step. The behaviour of a model can now be more precisely defined. It is a set of sequences of configurations, where each sequence starts from the initial configuration, and for every two consecutive configurations, $\sigma[k]$ and $\sigma[k+1]$, there must exist a computation step that converts $\sigma[k]$ to $\sigma[k+1]$. A sequence of configurations is referred to as a *trace*.

For a more in-depth description of all language constructs, the syntax, etc. please consult the help file from the, as of this writing, latest UPPAAL version 3.4.5.

2.2 Formal description

Now that the informal description of the constructs of an UPPAAL model has been given, the formal description is presented. This section is based on the semantical descriptions presented in the documentation accompanying the UPPAAL application, [BLL$^+$95], and [DMY03].

A UPPAAL model is a network of timed automata. A timed automaton is defined as a tuple:

$$\Phi = \langle \mathcal{V}, \mathcal{L}, \mathcal{A}, \Theta, \mathcal{T}, \Pi \rangle$$

consisting of:

- $\mathcal{V} = \mathcal{D} \cup \mathcal{C}$ is the finite set of variables. $\mathcal{V}$ is partitioned into $\mathcal{D} = \{d_1, ..., d_n\}$ the set of data variables, and $\mathcal{C} = \{c_1, ..., c_n\}$ the set of clocks. Data variables are integers or booleans, and they may be defined to be constants, and the clocks are always of type non-negative real, $\mathbb{R}_{\geq 0}$. Variables may be ordered in arrays.

- $\mathcal{L} = \{l_0, ..., l_n\}$ is the finite set of locations. Locations can carry the attribute *urgent* or *committed*. Urgent and committed locations are subsets of $\mathcal{L}$, thus the two sets are denoted by $\mathcal{L}_u$ and $\mathcal{L}_c$, respectively.

  A configuration is $\sigma = (l, v) \in \Sigma$, where Σ is a set of configurations, location $l \in \mathcal{L}$, and a type consistent valuation v of the variables in $\mathcal{V}$.

  A variable assignment is a mapping from clock variables C to the non-negative reals and data variables D to integers or booleans. For a variable assignment v and a delay d, $v \oplus d$ denotes the variable assignment such that $(v \oplus d)(x) = v(x) + d$ for any clock variable x and $(v \oplus d)(i) = v(i)$ for

any integer or boolean variable i. In other words, $\oplus$ changes only the value of clock variables, i.e., time progression only effects clock variables.

- $\mathcal{A} = \{\alpha_1, ..., \alpha_n\}$ is the finite set of synchronisation channels on which a network of automata can synchronise. From the set $\mathcal{A}$ the set $\mathcal{S}$ of synchronisation labels is formed. Synchronisation is either binary or one-to-many, i.e., one automaton sending and one or more receiving. Sending on channel α is denoted by $\alpha!$ and receiving by $\alpha?$. Also, as not all transition steps are synchronising steps, ϵ denotes an internal transition step without any synchronisation. The set of synchronisation labels possible is thus defined as: $\mathcal{S} = \{\alpha?, \alpha!, \epsilon\}$ for $\alpha \in \mathcal{A}$.

  An urgent channel is a special instance of a synchronisation channel on which automata must synchronise as soon as possible, thus the set of urgent channels is $\mathcal{A}_u \subseteq \mathcal{A}$. A channel can also be marked as a broadcast channel, which means that the synchronisation is one-to-many and non-blocking, thus $\mathcal{A}_b \subseteq \mathcal{A}$ where $\mathcal{A}_b$ is the set of broadcast channels. A channel c can be marked as both urgent and broadcast, thus $c \in \mathcal{A}_u \cap \mathcal{A}_b$

- The initial condition Θ specifies a start location $l_0 \in \mathcal{L}$ and an initial value of all variables, $\mathcal{V}_0$.

- $\mathcal{T}$ is the finite set of transitions. Each transition $\tau \in \mathcal{T}$ is a single relation:

$$\mathcal{T} \subseteq \Sigma \times \mathcal{S} \times \Sigma$$

  that relates one configuration and a synchronisation label to another configuration.

- Π is the finite set of location invariants, i.e., invariant conditions that must evaluate to *true* on a location, for that location to be active. The invariants are used to specify local restrictions on the progress of an automaton. An invariant is a boolean expression over variables, $\mathcal{V}$, but it is not permitted to set lower bounds on clock values in invariants.

  The location invariants are described by a total function *inv* that based on a configuration evaluates whether the conditional expression is true or not:

$$\mathit{inv} : \mathcal{L} \times \mathcal{V} \rightarrow \{\mathit{true}, \mathit{false}\}$$

Two additional total functions are introduced, g and u. The function g evaluates the guard condition for a transition based on the values of the variables:

$$g : \mathcal{V} \rightarrow \{\mathit{true}, \mathit{false}\}$$

and the function u evaluates the set of updates carried on a transition:

$$u : \mathcal{V} \rightarrow \mathcal{V}'$$

Three different computation steps are possible: internal transition steps, binary synchronisation steps, and broadcast synchronisation steps. The internal transition step is possible in both a single automaton and a network of automata, whereas the binary and broadcast synchronisation steps are only possible in a network of automata.

An internal transition step from l to l' can be taken, just when it is enabled, i.e., $g(v)$ evaluates to *true* on v and $inv(l', v')$ on v', where the resulting value assignment v' is the result of the update evaluation $u(v)$ and l' is the resulting active location. The internal transition step is defined as:

$$(l, v) \xrightarrow{\epsilon} (l', v') \quad \text{iff } g(v) \wedge \mathit{inv}(l', v') \\ \text{where } v' = u(v)$$

2.2.1 Network

Synchronisation steps can only be performed on the network level, thus before proceeding with the specification of synchronisation steps, the network level needs to be specified.

A configuration of a network of automata, $\overline{\Phi} = \langle \Phi_1, \ldots, \Phi_m \rangle$ is:

$$\overline{\sigma} = (\bar{l}, v)$$

where $\bar{l} = \langle l_1, ..., l_m \rangle$ contains the active locations for all automata and v is a type consistent valuation of the set of variables $\mathcal{V}_1 \cup ... \cup \mathcal{V}_m$.

It follows that the initial configuration of $\overline{\Phi}$ is $(\bar{l}_0, v_0)$, where $\bar{l}_0$ is the initial control vector, i.e., the initial locations of all the automata, and v_0 is the initial valuation of $\mathcal{V}$.

The invariant function must also hold when used on a network of automata. The function evaluates to *true* when the invariants hold for all individual automata:

$$inv : \mathcal{L} \times \mathcal{V} \rightarrow \begin{cases} true & \text{iff } \forall\, l \in \bar{l} : inv(l,v) \\ false & \text{otherwise} \end{cases}$$

A network lifts transitions of the individual automata, thus the possible transitions for the individual automaton is used as the basis for the formal description.

The basic binary synchronisation step performed on a network is defined as:

$$(\bar{l}, v) \xrightarrow{\alpha} (\bar{l}', v') \quad \text{iff } g(v) \wedge inv(\bar{l}', v') \quad \text{where } v' = u(v) \wedge \alpha \in \mathcal{A} \setminus \mathcal{A}_b$$

The basic broadcast synchronisation step is defined as:

$$(\bar{l}, v) \xrightarrow{\beta} (\bar{l}', v') \quad \text{iff } g(v) \wedge inv(\bar{l}', v') \quad \text{where } v' = u(v) \wedge \beta \in \mathcal{A}_b$$

Next follows the specification of the synchronisation steps on the network level, i.e., the changes of all effected automata.

In the internal transition step, i.e., a single automaton takes a transition without any synchronisation, only the automaton in question changes location, and the global set of variable values changes. The internal transition step is defined as:

$$\frac{\langle \ldots, l_i, \ldots \rangle \xrightarrow{\epsilon} \langle \ldots, l_i', \ldots \rangle}{(\langle \ldots, l_i, \ldots, l_k, \ldots \rangle, v) \xrightarrow{\epsilon} (\langle \ldots, l_i', \ldots, l_k, \ldots \rangle, v')} \quad \text{iff } g_i(v) \wedge inv(l_i', v') \wedge inv(l_k, v') \quad \text{where } v' = u_i(v)$$

for an internal step in a automaton Φ_i.

When a binary synchronisation step occurs two automata change location and there is an update of the global variables as well. The sender updates the variables first and the receiver updates last:

$$\frac{\langle ...,l_i,...\rangle \xrightarrow{\alpha!} \langle ...,l_i',...\rangle \quad \langle ...,l_k,...\rangle \xrightarrow{\alpha?} \langle ...,l_k',...\rangle}{(\langle ...,l_i,...,l_k,...\rangle, v) \xrightarrow{\alpha} (\langle ...,l_i',...,l_k',...\rangle, v')} \quad \begin{array}{l}\text{iff } g_i(v) \wedge g_k(v) \wedge \mathit{inv}(l_i', v') \wedge \mathit{inv}(l_k', v') \\ \text{where } v' = u_k(u_i(v)) \wedge \alpha \in \mathcal{A} \setminus \mathcal{A}_b\end{array}$$

for a sending automaton Φ_i and a receiving automaton Φ_k.

The broadcast synchronisation step has one sender and possible many receivers that all change locations and the global variable values also change. The sender updates first, and the receivers update in the same order as they have been defined in the model:

$$\frac{\langle ...,l_i,...\rangle \xrightarrow{\beta!} \langle ...,l_i',...\rangle \quad \langle ...,l_k,...\rangle \xrightarrow{\beta?} \langle ...,l_k',...\rangle \cdots \langle ...,l_n,...\rangle \xrightarrow{\beta?} \langle ...,l_n',...\rangle}{(\langle ...,l_i,...,l_k,...,l_n\rangle, v) \xrightarrow{\beta} (\langle ...,l_i',...,l_k',...,l_n'\rangle, v')}$$

$$\begin{array}{lll} \text{iff} & & g_i(v) \wedge g_k(v) \wedge \ldots \wedge g_n(v) \\ & \wedge & \mathit{inv}(l_i', v') \wedge \mathit{inv}(l_k', v') \wedge \ldots \wedge \mathit{inv}(l_n', v') \\ \text{where} & & v' = u_n(\ldots(u_k(u_i(v)))) \\ & \wedge & \beta \in \mathcal{A}_b \end{array}$$

for a sending automaton Φ_i and receiving automata $\Phi_k \ldots \Phi_n$.

It is also possible to have an empty set of receivers of a broadcast synchronisation, as the broadcast synchronisation is a non-blocking synchronisation call:

$$\frac{\langle ...,l_i,...\rangle \xrightarrow{\beta!} \langle ...,l_i',...\rangle}{(\langle ...,l_i,...,l_k,...\rangle, v) \xrightarrow{\beta} (\langle ...,l_i',...,l_k,...\rangle, v')} \quad \begin{array}{l}\text{iff } g(v) \wedge \mathit{inv}(l_i', v') \wedge \mathit{inv}(l_k, v') \\ \text{where } v' = u(v) \wedge \beta \in \mathcal{A}_b\end{array}$$

for a sending automaton Φ_i. Here, only the sender changes location and the global variable set also change.

2.2.2 Maximal delay

A side condition *MD* is given on all transitions, which returns the maximal delay allowed. If a process modelled by an automaton Φ is in a location l with a number of outgoing transitions with guards, the process may have to wait for these guards to become *true* in order to leave l. It is not desirable that the process waits in this location forever, thus some discrete transition must be taken within a certain time

bound. This bound should be the maximal time before all the guards are completely closed, i.e., they will never become *true* again. If the active location is an urgent or a committed channel, no time delay is allowed.

The maximal delay for an automaton is formalised as:

$$\mathit{MD}(l,v) = \begin{cases} 0 & \text{iff } \exists\, l \in \mathcal{L}_u \cup \mathcal{L}_c \\ max\{d | (l,v) \xrightarrow{\mathcal{S}} (l',v')\} & \text{otherwise} \\ \quad \text{iff } g(v \oplus d) & \\ \quad \wedge\ inv(l, v \oplus d) \wedge inv(l', v' \oplus d) & \\ \quad \text{where } v' = u_s(v) & \end{cases}$$

Also, $\mathit{MD} = 0$ when a configuration (l, v) has all outgoing transitions from the active location l completely closed.

Network

Next, the notion of maximal delay is extended to networks of automata. This insures that synchronisation on urgent channels happens immediately.

The maximal delay for a network of automata is formalised as:

$$\mathit{MD}(\bar{l},v) = \begin{cases} 0 & \text{iff } \exists\, \alpha \in A_u, l_i, l_j \in \bar{l} : \\ & (l_i, v) \xrightarrow{\alpha!} (l'_i, v') \;\&\; (l_j, v) \xrightarrow{\alpha?} (l'_j, v') \\ & \text{where } v' = u_j(u_i(v)) \\ & \vee\ \exists\, \beta \in A_u \cap A_b, l_i \in \bar{l} : \\ & (l_i, v) \xrightarrow{\beta!} (l'_i, v') \\ & \text{where } v' = u_i(v) \\ min\{\mathit{MD}(l,v) | l \in \bar{l}\} & \text{otherwise} \end{cases}$$

If a synchronisation channel α or β is an urgent channel, and the locations with the urgent channel is in the currently active set, then the maximal delay is 0, i.e., no delay is permitted. If on the other hand, there is no urgent channels pending on the active locations, then the shortest of all maximal delays of the processes of the network is the maximal delay, because this is the maximal allowed delay before some action must be taken.

2.3 Requirements specification language (TCTL)

UPPAAL's model-checker is able to verify certain properties such as reachability, i.e., whether certain configurations of the model are reachable from the initial configuration. This is done by constructing a timed computation tree and applying graph algorithms in order to examine the tree. The formulae that the model-checker is to prove or disprove are defined in a requirement specification language, which is a subset of branching time computational tree logic (TCTL) [DMY03, Dav03], where the time part comes from the possibility of constraints on clocks.

A TCTL formula consists of two parts; a local property and a temporal property. The local property describes a property that is to hold for a single configuration, whereas the temporal property specifies to what extend, in traces and configurations, the local properties are to hold, i.e., for all or a single trace, and for all or a single configuration within the given traces.

2.3.1 Local properties

A local property is specified by boolean expressions over location names, clocks, and data variables of the UPPAAL model in question.

Besides the standard relational operators, $<$, $<=$, $==$, $!=$, $>=$, and $>$ as well as the boolean operators *and*, *or*, *not*, and *imply*, the expressions may contain the special keyword *deadlock*, which evaluates to *true* only in the case of a deadlock, i.e., no transitions from the active location are enabled. Specific locations are referred to as $\Phi.l$, where Φ is a timed automaton in the UPPAAL model and l is a location in Φ. A local property has the form specified in the following BNF:

$$
\begin{array}{rll}
p ::= & \textit{deadlock} & \\
| & \Phi.l & \text{for } \Phi \in \overline{\Phi} \wedge l \in \mathcal{L}_\Phi \\
| & c \bowtie x & \text{for } c \in \mathcal{C},\ \bowtie \in \{<,<=,==,!=,>=,>\}, x \in \mathbb{Z} \\
| & c_1 - c_2 \bowtie x & \text{for } c_1, c_2 \in \mathcal{C},\ \bowtie \in \{<,<=,==,!=,>=,>\}, x \in \mathbb{Z} \\
| & d_1 \bowtie d_2 & \text{for } d_1, d_2 \in \mathcal{D} \cup \mathbb{Z},\ \bowtie \in \{<,<=,==,!=,>=,>\} \\
| & (p) & \text{for a local property } p \\
| & \text{not } p & \text{for a local property } p \\
| & p_1 \text{ and } p_2 & \text{for local properties } p_1 \text{ and } p_2 \\
| & p_1 \text{ or } p_2 & \text{for local properties } p_1 \text{ and } p_2 \\
| & p_1 \text{ imply } p_2 & \text{for local properties } p_1 \text{ and } p_2
\end{array}
$$

A local property evaluates to either *true* or *false* based on the configurations of the model. As the intuitive idea of the semantics of the above constructs should be clear, the semantical evaluation of the constructs are formally described.

Formal description

Given a configuration $\overline{\sigma} = (\bar{l}, v)$, a local property p holds in $\overline{\sigma}$ denoted $\overline{\sigma} \models p$, based upon the following rules:

$$
\begin{array}{lll}
\overline{\sigma} \models \text{deadlock} & \text{iff} & \text{no delay or action transitions are enabled in } \overline{\sigma} \\
\overline{\sigma} \models \overline{\Phi}.l & \text{iff} & l = l_i \in \bar{l} \text{ for } \overline{\Phi} = \overline{\Phi}_i \in \overline{\overline{\Phi}} \\
\overline{\sigma} \models c \bowtie x & \text{iff} & v(c) \bowtie x,\ \bowtie \in \{<,<=,==,!=,>=,>\} \\
\overline{\sigma} \models c_1 - c_2 \bowtie x & \text{iff} & v(c_1) - v(c_1) \bowtie x,\ \bowtie \in \{<,<=,==,!=,>=,>\} \\
\overline{\sigma} \models d_1 \bowtie d_2 & \text{iff} & v(d_1) \bowtie v(d_2),\ \bowtie \in \{<,<=,==,!=,>=,>\} \\
\overline{\sigma} \models (p) & \text{iff} & \overline{\sigma} \models p \\
\overline{\sigma} \models \text{not } p & \text{iff} & \neg(\sigma \models p) \\
\overline{\sigma} \models p_1 \text{ or } p_2 & \text{iff} & \overline{\sigma} \models p_1 \vee \overline{\sigma} \models p_2 \\
\overline{\sigma} \models p_1 \text{ and } p_2 & \text{iff} & \overline{\sigma} \models p_1 \wedge \overline{\sigma} \models p_2 \\
\overline{\sigma} \models p_1 \text{ imply } p_2 & \text{iff} & \neg(\overline{\sigma} \models p_1) \vee \overline{\sigma} \models p_2
\end{array}
$$

where $c \in \mathcal{C}$, , $x \in \mathbb{Z}$, and $d \in \mathcal{D} \cup \mathbb{Z}$.

2.3.2 Temporal properties

Temporal properties specify when a local property is to hold during a computation, in other words, they specify some or all configurations in the computational tree that is to satisfy some local property p. The two quantifiers $\forall$ and $\exists$ refer to both paths and configurations of the computation tree. The path aspect of a requirement formula is specified by A and E, and the configuration aspect is specified by [] and <>, for $\forall$ and $\exists$, respectively. By combining the path and configuration aspects it is possible to specify the following four types of formulae, where p is a local property; A[], A<>, E[], and E<>. The specification language also provides a fifth type, a *leads to* property. It is written as `-->`, and it is semantically equal to $A[]\ (p_1\ imply\ A <>\ p_2)$.

The five temporal property types evaluate to either *true* or *false* based on the evaluation of the local property for each of the configurations specified by the temporal property. This means that if some TCTL formula is to hold for some UPPAAL model $\overline{\Phi}$, the local property p is to hold for the set of configurations specified by the temporal properties. Intuitively, the semantics are:

A[] p	for all paths p holds for all configurations, i.e., p invariantly holds.
A<> p	for all paths p holds for some configuration, i.e., p is inevitable.
E[] p	for some path p holds for all configurations, i.e., p may always hold.
E<> p	for some path p holds for some configuration, i.e., p is reachable.
p_1 --> p_2	p_1 eventually leads to p_2.

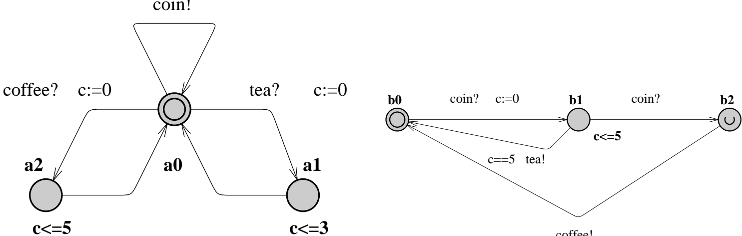
Formal description

Let $\Psi(\overline{\Phi})$ be the set of finite timed traces of the model $\overline{\Phi}$, a trace in $\Psi(\overline{\Phi})$ is $\overline{\sigma}[1, K]$ of length K. Letting the k'th configuration in a trace be denoted by $\overline{\sigma}[k]$, the following rules define the semantics of the temporal properties:

$\overline{\Phi} \models \text{A[]}\ p$	iff	$\forall\, \overline{\sigma}[1, K] \in \Psi(\overline{\Phi}) : \forall\, k \leq K : \overline{\sigma}[k] \models p$
$\overline{\Phi} \models \text{A<>}\ p$	iff	$\forall\, \overline{\sigma}[1, K] \in \Psi(\overline{\Phi}) : \exists\, k \leq K : \overline{\sigma}[k] \models p$
$\overline{\Phi} \models \text{E[]}\ p$	iff	$\neg(\overline{\Phi} \models \text{A<> not}\ p)$
$\overline{\Phi} \models \text{E<>}\ p$	iff	$\neg(\overline{\Phi} \models \text{A[] not}\ p)$
$\overline{\Phi} \models p_1 \texttt{-->} p_2$	iff	$\forall\, \overline{\sigma}[1, K] \in \Psi(\overline{\Phi}) : \forall\, k \leq K :$ $\overline{\sigma}[k] \models p_1 \Rightarrow \exists\, k' \geq k : \overline{\sigma}[k'] \models p_2$

2.4 Example

This section contains an example of a UPPAAL model and TCTL property formulae. The example is a coffee and tea vending machine, see Figure 2.2.



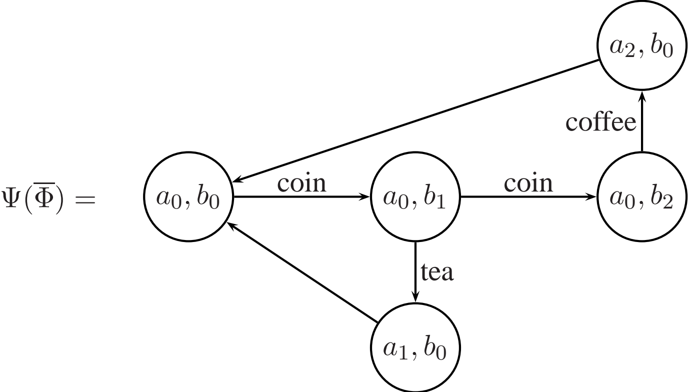
The `coffee_unit` sends coins to the `control_unit` and servers a cup of tea or coffee, when informed to. A cup of tea is server within three time units, $c <= 3$ and a cup of coffee within five time units, $c <= 5$. The `control_unit` dispenses either a cup of tea or coffee, depending on the amount of coins inserted. One coin dispenses a cup of tea if a second coin is not inserted within five time units, see invariant $c <= 5$. Location `b2` is urgent, meaning that when a second coin is received no delay is permitted before the coffee message is sent.

Figure 2.2: UPPAAL model of a coffee vending machine.

Computation graph

When UPPAAL formally verifies a network of automata, it takes advantage of various internal data representations. The data representation are not described here, see [BBD+02] for information about the internal data representations and their usage.

In order to be able to check for the existence of a message trace a computation graph is sufficient. The UPPAAL verification engine, *verifyta*, has support for outputting a UPPAAL model as a finite state machine, FSM - note that this is an internal unpublished version of verifyta. Clocks are not represented in the FSM format but data variables effect the state space. Figure 2.3 shows the FSM output as a computation graph for the coffee vending machine example.



The graph shows the UPPAAL location sets as computation configurations. There are no variables in the UPPAAL model used in this example. If there were variables, the possible variable values would be a part of the configurations, e.g., a variable with a range of three in every UPPAAL model location could triple the number of configurations.

Figure 2.3: FSM computation graph for the vending machine example.

UPPAAL TCTL queries

Next follows some example TCTL queries that can be checked via UPPAAL for the vending machine example.

- `A[] not deadlock`: This expression checks the model for deadlocks. If there is a possibility for a deadlock in the model, this query returns *false*.

- `control_unit.b1 --> ( coffee_unit.a1 or coffee_unit.a2 )`: The leads to operator tests for the possibility that a local property is eventually followed by some other property. This example tests that if the `control_unit` is in location `b1`, i.e., one coin has been deposited, then the model will eventually be in location `a1` or `a2`, which means that either a cup of tea or coffee is dispensed.
  The leads to operator must be used with caution, as it is easy to create a *false true* expression, i.e., an expression that is satisfied, but not because a property is eventually followed, but because the first property is never fulfilled. Consider `control_unit.b1 and coffee_unit.a2 -->(coffee_unit.a1 or coffee_unit.a2)` is always satisfied, because location `b1` and `a2` do never coincide.

2.5 Summary

UPPAAL is a tool for modelling, simulating, and verifying non-deterministic timed finite state automata. The supported data types are integers, booleans, and clocks. Synchronisation between processes are performed through channels, either in the form of a blocking binary one-to-one synchronisation, or a non-blocking broadcast one-to-many synchronisation. Communication, i.e., exchange of information must be performed through shared variables.

The verification of UPPAAL models is performed through the querying of a subset of TCTL expressions. TCTL expressions consist of two properties, a local property and a temporal property. The local property is a boolean expression over automata locations and conditions, whereas the temporal property expresses the range of the local property, i.e., to what extend in traces and configurations the local property must be satisfied.

The modelling language of UPPAAL, and the requirement specification language, TCTL, are specified informally as well as formally in order to be able to compare functionality with LSCs. The comparison is used to identify properties of LSCs that can be used to verify UPPAAL models.

Next chapter gives an introduction to LSCs, a limitation of LSC features, and description of how they are used to specify properties for UPPAAL model verification. The semantics of the selected LSC subset is formally described.

3 Live Sequence Charts

Contents

3.1 LSC constructs 26
3.1.1 Precharts and activation conditions 26
3.1.2 Universal and existential charts 28
3.1.3 Temperature 29
3.1.4 Instances 29
3.1.5 Messages 31
3.1.6 Conditions 34
3.1.7 Subcharts 35
3.1.8 Coregions 36
3.2 Extensions to the basic LSC constructs 38
3.2.1 If-then-else 38
3.2.2 LSC activation mode 39
3.2.3 Simultaneous regions 40
3.2.4 Time 41
3.2.5 Local invariants 42
3.3 LSC behaviour 43
3.4 Formal description 46
3.4.1 Semantics 48
3.5 Summary 53

As with MSCs, LSCs allow the user to specify scenarios by describing the inter-object communication. MSCs are often used in the early development stages for capturing use cases through scenarios, but as development progresses and the designers gain more knowledge about the problem domain as well as confidence in their ideas, scenarios characterising the use cases are discovered and defined. This leads to a desire for more expressive ways of modelling the use cases. LSCs are an extension of MSCs introducing among other things liveness, i.e., specification of mandatory versus possible behaviour.

This chapter presents an informal description of LSC features. The first part of the informal description is based on the original LSC description by Werner Damm and David Harel [DH01]. Extensions introduced by Klose in [Klo03] are considered. The graphical LSC notation used is that from the LSCEditor.

The informal description is partitioned into LSC constructs and LSC behaviour, the former being the constructive elements that LSCs are built from, and the latter being how these constructs are used to specify a model behaviour.

A subset of the LSC elements is selected based on what features are relevant for UPPAAL verification, and what elements that can be modelled by other elements. The selected LSC subset is used to reason about the application of LSCs as a requirements specification language for UPPAAL model verification. The limitation and application of LSC features are described when they are introduced.

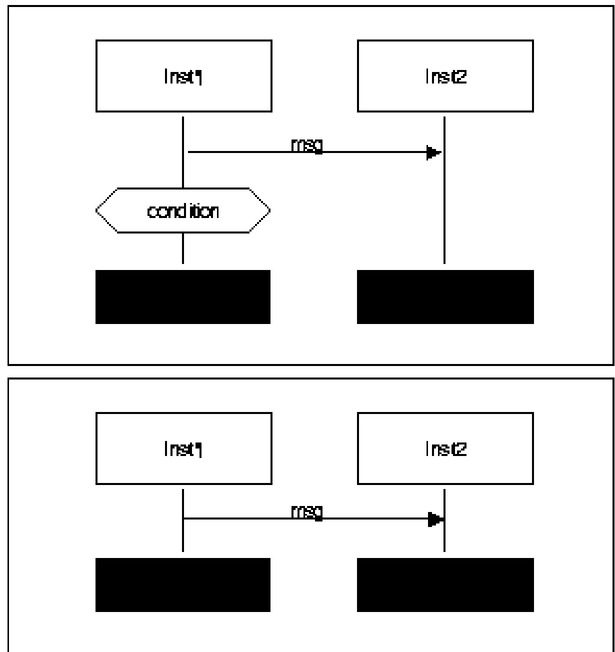
Finally, this chapter presents a formal semantics of the selected LSC subset.

3.1 LSC constructs

An LSC is a chart with instances, i.e., objects, along the horizontal axis and time down the vertical axis, see Figure 3.1. Interobject behaviour is specified as either messages or method calls between the instances. An LSC consists of several constructs used to define the interobject behaviour of a system.

3.1.1 Precharts and activation conditions

To define when a chart is to become active, i.e., when the system should start behaving according to the chart, each chart is coupled with an initial condition. This condition can be in the form of an *activation condition* (as in MSCs) reflecting some configuration of the system, but it can also be defined as a chart of its own, called a *prechart* as in Figure 3.1. In the case of a prechart it is required that the system exhibits the entire behaviour defined in the prechart before the chart itself becomes active. Both an activation condition and a prechart can be specified for a given chart.



An LSC with two instances. The top chart is a prechart and the bottom chart is a main chart.

Figure 3.1: LSC with prechart and main chart.

Limitation

An activation condition can be modelled by having a prechart with only a condition, see below for description of conditions. Thus, the activation condition construct is disregarded.

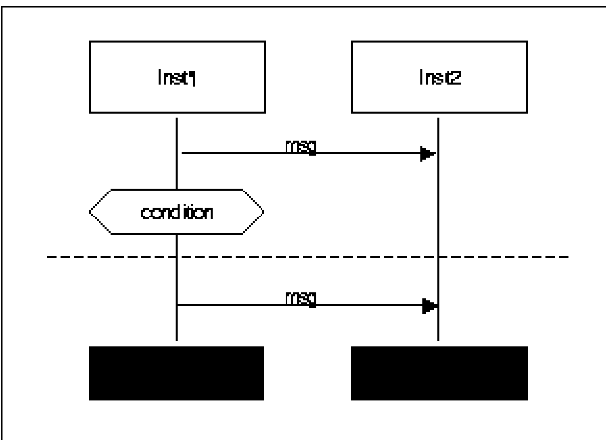
Application

Precharts are tested by traversing the UPPAAL computation graph. When a prechart is satisfied, the main chart is tested from the configuration reached by evaluating the prechart. Figure 3.2 displays the connection of the prechart and main chart of Figure 3.1, the condition must hold in the snapshot from the first message until the dashed line, and from the dashed line until the next message it is another snapshot, i.e., snapshots do not overlap from one chart to another.

Snapshots are used to divide a chart into segments, see Figure 3.3 on page 31. The temperatures of the segments at a given time are collected in the snapshots, and the temperature of the snapshot is hot if any segment in the snapshot is hot,

otherwise it is cold. Snapshots correspond to a sequence of configurations in the FSM computation graph and they are more precisely defined in Section 3.3.

A prechart supports the same constructs as a main LSC, which means that the following descriptions of LSC constructs must be applied.



The prechart and main chart of Figure 3.1 has been merged to illustrate that any conditions must hold until the dashed line, from where a new snapshot is in effect. This means that there are four snapshots in this LSC.

Figure 3.2: The overlap of prechart and main chart.

3.1.2 Universal and existential charts

An LSC has a chart mode that can be either *universal* or *existential*. An existential chart requires the existence of a trace satisfying the chart, while a universal chart requires that all traces conform to it. Generally, universal charts are more restrictive, and it should be possible to extract more restrictive properties from them. This is also the case, as existential charts yield a temporal property of *a trace exists*, E, and universal charts yield a temporal property of *for all traces*, A, see Table 3.1 on page 45. When drawing a universal chart, the box around the chart is a solid line, whereas a dashed line is used for existential charts.

Application

The chart modes, i.e., existential and universal, are handled such that a message trace in an existential chart only needs to exist, whereas a message trace in a universal chart must exist in all traces. This means that if a single trace in a computa-

tion graph does not conform to the specification in a universal chart, the chart does not hold.

3.1.3 Temperature

In the original LSC description in [DH01], a *temperature* is associated with locations, conditions, and messages. The temperature can be either *hot* or *cold*. *Locations* in an LSC are those points on an instance axis where, e.g., messages and conditions are attached. Making a location hot means that it has to be left, thus enforcing progress down the instance axis. Conversely, a cold location never needs to be left, and thus the next location may never be reached.

In the LSC subset adopted in this project, temperatures are not associated to locations, instead coregions and segments have a temperature. The meaning of the temperature notation for the various LSC elements are summarised in Table 3.1 on page 45.

The limitations and applications regarding the temperatures for the LSC construct are given for each construct as they are introduced.

3.1.4 Instances

Instances are the elementary building blocks of LSCs. Their graphical representation has been adopted directly from MSCs, thus instance lines consist of an instance head with the instance name, the instance end as a black box, and an instance axis which is a vertical line connecting the head and the end. As in MSCs the horizontal dimension represents the structural dimension, while the vertical dimension corresponds to the time dimension. Instances are model objects and an instance of a modelled system is a data-space induced by variable declarations and events. Variables used in LSCs may be globally or locally declared in the modelled system. Events may be conditions, sending and reception of messages, or creation and destruction of instances.

Hot instance segments, a segment is a line segment between two events, along an instance line means that progress along the instance line is mandatory. Utilising this feature progress is forced down the instance, but as soon as the instance line reaches a cold segment, progress is no longer enforced. A cold segment means that the next event never needs to be reached.

Messages not originating from one of the objects modelled as an instance line in an LSC are said to come from the *environment*, which can be either non-modelled objects or an external stimuli. An instance line for 'some other stimuli' in the model is thus introduced. It is labelled *environment* and like normal instance lines it may communicate with the instances of the chart.

Limitations

The processes in UPPAAL models are created at model construction time and exist throughout a model's life-time, this means that explicit object construction and object destruction is not possible in a UPPAAL model, thus it will not be considered.

To limit the scope of the features to be implemented, the environment instance is disregarded. This means that if there is communication between an instance and an object not represented in the chart, the communication is ignored as only the interobject behaviour of the instances in the chart are of interest.

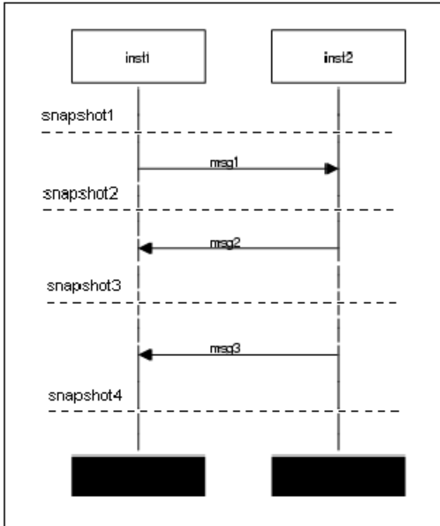
Application

When looking at a message trace, the temperatures of the segments do not matter. Even though there are some cold instance line segments the message trace must still exist, albeit it might never be completed in an actual model execution.

If the temperature of any snapshot is hot, progression is enforced for all instance lines in the given snapshot. In a universal chart this property can be tested with the *leads to* operator in TCTL, because if a snapshot is hot, then computation from the first snapshot must eventually progress to the second snapshot, i.e., the second snapshot must eventually be reached from the first.

Figure 3.3 illustrates temperatures of instance lines through a simple universal LSC with communication between two instances, and both hot and cold segments. First, the message sequence is tested, i.e., as described in Section 3.1.5 below. Second, the temperature of the segments can be used to generate properties for the chart.

In `snapshot1` both `inst1` and `inst2` are hot, so `msg1` has to be reached, and thereby `snapshot2` also has to be reached. In `snapshot2` only `inst2` is hot, but it forces all other instances, i.e., `inst1`, to progress. In `snapshot3` both instance line segments are cold, meaning that progression is not guaranteed. The possibility of progression must still exist, i.e., the message `msg3` must still be possible to be



The snapshots collect the temperatures of the segments in the snapshot. If a segment in the snapshot is hot, so is the snapshot, otherwise it is cold.

Figure 3.3: An LSC with hot and cold instance lines.

sent from `inst2` to `inst1`, and this is checked via the message trace check.

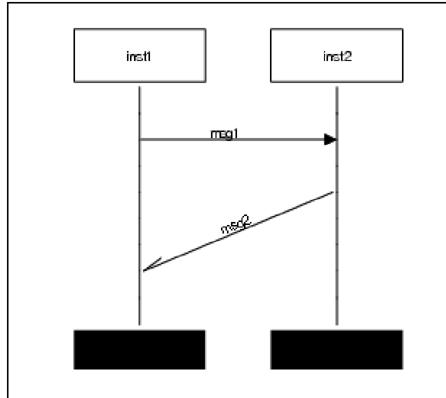
3.1.5 Messages

A message in an LSC is basically the same as a message in an MSC. Two kinds of messages are considered:

- *Synchronous:* The sender blocks until the receiver is ready to receive the message. Sending and reception of the message happens simultaneously. A horizontal arrow is used to denote a synchronous message. A delay from trying to send a message to the acceptance of the request should be expressed on the vertical axis as progress of time on the sender's side.

- *Asynchronous:* Time may pass between sending and reception of the message. After the message has been sent, progress may continue along the instance line before the message is received. An asynchronous message is graphically expressed with a slanted arrow.

Figure 3.4 shows the graphical notation of messages.



Whether a message is synchronous or not is specified through its angle with the horizon. msg1 is synchronous, as it is parallel with the horizon, and msg2 is asynchronous as the message is slanted.

Figure 3.4: Synchronous and asynchronous messages in an LSC.

Interobject method calls are represented by two synchronous messages; the method call and the return message, and the two messages are paired by widening the instance lines carrying the method body. A hot message is a message that, when sent, must be received, and a cold message means that after the message is sent, it is not required to be received.

Limitations

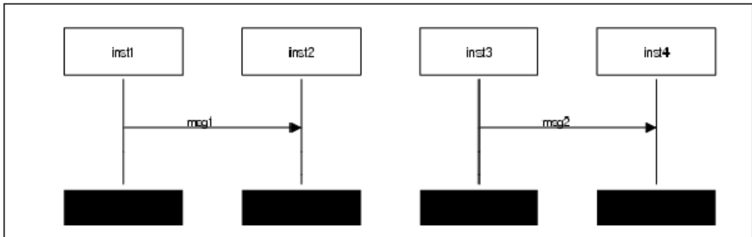
There are two categories of messages in UPPAAL models; binary and broadcast, both categories are synchronous, i.e., sending and reception is instantaneous, thus asynchronous messages in LSCs are discarded.

The binary messages in a UPPAAL model are handshakes, which means that there are always a sender and a receiver. Broadcast messages on the other hand do not need to have a receiver. This means that binary messages in the supported LSC features must be hot, i.e., if a message is sent it must be received. It also means that it is possible to have cold messages, but then the message must be a broadcast and it must be specified in a simultaneous region, whereas a single hot broadcast message needs not be specified in a simultaneous region. See description of simultaneous regions in Section 3.2.3.

Automata in UPPAAL cannot send messages to themselves, thus such messages are not included. If they were, it would be to describe internal action steps, but if

these steps were to be included in a specification it would end up being a complete description of an automata trace too detailed for a requirement specification.

In LSCs it is possible to have two messages at the same level that are not sent from the same simultaneous region. An example of this can be seen in Figure 3.5, where `msg1` and `msg2` are exchanged simultaneously.



Specifying simultaneity by two messages at the same time like this is disallowed. Any simultaneity must be specified through simultaneous regions.

Figure 3.5: Two messages exchanged at the same time.

UPPAAL supports sending of binary messages simultaneously using urgent locations, but because the computation graph does not contain clocks this cannot be tested. Thus, messages at the same vertical level is not supported, unless they are in a simultaneous region.

Application

When testing UPPAAL models using TCTL it is not possible to test whether a sequence of messages occurs in a trace, because there are no constructs in TCTL referring to messages. The main focus of TCTL is automata locations and variable values in locations, in other words configurations in UPPAAL models.

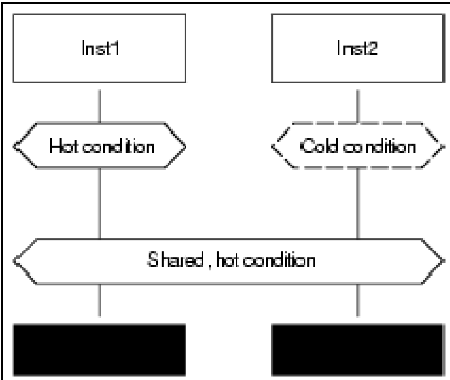
To test message sequences the messages are extracted from the LSC and the computation graph is traversed to prove whether the sequence exists.

If a broadcast message only has one receiver in the chart it may be specified by a hot message, but if the message has several receivers each receiver must be specified using a simultaneous region sending the same messages to all receivers, Hot messages are used when the receiver must receive it, and cold messages are interpreted as a broadcast that may be received. A broadcast message is always sent to all objects, but if a message is not specified in the LSC from a simultaneous region to an instance, then the broadcast message may not be received by that instance.

3.1.6 Conditions

Conditions may terminate a chart successfully or unsuccessfully depending of their temperature. A hot condition must be satisfied or the run is terminated unsuccessfully. A cold condition does not have to be satisfied, if it does not hold it simply means that the rest of the chart is disregarded, which is considered a successful termination of the chart.

Graphically, a hot condition is depicted as a solid hexagon, and a cold condition is a hexagon with a dashed line, as seen in Figure 3.6. A condition can span more than one instance meaning that the condition is specified to hold for all spanned instances. Such a *shared condition* may be used to synchronise instances because the condition will not be evaluated before all instances have reached the condition, and no instance will progress beyond the condition before it has been evaluated. The rules for hot and cold conditions apply to a shared condition in the same way as they do for normal conditions.



Hot conditions are given as solid hexagons and cold conditions are given as dashed hexagons. Conditions may be shared over several instance lines.

Figure 3.6: The types of conditions in LSCs.

A useful feature of conditions is the ability to construct forbidden scenarios. A prechart with the forbidden scenario may be combined with a universal chart containing just a single hot condition always evaluating to *false* [Bjø04].

Limitations

It is possible in LSCs to attach a condition to a simultaneous region and it is possible to have an isolated condition. Klose recommends always attaching a condition

to a simultaneous region because that gives a definite evaluation point for the condition [Klo03][pp. 118-123]. We choose to not differentiate between a condition attached to a simultaneous region and an isolated condition. A condition is to hold for a snapshot and should be evaluated for all possible UPPAAL configurations in that snapshot. As an isolated condition is intuitively closer to our definition of conditions, only isolated conditions are allowed.

Application

As with activation conditions, conditions appearing within a chart are boolean expressions over UPPAAL variables and automata locations.

Conditions are expressed as local TCTL properties and the UPPAAL verification engine is used for querying these properties. Each condition on an instance line in a universal chart is checked with a TCTL formula with the following form for each configuration in the snapshot:

A[] `configuration` **imply** `condition`

If the chart is existential, the formula is:

E<> `configuration` **and** `condition`

Shared conditions are conditions that span more than one instance line. A shared condition is a snapshot is the same as a single condition on all instances, thus a shared condition in an LSC is translated into single conditions.

If a hot condition evaluates to *false* the chart is terminated unsuccessfully, i.e., the chart does not hold. If a cold condition evaluates to *false* the LSC verification engine must ignore the rest of the chart, but otherwise report that the chart is satisfied.

3.1.7 Subcharts

A chart may include another chart, which is specified over a set of instances that may or may not be present in the parent chart. Subcharts may include itself, allowing infinite iteration and thus an infinite number of elements to be verified. Subcharts introduce chart scope, which means that cold conditions not satisfied only

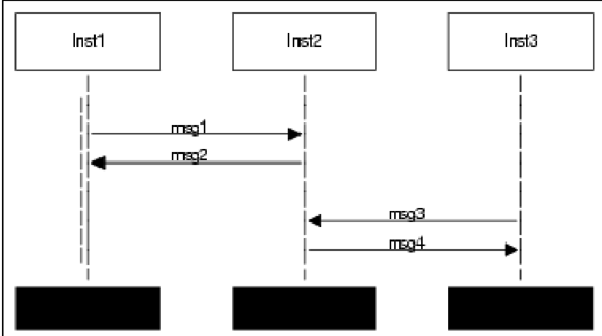
exit the subchart and not the parent chart. Combining subcharts and cold conditions yields classical if-then constructs and loop structures such as while-do and repeat-until. If-then constructs are simply subcharts beginning with cold conditions, and the loop structures are based on subchart iterations.

Limitation

As the LSC editor used in this project does not support subcharts in this form, they are not used. The LSCEditor supports an if-then-else construct, and as subcharts can be seen as a special instance of the if-then-else, excluding subcharts as a construct poses no problem. The if-then-else construct is described below.

3.1.8 Coregions

A coregion is used to indicate that partial ordering is imposed on the events contained in the region as opposed to the total ordering present outside of a coregion. A coregion is graphically represented as a vertical dotted line next to the instance axis as in Figure 3.7.



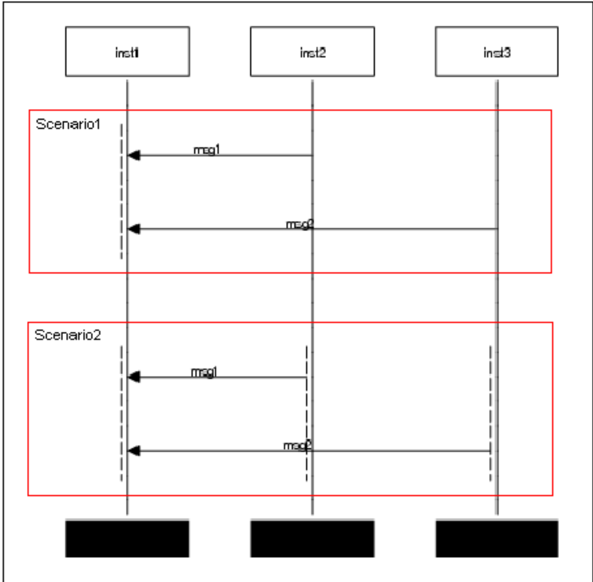
The coregion, which is given by a dashed vertical line, induces partial order on `inst1`, but because the messages are synchronous, the order is still preserved.

Figure 3.7: An LSC with a coregion.

Coregions in [DH01] also act as simultaneous regions, but [Klo03] uses an explicit construct for simultaneous regions, and thus limits coregions. Because the LSCEditor uses the representation of an explicit simultaneous region this representation is adopted.

Limitations

Coregions introduce partial order on a section of an instance line. As only synchronous messages are supported in the LSC subset, a coregion with partial order would be overridden if the other instance lines involved in the message passing in the coregion are not declared as coregions, like in `Scenario1` of Figure 3.8. If a message is sent, it will be received at the same time. Therefore, a coregion declaration on an instance line will count as a global coregion for all instance lines. This means that `Scenario1` of Figure 3.8 will be interpreted as `Scenario2`.



The interpretation of coregions is that coregions induces no order on messages for all instance lines. A specification like the one in `Scenario1` is thus interpreted as the specification of `Scenario2`.

Figure 3.8: How coregions are interpreted.

Note, that coregions may not overlap, because it would be uncertain how the semantics is with a message in two coregions. This possibility is disregarded.

Furthermore, if-then-else constructs inside coregions are not supported. It is not clear how an if-then-else would be interpreted inside a coregion and including this construct would add much to the complexity of the verification without contributing much to the expressive power of an LSC. Only hot messages are supported in a coregion to keep coregions as simple as possible, thus conditions and simultaneous regions are not supported either.

Application

When testing message traces within coregions it is required that multiple traces must be tested, because there is no ordering between the messages in a coregion, thus all possible orderings must be tested. The configurations resulting from a satisfied sequence are used in the remainder of the chart, and if there is more than one satisfied sequence, then all of these configurations must be tested.

A coregion is given a temperature and the temperature has the same semantics as snapshots. Thus, if a coregion is cold, progression is not enforced anywhere in the coregion.

A coregion in the LSCEditor has an explicit temperature. If a coregion is cold, then there is no guarantee that the coregion is ever left, i.e., progression is not enforced anywhere in the coregion.

3.2 Extensions to the basic LSC constructs

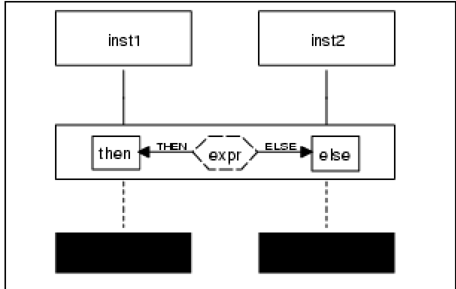
Next follows some extensions from [Klo03] to the basic LSCs. These extensions are LSC constructs, which solve some of the shortcomings in the core feature set presented in [DH01].

3.2.1 If-then-else

An if-then-else construct is added. The construct consists of a condition and two LSCs. If the condition evaluates to *true* the first subchart is activated, and if the condition evaluates to *false*, the second subchart is activated. In either case, the subchart not chosen is skipped. An example of an if-then-else construct can be seen in Figure 3.9.

Application

When a chart contains an if-then-else construct, the condition of the construct will determine the full path that must be searched for in the computation graph. The subchart of the construct will simply be appended to the parent trace. The only way to test the condition without limiting the condition construct is to use the verification engine of UPPAAL using TCTL as with a condition element.



The if-then-else construct states that if the condition evaluates to *true* the 'then' LSC must hold, otherwise it is the 'else' LSC that must hold.

Figure 3.9: LSC modelling an if-then-else construct.

When using if-then-else constructs it is recommend that cycles are avoided, unless well positioned conditions are included, or else an infinite sequence may be specified.

3.2.2 LSC activation mode

An LSC can be in one of three modes, *initial*, *invariant*, and *iterative*. The initial mode allows one test with the LSC, namely initially when the system starts. The invariant and iterative modes allow more test with the same LSC, the difference between them is that the invariant mode allows more simultaneous incarnations, i.e., a new incarnation can be activated while another is still in progress. This is not possible for the iterative chart, where a new incarnation may not be activated if an existing incarnation is already running.

Limitations

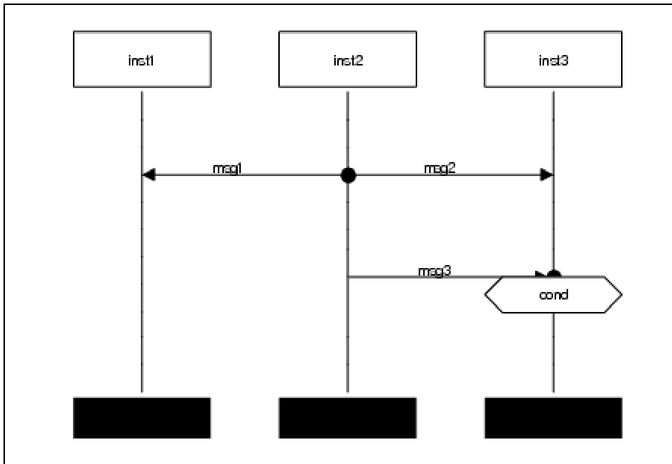
When doing model checking an ignored activation of a chart might be the one falsifying a property, thus iteration is disregarded [Klo03][pp. 77-79].

Initial mode is also disregarded because this can be modelled by adding the condition that every instance must be in their initial location to the activation condition. This means that the activation mode for LSCs is always invariant.

3.2.3 Simultaneous regions

A simultaneous region allows grouping of several elements that should be observed at the same time. This enables, e.g., association of conditions and messages to groups of events. It also allows message broadcast by sending several messages at the same point in time. If a number of cold asynchronous messages are sent in a simultaneous region, either all of the messages arrive simultaneously or none arrive at all.

Graphically, a simultaneous region is denoted with a filled circle surrounding the location on the instance axis, as can be seen in Figure 3.10.



Two simultaneous regions. One with two messages and one with a message and a condition.

Figure 3.10: LSC two simultaneous regions.

Limitations

As mentioned in Section 3.1.6 conditions are not allowed to be part of a simultaneous region. In fact, the only construct allowed in a simultaneous region is synchronous messages.

Application

Simultaneous regions are used for modelling broadcast communication. Broadcast communication in UPPAAL is non-blocking, thus sending of a message does not

ensure that anyone receives the message. This means that cold messages must be supported in simultaneous regions. Hot messages are also supported, which means that if a message is sent it must be received. This is not directly a UPPAAL feature, but a feature that can be modelled in an LSC and also verified, because the engine can take into account that a given message must be sent and received. As simultaneous regions model broadcasting all messages sent from the region must have the same label.

3.2.4 Time

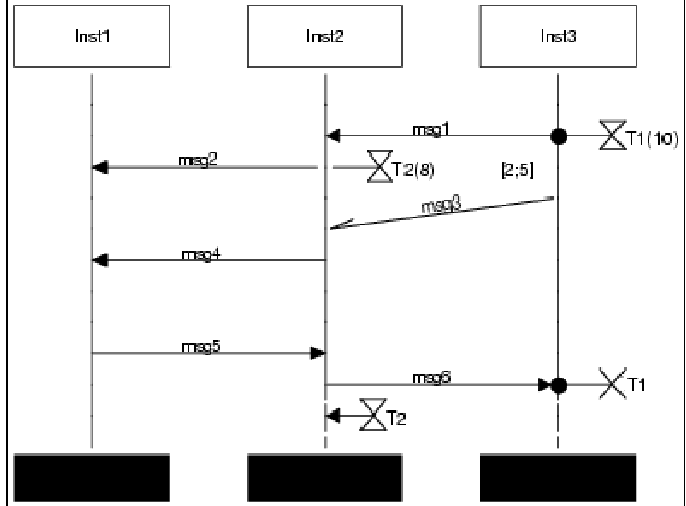
It is desirable to be able to specify time constraints in an LSC chart. Timers and timing intervals are introduced to provide this. The boundaries of the intervals can be given by two types of parenthesis, one for including the bound, '[' and ']', and one for excluding the bound, '(' and ')'.

A timer set is represented by an hour glass symbol labelled with a name and a duration. The hour glass is connected to the instance with a straight line. A timeout is also represented by an hour glass symbol, but contrary to a timer set it is connected to the instance with an arrow. A timer reset is represented with a large X, which is connected to the instance line with a straight line. There may be only one timeout or one timer reset associated with a timer set, and they may not span several instances. A timer set and its corresponding timeout or timer reset are connected with a vertical line or by labelling them with the same name.

A timing interval needs to be associated with two locations, whereas a timeout is an event itself, which can be used to delimit constructs such as conditions, messages, and even other timer sets. Figure 3.11 shows an example of an LSC with timing constraints. Apart from the timer reset (T1) and the timeout (T2) there is an example of a timing interval; the interval between `msg1` and `msg3` on `inst3`.

Limitations

Clocks in UPPAAL are modelled as a part of the set of data variables. When modelling an LSC there is access to the data variables of the UPPAAL model, and as the data components can be used in conditions, constraints on clocks can be modelled through conditions on clock variables. This means that the explicit clock modelling features, timer sets, timeouts, and timer resets, are not necessary, and we thus choose to limit the LSCs by not including them.



Inst3 must send msg3 at least 1 time unit, but less than 6 time units after msg1 has been received. This is given by the [2,5] time interval, between msg1 and msg3 on the inst3 instance line.

Figure 3.11: LSC illustrating Klose's timer constructs.

3.2.5 Local invariants

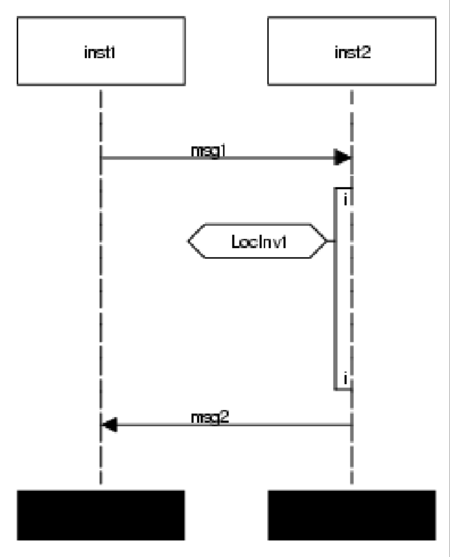
[Klo03] also introduces condition constructs that are to hold over a period of time called local invariants. Like conditions they have a temperature, which is interpreted as for conditions, i.e., using solid and dashed lines for hot and cold, respectively.

Local invariants are represented by a condition spanning a section of an instance line through connected start and end boundaries. The boundaries can either include or exclude the reference points denoted by i and e, respectively. A local invariant with including boundaries is depicted in Figure 3.12.

If local invariants start at an instance head, the reference point specifying the start of the local invariant must be exclusive, as it would otherwise act as an activation condition for the chart. This also means that adding the local invariant to the activation condition yields the same as inclusion.

Limitations

A local invariant in our LSCs would be the same as a set of conditions, one for each snapshot covered by the local invariant. Because snapshots set the range of a



Both boundaries of the local invariant are included in the invariant, given by the 'i'. Exclusion of a boundary is given by 'e'.

Figure 3.12: LSC with a local invariant.

condition, local invariants would have to be restricted to span a snapshot, thus have the same meaning as a condition. Local invariants are thus disregarded, because they can be modelled by conditions.

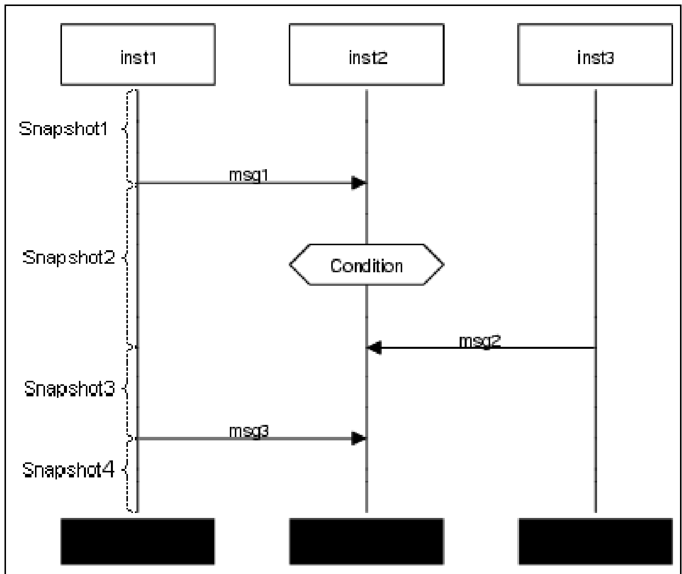
3.3 LSC behaviour

The understanding of the LSC subset adopted in this project except snapshots should be clear. Snapshots is now defined in full and a summary of the mandatory and provisional behaviour and notions of the constructs are summarised in Table 3.1 on page 45.

Snapshots are collections of instance lines separated only by messages, if-then-else, coregion, and simultaneous region constructs. The beginning and end of a chart are also snapshots. An empty chart thus only contain one snapshot, and likewise for charts containing only conditions.

In addition, the snapshot ending a prechart is the same snapshot that starts the main chart, recall Figure 3.2, and the snapshots starting and ending a subchart are the same as the ones ending at and starting from the subchart.

When considering a model, a snapshot consists of all possible FSM configurations at that point in the LSC. The snapshot for an LSC after a given synchronisation consists of the FSM configuration the synchronisation ends in as well as all configurations reachable by internal transition steps and synchronising steps that synchronise a process not represented in the chart with any other process, see Figure 3.13, and these steps are referred to as *intermediate transition steps*. The FSM computation graph is thus to be traversed in order to find the span of the snapshot for each trace in the model. If a condition is to be satisfied in a snapshot, the sequence of configurations in the snapshot must all satisfy the condition.



The messages separate snapshots, and as conditions are part of snapshots, all configurations in `snapshot2` must satisfy the condition `condition`.

Figure 3.13: An LSC divided into snapshots.

Element	**Semantics**	**Mandatory**	**Provisional**
Chart	Mode	*Universal:* All runs of the system must satisfy the chart.	*Existential:* At least one run of the system satisfies the chart.
Segment	Temperature	*Hot:* Progression is enforced down the instance.	*Cold:* Progression is not enforced.
Message	Temperature	*Hot:* If message is sent it will be received.	*Cold:* If message is sent it may be received.
Condition	Temperature	*Hot:* If condition does not hold chart is not satisfied.	*Cold:* If condition does not hold chart is satisfied.
Coregion	Temperature	*Hot:* A coregion is always left.	*Cold:* There is no guarantee that the coregion is left.

The notation and liveness properties for the selected subset of LSC constructs.

Table 3.1: Liveness properties of the selected LSC constructs.

3.4 Formal description

The previous sections presented an informal description of the LSC language and a comparison with UPPAAL models in order to establish the selected feature set used to verify UPPAAL models. The feature set is described formally based on [Klo03] and [DH01]. This section specifies when an LSC holds for a UPPAAL model, by first describing the abstract syntax of an LSC and second setting up semantical rules for interpreting a chart and its elements.

An LSC specification for a system is a tuple:

$$\Upsilon = \langle pch, mode, chart \rangle$$

- pch is an optional prechart that specifies a triggering behaviour leading to the point of activation of the LSC. It is an LSC in itself that needs to be traversed successfully before *chart* is traversed. A special case is an activation condition, which can be modelled as a hot condition in the prechart.
- $mode \in \{existential, universal\}$ is the mode of the chart.
- The *chart* is a finite sequence of elements connecting instance lines. An instance *inst* is a member of the finite set $\mathcal{I}$, and each instance line corresponds to an instance of an automaton Φ. Elements are: Messages m of the finite set $\mathcal{M}$, simultaneous regions *sim* of the finite set $\mathcal{SIM}$, if-then-else constructs *if* of the finite set $\mathcal{IF}$, and coregions *co* of the finite set *CO*. Simultaneous regions, if-then-else constructs, and coregions are sequences of elements.

  The elements are separated by snapshots s of the finite set $\mathcal{S}$. Snapshots may contain a number of conditions, at most one for each instance line. Only the UPPAAL configurations corresponding to the snapshot need to satisfy the conditions in the snapshot. Evaluation of condition c_x on $inst_x$ is handled by the function c_x, which gives a truth value for the condition, given a configuration of the network of automata. If a snapshot does not contain a condition, then it is said to contain the hot condition *true*. In the syntax the conditions and snapshots are given separately. In addition, if-then-else constructs contain conditions, and as these constructs are not carried by any specific instance line, a function c to evaluate the condition c over all instance lines is introduced. It returns the truth value of such a condition given a configuration and it evaluates to *true* just when the condition holds for all of the instance lines.

A chart has the form specified in the following BNF:

chart	::=	*snapshot hot-condition cold-condition element chart*$'$	
	$\vert$	*snapshot hot-condition cold-condition*	
snapshot	::=	*hot-snapshot* $\vert$ *cold-snapshot*	
hot-snapshot	::=	———	
cold-snapshot	::=	- - - - -	
hot-condition	::=	c_i *hot-condition*$'$ $\vert$ c_i	where condition $c_i \in \mathcal{C}$ and $inst_i \in \mathcal{I}$ is the corresponding instance line
cold-condition	::=	c_i *cold-condition*$'$ $\vert$ c_i	where condition $c_i \in \mathcal{C}$ and $inst_i \in \mathcal{I}$ is the corresponding instance line
element	::=	*hot-msg* $\vert$ *sim* $\vert$ *if-then-else*	
	$\vert$	*hot-coregion* $\vert$ *cold-coregion*	
sim	::=	*sim-element sim*$'$	
	$\vert$	*sim-element*	
sim-element	::=	*hot-msg* $\vert$ *cold-msg*	
hot-msg	::=	$inst_A \xrightarrow{m} inst_B$	where $inst_A, inst_B \in \mathcal{I}$ $\wedge\ inst_A \neq inst_B \wedge m \in \mathcal{M}$
cold-msg	::=	$inst_A \overset{m}{\dashrightarrow} inst_B$	where $inst_A, inst_B \in \mathcal{I}$ $\wedge\ inst_A \neq inst_B \wedge m \in \mathcal{M}$
if-then-else	::=	*condition chart*$'$ *chart*$''$	
condition	::=	c	where $c \in \mathcal{C}$
hot-coregion	::=	*hot-msg hot-coregion*$'$	
	$\vert$	*hot-msg*	
cold-coregion	::=	*hot-msg cold-coregion*$'$	
	$\vert$	*hot-msg*	

3.4.1 Semantics

In the following, It is defined inductively, when an LSC chart is satisfied or modelled by a finite subtrace $\overline{\sigma}[k, k'']$, denoted, $\overline{\sigma}[k, k''] \models$ *chart*.

If a prechart pch has been specified the subtrace $\overline{\sigma}[k, k'']$ must first satisfy pch before satisfying the LSC body *chart* as illustrated in Figure 3.14. The LSC holds for a UPPAAL model depending on the $mode$ of the chart:

- If $mode =$ *existential* there exists a trace $\overline{\sigma}[1, K]$ such that:

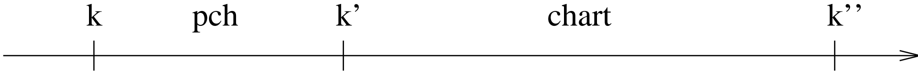
  $$\exists\ k, k', k'' : \overline{\sigma}[k, k'] \models pch \wedge \overline{\sigma}[k', k''] \models chart$$

  This essentially means, that at least a single trace of the system needs to satisfy the prechart as well as the chart.

- If $mode =$ *universal* then for all traces $\overline{\sigma}[1, K]$, it is the case that:

  $$\forall\ k, k', k'' : \overline{\sigma}[k, k'] \models pch \Rightarrow \overline{\sigma}[k', k''] \models chart$$

  This means that all traces of the system satisfying the prechart must also satisfy the chart.



A subtrace $\overline{\sigma}[k, k'']$ must first satisfy pch and then immediately satisfy *chart*.

Figure 3.14: The order of the charts to be satisfied by a trace.

Chart

The following rule specifies when a subtrace models a chart. It extracts a snapshot, the conditions in the snapshot, and an element from the chart and ensures that they as well as the rest of the chart hold for the subtrace in question.

First, if the snapshot and its conditions hold, second, the element is satisfied by the last configuration satisfying the snapshot, and third, the rest of the chart holds for the last configuration satisfying the element, then the chart is modelled by the subtrace. The subtrace also models the chart, if the snapshot and its hot conditions hold, but its cold conditions do not:

$$\overline{\sigma}[k,k''] \models \textit{snapshot hot-condition cold-condition element chart}'$$

$$\begin{aligned}
\text{iff } \exists\, i,k' : \quad & k \leq i < k' \leq k'' \\
& \wedge\, \overline{\sigma}[k,i] \models \textit{snapshot} \\
& \wedge\, \overline{\sigma}[k,i] \models \textit{hot-condition} \\
& \wedge\, \overline{\sigma}[k,i] \models \textit{cold-condition} \\
& \wedge\, \overline{\sigma}[i,k') \models \textit{element} \\
& \wedge\, \overline{\sigma}[k',k''] \models \textit{chart}' \\
\vee \quad & \\
& k \leq i < k' \leq k'' \\
& \wedge\, \overline{\sigma}[k,i] \models \textit{snapshot} \\
& \wedge\, \overline{\sigma}[k,i] \models \textit{hot-condition} \\
& \wedge\, \overline{\sigma}[k,i] \not\models \textit{cold-condition}
\end{aligned}$$

Snapshots

In UPPAAL models configuration changes are caused by transitions, internal transition steps or synchronisation steps, and as synchronisation steps would need to be specified in the chart in the form of messages in order for the successive configurations to be traversed, these steps are excluded from snapshots. It is thus a requirement that the subtrace to satisfy a snapshot only consists of internal transition steps or synchronisation steps where no instance line represent the sender or that none of the receivers are represented. Remember, these transition steps are referred to as *intermediate transitions*.

Snapshots are handled by the following rules. The first two rules handle the semantics of the temperature of the snapshots as well as making sure the subtrace is a sequence of intermediate transitions.

$\overline{\sigma}[k,k'] \models$ ———

$$\begin{array}{l}\text{iff} \quad \forall\, i : k \leq i \leq k' \Rightarrow \mathit{MD}(\overline{\sigma}[i]) < \infty \\ \text{where the transition } \overline{\sigma}[j] \xrightarrow{S} \overline{\sigma}[j+1] \\ \text{is an intermediate transition for } k \leq j < k'\end{array}$$

A subtrace satisfies a hot snapshot when the maximum delay of each configuration is finite. The maximum delay function *MD* defined in Section 2.2.2 is used for restricting the delay. Also, the subtrace may only consist of intermediate transitions:

As cold snapshots do not disallow the subtrace to take unbounded time, the progress restriction does not apply. Again, the subtrace may only consist of intermediate transitions:

$\overline{\sigma}[k,k'] \models$ - - - - -

$$\begin{array}{l}\text{where the transition } \overline{\sigma}[i] \xrightarrow{S} \overline{\sigma}[i+1] \\ \text{is an intermediate transition for } k \leq i < k'\end{array}$$

Condition

In order for a subtrace to satisfy a set of conditions in a snapshot the conditions must hold for all configurations in the subtrace, i.e., the conditions are invariant over the subtrace.

The hot conditions of a snapshot are satisfied when

$\overline{\sigma}[k,k'] \models c_i$ *hot-condition'*

$$\begin{array}{ll}\text{iff} & \overline{\sigma}[k,k'] \models \textit{hot-condition}' \\ & \wedge\, \forall\, j : k \leq j \leq k' \Rightarrow \overline{\sigma}[j] \models c_i\end{array}$$

and the following rule evaluates the cold conditions of the snapshot:

$\overline{\sigma}[k,k'] \models c_i$ *cold-condition'*

$$\begin{array}{ll}\text{iff} & \overline{\sigma}[k,k'] \models \textit{cold-condition}' \\ & \wedge\, \forall\, j : k \leq j \leq k' \Rightarrow \overline{\sigma}[j] \models c_i\end{array}$$

A configuration satisfies a condition if the condition holds for the instance line carrying the condition:

$$\overline{\sigma}[k] \models c_i$$

$$\text{iff} \quad c_i(\overline{\sigma}[k]) = \textit{true}$$

Likewise, if a condition is carried by an if-then-else it is satisfied by a configuration if the condition evaluates to *true*:

$$\overline{\sigma}[k] \models c$$

$$\text{iff} \quad c(\overline{\sigma}[k]) = \textit{true}$$

Messages

If the element is a hot message then the subtrace satisfies the hot message if the message corresponds to either a binary or a broadcast synchronisation over a channel between UPPAAL automata. At least two automata involved in the synchronisation must change active locations because the hot message is always received, and the automata must correspond to the ones specified in the LSC chart:

$$\overline{\sigma}[k, k'] \models \textit{inst}_A \xrightarrow{m} \textit{inst}_B$$

$$\begin{aligned} \text{iff} \quad & \overline{\sigma}[k] \xrightarrow{m} \overline{\sigma}[k'] \\ \text{where} \quad & \overline{\sigma}[k] = (\langle \ldots, l_i, \ldots, l_j, \ldots \rangle, v) \\ & \wedge \overline{\sigma}[k'] = (\langle \ldots, l'_i, \ldots, l'_j, \ldots \rangle, v') \\ & \wedge \Phi_i = \textit{inst}_A \wedge \Phi_j = \textit{inst}_B \end{aligned}$$

If the element is a cold message then the subtrace satisfies the cold message if the message corresponds to a broadcast synchronisation over a channel between UPPAAL automata. Furthermore, the automata involved in the synchronisation change active locations if the synchronisation is received. If it is not received by any automata, only the sender changes active location:

$$\overline{\sigma}[k,k'] \models \textit{inst}_A \overset{m}{\dashrightarrow} \textit{inst}_B$$

$$\begin{aligned}
&\text{iff } \overline{\sigma}[k] \xrightarrow{m} \overline{\sigma}[k'] \\
&\text{where} \quad \overline{\sigma}[k] = (\langle \ldots, l_i, \ldots, l_j, \ldots \rangle, v) \\
&\qquad \wedge \\
&\qquad\quad \overline{\sigma}[k'] = (\langle \ldots, l'_i, \ldots, l'_j, \ldots \rangle, v') \\
&\qquad\quad \vee\, \overline{\sigma}[k'] = (\langle \ldots, l'_i, \ldots, l_j, \ldots \rangle, v') \\
&\qquad \wedge \\
&\qquad\quad \Phi_i = \textit{inst}_A \wedge \Phi_j = \textit{inst}_B
\end{aligned}$$

Simultaneous regions

In order for a subtrace to satisfy a simultaneous region, it must satisfy all the elements in the simultaneous region:

$$\overline{\sigma}[k,k'] \models \textit{sim-element sim}'$$

$$\begin{aligned}
&\text{iff} \quad \overline{\sigma}[k,k'] \models \textit{sim-element} \\
&\qquad \wedge\, \overline{\sigma}[k,k'] \models \textit{sim}'
\end{aligned}$$

If-then-else

If the element is an if-then-else, then if the condition holds, the 'then' chart must be satisfied by the configuration, or else the 'else' chart must be satisfied. The subtrace satisfies the if-then-else if the subchart decided by the condition holds for the subtrace:

$$\overline{\sigma}[k,k'] \models \textit{condition chart}' \textit{ chart}''$$

$$\begin{aligned}
&\text{iff} \quad \overline{\sigma}[k] \models \textit{condition} \\
&\qquad \wedge\, \overline{\sigma}[k,k'] \models \textit{chart}' \\
&\ \vee \\
&\qquad \overline{\sigma}[k] \not\models \textit{condition} \\
&\qquad \wedge\, \overline{\sigma}[k,k'] \models \textit{chart}''
\end{aligned}$$

Coregion

In order for a subtrace to satisfy a coregion, it must satisfy all the elements in the region in any order. This is done by extracting arbitrary elements from the

coregion, which the subtrace must satisfy until the last element is extracted. In order to do this in a precise, yet simple, manner the elements in a coregion are collected in the set *SM* from which each element that is satisfied can be extracted. Below, $\overline{\sigma}[k', k''] \models SM \setminus \{e\}$ denotes that $\overline{\sigma}[k', k'']$ satisfies the remaining set of messages. The empty set is always satisfied.

The rule for hot coregions can now be defined. A hot coregion must adhere to the liveness requirements of hot snapshots, thus

$\overline{\sigma}[k, k''] \models$ *hot-coregion*

$$\begin{array}{ll} \text{iff} & \exists\, i, k' \;\exists\, e \in SM : \\ & \wedge\, k \leq i < k' \leq k'' \\ & \wedge\, MD(\overline{\sigma}[i]) < \infty \\ & \wedge\, \overline{\sigma}[i, k'] \models e \\ & \wedge\, \overline{\sigma}[k', k''] \models SM \setminus \{e\} \\ \multicolumn{2}{l}{\text{and the transition } \overline{\sigma}[j] \xrightarrow{S} \overline{\sigma}[j+1]} \\ \multicolumn{2}{l}{\text{is an intermediate transition for } k \leq j < k' - 1} \end{array}$$

If instead the coregion is cold, progress is not enforced. Using the set *SM* again:

$\overline{\sigma}[k, k''] \models$ *cold-coregion*

$$\begin{array}{ll} \text{iff} & \exists\, i, k' \;\exists\, e \in SM : \\ & \wedge\, k \leq i < k' \leq k'' \\ & \wedge\, \overline{\sigma}[i, k'] \models e \\ & \wedge\, \overline{\sigma}[k', k''] \models SM \setminus \{e\} \\ \multicolumn{2}{l}{\text{and the transition } \overline{\sigma}[j] \xrightarrow{S} \overline{\sigma}[j+1]} \\ \multicolumn{2}{l}{\text{is an intermediate transition for } k \leq j < k' - 1} \end{array}$$

3.5 Summary

LSCs are an extension of standard MSCs. The main extensions are a formal semantical basis, treatment of conditions as first-class citizens, and liveness properties. Liveness is added on both chart level and element level. On the chart level it is the difference between universal vs. existential charts, and on the element level it is the difference between hot and cold elements.

An informal description of LSCs are presented, including the limitations in the application of LSCs as a requirements specification language for UPPAAL. [DH99]

and [DH01] are used as a basis for the LSC specification and extensions presented by [Klo03] are considered as well. A subset of the LSC specification is identified and used for the diagrammatic requirements specifications for UPPAAL models. The subset consists of:

- *Charts:* Both existential and universal charts are supported, as well as precharts as the handling of these are not very different than a main chart. Activation conditions are disregarded, as they can be modelled with a condition in a prechart.
- *Messages:* Asynchronous messages are not supported as UPPAAL only supports synchronous communication. Binary synchronisation in UPPAAL must have both a sender and a receiver to exist, thus only hot messages are supported. In simultaneous regions both hot and cold messages are supported, as a broadcast message in UPPAAL may not have any receivers.
- *Conditions:* Both hot and cold conditions are supported including shared conditions, which are translated to single conditions on each of the instance lines to share the condition.
- *Coregions:* Coregions are supported, but in a slightly modified form. Coregions are made global across the structural dimension, meaning that no order is imposed on the ordering of messages within a coregion. Only messages are supported within them, No snapshots are within them, only messages. In addition, coregions have a temperature, specifying whether progression is enforced within the region.
- *Simultaneous regions:* Simultaneous regions are supported and their application is to model broadcast synchronisation in UPPAAL models. Both hot and cold messages are supported in a simultaneous regions, hot means that the message must be received and cold that it may or may not.
- *If-then-else:* One of the extensions of Klose is an if-then-else construct. An if-then-else consists of a condition and two subcharts. The evaluation of the condition decides which of the charts is to be traversed.

A formal specification of the semantics for the LSC subset used in the PEEL verification engine is presented. The specification is induced from the syntax, which makes it suitable for the implementation of PEEL

Next chapter gives an introduction to the tools used for specifying LSC diagrams, extracting and traversing UPPAAL models, and an introduction to PEEL and its implementation is presented.

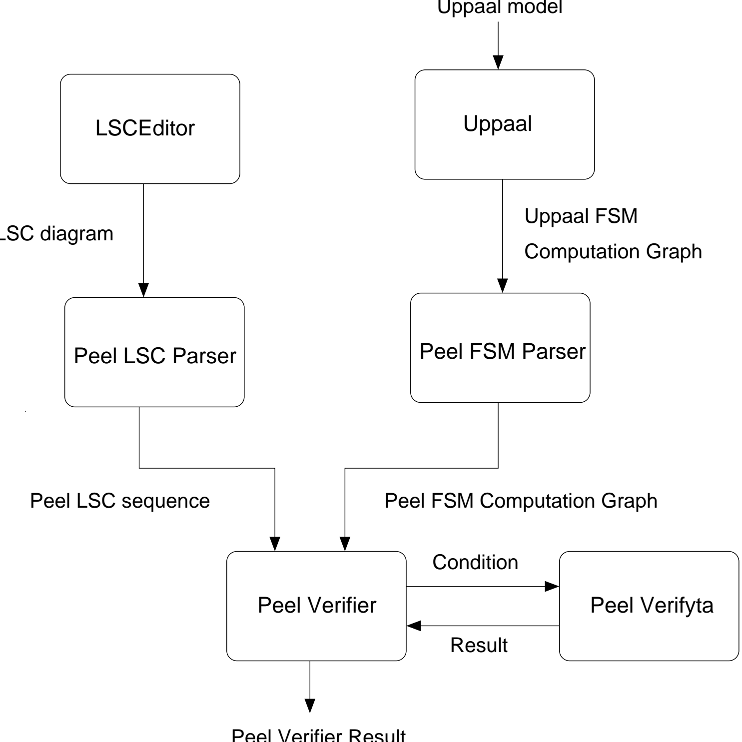
4 PEEL - Property Extraction Engine for LSCs

Contents

4.1 LSCEditor 56
4.1.1 LSC output format 58
4.1.2 Property extraction 59
4.2 UPPAAL computation graph 61
4.2.1 Graph representation 61
4.3 Verification algorithm 63
4.4 The running time of the algorithms 71
4.5 Optimisation 74
4.5.1 Conditions 74
4.5.2 Testing reuse 74
4.5.3 Hard drive I/O 75
4.5.4 Further improvements 75
4.6 Test 76
4.6.1 Regression testing 77
4.7 Summary 79

This chapter presents the tools used for specifying LSC diagrams, the LSCEditor, extraction of the computation graph from UPPAAL models, the FSM format, and the PEEL verification engine, consisting of the PEEL LSC Parser, PEEL FSM Parser, and PEEL Verifier.

Figure 4.1 presents the data-flow graph for the PEEL verification engine and the two additional tools used. The individual elements of this process are presented in this chapter.



An LSC diagram is made using the LSC editor and is saved to a file. This file is then parsed by the PEEL LSC Parser which extracts LSC elements and structures them into a sequence, complete with subchart hierarchy. The UPPAAL verification engine takes the UPPAAL model to be verified, and outputs an FSM computation graph. Note, in order to get this output, an internal version of UPPAAL supporting this feature must be used. The FSM computation graph is parsed by the PEEL FSM Parser into a PEEL FSM computation graph. The PEEL LSC sequence and the PEEL FSM computation graph are the inputs to the PEEL Verifier, which checks the sequence against the computation graph. The PEEL Verifier outputs whether the model satisfies the LSC specification or not.

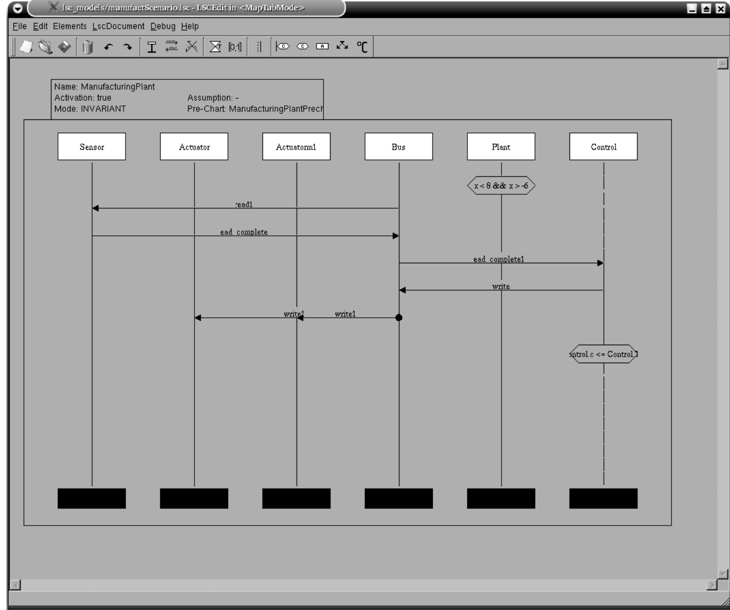
Figure 4.1: Data-flow diagram for PEEL.

4.1 LSCEditor

The editor used for the LSC diagrams, the LSCEditor, see Screenshot 4.2, is most kindly supplied by the Carl von Ossietzky Universität Oldenburg, Germany.

The LSCEditor is consistent with the features described in Klose's dissertation [Klo03]. Not all features are described here, only those included in the PEEL LSC subset.

As for the LSCEditor itself, it is quite intuitive to use. When creating a new dia-



Screenshot of the LSCEditor displaying a scenario from the Distributed Control experiment, see Chapter 5 for the experiment.

Figure 4.2: Screenshot displaying the LSCEditor.

gram, you must specify various settings before the chart is created, e.g., if the chart is universal or existential, and if the chart should have a prechart or not. When the chart is created, a number of instance lines may be added to the chart. When adding an instance it must be specified whether the beginning of the instance line should be hot or cold. An instance contains locations along the line (more can be added if needed). All the events in the chart must be associated to one or more locations.

When drawing a message between two instances, you specify the temperature of the message and the temperature of the instance line of the sending and receiving instance, after sending and reception, respectively.

When placing a condition on an instance line, the temperature of the condition as well as the temperature of the instance line below the condition must be specified. A boolean expression can be entered in the condition. There is a certain syntax when specifying conditions due to the dependency of the UPPAAL engine, and thereby TCTL:

- When referring to a specific location in an automaton, the automaton and location are separated by a dot, e.g., "train.stopped". Likewise, if a variable associated to an automaton should be tested, they are also separated by a dot, e.g., "train.x". The automaton name in the condition must of course correspond to a UPPAAL automaton.

See also the TCTL syntax in Section 2.3 for a more thorough description of the TCTL syntax used in conditions.

A coregion is placed on the vertical axis along the instance line attached to two locations. The coregion specifies that all messages along the instances in the horizontal direction have no order.

A simultaneous region is created by marking a location on an instance line as a simultaneous region, and then placing one or more messages in this location.

Another element which can be placed in the chart, is an if-then-else. This construct consists of a condition which, based on the boolean expression, decides which of the two subcharts should be used. This if-then-else construct is also placed in a location, and spans across all instances so that they will evaluate the subchart condition simultaneously.

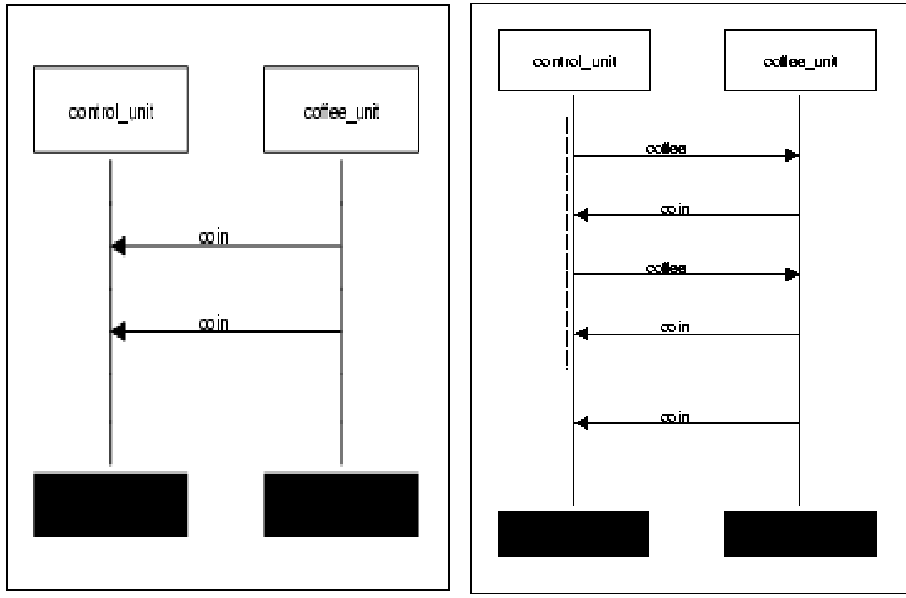
Examples of the above constructs can be seen in Section 5.2.

4.1.1 LSC output format

The LSCEditor has two LSC representation modes, standard mode, and map table mode. The standard mode does not allow labels to clash, e.g., two messages cannot carry the same message label. Also, the LSC is saved in one file, the *LSC-file*. The map table mode does not limit the usage of labels. The map table mode generates, in addition to the LSC file, a map file, the *map-file*, with all element ids, which means that labels are still unique, but can carry the same text.

PEEL uses the map table mode, because this enables the specification of more than one message with the same label. The labels refer to channel names, which means that a message between two instances with label `coin` is a synchronisation via the `coin` channel in the UPPAAL model.

Next follows examples of an LSC diagram, its map-file, and its LSC-file, see Figures 4.3, 4.4, and 4.5. The LSC is for the coffee vending machine example introduced in Chapter 2.



The left is the prechart, and the right is the main chart.

Figure 4.3: LSC for the coffee vending machine example.

```
RHAPSODY
lscid_9 = , ER, lscid_3, lscid_2, VERBATIM coffee;
lscid_15 = , ER, lscid_2, lscid_3, VERBATIM coin;
lscid_13 = , ER, lscid_3, lscid_2, VERBATIM coffee;
lscid_10 = , ER, lscid_2, lscid_3, VERBATIM coin;
lscid_14 = , ER, lscid_2, lscid_3, VERBATIM coin;
lscid_7 = , ER, lscid_5, lscid_6, VERBATIM coin;
lscid_8 = , ER, lscid_5, lscid_6, VERBATIM coin;
lscid_2 = , INST, , , VERBATIM control_unit;
lscid_3 = , INST, , , VERBATIM coffee_unit;
lscid_5 = , INST, , , VERBATIM control_unit;
lscid_6 = , INST, , , VERBATIM coffee_unit;
lscid_1 = , COND, , , VERBATIM ;
lscid_4 = , COND, , , VERBATIM ;
```

Figure 4.4: Map file for the vending machine example.

4.1.2 Property extraction

The *LSC-file* and the *map-file* is parsed into an intermediate format, documented in Appendix B, Figure B.1, from which the sequence of LSC elements is generated, see Figure B.2. The parsing into the intermediate format only includes those elements that are in the PEEL LSC subset, i.e., the supported LSC features.

```
lscdocument created_By_LSCEdit;
universal  lsc coffee3_test
activation condition : condexpr lscid_1 endexpr
activation mode invariant prechart coffee3pre;
instance hot lscid_2  'anzloc=26';
        hot concurrent;
                out hot lscid_9,129 to lscid_3 '2';
                in hot lscid_15,0 from lscid_3 '4';
                out hot lscid_13,0 to lscid_3 '6';
                in hot lscid_10,1 from lscid_3 '8';
        endconcurrent '1;2;3;4;5;6;7;8;9;';
        hot in hot lscid_14,1 from lscid_3 '11';
hot endinstance; end
instance hot lscid_3  'anzloc=26';
        hot in hot lscid_9,129 from lscid_2 '2';
        hot out hot lscid_15,0 to lscid_2 '4';
        hot in hot lscid_13,0 from lscid_2 '6';
        hot out hot lscid_10,1 to lscid_2 '8';
        hot out hot lscid_14,1 to lscid_2 '11';
hot endinstance; end
endlsc;
universal prechart lsc coffee3pre
activation condition : condexpr lscid_4 endexpr
activation mode initial;
instance hot lscid_5  'anzloc=22';
        hot in hot lscid_7,17 from lscid_6 '2';
        hot in hot lscid_8,1366909696 from lscid_6 '4';
hot endinstance; end
instance hot lscid_6  'anzloc=22';
        hot out hot lscid_7,17 to lscid_5 '2';
        hot out hot lscid_8,1366909696 to lscid_5 '4';
hot endinstance; end
endlsc;
```

Figure 4.5: LSC file for the vending machine example.

4.2 UPPAAL computation graph

The computation graph of UPPAAL is described as a symbolic state transition system. The people behind UPPAAL have provided a means of retrieving a representation of the state machine in the form of a file dump, the format used by FSM Visualizer [fsm].

In order to have PEEL use this format, a parser is required. First, the input format of FSM Visualizer is presented, followed by a textual and graphical representation of the graph, which is used in the implementation of the parser. Finally, the algorithm for traversing the graph in order to verify message sequences is given.

4.2.1 Graph representation

The input format used by the FSM Visualizer describes a graph, and it consists of three parts, state variable declarations, configurations, and transitions between the configurations. The parts are separated in the file by a single line containing the string "---", see Figure 4.6.

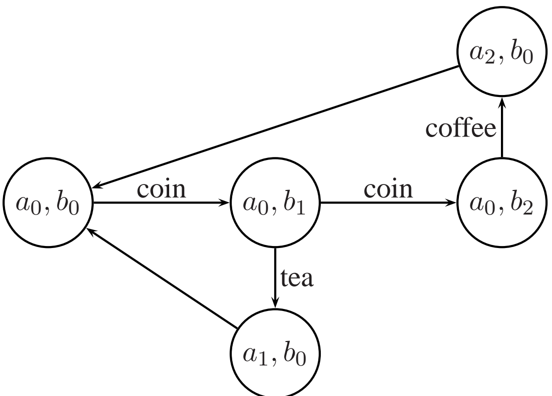
A variable declaration includes the name, the cardinality of the value domain, the type, and a list of the possible values of the variable. A configuration consists of a list of variable value indexes, one for each variable. Each transition includes two configuration IDs, one for the configuration being exited and one for the configuration being entered. Also, the synchronisation labels are included, giving the label of the sending automaton first followed by any receivers' labels. If the transition is internal, it is labelled with 'tau' and the automaton in question is given.

The above graph description in Figure 4.6 is parsed by the FSM parser and used by the PEEL verification engine to check if the LSC requirements are fulfilled. The graphical representation is displayed as a graph in Figure 4.7.

```
control_unit(3) State ''b0'' ''b0''  ''b0''
coffee_unit(3) State ''a0'' ''a2'' ''a1''
traceId(0)
---
1 1 0
0 0 1
0 1 18
1 0 6
2 0 10
0 2 14
---
1 2 ''fake''
2 2 ''control_unit.tau''
6 2 ''control_unit.tau''
3 2 ''control_unit.tau''
2 4 ''coffee_unit.coin! control_unit.coin?''
4 5 ''coffee_unit.coin! control_unit.coin?''
4 6 ''control_unit.tea! coffee_unit.coin?''
6 6 ''control_unit.tau''
5 3 ''control_unit.coffee! coffee_unit.coffee?''
3 3 ''control_unit.tau''
```

The textual FSM graph description for the coffee vending machine, see Figure 2.2 in Chapter 2.4 for an illustration.

Figure 4.6: FSM graph for the vending machine example



The graph shows the UPPAAL location sets as computation configurations. There are no variables in the UPPAAL model used in this example. If there were variables, the possible variable values would be a part of the configurations, e.g., a variable with a range of three in every UPPAAL model location could triple the number of configurations.

Figure 4.7: FSM computation graph for the vending machine example.

4.3 Verification algorithm

This section presents the algorithm that has been constructed for verifying an LSC chart. The algorithm is presented in a pseudocode format. The overall algorithm consists of seven functions; one that starts off the verification, a manager function that controls the flow of the algorithm, and five specialised algorithms that handle one LSC element type each. The specification for the algorithm is the semantics defined for LSCs in Section 3.4, and each 'holds' algorithm corresponds directly to a semantic rule found in that section.

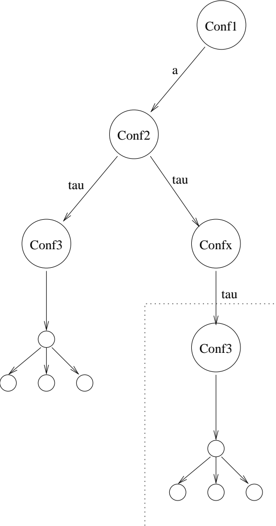
Algorithm 1, `startVerify`, iterates over all configurations in the FSM graph calling `traceRunner` for each configuration. In other words, it starts the verification with each configuration as the start-configuration. In case of a universal chart, all `traceRunner` calls must return *true* in order for the chart to be satisfied, this is handled in `startVerify`. If a universal chart is not satisfied, the error is not returned to this function, but is handled in the function where it is discovered. In the case of an existential chart that is not satisfied, this is also handled in this function through the `allTrue` variable. But if an existential chart is satisfied, i.e., a trace in the FSM graph is found that matches the LSC sequence, `traceRunner` will not return to this function.

```
 1: allTrue := true {only used for existential LSCs}
 2: for each configurations c in FSM Graph do
 3:    if ! traceRunner(c, 0) then
 4:       allTrue := false
 5:    end if
 6: end for
 7: if allTrue == true then
 8:    exit(success)
 9: else
10:    exit(error)
11: end if
```

Algorithm 1: void startVerify()

Algorithm 2, `traceRunner`, is designed as a depth-first graph traversal algorithm. `traceRunner` controls the verification algorithm through a switch-case structure that identifies the current element that needs to be satisfied, and runs the appropriate holds-function for that element. It also ensures that no trace needs to be traversed more than one time in the same tau-subtree thereby eliminating the possibility of running in an endless loop. An example of this can be seen in Figure 4.8. This is achieved through a colouring scheme, where a global counter is used to 'colour' configurations with the 'colour' of the current trace. An LSC sequence is always ended by an *EndOfSequence* element. If `traceRunner` reaches the EndOfSe-

quence element for the main chart and the chart is existential, the algorithm has found the trace that it was searching for and it exits with a success message. If the chart is universal, it simply returns *true*, as it is necessary that all traces reach the EndOfSequence element. This is controlled in Algorithm 1. If an EndOfSequence element is reached and the current chart is a subchart, the algorithm returns *true*. This will then propagate up to the if-then-else element where the subchart was started, and the trace will continue after that element. The algorithm is designed to use `traceRunner` to dive deeper into the recursion tree. This means that for each call of `traceRunner` the depth of the graph traversal is increased, except for a EndOfSequence element.



When traversing this computation tree, there is no need to go into the sub-tree in the dotted box, as `Conf3` has already been visited and its subtree traversed.

Figure 4.8: FSM computation graph traversion.

```
 1: if configuration has been visited in this trace then
 2:    return true
 3: else
 4:    mark this configuration as visited
 5: end if
 6: switch (type of sequence element)
 7:    case EndOfSequence:
 8:       if chart is a subchart then
 9:          return true
10:       else if chart is a prechart then
11:          return traceRunner(c, 0)
12:       else
13:          if chart is existential then
14:             exit(success)
15:          else
16:             return true
17:          endif
18:       endif
19:    case Snapshot:
20:       return holds_snapshot(configuration, sequenceIndex)
21:    case Message:
22:       allMessagesHold := true
23:       for all outedges in configuration that are not 'tau' edges do
24:          if not holds_message(outedge, sequenceIndex) then
25:             allMessagesHold := false
26:             if chart is universal then
27:                exit(error)
28:             end if
29:          end if
30:       end for
31:       return allMessagesHold
32:    case Coregion:
33:       if ! holds_coregion(configuration, sequenceIndex, messageList)
34:          if chart is universal
35:             exit(error)
36:          end if
37:       return true
38:    case IfThenElse:
39:       return holds_ifthenelse(configuration, sequenceIndex)
40:    case Simregion:
41:       allMessagesHold := true
42:       for all outedges in configuration that are not 'tau' edges do
43:          if ! holds_simregion(outedge, sequenceIndex) then
44:             allMessagesHold := false
45:             if chart is universal then
46:                exit(error)
47:             end if
48:          end if
49:       end for
50:       return allMessagesHold
51: end switch
```

Algorithm 2: bool traceRunner(configuration, sequenceIndex)

Algorithm 3, `holds_snapshot`, ensures that a snapshot is satisfied. It tests all the conditions in the snapshot, and if they are satisfied, the snapshot is satisfied. It takes into account the temperature of the condition, and whether the chart is a prechart or a main chart. If all conditions evaluate to *true*, the trace just continues, but if one condition fails, there are different outcomes. If the chart is a prechart and the condition is hot, the trace returns *true*, if the condition is cold, the rest of the prechart is disregarded and the algorithm jumps to the main chart. If the chart is the main chart and the condition is hot, the algorithm exits with an error, and if the condition is cold, the rest of the chart is disregarded. This means that if the trace is in a subchart and a cold condition evaluates to *false*, the subchart is exited, and the parent chart continues.

Algorithm 4, `holds_message`, on page 68 checks that an edge in the FSM graph is satisfied by the LSC chart. The edge is tested for its label, its source instance's label, and its target instance's label. If the edge is satisfied, the algorithm continues by calling `traceRunner` on the target configuration of the edge and incrementing the sequenceIndex, i.e., the next element in the LSC chart is to be tested. If the edge is not satisfied, its target and source instance lines are checked to see if they are defined in the chart at all. If one of them are not, the edge is treated as a 'tau' edge, and the trace is continued from the edge's target configuration.

Algorithm 5 `holds_coregion` on page 69 tests a list of messages to see if a trace through the FSM graph is possible that corresponds to a permutation of the list. This is done by matching the messages in the list one by one to edges in the FSM graph, and removing them from the list as they are found. If an edge is found which target or source instance line is not defined in the chart, the edge is treated as a 'tau' edge in the same way as in Algorithm 4. If a trace is found, the algorithm calls `traceRunner` and it continues with the sequence.

Algorithm 6, `holds_simregion` on page 70, checks that an edge in the FSM graph is satisfied by the simultaneous region defined in the sequence. It checks that each of the edge's receivers correspond to the ones defined in the LSC chart. If a receiver instance line in the edge is not defined in the LSC chart, that receiver is disregarded. If all receivers are either defined or disregarded the simultaneous region is satisfied and the algorithm continues by calling `traceRunner` on the target configuration of the edge.

Algorithm 7, `holds_ifthenelse` on page 70, handles an if-then-else construct. An if-then-else construct cannot fail in it self, but the rest of the sequence, be it the then-part or the else-part, can fail and that will be sent back by this algorithm. The algorithm saves the old sequence, and starts `traceRunner` with the

```
1:  allTrue := true
2:  if all conditions in snapshot evaluate to true then
3:     for all outedges in configuration do
4:        if outedge is a 'tau' transition then
5:           if ! traceRunner(outedge.targetConfiguration, sequenceIndex) then
6:              allTrue := false
7:           end if
8:        else
9:           if traceRunner(configuration, sequenceIndex+1) returns false then
10:             allTrue := false
11:          end if
12:       end if
13:    end for
14:    return allTrue
15: else
16:    if temperature of condition is Hot then
17:       if chart is mainchart then
18:          if chart is universal then
19:             exit(error)
20:          else
21:             return false
22:          end if
23:       else
24:          return true
25:       end if
26:    else
27:       if chart is prechart then
28:          if trace is in a subchart then
29:             return true
30:          else
31:             return traceRunner(configuration, 0) {traceRunner is started on the mainchart se-
                quence}
32:          end if
33:       else
34:          return true
35:       end if
36:    end if
37: end if
```

Algorithm 3: bool holds_snapshot(configuration, sequenceIndex)

new sequence, as decided by the condition. When the new sequence is returned `holds_ifthenelse` activates the former sequence, and continues the trace by calling `traceRunner` on the next sequence element.

```
 1: if edge is a broadcast message then
 2:    if only one receiver of edge is defined in chart then
 3:       continue with the edge
 4:    else
 5:       if chart is prechart then
 6:          return true
 7:       else
 8:          return false
 9:       end if
10:    end if
11: end if
12: if edge match the message in sequence[sequenceIndex] then
13:    return traceRunner(edge.targetConfiguration, sequenceIndex+1)
14: else
15:    if both target instance and source instance of edge is defined in chart then
16:       return chart is prechart
17:    else
18:       return traceRunner(edge.targetConfiguration, sequenceIndex)
19:    end if
20: end if
```

Algorithm 4: bool holds_message(edge, sequenceIndex)

```
1:  oneTrue := false
2:  if no messages in coregion then
3:     return traceRunner(configuration, sequenceIndex+1)
4:  end if
5:  if configuration has been visited in coregion-trace then
6:     return false
7:  else
8:     mark this configuration as visited
9:  end if
10: for all outedges in configuration do
11:    if outedge is a 'tau' edge then
12:       if holds_coregion(outedge.targetConfiguration, sequenceIndex, messagesInCoregion)
          then
13:          oneTrue := true
14:       end if
15:    else
16:       if edge is found in messagesInCoregion then
17:          if holds_coregion(outedge.targetConfiguration, seqIndex, messagesInCoregion - out-
             edge) then
18:             oneTrue := true
19:          end if
20:       else
21:          if target instance or source instance of edge is not defined in chart then
22:             if holds_coregion(outedge.targetConfiguration, sequenceIndex, messagesInCore-
                gion) then
23:                oneTrue := true
24:             end if
25:          else
26:             if chart is prechart then
27:                return true
28:             else
29:                if chart is universal then
30:                   exit(error)
31:                end if
32:             end if
33:          end if
34:       end if
35:    end if
36: end for
37: return oneTrue
```

Algorithm 5: bool holds_coregion(configuration, sequenceIndex, messagesInCoregion)

```
1:  if source instance of edge is not defined in chart then
2:    return traceRunner(edge.targetConfiguration, sequenceIndex)
3:  end if
4:  for each receiver in edge.receivers do
5:    if receiver is not defined in LSC chart then
6:      remove message from edge.receivers
7:    end if
8:  end for
9:  for each message in simregion do
10:   if message is found in edge.receivers then
11:     remove message from edge.receivers
12:   else
13:     if temperature of message is Hot then
14:       if chart is prechart then
15:         return true
16:       else
17:         return false
18:       end if
19:     end if
20:   end if
21: end for
22: if no messages left in edge.receivers then
23:   return traceRunner(edge.targetConfiguration, sequenceIndex+1)
24: else
25:   return false
26: end if
```

Algorithm 6: bool holds_simregion(edge, sequenceIndex)

```
1:  if condition evaluates to true then
2:    set active sequence to then-sequence and save old sequence
3:  else
4:    set active sequence to else-sequence and save old sequence
5:  end if
6:  if traceRunner(configuration, 0) then
7:    activate old sequence
8:    return traceRunner(configuration, sequenceIndex+1)
9:  else
10:   activate old sequence
11:   return false
12: end if
```

Algorithm 7: bool holds_ifthenelse(configuration, sequenceIndex)

4.4 The running time of the algorithms

In order to find the asymptotic upper bound of the worst case running time for the verification algorithm as a whole, each holds algorithm must be considered. First, the upper bounds of the five holds algorithms are approximated, Second, the worst case running time of the entire verification algoritm, i.e., `traceRunner` and `startverify`, is approximated using the results from the five holds algorithms.

The running times of the algorithms depend on the LSC specification and the UPPAAL model to be verified, especially the size of the model is a decisive factor.

The following factors have an influence on the running time:

a	The number of automata in the model.
c	The number of configurations in the FSM computation graph.
m	The number of messages in any coregion in the chart.
e	The number of elements in the sequence.

Snapshot

`holds_snapshot`, Algorithm 3, tests all conditions in all configurations reachable by intermediate transitions in the FSM graph. In the worst case all transitions are intermediate transitions, and thus `holds_snapshot` must be run c times, i.e., on all configurations in the FSM graph. For each configuration all conditions must be evaluated, so the worst case running time of `holds_snapshot` is:

$$\begin{aligned} T_3(c, a) &= c * a \\ &= O(ca) \end{aligned}$$

As `holds_snapshot` depends on both a and c, it runs in quadratic time in the worst case.

Message

`holds_message`, Algorithm 4, checks each receiver instance of the edge to see if more than one receiver for the edge is defined in the chart, as this would mean the edge is a broadcast. In the worst case all automata instances of the model must be checked, except the source automaton. Ignoring that `holds_message` runs in

constant time, the worst case running time is:

$$\begin{aligned} T_4(a) &= (a-1) \\ &= O(a) \end{aligned}$$

So `holds_message` runs in linear time.

Coregion

`holds_coregion`, Algorithm 5, takes as input a list of messages. In the worst case it must check each possible permutation of this list to see if that permutation is satisfied in every trace of the FSM graph. For a list of m messages, we get that:

$$\text{number of possible permutations} = m!$$

Each permutation of the list corresponds to a sequence of messages of length m. In the worst case, each message in the list, is found in the end of each possible trace. So for a graph of c configurations:

$$\text{maximal number of traces for each message} = (c-1)!$$

If the cost of checking one message in a coregion is a constant b, the worst case running time for holds coregion is:

$$\begin{aligned} T_5(c, m) &= m! * m * (c-1)! * b \\ &= O(m! * (c-1)!) \end{aligned}$$

`holds_coregion` runs in factorial time.

Simultaneous region

`holds_simregion` is very similar to `holds_message` described above. It also checks each receiver of the edge to see if it is defined in the chart. The maximum number of messages in a simultaneous region is the number of automata in the model, a, minus one for the source automaton. So the worst case running time for `holds_simregion` is:

$$\begin{aligned} T_6(a) &= (a-1) \\ &= O(a) \end{aligned}$$

So `holds_simregion` runs in linear time like `holds_message`.

If-then-else

An if-then-else construct consists of a single condition check, thus `holds_ifthenelse` runs in constant time.

$$\begin{aligned} T_7() &= k \\ &= O(1) \end{aligned}$$

Verification algorithm

To find the worst case running time for the entire algorithm, we must consider the case where all elements in the sequence are the most expensive element type. In the results above the `holds_coregion`, with its factorial time running time, is by far the most expensive.

Let the sequence consist of only snapshot and coregion elements. The number of coregions is e. The number of snapshot elements is then $e + 1$ and for each snapshot, `holds_snapshot` must be run on all configurations c, in all traces $(c - 1)!$, for each start configuration c.

The cost of all the snapshots are:

$$\begin{aligned} T_{\text{snap}}(c, a, e) &= c * (c - 1)! * c * (e + 1) * T_3 \\ &= O(c^2 * c! * e * a) \end{aligned}$$

The cost of all coregion element is:

$$\begin{aligned} T_{\text{co}}(c, e, m) &= c * (c - 1)! * e * T_5 \\ &= O(c! * e * m! * (c - 1)!) \end{aligned}$$

So the worst case running time of the algorithm is:

$$\begin{aligned} T_w(c, a, e, m) &= T_{\text{snap}} + T_{\text{co}} \\ &= (c^2 * c! * e * a) + (c! * e * m! * (c - 1)!) \\ &= O(c! * m! * (c - 1)!) \end{aligned}$$

The factorial running time is inherent in graph traversal algorithms, where this rather expensive running time is unavoidable when doing verification.

4.5 Optimisation

During the experiments carried out, see Chapter 5, it was discovered that especially the way conditions are tested presented a performance bottle-neck regarding execution time. This section contains a description of the condition testing, the problems, realised solutions, and possible further measures.

4.5.1 Conditions

The condition checking bottle-neck became apparent when starting the experiments with the Distributed control UPPAAL model, see Chapter 5 Section 5.3 for a presentation of the model. The UPPAAL model FSM graph contains 14.876 nodes, and 29.601 edges, which is a lot more than the other experiment models. The scenario tested is presented in the LSC diagram illustrated in Figure 5.17.

The LSC contains two conditions, the first one on the *Plant* instance line, which must be evaluated in all snapshots, because it is the only element on the instance line. The second condition is on the *Control* instance and it must be evaluated in the last two snapshots. The scenario is a universal chart meaning that all traces must be tested, and all traces must be valid for the chart to be valid.

The initial approach for testing conditions was to test them when they appeared, but when conducting the first tests on this scenario it was clear that this approach was extremely time consuming. Especially two factors had impact on the execution time, the number of conditions to be tested, and the way the conditions were tested using verifyta.

4.5.2 Testing reuse

The initial approach for evaluating conditions were to evaluate them when needed. This meant that a condition in a specific configuration might be evaluated more than once. This approach on the Distributed control experiment resulted in a total of 102.742.867 condition evaluations. By having each condition remember that it has been evaluated in a given configuration, unnecessary repetitive evaluations are avoided. This extension of PEEL resulted in a reduction of the amount of conditions to be evaluated. In the Distributed control experiment the amount of conditions to be evaluated dropped to 21.442.

4.5.3 Hard drive I/O

The second factor was the execution time of the verifyta call, where at least three hard drive I/O operations were involved in every condition evaluation, one for writing the query file, one for executing verifyta and saving the output to disc, and finally one for reading and checking the verifyta result. The impact of the hard drive I/O operations means that the effective CPU usage of verifyta is typically only about $3\% - 5\%$ [1].

It is possible to circumvent the disk I/O operations, as the UPPAAL tool also contains a socket server, which enable the usage of an ethernet instead of disk I/O. Both the hard drive I/O and the network access method has an overhead, but it is possible to disregard it by testing all possibilities for a condition in one verifyta execution. This can be done by collecting TCTL formulae for all possibilities into one query file and executing verifyta with this query file, which would insure 100% CPU usage. This approach was tested with the Distributed control experiment and the results is that one condition, $x < 8 \;\&\&\; x > -6$, tested in 14.875 configurations, verified with verifyta takes approximately 76 minutes [2].

4.5.4 Further improvements

Further options for improving the evaluation of conditions exist. One that is easily implemented is using a global testing in TCTL, by using A[] it is possible to test whether a condition is satisfied in all possible configurations, and E<> for testing whether a condition is satisfied somewhere in the model. If a condition is always satisfied it is not necessary to do any further testing, because it is always *true*, conversely, if E<> is not satisfied, then any specific testing evaluates to *false*. These expressions can be used succesfully in the Distributed control experiment, because the two conditions are always satisfied, which means a reduction to two conditions to be checked, but generally it is not possible to say anything about their applica-

[1] Different hardware was not tested regarding the hard drive I/O performance, so the actual impact of hard drive characteristics is not reasoned about.

[2] The test was executed on a system with dual Pentium 4 Hyperthreading 2.8 GHz CPUs. The *time* command on *NIX systems was used to measure the run time, which is the actual CPU usage time.

Two other systems were also tested and the results are: 48 minutes on a system with an Athlon XP 2000+ 1.6 GHz running Linux and 57 minutes on a system with a Pentium 4 2.4 GHz running Linux.

The amount of RAM consumed was about $60 - 80$ MB, thus the amount of RAM available was not an important aspect. The access time and bandwidth of RAM might be an issue effecting the run time, but these aspects are unknown.

bility, because it is specific to a UPPAAL model and the condition expression.

Another improvement is to check whether the conditional expression contains any clocks, because if it does not, it is possible to check the expression directly in the FSM configuration, as it contains the exact values af all variables. This is not implemented in PEEL, because type checking variables in a TCTL local property is beyond the scope of this project.

4.6 Test

When developing a verification tool it is of course very important that the tool itself is correct, otherwise the results cannot be trusted. There are different goals of the tests performed on a piece of software. Some may check if a communication protocol is implemented correctly and others may examine how well a piece of software scales. Different goals mean different methods and thus different types of tests [Dou99, EDE01]. Different tests include:

- *Component testing:* During component testing, which is sometimes referred to as unit testing, each new component is validated as an isolated system. Before being integrated with the rest of the system it is important that a new component behaves according to its specifications.
- *Integration testing:* When a new component has been made, and it has passed its component test, an integration test is performed to check if the new component behaves correctly in the context of the complete system.
- *System validation:* As perhaps the most fundamental testing phase, system validation is also the hardest to cover. Validation testing is performed on the complete system to determine if it conforms to the system requirements
- *Regression testing:* Regression testing originates from the principles of extreme programming [Ken99], where it is used as a part of validation testing. All the previous test scenarios that were created during component and integration testing can be run and should still be valid even if the system has changed.
- *Stress testing:* Stress testing a system can be very important, e.g., for systems that are required to run for long periods of time.

PEEL is only a prototype implementation to show that it is possible to use LSCs as requirements for verification of UPPAAL models, which means that it is not fully

```
-----PRE CHART----
|=| < true > Hot on control_unit |=| < true > Hot on coffee_unit |=|
coffee_unit --[coin]--> control_unit
|=| < true > Hot on control_unit |=| < true > Hot on coffee_unit |=|
coffee_unit --[coin]--> control_unit
|=| < true > Hot on control_unit |=| < true > Hot on coffee_unit |=|

-----MAIN CHART----
|=| < true > Hot on control_unit |=| < true > Hot on coffee_unit |=|
  |  control_unit --[coffee]--> coffee_unit
  |  coffee_unit --[coin]--> control_unit
  |  control_unit --[coffee]--> coffee_unit
  |  coffee_unit --[coin]--> control_unit

|=| < true > Hot on control_unit |=| < true > Hot on coffee_unit |=|
coffee_unit --[coin]--> control_unit
|=| < true > Hot on control_unit |=| < true > Hot on coffee_unit |=|

-----END SEQUENCE----
```

ASCII output of the sequence structure parsed by the PEEL LSC parser

Figure 4.9: PEEL sequence output

tested for actual verification usage. Some testing has been conducted to ensure that the results from the experiments performed are correct. The testing performed are mainly component testing and system validation, and the tests have been collected in a regression testing possibility.

4.6.1 Regression testing

There are two levels for the regression testing used with PEEL, the first is an ASCII output of the LSC and FSM data structures created by the PEEL LSC Parser, PEEL LSC Sequence builder, and PEEL FSM Parser respectively. See class diagrams in Figure B.1, B.2, and B.3 in Appendix B, and see ASCII output in Figure 4.9 and 4.10.

The second test level is a test suite of actual PEEL model verifications.

When performing an execution of PEEL it is possible to enable different debug output, see Figure 4.11. The options `-l`, `-s`, and `-f` enables an ASCII output of the parsed LSC structure, sequence of LSC elements, and the FSM computation graph. This is not an automated test in the sense that a *true/false* results is presented, but it is possible to inspect the structures and manually validate that they represent what is expected.

```
[coffee3_test : Universal]
-PreChart = coffee3pre
         [control_unit : lscid_2]
         |
         *-2: 0x807a870: [coffee]
         |
         *-4: 0x807a870: [coin]
         |
         *-6: 0x807a870: [coffee]
         |
         *-8: 0x807a870: [coin]
         |
         *-11: 0x807a870: [coin]

         [coffee_unit : lscid_3]
         |
         *-2: 0x807a8b8: [coffee]
         |
         *-4: 0x807a8b8: [coin]
         |
         *-6: 0x807a8b8: [coffee]
         |
         *-8: 0x807a8b8: [coin]
         |
         *-11: 0x807a8b8: [coin]

[coffee3pre : Universal]
-PreChart = None
         [control_unit : lscid_5]
         |
         *-2: 0x807a8e8: [coin]
         |
         *-4: 0x807a8e8: [coin]

         [coffee_unit : lscid_6]
         |
         *-2: 0x807a930: [coin]
         |
         *-4: 0x807a930: [coin]
```

ASCII output of the LSC structure parsed by the PEEL LSC parser

Figure 4.10: PEEL LSC structure output

```
Peel verification engine version 1.0b, June 2004
Usage: peel uppaal_model lsc_chart lsc_map [options]
Options:
         -c   Condition check mode: Default 1
                 1: Conditions are checked one-by-one when needed
                 2: Conditions are checked in batches
                    i.e., all configurations are checked at once
         -f   Output FSM parser structure.
         -h   Display this message.
         -l   Output LSC parser structure.
         -s   Output LSC element sequence.
         -v   Display Peel verification steps.
         -q   Runs Peel in quiet mode - only result is shown.
         -e   Turns off displaying of verifier errors.
         -u   Path to Uppaal verifyta.
         -S   Enable verifyta socket server - not yet supported
                 Default: ../../uppaal-3.4.5/bin-Linux/verifyta
         -U   Path to Uppaal verifyta with FSM output.
                 Default: ../verifyta/verifyta
```

Figure 4.11: PEEL help output

It is possible to perform a regression testing on a test suite of UPPAAL models and LSCs. The actual regression test is created by collecting a range of individual tests in a makefile, which provide an entry to start all the tests in one batch.

The individual tests included, are the experiments described in Chapter 5, and a collection of smaller LSCs for various UPPAAL models. The collection covers all the different LSC elements and also a set of LSCs that must not be satisfied are included.

4.7 Summary

A prototype implementation of an LSC verification engine for UPPAAL models was presented in this chapter. The LSCEditor used for making LSCs, and the UPPAAL verifyta FSM output was also described. The FSM computation graph traversing algorithm was presented in pseudo code and the running time of the algorithm was calculated to be $O(c! + c! * m! * (c-1)!)$. Finally, some optimisation options were presented, some are included in the current PEEL implementation others proposed as possible future optimisations.

The next chapter contains experiments which show that LSCs can be used to capture requirements specifications for UPPAAL models, and the experiments also demonstrate the usage of the various elements included in the selected LSC subset.

5 Experiments

Contents

5.1 Broadcast 81
5.2 Train-gate 84
5.3 Distributed control 89
5.4 Summary 94

The purpose of this chapter is to show some practical examples of UPPAAL requirements specification by the use of LSC charts. Also, this chapter demonstrates that the various elements in the selected LSC subset work in PEEL. The demonstration is done through two small example models and one large model. The UPPAAL models are presented with different properties which are verified through a number of LSC specifications using the LSC subset of PEEL.

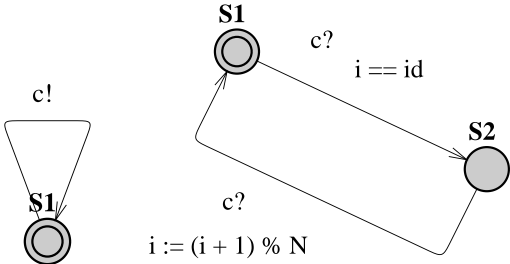
The example LSCs are applied to three UPPAAL models, Broadcast, which can be downloaded from the homepage [upp04], Train-gate, which is included with the UPPAAL application, and the Distributed control example, which has been constructed for testing the PEEL application.

5.1 Broadcast

The Broadcast experiment shows how to perform non-blocking one-to-many synchronisation. The sender broadcasts in every step and is never blocked.

The automata

The sender and the three identical receivers and their template are shown in Figure 5.1. When the sender broadcasts, all receivers capable of synchronising, i.e., if the guard allows them to take the transition, will synchronise. The receivers are instantiated with id's of 0, 1, and 1, respectively, and N is the number of receivers, i.e., three.



The sender is the left automaton and the receiver is the right. N is the number of receivers, three, and the id's of the receivers are 0, 1, and 1.

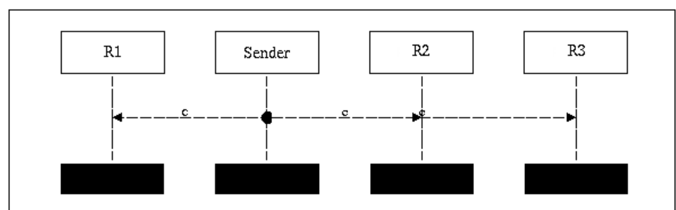
Figure 5.1: The sender and receiver templates.

The LSCs

The Broadcast experiment, which is illustrating the use of simultaneous regions, does not contain any clocks. This means that cold instance lines should be used, as models without clocks have no way of disallowing infinite delays to happen without using urgent locations and channels and these are not used in the model. But as the temperature of an instance line is ignored by PEEL (as specified in Chapter 4) the temperature is not relevant for these experiments.

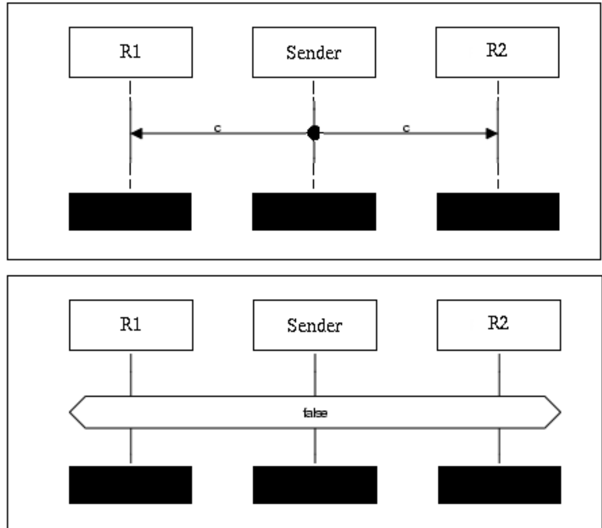
In Figure 5.2 the receivers may receive the cold broadcast message if the guard from the automaton in Figure 5.1 allows them to. In other words, the receivers will synchronise if $i == id$, where the id's of the receivers are 0, 1, and 1, respectively.

The LSC in Figure 5.3 specifies a forbidden scenario where R1 and R2 are not allowed to receive the message `c` at the same time, because their `id's` are different. If the prechart is satisfied, the main chart will terminate unsuccessfully as the hot condition evaluates to *false*.



This chart shows a simultaneous region where the sender broadcasts cold messages to the receivers. The receivers may receive the broadcast if their guard allows them.

Figure 5.2: LSC for the broadcast experiment.



This is an example of a forbidden scenario where the prechart specifies that the broadcast from the sender may never be received by both R1 and R2. If so, the main chart will always abort from a hot condition

Figure 5.3: LSC prechart and body for the broadcast experiment.

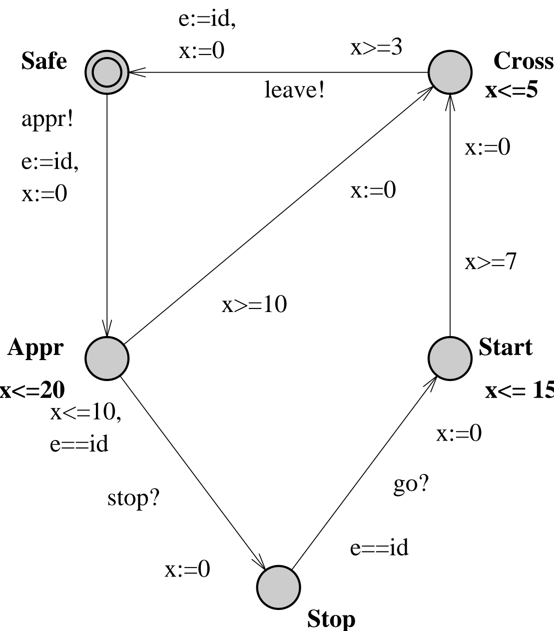
PEEL has been used to verify that the LSC charts for the Broadcast experiment specify the correct behaviour with respect to the corresponding UPPAAL automata. Statistical data such as the time taken to verify the specifications, the number of edges, configurations, and traces are shown in Table 5.1 on page 94. As can be seen in the table, this experiment is quickly verified due to the low number of configurations and edges. This experiment shows the use of messages and simultaneous regions.

5.2 Train-gate

The Train-gate experiment involves four trains, a gate and its queue. The gate is a critical region, which must only be crossed by one train at a time.

The automata

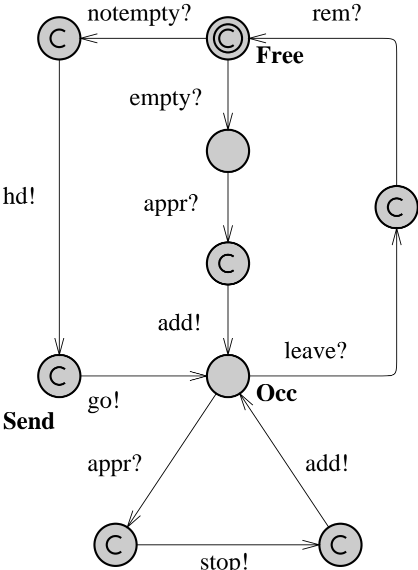
Figure 5.4 shows the automaton template modelling the behaviour of the trains, and Figure 5.5 contains the automaton template modelling the gate which all four trains must cross. The gate uses another automaton for lining up approaching trains in a queue, a queue automaton. The template of the queue automaton can be seen in Figure 5.6 on page 86.



A train will be approaching for maximum 20 time units and if it has not received a stop signal within 10 time units it will cross the gate, which will take between three and five time units. Also, a train may not wait in the queue for more than 15 time units.

Figure 5.4: The train template.

The general behaviour of the model is as follows: When a train approaches the gate, it is allowed to cross the gate if the queue is empty. Any other train approaching the gate, while a train is using the gate, will be commanded to stop and wait for



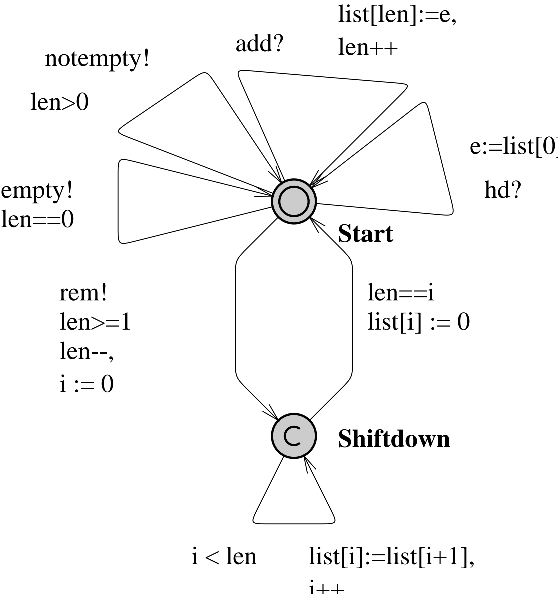
As can be seen, the use of committed locations only allows the gate to wait for trains approaching or leaving the gate, and the communication with the queue ensures that the queue is emptied when trains leave the gate.

Figure 5.5: The gate template.

the gate to be clear, i.e., when the first train has left the gate. Additional trains approaching the gate will be placed in the queue, which is emptied in FIFO order once the gate is clear.

The LSCs

In order to illustrate how LSCs can be used to specify complete scenarios the communication scenario for one of the trains and the gate is given in Figure 5.7 on page 87. It is a universal chart using an if-then-else construct to stop and restart the train. The subchart in Figure 5.8 on page 87 states that if the train ever notifies its approach, it will be stopped if the gate is unavailable, i.e., the length of the queue is larger than `0`. The 'else' subchart stops the train and eventually restarts it again allowing it to pass through and leave it. As for the 'then' subchart it is the case that every time the train approaches the gate, and it is available, the train should be able to pass through without being stopped. If another train tries to cross the gate with `Train1`, the message trace of the model will deviate from the sequence specified in the chart, and as the chart is universal, the chart will not hold for the model.



The order of the trains to use the gate is handled by using an array as a FIFO construct. Also, the shared variable e is used to communicate the id of the train to exit the queue and cross the gate.

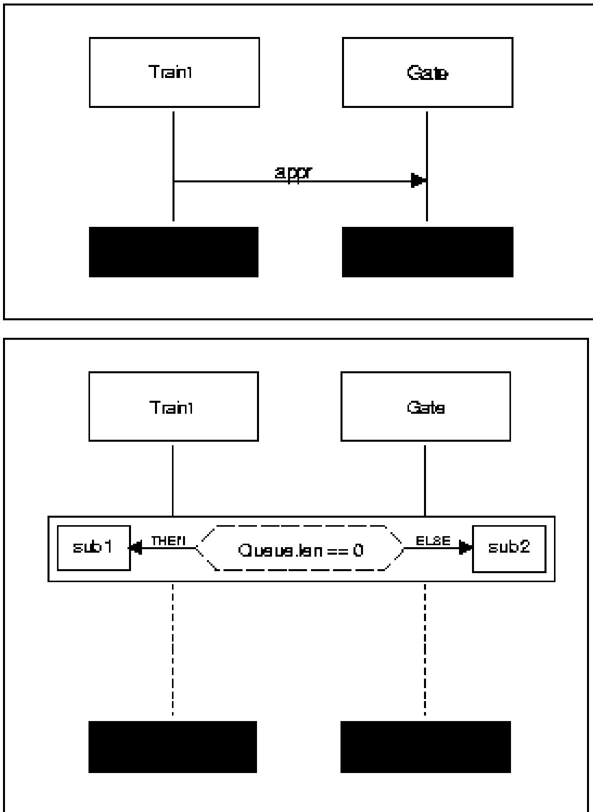
Figure 5.6: The queue template.

Notice that in the 'else' subchart in Figure 5.8 the queue is emptying as the other trains are crossing the gate, while Train1 is stopped, but it is not necessary to specify this in the subchart, as messages between the gate and other processes in the system are ignored.

Logically, the first train approaching the gate will not be stopped as no other train is being led through. This is expressed in the existential LSC in Figure 5.9 on page 88 for `Train1`. It illustrates how the existential mode of LSCs can be used for specifying a scenario that has to be possible.

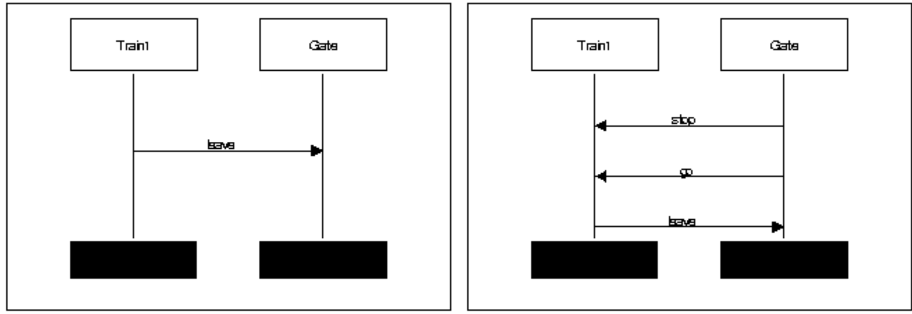
Another experiment is illustrated in Figure 5.10. It simply states that when Train1 and Train2 are approaching, the queue is not empty. Here, it is not necessary to include more trains, as any messages exchanged between the gate and other trains do not affect the fact that at least two trains are approaching. Nor is the order of the approaching trains important, which is illustrated by the use of a coregion.

Again, PEEL has been used to verify this model, see Table 5.1 on page 94 for details. This experiment has shown the use of messages, conditions, coregions, if-then-else constructs, and precharts which are all handled correctly by PEEL.



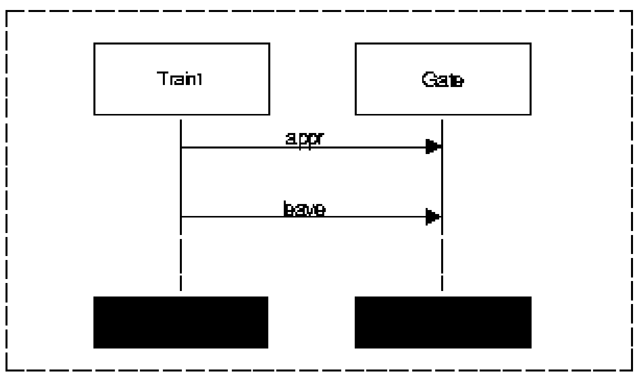
When Train1 approaches the gate, it will eventually pass through the gate. If there is a queue before the gate, the train has to wait until it gets permission to pass through, see the subcharts in Figure 5.8.

Figure 5.7: LSC prechart and body for Train-gate.



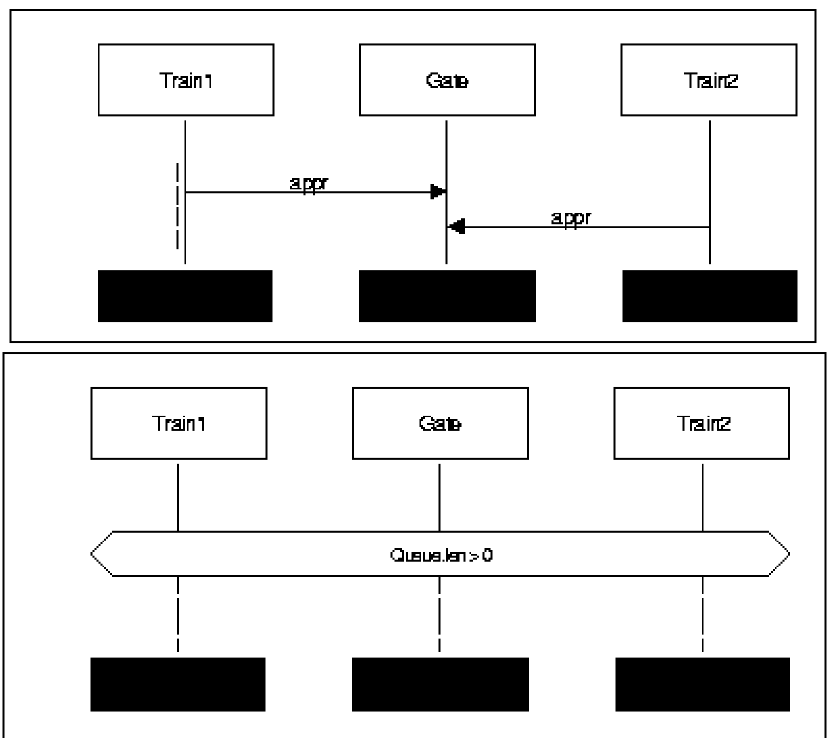
If the queue is empty the 'then' subchart (sub1) on the left specifies that Train1 may pass through the gate without any delay. If the queue is not empty the 'else' subchart (sub2) on the right specifies that the train is to stop at the gate and wait for other trains to pass through, before it will be allowed to pass through itself.

Figure 5.8: The subcharts for the Train-gate LSC



This existential LSC specifies that there exists a scenario such that when Train1 approaches the gate it will leave it without being stopped by a queue.

Figure 5.9: An existential LSC for the Train-gate experiment.



The chart specifies that when Train1 and Train2 are closing in on the gate, there is a queue. As the order of the two trains is insignificant their approach notifications are contained within a coregion.

Figure 5.10: LSC prechart and body for the queue in Train-gate.

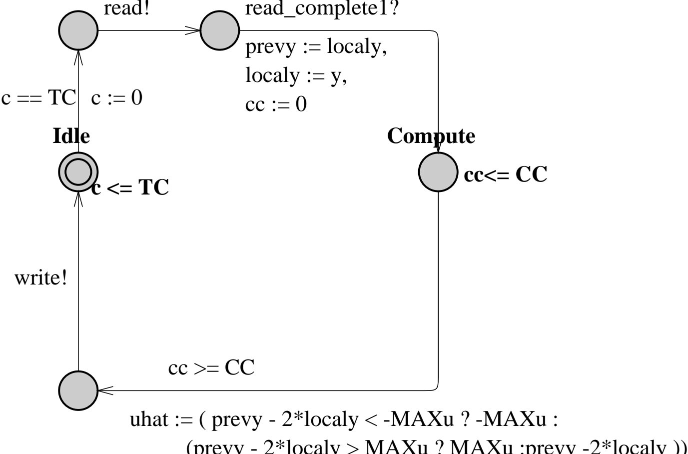
5.3 Distributed control

The distributed control is an experiment where a plant needs to be within a certain boundary. The plant controller ensures that this is so by having a sensor measure the state of the plant and two actuators to correct the plant. All communication between controller, sensor, and actuators goes through a bus.

The automata

Several global variables are declared, the constants `LAT`, `MAX`, and `MAXu` with values of 2, 8, and 3, respectively. `MAX` and -`MAX` are the upper and lower boundaries of the plant state `x` and the plant output variable `y`, and `MAXu` and -`MAXu` are the boundaries of the input variable `u` and the result of the controller's computations `uhat`.

In addition, the urgent channels `read`, `read1`, `read_complete`, `read_complete1`, and `write` as well as the broadcast channel `write1` are used in the model.

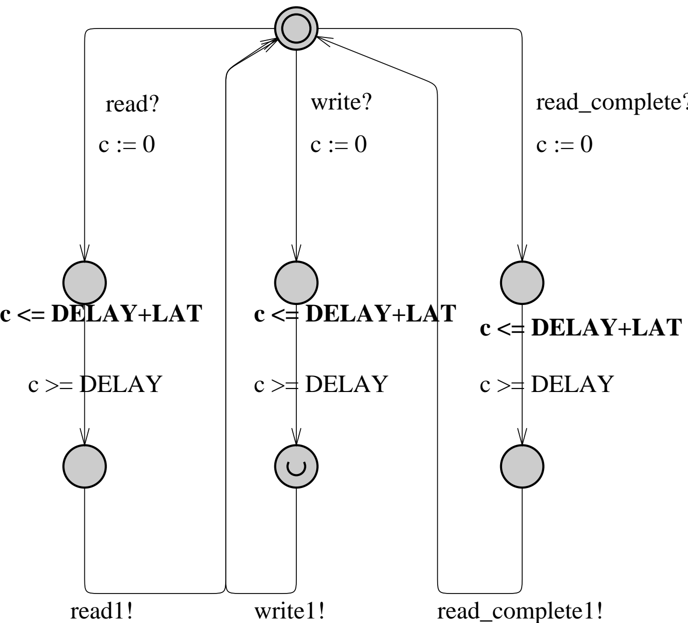


The controller controls the update cycle. `TC` and `CC` are constants of `10` and `1`, respectively, and `c` and `cc` are local clocks. The cycle is initiated every TC time units, and when the read has been completed the old value is stored for use in a later computation that takes `CC` time units. The computation result is the new value for the plant's input variable, `uhat`, and the value is required to be within `MAXu` and -`MAXu`.

Figure 5.11: The template of the controller.

The controller in Figure 5.11 initiates and controls the read and write cycles by sending messages to the bus.

When a message is sent to the bus in Figure 5.12, the message is delayed before it forwards the message to the receivers.



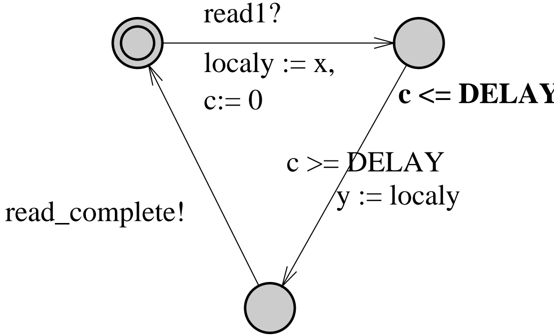
The bus delays messages by at least `DELAY` time units and at most `DELAY+LAT` time units for a write message and even more for other messages. `DELAY` is a local constant with the value `1` and `LAT` is also a local constant with value `2`.

Figure 5.12: The bus template.

When the bus forwards the message to the sensor in Figure 5.13, the sensor will update its output variable and notify the bus, and the bus will now notify the controller.

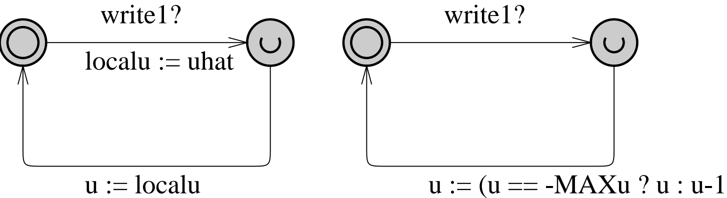
The two actuators `Actuator` and `Actuatorm1` are depicted in Figure 5.14. Both actuators wait for a write after which it updates the plant's input variable. `Actuator` updates it with the input value calculated by the controller, and `Actuatorm1` decreases the input value by one if the value stays above a defined lower boundary of the input variable.

Figure 5.15 shows the automaton template of the plant. The plant updates its plant state `x` regularly and this variable must stay within certain boundaries.



When the sensor is requested to read, it will wait DELAY time units (DELAY is again a local constant), i.e., 0 time units, after which the output variable is updated and a read_complete is signalled.

Figure 5.13: The template of the sensor.

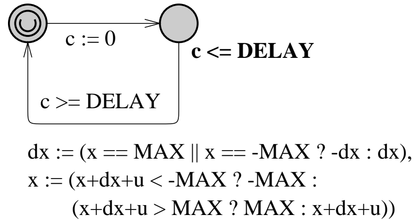


The `Actuator` is to the left and `Actuatorm1` is to the right. Both wait for a write request, but where `Actuator` updates the input variable by the value calculated by the controller, `Actuatorm1` decreases the value of the input variable by 1 if it stays above -`MAXu`.

Figure 5.14: The templates of the actuators.

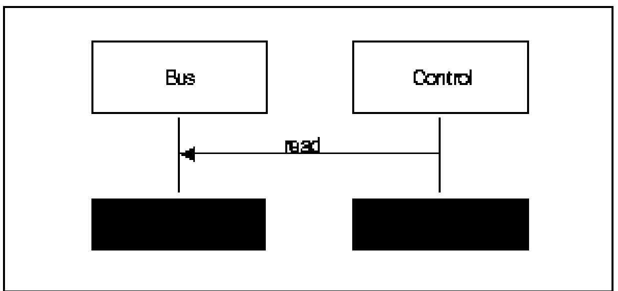
The LSCs

The behaviour of the model is controlled by the controller, which initiates read-write cycles. The cycle can be specified as one chart with a prechart, see Figure 5.16 and Figure 5.17. The bus forwards all `read`, `read_complete`, and `write` messages. Two conditions state that the cycle must be completed within TC time units and that the plant state `x` must be within -8 and 8 as specified in the condition (the values of `MAX` and -`MAX`). The simultaneous region specifies that when the bus forwards the write message through the `write1` broadcast message, only the actuators may receive it.



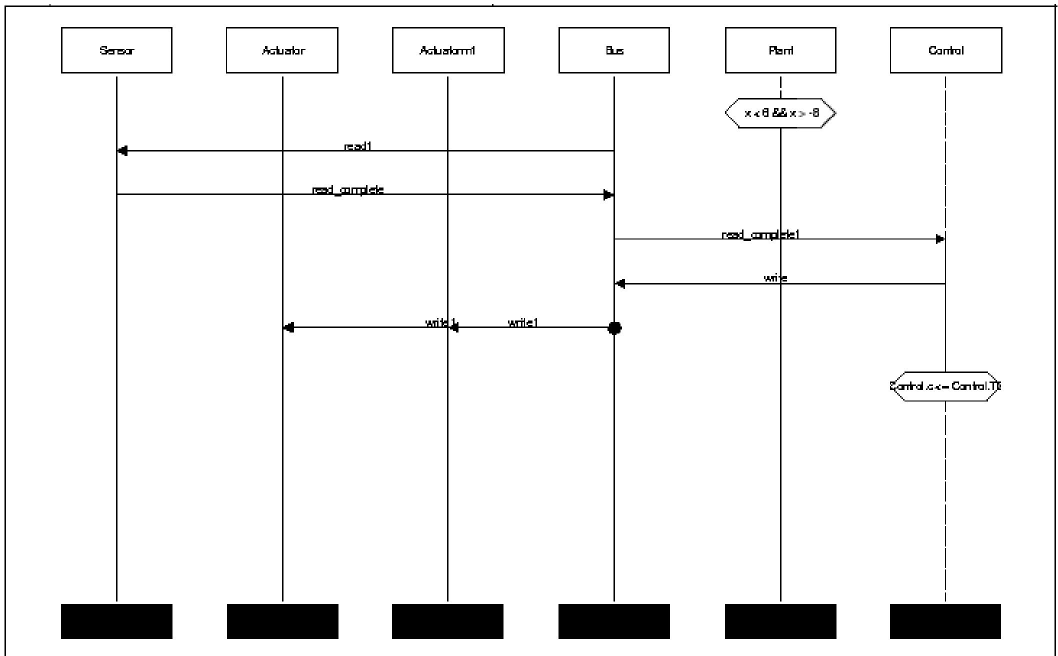
The plant updates its state after `DELAY` time units, i.e., `10` time units, and it first updates `dx` to be either `1` or `-1` depending on the direction of the sawtooth movement of `x` between its boundaries `-MAX` and `MAX`. `x` is then set to be `dx` added the value of the input variable. However, `x` is updated only if the new value is within its boundaries.

Figure 5.15: The template of the plant.



A read sent from the controller to the bus initiates the cycle for the Distributed control experiment.

Figure 5.16: LSC prechart for the read-write scenario.



All communication goes through the bus. The cycle must be completed within a certain period of time, TC time units, and the plant state must always be within a certain boundary as specified by the condition containing the expression "x <= 8 && x >= -8".

Figure 5.17: LSC chart specifying the read-write cycle of the Distributed control experiment.

PEEL has also been used to verify this model. As can be seen in Table 5.1 this chart takes significantly longer to verify than the other charts. This is because of the large number of configurations and edges that must be tested in the computation graph.

This chart shows the use of messages, conditions, and simultaneous regions which PEEL handles correctly.

Another chart has also been used to test the Distributed control model, it is the same as the one Figure 5.17, except that x is not verified to be between -8 and 8 in the condition, but instead between -3 and 3 (the values of -MAXu and MAXu), which are the boundaries of the input variable u.

LSC	Configurations	Edges	Variables	Succ. Traces	Time
Broadcast 1	5	5	5	8	0, 1 secs
Broadcast 2	5	5	5	0	0, 1 secs
Train-gate 1	1.846	2.247	14	3508	6.4 secs
Train-gate 2	1.846	2.247	14	1	0.1 secs
Train-gate 3	1.846	2.247	14	19160	5.0 secs
Distributed C. 1	14.876	29.601	15	85.462.160	3 min 30 secs
Distributed C. 2	14.876	29.601	15	0	4.0 secs

Broadcast 1 is the result from Figure 5.2 and Broadcast 2 is for Figure 5.3. The 3 Train-gate rows contain data for the charts from Figures 5.7, 5.9, and 5.10 respectively. Distributed C. 1 contains data for the chart in Figure 5.17, and Distributed C. 2 contains data for the same chart with the changed condition. The number of successful traces necessary to verify the LSC is specified for each LSC together with the time taken for PEEL to verify the charts.

Table 5.1: Statistical data for the models presented in this chapter.

5.4 Summary

This chapter has shown experiments with three UPPAAL models which have been specified by LSCs. PEEL has been used to verify that the UPPAAL models behave according to the scenarios specified by the respective LSCs. One motivation for this chapter is to show with a practical example, that LSCs in general are useful for constructing requirements specifications. Another reason is to show that all elements from the LSC subset work in PEEL and that they can be used for verifying UPPAAL models.

Next chapter contains a summary of the work presented in this project, an evaluation of the results, future work directions, and a final conclusion.

6 Evaluation and Conclusion

Contents

6.1 Project summary 95
6.2 Evaluation 98
6.2.1 LSCs and TCTL 98
6.2.2 LSCs as requirements specification for UPPAAL models 100
6.2.3 The PEEL verification engine 100
6.3 Future work 101
6.3.1 Integration of LSCs in UPPAAL 101
6.3.2 Symbolic instances 103
6.4 Conclusion 103

This chapter contains a summary of the work presented and an evaluation of the results obtained in this project. Next the subject of LSCs is put into perspective in a section presenting possible future work directions. Finally, a conclusion on the subject of using LSCs as a diagrammatic method for specifying requirements for UPPAAL models and in general is presented.

6.1 Project summary

This section gives a summary of the work presented about UPPAAL in Chapter 2, LSCs in Chapter 3, and PEEL in Chapters 4 and 5.

UPPAAL

A brief introduction to UPPAAL is presented, the dynamic behaviour of UPPAAL models is seen as a sequence of configurations, where a configuration consists of automata locations and variable values. Two types of configuration steps exist,

internal transition step and *synchronisation step*, the former being a configuration change where no synchronisation is performed. *A sequence of configurations is a trace.*

The formal semantics for UPPAAL models is defined as a transition system for an automaton and for a network of automata.

The requirements specification language for UPPAAL models is *branching timed computation tree logic*, TCTL. TCTL expressions consist of a *local* and a *temporal* property part. The local part is an expression over automata locations and variable values, and the temporal part describes the temporal extent of the computation tree where the local property must be satisfied.

Live sequence charts

Live Sequence Charts (LSCs) are an extension of Message Sequence Charts (MSCs). The main extensions are a formal semantical basis, conditions as first-class citizens, and liveness properties. MSCs are used for initially capturing use cases in the form of informal and abstract use cases, but as the development procedure advances, the charts are refined into precise specifications, and on this basis the need for more expressive charts is created.

The LSC constructs proposed in the initial work by Werner Damm and David Harel [DH99, DH01] are used as the basis, and extensions to this basis are taken from Jochen Klose's dissertation [Klo03].

An LSC subset suitable for UPPAAL requirements specification is identified and the application of the subset is described. The elements in the identified subset are:

- *Charts:* Precharts, existential charts, and universal charts are supported.
- *Messages:* Simultaneous regions support both hot and cold synchronous messages, whereas only hot synchronous messages may be used outside simultaneous regions.
- *Conditions:* Both hot and cold conditions are supported including shared conditions.
- *Coregions:* Coregions are supported, but in a slightly modified form. Coregions are made global across the structural dimension and they have a temperature specifying whether progression is enforced within the region. They do not contain snapshots, only hot, synchronous messages.

- *Simultaneous regions:* Simultaneous regions are supported, but can only be used in combination with hot and cold synchronous messages.
- *If-then-else:* If-then-else constructs containing a condition and two sub-charts. The evaluation of the condition decides which of the charts is to be traversed.

The semantics of the LSC subset, as specification for UPPAAL models, is formally specified and the specification is used for the PEEL implementation.

PEEL

PEEL, a prototype verification engine for verifying UPPAAL models with LSCs has been implemented. PEEL implements most of the selected LSC feature set, and provides a proof of concept that it is possible to verify UPPAAL models with LSCs, especially with regard to message traces.

PEEL consists of three components:

- PEEL LSC Parser: The PEEL LSC Parser takes as input an LSC, and extracts the features which are relevant compared to the selected LSC feature set. The extracted LSC elements are then ordered as a sequence of elements to be verified.
- PEEL FSM Parser: The PEEL FSM Parser takes as input a UPPAAL model and first produces an intermediate FSM output with the UPPAAL verifyta tool, and secondly the FSM output is parsed into an FSM computation graph ready for traversion by the verification algorithm.
- PEEL Verifier: The PEEL Verifier uses the sequence of LSC elements and verifies the sequence using the FSM computation graph. The running time of the graph traversion algorithm depends on the characteristics of the FSM computation graph and the LSC chart, especially the number of configurations and the number of elements in the chart and in the coregions.

The PEEL implementation is tested using a set of experimental cases, which shows that LSCs can be used for specifying proper requirements for various UPPAAL models. The experiments also show that the different features in the selected LSC subset can be verified with PEEL.

6.2 Evaluation

This section contains an evaluation of the results obtained in this project. The evaluation is based mainly on the experiences obtained through the experiments with PEEL in Chapter 5. The project goals described in Section 1.2 are the basis for this evaluation. The goals are:

- Introduction of LSCs as a supplement to TCTL in UPPAAL.
- Introduction of LSCs as a diagrammatic requirement specification language for UPPAAL.
- PEEL as a prototype verification engine.

Discussions relating to LSCs refer to the selected subset of LSC features unless otherwise stated. The selected features are described in Chapter 3.

6.2.1 LSCs and TCTL

UPPAAL verification is currently done by specifying properties using TCTL formulae. The introduction of LSCs provide a visual diagrammatic method for specifying properties. The following discusses the differences between the two approaches in different areas; focus, intuitiveness, and prerequisites.

LSCs and TCTL are languages of different paradigms. LSC diagrams is a visual specification language inspired by elements from the UML terminology while TCTL is based on tree logic, and their foci are fundamentally different.

The focus of TCTL lies in the computation tree and its branches and nodes, i.e., configurations. There are two aspects of a TCTL formula, the temporal property and the local property, they denote the branches of the tree and the configurations of the branch, respectively. An edge in the computation tree has no notation in TCTL, and thus cannot be referred to, which means that messages in the model cannot be specified either. This fact is one of the main motivations for choosing LSCs as a supplemental requirement specification language for UPPAAL.

LSCs focus on scenarios, which consist of automata instances, interobject communication between them, conditions, and liveness requirements to be fulfilled at the appropriate snapshots. The messages that each instance may send or receive are

specified in a chronological order along the instance line. Also, by using universal and existential charts it is possible to specify whether the scenarios are provisional or mandatory. In addition, precharts in LSCs can be used for specifying scenarios that may never happen, meaning that the practitioner may specify required, allowed, and forbidden scenarios.

As LSCs is a visual, object-oriented means of specification, just like UML, and as UML is widely used in the industry, this way of specifying properties to verify will be intuitive to most practitioners, and if, or when, UML adopts LSCs, LSCs may well be used in most CASE tools and practitioners will be comfortable specifying requirements with LSCs, just like they are used to using sequence diagrams today. LSCs are thus a high level way of specifying interobject communication.

TCTL on the other hand is not that intuitive for practitioners not experienced in logics. There are several pitfalls in general logics and there are also pitfalls regarding the logics used for TCTL. An example is the imply operator:

$$p \text{ imply } q$$

if the p property is always false, then nothing is ever tested about the q property, but the expression is always true nonetheless.

A practitioner of LSCs needs only to posses knowledge about the overall requirements of the software in order to be able to specify the requirements in an LSC. If using TCTL, it is necessary to posses more detailed knowledge, e.g., about automata locations and possible configurations. For instance, if a property must hold for a subrange of configurations, then it is necessary to have explicit knowledge about the configurations and use this in constructing the correct logical expressions. In an LSC, the configurations are hidden, e.g., when using a condition, the condition is implicitly tested in the configurations determined by other charts elements.

By using LSCs it is not required to have extensive knowledge of validation and verification. Instead, the practitioner needs to know what scenarios are interesting and relevant to test. In small systems it may be easy to pinpoint the scenarios that in effect verify the model, but in large complex systems it is not trivial at all.

When specifying the scenarios including the conditions and the liveness requirements that are to be satisfied in different snapshots, the automata templates need not to be known. How each automaton is modelled including the knowledge of locations is irrelevant for the scenarios to be specified. Only the names of the

messages to be exchanged and the names of the objects exchanging them need to be known for specifying message traces and for specifying conditions the variable names must be known. Configurations and locations are thus hidden by the LSCs.

6.2.2 LSCs as requirements specification for UPPAAL models

The work with PEEL during the development and testing of the engine itself, and during the specification and testing of the experiments, has provided us with an introduction to the usage of LSCs as a requirements specification language for UPPAAL models. It is apparent that LSCs supplements TCTL with the possibility to verify that a given message or trace exists, either in a single trace or in all traces, in a UPPAAL network of automata. TCTL does not have any means for testing that a given message or message trace exists and this feature is the main benefit from the usage of LSCs. The LSCs diagrammatic requirements specification is not meant as a replacement for TCTL, but as a supplement.

LSCs do not only provide the possibility to test for a simple message trace between instances, but provide powerful constructs in the form of precharts, subcharts, coregions, and simultaneous regions that enable the construction of complex scenarios to be tested.

LSCs are more expressive than standard UML MSCs, which also means that they are more complicated to construct and understand, because of the larger set of chart constructs. The two chart representations are very similar and practitioners familiar with MSCs should have an advantage if switching to LSCs. A visualisation of a scenario provides a good means for generating an overview of a component, or the inter workings of several components, and this is also the case when doing requirements specification for UPPAAL models.

6.2.3 The PEEL verification engine

The development of PEEL, a prototype LSC verification engine for UPPAAL models, is successful. Most of the elements in the selected LSC subset have been implemented and their usage is demonstrated through the conducted experiments. The PEEL prototype and the conducted experiments show that it is possible to use LSCs as a requirements specification language for UPPAAL models.

Apart from supplementing TCTL with tests for message trace, the specification of message traces in LSCs for a UPPAAL model is intuitive and straightforward, among other things because LSCs are similar to UML MSCs, which are well known to many practitioners.

The asymptotic upper bound for the running time has been found to be

$$O(c! * m! * (c-1)!)$$

where

c	is the number of configurations in the FSM computation graph,
e	is the number of elements in prechart, chart, and subcharts, and
m	is the maximum number of messages in any coregion.

Verification algorithms for finite automata normally have to traverse the entire computation graph, which gives an upper bound of $O(c!)$. The reason the running time of PEEL is because the sequences of internal transition steps and synchronisation steps influencing processes not present in the chart needs to be traversed for each element in the chart.

Optimisation can be performed regarding constant factors of the running time, e.g., improve the performance on condition testing.

6.3 Future work

This section presents some possible future directions, introduction of symbolic instances in LSC specification, and an integration of LSCs in UPPAAL.

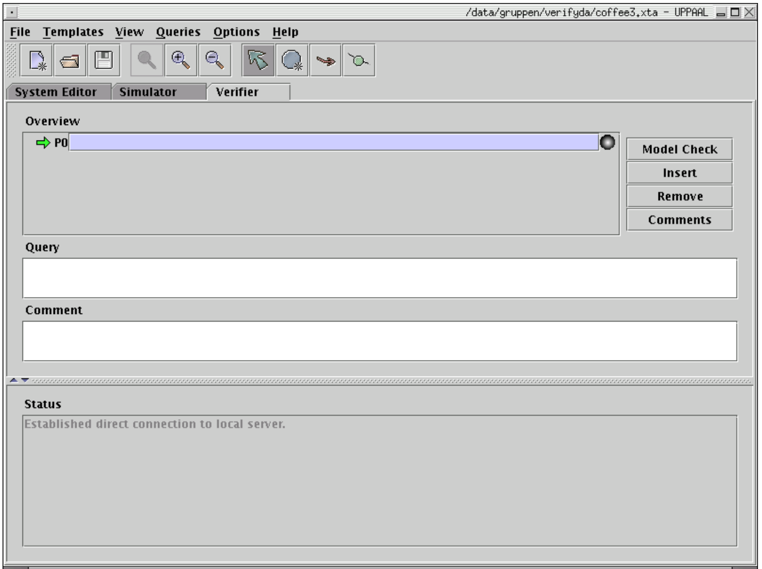
6.3.1 Integration of LSCs in UPPAAL

Currently, it is only possible to specify properties in TCTL in UPPAAL. It would be very useful if it was possible to specify LSCs within UPPAAL itself. The PEEL engine could then be used by UPPAAL's verification so that the result of the LSC specification could be viewed in UPPAAL.

This could consist of the following extensions:

- Inclusion of the LSCEditor in UPPAAL's GUI. The LSCEditor could simply be an extra tab next to the "verifier" tab. See Figure 6.1.

- Integrate the PEEL engine on top of UPPAAL's verification engine.
- Show the trace from a satisfied or unsatisfied scenario in the "simulation trace" from the "simulator" tab in UPPAAL.



Screenshot of UPPAAL, where the LSCEditor could have its own tab right next to the "verifier" tab.

Figure 6.1: Screenshot of UPPAAL.

This could mean, that when a practitioner is creating LSC specifications in the LSCEditor, he may choose the names of the LSC instances directly from a list of automata names. When writing expressions in conditions, he could choose from a list of UPPAAL locations and variables. This would be an advantage, because when the LSCEditor and UPPAAL are separated, the practitioner must be careful to name LSC instances, locations and variables exactly as they are called in UPPAAL. Thus, this extension enables the practitioner to specify requirements specifications to UPPAAL models in LSCs without having deep insight into the UPPAAL model.

Obviously, it could be argued that it requires insight into the UPPAAL model to specify proper requirements, but still, the goal is to lift the practitioner to a higher level of abstraction, and the easy naming of constructs such as instances, locations, and variables certainly helps to lift the level of abstraction.

6.3.2 Symbolic instances

The experiments performed during this project, especially the Train-gate experiments, revealed that in some cases an easier way of LSC specification exists.

The Train-gate model contains four trains, a gate, and a queue, cf. Section 5.2. Specifying LSCs with one of the trains like in Figure 5.8, it is actually required to include one LSC for each train, as the instance in the LSC must be a specific object. The number of LSCs required for specifying that when any two trains approach the gate the queue must not be empty, requires 12 LSCs.

A solution could be to introduce symbolic instances in addition to the current specific instances, such that instances instead of referring to objects may refer to the type of object, or in UPPAAL terms, to a template. This would mean that it would be possible to specify the above scenario using only one LSC, and it would cover all train instances, including any instances to be introduced in the future, whereas the current way would require new LSCs for any new train instance in the model.

Also, having two instances of the same template in a chart would refer to any two unique instances. Then it would also be easy to specify any combination of objects of the same type, just as needed in the LSC mentioned above.

6.4 Conclusion

This project has presented LSCs as a new approach for specifying requirements specification for UPPAAL models. LSCs are not meant to replace the existing specification with TCTL formulae, but is meant as a supplement. The main extension that the LSC approach provides is that of message trace verification.

Our opinion is that LSCs are a good approach for specifying message traces for UPPAAL models. PEEL implements the formal specification of the selected subset of LSC features, and thereby provides a proof of concept that it is possible to use LSCs for requirements specification for UPPAAL. In general our opinion is that LSCs are a good way to specify scenarios.

A UPPAAL Expressions

This appendix contains a syntax of invariants, guards, and updates allowed in UPPAAL. The syntax is presented in the following BNF. The information in this appendix is taken from the documentation accompanying the UPPAAL implementation.

The syntax of expressions is defined by the grammar for Expression:

```
Expression    ::=   ID
              |     NAT
              |     Expression '[' Expression ']'
              |     '(' Expression ')'
              |     Expression '++' | '++' Expression
              |     Expression '--' | '--' Expression
              |     Expression Assign Expression
              |     Unary Expression
              |     Expression Binary Expression
              |     Expression '?' Expression ':' Expression
              |     Expression '.' ID
              |     'deadlock' | 'true' | 'false'

Assign        ::=   ':=' | '+=' | '-=' | '*=' | '/=' | '%='
              |     '|=' | '&=' | '^=' | '<<=' | '>>='

Unary         ::=   '-' | '!' | 'not'

Binary        ::=   '<' | '<=' | '==' | '!=' | '>=' | '>'
              |     '+' | '-' | '*' | '/' | '%' | '&'
              |     '|' | '^' | '<<' | '>>' | '&&' | '||'
              |     '<?' | '>?' | 'or' | 'and' | 'imply'
```

The use of the deadlock keyword is restricted to the requirement specification language, TCTL.

Associativity and precedence

UPPAAL operators have the following associativity and precedence, listed from the highest to lowest. Operators borrowed from C keep the same precedence relationship with each other, see Table A.1.

<table>
<tr><th>Associativity</th><th>Precedence</th></tr>
<tr><td>left</td><td>() []</td></tr>
<tr><td>right</td><td>! ++ −− − (unary)</td></tr>
<tr><td>left</td><td>∗ / %</td></tr>
<tr><td>left</td><td>− +</td></tr>
<tr><td>left</td><td>>> <<</td></tr>
<tr><td>left</td><td>>? <?</td></tr>
<tr><td>left</td><td>= >= <= > <</td></tr>
<tr><td>left</td><td>== !=</td></tr>
<tr><td>left</td><td>&</td></tr>
<tr><td>left</td><td>^</td></tr>
<tr><td>left</td><td>|</td></tr>
<tr><td>left</td><td>&&</td></tr>
<tr><td>left</td><td>||</td></tr>
<tr><td>right</td><td>?:</td></tr>
<tr><td>right</td><td>:= += −= ∗= /= %= &= |= =
>>= <<= ^=</td></tr>
<tr><td>right</td><td>not</td></tr>
<tr><td>left</td><td>and</td></tr>
<tr><td>left</td><td>or imply</td></tr>
</table>

Table A.1: Associativity and precedence for operators in UPPAAL expression.

Expressions involving clocks

Expressions involving clocks are restricted by the type checker. There are three categories for expression including clocks:

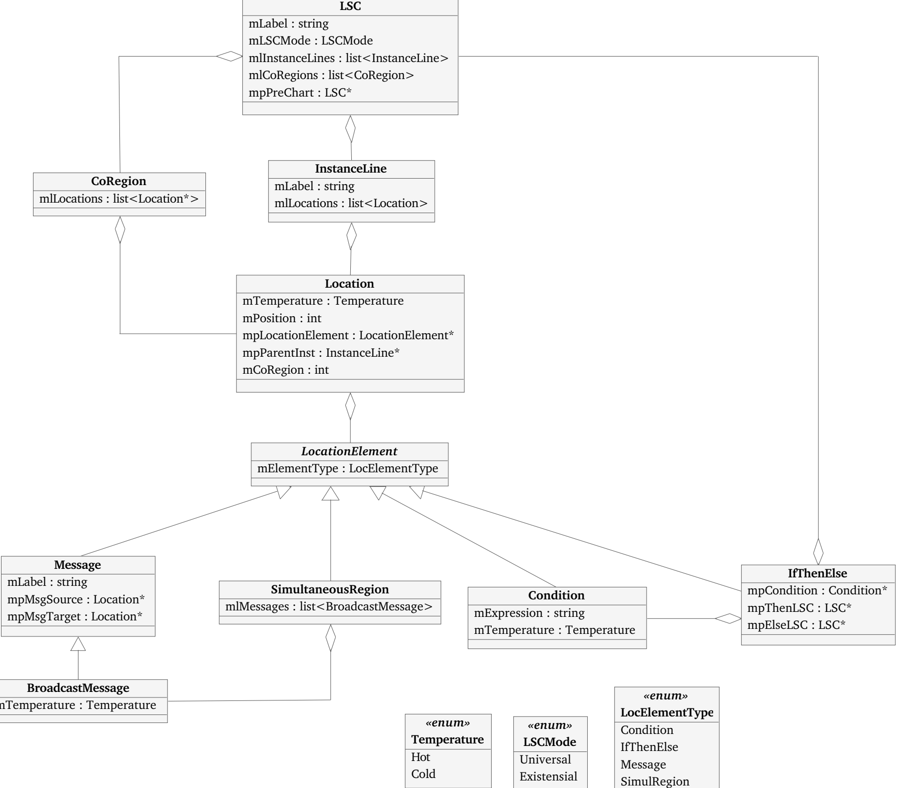
- *Invariants:* An invariant is a conjunction of upper bounds on clocks and differences between clocks, where the bound is given by a side effect free integer expression.

- *Guards:* A guard is a conjunction of bounds (both upper and lower) on clocks and differences between clocks, where the bound is given by a side effect free integer expression.

- *Constraints:* A constraint is a boolean combination (involving negation, conjunction, disjunction and implication) of bounds on clocks and differences between clocks, where the bound is given by a side effect free integer expression.

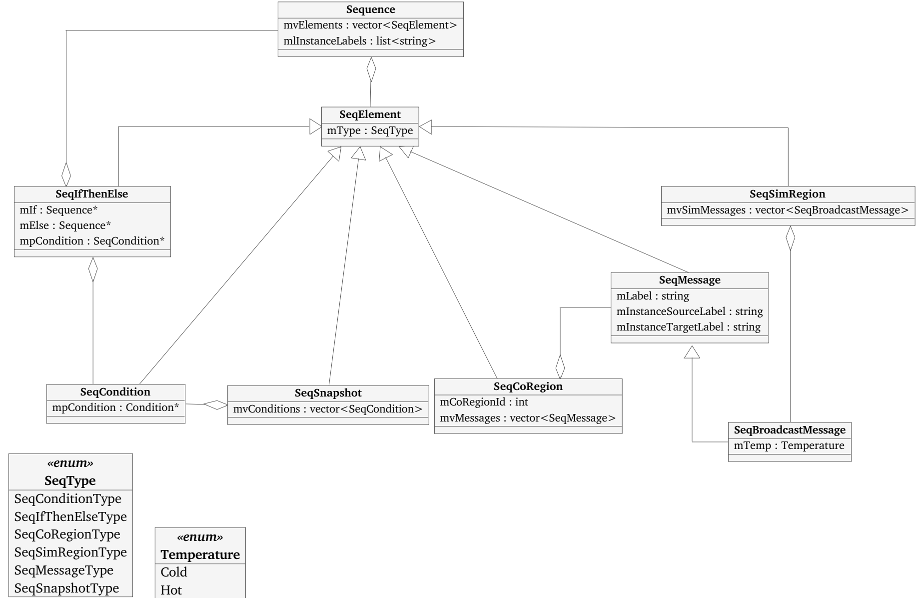
In addition, any of the three expressions can contain expressions (including disjunctions) over integers, as long as invariants and guards are still conjunctions at the top-level. The full constraint language is only allowed in the requirement specification language.

B PEEL Diagrams



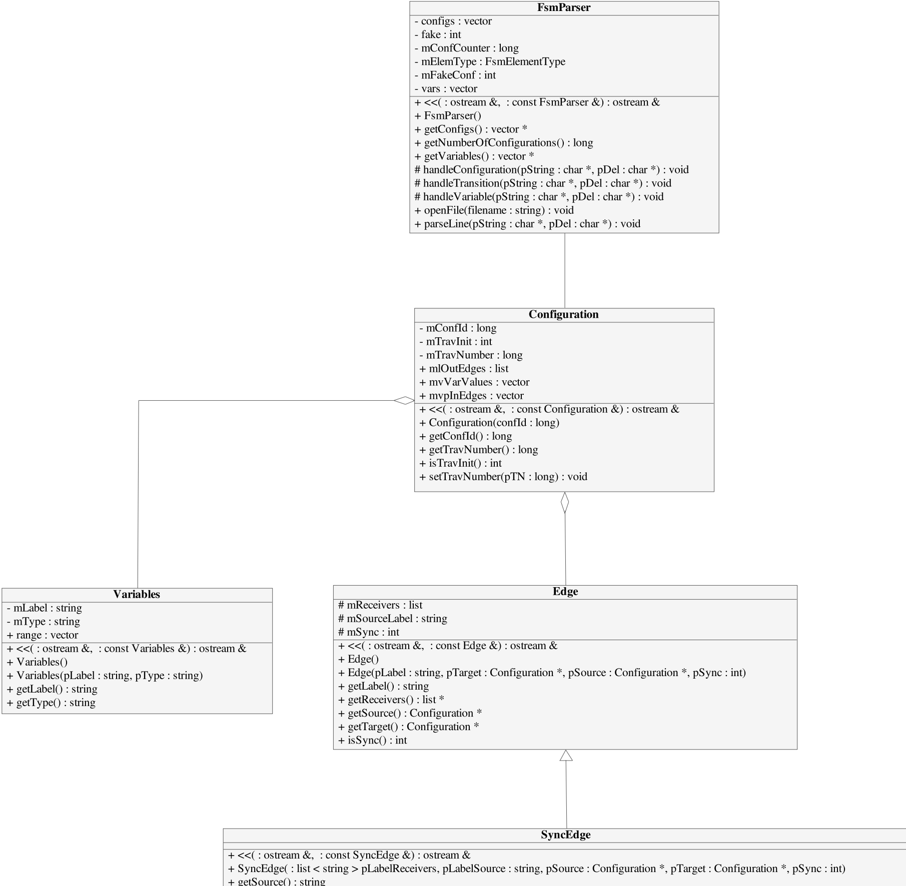
Class diagram of the data structure for holding the extraction of the LSC diagram. This data structure is an intermediate format used for generating the sequence of LSC elements to be verified.

Figure B.1: Class diagram of the PEEL LSC Parser output.



Class diagram of the data structure for holding the sequence of LSC elements to be verified. This data structure is the one being used together with the FSM computation graph to verify an LSC against a UPPAAL model.

Figure B.2: Class diagram for the sequence of LSC elements.



Class diagram of the data structure for holding the FSM computation graf. This data structure is traversed in order to verify that the LSC sequence is present, and that all properties hold.

Figure B.3: Class diagram of the PEEL FSM Parser output.

Bibliography

[BBD+02] Gerd Behrmann, Johan Bengtsson, Alexandre David, Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL Implementation Secrets. In *Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 3–22. Springer-Verlag, September 2002.

[BDW00] Tom Bienmüller, Werner Damm, and Hartmut Wittke. The STATEMATE Verification Environment – Making it real. In E. Allen Emerson and A. Prasad Sistla, editors, *12th international Conference on Computer Aided Verification, CAV*, number 1855 in Lecture Notes in Computer Science, pages 561–567. Springer Verlag, 2000.

[BDW+02] Jürgen Bohn, Werner Damm, Hartmut Wittke, Jochen Klose, and Adam Moik. Modeling and validating train system applications using statemate and live sequence charts. In H. Ehrig, B. J. Krämer, and A. Ertas, editors, *In Proceedings of the Sixth Biennial World Conference on Integrated Design and Process Technology*. Eds. Society for Design and Process Science, `www.sdpsnet.org`, 2002.

[BG01] Annette Bunker and Ganesh Gopalakrishnan. Using live sequence charts for hardware protocol specification and compliance verification. In *IEEE International High Level Design Validation and Test Workshop*. IEEE Computer Society Press, November 2001.

[BHK03] Yves Bontemps, Patrick Heymans, and Hillel Kugler. Applying LSCs to the specification of an air traffic control system. In Sebastian Uchitel and Francis Bordeleau, editors, *Proc. of the 2nd Int. Workshop on "Scenarios and State Machines: Models, Algorithms and Tools" (SCESM'03), at the 25th Int. Conf. on Soft. Eng. (ICSE'03)*, Portland, OR, USA, May 2003. IEEE. available at `http://www.info.fundp.ac.be/~ybo`.

[BJK+01] Udo Brockmeyer, Bernhard Josko, Jochen Klose, Ingo Schinz, and Bernd Westpal. Towards formal verification of rhapsody/UML designs. http://wooddes.intranet.gr/ecoop2001/SubmittedPapers/TowardsFormalVerificat%ionOfRhaspodyUML_Design.pdf, Visited 28/05 2004, 2001.

[Bjø04] Dines Bjørner. Book II - software Engineering and Formal Methods. Outline of a Series of Textbooks on Formal & Practical Aspects of SOFTWARE ENGINEERING - http://www.imm.dtu.dk/~db/the-se-books/, Visited 28/05 2004, 2004.

[BLL+95] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL — a Tool Suite for Automatic Verification of Real-Time Systems. pages 232–243, 1995. RS-96-58.

[BS03] Annette Bunker and Konrad Slind. Property Generation for Live Sequence Charts. http://www.cs.utah.edu/~slind/papers/lscAssert.html, Visited 28/05 2004, 2003.

[Dav03] Alexandre David. *Hierarchical Modeling and Analysis of Timed Systems*. PhD thesis, Uppsala University, 2003.

[dBBGdR03] Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors. *Formal Methods for Components and Objects, First International Symposium, FMCO 2002, Leiden, The Netherlands, November 5-8, 2002, Revised Lectures*, volume 2852 of *Lecture Notes in Computer Science*. Springer, 2003.

[DH99] Werner Damm and David Harel. Lscs: Breathing life into message sequence charts. In *Proceedings of the IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, page 451. Kluwer, B.V., 1999.

[DH01] Werner Damm and David Harel. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design*, 19(1):45–80, 2001.

[DMY03] Alexandre David, M.Õliver Möller, and Wang Yi. Verification of UML statecharts with real-time extensions, 2003. http://www.it.uu.se/research/reports/2003-009/, Visited 28/5 2004.

[DOTY95] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The Tool KRONOS. In Rajeev Alur, Thomas A. Henzinger, and Eduardo D. Sontag, editors, *Proceedings of the DIMACS/SYCON workshop on Hybrid systems III : verification and control*, volume 1066 of *Lecture Notes in Computer Science*, pages 208–219, Rutgers University, New Brunswick, NJ, USA, 22–25 October 1995. Springer-Verlag New York, Inc.

[Dou99] Bruce Powel Douglas. *Doing Hard Time. Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*. Addison Wesley, June 1999.

[DW03] Werner Damm and Bernd Westphal. Live and let die: LSC-based verification of UML-models. In *Formal Methods in Components and Objects*, volume 2852 of *Lecture Notes in Computer Science*, pages 99–135. Springer-Verlag Heidelberg, 2003.

[EDE01] Eran Gery, David Harel, and Eldad Palachi. *Rhapsody: A Complete Life-Cycle Model-Based Development System*. `http://www.ilogix.com/whitepaper_PDFs/Rhapsody-ifm.pdf` 28/5 2004, 2001.

[fsm] Interactive Visualization of State Transition Systems - FSM format. `http://www.win.tue.nl/~fvham/fsm/`, Visited 28/05 2004.

[HLN+88] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, and A. Shtul-Trauring. Statemate: a working environment for the development of complex reactive systems. In *Proceedings of the 10th international conference on Software engineering*, pages 396–406. IEEE Computer Society Press, 1988.

[HT03] David Harel and P. S. Thiagarajan. Message sequence charts. In Bran Selic, Luciano Lavagno, and Grant Martin, editors, *UML for real: design of embedded real-time systems*, pages 77–105. Kluwer Academic Publishers, 2003.

[i-l04] I-Logix. `http://www.i-logix.com`, Visited 28/05 2004.

[Ken99] Kent Beck. *Extreme Programming EXplained: Embracing Change*. Addison Wesley, October 1999.

[KKR02] Jochen Klose, Thomas Kropf, and Jürgen Ruf. A visual approach to validating system level designs. In *Proceedings of the 15th interna-*

tional symposium on System Synthesis, pages 186–191. ACM Press, 2002.

[Klo03] Jochen Klose. *Live Sequence Charts: A graphical formalism for the specification of communication behavior. PhD thesis*. PhD thesis, Universität Oldenburg, Germany, 2003.

[KW02] Jochen Klose and Bernd Westphal. Relating LSC specifications to UML models. In *Preceedings of the 2nd International Workshop on Integration of Specification techniques for Applications in Engineering (INT 2002)*, 2002. `http://tfs.cs.tu-berlin.de/~mgr/int02/papers/klose.pdf`, Visited 28/5 2004.

[LPY97] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1):134–152, December 1997.

[RAJGJ03] Jens Gorm Rye-Andersen, Mads Jensen, René Gøttler, and Michael Jakobsen. RIV: **R**hapsody case-study, environment **I**ntegration, model-driver **V**alidation. Master's thesis, Aalborg University, Center for Embedded Software Systems, CISS, December 2003. Theme: Modelling and verification tools for embedded programming.

[spi04] Spin - Formal Verification. `http://spinroot.com/spin/whatispin.html`, Visited 28/5 2004.

[upp04] UPPAAL homepage. `http://www.uppaal.com`, Visited 01-04-2004.

[vis04] IAR Systems homepage. `http://www.iar.com/`, Visited 28/5 2004.

[vL00] Axel van Lamsweerde. Formal specification: a roadmap. In *Proceedings of the conference on The future of Software engineering*, pages 147–159. ACM Press, 2000.