Capturing Behavioral Requirements and Testing Against Them by Means of Live Sequence Charts

Pusinskas, Saulius

Publication date:
2010

Document Version
Accepted author manuscript, peer reviewed version

Link to publication from Aalborg University

Citation for published version (APA):
Pusinskas, S. (2010). *Capturing Behavioral Requirements and Testing Against Them by Means of Live Sequence Charts*. Department of Computer Science, Aalborg University.

Aalborg University

Department of Computer Science

Capturing Behavioral Requirements and Testing Against Them by Means of Live Sequence Charts

Saulius Pusinskas

PhD dissertation

Contents

Abstract

This thesis is the result of a project carried out under the Industrial PhD Fellowship Programme at the Ericsson Telebit A/S (currently TietoEnator) as employer and matriculation as PhD student at the Center of Embedded Software Systems (CISS) at Aalborg University.

The programme is administered by Aalborg University (AAU) and co-funded by the Danish Ministry of Science, Technology and Innovation. Aalborg University and TietoEnator A/S.

The thesis aims to ease the activity of testing a system's implementation against the informal requirements stated about the system's behavior, i.e. the system specification.

Often in a new project, the specification and the implementation is developed in parallel which is not a very good practice. Having no specification or having a specification at the very early stages, the requirements are still very unstable and thus insufficient for referring to while writing the test cases.

It is a better situation when the specification is available beforehand. For most of standardized communication protocols, the specification is given in terms of the so-called request-for-comments (RFC) documents that include detailed description and requirements, of what properties the protocol implementation should possess.

In the RFC documents, requirements are present that span the behavior part of the protocol implementations, their data part, and some more abstract aspects, like accessibility of internal variables or preferred algorithms to implement some part of functionality.

Writing the test cases manually based on such informal specifications requires multiple reading of the specification document, collecting all the parts and contexts for the situation under question, defining the test scenario and writing it in the form of a test script. Errors and misinterpretation can be injected into the test script from many places. Yet another problem occurs when the specification changes, often rendering the existing test cases unusable.

Most of these problems could be bypassed by having a formalized version of the RFC, the so-called formalized specification. Test scripts are then generated automatically, and the only place worth of attention upon changed specification is the formalized specification itself.

The aim of this PhD project is to identify a suitable format of such specification, such that it can capture sufficient amount of requirements, and to develop tool support for automatic transformation of those into the test cases. The specification must be easy to understand by the requirements engineers, and have a formal semantics so that the test against requirements can be automated. Highest preference is given to the visual formalisms because of their obviousness and steep learning curve. On the other hand, the visual formalisms are not convenient for tool support, therefore a translation must be established from the visual formalism towards a simple format such as finite state machines, for which enough testing tools have been developed and available.

The visual formalism chosen is Live Sequence Charts (LSC). The features of the formalism are collected from two different semantics and extended with additional features. The specification in LSC is then translated into timed automata, and successfully used in the on-line real-time test.

The aim of the thesis is achieved through carrying out the three tasks:

- Analysis of the requirements formalization from the specifications (including RFC's), in Live Sequence Charts. The analysis gives the estimate of how many and which types of communication protocol requirements can be efficiently formalized. Having the profile of requirements to formalize, the constructs of LSC visual formalism can be borrowed or derived. This provides easy and efficient capturing capabilities of the chosen requirements.

- The tool support for the formalism analysis and automated test generation. Parts of LSC semantics are borrowed from several sources and extended with some constructs and attributes of LSC charts. As a result of that, a prototype tool is needed in order to operate on the LSC charts based on their defined semantics. To enable the testing using Live Sequence Charts, the tool chain is defined and some of its parts developed that altogether provide the needed functionality. The developed parts of the tool chain are the LSC charts editor and translator from LSC to the format where automatic test is enabled. It should be noted that the tool chain is not optimized, it is merely a proof of concept.

- Evaluation of the formalism and tool chain through academic and industrial case studies. Case studies provide the prospects for the approach of using the visual formalism to formalize requirements and then run the test against such formalized requirements.

An analysis of RFC requirements is presented in chapter 1. Several RFC documents are taken and their requirements sorted into the behavioral ones, the data-related ones and so on. A classification of requirements is made, and it is discussed which of the classes that may be formalized using the scenarios and Live Sequence Charts.

The testing tool analysis and conformance relations overview is presented in chapter 2. The existing tools and their combinations can not support the formalization of requirements in the selected variant of LSC and test against the requirements. Therefore the existing tools such as an LSC editor are upgraded and new ones such as the LSC to UppAal translator developed. These upgraded and developed tools help to fill in the gaps in the tool chain. The chain then allows for testing against specifications consisting of requirements in LSC scenarios. The tool chain in detail is presented in figure 1.

The formal semantics of Live Sequence charts and UppAal timed automata are presented in chapters 3 and 4, respectively. The chapters begin with the Smart Lamp example whose UppAal model comes together with the UppAal TRON tool. The example is presented in the corresponding formalism. After the example, the formal semantics of the formalisms follows.

The translation process from LSC to UppAal timed automata is described in chapter 5. The concept of translation is presented, and the semantic correspondence between these two formalisms is formulated.

Chapter 6 overviews the technical side of the tool chain. Aspects are covered from storing the LSC charts in predefined format to the in-detail translation from LSC to UppAal, where the LSC chart attributes and constructs are translated into the UppAal timed automata components such as locations, transitions, communication channels and so on.

Case studies are presented in chapter 7. Several of the case studies are academic, e.g. the smart lamp which has already been introduced in section 3.2. There is also an industrial example, which has the RFC state-machine requirements captured for the Dynamic Host Configuration Protocol [Dro97] client part.

Chapter 8 includes conclusions of the PhD project. There are also proposals of improvements introduced for the tool chain and its components.

The thesis does not include appendices. The important information regarding the case studies, defined LSC syntax and semantics, and translation details are present in chapters 3 through 7.

Contribution

The contributions of thesis are

- Determining what set of LSC constructs is sufficient for capturing the informal textual requirements and motivating the choice through examples and industrial case studies.
- Defining the semantics for the chosen LSC constructs or adapting one from its existing semantics
- Translation from LSC specification to a UppAal timed automata model
- Showing that the UppAal tools family can successfully use the translated specifications for several purposes:
  - testing with UppAal TRON is feasible with large translated specifications (specifications consisting of many more charts can be used in testing than ones for model checking), and several smaller timed and untimed models succeeded as test oracles for their corresponding implementations
  - model checking of the networks of timed automata against properties expressed as LSC in UppAal was feasible for specification with limited amount of translated charts, for untimed specifications such as Automatic Telling Machine (ATM, section 7.1). It is less efficient because of translating, and not building the model directly in UppAal
  - Simulation of translated models in UppAal family tools is possible, but the primary functions of these tools are hard to perform on translated models because of the LSC semantics maintenance mechanism overhead in the models. Obviously, the tools maintaining and operating directly on LSC charts would be preferred to the existing tools.
  - Several case studies have been performed on the translated LSC specifications. They have served in defining the necessary set of LSC constructs to be used while capturing requirements. Case studies have also allowed to estimate suitability of translated specifications for testing. Comparison of performance has been carried out for one of the case studies, where the tanslated LSC specification of the intelligent mouse has been compared against its model defined in UppAal. An industrial case study has been carried out, where the LSC specification of DHCP client has been defined. Performance test has been run over the client's implementation against its translated untimed specification, to show that the implementation conforms to the specification.

Rich picture of the contribution

In figure 1, the set-up of the requirements capturing (formalization), translation and utilizing is shown. The boxes with the text inside in the picture are as follows:

LSC editor is the LSC editor originally taken from Oldenburg University, Department of computer science and used in Master thesis "Property Extraction Engine for LSC" [RAJtJ04]. Written in JAVA, the LSC editor has been extended with additional constructs.

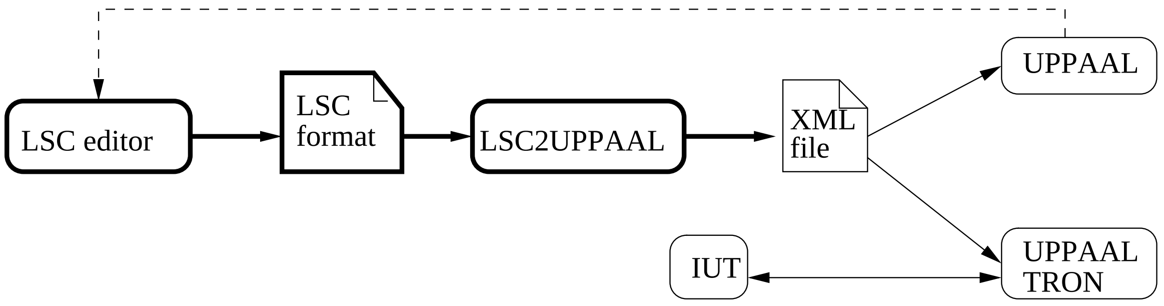

Figure 1: Rich picture of the thesis contribution. The tool chain is represented which helps us achieve the goals of requirements formalizing, translation to UppAal models and utilizing them. Personally developed parts of the tool chain are represented by thickened frames and arrows.

LSC format is the extension of the LSC chart format originally defined for the LSC editor. The syntax of the data in the LSC format file is presented in BNF form in section 6.1.

LSC2UPPAAL is the translator, that has enabled most of contribution of this PhD thesis. It translates the LSC chart, submitted as properties and observed properties, into the UppAal model that consists of timed automata from the properties in LSC plus some auxiliary automata to enforce the LSC semantics translation and optionally some observer automata from observed properties, respectively.

XML file is the UppAal model file stored in well defined syntax. It is the standard format used by UppAal and UppAal TRON tools.

UPPAAL [LPY97] is the main tool used in debugging the LSC specifications so far.

UPPAAL TRON [MNL03] is the tool which makes most use of the translated LSC specifications. The translated specifications are used for generating tests of *IUT* for *TIOCO*-conformance. The *TIOCO* conformance relation is described in section 2.2.

IUT is arbitrary implementation which is tested against its specification using the tool UppAal TRON.

Arrows connecting the boxes have the following meaning:

LSC editor $\longrightarrow$ LSC format represents the saving procedure of the LSC charts drawn in the LSC editor for their later reusal in translation. The LSC format file is presented in BNF form in section 6.1.

LSC format $\longrightarrow$ LSC2UPPAAL represents the parsing of the previously saved sets of LSC files for translation into UppAal automata.

LSC2UPPAAL $\longrightarrow$ XML file represents the actual translation of the LSC specifications into the UppAal networks of timed automata. Translation can be performed in several flavors and depend on the purpose of using the translated LSC charts. The chosen flavor of translation has been described in chapter 5.

XML file $\longrightarrow$ UPPAAL stands for a few applications of the translated UppAal model:

- Simulation and debugging of the LSC specification that has been translated into NTA. Besides the translated automata, several auxiliary automata and variables are necessary to transfer into the model as well. The amount of auxiliary variables and automata may vary depending on how the translation is defined.
- Extracting the LSC properties translated into observer TA and pasting them into the models originally constructed within UppAal. Several variants of the translation exist, where some of them do not need auxiliary automata. Minimal requirements for the model where the translated property is pasted is to have the channels broadcast and the variables referred in observer automata shared, so that the observer automaton is updated about the events and values of the necessary variables.

XML file $\longrightarrow$ UPPAAL TRON stands for using the specifications defined in LSC for on-line real-time testing using the tool UppAal TRON. Monitoring can also be performed given the IUT, its environment and the LSC models of them both translated.

UPPAAL TRON $\leftrightarrow$ IUT stands for performing the test of implementation against specification that consists of translated LSC charts.

UPPAAL $\longrightarrow$ LSC editor stands for debugging of translated LSC specifications, by simulating the corresponding NTA in UppAal. It is not the part of the test against requirements, rather means of feedback while debugging and modifying the formalized specifications. The line is thus dashed. Typically the feedback consists of the counter-example event trace in an UPPAAL editor when a property has been violated. It can also be the event trace that leads to the configuration of the model which is not supposed to be reached.

Related work

Live Sequence Charts have been introduced as extension of message sequence charts in [DH01]. Several new concepts and constructs have been added to LSC like multiple instances and symbolic variables [MHK02], discrete time and forbidden elements [HKP04]. The behavior of the LSC specifications is defined by means of *supersteps*, possibly alternating sequence of actions in response to a single event from the environment. Altogether these concepts are implemented in a simulation tool Play-engine which is presented in [HM03].

An automata based semantics of Live Sequence Charts has been introduced in [KW01], [BDK+04], [BDK+04]. Approach of efficient formal verification of Live Sequence charts is proposed in [BBD+04], [KTWW06].

Symbolic execution of the behavioral requirements by using Constraint Logic Programming (CLP) has been demonstrated in [WRYC04]. The Play-engine [HM03] can be used for playing scenarios in and simulating (playing out) them later. Discrete time (clock tick events) are used as a means to define and simulate the timed scenarios.

Automatic testing against communicating automata is implemented in the TorX tool [TB03]. The test verdicts are based on IOCO conformance relation. For conformance testing against timed automata, the tool UppAal TRON [MS03],[MNL03],[MLN03],[MLN04],[LMNS05] is available. Play-engine [HM03] is used in testing the implementations against the LSC charts directly.

Converting LSC into the code has been recently analyzed in [MH06].

An on-the-fly algorithm for model-based test suite generation based on the coverage of selected (coverage) items is introduced in [HP07]. Test generation from timed automata is described in [NS01]. Among recent results in testing against discrete-time LSC specifications the Play Engine [HM03] has been deployed. On-line real-time tests are carried out using the UppAal TRON tool. There is also a tool, Timed Test Generator (TTG) [KT04], built on top of the IF environment [BGMO04].

Chapter 1

Requirements

In this chapter, an overview over the requirements for the communication protocols is performed. Requirement types and their behavioral properties are identified that need to be captured in some, preferably visual, formalism. RFC documents [Pos81, Bra89, Dro97] are used in analysis since their requirements are most frequently applied to the behavior of the IP products developed.

Requirements from RFC have several priorities, and these are described in section 1.4.1. Requirement priorities are disregarded by the known testing tools, so their priority maintenance must be performed outside such tool.

Emphasis in this chapter is put on how to analyze and formalize the requirements. The set of formalized requirements can be obtained by formalizing the already available specifications and by analyzing the high-level requirements of the system yet to be designed, by splitting them into more concrete requirements and reshuffling them. Both ways aim at the same result, namely the set of requirements in a formal language, so techniques and formalisms are preferred that are supported in these ways.

Four types of requirements are identified from RFC documents that are presented in section 1.4.2. Two requirement types are chosen that are simple to capture and that constitute the largest part of the overall requirements. These two types are then dealt with during the thesis.

1.1 Definition of the requirement

According to [oEE90], requirement is defined as (1) A condition or capability needed by a user to solve a problem or achieve an objective; (2) a condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents; (3) a documented rep- resentation of a condition or capability as in (1) or (2)..

The concept of requirement will be mainly used as the part (2) of the definition as the system is in focus. Often requirement will be referred to as a property.

1.2 Requirement formalization

The requirement formalization, or requirement capturing, is the first phase of converting the system properties from the spoken English into the language with formal semantics. Capturing requirements in formal semantics leaves no place for ambiguities, mis-interpretations and other misunderstandings that originate from informal language. Formalized requirements are clearly stated, therefore remaining contradictions among them can be detected by some means, re-defined and removed if needed.

Requirements spanning over communication events, timing and data parts of the protocols will be formalized. Formalization of such requirements will be performed into the language with maintenance of discrete and boolean variables and optionally data structures.

One of the ways to formalize requirements is in LTL or CTL [WRHM06]. Behavioral requirements are proposed to formalize by means of refined finite state machines [PMBW00]. Each of these proposals have drawbacks. For example, the finite state machines and their possible interactions being hidden from the analyst eyes and harder to understand than scenarios. The LTL or CTL approach is also cumbersome when capturing non-trivial requirements.

A representative set of formalisms candidates for requirements with discrete time follows:

- The KAOS (Keep All Objectives Satisfied) approach [vL03]. The requirements in this approach are formalized into the RT-LTL formulas. The functional and non-functional requirements, such as security, safety, accuracy, cost or performance, are aimed to formalize.

- Subset of use cases, the so called Timed Use Cases Maps [HRD06], describe Use Cases interactions using global and local timers.

- Visual Timed Events Scenario [BKO05] capture events interactions, where event is an action that potentially occurs inside the system.

- Timing Diagrams. [CF05] provides the LTL-based logics for timing diagrams.

- UML Real-Time Profile [OMG02] where variety of aspects is described such as timing, performance, resources, schedulability.

- Metric Interval Temporal Logics (MITL) [AFH91], variation of LTL, where timing constraints are specified by means of intervals. Translation form MITL formulas to timed automata using a simple procedure is defined in [MNP06].
- Temporal Rover, the software where the textual requirements can be coded as the statechart-assertions or MSC-assertions. Unlike the text-based temporal assertions, statechart assertions are visual, intuitive, and resemble statechart design models [DSD07].
- Live Sequence charts as extension to Message Sequence Charts, using discrete clock [DH01] [HM03] [BGM04] [DW05].

Out of the overviewed ones, the visual and thus more readable ones are the Live Sequence Charts and UML diagrams.

The Live Sequence Charts have many advantages to mention:

- LSC is going to become a part of UML
- Rich set of constructs is used in LSC. This results in capturing the requirements of arbitrary complexity in a small amount of LSC onstructs. Such LSC are easy to read and interpret
- LSC is based on scenarios, what is close to the human perception
- As a graphical formalism, it is preferred due to its convenience over the textual ones such as LTL
- Temperature of the LSC constructs allows overriding restrictions by events, what makes it easy to ”shape” the original LSC chart upon slight change of requirements, without substantially distorting or redefining the LSC chart

1.3 Requirements analysis

Requirements analysis and processing is applied when there exists no detail concept of the system yet. Some high-level requirements might be only available, telling what the system should be capable of and using what techniques it should be achieved. Result of the requirement analysis results in concrete, captureable requirements towards the system or its parts.

The two methodologies of requirements analysis are chosen by W2C Working Group for Web Services Architecture (web services protocol specifications comprise the larger part of the RFCs). These methodologies are the Critical Success Factor (CSF) Analysis method, which is supplemented through the use of gathering Usage Scenarios [Gro04].

The Critical Success Factors Analysis methodology for determining requirements is a top-down means of determining requirements based on the needs of the organization [Gro04]. Informally, this corresponds to determining hierarchy and dependence of requirements.

By analyzing the steps necessary to achieve success, and cross-referencing them against problems to be solved, a complete set of requirements can be determined that can then be correlated with specific user scenarios. Each of the requirements should apply to at least one user scenario, and generally more than one [Gro04].

Such interaction of these methodologies indicates that the requirements are covered by the user scenarios. This is exactly what happens in specifications consisting of visual scenarios like Live Sequence Charts.

Concrete requirements from RFC documents are analyzed in section 1.4. Test framework is presented in section 2.3.

Ways to define the specification

It is in our interest to operate over specifications, not only the loose requirements. Specification of a system consists of a set of requirements that apply to the same system. Therefore preference will be given to notations that maintain both standalone requirements and specifications thereof.

Yves Bontemps PhD thesis [Bon05] has several of the formats presented to capture specifications; the favored one in his thesis has been Live Sequence Charts. So has Bunker et al [BGM04], also motivating usage of LSC instead of MSC [BG01] in the formal specifications of hardware protocols. There are mainly two groups of notations to define behavioral specifications of the systems:

- Formal notations like Z-notation, temporal logic, duration calculus [CHR91]
- Visual languages (MSC, LSC, Statecharts, UML, SDL, diagrams) among whom are scenario based languages, covered in [AE03]

As can be seen, the Live Sequence Charts are used to specify both standalone requirements and the specifications. This is another advantage offered by LSC.

1.4 RFC requirements: case study

In RFC documents, requirements are typically oriented at the behavior, functionality, data constraints and other features of specific protocols. For IP protocols, a concept of a *host* is often used, which means the instance of the protocol implementation which runs on the hardware (computers connected to intranet or Internet, routers etc.) and communicating with other instances of the protocol implementations on the same or different hardware pieces.

The classification of requirements is very much debated. Some argue that there should not be any classification. We will choose such classification of requirements in such a way that makes an easy distinction between several classes of requirements. The first classification is the functional and non-functional requirements, the second classification is related with the behavior and static constraints of the systems modelled.

1.4.1 Requirement groups by priority

The requirements in RFCs can be classified into the five groups in general, that form the three levels of priority, that are the MUST (MUST NOT), SHOULD (SHOULD NOT) and MAY. The description of the groups often comes as a part of the RFC document:

```
Throughout this document, the words that are used to
define the significance of particular requirements are
capitalized. These words are:

      "MUST"

        This word or the adjective "REQUIRED" means that the
        item is an absolute requirement of this specification.

      "MUST NOT"

        This phrase means that the item is an absolute prohibition
        of this specification.

      "SHOULD"

        This word or the adjective "RECOMMENDED" means that there
        may exist valid reasons in particular circumstances to ignore
        this item, but the full implications should be understood and
        the case carefully weighed before choosing a different course.

      "SHOULD NOT"

        This phrase means that there may exist valid reasons in
        particular circumstances when the listed behavior is acceptable
        or even useful, but the full implications should be understood
        and the case carefully weighed before implementing any behavior
```

```
         described with this label.

       "MAY"

         This word or the adjective "OPTIONAL" means that this item is
         truly optional.  One vendor may choose to include the item
         because a particular marketplace requires it or because it
         enhances the product, for example; another vendor may omit the
         same item.

An implementation is not compliant if it fails to satisfy
one or more of the MUST requirements for the protocols it
implements. An implementation that satisfies all the MUST and all
the SHOULD requirements for its protocols is said to be
"unconditionally compliant"; one that satisfies all the MUST
requirements but not all the SHOULD requirements for its protocols
is said to be "conditionally compliant".
```

The conformance test can only reveal what requirements are dissatisfied by the behavior of the implementation under test. The SHOULD and MUST requirements are treated in the same way, and priority of the violated requirement needs to be identified outside the test procedure.

Some examples of the MUST requirements:

1. *[From the DHCP client point of view,] any arriving DHCP messages MUST be silently discarded" [Dro97, p. 36]*

2. *"The DHCP client MUST NOT include a 'server identifier' in the DHCPREQUEST message" [Dro97, p. 39]*

3. *"DHCPINFORM messages MUST be directed to the 'DHCP server' UDP port" [Dro97, p. 39].*

Some examples of the SHOULD requirements:

1. *"The client SHOULD wait a random time between 1 to 10 seconds [prior to sending a DHCPDISCOVER message while in INIT state] to desynchronize the use of DHCP at startup" state [Dro97, p. 36]*

2. *"When allocating the new address, the server SHOULD check that the offered network address is not already in use; e.g. the server may probe the offered address with an ICMP Echo Request" [Dro97, p. 14]*

3. *"The client SHOULD perform a check on a suggested address in order to ensure that the address is not already in use" [Dro97, p. 38].*

Some examples of the MAY scenarios:

1. *"The server MAY choose to mark addresses offered to clients in DHCPOFFER messages as unavailable" [Dro97, p. 16].*

2. *"A DHCPREQUEST message MAY [apart the 'server identifier' option] include other options specifying desired configuration values".*

The MAY statements (options) do not put restrictions on the protocol implementation. As requirements, they are useless since they do not put any restrictions on the behavior. When building the model, they are vital since they describe what behavior of the model can be expected in particular situation, and help with the estimate, what can be the state of the model afterwards.

1.4.2 Types of requirements in RFC

Depending on the RFC specification and subject, the requirements can be sorted into sets according to several criteria. The primary criterion of sorting is the number of protocols involved in the requirement. The two sets can be defined for the requirements in general:

- The requirements spanning over a single protocol. Some examples follow:
  1. *If TTL field contains the value zero, then the [IP] datagram must be destroyed.*
  2. *The least (largest) legal value for parameter is X (some integer number).*
- The requirements spanning over several protocols. In comparison with the first set, such requirements need more details taken into account such as conventions and naming for several protocols, also merging of specifications from several protocol models. Some examples of such requirements follow:
  1. *If the client detects that the address is already in use (e.g., through the use of ARP), the client MUST send a DHCPDECLINE message to the server and restarts the configuration process.* Here two different protocols - DHCP and ARP - are mentioned in the requirement.
  2. *Echo Reply messages MUST be passed to the ICMP user interface, unless the corresponding Echo Request originated in the IP layer.* As in the earlier example, ICMP and IP protocols are present in the requirement. The scenarios that mention the user interface (informing the user, displaying the error message or similarly formulated) are also assigned to the group of several protocols.

The further focus will be set on the class of requirements spanning a single protocol.

The requirements spanning over a single protocol can then be sorted further into several groups according to the type of constraints they introduce onto the data, events and timing of events. Four groups have been defined:

1. Requirements that specify the reactive behavior of the host. They are expressed in scenarios where upon timeout, arrival of some packet or other external event, certain reaction is expected (or, prohibited). Based on a set of such requirements, the behavior of the protocol can be modelled as a finite state machine or timed automaton. The reactive requirements form the largest group (around a half of all the requirements in the three RFCs studied so far, namely [Pos81, Bra89, Dro97].

2. Data constraints for the arriving or sent packets and internal data of the host. Observer automata are best suited for such a purpose, since certain checking must be performed upon an event of sending or receiving the data packet or command. Such constraints can also be expressed in scenarios if coupled with the events like packet sending (receiving) or timeouts. Implementation of such scenarios consists of invoking a script (procedure) to check all the applicable constraints on an incoming or outgoing packet or a timeout after a packet or event. Such scripts or procedures should be implemented separately from the model, in the "glue" layer since they do not affect the behavior or data part of the model. Besides, it is costly to maintain the model with too many data variables, and utility of such too-precise model is questionable. Such data constraints form the second largest group, formed of approximately a quarter of all the requirements.

3. Configuration requirements for the protocols. They are usually statements of the form *"The protocol implementation MUST support the modes X_1, X_2, …, X_n of the feature X, and it SHOULD default to X_i for some $i \leq n$."* Such requirements are tested manually, and the best what the automatic testing can benefit with, is to detect which mode the system under test is currently in.

4. Implementation requirements for the protocol. They are usually statements of the form *"The protocol implementation MUST include a mechanism A to deal with the problem B"*, or *"There MUST be means for the functionality X of the implementation"*. A part of these requirements are too abstract to be modelled. For the rest of them, it is possible to detect whether the mechanism (algorithm) is implemented correctly via testing against a set of scenarios that comprise the model of that mechanism or algorithm.

Requirement from the first or second group corresponds to the functional requirement, that by definition, is a software requirement that specifies a function that a system or its component must be capable of performing. These are software requirements that define behavior of the system, that is, the fundamental process or transformation that software and hardware components of the system perform on inputs to produce outputs..

On the contrary, definition of the non-functional requirement is a software requirement that describes not what the software will do, but how the software will do it. Examples include software performance requirements, software external interface requirements, software design constraints and software quality attributes. Non-functional requirements are usually difficult to test; therefore, they are usuallyevaluated subjectively..

This definition describes very much the requirements of the fourth group.

The first two groups of the four have been chosen to investigate due to several reasons:

- These two groups cover the major part (three quarters) of the requirements
- The requirements of these two groups are typically tested using test scripts
- In most cases, the requirements of the group 4 can be expressed as a set of the requirements of group 1 and 2. The same applies to the group 3 requirements: the first two groups would be used to detect the current configuration defaults.

- It is too time- and resource- consuming to investigate thoroughly groups 3 and 4 because of their high level of abstraction.

Further investigation has found a set of properties that draw a line between the two chosen groups of the requirements. The requirement belongs to the group 2 (i.e. is not reactive) if

- The requirement is applied to every packet, and no causality relation is defined between the packet of interest and the other (previous, successive) packets. The requirement can also refer to the type of the internal variables used in the protocol implementation.
- No timing constraints are present in the requirement.
- No conditions are present and variables affected during the check of requirement that affect the system behavior afterwards.
- The judgement (restriction) does not depend on any additional data stored or configuration or the host.

Some examples of the data part addressing requirements:

1. *"The DHCP client MUST NOT include a 'server identifier' in the DHCPREQUEST message"*
2. *"DHCPINFORM messages MUST be directed to the 'DHCP server' UDP port"*
3. *"The client broadcasts a DHCPREQUEST message that MUST include the 'server identifier' option to indicate which server it has selected, and that MAY include other options specifying desired configuration values"*
4. *"The server MUST broadcast the DHCPNAK message to the 0xffffffff broadcast address because the client may not have a correct network address or subnet mask, and the client may not be answering ARP requests"*

The requirement belongs to the group 1 (i.e. is reactive) if

- Timing constraints are present in the requirement
- Causality relationships are defined between the packets
- Conditions are present that affect the behavior of the system afterwards
- The change of the behavior or the state of the model is caused by the incoming or outgoing packets, and not the external configuration (then, the requirement would belong to the third group).

Some examples of the behavior part addressing requirements:

1. *"The client SHOULD wait a random time between 1 to 10 seconds [prior to sending a DHCPDISCOVER message while in INIT state] to desynchronize the use of DHCP at startup state"*

2. *"When allocating the new address, the server SHOULD check that the offered network address is not already in use; e.g. the server may probe the offered address with an ICMP Echo Request"*

3. *"The client SHOULD perform a check on a suggested address in order to ensure that the address is not already in use"*

4. *"Any configuration parameters in the DHCPACK message SHOULD NOT conflict with those in the earlier DHCPOFFER message to which the client is responding."*

5. *"If the selected server is unable to satisfy the DHCPREQUEST message (e.g., the requested network address has been allocated), the server SHOULD respond with a DHCPNAK message."*

6. *"If the client used a 'client identifier' when it obtained the lease, it MUST use the same 'client identifier' in the DHCPRELEASE message."*

7. *"If [during the abbreviated address-acquisition procedure] the client receives a DHCPNAK message, it cannot reuse its remembered network address. It MUST instead request a new address by restarting the configuration process, this time using the (non-abbreviated) procedure"*

The requirements of the form *"Messages with some data pattern P MUST be silently discarded"* are not taken into account in the model, it is the task for the glue layer to pass only the messages and packets with correct syntax, unless there is a convention in the model to represent all malformed packets with a certain event.

Chapter 2

Testing

In this chapter, an overview over some conformance test relations is given, and the tools are presented that run the conformance test according to these relations. Usage of requirements and scenarios as the basis for the model to test against is discussed. The sketch is given, how the visually expressed requirements can enhance the testing process when compared to the conventional test scripts.

2.1 Model based testing

Correspondence of the implemented system towards its specification is assessed through testing. One of the thesis aims is to improve the testing so that it achieves higher quality and speed in comparison with standard methods such as the manual testing or script writing. The obvious solution to improve testing performance is automation of the testing. Certain details and theories behind the automated testing are covered in this chapter.

A finite test is an experiment with the system under test. The formal model of the system is used as a reference, and the result of an experiment is one of the values from the set Yes / No / Inconclusive.

2.1.1 Difficulties arising from the informal description of the system

A number of standards have been defined to maintain the software products by different vendors interoperable. The verbal description of the standards for software like communication protocols is publicly available for evaluations, alterations and requests.

Internet Engineering Task Force (IETF) organization takes care of the communication protocol standards from their proposal to the final version, when they are assigned a status of Request For Comments (RFC). The vendors need their implementation of the protocols to be evaluated and tested for conformance with respect to the standard in order to get a compliance certificate.

The TietoEnator A/S company has development of the IP products among its activities. Complying to the communication protocol standards is one of the absolute requirements during the software development. This has motivated the search of a convenient way to comply to the requirements and assess the compliance through the test. This thesis attempts to do that.

In most cases, it takes longer than expected to obtain the compliance certificate for the implementation. Several attempts and numerous tests as well as alterations of the implementation are performed in the mean time. Typical reason for that is misinterpretation of the spoken language-based protocol specification in the RFC document. Another reason is the manual creation of the conformance test cases. As the software is becoming more complex and more safety-critical, the manually written test cases tend to be insufficient. A model-based approach allows teams to build software systems with measurably higher quality, in less time than with non-model based approaches [BBNS03].

The semantics of the spoken English language varies depending on the country, region or culture. Cases of deliberately misinterpreting the semantics is possible in some cases in order to ease own job or decrease the costs of the system under development. In contractual projects, the customer usually writes the requirements document, but it has so many ambiguities, uncertainties and gaps that developers must evaluate it carefully, incrementally refining and formalizing information until they have produced a functional specification [PKA94].

2.1.2 Candidates of formal languages to capture the informal specifications

Several specification languages have been developed that describe the hardware and the software of the systems (respectively, data and behavior parts of the protocols). These languages are of two types, textual and visual. The textual ones as a rule have well developed representation of the data part of the protocol. An overview of the hardware specification languages given by [BGM04]. The intuitive, graphic nature of the visual languages makes them easy to learn, and the specifications prepared in such languages are easy to learn and understand. From the visual languages, Message Sequence Charts (MSC) [IT99] and statecharts [HLN$^+$88] are introduced as a part of the Unified Modelling Language (UML), as well as Specification and Description Language [CCI92]. However, these languages lack the means to encode loops, conditions, unordered events, broadcast. Neither do they support the requirement priority levels like in RFCs. Among recent achievements in the field, the tool MSCan [BKSS06] is available, which allows checking the syntactic properties of MSC specifications and provides the protocol designer with a great variety of facilities to analyze the MSCs. However, closest to the capturing of the RFC requirement hierarchy is the Live Sequence Charts language which, according to [BGM04], has some features that make the language the promising candidate for protocol compliance verification. Together with the loop, broadcast and unordered events constructs, the LSC language makes a clear difference between mandatory (opposite - prohibited) and possible behavior or data both inside the system and at its boundaries (interfaces).

The RFC specifications like the RFC 1122 [Bra89] ("Requirements for Internet hosts - Communication layer") are too complex to comprehend and model them as finite state machine yet because of another reason: such specifications do not include the state-machine model of the protocols to test. Typically only the restrictions are present in such documents that deal with the behavior of the hosts and some data constraints on the format of the data packets that are sent or received. The scenarios are more efficient means in representing restrictions on data or behavior part of the protocol. The LSC scenarios can equally easily describe both the prohibited and necessary behavior or restrictions on data. Depending on the complexity of the specification and the number of restrictions in it, the LSC can also be overkill. But neither of the known formalisms provide such flexibility of composing the specification of requirements and compact representation of those.

An expressive formal model of the protocol standard is preferred to the (ambiguous) informal verbal description. Such a modeling formalism would substantially simplify the procedure of developing the standard-compliant implementation. It could be developed from the first attempt and fine-tuned in few modifications thus achieving shorter time-to-market. The formal model itself could be verified against particular properties to prove consistency of its requirements and serve as an unbiased specification during the implementation. In the first case, this would advance and accelerate the development of custom protocol standards; in the latter case, the formal techniques and semantics would serve as a powerful tool at the stages of the system development from its design to implementation.

2.1.3 Testing against specifications

Black-box and white-box testing

Black-box test design is usually described as focusing on testing functional requirements. Synonyms for black-box include: behavioral, functional, opaque-box, and closed-box. White-box test design allows one to peek inside the "box", and it focuses specifically on using internal knowledge of the software to guide the selection of test data. Synonyms for white-box include: structural, glass-box and clear-box.

Black- and white- box testing is often referred to as behavioral and structural testing.

On-line and off-line testing

Both off-line and on-line testing use the system model to generate the test cases for its implementation.

The difference between the on-line and off-line test is whether the model is updated according to the implementation reaction to inputs during the execution of the test case.

The on-line test allows more flexibility, namely choosing and submitting the inputs to the implementation more precisely having estimated its current state by means of outputs or their absence.

The test of the implementation against the system model in finite state machines is performed according to IOCO [Tre99] and TIOCO [BT01] conformance relations [BB04a], that are presented in section 2.2. The same relation can be applied to the testing against the set of scenarios, when the state-like information like allowed actions and delays can be calculated from the model in particular state.

Strong side of the off-line test is that it allows employing the model-checking techniques when the model of the system to test is of reasonable size. The model checking in certain cases allows obtain the set of test cases that satisfy some predefined coverage criterion. The approach of defining the coverage criteria for the models and then generating test suites from them is used by Hessel et al [HP07].

As mentioned in [MLN04], there are several advantages of online testing:

- testing may potentially continue for a long time (a single test run may take hours or even days), and therefore very long, intricate, and stressful test cases may be executed
- the state-space explosion problem experienced by many offline test generation tools is reduced because only a very limited part of the statespace need to be stored at any point in time
- online test generators often allow more expressive specification languages, especially wrt. allowed non-determinism in real-time models, e.g. allowing timing uncertainty where an output event is expected in some interval of time, say between 1 and 5 time units from now. An online test generation algorithm is automatically adaptive to this non-determinism.

2.2 IOCO and TIOCO conformance relations

A conformance relation defines what behaviors of implementation under test are considered correct compared to its specification. In the non-timed setting the formal conformance relation IOCO has been proved suitable in practice and used in many model based test tools [MLN03]. The black-box conformance test of implementations against their specifications will be performed according to IOCO and TIOCO conformance relations. These relations are defined for (timed) finite state automata, but trace-based semantics and translation from scenarios to timed automata enables this kind of test for the specifications defined by means of LSC charts as well. Input-output conformance relation has first been introduced by [Tre99]. Input-output conformance relation for quiescent systems is presented in [BB04b]. Framework for black-box conformance testing of real-time systems is proposed in [KT04]. Among the recent advances in the field is definition of the input-output conformance relations for hybrid systems [vO06].

2.2.1 IOCO conformance relation

The IOCO testing theory reasons about black-box conformance testing of software components. Implementations are treated and specifications modeled as labeled transitions systems (LTS) with inputs and outputs. The *quiescence* notion is introduced which means absence of output at the implementation (no enabled outputs, just inputs).

The (usually infinite) set of *ioco* test-cases allows in theory to distinguish all faulty from *ioco*-correct implementations by executing the test cases.

A labeled transition system (LTS) is a tuple $(S, s_0, Act_\tau, \rightarrow)$ where S is set of states, $s_0 \in S$ initial state, $Act_\tau = Act \cup \tau$ consists of a set of observable actions Act and the distinguished internal (unobservable, silent) action τ, and $\rightarrow \subseteq S \times Act \cup \{\tau\} \times S$ the transition relation. τ is the invisible action. Transitions $(s, a, s\prime) \in \rightarrow$ can also be written $s \xrightarrow{a} s\prime$. Set of all transition systems over label set Act is denoted $\mathcal{L}(Act)$.

The Act consists of input label set Act_{in}, output label set Act_{out} such that $Act_{in} \cap Act_{out} = \emptyset$, $\tau \notin Act_{in} \cup Act_{out}$. Elements from Act_{in} (the inputs) are suffixed with "?" and those from Act_{out} (outputs) with "!".

Transition systems without infinite sequences of transitions with only internal actions are called *strongly converging*. For technical reasons we restrict LTS(Act) to strongly converging transition systems.

A computation is a (finite) sequence of transitions:

$$s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \xrightarrow{\alpha_3} \ldots \xrightarrow{\alpha_{n-1}} s_{n-1} \xrightarrow{\alpha_n} s_n \tag{2.1}$$

A trace σ is a sequence of observable actions of a computation. The finite set of all sequences over a set of actions Act is denoted by Act^* with ϵ denoting the empty sequence.

Some of the notations for LTS are presented in the definition 2.2.1 that will be used further in the thesis:

Definition 2.2.1 *A LTS notation for a LTS $\langle s, s_0, Act, \rightarrow\rangle$ where states $s, s\prime \in S$, state set*

$S\prime \subseteq S$, actions $a_i \in Act$, $\alpha_i \in Act_\tau$, and trace $\sigma \in Act^*$

$$\begin{array}{rcl}
s \xrightarrow{\alpha} s\prime & =_{def} & (s, \alpha, s\prime) \in \rightarrow \\
s \xrightarrow{\sigma} s\prime & =_{def} & \exists s_0 \ldots s_n :\ s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} s_n \ \ \textit{where} \\
 & & s = s_0,\ s\prime = s_n,\ \sigma = (\alpha_1\alpha_2 \ldots \alpha_n) \\
s \xrightarrow{\sigma} & =_{def} & \exists s\prime : s \xrightarrow{\sigma} s\prime \\
s \overset{\epsilon}{\Rightarrow} s\prime & =_{def} & s = s\prime \ \textit{or}\ s \xrightarrow{\tau\tau\ldots\tau} s\prime \\
s \overset{a}{\Rightarrow} s\prime & =_{def} & \exists s_1, s_2 \in S : s \overset{\epsilon}{\Rightarrow} s_1 \overset{a}{\Rightarrow} s_2 \overset{\epsilon}{\Rightarrow} s\prime \\
s \overset{\sigma}{\Rightarrow} s\prime & =_{def} & \exists s_0, s_n :\ s_0 \overset{\alpha_1}{\Rightarrow} s_1 \overset{\alpha_2}{\Rightarrow} \ldots \overset{\alpha_n}{\Rightarrow} s_n \ \ \textit{where} \\
 & & s = s_0,\ s\prime = s_n,\ \sigma = (\alpha_1\alpha_2 \ldots \alpha_n) \\
s \overset{\sigma}{\Rightarrow} & =_{def} & \exists s\prime : s \overset{\sigma}{\Rightarrow} s\prime \\
\boldsymbol{traces(s)} & =_{def} & \{\sigma \in Act^* | s \overset{\sigma}{\Rightarrow}\} \\
\boldsymbol{s\ after}\ \sigma & =_{def} & \{s\prime \in S\prime | s \overset{\sigma}{\Rightarrow} s\prime\}
\end{array}$$

A LTS $L \in \mathcal{L}(Act_{in} \cup Act_{out})$ is called the input/output transition system (IOTS) if it is input enabled, i.e. $\forall s \in S, \forall act_{in} \in Act_{in}.\ \exists s\prime \in S.\ s \xrightarrow{act_{in}} s\prime$. Input-enabledness ensures the deadlock-freeness. Quiescence is different from deadlock and means that there are no enabled outputs or internal actions available, just inputs. Let the quiescence label be $\delta \notin Act_{in} \cup Act_{out} \cup \{\tau\}$. Quiescent state is denoted $\delta(s)$ iff $\forall act \in Act_{out} \cup \{\tau\}.\ \nexists s\prime \in L.s \xrightarrow{act} s\prime$. Quiescent trace is such trace which terminates with quiescence. If δ can appear in a trace at any place, then we have traces with repetitive quiescence where outputs are refused and inputs after such outputs can occur. Such traces are called *suspension traces.*

Definition 2.2.2 *Let $p \in IOTS(Act_{in}, Act_{out})$, a quiescence action $\sigma \notin Act$ and $s \in S$:*

$$\begin{array}{rcl}
s \xrightarrow{\delta} s & =_{def} & \nexists a \in Act_{out} \cup \{\tau\} : s \xrightarrow{a} \\
\mathbf{Straces(s)} & =_{def} & \{\sigma \in (Act \cup \{\delta\})^* | s \overset{\sigma}{\Rightarrow}\} \\
 & & \textit{where}\ \overset{\sigma}{\Rightarrow}\ \textit{includes}\ \delta\ \textit{transitions}\ s \xrightarrow{\delta} s
\end{array}$$

Conformance testing is used for testing the functionality of a system with respect to systems specification. A test is used to define whether an implementation conforms to the specification by performing experiments on the implementation and observing reactions.

A test case is derived from the test specification which depicts the behavior of a tester. The test is then executed on the implementation. A test verdict - *pass* or $fail$ - indicates (non-)conformance of the implementation to the specification.

The aim of the conformance testing is to define the correctness of implementation with the respect to a specification. The behavior of a system can be expressed in term of traces of observable actions. Thus the implementation relation can be expressed through trace preorder between an implementation and a specification.

This relation assumes that the specifications exist as a LTS with distinguished input and output (but not necessarily as an IOTS). The implementation behavior is modeled as an IOTS: *ioco* $\subseteq IOTS(Act_{in}, Act_{out}) \in LTS(Act_{in}, Act_{out})$.

Definition 2.2.3 *Let p be a state in LTS, $Spec, Impl \in \mathcal{L}(Act_{in} \cup Act_{out})$, and implementation $Impl$ be an $IOTS(Act_{in}, Act_{out})$. Then,*

$$\begin{aligned} out(s) &=_{def} \{o \in Act_{out} | s \xrightarrow{o}\} \cup \{\delta | \delta(s)\} \\ out(S) &=_{def} \bigcup \{out(p) | p \in P\} \\ Impl\ \mathbf{ioco}\ Spec &=_{def} \forall \sigma \in \mathbf{Straces}(spec) : out(Impl\ \mathbf{after}\ \sigma) \subseteq out(Spec\ \mathbf{after}\ \sigma) \end{aligned}$$

2.2.2 Test case derivation to check IOCO

From the specification of the system , the test cases can be derived that are sound with respect to *ioco*, namely that their execution will never lead to a test failure if the implementation is *ioco*-correct. The test cases are deterministic, finite, non-cyclic LTS with two special states *pass* and $fail$ which are supposed to be *terminating.* Test cases are denoted in a process-algebraic notation, with the following syntax: $T \longrightarrow pass \mid fail \mid a;T \mid \sum_{i=1}^{n} a_i T_i$ for $a_1, \ldots a_n \in Act_{in} \cup Act_{out} \cup \{\delta\}$. Assuming the LTS $L \in \mathcal{L}(Act_{in} \cup Act_{out})$ as a specification, test cases are defined recursively with finite depth starting with $S = \{L.s_0\}$ according to the definition 2.2.4.

Definition 2.2.4 Test cases definition according to IOCO conformance relation
A test case θ is a LTS $\langle S, s_0, Act_\delta, \rightarrow\rangle$ s.t.

- *θ is deterministic and has finite behavior*
- *S contains the only permitted terminal states **pass** and **fail** s.t.*
  $\forall s \in \{\boldsymbol{pass}, \boldsymbol{fail}\}\ \nexists a \in Act_\delta . s \xrightarrow{a}$
- *Whenever there is an input or output available, the state is not terminal, i.e.*
  $\forall s \in S \setminus \{\boldsymbol{pass}, \boldsymbol{fail}\} \exists a \in Act_\delta.\ s \overset{a}{\Rightarrow}$
- *A test suite Θ is a set of test cases*

2.2.3 TIOCO conformance relation

In *tioco*, conformance of implementation is defined between the specification as the timed labeled transition system (TLTS) and implementation as a timed input-output transition system (TIOTS). TLTS and TIOTS are LTS and IOTS, respectively, with an explicit notion of time and delay steps (the d steps) in addition to $Act \cup \tau$ steps. The *tioco* is an extension of *ioco* with time.

Timed labeled transition system (TLTS) differs from LTS in extended action alphabet with delay action $\delta \in \mathbb{R}_+$.

Definition 2.2.5 Timed labeled transition system
Timed labeled transition system (TLTS) is a tuple $\langle S, s_0, Act_{\tau\delta}, \rightarrow\rangle$ where

- *S is the set of states*

- *$s_0 \in S$ is the initial state*
- *$Act_{\tau\delta}$ is the action set consisting of observable actions Act, silent action $\{\tau\}$ and delay action $\{\delta \in \mathbb{R}_+\}$. Its subset $Act_\delta = Act \cup \{\delta | \delta \in \mathbb{R}_+\}$*
- *$\rightarrow \subseteq S \times Act_{\tau\delta} \times S$ is the transition relation which satisfies the following constraints:*
  - *Time determinism: whenever $s \xrightarrow{\delta} s'$ and $s \xrightarrow{\delta} s''$, $s' = s''$*
  - *Time additivity: $\forall s, s'' \in S$, $\exists s' \in S$ s.t. $s \xrightarrow{\delta_1} s' \xrightarrow{\delta_2} s''$ iff $s \xrightarrow{\delta_1+\delta_2} s''$*
  - *Null delay: $\forall s, s' \in S$, $s \xrightarrow{0} s'$ iff $s = s'$*

The definition of the TLTS is lifted from previous definition 2.2.1 with the following additions:

Definition 2.2.6 *A LTS notation for a TLTS $\langle s, s_0, Act_{\tau\delta}, \rightarrow\rangle$ where states $s, s' \in S$, state set $S' \subseteq S$, actions $a \in Act$, $\alpha \in Act_{\tau\delta}$ and $d \in Act_\delta$*

$$\begin{aligned}
s \xrightarrow{\alpha} s' \quad &=_{def} \quad (s, \alpha, s') \in \rightarrow \\
s \xrightarrow{\alpha} \quad &=_{def} \quad \exists s' :\ s \xrightarrow{\alpha} s' \\
s \xrightarrow{\sigma} s' \quad &=_{def} \quad \exists s_0 \dots s_n :\ s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \dots \xrightarrow{\alpha_n} s_n \quad \textit{where} \\
&\qquad\quad s = s_0,\ s' = s_n,\ \sigma = (\alpha_1 \alpha_2 \dots \alpha_n),\ \alpha_i \in Act_{\tau\delta} \\
s \overset{\delta}{\Rightarrow} s' \quad &=_{def} \quad s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \dots \xrightarrow{\alpha_n} s_n \quad \textit{such that } s_n = s, \\
&\qquad\quad s_0 = s,\ \forall i \in [1, n] : a_i = \tau \vee a_i = \delta_i,\ \textit{and } \delta = \sum_{i | \alpha_i = \delta_i} \delta_i \\
s\ \textbf{\textit{after}}_t\ \delta \quad &=_{def} \quad \{s | s \in S \wedge s' \overset{\delta}{\Rightarrow} s\} \\
S'\ \textbf{\textit{after}}_t\ \delta \quad &=_{def} \quad \{s | s \in S, s' \in S', s' \overset{\delta}{\Rightarrow} s\} \\
\xrightarrow{\tau}{}^{*} \quad &=_{def} \quad \textit{the reflective and transitive closure of } \xrightarrow{\tau} \\
s \overset{\epsilon}{\Rightarrow} s' \quad &=_{def} \quad s = s' \textit{ or } s \xrightarrow{\tau}{}^{*} s' \\
s \overset{a}{\Rightarrow} s' \quad &=_{def} \quad \exists s_1, s_2 \in S : s \overset{\epsilon}{\Rightarrow} s_1 \overset{a}{\Rightarrow} s_2 \overset{\epsilon}{\Rightarrow} s' \\
s \overset{\sigma}{\Rightarrow} s' \quad &=_{def} \quad \exists s_1, s_2, \dots, s_n \in S \textit{ s.t. } s \overset{d_1}{\Rightarrow} s_1 \overset{d_2}{\Rightarrow} s_2 \dots \overset{s_n}{\Rightarrow} s_n \textit{ where } s_n = s', \\
&\qquad\quad \sigma = d_1 d_2 \dots d_n \textit{ and } d_i \in Act_\delta \\
s \overset{\sigma}{\Rightarrow} \quad &=_{def} \quad \exists s' \in S \textit{ s.t. } \overset{\sigma}{\Rightarrow} s' \textit{ where } \sigma \in Act_\delta^* \\
s'\ \textbf{\textit{after}}_t\ \sigma \quad &=_{def} \quad \{s | s \in S \wedge s' \overset{\sigma}{\Rightarrow} s\} \textit{ where } \sigma \in Act_\delta^* \\
S'\ \textbf{\textit{after}}_t\ a \quad &=_{def} \quad \{s | s \in S \wedge \exists s' \in S' \textit{ s.t. } s' \overset{a}{\Rightarrow} s\}
\end{aligned}$$

Informally a timed automaton is an automaton extended with a concept of a clock which defines the timed behavior of the automaton. In addition to components of LTS, the timed automaton includes set of clocks $\mathcal{C}$, that have real non-negative valuations increasing at the same rate in the system. Set of clock valuations can be reset using the set of assignments $R(\mathcal{C})$ over arbitrary subset of $\mathcal{C}$, with the syntax is $c := x$ where $c \in \mathcal{C}$ and x is a non-negative integer. Guards $G(\mathcal{C})$ over set of clocks $\mathcal{C}$ allow specification of timing constraints. Guards are specified by the grammar $g ::= \gamma | g \wedge g$ where clock constraints γ have the syntax $c_1 \sim x$ or $c_1 - c_2 \sim x$ where $c_1, c_2 \in \mathcal{C}$, x is a non-negative integer and $\sim \in \{<, \leq, =, \geq, >\}$.

Definition 2.2.7 (Timed automaton)
A timed automaton T is a tuple $(L, l_0, \mathcal{C}, Act_{\tau\delta}, \rightarrow, I)$ where

- *L is a finite set of locations*
- *$\mathcal{C}$ a set of clock variables*
- *$Act_{\tau\delta}$ a set of labels*
- *$l_0 \in L$ is an initial location*
- *$\rightarrow \subseteq L \times G(\mathcal{C}) \times Act_{\tau\delta} \times 2^{\mathcal{C}} \times L$ is the set of edges*
- *$I: \ L \rightarrow G(\mathcal{C})$ assigns invariants to locations.*

  We define $\mathcal{A}(Act_{\tau\delta})$ to be the set of timed automata over the label set $Act_{\tau\delta}$.

The semantics of a timed automaton T is defined by associating a timed label transition system S_T with T. A state s of a timed automaton is a pair $\langle l, \vec{v} \rangle$ where $l \in L$ is a location and $\vec{v}$ is a valuation of all clocks in $\mathcal{C}$.

A clock valuation is a function $\vec{v} : \mathcal{C} \rightarrow \mathbb{R}^*_{\geq 0}$. For $d \in \mathbb{R}_{\geq 0}$, we define $(\vec{v} + d)(c) = \vec{v}(c) + d$.

The edges of timed automata $(l, g, a, r, l\prime)$ are abbreviated $l \xrightarrow{g,a,r} l\prime$ where $g \in B(\mathcal{C})$ are guards, and $r \in R(\mathcal{C})$ the clock resets; both guards and invariants are clock constraints.

The valuation $\vec{v}$ must always satisfy the invariant constraints in automatons current location l: $\vec{v} \models I(l)$. There are two types of transitions in S_T:

- a δ-delay transition $\langle l, \vec{v} \rangle \xrightarrow{\delta} \langle l, \vec{v} + \delta \rangle$ where $\delta \in \mathbb{R}_{\geq 0}$, and $\forall \delta\prime \leq \delta : \ \vec{v} + \delta \models I(l)$
- an a-action transition $\langle l, \vec{v} \rangle \xrightarrow{a} \langle l\prime, \vec{v}\prime \rangle$ if an edge $l, g, a, r, l\prime \in \rightarrow$ exists such that $\vec{v} \models g$, $\vec{v}\prime = r(\vec{v})$ and $\vec{v}\prime \models I(l\prime)$.

Definition 2.2.8 *Let $p \in S$ be a state in TLTS, specification Spec $\in TLTS(Act_{in} \cup Act_{out} \cup \{\delta\})$ where $\delta \in \mathbb{R}_{\geq 0}$, and implementation Impl be an $TIOTS(Act_{in}, Act_{out})$. Then,*

$$\begin{aligned}
ttraces(p) &=_{def} \{\sigma \in (Act \cup \{\delta\})^* | p \overset{\sigma}{\Rightarrow}\} \\
out(p) &=_{def} \{\alpha \in Act_{out} | p \overset{\alpha}{\Rightarrow}\} \\
out(S) &=_{def} \bigcup \{out(p) | p \in S\} \\
Impl \ \mathbf{tioco} \ Spec &=_{def} \forall \sigma \in ttraces(spec) : out(Impl \ \mathbf{after}_t \ \sigma) \subseteq out(Spec \ \mathbf{after}_t \ \sigma)
\end{aligned}$$

2.2.4 Test case derivation to check TIOCO

The test case derivation rules for TIOCO resembles very much those for IOCO in definition 2.2.4 with several exceptions:

- Act_δ in TIOCO means $Act_{in} \cup Act_{out} \cup \{\delta | \delta \in \mathbb{R}_{\geq}\}$
- The test cases can be infinite in time.

Timed conformance test with discrete clocks is implemented in Timed Test Generator (TTG) [ttg04] over the IF environment [BFG$^+$00].

The tools supporting the TIOCO testing are TorX [BFdV$^+$99] [TB03] and UppAal TRON [MNL03].

2.2.5 TIOCO: example

A coffee brewing machine is taken as an example. The machine receives the coin from the user (the *coin*? synchronization), and upon request from user (the *req*? synchronization), produces the weak coffee (the *weakCoffee*! synchronization) in the interval of one to three time units if the request was made within five time units from inserting the coin, or the strong coffee (the *strongCoffee*! synchronization) within one to five time units from request, if the request has been made three time units or later after inserting coin.

The timed automaton of the coffee brewing machine is presented in figure 2.1. The corresponding TIOTS has inputs *coin*? and *req*?, outputs *weakCoffee*! and *strongCoffee*! and time constraints as specified by the invariants $x \leq 3$ at the location $L2$, $x \leq 5$ at th e location $L3$ and guards elsewhere in the figure.

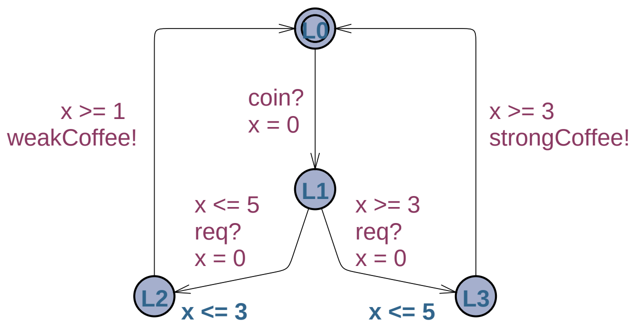

Figure 2.1: The coffee brewing machine model specified by means of timed automaton. Clock x is used to restrain the model staying in locations $L0$ to $L3$ that represent the states of the machine, inputs to it are *coin*? and *req*?, outputs are *weakCoffee*! and *strongCoffee*!, location invariants at the locations $L2$ and $L3$, and guards elsewhere.

Some exemplifying traces and possible steps of the TIOTS after these traces are presented in the table 2.1.

Trace, σ	Out (s $\mathbf{after_t}$ σ)
$coin? \cdot 2$	$\mathbb{R}_{\geq 0}$
$coin? \cdot 4 \cdot req? \cdot 1$	$\{weakCoffee!, strongCoffee!\} \cup [0,4]$
$coin? \cdot 4 \cdot req? \cdot 2$	$\{weakCoffee!, strongCoffee!\} \cup [0,3]$
$coin? \cdot 5 \cdot req? \cdot 3$	$\{strongCoffee!\} \cup [0,2]$
$coin? \cdot 5 \cdot req? \cdot 5$	$\{strongCoffee!, 0\}$

Table 2.1: Exemplary timed traces σ and the sets of possible steps for specification, s After σ.

2.3 Test framework

The practical benefit of this PhD thesis is improvement in transition from informal requirements to test cases. Figure 2.2 demonstrates the improvement from currently used script-based framework to desired test framework capable of maintain the online automatic test. By online it is meant that tests are derived, executed and checked simultaneously while maintaining the connection to the system in real-time.

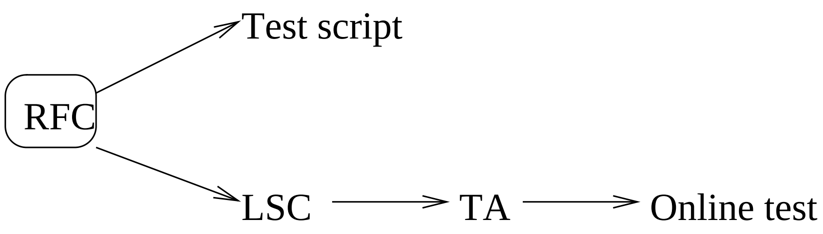

Figure 2.2: Scheme of the current and desired utilization of requirements contained. The upper path *RFC* $\rightarrow$ *Test script* stands for the current activity of extracting the requirements from RFC manually and coding them into the script. The lower path *RFC* $\rightarrow$ *LSC* $\rightarrow$ *TA* $\rightarrow$ *Online test* stands for capturing the requirements in the intermediate format (LSC), which is easily understandable by humans and also translateable to other formalism (TA) that is operated over by the machines. The translated requirements by means of timed automata are then used to generate and run the tests.

Capturing of requirements into he LSC charts are presented in chapter 3 and particularly in section 2.3. The UppAal TA syntax and formal semantics can be found in chapter 4. Translation from LSC to UppAal TA is explained in chapter 5, and performance of the translated specifications in conformance test can be found for two case studies in chapter 7.

Chapter 3

Live Sequence Charts

Live Sequence Charts have been introduced as extension of message sequence charts in [DH01]. Several new concepts and constructs have been added to LSC like multiple instances and symbolic variables [MHK02], time and forbidden elements [HKP04]. Altogether these concepts are presented in [HM03].

Converting LSC into the code has been recently analyzed in [MH06].

In parallel, an automata based semantics of Live Sequence Charts has been introduced [BDK$^+$04]. The derived semantics of LSC mainly bases on the latter source, with exception of owned events (introduced in section 3.8), forbidden elements from [HKP04] and own defined ignored elements by means of scopes (both introduced in section 3.20).

The LSC in this thesis will be introduced in the following order:

- Informal definition of LSCs
- Formal definition (operational semantics). By gradually increasing amount of maintained LSC constructs, the semantics of all supported constructs will be presented.

3.1 Concept of capturing requirements in LSC

To demonstrate the capturing of the requirement in the LSC chart, the RFC requirement is from RFC 2131 [Dro97], section 3.2, bullet 3:

```
... If the client [in state REBOOTING] receives a DHCPNAK message
[as a reply to its DHCPREQUEST message], it cannot reuse its
remembered network address. It must instead  request a new address
by restarting the configuration process, ...
```

The IP address value is the data part of the protocol, and it is decided to leave it out during the current requirement formalization. It will only be focused on the state of the DHCP client and the events (or, messages in real life) the client is receiving or transmitting.

Several things in addition must be taken into account, such as:

- Driving the DHCP client into the state REBOOTING
- Triggering the DHCP client to send the DHCPREQUEST message, or waiting until such a message is sent
- Generating the DHCPNAK message as reply to the client's DHCPREQUEST
- Detecting through explicit means (the configuration start message) that the client indeed has restarted the configuration address

The manually written test script consists of the following phases:

1. Start the DHCP client
2. Drive the DHCP client into the REBOOTING state. For particular protocol, it consists of assigning the client an IP address and triggering the address renewal procedure
3. Wait until the client issues the DHCPREQUEST message, implying that it is in the REBOOTING state
4. Construct and issue the DHCPNAK message as response to the DHCPREQUEST message
5. Wait until the next message from the client
6. Check that the message is DHCPCISCOVER, which is always issued upon (re)start of configuration process

The LSC chart representing that requirement is depicted in figure 3.1. The big dashed hexagon there denotes prechart, this is the phase when the events in the LSC chart are observed. Inside the hexagon there is a message with label *m_nak* and the thin hexagon (the condition) with the rectangle (the assignment) attached to the end of message arrow. Placement of the condition and assignment to the end of message means that the message event is allowed to happen (typically, some shared variables such as the *cstate* in particular setup are assigned certain values upon the message event), and then the condition is evaluated. Assignment attached to the condition is only executed if the condition evaluates to true.

The figure can be read technically as follows: when the *m_nak* event is observed and the shared variable *cstate* has assigned one of the values from the set $\{S\_REQUESTING, S\_RENEWING, S\_REBINDING, S\_REBOOTING\}$, the *cstate* shared variable is set to the value S_INIT.

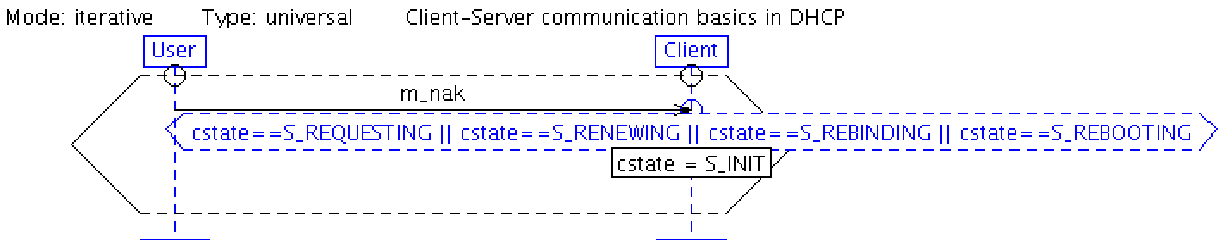
Mode: iterative
Type: universal
Client-Server communication basics in DHCP
User
Client
m_nak
cstate==S_REQUESTING || cstate==S_RENEWING || cstate==S_REBINDING || cstate==S_REBOOTING
cstate = S_INIT

Figure 3.1: Requirement from Section 3.1 captured in Live Sequence Chart. Particular chart aggregates all states of the DHCP client, where the DHCPNAK message drives the client into the INIT state.

The LSC chart focuses only on the client's state change upon receiving the DHCPNAK message while in REBOOTING or behaviorally equivalent state with respect to reacting to the DHCPNAK message.

Differently from the test script, the other phases are performed, or fall under scope, of other LSC charts that constitute the specification. There are several LSC charts describing how to drive the DHCP client into the particular state; they are not displayed here for the sake of brevity.

3.2 Example of the system model in LSC

The system model taken as an example and captured in Live Sequence Charts is the Smart Lamp. It is a timed specification, whose UppAal model with JAVA implementation comes originally with the UppAal TRON tool [LMN04], [LMNS05]. The lamp specification has been reverse-engineered from the simplified UppAal model. The specification has been defined in LSC. It then has been translated into the UppAal network of timed automata by means of the LSC to UppAal translation tool developed during the thesis.

3.2.1 Brief description of the system

The system consists of the user, wire of the lamp, the dimmer and switch processes that turn the lamp on or off and change its brightness, and the lamp bulb which reflects the brightness assigned. System has the input events from user, called *grasp* and *release*, and outputs the brightness of the lamp as an *setlevel* event with brightness level *op* set up.

The user keeps grasping and releasing the wire, and intervals between grasping and releasing are treated respectively by dimmer and switch. The switch can recognize the grasp-release event pair within certain interval as *touch* internal event. So does the dimmer with *starthold* internal event upon another interval, causing non-determinism since the intervals partially overlap. The *starthold* is always followed by *endhold* when the user releases the wire.

The switch toggles the lamp state and brightness upon the *touch* event, while the dimmer periodically updates the lamp brightness between *starthold* and *endhold* events. Both they affect the brightness by an event *setlevel* and discrete value of brightness, *op*.

More detail description of the model by means of requirements for the processes is in section 3.2.2.

The requirements of the system model will be expressed in Live Sequence Charts. It will be done in few steps:

- Processes, or actors, are identified that constitute the system and its environment
- Communication between the processes is decided by means of labeled messages and shared variables
- Variables and constants are defined to maintain the global state of the overall system and to implement its particular requirements
- Requirements expressed as scenarios, split into more simple scenarios if necessary and captured in LSC charts

Identification of actors, communication among them and defining global and local variables with constants is an easy step given the UppAal model. More challenging is stating the requirements based on the existing UppAal model and capturing them in LSC charts in sections 3.2.2 and 3.2.3, respectively.

3.2.1.1 Actors and communication among them

Several actors of the system have been identified based on the existing model. The hardware of the lamp with non-trivial logic includes the triple of wire, dimmer and switch who are activated upon the impact of the user through wire and react with changing the lamp state or brightness. The full list of the processes is:

- User
- Wire
- Switch
- Dimmer
- Lamp

Obviously, the $User$ actor belongs to the environment. So does the *Lamp* since it represents the result of the process interaction inside the machinery of the smart lamp. The rest of the processes constitute the system.

The actors between the environment and the system and inside system communicate via events. Set of events is defined that are directed from $User$ to $Wire$:

- grasp - when the user grasps the wire
- release - when the user releases the wire

Set of events is defined that are directed from $Wire$ to *Dimmer*:

- starthold - when the dim signal comes to the dimmer

Set of events is defined that are directed from $Wire$ to *Switch*:

- touch - the switch is triggered

Event directed from *Switch* to *Lamp* and from *Dimmer* to *Lamp* is defined:

- setlevel - specifies the brightness level of the lamp

It is no difference to the environment (the lamp) whether it is a switch or a dimmer who sets the lamp brightness level. But describing the logic inside the lamp needs such distinction. For that purpose, the *setlevel* events originating from Switch and Dimmer, the boolean variable $from\_dimmer$ is introduced which is always set when the *setlevel* event is triggered by dimmer, and reset otherwise.

Communication scheme among the actors is depicted in figure 3.2.

Set of constants and variables is defined for the model. The constants are as follows:

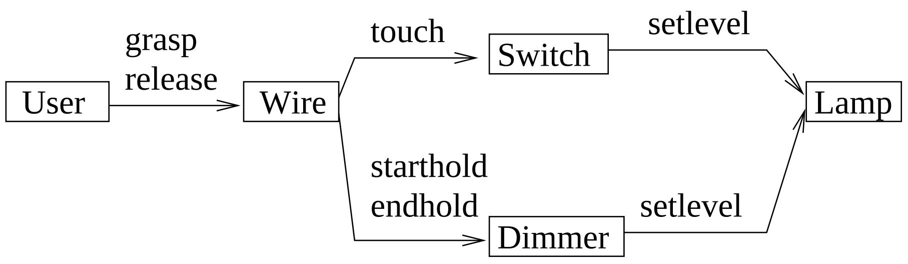

Figure 3.2: Communication among the entities in the smart lamp specification. The instances (processes) are depicted inside boxes, the arrows show direction of communication, and labels above arrows represent events used for particular communication channel.

- MAX_LEVEL = 10 is the maximal discrete level of brightness possible to achieve by the lamp
- eps = 20 is the minimal delay in time units which is recognized as the touch event
- tau = 5 is the time measurements tolerance in time units
- DELTA = 50 is the minimal delay between grasping and releasing the wire, recognized as a starthold event
- T_DIM = 100 is the minimal time interval between two events of lamp dimming

The integer variables are as follows:

int op is a shared variable to transfer the desired lamp brightness value over the synchronization channel

int level stores the lamp brightness level

int oldlevel stores the previous level of the lamp brightness (when lamp gets switched off)

bool dimming flag indicates whether brightness of the lamp increases or decreases with time

bool from_dimmer indicates whether the *setlevel* event has been triggered by Dimmer

bool on flag indicates whether lamp is on or off

3.2.2 Requirements

1. The user can *grasp* and then *release* the wire
2. The *grasp* - *release* event sequence with time interval $[eps, DELTA + tau]$ triggers the *touch* event within *tau* time units
3. The *grasp* event triggers the *starthold* event within time interval $[DELTA, DELTA + tau]$ if there has not been events *touch* or *release* observed meanwhile

4. The event sequence *grasp* - *release*, in which the *grasp* event has triggered the *starthold* event, must be followed by the *endhold* event within *tau* time units from the *release* event

5. The lamp brightness level is changed in the interval $[T\_DIM, T\_DIM + eps]$ after the last change of brightness, if there is no interrupting event observed meanwhile

6. The lamp turns off or on within the *tau* time units from the *touch* event. The brightness of the lamp and its state (*on*) changes upon every touch, between their values *oldvalue* and 1, and 0 and 0, respectively.

7. Lamp dimming always obtains different derivative than in previous dimming

8. The brightness derivative of the lamp changes when the brightness reaches maximal or minimal values, also every time the user starts holding the wire

9. The lamp changes the brightness level within *tau* time units after its new level is set by Dimmer

10. The lamp changes the brightness level within *tau* time units after its new level is set by Switch

3.2.3 Requirements captured as LSC scenarios

First off, the way of user affecting the system (the wire) is described in figure 3.3. It is reasonable to assume that the user first grasps the wire, and then releases it until next grasp. The input to the system will be a sequence of *grasp* - *release* events, and precisely such sequence of events is captured in the LSC chart.

Every LSC chart has *type* and *mode* attributes, that are explained in section 3.5. For particular chart, its type is universal, its mode is invariant and the chart has no prechart (satisfaction of the LSC charts without precharts can be found in section 3.6). The universal type and absence of prechart means that the copies of the chart can activate any time unless restricted elsewhere (in other charts). Particular copy of the chart activates by generating or observing the *grasp* event, and deactivates by generating or observing the next event which is *release*. Generation and observation of the mesage events in the LSC chart is explained in section 3.8.

Invariant mode means that the copies of the chart can activate any time if no violation occurs. The exemplary sequence of events generated by such chart could be $((grasp)^*(release))^*$ where each new copy produces *grasp* when activating and at the same time makes the other copies, who have already produced their *grasp* event, deactivate, since the event sequence *grasp*-*grasp* is not accepted by the specified event *preorder grasp*-*release* in the copies.

There is a solid instance line between the mesage events *grasp* and *release*, which means the *hot cut* when the *grasp* event has taken place. Definition of the cuts is presented in section 3.4.0.2, and temperature of the LSC constructs such as cuts has special meaning during the LSC chart execution, which is presented in section 3.20.1. Thus, no external event like *grasp* from another copy of chart, can occur before *release* event is progressed over in the original copy. This restricts the valid words, generated or accepted by the LSC chart copies, precisely to $((grasp)(release))^*$.

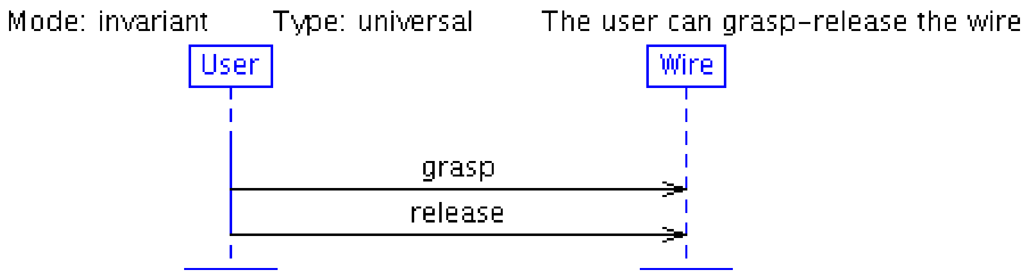

Figure 3.3: The requirement 1 *"The user can grasp and release the wire"* captured in LSC. Due to the chart type and mode and absence of the prechart, the only sequence of the events allowed is $((grasp)(release))^*$.

The system reaction to the user input now needs to be specified, like in figure 3.4. Particular situation is taken where the *grasp* event triggers the *starthold* event inside the system. From requirements the prerequisites to trigger the *starthold* are suitable time window from the *grasp* and no interference from parallel requirements, namely *touch* event originating in $Wire$ and used to trigger the *Switch* instead. There is also another sequence of events that prevent the *starthold* from happening, namely premature occurrence of *release* event, which might occur so soon after the *grasp* that it triggers neither *starthold* nor *touch* events.

Universal chart of iterative mode is chosen with prechart. The *grasp* event is in the prechart, meaning that the event must be observed in other charts and not generated in this chart. The message event is coupled with the assignment which resets the local clock t.

Main chart follows right away, suggesting that now the *starthold* event should be generated under corresponding time window, as specified by the assertion
$t >= DELTA \&\& t <= DELTA + tau$ in the attached condition. The red color of condition means *hot temperature* of the condition, thus any occurrence of the *starthold* event outside the specified time window (i.e. violation of condition) causes the violation of LSC.

Below the chart, the *forbidden scope* (rectangle with the word $Forbidden$ in the left top corner) is located with two messages inside. Concept of scopes is introduced in section 3.20. These message are events that terminate the chart prematurely. There are a couple of *cuts* (the dashed line segments surrounding the *starthold* message with condition) in the main chart with the same pattern as the rectangle of the scope. Whenever the next event to happen is between such cuts, the scope restrictions apply to the chart. In particular case, it is the *starthold* event in the mainchart. Since all the instance lines before the *starthold* event are not solid, this means cold temperature of the cut where the *starthold* is allowed to occur, so the observed events of forbidden scope would rather prematurely terminate the chart instead of causing its violation. Ordering of the messages in the scope does not matter, the scope is rather a container for the restricted messages.

Obviously, all *starthold* events must be followed by *endhold* after the user releases the wire, and that is captured in figure 3.5. The scenario is straight-forward, the *grasp* - *starthold* - *release* event sequence must be observed, that together are the prerequisite for *endhold* event to occur.

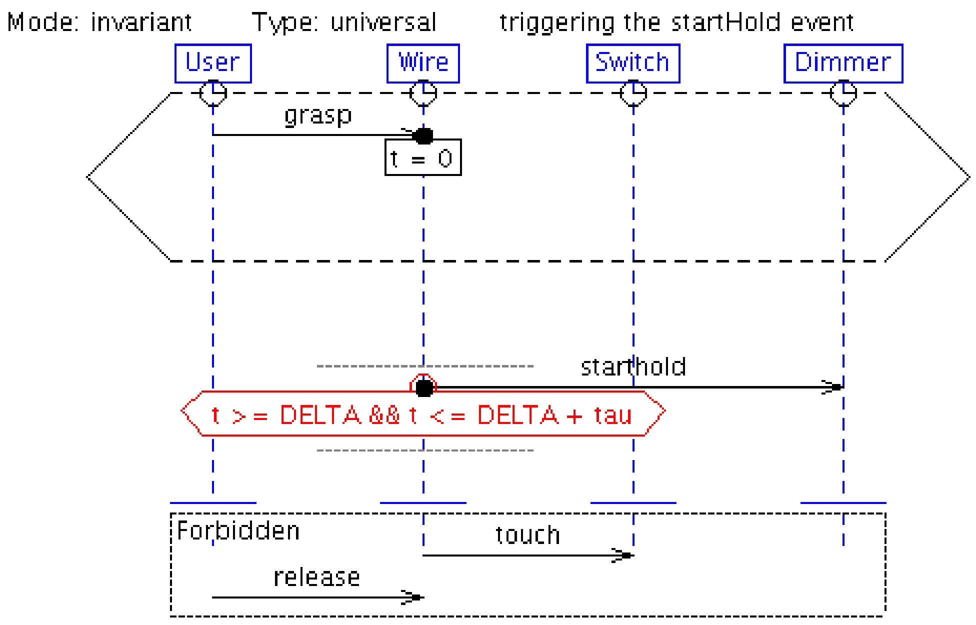

Figure 3.4: Requirement 3, situation captured when the *starthold* event should occur, captured in LSC. Besides the *grasp* event and sufficient time spent while user keeps the wire, the violating events like *release* and *touch* (as alternative to *starthold*) must also be considered. The clock restrictions constrain the occurrence of the event in time.

Universal chart of invariant mode with prechart is chosen to capture the requirement. In the prechart, sequence of events reside (*grasp*, *starthold*, *release*) that must be matched before the chart being allowed to proceed into the mainchart and trigger the *endhold* event. The time restriction is imposed on the time interval between the *release* and *endhold* events, namely resetting the local clock $t = 0$ on observing the *release* event and attaching hot condition with upper bound on that clock $t < tau$ to the *endhold* event. As in previous examples, the hot condition is satisfied if the event is observed or triggered within specified time bounds or causes the chart violation otherwise.

There is a solid instance line between the *release* and *endhold* which makes the non-enabled events like *starthold* or *release* violate the chart in the cut, where the *endhold* event is expected to happen. Note that *grasp* event is ignored in that cut. This is in order to avoid the chart restraining the occurrence of the next *grasp* event in corresponding chart. The charts describing interactions inside the system are not supposed to put constraints on the system environment; it is up to the system architect to define restrictions on the environment separately from requirements on the system.

When the requirement about the sequence of events triggering *endhold* is defined, the change of the system state upon *endhold* must be specified as in figure 3.6. In particular, the *oldlevel* must store the achieved brightness of the lamp, so that it is preserved, say upon switching the lamp off and on again.

The message event with assignment is placed in the prechart of universal chart. It means that the *endhold* event must be observed whenever it happens, and the corresponding function in the assignment attached to the event must be executed.

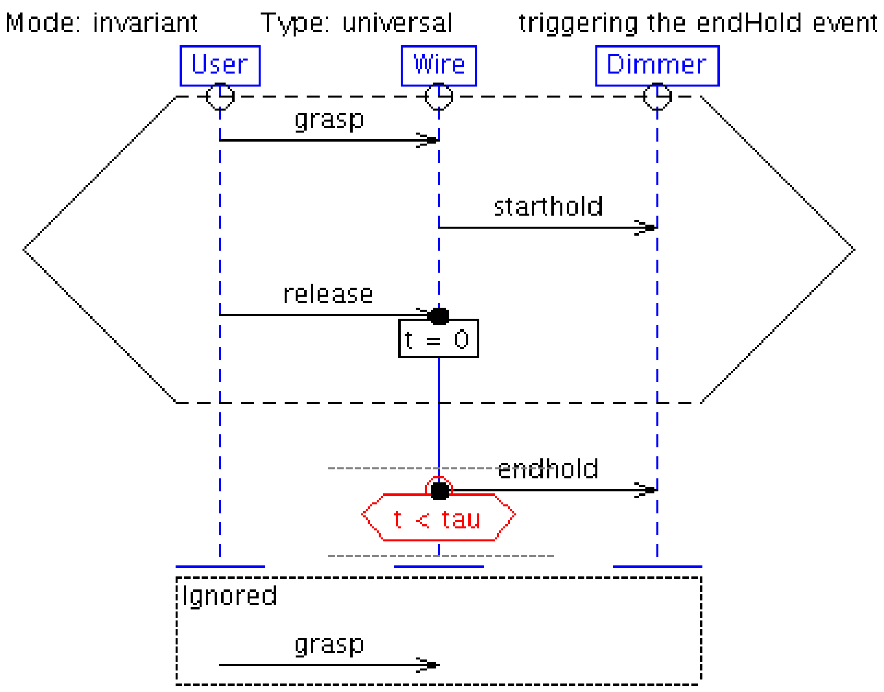

Figure 3.5: Requirement 4, situation when the *endhold* event can take place, captured in LSC. Obviously, the *starthold* event must have happened before and the user must have released the wire afterwards. The clock restrictions constrain the occurrence of the event in time.

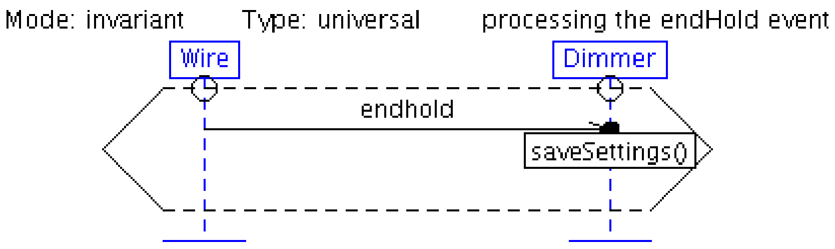

Figure 3.6: Part of requirement 6, updates of the *oldlevel* variable described, when the *endhold* event takes place. The level of brightness is then stored in *oldlevel* in order to reuse it when the user later starts holding or touches the wire.

The function *saveSettings*() in LSC from figure 3.6 consists of the following code:

```
void saveSettings() {
  oldlevel = level;
}
```

The reaction of the system to the *grasp-release* sequence from user, which is alternative to *starthold*, is depicted in figure 3.7. The scenario is similar to that triggering the *endhold* event, just the activating events sequence is *grasp* - *release* with correct timing, and no events like *starthold* interfering.

Attention should be paid to the resetting of the local clock t upon observing both *grasp* and *release* events. Before resetting the clock at the *release* event, its value is checked in the cold condition. Does its value appear outside the interval specified in condition, the cold violation occurs and the chart is reset.

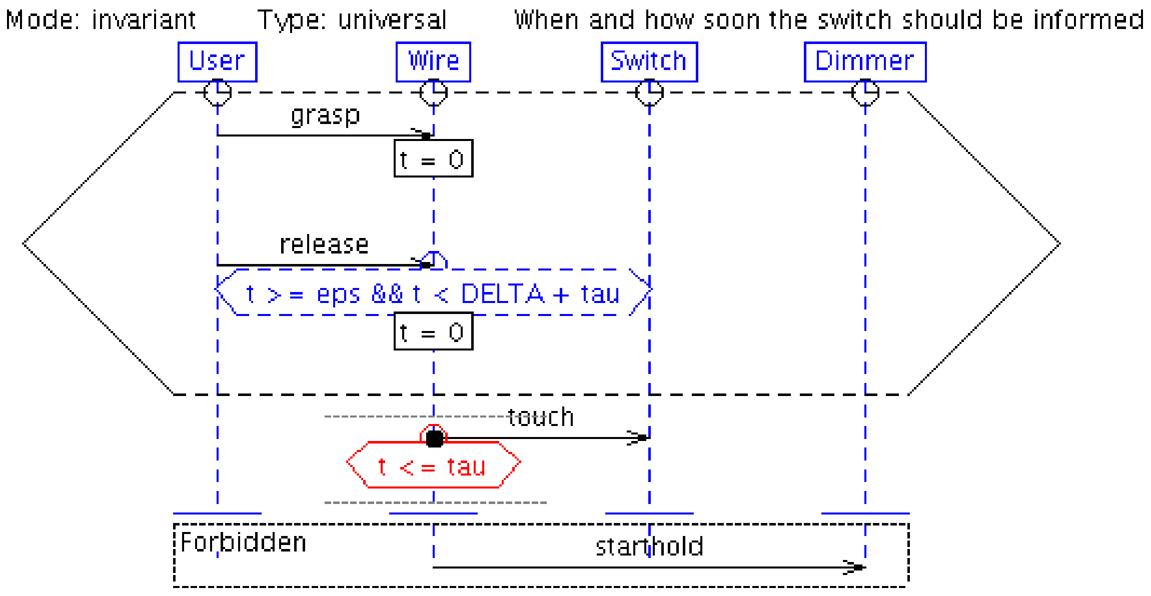

Figure 3.7: Requirement 2, situation when the *touch* event can occur, captured in LSC. Prerequisites for that is the user grasping and releasing the wire within a short time span, and no *starthold* event being triggered by the user bahavior. The local clock t is reused for measuring the time interval between *grasp* and *release*, and also between *release* and *touch*.

Scenario, how the system reacts to the *starthold* event, is described in figure 3.8. The previously saved lamp brightness must be revoked within the specified time interval (*eps* time units). Note that there is no restriction regarding the *endhold* event. This means that once the *starthold* event has been triggered, *regainBrightness*() function is executed regardless of the occurrence of *endhold* in a mean time.

The function *regainBrightness*() in LSC from figure 3.8 consists of the following code:

```
void regainBrightness() {
  from_dimmer = 1;
  level = oldlevel;
  dimming = !dimming;
  on = 1;
  op = level;
}
```

Requirements specifying how the dimmer keeps proceeding until the user releases the wire, is captured in figure 3.9. The brightness changing function is specified, and it is regularly executed. The LSC chart is of invariant mode, and progressing along one copy of the chart does not violate the preorder in its other optionally active copies, what means several copies can be active at a time, even when some of them are in their mainchart. The chart with identic event in the prechart and mainchart is auto-generating, because one copy performs an event upon terminating, which activates another copy.

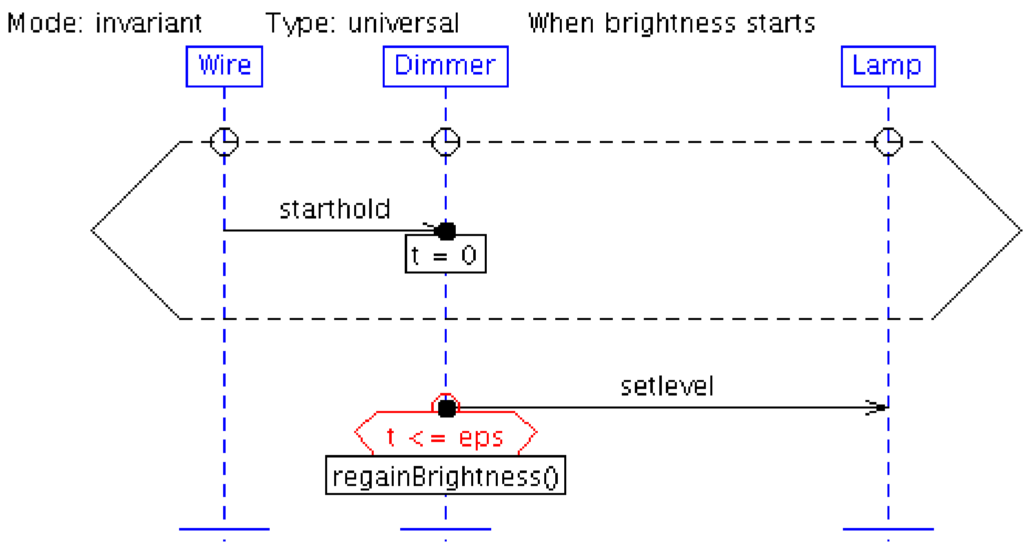

Figure 3.8: Requirement 7, behavior of the dimmer after the user starts holding the wire. It must set the last saved brightness of the lamp within specified time interval. The chart activates another one in figure 3.9, which keeps changing the lamp brightness until the user releases the wire.

The *setlevel* event from dimmer is recognized in prechart by checking the global variable $from\_dimmer$. The *setlevel* event in mainchart has also set the $from\_dimmer$ variable. One thing that can stop the sequence of autogenerating *setlevel* events is the *endhold* event. The *touch* event could also stop the autogeneration, even if it is unlikely to occur before *endhold*. Another event which can stop autogeneration is *setlevel* from the switch. All these cases are mentioned in forbidden scopes below the charts and apply to the *setlevel* event in mainchart.

In the chart there are two kinds of simregions, one with the condition attached to the start of message, another to the end. In case of hot conditions, they make no difference. In case of cold condition, only the simregion with the condition at the end of the message allows the message event to happen regardless of the evaluation of condition. Condition being at the start of the message, and being evaluated to false upon observation of such message event causes violation independently from its temperature.

The function *stepBrightness*() performs book-keeping of the brightness derivative, last brightness level, and sets up the shared variable *op* with the intended brightness of the lamp:

```
void stepBrightness()
{
  if (level >= MAX_LEVEL && !dimming) dimming = 1;
  if (level <= 0 && dimming) dimming = 0;

  if (dimming) level -= 1;
  else level += 1;

  from_dimmer = 1;
  op = level;
```

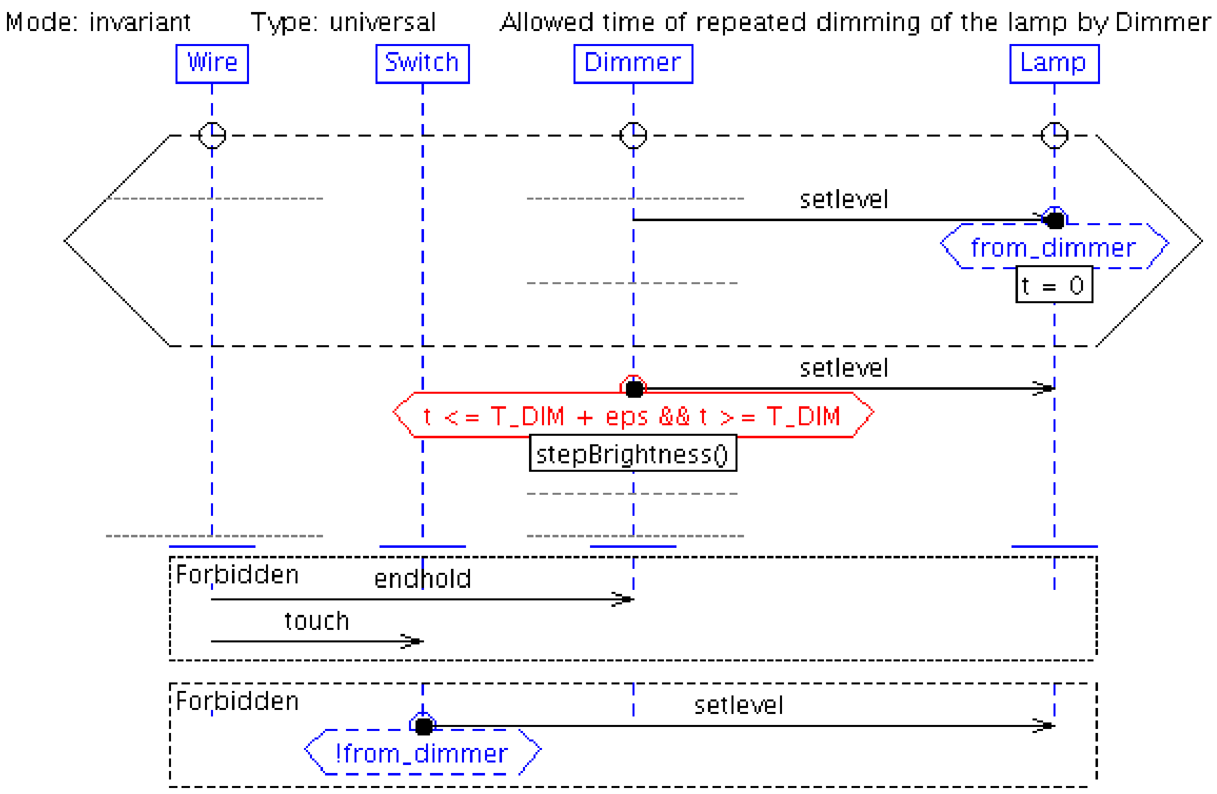

Figure 3.9: Requirements 5 and 8, describing behavior of the dimmer, captured in LSC chart. The chart is autogenerating, i.e. once activated, it continues activating its new copies by means of advancing in current copy. This is an example of how a LSC chart describes periodic action. Only *setlevel* events are accounted that occur upon the variable *from_dimmer* being set, otherwise the chart exits right after evaluating minimal event or when its second *setlevel* event falls under the forbidden scope which has restriction !*from_dimmer*.

```
}
```

Functioning of the switch is captured in LSC, shown in figure 3.10. Every time the *touch* event is observed, the lamp level is accordingly updated and the previous brightness together with lamp state saved or loaded. The variable *from_dimmer* is reset in the function *toggleLamp*().

The function *toggleLamp*() is called when the user touches the wire. The lamp state (on / off) and brightness (last stored / zero) is updated depending on whether the lamp was on or off before touching the wire:

```
void toggleLamp()
{
from_dimmer = 0;
  if (!on)
  {
level = oldlevel;
op = oldlevel;
```

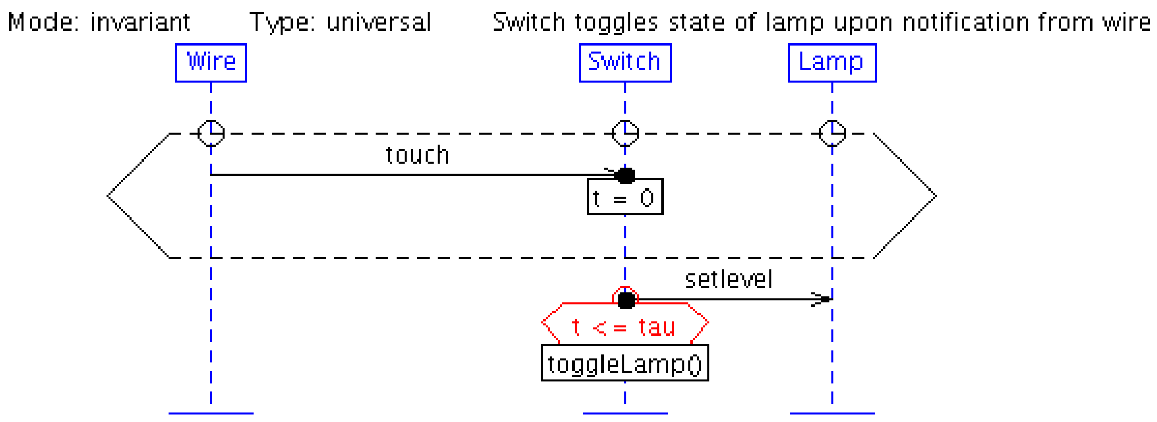

Figure 3.10: Requirement 6, behavior of the switch, captured in LSC. It must switch the lamp within specified time interval, once it is actuated.

```
on = 1;
  }
  else
  {
oldlevel = level;
op = 0;
on = 0;
  }
}
```

Last but not least, the effect of the switch and dimmer on the lamp is described in LSC charts from figures 3.11 and 3.12. These are identic from the LSC point of view, since the only thing that differs in them is the source of the *setlevel* event which is ignored in LSC. These two charts are duplicates from the LSC semantics point of view, assigning the same variables with same value twice. Although there is no side effect in particular case, such practice should be avoided, as it can easily become the case (imagine increasing some variable instead of assigning the value as here).

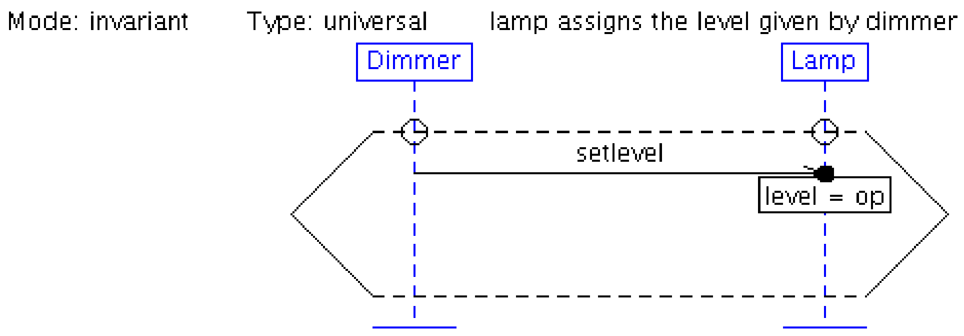

Figure 3.11: Requirement 9, describing how the level of the lamp brightness is affected by the dimmer. A shared variable *op* is used to pass the desired value of brightness, while the variable is initialized in precondition with the function *stepBrightness*() in LSC from figure 3.9.

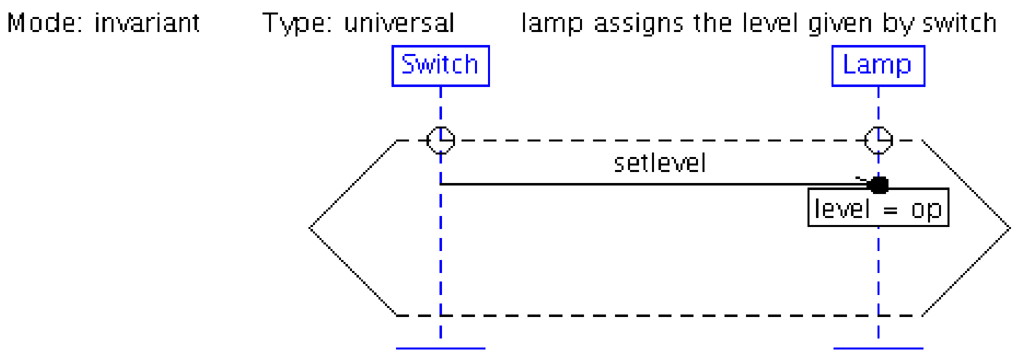

Figure 3.12: Requirement 10, describing how the level of the lamp brightness is affected by the touch. A shared variable *op* is used to pass the desired value of brightness, while the variable is initialized in precondition with the function *toggleLamp*() in LSC from figure 3.10.

Requirement	Capturing chart
1	3.3
2	3.7
3	3.4
4	3.5
5	3.9
6	3.10, 3.6
7	3.8
8	3.9
9	3.11
10	3.12

Table 3.1: Requirements from section 3.2.2 and corresponding figures capturing them.

Table 3.1 shows what requirements are captured by particular LSC charts.

3.2.3.1 Capturing the specification in LSC - comments

Messages have been the inseparable part of LSC. They have been used as the communication events among the processes that constitute the system and its environment.

Conditions and assignments are vital in storing the system state and passing the data between processes.

The LSC constructs are ordered along the instance lines of the LSC charts. Because of such ordering, there are sets of events that are *enabled* (the ones to happen next while respecting preorder) and *non-enabled* ones that would violate the event preorder in LSC if happening while LSC chart is in particular cut. In order to add and subtract the events from these sets according to own needs, the forbidden and ignored scopes are necessary.

In [HM03], only forbidden scopes are mentioned, but forbidden scopes only are not enough in our examples and specifications. It can be seen in figure 3.5 that ignored scope ads more flexibility in the situations where preorder among the LSC events would be too restrictive.

We have also found simregions in the scopes necessary, when the condition is coupled with the message. Figure 3.9 illustrates how the simregion in the forbidden scope combines the instance abstraction and forbidden event. Simregions in scopes allow more precise identification what events and in what situations the events of the scopes apply to the chart.

Temperature of the cuts, conditions, scopes and events in the scopes is another necessary attribute allowing flexible manipulation of various situations occurring in the chart. These can be specified as violation of requirements or just relaxing particular scenario from completing according to predefined pattern. In [HM03], the temperature is used for cuts, conditions and scopes. Temperature of events in the scopes however is not addressed there.

Precharts, type and mode of the charts have been necessary to specify the if-then situations when upon matching one pattern of actions, another pattern is forced to complete.

3.3 Formal semantics of Live Sequence Charts

To understand the formal semantics of the Live Sequence Charts is a non-trivial task which requires some time, examples and step-by-step introduction of various LSC constructs, like it is successfully done in [HM03]. This especially applies to the flavour of LSC presented in the thesis, since it has some unique constructs and subtleties to introduce.

To keep the formal semantics presentation easy and understandable, the rest of the chapter is split into three parts:

1. Message-only subset of LSC is taken, together with instance line an prechart constructs. These three kinds of constructs constitute the very basis of the LSC chart and its semantics. Preorder relation, upon which the progress of the LSC chart is based, can be found in section 3.4.0.2.

   Two properties of the LSC chart that distinct it from the MSC chart and solves one of the MSC shortcomings, are its type and mode (explained in section 3.5). Besides them, additional property, the chart role, is presented, which specifies whether the LSC chart is a part of specification or performs a function of an observer automaton, whose purpose is to determine whether the system proves or disproves the property implied by the scenarios of that chart. All these three properties of the LSC chart are presented in section 3.5.

   Satisfaction of the LSC chart disregarding its role is presented in section 3.6. This is the simplest example how the type and mode of the LSC chart affect its satisfaction.

   Type and role of the LSC chart altogether influence, when event specified in the chart must be observed, and when it can be triggered on behalf of that chart. Event generation and matching is an important feature which allows development of the system model which is not only re-active with respect to its environment but also able to start the communication with each other or environment. Event generation and matching can be found in section 3.8.

   When a set of LSC charts with type, mode and role properties each constitute the model, the satisfaction of the overall model remains simple and is based on the occurrence of events and abstracts from the event generation or matching in particular chart. The satisfaction relation of the whole model against the sequence of events is presented in section 3.9.

2. The message-only semantics is extended with the discrete variables and clocks, as in section 3.10. The visual counterparts of these data structs, the conditions and assignments, forming atomic structures called simregions, are introduced in section 3.11. The state of the LSC chart, called cut, is similar to the configuration of the finite state machine, is introduced in section 3.12, as simply to keep count of the passed messages is not enough when discrete variables and clocks are involved. Prechart and mainchart of the LSC chart are explicitly defined in section 3.13, and transition from prechart to mainchart described. Operational semantics of the LSC chart and set of LSC charts are revisited in sections 3.14 through 3.17, this time involving the variables with clocks and restrictions on them.

3. Temperature of the LSC properties and visual constructs affects the violation of the preorder or dissatisfaction of the conditions. Detail description of the temperature property

and its interpretation in certain situations can be found in section 3.18.

Subchart types, that are loop, if-then-else construct and the simple subchart, influence (restrict or relax) the preorder of constructs in the LSC chart. They are presented in section 3.19.

Scopes, more precisely - ignored and forbidden scopes, are used to increase or decrease the set of non-enabled and ignorable message events in some cuts. Semantics of the scopes and examples of their usage is presented in section 3.20.

Finally, the LSC subchart semantics needs to have the temperature, scopes and subcharts extensions applied. This is done in section 3.20.1. Satisfaction of the LSC specification consisting of set of LSC charts remains unchanged at its level.

3.4 Preorder-based semantics of LSC

To abstract from numerous LSC constructs and features, basic traits of the LSC charts and specifications are introduced and exemplified. The LSC chart in its strongest abstraction is stripped of all its constructs, where the remaining visual constructs are messages, precharts and instances, and the chart properties such as type, mode and role are taken into account.

The instance lines are mentioned solely because the messages of the LSC chart originate and terminate on the instance lines. It is the ordering of messages along the instance lines which is important. The messages create the ordering among themselves by originating or terminating on the same instance line where others do. Thus, the LSC chart in its strongest abstraction can be treated as the labeled preorder, the labels being same as the message labels.

3.4.0.2 Labeled partial order (LPO)

A (finite) Σ-labeled LPO $\mathcal{L}$ is a tuple $\langle L, \leq, \lambda, \Sigma \rangle$ where

- L is a finite set of locations
- $\leq \subseteq L \times L$ is a partial order relation on L. It is reflexive ($l \leq l$), anti-symmetric ($l \leq l\prime \wedge l\prime \leq l \Rightarrow l = l\prime$), and transitive ($l \leq l\prime \wedge l\prime \leq l\prime\prime \Rightarrow l \leq l\prime\prime$) relation.
- $\lambda : L \rightarrow \Sigma$ is the labeling function.

Elements of the tuple $\mathcal{L}$ will be referred through dot, for example locations of $\mathcal{L}$ will be denoted as $\mathcal{L}.L$.

A system's behavior is a set of all its executions, that are sequences of events. Partial order is related to sequences of events via linearization.

A linearization of a partial order $\mathcal{L} = \langle L, \leq, \lambda \rangle$ is a word $u = e_0, \ldots, e_n \in \Sigma^*$ such that the LPO $\langle L_u, \leq_u, \lambda_u \rangle$ is isomorphic to $\langle L, \leq_u, \lambda \rangle$ with $\leq \subseteq \leq_u$ where

- the locations are indices in u, $L_u = \{0, \ldots, n\}$,
- the total ordering is ordering of natural numbers: $i \leq_u j \Leftrightarrow i < j$ for $0 \leq i, j \leq n$.
- the labeling function maps each index to that symbol at that position, $\lambda_u(i) = e_i$.

A LPO is *empty* if its location set is empty set.

Subset of locations in the LPO $\mathcal{L} = \langle L, \leq, \lambda \rangle$ that are not preceded by other locations from same LPO with respect to the relation $\leq$ will be called the *activating* locations and denoted $\mathit{activating}(L)$:

$$l \in \mathit{activating}(L) \underset{defn}{\Longleftrightarrow} \forall l\prime \in L.\ l\prime \leq l \Rightarrow l = l\prime \tag{3.1}$$

A finite or infinite word $\gamma \in \Sigma^\infty$ satisfies an LPO $\mathcal{L} = \langle L, \leq, \lambda, \Sigma \rangle$, denote by $\gamma \vDash \mathcal{L}$ iff

- $\gamma \in \Sigma^*$ and γ linearizes $\mathcal{L}$
- $\gamma \in \Sigma^w$ and $\exists w \in \Sigma^*.\ \ w \sqsubset \gamma$ and $w \models \mathcal{L}$

Behavior of a LPO, i.e. its language, is its set of linearizations.

There are two forms of LSC: existential LSC and universal LSC.

- A Universal LSC chart is a couple
$$\Box(P, M) \tag{3.2}$$
where P and M are Σ-LPOs. P is called *prechart* and non-empty M is called *mainchart*.

- An existential LSC chart is
$$\diamond\, M \tag{3.3}$$
where M is a non-empty Σ-LPO.

A cut c of the LSC chart represented by the preorder $\mathcal{L}$ is a downwards-closed subset of locations, i.e. $c \subseteq L$ s.t.:
$$\forall l \in c,\ \forall l' \in \{L/c\}.\ \ l \leq l' \tag{3.4}$$

Example of the labeled preorder

An example is given of a $\{a, b, c, d\}$-LPO with locations $L = \{l_1, l_2, l_3, l_4\}$, preordered pairs of locations $\{l_1, l_2\}, \{l_1, l_3\}, \{l_3, l_4\}, \{l_2, l_4\}$, and labeling function λ which labels l_1 with a, l_2 with b, l_3 with c and l_4 with d.

$$\begin{tikzcd}
 & l_1 : a \arrow[dl, "\leq"'] \arrow[dr, "\leq"] & \\
l_2 : b \arrow[dr, "\leq"] & & l_3 : c \arrow[dl, "\leq"'] \\
 & l_4 : d &
\end{tikzcd}$$

Figure 3.13: A sample $\{a, b, c, d\}$-LPO. Location $l_1 \ldots l_4$ labels follow after the colon, i.e. $l_1 : a$ means $\lambda(l_1) = a$.

Linearizations of the sample LPO are $\{abcd, acbd\}$.

Activating location of the LPO locations is $activating(L) = \{l_1\}$.

Assuming the LSC chart represented by the example LPO, its possible cuts are $\{\emptyset, \{a\}, \{a, b\}, \{a, c\}, \{a, b, c\}, \{a, b, c, d\}\}$.

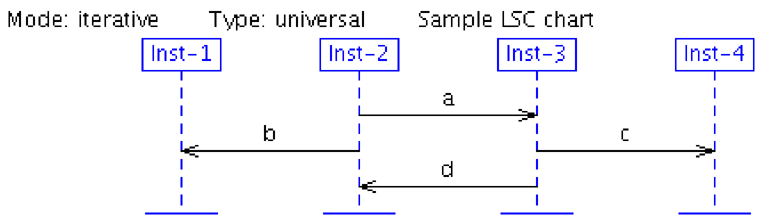

Figure 3.14: Sample LSC chart with instances and labeled messages.

Example of the LSC chart as the labeled preorder

Sample LSC chart *ch* is given in figure 3.14.

The syntax of the chart in figure 3.14 is as follows:

$Insts(ch)$ are the instance lines of the chart ch, labeled $Inst-1$ through $Inst-4$;

$Locs(i, ch)$ is the set of locations on the instance i of the chart ch. Each instance line has at least two locations: one right below the instance head and another at the bottom of the instance line. Overall set of locations in the chart is denoted as $Locs(ch) = \bigcup_i Locs(i, ch)$ for all instances i of the chart ch. Locations are placed on the instance line and identified by their non-negative position $pos : Locs \to \mathbb{N} \cup 0$.

$Msgs(ch)$ are defined in the chart ch, with labeling function $label : Msgs \to Labels$ that maps each message to the global set of labels $Labels$. The messages also have functions that map them to the origination and termination location $orig : Msgs(ch) \to Locs(ch)$ and $term : Msgs(ch) \to Locs(ch)$ for the chart ch.

$\leq$ is the preorder relation over the chart locations. It is defined as follows:

- Locations from the same instance line are ordered according to their position on the instance line:
$$\forall l, l\prime \in Locs(i, ch).\ pos(l) < pos(l\prime) \Rightarrow l \leq l\prime \tag{3.5}$$
- Locations at the originating and terminating point of the message are equal in order:
$$\forall l, l\prime \in Locs(ch), \forall m \in Msgs(ch).\ l = orig(m) \land l\prime = term(m) \Rightarrow l \leq l\prime \land l\prime \leq l \tag{3.6}$$
- Preorder relation $\leq$ is transitive:
$$\forall l, l\prime, l\prime\prime \in Locs(ch).\ l \leq l\prime \land l\prime \leq l\prime\prime \Rightarrow l \leq l\prime\prime \tag{3.7}$$

$\leq$ preorder relation is lifted from locations to the messages:

$$\begin{aligned}\forall m, m\prime \in Msgs(ch).\ & orig(m) \leq orig(m\prime) \lor orig(m) \leq term(m\prime) \\ & \lor term(m) \leq orig(m\prime) \lor term(m) \leq term(m\prime) \Rightarrow m \leq m\prime\end{aligned} \tag{3.8}$$

Locations of the LSC chart are as follows:

- l_{1t}, l_{11}, l_{1b} belong to the instance $Inst - 1$ where l_{1t} is the top location with position 0, l_{11} the middle location where message b terminates, and l_{1b} the bottom location of the instance.
- $l_{2t}, l_{21}, l_{22}, l_{23}, l_{2b}$ belong to the instance $Inst - 2$ where at l_{21} message a originates, at l_{22} the message b originates and at l_{23} the message d terminates.
- $l_{3t}, l_{31}, l_{32}, l_{33}, l_{3b}$ belong to the instance $Inst - 3$ where at l_{31} message a terminates, at l_{32} the message c originates and at l_{33} the message d originates.
- l_{4t}, l_{41}, l_{4b} belong to the instance $Inst - 4$ where at l_{41} the message c terminates.

According to the syntax presented, the LSC chart in figure 3.14 can be treated as the LPO from the figure 3.13.

3.5 Type, mode and role of the LSC chart

Every LSC chart has the three attributes associated with it:

Type is universal or existential. Charts of existential type are used to observe whether the sequence of events exposes specific pattern (word) under certain circumstances. More details about when the pattern should be exposed, depends on the mode of the existential chart.

Universal chart, in contrary, puts restrictions on the sequence of events. It matches the activating pattern with its prechart, just like the existential chart, and if matched, requires that another pattern is matched afterwards. Upon observing mismatching pattern in its mainchart, the universal chart *violates*. Details of matching prechart and mainchart also depend on the mode of the universal chart.

Mode of LSC chart is one of initial, iterative or invariant.

Charts of the initial mode match the pattern until the first activation followed by mismatch or successful completion of match.

Charts of iterative mode and invariant mode have the same semantics if these are of existential type. The sequence of events is always matched against the pattern until a match is detected.

Universal type, iterative or invariant mode charts keep matching their prechart from any occurrence of minimal event. Once the prechart is matched for iterative universal chart, prechart matching is stopped until mainchart is matched (or violated).

Invariant mode universal type chart always keeps matching prechart, and mainchart when corresponding prechart has been matched.

Role is specification or property. This attribute is not defined in any known flavor of LSC semantics, it is introduced by us in order to specify the purpose of the LSC chart. The purpose of the LSC chart can be briefly described as part of executeable specification or as a property to match the sequence of events against. The charts with the role of property match the sequence of events against the pattern defined by their preorder relation. The charts with the role of specification are allowed themselves to generate events, it is described in more details in section 3.8. There is certain operational semantics of the specification role charts (also in section 3.8).

3.6 Satisfaction relation for the LSC chart

Satisfaction relation of the LSC chart depends on the chart type and mode. The role of the chart is disregarded in this section, and it is assumed that the LSC charts are used solely for the observation and matching of events.

Satisfaction of the universal chart based on its mode is as follows:

- Chart of initial mode is satisfied if the trace linearizes it the very fist time when its prechart is satisfied.
- Chart of iterative mode is satisfied when its mainchart is satisfied every time after its prechart satisfied, but the new attempt of prechart matching does not take place until the previous entered mainchart have been linearized or violated.
- Chart of invariant mode is satisfied iff the trace linearizes its mainchart every time after linearizing its prechart; several attempts of matching prechart or mainchart can be performed at a time.

Y. Bontemps in his PhD thesis [Bon05] has given definition for the satisfaction relation of existential type chart $\diamond(M)$ consisting of the LPO M, and universal type, invariant mode chart $\Box(P, M)$ consisting of the LPO P as prechart, and LPO M as mainchart:

- $\gamma \vDash \diamond(M)$ iff it is eventually matched in γ:

$$\exists u \in \Sigma^*, \quad \exists \gamma\prime \in \Sigma^w. \;\; u\gamma\prime = \gamma \wedge \gamma\prime \vDash M \tag{3.9}$$

- $\gamma \vDash \Box(P, M)$ (of invariant mode) iff, whenever the prechart is matched in γ, the main chart is matched afterwards:

$$\forall u, v \in \Sigma^*, \quad \forall \gamma\prime \in \Sigma^w, \quad uv\gamma\prime = \gamma. \;\; v \vDash P \Rightarrow \gamma\prime \vDash M \tag{3.10}$$

An additional relation is needed that indicates when the finite trace satisfies the preorder exactly once, and any truncated version of that trace does not satisfy the preorder. We define $w \vdash P$ as

$$\forall u, v \in \Sigma^*, \quad v \neq \emptyset \wedge uv = w. \;\; u \nvDash P \wedge uv \vDash P \tag{3.11}$$

Introducing universal charts with empty precharts Differently from [Bon05] and [DH01], the selected variant of universal LSC charts allows them have their precharts empty (without any events inside). This lets one specify how the modeled system can or should behave regardless the preconditions expressed in events. When the universal chart is present with the empty prechart, its activation is unrestricted, i.e. may occur eventually zero or more times. Such chart is still treated satisfied even if it has never activated. Informal semantics of such

LSC chart with sequence of events E in mainchart implies that "the sequence of events E *can*, or *is allowed* to happen".

For the universal chart of initial mode whose prechart is non-empty, the requirement to linearize the mainchart holds once, i.e. when the prechart is linearized for the first time:

- $\gamma \vDash_{initial} \Box(P, M)$ (of initial mode) with non-empty prechart iff, when the prechart is matched in γ for the first time, the main chart is matched afterwards:

$$\forall u \in \Sigma^*, \quad \forall \gamma\prime \in \Sigma^w, \quad u\gamma\prime = \gamma. \;\; u \vdash P \wedge \gamma\prime \vDash M \tag{3.12}$$

Universal type chart of initial mode and empty prechart is mainly treated as the existential chart. The difference is, that once one of the activating events of the chart have happened, it must complete:

- $\gamma \vDash_{initial} \Box(P, M)$ (of initial mode) with an empty prechart iff, when one of the activating events is matched in γ for the first time, the main chart is matched afterwards:

$$\begin{aligned} \forall u \in (\Sigma/\lambda(activating(M.L)))^*, \quad \forall v \in \lambda(activating(M.L)), \quad \forall \gamma\prime \in \Sigma^w, \\ uv\gamma\prime = \gamma. \;\; v\gamma\prime \vDash M \end{aligned} \tag{3.13}$$

For the universal chart of iterative mode with non-empty prechart, the requirement to linearize the mainchart holds every time from the start, but the new matching of prechart is not performed during current matching of mainchart. Definition of satisfaction relation is thus defined as satisfaction of consecutive chunks in the (infinite) trace, where every chunk satisfies the chart:

- $\gamma \vDash_{iterative} \Box(P, M)$ (of iterative mode with non-empty prechart) iff, when the trace consists of chunks $w_1, w_2, \ldots w_n$, where in each chunk prechart is satisfied for a single time, and right afterwards the main chart is matched:

$$\begin{aligned} \forall w_i \in \Sigma^*, \quad \forall \gamma\prime \in \Sigma^w, \quad \forall \gamma = \{w_i\}\gamma\prime, \\ \gamma\prime \nvDash P \wedge \forall w_i, \quad \exists u_i, v_i \in \Sigma^*. \;\; u_i v_i = w_i \wedge u_i \vdash P \wedge v_i \vdash M \end{aligned} \tag{3.14}$$

Universal chart of iterative mode with empty prechart has similar rules of being satisfied by the trace. Difference from the chart which has the prechart is that the activating symbols are enough to observe instead of waiting until the prechart is satisfied:

- $\gamma \vDash_{iterative} \Box(P, M)$ (of iterative mode) with empty prechart iff, when the trace consists of chunks $w_1, w_2, \ldots w_n$, where in each chunk the mainchart is activated for a single time, and right afterwards the main chart is matched:

$$\begin{aligned} \forall w_i \in \Sigma^*, \quad \forall \gamma\prime \in (\Sigma/\lambda(activating(M.L)))^w, \quad \forall \gamma = \{w_i\}\gamma\prime, \\ \forall w_i, \quad \exists u_i \in (\Sigma/\lambda(activating(M.L)))^*, \\ \forall x_i \in \lambda(activating(M.L)), \forall v_i \in \Sigma^*. \quad u_i x_i v_i = w_i \wedge x_i v_i \vdash M \end{aligned} \tag{3.15}$$

Universal chart of invariant mode with an empty prechart has the satisfaction relation also different from that with the non-empty prechart. It is easy to guess that matching of the prechart is replaced with observing the activating event in the mainchart:

- $\gamma \vDash \Box(P, M)$ (of invariant mode) with an empty prechart iff, whenever the activating event of the mainchart is matched in γ, the main chart is matched afterwards:

$$\begin{aligned} \forall \gamma \in \Sigma^w, \ \forall u \in \Sigma^*, \quad \forall v \in \lambda(activating(M.L), \\ \forall \gamma\prime \in \Sigma^w. \quad uv\gamma\prime = \gamma \wedge v\gamma\prime \vDash M \end{aligned} \tag{3.16}$$

A language L satisfies an existential LSC chart ($L \vDash \diamond(M)$ if $\exists \gamma \in L. \ \ \gamma \vDash \diamond(M)$).

A language L satisfies a universal LSC chart ($L \vDash \Box(P, M)$ if $\forall \gamma \in L. \ \ \gamma \vDash \Box(P, M)$).

Example

Let the sample LSC chart consist of prechart with locations $l_1 : a, l_2 : b$, where $l_1 \leq l_2$, and mainchart with locations $l_3 : c, l_4 : b, l_5 : a$, where $l_3 \leq l_4$ and $l_4 \leq l_5$. Locations in LPO relate to the message events in the LSC chart with same labels as locations are labeled.

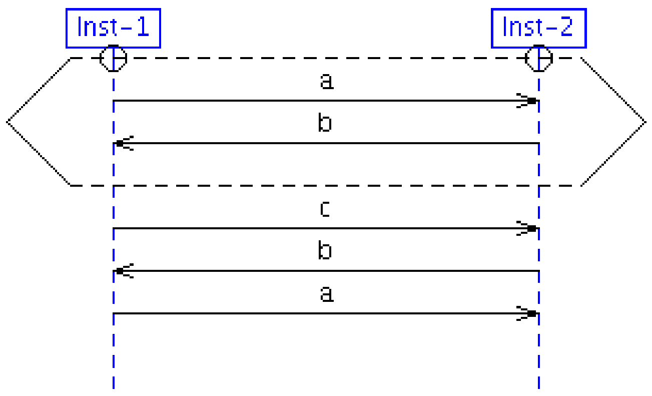

Figure 3.15: A sample LSC whose prechart includes totally ordered events labeled a, b and mainchart with totally ordered events labeled c, b, a, respectively.

Let pr be the prefix over alphabet $\{a, b, c\}$ which does not satisfy the prechart of the sample LSC chart, which is ab. Grammar of pr could be $((a^*c)^*b^*)^*a^*$.

Let pr_b be the prefix which does not satisfy the prechart, and also does not start with b.

Finite word satisfying such LSC chart of initial mode would be $pr.abcba(a|b|c)^*$. The first activation and linearizing of the mainchart satisfies the chart of initial mode and no more restrictions are applied to the word afterwards.

Finite word satisfying such LSC chart of iterative mode would be $pr.(abcba.pr)^+$. The mainchart of iterative mode chart must complete before the prechart is matched again.

Finite word satisfying such LSC chart of invariant mode would be $pr.(abcb(\emptyset|a.pr_b))^+.pr$. Here, the prechart can be started to match for the next time while the mainchart is still not matched completely, or matching is completed and the next prechart is not linearized immediately.

Example with an empty prechart Assume the prechart from figure 3.15 with an empty prechart, i.e. without events $\{a, b\}$.

The initial mode chart then would be satisfied by the words $(a|b)^*cba(a|b|c)^*$.

The invariant and iterative mode charts would be satisfied by the words $(a|b)^*(cba(a|b)^*)^*$.

3.7 Message and instance abstractions in LSC

The instance and message abstractions are used in [HM03], and the authors find them very useful. Same abstractions have been accepted in our flavor of LSC. Slight variations from the existing syntax and semantics are introduced in order to make the abstractions more flexible and convenient to specify.

3.7.1 Instance abstraction - example

In figure 3.16, the sample requirement is written in the style of [HM03]. It includes the instance abstraction $:: Client$ which applies to several concrete instances $Client1$ and $client2$. The corresponding charts are depicted in figures 3.16 (the abstract instance) and 3.17 with 3.18 (concrete instances).

In [HM03] it is described that a new copy, or incarnation, of the LSC chart is activated upon observing the corresponding message event. By corresponding event it is meant that concrete instances are involved in event that are the same or can be abstracted by the abstract instances specified in the LSC chart. Instance pair $\{Client1, Server\}$ can be abstracted by another pair $\{:: Client, Server\}$ as one of the instances ($Server$) is the same, and $Client1$ can be abstracted by $:: Client$. Another pair, say $\{Client1, Server2\}$ with an instance $Server2$ non-equal with $Server$, can not be abstracted by $\{:: Client, Server\}$.

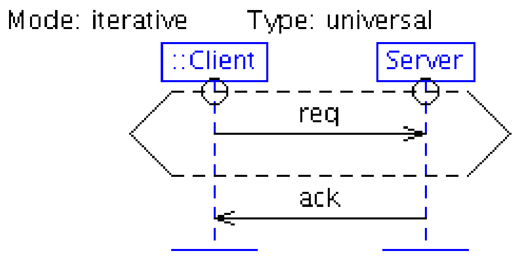

Figure 3.16: The requirement "To every *req* message from any client, the server must send the *ack* message to that client" formalized in LSC. Instance abstraction of the client is used and denoted as $:: Client$. The incarnations of the chart must bind particular instance of the client to some concrete value upon every event from any of the clients.

Given the instance abstraction, the events *req* and *ack* from figure 3.17 should not match those in figure 3.18, and vice versa, for the abovementioned reasons (the instances $Client1$ and $Client2$ are not the same and can not abstract each other).

Without an instance abstraction, several identic LSC charts with concrete instances like in figures 3.17 and 3.18 would be defined instead of chart with an instance abstraction like in figure 3.16. Not using the instance abstraction introduces several shortcomings:

- Redundant work needs to be done, namely number of charts defined with all the pairs of concrete instance names instead of one chart with abstract instance name(s)

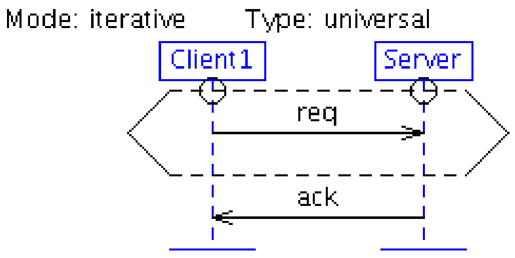

Figure 3.17: Partial case of the LSC from figure 3.16, applied to the concrete instance *Client*1 which can be abstracted by :: *Client*.

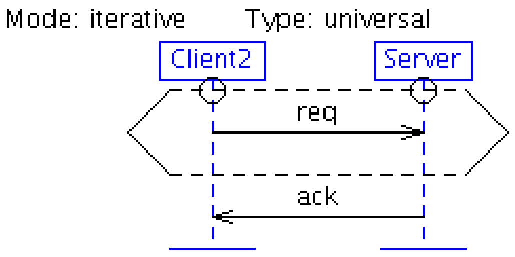

Figure 3.18: Partial case of the LSC from figure 3.16, applied to the concrete instance *Client*2 which can be abstracted by :: *Client*.

- Many charts need to be changed at a time as opposed to one chart with instance abstraction

There are two ways to maintain instance abstraction in LSC charts, that are

- Defining relation and abstraction of the instance names outside the LSC charts
- Encoding the relation and abstraction of the instance names through shared variables

Defining relation and abstraction of instances outside the LSC is used in [HM03]. It has lots own shortcomings:

- The instance abstractions need to be defined outside the LSC charts, in the file of instance names mapping. This limits the operation over the LSC charts. In particular, it needs additional checks when composing the specifications of arbitrary sets of LSC charts with different instance names
- Certain rules for abstraction need to be followed in order for the abstractions to make sense
- For every instance abstraction, there are several concrete instances. When an abstract instance in the LSC chart is bounded to particular instance, there is no way to implicitly refer to other concrete instances falling under same abstraction. One typically refers

to such instances when one wants to forbid or ignore events that span over these instances. In [HM03], referring to such instances is anyway implemented explicitly through their ID. Besides, additional graphical notations (balloons with expressions identifying which instance IDs are affected) are used to specify forbidden events related to instance abstraction. This adds more confusion than convenience.

In general, implicit instance abstraction and defining the abstraction relation outside the charts has proved to be less flexible and more complicated than the explicit abstraction approach. Therefore we stick to the latter approach.

Instance abstraction - implementation Shared integer variables, like the source instance ID and destination instance ID, simplify the instance abstraction substantially. It is possible to specify particular abstraction and instance binding directly in the chart by using local variables of the LSC chart. Besides, a single expression can refer to arbitrary set of concrete instances and restrict or forbid events involving those instances.

Instance abstraction - example Instance abstraction is not used in the primitive case studies of the thesis. The only example can be found in this section. However, it will be necessary in the real life case studies where the system scaling takes place.

3.7.2 Message abstraction

The message abstraction is implemented generally using same techniques as the instance abstraction described in section 3.7.1. Shared variables are used in addition to the message label to "extend" the message event identifier. Ignored and forbidden scopes can efficiently isolate the events from matching or occurring in specified charts, exactly like in the instance abstraction.

Message abstraction - implementation In [HM03], the message abstraction is also supported through usage of the variables. Our approach is similar to the former, the only difference again being explicit specification of the message type through shared integer variables.

Message abstraction - example Message abstraction is used extensively in the case studies throughout the thesis. It can be typically found in the setups where several different messages are used between two instances, or the message with same label is used among more than two instances, to distinct the source and destination of the message event. A sample message abstraction can be seen in section 3.2.3, figure 3.9. There, the abstract *setlevel* message event is coupled with the shared boolean variable, $from\_dimmer$. The variable is set whenever the *setlevel* event involves instances *Dimmer* and *Lamp*, and reset when *Switch* and *Lamp* instances are involved. The message event coupled with the variable is used both in the chart and in the forbidden scope below he chart. The restrictions over the variable are clearly visible and easy to perceive.

3.8 Event generation and matching by the LSC chart

Satisfaction relation for the event sequence against a LSC chart has been defined in section 3.6. The satisfaction has been defined assuming the property role of the LSC chart (matching the events only). Thus it has been assumed that the message event generation and assigning the shared variables their values is performed outside the LSC charts.

LSC charts can also be assigned the role of specification. Then the behavior of the LSC specification (the model) is defined by means of the set of specification-role charts that constitute the model. By behavior it is meant, what events and when can that set of LSC charts generate.

Such a LSC specification is started and usually driven by the universal charts whose precharts are empty. It is those charts that can activate spontaneously and generate the events in their maincharts that are matched in other charts.

The event is generated in one LSC chart, and all other LSC charts match it. The event generation mechanism is like a message broadcasting among several processes, where one process broadcasts the event, and the others match it.

Since the LSC charts with match-only semantics are suitable for testing, the event generation-capable semantics allows construct the specifications consisting of LSC charts. Their behavior can thus be simulated without an external event generator. In other words, the whole system with its environment can be modeled and simulated through event generation-capable LSC charts.

The second and even more important reason for introducing the generation and matching of events is instance and message abstraction. During the abstraction, there should be only one LSC chart where the shared variables are set up upon the message event. Generation of the event fulfills this requirement, since there is only one LSC chart where the event is generated. Having no means to identify a single LSC chart during a mesage event, would make the implementation of instance and message abstractions impossible.

To distinct the generated and matched event through its label, the event labels must be extended.

Assume that the infinite word $\gamma \in \Sigma^w$ has its symbols extended to those from alphabet $\Sigma^w_{?!} \subseteq (\Sigma \times \{?\}) \cup (\Sigma \times \{!\})$.

Upon occurrence of an event $e \in \Sigma$, it is then substituted with two extended events, $e?$ and $e!$. The LSC charts in the system specification then have to match one of these two extended events.

All property role charts and precharts of specification role charts can only match the elements of $\Sigma_?$. The maincharts of the specification charts can either match events from alphabet $\Sigma_?$ or events from $\Sigma_!$ that correspond to the triggering of the event on behalf of the particular chart.

Relation *orig* is a homomorphism which maps the symbols from extended alphabets to those of the original one:

$$\begin{gathered} orig : \Sigma_{?!} \longrightarrow \Sigma \\ \forall \sigma \in \Sigma.\ orig(\sigma!) = orig(\sigma?) = \sigma \end{gathered} \tag{3.17}$$

Relation *orig* is extended to the sequences of events:

$$\begin{gathered} orig : (\Sigma_{?!})^* \longrightarrow (\Sigma)^* \\ orig(\sigma_1\sigma_2 \ldots \sigma_n) = orig(\sigma_1)orig(\sigma_2) \ldots orig(\sigma_n) \end{gathered} \tag{3.18}$$

Now, the satisfaction relation can be redefined for existential type charts. Note that the existential type charts, LSC charts of property role and precharts of the specification role universal charts always operate on the subset of extended alphabet $\Sigma_?$:

- $\gamma \in \Sigma_?^w \vDash \diamond(M)$ iff it is eventually matched in γ:

$$\exists u \in \Sigma_?^*,\quad \exists \gamma\prime \in \Sigma_?^w.\ \ u\gamma\prime = \gamma \wedge orig(\gamma\prime) \vDash M \tag{3.19}$$

Satisfying the universal charts of property role Satisfaction relation for the universal type, invariant mode, property role chart is redefined as follows:

- $\gamma \in \Sigma_?^w \vDash \square(P, M)$ (of invariant mode) with non-empty prechart iff, whenever the prechart is matched in γ, the main chart is matched afterwards:

$$\forall u, v \in \Sigma_?^*,\quad \forall \gamma\prime \in \Sigma_?^w,\quad uv\gamma\prime = \gamma.\ \ orig(v) \vDash P \Rightarrow orig(\gamma\prime) \vDash M \tag{3.20}$$

- $\gamma \in \Sigma_?^w \vDash \square(P, M)$ (of invariant mode and property role) with an empty prechart iff, whenever the activating event of the main chart is matched in γ, the main chart itself is matched afterwards:

$$\begin{gathered} \forall u \in (\Sigma_?/\{\lambda(activating(M.L)) \times \{?\}\})^*,\quad \forall v \in \{\lambda(activating(M.L)) \times \{?\}\}, \\ \forall \gamma\prime \in \Sigma_?^w.\ \ uv\gamma\prime = \gamma.\ \ orig(v\gamma\prime) \vDash M \end{gathered} \tag{3.21}$$

For the universal chart of initial mode and property role, the requirement to linearize the mainchart holds once, i.e. when the prechart is linearized or the prechart is empty and the mainchart is activated for the first time:

- $\gamma \in \Sigma_?^* \vDash_{initial} \square(P, M)$ (of initial mode, property role and non-empty prechart) iff, when the prechart is matched in γ for the first time, the main chart is matched afterwards:

$$\forall u \in \Sigma_?^*.\ \ \forall \gamma\prime \in \Sigma_?^w.\ \ u\gamma\prime = \gamma.\ \ orig(u) \vdash P \wedge orig(\gamma\prime) \vDash M \tag{3.22}$$

- $\gamma \in \Sigma_?^* \vDash_{initial} \Box(P, M)$ (of initial mode, property role and empty prechart) iff, once the mainchart is activated by γ for the first time, it is linearized straight away:

$$\forall u \in (\Sigma_?/\{\lambda(\mathit{activating}(M.L)) \times \{?\}\})^*, \forall v \in \{\lambda(\mathit{activating}(M.L)) \times \{?\}\}, \quad \forall \gamma\prime \in \Sigma_?^w, \quad uv\gamma\prime = \gamma. \;\; \mathit{orig}(v\gamma\prime) \vDash M \tag{3.23}$$

For the universal chart of iterative mode and property role, the requirement to linearize the mainchart holds every time from the start, but the new matching of prechart (if non-empty) or activation of the next incarnation of mainchart (if prechart is empty) is not performed during current matching of mainchart. Definition of satisfaction relation is thus defined as satisfaction of consecutive chunks in the (infinite) trace, where every chunk satisfies the chart:

- $\gamma \in \Sigma_?^* \vDash_{iterative} \Box(P, M)$ (of iterative mode) with non-empty prechart iff, when the trace consists of chunks $w_1, w_2, \dots w_n$, where in each chunk prechart is satisfied for a single time, and right afterwards the main chart is matched:

$$\forall w_i \in \Sigma_?^*, \quad \forall \gamma\prime \in \Sigma_?^w, \quad \forall \gamma = \{w_i\}\gamma\prime, \quad \mathit{orig}(\gamma\prime) \nvDash P \wedge \forall w_i, \quad \exists u_i, v_i \in \Sigma_?^*. \;\; u_i v_i = w_i \wedge \mathit{orig}(u_i) \vdash P \wedge \mathit{orig}(v_i) \vdash M \tag{3.24}$$

- $\gamma \in \Sigma_?^* \vDash_{iterative} \Box(P, M)$ (of iterative mode) with empty prechart iff the trace consists of chunks $w_1, w_2, \dots w_n$, where in each chunk the main chart is matched once it is activated:

$$\forall w_i \in \Sigma_?^*, \quad \forall \gamma\prime \in (\Sigma_?/\{\lambda(\mathit{activating}(M.L)) \times \{?\}\})^w, \quad \forall \gamma = \{w_i\}\gamma\prime, \quad \forall w_i, \quad \exists u_i \in (\Sigma_?/\{\lambda(\mathit{activating}(M.L)) \times \{?\}\}), \quad x_i \in \{\lambda(\mathit{activating}(M.L)) \times \{?\}\}, v_i \in \Sigma_?^*. \;\; u_i x_i v_i = w_i \wedge \mathit{orig}(x_i v_i) \vdash M \tag{3.25}$$

Satisfying universal charts of specification role Slight difference between the universal charts with non-empty and empty precharts has been introduced already in section defined in section 3.6. Here, these very same relations are applied to the extended alphabets.

Satisfaction relation of the word $\gamma \in \Sigma_{?!}^*$ against the universal type, initial mode, specification role chart:

- $\gamma \in \Sigma_{?!}^* \vDash_{initial} \Box(P, M)$ (of initial mode) with non-empty prechart iff, when the prechart is matched in γ for the first time, the main chart is matched afterwards:

$$\forall u \in \Sigma_?^*, \; \forall v \in \Sigma_{?!}^*, \; \forall \gamma\prime \in \Sigma_?^w, \quad uv\gamma\prime = \gamma. \;\; \mathit{orig}(u) \vdash P \wedge \mathit{orig}(v) \vdash M \tag{3.26}$$

- $\gamma \in \Sigma_{?!}^* \vDash_{initial} \Box(P, M)$ (of initial mode) with an empty prechart iff, the activating symbol of the main chart is matched in γ for the first time, the whole main chart is generated matched starting with that symbol:

$$\forall u \in (\Sigma_?/\{\lambda(activating(M.L)) \times \{?\}\})^*,\ \forall v \in \{\lambda(activating(M.L)) \times \{?,!\}\}, \tag{3.27}$$
$$\forall x \in \Sigma^*_{?!},\ \forall \gamma\prime \in \Sigma^w_?.\ \ uvx\gamma\prime = \gamma.\ \ orig(vx) \vdash M$$

Satisfaction relation of the word $\gamma \in \Sigma^*_{?!}$ against the universal type, iterative mode, specification role chart also depends on whether the prechart is empty or not:

- $\gamma \in \Sigma^*_{?!} \vDash_{iterative} \Box(P, M)$ (of iterative mode and with non-empty prechart) iff, when the trace consists of chunks $w_1, w_2, \ldots w_n$, where in each chunk prechart is satisfied for a single time, and right afterwards the main chart is matched:

$$\forall w_i \in \Sigma^*_{?!},\ \ \forall \gamma\prime \in \Sigma^w_{?!},\ \ \forall \gamma = \{w_i\}\gamma\prime, \tag{3.28}$$
$$orig(\gamma\prime) \nvDash P \wedge \forall w_i,\ \ \exists u_i \in \Sigma^*_?, v_i \in \Sigma^*_{?!}.\ \ u_i v_i = w_i \wedge orig(u_i) \vdash P \wedge orig(v_i) \vdash M$$

- $\gamma \in \Sigma^*_{?!} \vDash_{iterative} \Box(P, M)$ (of iterative mode and with empty prechart) iff, when the trace consists of chunks $w_1, w_2, \ldots w_n$, where in each chunk the main chart is matched upon observing one of its activating events:

$$\forall w_i \in \Sigma^*_{?!},\ \ \forall \gamma\prime \in \Sigma^w_{?!},\ \ \forall \gamma = \{w_i\}\gamma\prime, \tag{3.29}$$
$$\forall w_i,\ \ \exists u_i \in (\Sigma_?/\{\lambda(activating(M.L)) \times \{?\}\}),$$
$$x_i \in \{\lambda(activating(M.L)) \times \{?,!\}\}, v_i \in \Sigma^*_{?!}\ \ u_i x_i v_i = w_i \wedge orig(x_i v_i) \vdash M$$

Satisfaction relation of the word $\gamma \in \Sigma^*_{?!}$ against the universal type, invariant mode, specification role chart:

- $\gamma \in \Sigma^w_{?!} \vDash \Box(P, M)$ (of invariant mode with non-empty prechart) iff, whenever the prechart is matched in γ, the main chart is matched afterwards:

$$\forall u, v \in \Sigma^*_?,\ \forall \gamma\prime \in \Sigma^w_{?!},\ \ uv\gamma\prime = \gamma,\ \ v \in (\Sigma_?)^* \wedge orig(v) \vDash P \wedge \gamma\prime \vDash M \tag{3.30}$$

- $\gamma \in \Sigma^w_{?!} \vDash \Box(P, M)$ (of invariant mode with empty prechart) iff for each symbol that belongs to the set of activating symbols of the mainchart, there is indeed a mainchart copy satisfied afterwards in the word:

$$\forall \gamma\prime \in \Sigma^w_{?!}, \forall w \in \Sigma^*_{?!},\ \ w \sqsubset \gamma,\ \ \forall i.\ orig(w_i) \in \lambda(activating(M.L)), \tag{3.31}$$
$$\exists j \geq i.\ w_i \ldots w_j \vdash M$$

Summary of satisfaction relations

The satisfaction relations for the LSC charts depending on their type, mode, role and emptiness of the prechart are presented in table 3.2.

Type	Mode	Role	Empty prechart	Equation
existential	all	all	no	3.19
universal	initial	property	no	3.22
		property	yes	3.23
		specification	no	3.26
		specification	yes	3.27
	iterative	property	no	3.24
		property	yes	3.25
		specification	no	3.28
		specification	yes	3.29
	invariant	property	no	3.20
		property	yes	3.21
		specification	no	3.30
		specification	yes	3.31

Table 3.2: Definition of the type, mode and role attributes of the LSC chart.

Example

In figure 3.15, the LSC chart has been depicted with events a, b in the prechart and c, b, a in the mainchart.

The word linearizing such chart of property role despite its mode would be
$W = \{a?\ b?\ c?\ b?\ a?\}$, and for the specification role the word would be
$W = \{a?\ b?\ (c?|c!)\ (b?|b!)\ (a?|a!)\}$.

The set of infinite words that satisfy the chart of specification role, initial mode would be of the form $VW\gamma$ where the prefix is $V = (b?|c?)^*$ and suffix is $\gamma = \Sigma_?^w$.

If the chart would have specification role, initial mode and empty prechart, the satisfying words would match the pattern $VW\gamma$ where $V = (a?|b?)^*$, $W = (c?|c!)(b?|b!)(a?|a!)$ and $\gamma = \Sigma_?^w$.

3.9 Satisfaction relation for the LSC specification

LSC specification consists of non-empty set of the specification role universal type charts and arbitrary amount of other role or type charts.

Infinite word $\gamma \in \Sigma^w$ satisfies the specification iff it can be multiplied into as many copies as many charts and the copies modified in the following way:

- Every symbol of the word $\sigma \in \Sigma$ is extended into $(\sigma\prime \in \Sigma_{?!} | orig(\sigma\prime) = \sigma)$ in the copy.
- In all copies of the words $\gamma = \{\sigma\prime_i\}$, all symbols with same index i, there is only one $\sigma\prime_i \in \Sigma_!$ and all the rest are $\sigma\prime_i \in \Sigma_?$, such that $\sigma\prime_i \in \{\sigma?, \sigma!\}$.
- The copies of the word are distributed in such a way that all the charts are satisfied by their assigned copies.

Assume a LSC system consisting of universal and existential charts $Sys = \{Ch_i\}$.

Amount of charts in the system is denoted $n = |Sys|$.

Assume set of infinite traces $\{\kappa_i\}, \kappa_i \in (\Sigma^w_{?!})$, such that corresponding symbols in these traces have same original, at most one of these symbols is from $\Sigma_?$:

$\forall t, r \in \{\kappa_i\}, \forall j \in \mathbb{N},\ orig(t_j) = orig(r_j) \wedge (t_j \in \Sigma_? \vee r_j \in \Sigma_?)$.

Also, there is always one trace whose j_{th} symbol is from $\Sigma_!$:

$\forall j \in \mathbb{Z},\ \exists \kappa \in \{\kappa_i\}.\ \kappa_j \in \Sigma_!$

$\forall j \in \mathbb{Z}, \forall \kappa, \kappa\prime \in \{\kappa_i\}.\ \kappa \neq \kappa\prime \wedge \kappa_j \in \Sigma_! \Rightarrow \kappa\prime_j \in \Sigma_?$

Infinite word satisfies the system, denoted $\gamma \in \Sigma^w \models Sys = \{Ch_i\}$, iff there exists such set of abovementioned traces, each of whom reduces to γ:

$\gamma \models Sys \Leftrightarrow \exists\{\kappa_i\}.\ |\{\kappa_i\}| = |\{Ch_i\}| \wedge \forall \kappa \in \{\kappa_i\}.\ orig(\kappa) = \gamma$.

Example

Let the LSC specification *LSCSpec* consist of four LSC charts, depicted in figures 3.19, 3.20, 3.21 and 3.22. Assuming that these all LSC charts are of the specification role, exemplary prefix of the word generated by the LSC charts altogether and their progress upon generating or accepting symbols of that word is depicted in figure 3.23.

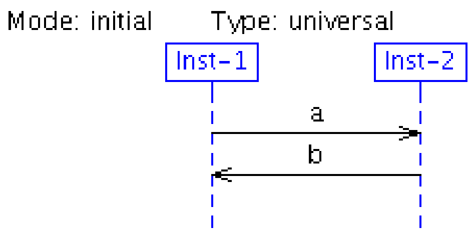

Figure 3.19: A sample universal type, specification role LSC of initial mode.

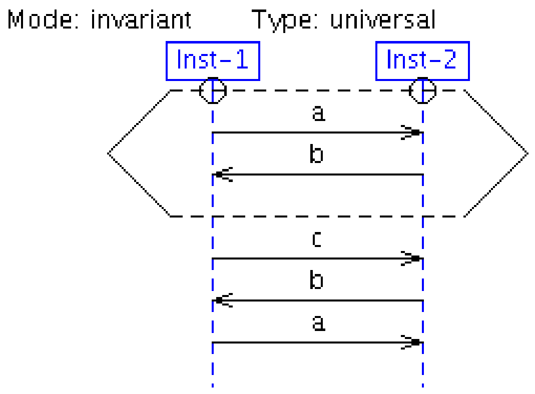

Figure 3.20: A sample universal type, specification role LSC of invariant mode.

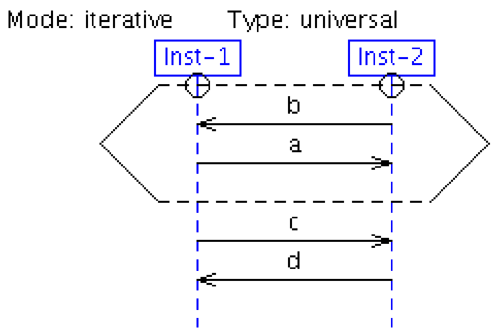

Figure 3.21: A sample universal type, specification role LSC of iterative mode.

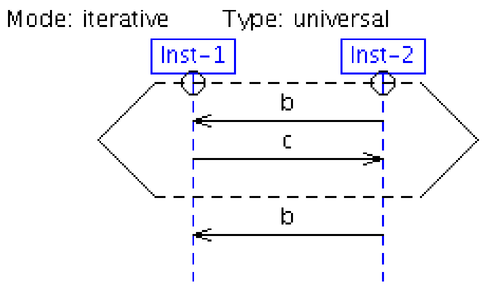

Figure 3.22: A sample universal type, specification role LSC of iterative mode.

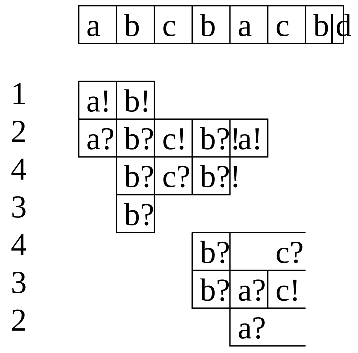

Figure 3.23: Word satisfying the LSC specification *LSCSpec*. Below the symbols, activations of corresponding LSC charts are presented with the chart number on the left. Labels with ?, ! represent symbols from alphabets $\Sigma_?$ and $\Sigma_!$, respectively. Labels with ?! denote that one LSC chart can take the symbol from $\Sigma_!$ while others from $\Sigma_?$. Expression $b|d$ at the end of the word means that *LSCSpec* will accept the any of these symbols can come next in the word.

3.10 Data and time components used in LSC

Data and time components used in LSC charts can be grouped into variables, clocks, channels, guards and invariants, and assignments.

Variables used in LSC charts are integers. Finite set of the variables Var is operated on in the LSC charts.

Clocks are the clock variables. Finite set of the clocks is denoted $clocks$.

Channels Let $Chan$ be a finite set of synchronization channels. Communication among the LSC charts of the specification occurs via channels from $Chan$.

Synchronizations For each $c \in Chan$, there are two possible synchronizations $c!, c? \in Sync$ where $Sync$ is the set of all possible synchronizations.

Guards and invariants are the boolean expressions over the variables and clocks. Data constraint has the form $E \bowtie E$ where E is arithmetic expresionover Var and $\bowtie \in \{<, \leq, =, \geq, >\}$. Clock constraint has the form $x \bowtie n$ or $x - y \bowtie n$ where $x, y \in Clocks$, $n \in \mathbb{Z}$ and $\bowtie \in \{<, \leq, =, \geq, >\}$. A guard is a finite conjunction over data and clock constraints. An invariant is a finite conjunction over data constraints and clock constraints.

Sets of guards and invariants are denoted $Guard$ and $Invariant$, respectively, and both include constants **true** and **false** in addition.

Assignments A clock reset is of the form $x := 0$ where $x \in Clocks$. A data assignment (update) is of the form $v := E$ where $v \in Var$ and E is an arithmetic expression over Var.Sets of clock resets and variable updates are denoted $Reset$ and $Update$, respectively.

3.11 Visual constructs in the LSC chart

The LSC constructs maintained in this iteration of the LSC chart syntax are messages from set $\mathbf{M}$, conditions from $\mathbf{D}$ and actions from $\mathbf{A}$. Union of these sets represents all the possible LSC constructs separately and is denoted as $\mathbf{U} = \mathbf{M} \cup \mathbf{D} \cup \mathbf{A}$.

Messages have a label:

$$label : \mathbf{M} \rightarrow Chan \tag{3.32}$$

Label of a message m is obtained via $label(m)$.

Conditions have guards and invariants (may be multiple in a condition):

$$\begin{aligned} guard &: \mathbf{D} \times Guard \\ invar &: \mathbf{D} \times Invariant \end{aligned} \tag{3.33}$$

Set of guards (invariants) belonging to the condition are accessed via relations *guards* and *invariants*:

$$\begin{aligned} \forall d \in \mathbf{D}.\ guards(d) &= \{g \in Guard | guard(d, g)\} \\ \forall d \in \mathbf{D}.\ invars(d) &= \{i \in Invariant | invar(d, i)\} \end{aligned} \tag{3.34}$$

Clock resets and assignments related to LSC action constructs:

$$\begin{aligned} reset &: \mathbf{A} \times Reset \\ \forall a \in \mathbf{A}.\ resets(a) &= \{r \in Reset | reset(a, r)\} \\ update &: \mathbf{A} \times Update \\ \forall a \in \mathbf{A}.\ updates(a) &= \{r \in Update | update(a, r)\} \end{aligned} \tag{3.35}$$

Set of instances $\mathbf{I}$ are owned by the LSC chart.

The LSC message, action and condition constructs can be aggregated into atomic structures, called *simregions*. In the labeled preorder semantics, each simregion corresponds to a single location, just like the standalone message constructs did in section 3.4.0.2. Simregions are atomic constructs in a sense that their comprising constructs (messages, conditions, actions) are processed in a single step.

Set of simregions $\mathbf{S}$ is defined that are subsets of LSC constructs from $\mathbf{U}$:

$$\mathbf{S} = \{s | s \subseteq \mathbf{U}\} \tag{3.36}$$

Simregions have one or more instances assigned to each:

$$\begin{aligned} insts &\subseteq \mathbf{S} \times \mathbf{I} \\ \forall s \in \mathbf{S} \exists i \in \mathbf{I}.&\ i \in insts(s) \end{aligned} \tag{3.37}$$

LSC elements in a simregion are interpreted in an atomic step. Simregions access the LSC properties such as labels, actions, invariants and guards of their contained LSC components relations:

$$\begin{aligned}
\forall s \in \mathbf{S}.\ labels(s) &= \{chan \in Chan | \forall msg \in s.\ chan = label(msg)\} \\
\forall s \in \mathbf{S}.\ guards(s) &= \{g \in Guards | \forall cond \in s.\ g = guard(cond)\} \\
\forall s \in \mathbf{S}.\ invars(s) &= \{i \in Invars | \forall cond \in s.\ i = invar(cond)\} \\
\forall s \in \mathbf{S}.\ resets(s) &= \{r \in Resets | \forall act \in s.\ r = reset(act)\} \\
\forall s \in \mathbf{S}.\ updates(s) &= \{r \in Updates | \forall act \in s.\ r = update(act)\}
\end{aligned} \tag{3.38}$$

It is important how the conditions and actions are attached to the message if any in simregion. Relation *isPrec* determines for simregions the relative position of conditions and actions w.r.t. messages and stands for precondition:

$$isPrec : \mathbf{S} \rightarrow \mathbb{B} \tag{3.39}$$

3.11.1 Well-formedness rules for simregions

There are several well-formedness rules for simregions:

- no two LSC constructs of the same kind in any union:

$$\begin{aligned}
\forall s \in \mathbf{S}, \forall m, n \in \mathbf{M}.\ m \in s \wedge n \in s \Rightarrow m = n \\
\forall s \in \mathbf{S}, \forall m, n \in \mathbf{D}.\ m \in s \wedge n \in s \Rightarrow m = n \\
\forall s \in \mathbf{S}, \forall m, n \in \mathbf{A}.\ m \in s \wedge n \in s \Rightarrow m = n
\end{aligned} \tag{3.40}$$

- simregions are never empty:
  $\forall s \in \mathbf{S} \exists e \in \mathbf{U}.\ e \in u$
- simregions without messages and simregions are always preaction:
  $\forall s \in \mathbf{S}.\ \nexists m \in \mathbf{M}.\ hasMsg(s, m) \Rightarrow isPrec(s)$
- no two simregions share same construct:
  $\forall s, s\prime \in \mathbf{S}, \forall u \in \mathbf{U}.\ u \in s \wedge u \in s\prime \Rightarrow s = s\prime$

Example

In figure 3.24, several types of simregions are presented. Properties of each simregion and LSC constructs belonging to it are described:

- Simregion s_1 at the left upper corner consisting of message, condition and assignment, with the following properties:

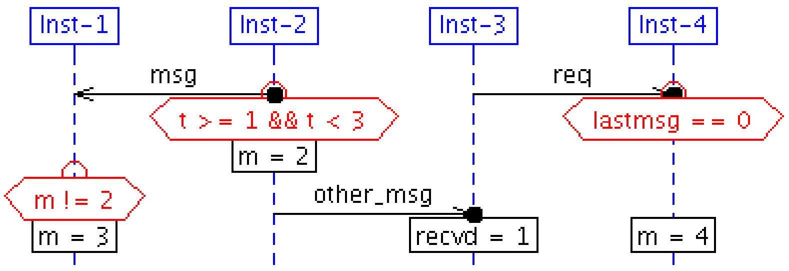

Figure 3.24: Examples of simregions.

- $labels(s_1) = \{msg\}$, $invariants(s_1) = \{t \geq 1, t < 3\}$ given that t is a clock, $updates(s_1) = \{m = 2\}$, $insts(s_1) = \{Inst - 1, Inst - 2\}$, $isPrec(s_1) = \mathbf{tt}$

- Simregion s_2 at the right upper corner consisting of a message and condition, with the following properties:
  - $labels(s_2) = \{req\}$, $guards(s_2) = \{lastmsg == 0\}$ given that $lastmsg$ is a variable of integer type, $insts(s_2) = \{Inst - 3, Inst - 4\}$, $isPrec(s_2) = \mathbf{ff}$

- Simregion s_3 at the left lower corner, consisting of a condition and assignment, with the following properties:
  - $guards(s_3) = \{m! = 2\}$ given that m is a variable of integer type, $updates(s_3) = \{m = 3\}$, $insts(s_3) = \{Inst - 1\}$, $isPrec(s_3) = \mathbf{tt}$

- Simregion s_4 at the right lower corner consisting of an assignment, with the following properties:
  - $updates(s_4) = \{m = 4\}$, $insts(s_4) = \{Inst - 4\}$, $isPrec(s_4) = \mathbf{tt}$

- Simregion s_5 at the middle lower part of LSC, consisting of a message and assignment, with the following properties:
  - $labels(s_5) = \{other\_msg\}$, $updates(s_5) = \{recvd = 1\}$, $insts(s_5) = \{Inst - 2, Inst - 3\}$, $isPrec(s_5) = \mathbf{ff}$

3.12 Definition of the cut

Assume the prechart or mainchart of the LSC chart as the LPO $\mathcal{L} = \langle L, \leq, \lambda, \mathbf{S} \rangle$ where

L is the set of locations,

$\leq$ is preorder relation over the locations

$\mathbf{S}$ is the set of simregions

λ is the labeling function for locations such that $\lambda : \mathcal{L} \rightarrow S$

The cut of such LPO is defined according to the equation 3.4 in section 3.4.0.2. Same concept of the cut will be applied to the prechart or mainchart of LSC chart.

A cut $c\prime$ is an s-successor of a cut c if $s \in \mathbf{S}$ and there is a location l such that

1. $\lambda(l) = s$
2. $l \notin c$
3. $c\prime = c \cup \{l\}$

A simregion s is called *enabled* from cut c iff there exists a cut $c\prime$ which is a s-successor of c.

Example

In section 3.11, figure 3.24, an example of the LSC with mainchart only is given. Let us assume that mainchart as LPO. Assuming that its locations are $l_1 \ldots l_5$, labeled with simregions s_1 to s_5 respectively, the following preorder relation holds: $l_1 \leq l_3$, $l_1 \leq l_5$, $l_2 \leq l_3$, $l_2 \leq l_5$.

The possible cuts are $\{\emptyset\}$, $\{l_1\}$, $\{l_2\}$, $\{l_1, l_2\}$, $\{l_1, l_3\}$, $\{l_2, l_4\}$, $\{l_1, l_2, l_3\}$, $\{l_1, l_2, l_4\}$, $\{l_1, l_2, l_5\}$, $\{l_1, l_2, l_3, l_4\}$, $\{l_1, l_2, l_3, l_5\}$, $\{l_1, l_2, l_4, l_5\}$, $\{l_1, l_2, l_3, l_4, l_5\}$.

3.13 LSC subchart

LSC subchart is the labeled preorder with set of visible LSC constructs and variables with clocks referred from there.

LSC subchart is a tuple $\mathcal{S} = \langle \mathcal{L}, Var, Clocks \rangle$ where

$\mathcal{L}$ is labeled preorder introduced earlier: $\mathcal{L} = \langle L, \leq, \lambda, \mathbf{S} \rangle$

Var is the set of variables referred from the LSC constructs of the preorder

$Clocks$ is the set of clocks referred from the LSC constructs of the preorder

3.13.1 Configuration of the LSC subchart

A configuration of a LSC subchart $\mathcal{S} = \langle \mathcal{L}, Var, Clocks \rangle$ is a tuple (c, ϵ, ν) where

c is the cut

ϵ is the value of the variables: $\epsilon : Vars \to (\mathbb{Z})^*$

ν maps every clock to non-negative real number: $\nu : Clocks \to (\mathbb{R}_{\geq 0})^*$. For $d > 0$, notation $(\nu + d) : Clocks \to (\mathbb{R}_{\geq 0})^*$ describes the function ν shifted by d such that $\forall x \in Clocks.\ (\nu(x) + d) = \nu(x) + d$

3.14 Trace semantics of LSC subchart

Introduction of variables and clocks to the LSC charts makes the semantics more complex than the primitive preorder. The concepts of the LSC chart and cut advancement remain the same, but the operational semantics needs to be redefined for the sake of simplicity and understandability. Approach of the step based operational semantics is thus taken.

The LSC subchart advances according to legal steps, that are either delays or actions.

Notation $\epsilon, \nu \vDash \phi$ will be used to indicate that the boolean expression ϕ is satisfied with the values of the variables ϵ and clocks ν.

Assignments a will be used as transformers of the functions ϵ and ν; the resulting evaluations will be denoted $a(\epsilon),\ a(\nu)$.

Configuration of the subchart will be denoted (c, ϵ, ν) where c is its cut, $\epsilon : Vars \longrightarrow \mathbb{Z}^*$ function maps the integer variables to their values, and $\nu : Clocks \longrightarrow (\mathbb{R}_{\geq 0})^*$ maps the clocks to their values.

Simple action step

For an LSC subchart configuration (c, ϵ, ν), a simple action step is enabled if there is a transition $c \xrightarrow{s} c\prime$ where

- $c\prime$ is a s-successor of c
- no messages are in the simregion: $\nexists m \in \mathbf{M}.\ hasMsg(s, m)$

Simple action step will be denoted $(c, \epsilon, \nu) \overset{\tau}{\Longrightarrow} (c\prime, a(\epsilon), a(\nu))$

Synchronization step

For an LSC subchart configuration (c, ϵ, ν), a synchronization step is enabled if there is a transition $c \xrightarrow{s} c\prime$ where

- $c\prime$ is a s-successor of c
- the simregion has a message: $\exists m \in \mathbf{M}.\ hasMsg(s, m)$
- synchronization label belongs to the message inside simregion: $a \in labels(s)$

Synchronization step can be denoted $(c, \epsilon, \nu) \overset{a?}{\Longrightarrow} (c\prime, a(\epsilon), a(\nu))$ or
$(c, \epsilon, \nu) \overset{a!}{\Longrightarrow} (c\prime, a(\epsilon), a(\nu))$.

Delay step

The delay step increases the values of the clock variables:
$(c, \epsilon, \nu) \overset{d}{\Longrightarrow} (c, \epsilon, (\nu + d))$.

Delay step is allowed if the LSC subchart still has the enabled transitions afterwards: $\exists s \in \mathbf{S}.\ (c, \epsilon, (\nu + d)) \overset{a}{\Longrightarrow} s \quad \vee \quad (c, \epsilon, (\nu + d)) \overset{\tau}{\Longrightarrow} s$

Well-formed sequence of steps

Let $\mathcal{S}$ be a LSC subchart $\mathcal{S} = \langle \mathcal{L}, Var, Clocks \rangle$. A sequence of configurations $\{(c, \epsilon, \nu)\}^K = \{(c, \epsilon, \nu)^0, (c, \epsilon, \nu)^1, \ldots\}$ of length $K \in \mathbb{N}$ is called a well-formed sequence for $\mathcal{S}$ iff

- sequence starts in minimal cut, i.e. $(c, e, \nu)^0 = (c_\top, e, \nu)$
- for $(c, \epsilon, \nu)^K$, c^K is $c^K = c_\bot$, or no further step is enabled

Sequence of steps satisfying the LSC subchart

A well-formed sequence of steps is called a timed trace for LSC subchart $\mathcal{S}$ if in addition the following holds:

- for every $k < K$, the configuration k is optionally changed externally to $k\prime$ so that $k\prime$ and $k + 1$ are connected via a simple action step, a synchronized action step or a delay step. This change of configuration is usually performed by other progressing LSC subcharts

A timed trace $TT = (Chan \times \{?, !\} \cup \mathbb{R}_{\geq 0} \cup \tau)^*$ satisfies the LSC subchart of the LSC chart if it has a well-formed prefix that drives the configuration with initial cut to the configuration with the maximal cut:

$$TT \models \mathcal{S} \Rightarrow \exists Tr \sqsubset TT.\ (c_\top, \epsilon, \nu) \overset{Tr}{\rightarrow} (c_\bot, e\prime, \nu\prime) \tag{3.41}$$

An infinite timed trace $TT = (Chan \times \{?, !\} \cup \mathbb{R}_{\geq 0} \cup \tau)^w$ satisfies the LSC subchart if it has a well-formed prefix that satisfies that LSC subchart:

$$\begin{aligned} TT \models \mathcal{S} \Rightarrow \exists Tr \in (Chan \times \{?, !\} \cup \mathbb{R}_{\geq 0} \cup \tau)^*, \exists Cont \in (Chan \times \{?, !\} \cup \mathbb{R}_{\geq 0} \cup \tau)^w. \\ TT = TrCont \wedge Tr \models \mathcal{S} \end{aligned} \tag{3.42}$$

Example

In section 3.11, an example LSC is given in figure 3.24. Let its configuration be $(c, \{\epsilon(lastmsg), \epsilon(m), \epsilon(recvd)\}, \{\nu(t)\})$. Sample set of steps satisfying the LSC subchart can be $(c_\emptyset, \{0, 0, 1\}, \{1.23\}) \overset{msg}{\Rightarrow} (c_1, \{0, 2, 1\}, \{1.23\}) \overset{\tau}{\Rightarrow} (c_{13}, \{0, 3, 1\}, \{1.23\}) \overset{req}{\Rightarrow} (c_{123}, \{0, 3, 1\}, \{1.23\}) \overset{other\_msg}{\Rightarrow} (c_{1235}, \{0, 3, 1\}, \{1.23\}) \overset{\tau}{\Rightarrow} (c_{12345}, \{0, 4, 1\}, \{1.23\})$.

The value of m should be changed somewhere outside the LSC subchart to a value different from 2 externally while the LSC subchart has been in configuration with cut c_1. Otherwise the sequence of steps would not satisfy the LSC subchart.

3.15 Definition of the LSC chart

LSC chart is a tuple $LSC = \langle Type, Mode, Role, \mathbf{I}, \mathcal{S}_p, \mathcal{S}_m \rangle$ where

Type is the chart type from the set $\{existential, universal\}$

Mode is the chart mode from the set $\{initial, iterative, invariant\}$

Role is the chart role from the set $\{property, specification\}$

I is the set of instances in the LSC chart

$\mathcal{S}_p$ is the LSC subchart called *prechart*. Existential charts consist of the prechart only. For universal charts, prechart is optional.

$\mathcal{S}_m$ is the LSC subchart called *mainchart*.

The type, mode and role properties of the LSC chart specify its flavors of the behavior.

3.15.1 Well-formedness of the LSC charts

There are several rules regarding the well-formedness of the LSC charts:

- LSC charts of the existential type do not have the mainchart
- Variables and clocks of the mainchart include those of the prechart. This is necessary to access the variables at the mainchart that have been preset (initialized) at the prechart.

Minimal cut of the LSC chart is either minimal cut of its prechart, if such exists, or minimal cut of mainchart otherwise.

Maximal cut of the LSC chart is either maximal cut of its mainchart if mainchart exists, and maximal cut of its prechart otherwise.

3.16 Satisfaction relation for the LSC chart

As in the case of messages-only semantics, the satisfaction relation of the chart depends on the LSC chart type, role and mode (described in section 3.8).

Since clocks are introduced to capture the state of the LSC subchart, the delay action has to be introduced into the action alphabet as well. The delay action will be denoted as the non-negative real number and correspond to the amount of time units delayed.

Alphabet of the timed trace is thus redefined as $\Sigma_{?!} \in (\mathit{Sync} \cup \mathbb{R}_{\geq 0} \cup \tau)$ to include the mesage events, silent actions and delays, where $\mathit{Sync} \equiv \mathit{Chan} \times \{?,!\}$ stands for the already known extended alphabet of the message events. Reduced versions of $\Sigma_{?!}$ are $\Sigma_{?} \in (\mathit{Chan} \times \{?\} \cup \mathbb{R}_{\geq 0} \cup \tau)$ and $\Sigma_{!} \in (\mathit{Chan} \times \{!\} \cup \mathbb{R}_{\geq 0} \cup \tau)$, each supplemented with delay and silent action symbols besides the already defined extended alphabet over the message events.

Relation previously defined in equation 3.11 needs redefining. As before, it indicates whether the finite trace satisfies the LSC subchart $\mathcal{S}$ exactly once, and any truncated version of that trace does not satisfy the subchart. Relation $\vdash$ is now expressed as

$$\forall u, v \in \Sigma_{?!}^{*}, \quad \forall w \in \Sigma_{?!}^{*}, \quad v \neq \emptyset \wedge uv = w, \quad u \nvDash \mathcal{S} \wedge uv \vDash \mathcal{S} \Rightarrow w \vdash \mathcal{S} \tag{3.43}$$

Satisfaction of the LSC charts depending on their type, mode and role is already introduced in section 3.8, table 3.2. The only difference is usage of the extended alphabets $\Sigma_{?!}$, its subsets $\Sigma_{?}$ with $\Sigma_{!}$ and relation $\vdash$ from the current section.

3.17 LSC specification and its satisfaction relation

LSC specification $LSCSpec$ is a tuple $\langle LSCs, Vars, Clocks\rangle$ where

$LSCs$ is the set of LSC charts participating in specification

$Vars$ is the set of variables used by the LSC charts who constitute the specification. All variables in $Vars$ are accessed by one or several LSC charts from $LSCs$.

$Clocks$ is the set of clocks, also accessed from $LSCs$.

Configuration of the $LSCSpec$, similarly to that of a subchart, is denoted $(cuts, \epsilon, \nu)$ where:

$cuts$ is multiset of cuts from precharts and maincharts of $LSCs$ which is populated according to rules introduced below. Size of multiset is variable, but bounded.

ϵ is the mapping function for discrete variables $e : Vars \to \mathbb{Z}^*$

ν is the mapping function for the clocks $\nu : Clocks \to \mathbb{R}^*_{\geq 0}$

Similarly to transformation functions of the LSC subchart, such functions are introduced to $LSCSpec$ and denoted $a(cuts), a(\epsilon), a(\nu)$.

LSC specification accepts timed traces over alphabet $\Sigma = Sync \cup \mathbb{R}_{\geq 0} \cup \tau$.

3.17.0.1 Cuts transformation function

Cuts transformation function $a(cuts)$ passes several phases when specification accepts a symbol. Resulting configuration is $a(cuts) = cuts \cup startcuts \setminus termcuts \cup maincuts$ where

$cuts$ is the multiset of cuts before the step

$startcuts$ is the set of cuts from LSC charts that will be activated during the simple or synchronized action step. If corresponding LSC has a prechart, these are the precharts, and maincharts otherwise. Few cases are possible:

- In case of silent action where the cut of LSC subchart is not the minimal cut, $startcuts$ is empty set. This situation corresponds to internal step of the chart cut which has already started and not yet completed.
- In case of silent action where a chart is activated (and its prechart if any, or mainchart, starts in initial cut), $startcuts$ includes that cut. This case corresponds to activation of LSC chart via silent event. No LSC subcharts are in $startcuts$ from iterative mode LSCs whose mainchart is already in $cuts$.
- In case of synchronized action syn, $startcuts$ are minimal cuts of $LSCs$ with $\overset{syn!}{\Rightarrow}$ or $\overset{syn?}{\Rightarrow}$ except those iterative mode LSCs whose mainchart configuration is already in $cuts$.

termcuts is the set of cuts which, after accepting symbol given to *LSCSpec*, will become maximal cuts in their LSC subcharts. This applies both to prechart and mainchart cuts. In addition to that, all the active cuts of the iterative mode LSC prechart are included in *termcuts* if any of those will become maximal after accepting the symbol.

maincut is the set of minimal cuts of main charts. These are mainchart cuts whose prechart cuts are in *termcuts* set. Only one mainchart cut is allowed into *maincuts* for each iterative mode LSC.

3.17.0.2 Silent action step

Simple action is performed by one LSC subchart at a time. Its cut is advanced and certain simregion without message stepped over. Guards and invariants (precondition) must be satisfied in order for the cut to advance. Data and clocks values are changed according to the update and reset actions (preaction) of all assignments in the assignment of that simregion.

Let s be the simregion between current cut c and next cut $c\prime$ in LSC subchart, which is advancing according to own simple action step: $conf = (c, \epsilon, \nu)$, $c \in (cuts \cup startcuts)$. Guards and invariants related to the simregion are $guards(s)$ and $invars(s)$, respectively. Updates of variables and resets of clocks are $updates(s)$ and $resets(s)$. Variables and clocks referred from simregion will be denoted $Vars(s)$ and $Clocks(s)$.

Simple action is allowed iff there are no other simple actions where simregion with condition is stepped over, or if such simregion is going to be stepped over. This restriction maintains ASAP evaluation semantics of conditions.

Simple action step is denoted $(cuts, \epsilon, \nu) \overset{\tau}{\Rightarrow} (cuts\prime, \epsilon\prime, \nu\prime)$ where

- $cuts\prime = cuts \cup startcuts/termcuts \cup maincuts$, as specified before
- $\forall cond \in guards(s).\ Vars \vDash cond$
- $\forall cond \in invars(s).\ Clocks \vDash cond$
- $\epsilon\prime = \epsilon[updates(s)/Vars(s)]$
- $\nu\prime = \nu[resets(s)/Clocks(s)]$

3.17.0.3 Synchronized action step

Let $cuts_{syn} \subseteq (cuts \cup startcuts)$ be a set of subchart cuts with enabled synchronized action step $\overset{syn!}{\Rightarrow}$ or $\overset{syn?}{\Rightarrow}$. Let s_i be the simregion between current and next cut in any of these configurations. Guards and invariants related to the simregion are $guards(s_i)$ and $invars(s_i)$, respectively. Updates of variables and resets of clocks are $updates(s_i)$ and $resets(s_i)$. Variables and clocks referred from simregion will be denoted $Vars(s_i) \subseteq Vars$ and $Clocks(s_i) \subseteq Clocks$.

Synchronized action is allowed iff there are no other simple actions where simregion with condition is stepped over. This restriction maintains ASAP evaluation semantics of conditions.

Synchronized action step is denoted $(cuts, \epsilon, \nu) \overset{syn}{\Rightarrow} (cuts\prime, \epsilon\prime\prime, \nu\prime\prime)$ where

- $cuts\prime = cuts \cup startcuts \setminus termcuts \cup maincuts$, as specified before
- All preconditions of simregions stepped over are satisfied by the current values of variables and clocks where for all i, $isPrec(s_i) = \mathbf{tt}$:
  - $\forall cond \in guards(s_i).\ isPrec(s_i) = \mathbf{tt} \Rightarrow Vars \vDash cond$
  - $\forall cond \in invars(s_i).\ isPrec(s_i) = \mathbf{tt} \Rightarrow Clocks \vDash cond$
- Initial values of variables and clocks are updated according to the preactions of simregions stepped over where $isPrec(s_i) = isPrec(s_j) = \ldots = isPrec(s_k) = \mathbf{tt}$:
  - $\epsilon\prime = \epsilon[updates(s_i)/Vars(s_i)][updates(s_j)/Vars(s_j)]\ldots[updates(s_k)/Vars(s_k)]$
  - $\nu\prime = \nu[resets(s_i)/Clocks(s_i)][resets(s_j)/Clocks(s_j)]\ldots[resets(s_k)/Clocks(s_k)]$
- Postconditions and postactions are evaluated (performed) on the variables and clocks affected by preactions where for all i, $isPrec(s_i) = \mathbf{ff}$:
  - $\forall cond \in guards(s_i).\ isPrec(s_i) = \mathbf{ff} \Rightarrow Vars\prime \vDash cond$
  - $\forall cond \in invars(s_i).\ isPrec(s_i) = \mathbf{ff} \Rightarrow Clocks\prime \vDash cond$
- Initial values of variables and clocks are updated according to the postactions of simregions stepped over where $isPrec(s_i) = isPrec(s_j) = \ldots = isPrec(s_k) = \mathbf{ff}$:
  - $e\prime\prime = \epsilon\prime[updates(s_i)/Vars(s_i)][updates(s_j)/Vars(s_j)]\ldots[updates(s_k)/Vars(s_k)]$
  - $\nu\prime\prime = \nu[resets(s_i)/Clocks(s_i)][resets(s_j)/Clocks(s_j)]\ldots[resets(s_k)/Clocks(s_k)]$
- There is one subchart whose cut $c_i \in cuts_{syn}$ advances according legal step $c_i \overset{sync!}{\Rightarrow}$ and all other subcharts whose cuts c_j are in $cuts_{syn}$, advance according to their legal steps $c_j \overset{sync?}{\Rightarrow}$.

3.17.0.4 Delay step

Delay step is denoted $(cuts, \epsilon, \nu) \overset{d}{\Rightarrow} (cuts, \epsilon, \nu\prime)$ where

- $\nu\prime = (\nu + d)$, i.e. all clock values are increased by d.

Delay step is allowed iff all configurations from *cuts* still have a delay, action or synchronized action step enabled from $(c_i, \epsilon_i, (\nu_i + d))$ where (c_i, ϵ_i, ν_i) is their current configuration, i.e. the specified delay is a part of a well-formed trace for all configurations in $actconfs$.

Additional requirement for delay action is that there are no other simple actions from *cuts* where simregion with condition is stepped over. This restriction maintains ASAP evaluation semantics of conditions. This means, no currently active chart can wait with simregion enabled that does not include message, but does condition. However, charts from *startcuts* with minimal cuts enabling simregions of such kind can activate at any time. One cannot force to activate such chart infinitely often, as such semantics would prevent of taking actions of different kind.

Well-formed sequence of steps

A sequence of $LSCSpec$ configurations
$\{(cuts, \epsilon, \nu)\}^K = \{(cuts, \epsilon, \nu)^0, (cuts, \epsilon, \nu)^1, \ldots\}$ of length $K \in \mathbb{N}$ is called a well-formed sequence for $LSCSpec$ iff

- $LSCSpec$ starts in initial configuration which is $(\emptyset, [Vars \mapsto 0^*], [Clocks \mapsto 0^*])$
- for $(cuts, \epsilon, \upsilon)^K$, no further step is enabled

Sequence of steps satisfying the LSC specification

A well-formed sequence of steps is called a timed trace for a specification $LSCSpec$ if in addition the following holds:

- for every $k < K$, the two subsequent configurations k and $k+1$ are connected via a simple action step, a synchronized action step or a delay step, i.e.
  $(cuts, \epsilon, \nu)^k \overset{a}{\Rightarrow} (cuts, \epsilon, \nu)^{k+1}$ or $(cuts, \epsilon, \nu)^k \overset{d}{\Rightarrow} (cuts, \epsilon, \nu)^{k+1}$ or
  $(cuts, \epsilon, \nu)^k \overset{\tau}{\Rightarrow} (cuts, \epsilon, \nu)^{k+1}$.

A timed trace $TT = (Chan \cup \mathbb{R}_{\geq 0} \cup \tau)^*$ satisfies the LSC specification $LSCSpec$ if it has a well-formed prefix that is a timed trace for $LSCSpec$:

$$TT \vDash LSCSpec \Rightarrow \exists Tr \sqsubset TT.\ (\emptyset, [Vars \mapsto 0^*], [Clocks \mapsto 0^*]) \xrightarrow{Tr} (cuts, \epsilon, \nu) \tag{3.44}$$

An infinite timed trace $TT = (Chan \times \{?, !\} \cup \mathbb{R}_{\geq 0} \cup \tau)^w$ satisfies $LSCSpec$ if it has a well-formed prefix that satisfies the $LSCSpec$:

$$\begin{aligned} TT \vDash LSCSpec \Rightarrow \exists Tr \in (Chan \cup \mathbb{R}_{\geq 0} \cup \tau)^*, \exists Cont \in (Chan \cup \mathbb{R}_{\geq 0} \cup \tau)^w. \\ TT = Tr.Cont \wedge Tr \vDash LSCSpec \end{aligned} \tag{3.45}$$

3.18 Temperature of the cuts and LSC subchart constructs

Conditions in the LSC subchart are assigned the temperature property, $isHot$:

$$isHot : \mathbf{D} \rightarrow \mathbb{B} \tag{3.46}$$

Similarly to conditions, the cuts are also assigned temperature. For a set of cuts $\mathbf{C}$, relation $isHot$ is defined:

$$isHot : \mathbf{C} \rightarrow \mathbb{B} \tag{3.47}$$

Example

In figure 3.25 a LSC chart with cold and hot cuts, cold and hot conditions is presented.

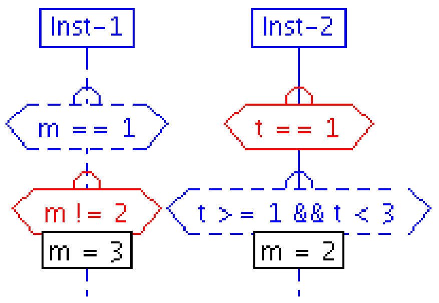

Figure 3.25: A sample LSC with cold and hot cuts, cold and hot conditions.

Let the simregions on instance $Inst - 1$ be s_1 and s_3, and on instance $Inst - 2$ be s_2 and s_4, respectively, numerating from top to bottom. Conditions of simregions s_1 and s_4 have cold temperatures (denoted by blue dashed border), while conditions of simregions s_2 and s_3 have hot temperatures (red continuous border of the condition).

Cuts of cold temperature are $\{s_2, s_4\}$, $\{s_1, s_2, s_4\}$ and $\{s_1, s_2, s_3, s_4\}$ (the maximal cut). Minimal cuts are always of hot temperature, and the maximal ones are of cold temperature. A cut is hot if at least one of enabled simregions has the solid instance line before itself. A cut is cold if all the enabled simregions have dashed instance line segments before them.

Dashed or solid instance line segment before the simregion means that the cut is not obliged (resp. obliged) to progress beyond that simregion.

3.19 Subcharts in a LSC subchart

The LSC subchart has so far been defined as preorder of locations labeled with simregions. LSC subchart is either a prechart or mainchart for the LSC chart.

Subchart construct is a subset of locations from labeled preorder L which has type of ordinary subchart, infinite loop or if-then-else construct.

The LSC subchart is itself the subchart (of ordinary subchart type) which holds all the locations from labeled preorder L.

Assume set of subcharts $\mathcal{SC}$.

The labeled preorder definition $\mathcal{L} = \langle L, \leq, \lambda, \mathbf{S}\rangle$ is extended with subcharts, i.e. $\mathcal{L} = \langle L, \leq, \lambda, \mathbf{S}, \mathcal{SC}\rangle$:

L is the set of locations,

$\leq$ is preorder relation over the locations and subcharts

$\mathbf{S}$ is the set of simregions

$\mathcal{SC} \subseteq 2^L$ is the set of subcharts

λ is the labeling function for locations such that $\lambda :\ \mathcal{L} \rightarrow \mathbf{S}$

Several new relations among the labeled preorder elements are defined.

Subcharts span over at least one instance, just like simregions:

$$\begin{gathered} insts : \mathcal{SC} \times \mathbf{I} \\ \forall sc \in \mathcal{SC}, \exists inst \in \mathbf{I} \cap insts(sc) \end{gathered} \tag{3.48}$$

All subcharts hold at least one location inside:

$$\forall sc \in \mathcal{SC}, \exists l \in L.\ l \in sc \tag{3.49}$$

Locations inside subchart are referred by *locs* relation:

$$\forall sc \in \mathcal{SC}.locs(cs) = \{l \in L | l \in sc\} \tag{3.50}$$

Locations belonging to one subchart must either belong to other subchart altogether or none of them:

$$\forall sc, sc\prime \in \mathcal{SC}.\ locs(cs) \cap locs(cs\prime) = \emptyset \vee locs(cs) \subseteq locs(cs\prime) \vee locs(cs\prime) \subseteq locs(cs) \tag{3.51}$$

Preorder relation applies among subcharts (and locations inside them) and locations outside them:

- Subchart sc_a can be placed in another subchart sc_b only when set of instances and locations of sc_a are both inside set of instances and locations of sc_b, respectively

- Preorder relation between location and subchart at the same level is defined directly by $\leq$. "Same level" means that the subchart is not inside another subchart and is in some subchart or LSC subchart together with that location
- Preorder between two locations of the same level is obvious, given by $\leq$ relation
- Preorder relation between two subcharts of the same level apply to all the locations and subcharts inside

Entered subcharts are those who hold the locations whose simregions are enabled from current cut.

Cut before subchart is the set difference of locations from current cut and locations belonging to that subchart.

Cut after subchart is the set union of locations from current cut and locations belonging to that subchart.

If-then-else subchart does not have preorder relation for its if- and else-subcharts. As soon as one of them is complete, the cut is extended with locations from both if- and else- subchart.

Example

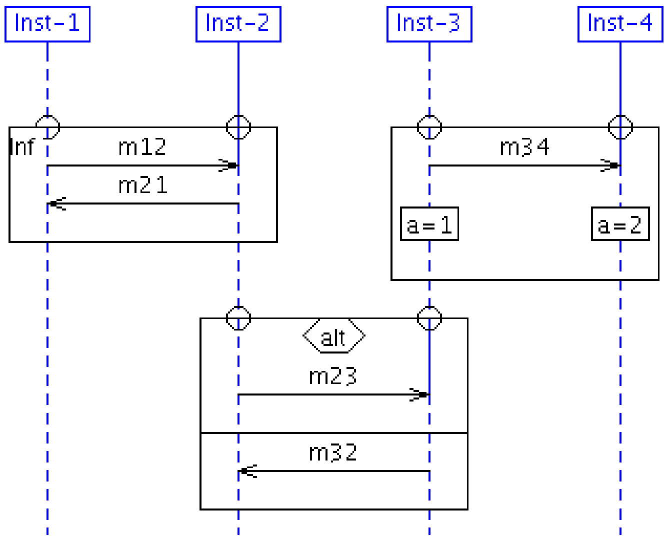

Figure 3.26: A sample LSC with three kinds of subcharts, i.e. subchart, loop and if-then-else construct. The (infinite) loop construct spans over instances $\{Inst-1, Inst-2\}$, if-then-else construct over $\{Inst-2, Inst-3\}$ and subchart over $\{Inst-3, Inst-4\}$. In the if-then-else construct, the else-construct comes after if-construct, and expression *alt* shows non-dterministic choice of execution branches, either $m23$ or $m32$.

3.20 Scope constructs

It is not always sufficient to use preorder relation to restrict out-of-order occurrence of non-enabled events in LSC chart, or trigger a violation of preorder based on the values of variables and clocks, therefore the scope constructs are introduced.

Another feature enabled by the ignored and forbidden scope constructs is the message and instance abstraction (see section 3.8). Together with the message event, some shared variables can be updated that help identify the instances or other attributes encoded in the shared variables, participating in that event.

Scope constructs are basically simregions with message construct, optionaly bundled with condition construct. There are two types of scopes: forbidden and ignored ones, and their sets are denoted $\mathbf{S_F}$ and $\mathbf{S_I}$, respectively.

Sets $\mathbf{S_F}$ and $\mathbf{S_I}$ are disjunct, and both these are subsets of simregions $\mathbf{S}$.

Scopes have a temperature property:

$$isHot : (\mathbf{S_F} \cup \mathbf{S_I}) \to \mathbb{B} \tag{3.52}$$

The labeled preorder $\mathcal{L} = \langle L, \leq, \lambda, \mathbf{S} \rangle$ now has its labeling function changed so that its labeling function for locations does not map locations to simregions of scopes: $\lambda : \mathcal{L} \to (\mathbf{S}/(\mathbf{S_F} \cup \mathbf{S_I}))$.

The cuts may have zero to several scopes assigned to them. Relation $scopes : \mathbf{C} \times (\mathbf{S_F} \cup \mathbf{S_I})$ assigns ignored and forbidden scopes to the cut, whose subsets by scope type are $forbiddenScopes : \mathbf{C} \times \mathbf{S_F}$ and $ignoredScopes : \mathbf{C} \times \mathbf{S_I}$.

3.20.0.5 Overriding the scopes

Sorted according to increasing importance, the scope constructs are

- Ignored scope without condition
- Ignored scope with condition
- Forbidden scope without condition
- Forbiden scope with condition

This ordering means, that, for example, the forbidden scopes have higher priority than ignored scopes, and that having scope with condition applying to the same event as the scope without condition, the rules of the former apply.

Example

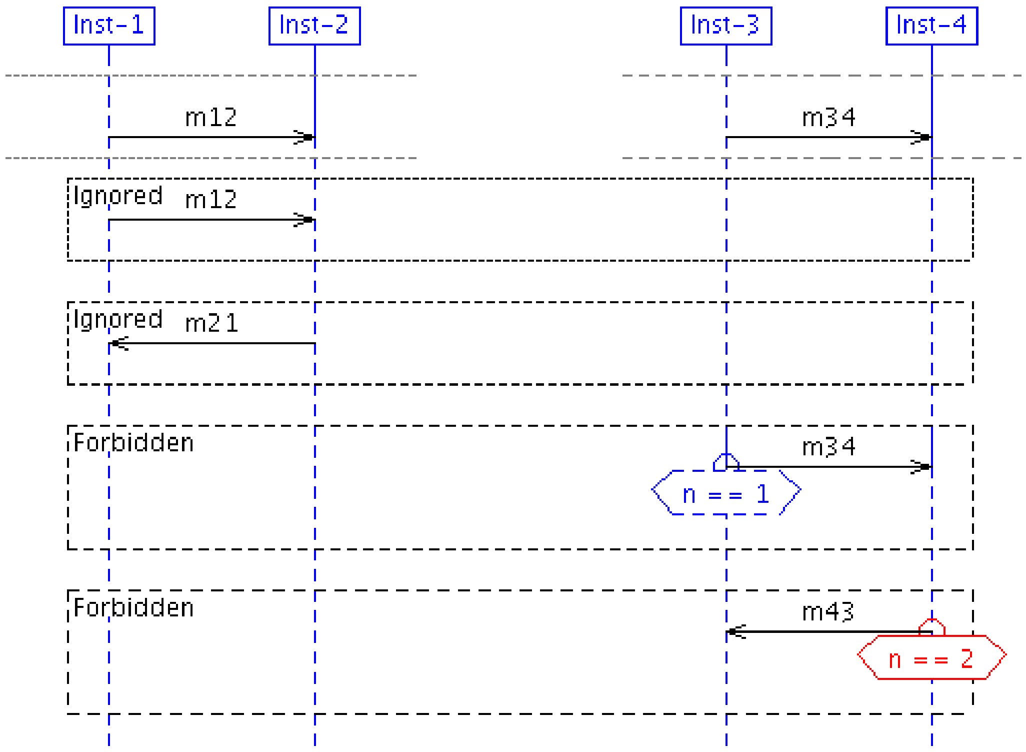

Figure 3.27: A sample LSC with different kinds of scopes applying to its cuts. The simregions of the LSC are s_1 with message $m12$ and s_2 with message $m34$. The cuts of the LSC chart are $\{\emptyset\}$, $\{s_1\}$, $\{s_2\}$ and $\{s_1, s_2\}$. To the cut $\{\emptyset\}$, all four scopes apply because they apply to at least one of enabled simregions from that cut (i.e. s_1 and s_2). To the cut $\{s_2\}$, two ignored scopes apply, and to the cut $\{s_1\}$, two forbidden scopes apply. Temperature of the scope is determined by the solid (hot) or dashed (cold) instance line segments above it. Temperature of the condition in the scope overrides the temperature of the scope.

3.20.1 Trace semantics of LSC subchart with scopes and subcharts

Scopes and subcharts add the concept of non-enabled event, ignored event and forbidden event.

As in semantics without subcharts, the only kind of steps the LSC chart can use are valid steps, where conditions etc. are satisfied. Scopes and subcharts have added more functionality to the LSC charts, namely not necessarily exiting after observing out-of-order event or evaluating condition to false.

3.20.1.1 LSC constructs influencing the temperature of violation

When a message event is observed or simregion s with message whose label is *label* is stepped over, several factors decide if the violation occurs and what kind of violation it is:

- Temperature of the current cut. Let the current cut be c, then temperature of the cut is $isHot(c)$.
- Temperature of condition in a simregion if any stepped over. Let the set of conditions in LSC subchart be $\mathbf{D}$, and $cond(s) = \{d \in \mathbf{D} | hasCond(s, d)\}$. Whenever $cond \neq \emptyset$, temperature of condition is $isHot(cond)$.
- Temperatures of the scopes applying to current cut. Scopes with same message label as non-enabled event or stepped over simregion affect, whether it will result in violation, cold violation or accepting the event without progressing the cut.
- Temperatures of conditions in scopes if any. Even if the cut or condition temperatures of the stepped-over simregion suggest that the violation is hot, the condition temperatures of the scopes might override this to cold violation. Example: a cold scope or a scope with a cold condition applying to the non-enabled event in a hot cut or a simregion with hot unsatisfied condition.

Sorted according to increasing importance, the factors deciding the temperature of violation are

- Temperature of the current cut
- Temperature of the violating condition in a simregion which is stepped over
- Applying ignored scopes without condition
- Applying ignored scopes with condition
- Applying forbidden scopes without condition
- Applying forbidden scopes with condition

Scope is said to **apply** to the message event or simregion with message, if any of the following holds:

- Scope has no condition, and label of the message in the scope coincides with the label of the message event or the one in simregion
- Scope has a condition, the label of its message coincides with that from stepped-over simregion (message), and the condition evaluates to true under current values of variables and clocks.

There are three possible outcomes of observing the non-enabled message event:

- Ignoring the event. This can happen if all the following set of predicates holds:
  1. Ignored scopes apply to the current cut that have message with same label as the message event
  2. No forbidden scopes apply to the current cut with same label
- Cold violation. This happens on any of situations like:
  - The current cut temperature is cold, no scopes apply with same label
  - any of the following holds:
    * No ignored scopes apply and forbidden scopes apply that have hot temperature or any temperature and hot condition temperature
    * Some forbidden scope(s) apply that are of cold temperature without condition or of any temperature with cold condition
- Hot violation. This happens in any of these situations:
  - The current cut temperature is hot and no scopes apply
  - Some forbidden scope(s) apply that have hot temperature or any temperature and hot condition temperature

There are four possible outcomes upon stepping over simregion with the message event whose label is *syn*:

- Successfully advancing. This can happen in two cases:
  - Transition $c \overset{syn!}{\Rightarrow}$ is taken
  - Set of requirements is satisfied:
    1. Transition $c \overset{syn?}{\Rightarrow}$ is taken
    2. Condition in the stepped-over simregion is satisfied, if any
    3. No ignored scopes with same label apply to the current cut or none of their conditions are satisfied
    4. No forbidden scopes with same label apply to the current cut or none of their conditions are satisfied
- Ignoring the event. This can happen if all the following set of predicates holds:

1. Transition $c \overset{syn?}{\Rightarrow}$ is taken
2. Ignored scopes apply to the current cut that have message with label syn
3. No forbidden scopes apply to the current cut with label syn

- Cold violation. This happens if transition $c \overset{syn?}{\Rightarrow}$ is taken and any of the following situations apply:
  - The current cut temperature is cold, no scopes apply
  - Independently from the temperature of the cut and condition of simregion, scopes apply in one of combinations:
    - No ignored scopes apply
    - No forbidden scopes apply that have hot temperature or any temperature and hot condition temperature
    - Some forbidden scope(s) apply that are of cold temperature without condition or of any temperature with cold condition
- Hot violation. This happens if transition $c \overset{syn?}{\Rightarrow}$ is taken and any of the following situations apply:
  - The current cut temperature is hot and no scopes apply
  - Some forbidden scope(s) apply that have hot temperature or any temperature and hot condition temperature

Example

In figure 3.27, the scopes cause the following outcomes:

- The uppest scope (ignored, with message $m12$) applies through instance $Inst-1$ to cuts $\{\emptyset\}$, $\{s_2\}$. Because of scope, symbol $m12?$ is ignored.
- The second uppest scope (ignored, with message $m21$) applies through instance $Inst-2$ to cuts $\{\emptyset\}$, $\{s_2\}$. Because of scope, symbol $m21?$ is ignored.
- The third scope from top (forbidden, with message $m34$ and condition $n == 1$) applies through instance $Inst-3$ to cuts $\{\emptyset\}$, $\{s_1\}$. Because of scope, symbol $m34?$ causes a cold violation (temperature of condition overrides that of the scope) when variable n is equal to one.
- The bottom scope (forbidden, with message $m34$ and condition $n == 1$) applies through instance $Inst-3$ to cuts $\{\emptyset\}$, $\{s_1\}$. Because of scope, symbol $m43?$ causes a hot violation (temperature of condition overrides that of the scope) when variable n is equal to two.

3.20.1.2 Effect of scopes and subcharts on the transition

In case of cold violation, the cut advances beyond the subcharts who span over the instances of simregion with cold failing condition or non-enabled event if no scopes applied, or maximal set of instances of the scope(s) who caused the cold violation. If a cold violation occurs inside an If- or Else- subchart of If-Then-Else structure, the cut is advanced beyond the whole innest If-Then-Else.

In case of success when the cut advances in a loop and stepped-over simregion union with passed simregions include all simregions belonging to the loop, the cut is placed at the beginning of that loop.

In case of success where simregions If- or Else- subchart are passed, the simregions of whole If-Then-Else structure are included in the cut.

Example

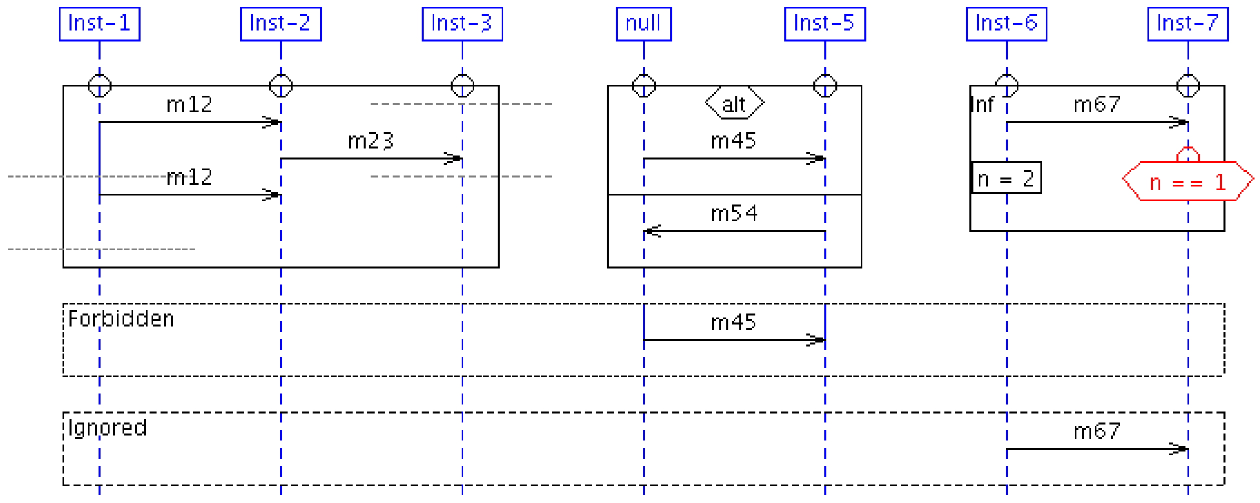

Figure 3.28: A sample LSC with scopes and subcharts.

In figure 3.28, the sample LSC with the subcharts and scopes is displayed. Simregions in the LSC are s_1 (the upper message $m12$ in subchart over instances $Inst-1$, $Inst-2$, $Inst-3$), s_2 (the message $m23$ in subchart over instances $Inst - 1$, $Inst - 2$, $Inst - 3$), s_3 (the lower message $m12$ in subchart over instances $Inst-1$, $Inst-2$, $Inst-3$), s_4 (the upper message $m45$ in if-then-else over instances $Inst-4$, $Inst-5$), s_5 (the upper message $m54$ in if-then-else over instances $Inst - 4$, $Inst - 5$), s_6 (the upper message $m67$ in subchart over instances $Inst - 6$, $Inst-7$), s_7 (action at instance $Inst-6$ in loop over instances $Inst-6$, $Inst-7$), s_8 (condition at instance $Inst - 7$ in loop over instances $Inst - 6$, $Inst - 7$).

Forbidden scope causes hot violation upon transition $\overset{m43?}{\Rightarrow}$ from cuts including s_1 and not s_3. Ignored scope makes the LSC ignore the events $m67?$ from cuts which do not include s_2.

From cut including simregions s_6 and s_8, $\overset{\tau}{\Rightarrow}$ with stepping over simregion s_7 results in cut where s_6, s_7 and s_8 are excluded, i.e. upon completion of the loop, the cut is returned to the beginning of the loop.

The transition where s_4 or s_5 are stepped over, adds both s_4 and s_5 to the cut. This is how an if-then-else structure works: upon completing one branch, simregions from another one are added to the cut automatically.

Chapter 4

UPPAAL

This chapter is about UppAal tool. Example of the same Smart Lamp model in section 4.1 precedes the formal semantics in section 4.2.

The UppAal is an integrated tool environment for modeling, validation and verification of real-time systems modeled as the networks of the timed automata, extended with the data types as integers, booleans, arrays, records etc.

Motivation for choosing the UppAal timed automata format as the destination formalism to translate the Live Sequence charts is manyfold. First of all, the UppAal tool environment has proved itself for debugging and modeling the specifications of communication systems. It also deals with the real-time, which is not supported by the known LSC semantics based tools yet. Last but not least, the real-time on-line test tool UppAal TRON is based on the UppAal engine, and supports same syntax. Therefore, smooth and convenient manipulation of the translated LSC specifications has been enabled by the tool family.

As the Aalborg University CS Department has access to the UppAal tool source code, there are opportunities to build entirely LSC based tool environment based on UppAal.

4.1 Example of the model in UPPAAL - Smart Lamp

The Smart Lamp model has been defined by means of LSC. In this section, same model is presented, how it is defined in UppAal.

4.1.1 Actors

In UppAal, it is convenient to dedicate one timed automaton per actor (process). Therefore, having same actors identified as in section 3.2.1.1, one timed automaton is defined for each.

User

The *User* automaton is displayed in figure 4.1. It consists of two locations connected with transitions in a circle. The left location is initial (with the circle inside). Upon initializing the *grasp*! synchronization, the automaton transits to its another state, from where it returns to the initial one by initializing the *release*! synchronization.

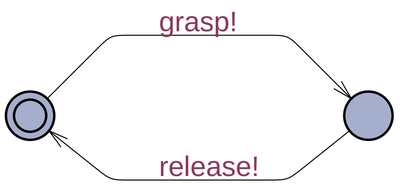

Figure 4.1: The *User* process modeled as the UppAal timed automaton.

Wire

The *Wire* timed automaton displayed in figure 4.2 is the interface between the *User* and *Switch* with *Dimmer*. Its initial location *idle* corresponds to the situation where no user interaction has been observed for a while. When the user grasps the wire, transition with *grasp*? is taken, clock x reset and automaton transits to state *ignoring*. Then, depending on how soon the user releases the wire, three outcomes are available:

- The *Wire* is released too fast (the clock x value being less than $epsilon + tolerance$ time units, as opposed to $eps + tau$ in LSC model), and the time interval between grasp and release qualifies neither for the touch nor for the hold events. Such sequence of events is treated as noise, and the automaton is returned to its initial location.

- *release* is observed within $[epsilon, delta + tolerance]$ time units from *grasp*, making automaton transit to *touching* location, and it qualifies for the *touch* event (automaton transiting to *touched* location upon *release* and initializing the *touch*! synchronization upon transiting to initial location within *tolerance* time units from observing the *release*).

- *release* is not observed within *delta* time units from *grasp*. Then, the *starthold* synchronization is initiated within *tolerance* time units, the automaton transiting to the *holding* location, and it is waited for *release* event, which is then followed by the *endhold* synchronization and the automaton returns to its initial state.

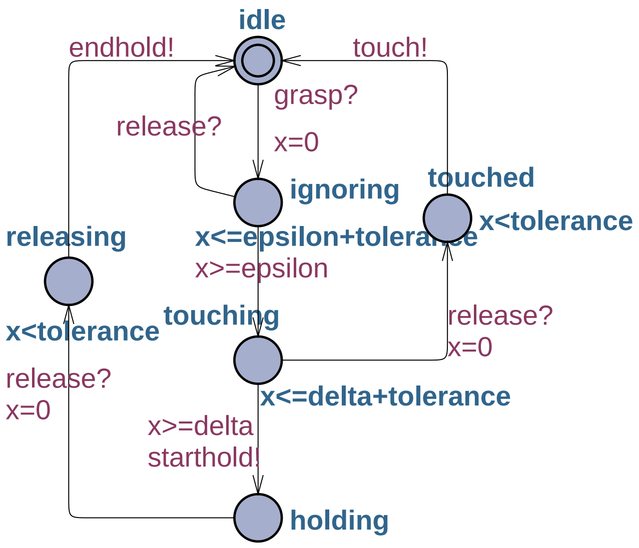

Figure 4.2: The *Wire* process modeled as the UppAal timed automaton.

Dimmer

The *Dimmer* automaton (figure 4.3) consists of four locations, two committed (with letter "c" inside) whom the automaton must leave immediately, and two locations where the time can pass. The initial location *Idle* corresponds to the dimmer when deactivated. Upon the *starthold* event, transition is taken to the committed location, and the lamp state *on* is set to 1 and brightness level L is restored from the last saved brightness level OL. Then, transition is taken to non-committed location *Active* where the *setlevel* synchronization is supposed to update the lamp brightness with values set in the previous transition.

The *Active* location has the invariant on it, forcing to take the transition to committed location and back every $[delay, delay+tolerance]$ time units. During such a loop, the brightness is recalculated and updated.

When the user releases the wire, the *endhold* event is generated, and the dimmer is returned into its initial location, saving the newest value of brightness in variable OL.

Switch

Purpose of the *Switch* process (figure 4.4) is to toggle the lamp state and the brightness upon the *switch* event. Two possible sets of the lamp state *on* with brightness value L are $\{0, 0\}$ and $\{1, OL\}$, respectively. The change of these is performed from initial location upon the *touch*

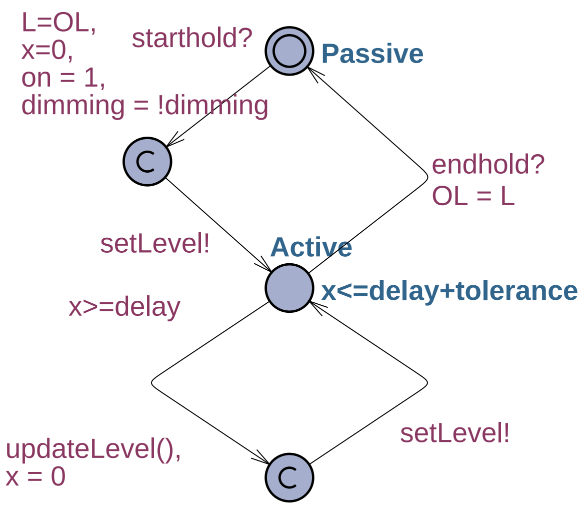

Figure 4.3: the *Dimmer* process modeled as the UppAal timed automaton.

event, and transition to the locations $goingOff$ or $goingOn$ is chosen dependent on the current lamp state on.

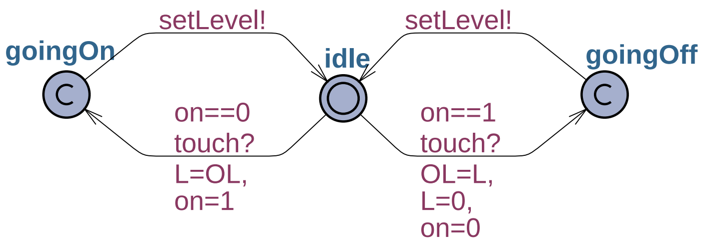

Figure 4.4: The *Switch* process modeled as the UppAal timed automaton.

Lamp

The Lamp automaton is not modeled explicitly, its brightness and state is set in the $Wire$ and $Switch$ processes.

4.1.2 Requirements

Requirements from section 3.2.2 have actually been derived from the standalone behavior and interactions of the automata presented in the section 4.1.1. The requirements were derived by trying to cover all the possible interactions of the automata. The smart lamp UppAal model being a small model whose core consists of four automata, it has been easy to cover the interactions. For larger models, defining requirements directly in LSC is more efficient instead.

4.2 The timed automata model of UPPAAL

UppAal is a toolbox for modeling, verification and simulation of real-time systems jointly developed by Uppsala University and Aalborg University.

Modeling language used in UppAal is enriched dialect of timed automaton formalism, maintaining real-timed clicks and finite control structure.

4.2.1 Basic definitions

Basic definitions of the syntax and semantics of the timed automata is given in this section. The set of clocks is denoted by C, and by $B(C)$ is denoted the set of conjunctions over simple conditions of the form $x \bowtie c$ or $x - y \bowtie c$ where $x, y \in C$, $c \in \mathbb{N}_0$ and $\bowtie \in \{<, \leq, =, \geq, >\}$. A timed automaton is a finite directed graph annotated with conditions over and resets of non-negative real valued clocks.

Definition 4.2.1 *A timed automaton is a tuple (L, l_0, C, A, E, I) where L is a set of locations, $l_0 \in L$ is the initial location, C is the set of clocks, A is a set of actions, co-actions and the internal τ-action, $E \subseteq L \times A \times B(C) \times 2^C \times L$ is a set of edges between locations with an action, a guard and a set of clocks to be reset, and $I : L \to B(C)$ assigns invariants to locations.*

A clock valuation is the function $u : c \to \mathbb{R}_{\geq 0}$ from the set of clocks to the non-negative reals. Let $\mathbb{R}^C$ be the set of all clock valuations. Let $u_0(x) = 0$ for all $x \in C$. Guards and invariants will be considered as clock valuations, and notation $u \in I(l)$ will mean that u satisfies $I(l)$.

Definition 4.2.2 *Let (L, l_0, C, A, E, I) be a timed automaton. Its semantics is defined as a labeled transition system $\langle S, s_0, \to \rangle$ where $S \subseteq L \times \mathbb{R}^C$ is the set of states, $s_0 = (l_0, u_0)$ is the initial state, and $\to \subseteq S \times \{\mathbb{R}_{\geq 0} \cup A\} \times S$ is the transition relation such that:*

- $(l, u) \xrightarrow{d} (l, u + d)$ *if* $\forall d\prime : 0 \leq d\prime \leq d \Longrightarrow u + d\prime \in I(l)$
- $(l, u) \xrightarrow{a} (l\prime, u\prime)$ *if there exists* $e = (l, a, g, r, l\prime) \in E$ *s.t.* $u \in g, u\prime \in [r \mapsto 0]u$, *and* $u\prime \in I(l)$

where for $d \in \mathbb{R}_{\geq 0}$, $u + d$ maps each clock $x \in C$ to the value $u(x) + d$, and $[r \mapsto 0]u$ denotes the clock valuation which maps each clock in r to 0 and agrees with u over $C \setminus r$.

Timed automata are often composed into *a network of timed automata* over a common set of clocks and actions consisting of n timed automata $\mathcal{A}_i = (L_i, l_i^0, C, A, E_i, I_i), 1 \leq i \leq n$. A location vector is $\bar{l} = \{l_1, \ldots, l_n\}$. The invariant functions are composed into a common function over location vectors $I(\bar{l}) = \wedge_i I_i(l_i)$. The location vector $\bar{l}$ with its ith element l_i replaced with $l_i\prime$ will be denoted $\bar{l}[l_i\prime/l_i]$.

Definition 4.2.3 *Let $\mathcal{A}_i = (L_i, l_i^0, C, A, E_i, I_i)$ be a network of n timed automata. Let $\bar{l}_0 = (l_1^0, \ldots, l_n^0)$ be initial location vector. The semantics is defined as a transition system $\langle S, s_0, \rightarrow\rangle$, where $S = (L_1 \times L_2 \times \ldots \times L_n) \times \mathbb{R}^C$ is the set of states, $s_0 = (\bar{l}_0, u_0)$ is the initial state, and $\rightarrow \subseteq S \times S$ the transition relation is defined by*

- $(\bar{l}, u) \rightarrow (\bar{l}, u + d)$ *if* $\forall d\prime : 0 \leq d\prime \leq d \Longrightarrow u + d\prime \in I(\bar{l})$
- $(\bar{l}, u) \rightarrow (\bar{l}[l_i\prime/l_i], u\prime)$ *if there exists* $l_i \overset{\tau g r}{\rightarrow} l_i\prime$ *s.t.* $u \in g$, $u\prime = [r \mapsto 0]u$ *and* $u\prime \in I(\bar{l})$
- $(\bar{l}, u) \rightarrow (\bar{l}[l_j\prime/l_j, l_i\prime/l_i], u\prime)$ *if there exist* $l_i \overset{c?gr}{\rightarrow} l_i\prime$ *and* $l_j \overset{c!gr}{\rightarrow} l_j\prime$ *such that* $u \in (g_i \wedge g_j)$, $u\prime = [r_i \cup r_j \mapsto 0]u$ *and* $u\prime \in I(\bar{l})$.

The following features in timed automata language supported by UppAal are used to maintain the LSC semantics:

- Tempplates automata are defined with arbitrary set of parameters with the types of integer, chan or clock. These parameters are substituted for a given argument in the process declaration.
- Broadcast channels are used in broadcast synchronization such that the sender $c!$ can synchronize with arbitrary number of receivers $c?$. Any receiver who is able to synchronize in its current state must do so. the $c!$ action can be performed also without receivers (it is non-blocking).
- Committed locations are defined as a subset of locations. No delay steps are allowed from the state whose vector includes committed locations. The outgoing transition must originate from the committed state in at least one automaton.
- Templates, discrete variables and constants are used. They reduce amount of locations in timed automata and make them convenient to read.

4.2.1.1 Expressions in UPPAAL

Expressions in UppAal range over clocks and integer variables. The expressions are used with the following labels:

Guard is an expression that is side-effect free, evaluates to a boolean, has only clocks, integer variables and constants referenced; clocks and clock differences are compared only to integer expressions; guards over clocks are essentially conjunctions although expressions over integers can be in disjunction.

Synchronization label is either of the form *Expression*! or *Expression*? or an empty label. The expression must be side-effect free, evaluate to a channel, and only refer to integers, constants and channels.

Assignment is a comma-separaed list of expressions with a side effect; expressions must only refer to clocks, integer variables, and constants and only assign integer values to clocks.

Invariant is an expression that is side effect free, references only clocks, constants and integer variables, is a conjunction of conditions of the form $x < e$ or $x \leq e$ whenever x is a clock and e evaluates to integer.

UppAal model

An UppAal model is a tuple $\langle \vec{A}, Vars, Clocks, Chan, Type \rangle$ where

- $\vec{A}$ is a vector of processes $A_1, A_2, \ldots, A_n$. Parts of a process A_i will be referred to as the indexed notations L_i, l_i^0, T_i.
- $Vars$ is a set of integer variables available defined in the model. It is an union of all $Vars_i$ from processes in the NTA
- $Clocks$ is a set of clocks such that $Clocks \cap Vars = \emptyset$. Like $Vars$, $Clocks$ is the union of all $Clocks_i$ in the processes constituting the NTA
- $Type$ is a type function extending the type of locations to ordinary and committed ones.

Configuration

Configuration of an UppAal model $\langle \vec{A}, Vars, Clocks, Chan, Type \rangle$ is a triple $(\vec{l}, e, v)$ where $\vec{l}$ is a vector of locations, e is the environment for discrete variables and v is a clock evaluation:

- $\vec{l} = (l_1, l_2, \ldots, l_n)$ where $l_i \in L_i$ is current location of process A_i
- $e : Vars \to (\mathbb{Z})^*$ maps every variable v to its integer value
- $v : Clocks \to \mathbb{R}_{\geq 0}$ maps the clocks to non-negative numbers.

Configuration of the model corresponds to the state in definition of the NTA semantics.

The vector $\vec{l}$ is called situation (the currently occupied states in the automata of the model), pair $(\vec{l}, e)$ discrete part and v the continuous part.

4.2.2 Semantics of the UPPAAL model

UppAal models evolve through legal steps, either actions or delays. All the legal steps define behavior of the model. For configuration $(\vec{l}, e, v)$ a simple action is enabled if there is a τ-transition in the underlying NTA. A synchronized action step is *enabled* iff for a channel b there exists the binary synchronization transition in the underlying NTA.S A synchronized broadcast action step is *enabled* iff for a channel b there exists the broadcast synchronization transition in the underlying NTA. A delay step with delay d is *enabled* iff such delay step is allowed in the underlying NTA.

4.2.3 Well-formed sequence / timed trace

Let $M = \langle \vec{A}, Vars, Clocks, Chan, Type \rangle$ be a UppAal model. A sequence of configurations $\{(\vec{l}, e, \upsilon)\}^K = \{(\vec{l}, e, \upsilon)^0, (\vec{l}, e, \upsilon)^1, \ldots\}$ of length $K \in \mathbb{N} \cup \infty$ is called a well-formed sequence for M iff

- $(\vec{l}, e, \upsilon)^0 = ((l_1^0, \ldots, l_n^0), [vars \mapsto (0)^*], [Clocks \mapsto (0)^*])$
- if $K < \infty$, then for $(\vec{l}, e, \upsilon)^K$ no further step is enabled
- if $K = \infty$ and $(\vec{l}, e, \upsilon)^K$ contains finitely many k such that $(\vec{l^k}, e^k) \neq (\vec{l^{k+1}}, e^{k+1})$, then eventually every clock exceeds every bound $(\forall x \in Clocks \forall c \in \mathbb{N}.\ \exists k.\ \upsilon^k(x) > c)$.

A well-formed sequence for M is called a timed trace for M if in addition the following holds:

- for every $k < K$, the two subsequent configurations k and $k + 1$ are connected via a simple action step, a synchronized action step, synchronized broadcast action step or a delay step, i.e.
$(\vec{l}, e, \upsilon)^k \overset{a}{\Rightarrow} (\vec{l}, e, \upsilon)^{k+1}$ or $(\vec{l}, e, \upsilon)^k \overset{d}{\Rightarrow} (\vec{l}, e, \upsilon)^{k+1}$ or $(\vec{l}, e, \upsilon)^k \overset{\tau}{\Rightarrow} (\vec{l}, e, \upsilon)^{k+1}$.

Trace semantics

Let M be a UppAal model. Then the trace semantics of M, denoted $\mathcal{T}(M)$, is the set of well-formed traces.

4.2.4 The TCTL subset maintained by UPPAAL

There is a subset of *timed computation tree logic* (TCTL) [ACD93] maintained by UppAal tool. The primitive expressions can include location names, variables and clocks from the modeled system.

4.2.4.1 Local properties

A local property is a condition that for specific configuration is either **true** or **false**. Condition can involve locations, variables and clocks which are only compared to integer values.

Definition 4.2.4 (Local Property)
Given the UppAal *model* $\langle \vec{A}, Vars, Clocks, Chan, Type \rangle$. *A formula* ϕ *is* a local property *iff it is formed according to the following rules:*

$$
\begin{array}{rll}
\phi ::= & \mathit{deadlock} & \\
 & \mid A.l & A \in \vec{A},\ l \in L_A \\
 & \mid x \bowtie c & x \in \mathit{Clocks},\ \ \bowtie \in \{<, \leq, ==, \geq, >\},\ c \in \mathbb{Z} \\
 & \mid x - y \bowtie c & x, y \in \mathit{Clocks},\ \ \bowtie \in \{<, \leq, ==, \geq, >\},\ c \in \mathbb{Z} \\
 & \mid a \bowtie b & a, b \in \mathit{Vars} \cup \mathbb{Z},\ \ \bowtie \in \{<, \leq, ==, \geq, >\},\ c \in \mathbb{Z} \\
 & \mid (\phi_1) & \phi_1 \ \mathit{a\ local\ property} \\
 & \mid \mathit{not}\ \phi_1 & \phi_1 \ \mathit{a\ local\ property} \\
 & \mid \phi_1\ \mathit{or}\ \phi_2 & \phi_1, \phi_2 \ \mathit{local\ properties} \\
 & \mid \phi_1\ \mathit{and}\ \phi_2 & \phi_1, \phi_2 \ \mathit{local\ properties} \\
 & \mid \phi_1\ \mathit{imply}\ \phi_2 & \phi_1, \phi_2 \ \mathit{local\ properties}
\end{array}
$$

The truth value of a local property can be efficiently evaluated in a configuration s.

Definition 4.2.5 (Validity of a Local Property)
A local property ϕ is valid in configurations $s \in (\vec{l}, e, \upsilon)$, denoted $s \vDash_{loc} \phi$, iff it is valid according to the following structural definitions:

$$
\begin{array}{ll}
s \vDash_{loc} \mathit{deadlock} & \mathit{iff\ no\ delay\ or\ action\ steps\ enabled\ in}\ s \\
s \vDash_{loc} A.l & \mathit{iff}\ l = l_i \in \vec{l}\ \mathit{for}\ A = A_i \in \vec{A} \\
s \vDash_{loc} x \bowtie c & \mathit{iff}\ \upsilon(x) \bowtie c, \bowtie \in \{<, \leq, ==, \geq, >\} \\
s \vDash_{loc} x - y \bowtie c & \mathit{iff}\ \upsilon(x - y) \bowtie c, \bowtie \in \{<, \leq, ==, \geq, >\} \\
s \vDash_{loc} a \bowtie b & \mathit{iff}\ e(a) \bowtie e(b), \bowtie \in \{<, \leq, ==, \geq, >\} \\
s \vDash_{loc} (\phi_1) & \mathit{iff}\ s \vDash_{loc} \phi_1 \\
s \vDash_{loc} \mathit{not}\ \phi_1 & \mathit{iff}\ \neg(s \vDash_{loc} \phi_1) \\
s \vDash_{loc} \phi_1\ \mathit{or}\ \phi_2 & \mathit{iff}\ s \vDash_{loc} \phi_1\ \mathit{or}\ s \vDash_{loc} \phi_2 \\
s \vDash_{loc} \phi_1\ \mathit{and}\ \phi_2 & \mathit{iff}\ s \vDash_{loc} \phi_1\ \mathit{and}\ s \vDash_{loc} \phi_2 \\
s \vDash_{loc} \phi_1\ \mathit{imply}\ \phi_2 & \mathit{iff}\ \neg(s \vDash_{loc} \phi_1)\ \mathit{or}\ s \vDash_{loc} \phi_2
\end{array}
$$

4.2.4.2 Temporal properties

There are five cases of temporal properties that UppAal can effectively verify. Validity of these temporal properties is via the trace semantics.

Definition 4.2.6 (Temporal properties)
Let $M = \langle \vec{A}, Vars, Clocks, Chan, Type \rangle$ be an UppAal model. Let ϕ and ψ be the local properties. Validity of temporal properties is defined for the classes $A\Box$, $A\diamond$, $\longrightarrow$ as follows:

- $M \vDash A\Box\phi$ iff $\forall\{(\vec{l}, e, \upsilon)\}^K \in \mathcal{T}(M).\ \forall k \leq K.\ (l, e, \upsilon)^k \vDash_{loc} \phi$

- $M \vDash A \diamond \phi$ iff $\forall\{(\vec{l}, e, \upsilon)\}^K \in \mathcal{T}(M).\ \exists k \leq K.\ (l, e, \upsilon)^k \vDash_{loc} \phi$

- $M \vDash \phi \longrightarrow \psi$ iff $\forall\{(\vec{l}, e, \upsilon)\}^K \in \mathcal{T}(M).\ \forall k \leq K.\ (l, e, \upsilon)^k \vDash_{loc} \phi \Rightarrow$
$\exists k\prime \geq k.\ (l, e, \upsilon)^{k\prime} \vDash_{loc} \psi$

Two temporal property classes dual to $A\Box$ and $A\Diamond$ are defined as follows

- $M \models E \Diamond \phi$ iff $\neg(M \models A\Box \ not \ \phi)$
- $M \models E\Box\phi$ iff $\neg(M \models A\Diamond \ not \ \phi)$

Chapter 5

Translation from LSC to UPPAAL

The formal semantics of the Live Sequence Charts and UppAal has been defined in chapters 3 and 4, respectively. Translation from the former to the latter is introduced through an example in section 5.1. Correspondence of the semantics is proven through the step and configuration correspondence in section 5.2.

5.1 Translation from LSC chart to UPPAAL TA

In this section, the full translation procedure will be described from the LSC chart to UppAal automaton. As sample LSC chart from Smart Lamp specification will be taken, namely figure 3.9 in Section 3.2. The translation for the LSC chart to the UppAal TA is performed in several steps:

1. Preorder $\leq$ among the simregions of LSC elements and subcharts is determined.
2. The object tree is built of the simregions and subcharts, where the simregions stand for leaves, subcharts with prechart stand for branches
3. The valid execution paths are calculated by traversing the object tree according to the preorder relation among its constructs. Result of traversal is stored by means of possible cuts and transitions between these cuts.
4. Cuts that have same set of simregions but different sets of subcharts, are aggregated into the cut groups
5. Sets of simregions are determined that connect cut from one cut group to cut in another cut group
6. UppAal TA template is built based on the information about the cut groups and simregions between these groups. UppAal locations and transitions are generated for the elements of these sets.

5.1.1 Determining the preorder among the LSC elements

The LSC elements (excluding subcharts and prechart) and simregions thereof are placed into the instances of container class, $Token$. The subchart components are represented by several instances of $Token$, an instance representing entry point, exit point or middle separator (for if-then-else construct). Token list is contained in the class $TokenNwk$.

The following tokens have been identified, as shown in the figure 5.1:

1. The top of the prechart
2. The simregion in the prechart
3. The bottom of the prechart
4. The simregion in the mainchart
5. The top of the first scope
6. One message inside the first scope
7. Second mesage inside the first scope

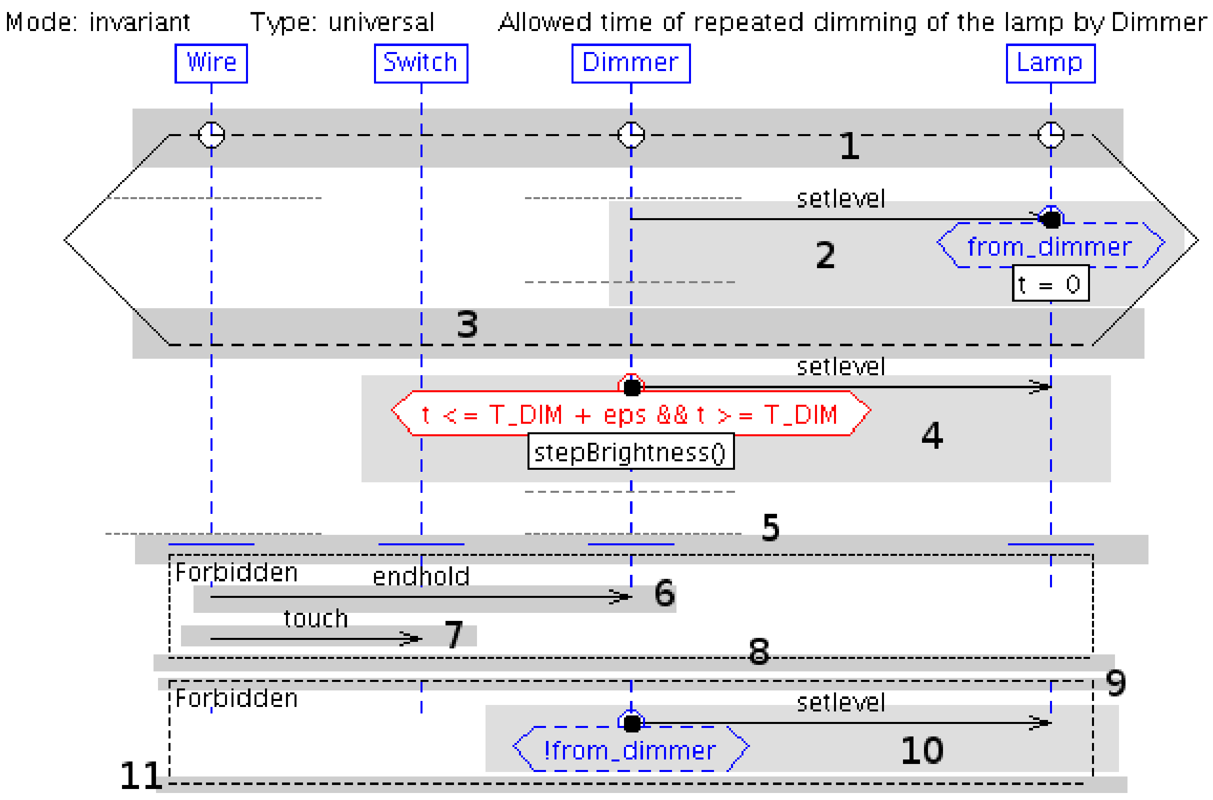

Figure 5.1: LSC chart from figure 3.9, Section 3.2 with its constructs analyzed as tokens. The tokens are depicted as the grayish rectangles with the number specified at them. Eleven tokens are identified, that mark the start and end of subcharts and simregions.

8. Bottom of the first scope

9. Top of the second scope

10. Simregion in the second scopetaken

11. Bottom of the second scope

5.1.2 Building the object tree of the LSC constructs

The object tree is an instance of the class *SubLSCNwk*. It consists of the objects that are instances of the class *SubLSC* (branches or leaves) and include tokens or other instances of *SubLSC*. This step enables the view of LSC chart as the set of containers (subcharts) with their own preorder relation and preorder relation of their contained objects. Same preorder relation is preserved as described in section 3.19.

The construction of the *SubLSC* tree is represented in figure 5.2. The *SubLSC* instance of prechart aggregates that of the simregion inside, instance of simregion inside mainchart is at the same level as the one of prechart (belongs to the same top *SubLSC* instance). The scope *SubLSC* instances are stored separately from the top *SubLSC* instance, but they behave

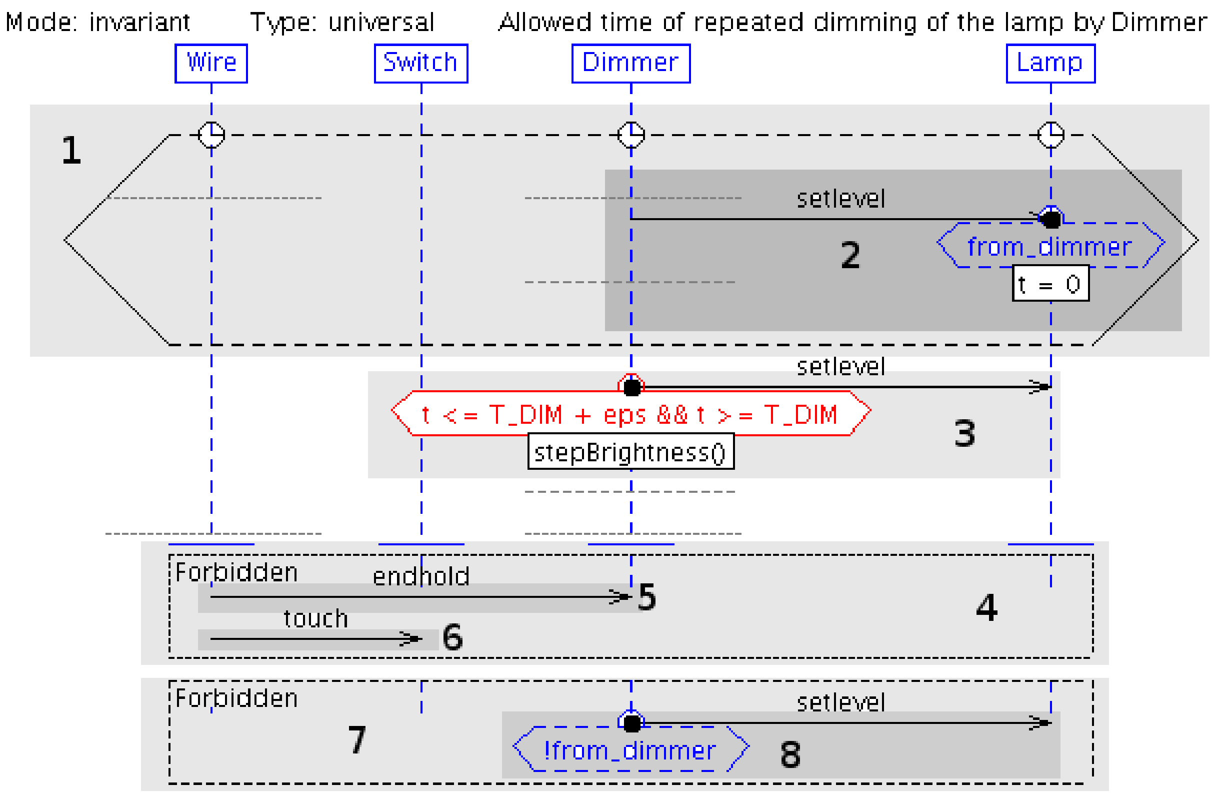

Figure 5.2: LSC chart from figure 5.1 with tokens aggregated inside subcharts and scopes. All the *subLSC* instances represented in grayish rectangles, are aggregated by the global *subLSC* instance. It includes the prechart *subLSCs* with its simregion *subLSC*, *subLSC* of simregion inside mainchart and those of scopes with their messages and simregions inside.

themselves like the top *SubLSC* instance, by containing the instances of the messages and simregions inside.

The top *SubLSC* of the LSC chart structure is as follows:

- the top instance, usually assigned ID = 0, spans over the whole chart
  - Instance of the prechart, ID = 1
    - Instance of the simregion inside the prechart, ID = 2
  - Instance of the simregion inside the mainchart, ID = 3

The *SubLSC* structure of the first scope is

- instance of the forbidden scope, ID = 4
  - instance of the first message in the scope, ID = 5
  - instance of the second message in the scope, ID = 6

The *SubLSC* structure of the second scope is

- instance of the forbidden scope, ID = 7
  - instance of the message in the scope, ID = 8

5.1.3 Finding the valid execution paths, cuts and progresses

Instance of *SubLSCNwk* in this phase is used to obtain set of possible cuts of class *SubCut* and simregions that connect such cuts, of class *CutPogress*, that constitute the execution graph of the LSC chart. Cuts and progresses are contained in the *SubCutNwk* class.

Instances of classes *SubCut* and *CutProgress* are constructed by respecting the pre-order relation $\leq$. Cuts that involve simregions and subchart entry (exit) points from *SubCut* correspond to sets of passed cuts. Progresses from *CutProgress* correspond to simregions or subchart entry (exit) points. Set of *CutProgress* and *SubCut* constitute the directed LSC graph. The maximal cut and the minimal cut of the LSC chart correspond to the single location in such graph.

The *SubCut* instances for the LSC chart with *SubLSC* instances presented in figure 5.2 would include the following IDs of the *SubLSC*:

1. $\{\emptyset\}$
2. $\{1\}$
3. $\{ 1, 2\}$
4. $\{ 1, 2, 3\}$

The cuts would have the non-enabled events assigned to them, namely

1. $\{\emptyset\}$ having non-enabled events as defined in *SubLSC* ID $=\{2, 3\}$. No scopes whose IDs are 5, 6, 8, apply to that cut, since the cut does not fall between the cuts specifying the validity of corresponding scopes.
2. $\{1\}$ having enabled event as in *SubLSC* ID = 2 and non-enabled events as defined in *SubLSC* ID $=\{3, 5, 6, 8\}$, because the scopes now apply to particular cut. Note that *SubLSC* with ID = 8 has its condition $!from\_dimmer$ supplement to that at the *SubLSC* with ID = 1, which is $from\_dimmer$.
3. $\{ 1, 2\}$ has enabled event in *SubLSC* ID = 3 and non-enabled events as defined in *SubLSC* ID $=\{2, 5, 6, 8\}$.
4. $\{ 1, 2, 3\}$ has no more enabled events, it denotes the successful completion of the chart when condition in simregion of *SubLSC* with ID = 3 is satisfied. Violation of that hot condition would result in the chart violation.

5.1.4 Aggregating the cuts into the cut groups

Representation of the LSC graph as the collection of *SubCut* and *CutProgress* instances is necessary because of the subchart semantics, but too detail. Each cut group consists of the cuts with same set of simregions and arbitrary sets of subchart entry and exit points. Since subcharts can be entered and exited without performing any action, only one representative cut is chosen from the cut group which has all valid subcharts entered or exited, and its all enabled elements are simregions.

The cuts whose all successors are accessible through simregions, will be called stable cuts.

Class of *AbstractLSCGraph* is introduced which contains groups of cuts in instances of the class *CutGroup*. Each group of cuts have one stable cut and arbitrary amount of its unstable predecessors (those who do not include enabled subchart entry or exit points). All the cuts in the cut group have same set of the terminal LSC structures in their histories.

Stable cuts in the example are the cuts that include *SubLSC* instances with IDs $\{1\}$, $\{1, 2\}$ and $\{1, 2, 3\}$. The one with IDs $= \{\emptyset\}$ is unstable because it has not entered the prechart (ID $= 1$) yet.

5.1.5 Finding the connecting events among the cut groups

The *AbstractLSCGraph* instance stands for the directed graph, whose vertices are the cut groups, and the edges are *CutProgress* instances.

Multiple edges can connect two cut groups, and some of them can include non-simregions. For example, the if-then-else construct with one of its branch empty allows non-deterministic execution of the constructs in the non-empty branch. In abstract LSC graph, this is represented in two paths between corresponding cut groups, one through terminals in the non-empty if-then-else branch, another through an empty branch.

Because of empty branches, the cut groups may have several other groups accessible directly, what would not be obvious when analyzing the graph represented by the *SubcutNwk* instance.

In our example, transitions from one cut group to another happen through stepping over the *SubLSC* instances with IDs $= 2$ and 3, respectively. No branching or preorder with several enabled events from one cut group are present in particular chart.

5.1.6 Constructing the TA from the abstract LSC graph

The UppAal TA is generated from the abstract LSC graph. Each stable cut group is assigned a UppAal location, and the edges with simregions are translated into transitions connecting those locations.

Depending on the semantics implemented, the transitions, non-enabled, forbidden and ignored events can be represented in several ways: as a single transition or two transitions with an additional location in the middle. Details of this step depends on particular translation.

The translated chart can be seen in figure 5.3. The locations corresponding to the cuts are the initial location labeled $INIT$ and the non-committed location labeled $Cut20$. Labeling the rest of locations has the following conventions:

P letter in the committed location label means that execution path of performing the non-violating owned event goes through the path involving that location, for example, triggering the message or performing silent action like successful evaluation of condition or executing the assignment

p letter means same as P, just non-owned event, like matching the message event triggered in another chart, with no violations raising from that matching

L means looping, in particular when ignored message is recognized. From so labeled location, execution typically returns back to the location where it was before entering the location labeled with L.

F letter in the committed location label means that execution path of performing the violating owned event goes through the path involving that location, for example, triggering the message with violated postcondition or evaluating standalone condition to false

f letter means violation occurring from non-owned event, like matching the message event triggered in another chart, with violation raising from that matching

The middle number shows the cut ID where the path originates. The source cut may be non-trivial to deduce because of possible if-then-else constructs with empty branches, what makes enabled some events that are non-enabled from current cut.

The final number numerates the outcomes from the same cut. Typically, there are passing outcomes, looping outcomes, failing outcomes that are caused by failing conditions and failing outcomes caused by non-enabled events.

The labeling of the stable UppAal locations can be replaced with the cut representation, where the location label includes a sequence of integers, each standing for the cut position on particular instance line.

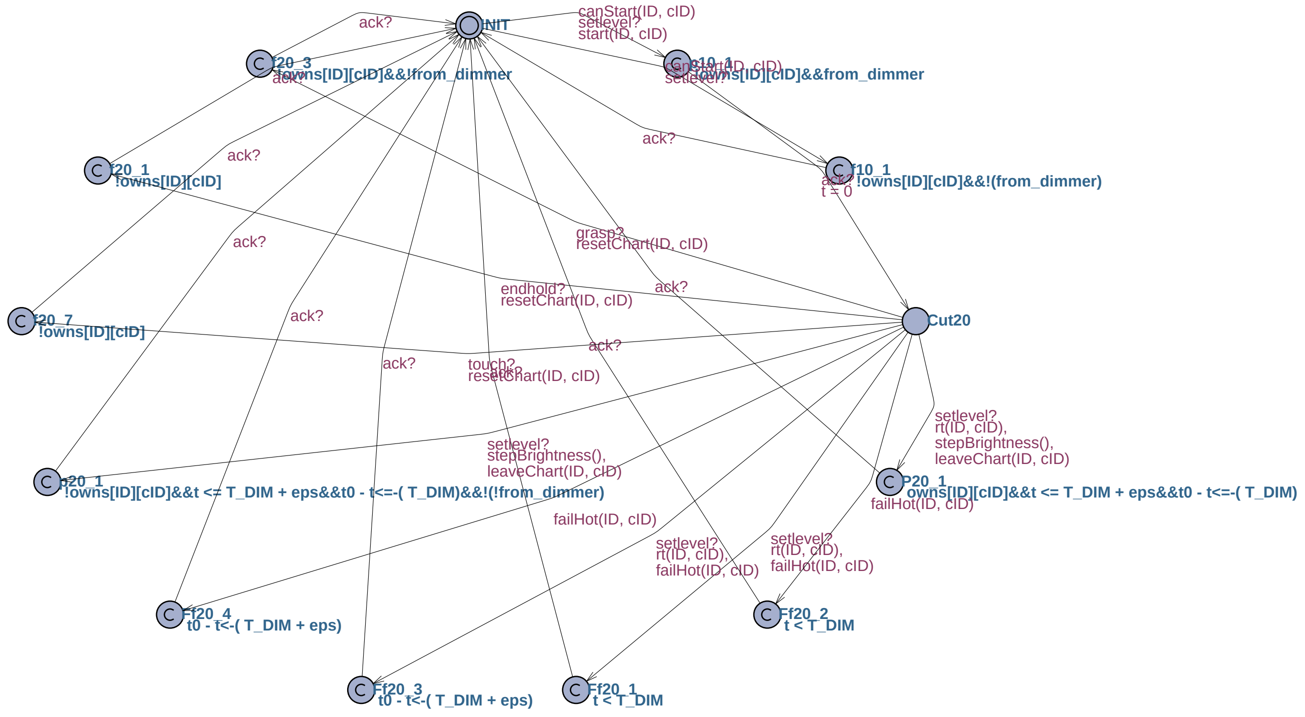

Figure 5.3: LSC chart from figure 3.9 translated into UppAal automaton. The cut groups in this chart are represented by non-committed locations (which is not always the case, i.e. for cuts with enabled standalone conditions). The first cut group and the last cut group are presented by the same initial location. Structures of committed cuts and transitions connecting them with the non-committed cuts correspond to stepping over simregions. The LSC semantics specific limitations are encoded by guards and invariants. The progress of the automaton along its locations and transitions happens from 12 o'clock (the initial location) clockwise, until it returns to the initial location.

5.2 Semantics correspondence between the LSC and UPPAAL

In this section, the semantics correspondence will be explained between the LSC specification *LSCSpec* consisting of LSC charts to UppAal timed automata network NTA as a set of timed automata.

With every configuration in *LSCSpec* we can associate one from translated NTA. And the trace from *LSCSpec* corresponds to the projection of a trace in NTA, similar connection holds the other way. It follows that both models are equivalent with respect to the TCTL properties checkable in UppAal.

5.2.1 Configurations in LSC and UPPAAL

In LSC specification *LSCSpec*, the configuration $(cuts, \epsilon, \nu)$ consists of the cuts from active LSC charts and the values of global and local variables and clocks.

In UppAal network of timed automata NTA, its configuration $(\vec{l}, e, v)$ consists of the locations occupied by its constituting timed automata and the values of global, local clocks and variables.

Initial configuration of *LSCSpec* includes empty set of cuts and all variables with clocks reset. The initial configuration corresponds to the NTA initial configuration with all the TA in their initial locations and all variables with clocks reset.

Each configuration in *LSCSpec* has a corresponding configuration in NTA. The opposite does not hold because of the two steps in NTA corresponding to a single one in *LSCSpec*, thus intermediate committed locations in the translated automata and the temporary valuations of the variables and clocks then do not have correspondence in *LSCSpec*.

We call the configurations of the UppAal model which have the counterpart in *LSCSpec* the *stable* configurations.

Definition 5.2.1 (Stable / unstable NTA configurations)
Given the LSC specification LSCSpec and the corresponding UppAal *model NTA, where every configuration of the LSCSpec, $(cuts, \epsilon, \nu)$, has a counterpart in NTA. A stable configuration of NTA is a configuration $(\vec{l}, e, v)$ such that*

- *No broadcast synchronization ack is enabled, what would mean the configuration between two steps that correspond to a single step in LSCSpec*
- *No $l \in \vec{l}$ is committed or the only outgoing broadcast synchronization is _cond_, meaning the configuration which corresponds to that in LSCSpec with standalone conditions enabled in active charts.*

Every consistent configuration in NTA that does not qualify these requirements, is unstable *configuration.*

There is a mapping from each configuration of the LSC chart copy (c, ϵ, ν) (Section 3.15) to the configuration (l, e, υ) of timed automaton translated from that LSC chart copy. Let us denote that relation *trans*.

Definition 5.2.2 (Matching configurations)
Given is a LSC specification LSCSpec and its proper configuration $c := (cuts, \epsilon, \nu)$. A configuration $s := (\vec{l}, e, \upsilon)$ of corresponding NTA is a matching configuration, denoted $c \sim s$ if the following holds:

1. *Every cut from LSCSpec has corresponding location in related (through relation trans) automaton in NTA occupied: $\forall c \in cuts.\ trans(c).l \in \vec{l}$*
2. *For all automata TA who correspond to the LSC chart copies and whose cut is not matched from c in LSCSpec, their cut is initial cut of TA. Auxiliary automata used to maintain LSC semantics in NTA like event generator or consumer, don't count.*
3. *All variables in LSCSpec have their counterparts in NTA assigned to same values:$\forall v \in Var(\epsilon).\ \epsilon(v) = e(v)$. The auxiliary variables used to maintain LSC semantics in NTA don't count.*

The relation $\sim$ ignores the history, auxiliary automata and the values of auxiliary variables needed to maintain LSC semantics on the translated LSC charts in *NTA*. By construction of the steps, however, for every reachable configuration of *LSCSpec*, only one stable configuration in *NTA* is reachable.

5.2.2 Correspondence of steps

The translated version *NTA* of *LSCSpec* is a refinement in the sense that every step in *LSCSpec* corresponds to two steps in *NTA*.

A delay step of arbitrary duration is always possible if no committed location in stable *NTA* configuration exists, respectively if there are no enabled standalone conditions in the active cuts of the *LSCSpec* configuration. There are no invariants on the occupied locations in the stable configurations of *NTA*. If a too long delay leaves no enabled steps in *LSCSpec*, so does such delay in translated *NTA*.

A transition involving silent step is allowed in *LSCSpec* whenever it becomes enabled, with exception of situation when there are enabled standalone conditions in active charts, and the action is not one of those. The silent step is either performing statement of a standalone assignment or evaluating the standlone condition with the assignment optionally attached to it.Exactly same calculations are performed in the matching configuration of *NTA*, where the corresponding assignment (if any) is performed on the first transition to committed location which includes the expression from condition (if any) as invariant. The book-keeping of the TAs in *NTA* is performed during the first transition (corresponds to activation status change of the chart whose minimal event has been that silent step, or deactivation status of the chart is set, if the silent step has been last in the chart, or deactivation of *all other* active translated copies if the silent action used to be the last event in prechart of a iterative mode chart).

The steps involving the message events fall under the same semantics. Just in case of message, typically several LSC charts in *LSCSpec* and respectively several automata in NTA have their status updated during the book-keeping phase and variable values changed during execution of specified constructs (assignments, standalone or in simregions with conditions or messages). The assignments as the post-actions are performed during the second step of translated NTA, while pre-actions on the first transition. Preconditions and postconditions are placed as the invariants in the middle locations of TA for particular step.

The steps violating some charts in the *LSCSpec* are forbidden by the book-keeping back in translated NTA. This includes too short delay, too long delay, non-enabled event causing the violation or evaluating the hot condition thus causing the violation.

We can relate one step in a *LSCSpec* to a pair of steps in NTA where the first and last configurations are stable, and the middle configuration is not.

Claim 5.2.1 (Step encoding)
Syntax and semantics of the UppAal *model by means of legal steps is presented in section 4.2.2. Same is given for LSC specification in section 3.17 through 3.20.1. Assume the* UppAal *model being the translation of the LSC specification according to the description in section 5.1 and chapter 6.*

For a LSC specification LSCSpec exists a step between two configurations $(cuts, \epsilon, \nu)$ *and* $(cuts\prime, \epsilon\prime, \nu\prime)$ *if and only if for the* UppAal *model* NTA *there exists a corresponding sequence*

$$(\vec{l}, e, \upsilon) \overset{a}{\Rightarrow} (\vec{l}_1, e_1, \upsilon_1) \overset{ack?}{\Rightarrow} (\vec{l\prime}, e\prime, \upsilon\prime)$$

where $(cuts, \epsilon, \nu) \sim (\vec{l}, e, \upsilon)$, $(cuts\prime, \epsilon\prime, \nu\prime) \sim (\vec{l\prime}, e\prime, \upsilon\prime)$, $(\vec{l}_1, e_1, \upsilon_1)$ *is unstable configuration, and* $\alpha \in \{a, \tau\}$.

5.2.3 Correspondence of traces

When the step relation of *LSCSpec* is related to that of NTA, we relate the sets of traces. For every timed trace of *LSCSpec* there exists exactly one timed trace in NTA.

Proposition 5.2.1 (Correspondence of LSC and UPPAAL model)
Given the LSC specification LSCSpec and UppAal *model* NTA *translated from LSCSpec. For every timed trace* $\sigma = \{(cuts, \epsilon, \nu)_i\}_{i \geq 0}$, *there exists a corresponding timed trace* $\hat{\sigma} = \{\vec{l}, e, \upsilon\}$ *of* NTA *such that*

$$\begin{aligned} \forall i.\ \exists k, k\prime, k < k\prime. \qquad & (cuts, \epsilon, \nu)_i \sim (\vec{l}, e, \upsilon)_k & \wedge \\ & (cuts, \epsilon, \nu)_{i+1} \sim (\vec{l}, e, \upsilon)_{k\prime} & \wedge \\ & \forall k < j < k\prime.\ (\vec{l}, e, \upsilon)_j \ \ \textit{is unstable} & \end{aligned}$$

Conversely, for every timed trace $\hat{\sigma} = \{(\vec{l}, e, \upsilon)_j\}_{j \geq 0}$ *of* NTA *there exists a corresponding timed trace* $\sigma = \{(cuts, \epsilon, \nu)_i\}_{i \geq 0}$ *such that*

$\forall k, k\prime, k < k\prime.$ *if* $(\vec{l}, e, \upsilon)_k$ *and all* $j,\ k < j < k\prime.$ $(\vec{l}, e, \upsilon)_j$ *unstable, then* $\exists i,\ (\vec{l}, e, \upsilon)_k \sim (cuts, \epsilon, \nu)_i \quad \wedge (\vec{l}, e, \upsilon)_{k\prime} \sim (cuts, \epsilon, \nu)_{i+1}$

The NTA does not yield maximally extended finite traces ending with unstable configurations. This entails that all trace properties that UppAal can establish for NTA, also hold for $LSCSpec$.

Corollary 5.2.2 (Translation from $LSCSpec$ to NTA sound and complete)
A timed property ϕ from the TCTL fragment in Section 4.2.4 holds in an LSC specification model LSCSpec iff the corresponding property $\hat{\phi}$ holds in translated model NTA.

By proposition 5.2.1, the sets of traces match modulo the unstable configurations contained in the traces of NTA. Local properties cannot refer to the book-keeping variables and auxiliary variables in the unstable configurations and by our well-formedness conditions the values of variables in $Var(LSCSpec)$ change at most once between two stable configurations.

For TCTL fragment in Section 4.2.4 it suffices to quantify over traces.

Holding of timed properties in $LSCSpec$ and NTA is the proof for translation soundness itself. However, there are properties in UppAal that refer to the auxiliary variables mentioned in section 6.4, namely $e\_satisfied[ID]$ and $u\_unsatisfied[ID]$ that stand for satisfaction of (translated) existential type, property-role LSC chart and violation of the universal type, property role chart.

Chapter 6

Implementation

This chapter is about how the system specifications are translated from sets of charts (the so-called LSC specifications) to UppAal networks of timed automata.

The translation approach is described where the LSC charts have their type, mode and role translated, and the translated networks of TA can serve both as the system model (specification) and the observer automata in the same UppAal model. Translation is left out where the LSC charts translate solely into the properties to observe.

The chapter includes description of each developed and used part, and how they are implemented.

Chapter concludes on the implementation efforts.

6.1 LSC file format

The LSC charts are edited using the extension of the LSC editor originally given by the OFFIS group at the Carl von Ossietzky Universität Oldenburg. The chart is saved to a text file and loaded from a text file. Re-editing the charts, translating them into UppAal TA needs their unambiguous storage format defined.

The LSC elements (messages, conditions, assignments, simregions, coregions, subcharts) are represented one line per construct. The textual format, although being simple, allows encoding a number of LSC constructs. They all include ID, span over some instances, have the top, optionally middle and bottom positions and sometimes the expression inside, like conditions or assignments.

Preorder of LSC elements and their correctness of placement is left for the user and the *LSC2UPPAAL* translator. The LSC file format in BNF is presented as follows:

```
FILE ::= TYPE \n MODE \n DESCRIPTION \n INSTANCES \n CHART
TYPE ::= type (existential | universal)
MODE ::= mode (initial | iterative | invariant)
INSTANCES ::= INSTANCES \n INSTANCES
| instance INST_ID NAME (TEMP)+
CHART ::= chart begin \n CONTENTS \n chart end
CONTENTS ::= CONTENTS \n
CONTENTS
| prechart ELEM_ID (INST_ID)+ YLOCATION YLOCATION
| loop     ELEM_ID (INST_ID)+ LOOPCOUNTER YLOCATION YLOCATION
| subchart ELEM_ID (INST_ID)+ YLOCATION YLOCATION
| ifelse   ELEM_ID (INST_ID)+ BEXPR YLOCATION YLOCATION YLOCATION
| simregion ELEM_ID INST_ID YLOCATION
| coregion ELEM_ID INST_ID YLOCATION YLOCATION
| message ELEM_ID INST_ID INST_ID YLOCATION TEMP NAME
| condition ELEM_ID (CARRIERS)+ YLOCATION TEMP EXPR LOCACT_ID LOCACT_ID
| locAction ELEM_ID INST_ID YLOCATION EXPR
| scope ELEM_ID PATTERN TEMP YLOCATION YLOCATION
| cut ELEM_ID PATTERN (INST_ID)+ (YLOCATION)+

BEXPR ::= <boolean expression>
EXPR ::= <expression>
TEMP ::= 0 | 1
LOOPCOUNTER ::= <numeral>
CHART_ID ::= <numeral>
INST_ID ::= <numeral>
ELEM_ID ::= <numeral>
CUT_ID ::= <numeral>
LOCACT_ID ::= <numeral>
XLOCATION ::= <numeral>
YLOCATION ::= <numeral>
PATTERN ::= <numeral>
```

6.2 Command line arguments for the LSC2UPPAAL tool

The translation tool $LSC2UPPAAL$ uses the following command line arguments:

$./LSC2UPPAAL - ddecl\_file - ooutput\_file - sspec\_files - pprop\_files$

where

$decl\_file$ is the full path to the model declarations file name. The file includes declarations of the common functions and shared variables. These are copied into the UppAal model file.

$output\_file$ is the full path to the UppAal model file which is created as the result of the translation.

$spec\_files$ is the list of files where the LSC charts are saved. These files are translated one by one into the UppAal TA templates. Since role of the LSC chart is not specified in the LSC file, the key $-s$ defines the translation of the LSC charts according to the specification-role semantics.

$prop\_files$ is the list of files that are translated into the UppAal TA templates according to the property-role LSC chart semantics.

The LSC files must have their syntax compliant to that presented in section 6.1. The format of the model declarations file name is based on the markers, that is, the text between a pair of certain markers is copied into the corresponding attribute of the UppAal model file. Such format is primitive and intuitive, and thus not presented here.

6.3 LSC chart

Every LSC chart is translated into a UppAal timed automaton template. In UppAal, there are several properties of the chart and its copies accessible as the variables:

Chart ID **ID** is an chart-unique non-negative integer assigned to the translated LSC chart. There is a one-to-one correspondence between translated LSC and the chart ID.

Copy ID **cID** of the chart identifies particular copy among several identic copies of translated LSC chart.

Copy ID is specified as intervals in the template, so are they initialized in as many copies as specified in cID.

Tuple $(\mathbf{ID}, \mathbf{cID})$ uniquely identifies the UPPAAL automaton among the others. Each chart-specific TA is instantiated with unique value of the tuple. Such identification is used for several purposes that maintain the LSC specification satisfaction relation, described in section 3.17:

- Denote which copies of the chart are active, in mainchart, resetting or deactivated. This corresponds to the cuts transformation function, section 3.17.0.1.
- Specify which copy of the chart should activate upon next minimal event observed. There must be only one copy at a time activating over an event.
- In case of violation, identify the violating charts through their IDs.
- Identify the universal type chart according to which an event is performed. This is implementation of the event generation - matching, introduced in section 3.8.

Each translated chart has a set of global variables and flags associated to it, used in maintaining semantics of the translated LSCs.

LSC chart-specific constants in UppAal model are:

- Number of copies defined for particular chart. The number must be bounded and as small as possible in order to perform model checking efficiently. That number can be automatically determined from the LSC chart structure such as amount of repeating message events, ignored and forbidden elements, and preorder relation among the LSC constructs in the LSC chart. Currently, number of copies of the LSC chart is defined manually.
- Chart type and mode. Currently, type and modes of the translated LSC charts are stored in two global arrays. However, type and mode can also be specified through the argument list of the TA template, or implicitly encoded into the template through possible execution paths and accepted or generated events that correspond to synchronizations in UppAal.

LSC chart-specific variables in UppAal model are:

- Index of the chart copy to activate upon minimal event
- Flag which allows activation of new copies upon the minimal event. The flag is reset for completed charts of initial mode and when universal chart of iterative mode has one of its copies in mainchart.
- Flag which is set upon completion of the existential property-chart. It is referred to during the model checking when a property of the kind "some scenario is eventually observed" is verified for such chart.
- Flag which is set upon violation of the universal property-chart. It is referred to during the model checking to verify some universal property where prechart stands for activation condition, and mainchart for requirement to be satisfied.

Chart copy-specific variables in UppAal model are:

- Flag that indicates the ownership of the event. This is the translated concept of the event generation, introduced in section 3.17.
- Resetting-flag used to reset the chart copies, according to the cut transformation function, defined in 3.17.0.1.
- Flag that indicates whether particular copy of the chart is currently active. This is also means to maintain the cuts transformation function.

System-specific constants are:

- Number of LSC charts in LSC specification that are translated into the UppAal process templates
- Maximal number of copies a process template can have. The numbers of the next chart copies to activate are stored in array, and this constant defines the upper bounds to the elements of that array. The constant becomes redundant if the LSC charts are translated into a single-copy process templates that would correspond to several copies of the LSC chart at a time.

Translated LSC system-specific variables are:

- Token, a non-negative integer which is non-deterministically assigned a bounded positive value upon the message synchronization event. Each active (or able to activate) copy of the process template decreases the token by one, if it can own the message event. The copy which decreases it to zero, becomes an owner of the event. There is exactly one owner per mesage synchronization event. This is how the event generation and matching mechanism (section 3.8) is implemented.
- Error flag. It is set upon any of the translated LSC charts violating. By constraining the legal values for the flag, it is possible to perform the simulation or verification using only those moves which do not result in violation (and thus error).

- Error log (optional). It is a bounded array where failing copies of translated LSC charts insert their identifiers. Such a simple error mechanism allows one determine, which charts would be violated by an event.
- Identifier of the event-owning chart. Certain book-keeping is performed during simulation, where searching for the owner of event in the array of charts would decrease performance. Therefore the owner chart ID and copy ID is stored in global variables.
- Clock $t0$ which is reset upon every turn. This enables translating the clock constraints in LSC charts of the form $Clock > val$ into UppAal supported form of clock differences $t0 - Clock <= -val$, respectively $Clock >= val$ to $t0 - Clock < -val$.

Example

Certain LSC chart is translated into UppAal template whose instantiation parameters are like

```
const int[6,6] ID, const int[0,1] cID
```

where

ID initializes the ID of the template to 6

cID creates two instances of template upon initialization, one with $cID = 0$, another with $cID = 1$.

Note that upper bound of cID must be equal or larger that maximal amount of chart copies that can be active at a time after arbitrary sequence of events. Reason for that is, that there always must be one inactive copy, ready for activation upon minimal event.

Amount of LSC charts, their corresponding number of copies and maximal number of copies per chart is declared as follows:

```
 const int NUM_LSC_CHARTS = 4;
 const int NUM_CHART_COPIES[NUM_LSC_CHARTS] = {2,4,5,3};
 const int MAX_CHART_COPIES = 5;
```

Maximal value of token is declared like

```
const int MAX_TOKEN_VALUE = 3;
```

The value must be non-negative and equal or larger than amount of active or activating chart copies that are able to progress upon accepting symbol $Chan!$ for some channel $Chan$. Token is initialized with non-deterministic value from the beforementioned interval to determine which of such copies will accept $Chan!$, since all other copies will have to accept $Chan?$.

The cID of the next copy to activate and flag forbidding to activate the copies of particular translated chart are defined as

```
int nextCopy[NUM_LSC_CHARTS];
bool dontActivate[NUM_LSC_CHARTS];
```

Copy-specific information like owning the event, resetting and active flags are declared as

```
bool owns[NUM_LSC_CHARTS][MAX_CHART_COPIES];
bool resetting[NUM_LSC_CHARTS][MAX_CHART_COPIES];
bool active[NUM_LSC_CHARTS][MAX_CHART_COPIES];
```

It should be noted that all-but-one chart copies must reset once a copy in an iterative LSC chart enters its mainchart. Thus it is explicitly specified which copies must reset upon this event, and *dontActivate* flag is set for the entire set of LSC chart copies.

6.4 LSC chart type

Existential LSC charts are used purely as the properties to check the model against. The user of the specification is typically interested whether a model of the system exposes behavior that makes the existential chart successfully complete.

Universal charts are used to define the requirements for behavior of the model. They have optionally a prechart which corresponds to any of its defined activation conditions or events. Once the prechart is successfully completed, the mainchart is entered which specifies what actions should be taken after completion of prechart.

Satisfaction of the existential and universal charts depending on their mode and role are specified in section 3.16.

6.4.1 Implementation

There is an entry for every translated chart which defines chart's type:

```
//! LSC types enumerated
const int LSC_TYPE_EXISTENTIAL = 0;
const int LSC_TYPE_UNIVERSAL = 1;


//! chart types for the charts in the system
const int CHART_TYPE[NUM_LSC_CHARTS] = {
LSC_TYPE_UNIVERSAL,
LSC_TYPE_UNIVERSAL,
...
LSC_TYPE_EXISTENTIAL
};
```

Information, whether an existential type chart has been satisfied or universal type chart has been violated, is stored in global variables:

```
bool e_satisfied[NUM_LSC_CHARTS];
bool u_unsatisfied[NUM_LSC_CHARTS];
```

Reachability or safety properties are checked by reachability like $E <> e\_satisfied[ID]$ or $E <> u\_unsatisfied[ID]$ for a chart with identifier ID.

Flag for existential type chart is initially reset and is set once it completes. For universal type chart, the flag is initially reset since not completing the prechart implies no necessity to have mainchart completed. The flag can be set upon simulation or verification when the set of events allows the completion of the prechart at some point and violates the mainchart afterwards.

6.5 LSC chart mode

As already mentioned in section 3.5, LSC charts are of 3 modes: initial, iterative and invariant. Satisfaction of the LSC charts depends on their type, mode and role. It is defined in section 3.16.

6.5.1 Implementation

There is an entry for every translated chart which defines chart's mode:

```
//! LSC modes enumerated
const int LSC_MODE_INITIAL = 0;
const int LSC_MODE_ITERATIVE = 1;
const int LSC_MODE_INVARIANT = 2;

//! chart modes for the charts in the system
const int CHART_MODE[NUM_LSC_CHARTS] = {
LSC_MODE_ITERATIVE,
LSC_MODE_INVARIANT,
LSC_MODE_INITIAL
};
```

LSC chart activation mechanism is implemented by updating the entries in arrays *dontActivate*[] and *nextCopy*[].

Depending on the chart's mode, the UppAal transitions are decorated with different update functions for situations when chart completes, terminates in prechart (mainchart) or enters mainchart.

Activating the chart copy Function *bool canStart*$(intID, intcID)$ checks whether the chart copy identifier allows the chart activating, and whether the copies of the chart can activate at all (in case of universal type, iterative mode chart). This function implements the formation of *startcuts* cut set, mentioned in section 3.17.0.1.

Entering the mainchart The function *void enterMain*$(intID, intcID)$ only applies to the universal type charts of the iterative mode. All other copies are marked as resetting upon one of the copies entering the mainchart. The function partially influences formation of the *termcuts* and *maincuts* cut sets, defined in section 3.17.0.1.

Leaving the mainchart Leaving the mainchart of the LSC chart corresponds to the copy of the chart template being reset. In case of iterative mode LSC chart, new copies are again allowed to activate upon minimal events. Templates of the initial mode existential charts have their flags set that prevent further activation of the chart copies. This functionality is implemented in function *void leaveChart*$(intID, intcID)$ which influences the formation of the *termcuts* set defined in section 3.17.0.1.

Completing the chart (of existential property-role charts) Function $void\ propPass(int\ ID,\ int\ cID)$ decorates the success transition to the maximal cut of the existential property-role chart. The flag $e\_satisfied[id]$ is set for the chart and the entries in $dontActivate[]$ array are set since the properties are already satisfied and no more activations of the charts are needed.

Violating the chart (of universal property-role charts) Function $void\ propFail(int\ ID,\ int\ cID)$ is invoked upon the hot violation of the property-role chart. Besides leaving the chart, the error flag and error log is updated for the specification-charts. Alternatively, the $u\_unsatisfied[id]$ flag is set for the property-charts.

6.6 Translation of messages, conditions, assignments

There are three kinds of the constructs that constitute the simregions, that are messages, conditions and assignments. These have been introduced in section 3.11.

For each kind of simregion (figure 6.1) there are at least one unstable (commited) location (figure 6.2) and one stable location dedicated, and a set of transitions connecting these locations among themselves and with the others.

Typical events and simregions thereof in LSC sre depicted in figure 6.1.

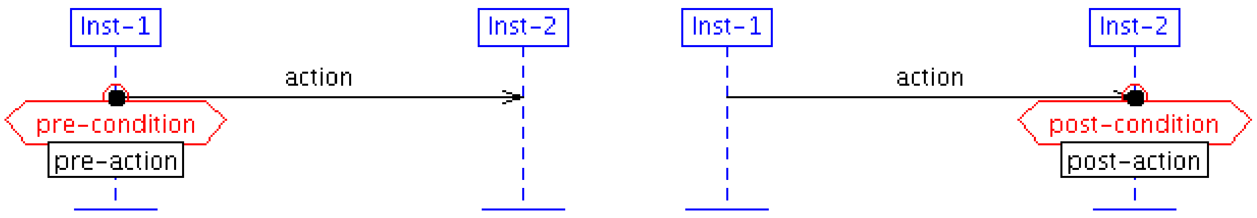

Figure 6.1: The types of simregions usually met in LSC. Smaller simregions with less constructs joined together are also met, up to single LSC constructs.

Events and simregions thereof from figure 6.1 are translated into UPPAAL using template depicted in figure 6.2.

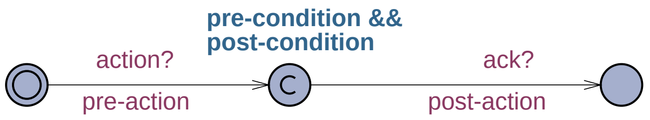

Figure 6.2: The typical combination of UPPAAL constructs used to translate the simregion in LSC. Middle location is unstable, it holds invariants and is exited immediately from being entered; the left and right locations are stable, they correspond to some cut and its successor, respectively. Preaction is placed as the update of the first transition leading to unstable location, while postaction is placed as the update of the second transition.

The five elements of LSC simregion are placed onto the UPPAAL frame of constructs as follows:

preaction translates into the update of the first transition. Besides the preaction, the book-keeping functions mentioned in section 6.5 can be placed into the update of the first transition. Some book-keeping functions, like those allowing particular template to activate, are placed on the guard of the transition.

precondition, postcondition translate to invariants of the middle location. This means, that the postcondition is unable to check the result of the postaction, and the variables involved in preconditions should not be affected during preaction, since it can easily disable the entire move, and it will be hard to debug. The chart state updating functions from section 6.5 are typically placed into the preaction.

event label translates to the synchronization of the first transition.

postaction translates into the update of the second transition. For the iterative mode charts, there are two transitions originating in unstable location, one with the guard for not resetting, another with the guard of resetting to the initial location. That is how the all-but-one copies of the same LSC chart are reset according to the LSC semantics.

6.7 Subcharts

LSC subcharts involve prechart, loop, subchart and If-Then-Else construct, that have been formerly introduced in section 3.19.

None of these are translated explicitly into UppAal constructs since it is operated over simregions in LSC chart. Subcharts only affect the preorder relation among simregions, therefore translation affects the enabled transitions between the cut groups.Each subchart affects the transitions in the following way:

prechart affects the simregions inside so that none of these have the update $rt[ID][cID]$ inside, that is, no event generation in prechart is possible.

For the universal chart of iterative mode, the last simregion inside the prechart has update $enterMain[ID][cID]$ what makes all other active copies to reset, thus maintaining semantics of the iterative mode chart entering its mainchart.

As all subchars, the prechart affects the preorder relation among the simregions in the LSC chart. This results in fewer valid execution paths through the LSC chart. In particular, cuts with elements from mainchart can not be entered, if there are still elements from the prechart to be taken.

subchart only affects the preorder of the simregions inside the LSC chart. No other influence is intended to be done by using this construct.

loop acts in the same way as the subchart. The only explicit artifact of the loop is the transition from the last cut inside the loop is not directed to the location of the next cut according to the preorder relation. The transition is instead directed to the location that corresponds to the cut right before the loop.

if-then-else construct, besides performing the function of the prechart, splits the execution path into two at the minimal event inside. It also and joins these paths together again, after maximal events inside of the if- or else-branches have taken place. This is how the non-deterministic choice is implemented.

6.8 Translation of non-trivial LSC semantics aspects

There are several aspects in LSC semantics that are non-trivial to translate into the UppAal automata. These aspects are shortly revisited in this section, with the solutions described.

6.8.1 Activating of the chart copies

In LSC, only one chart copy can activate at a time. In UppAal, several copies of the TA template are instantiated at the beginning, and these must follow the behavioral activation pattern like specified in LSC.

Solution

Every template has guard *bool canActivate*(ID, cID) and update *void activate*(ID, cID) on its transitions corresponding to minimal events in LSC chart. Global array *nextCopy* is defined where there is an integer entry for each template.

The former function restricts whether particular copy of template is referred by a corresponding element in array *nextCopy*, this allows to activate only one copy at a time. The second function sets the status flags for particular template, like *active*$[ID][cID]$ and recalculates the *nextCopy*, which will be the copy of template whose *active*$[ID][cID]$ is not set and cID is the smallest among such inactive copies of the template.

6.8.2 Entering mainchart for iterative chart

When a copy of iterative mode LSC chart enters its mainchart, other copies according to the semantics of iterative mode LSC chart must immediately deactivate.

Solution

Transitions that lead from the cuts in prechart to the minimal cut in the mainchart, are decorated with function *void* $enterMain(intID, intcID)$. This function sets the *resetting* flags for all other copies of the TA template. The copies return to their initial locations and have their flags *active* and *resetting* cleared upon the next synchronization.

6.8.3 Owned and non-owned events

In a LSC specification $LSCSpec$, a symbol *Sync* is accepted only if one of the cuts from $cuts \cup startcuts$ with some enabled simregion labeled with *Sync* can accept the *Sync*! symbol, and the rest of such cuts can accept the *Sync*? symbol.

Solution

In UppAal model, a global non-negative variable *token* is declared. It is non-deterministically set to some positive value upon every occurrence of synchronization event in the model. The maximal value of *token* is chosen such that it is not less than maximal amount of active template copies able to accept the *Sync*! symbol for any of synchronizations from *Sync*.

Every copy of template which can accept the *Sync*!, decreases its value by one. The copy which decreases *token* to zero, progresses according to rule $\overset{Sync!}{\Rightarrow}$, while others progress according to rule $\overset{Sync?}{\Rightarrow}$.

6.9 Populating UPPAAL locations for LSC cuts

Translation from LSC chart to the UppAal template involves the step where the cut groups are defined and direct transitions among them determined (section 5.1). Every cut group has one UppAal location. Locations are assigned or populated for the cut group according to the rules:

- The first cut group (that includes the minimal cut of the LSC chart) and the last cut group (that includes maximal cut of the LSC chart) are given a reference to externally created initial UppAal location.
- For all other cut groups the non-initial UppAal location is created. Location is optionally marked as committed, it depends on the enabled transitions from the cut group, namely whether there are enabled transitions that include simregions with conditions without messages.

6.9.1 Priority of the events

In UppAal the location can be marked as normal, urgent or committed.

To enable the ”as-soon-as-possible” condition evaluation in LSC, the UppAal locations are marked as committed for the cut groups who have the standalone conditions in simregions of outgoing transitions; only such simregions are translated from committed locations. This does not apply to the initial cuts.

6.10 Populating UPPAAL locations for LSC progresses

A progress by default uses its source cut and target cut locations. Extra locations are populated as middle committed locations for implementing LSC constructs:

- For assignments, one middle location is populated. It is committed therefore forces the LSC to progress immediately to the location of another cut after executing the assignment.

  Stepping over the simregion that consists of assignment in LSC, corresponds to the τ simple action in *LSCSpec*. Same silent action is used in translated UppAal template. In both semantics, a single chart (template) progresses and some variable (clock) values are optionally updated.

  Simple action in *LSCSpec* that involves a sole assignment, corresponds to silent action in UppAal model. Following transition is used for performing book-keeping of resetting LSC copies.

- For conditions, their expression is analyzed and split into disjunct parts (sub-conditions) which afterwards are normalized to UppAal supported form. The normalized form is with clocks and clock differences related through the "strictly less" or "less-or-equal" to some integer value. A middle location is populated and labeled with each such sub-condition as invariant.

  Expression of the condition is negated and then split into disjunct parts similarly as the pre-negated expression. Again, committed middle locations are populated and labeled with resulting sub-expressions. They are a part of the negative flow which corresponds to the chart behavior upon not meeting the restrictions of the condition.

  Simple action in *LSCSpec* that involves condition, optionally assignment but not a message, corresponds to a set of silent actions in UppAal model, where one of the actions is chosen from the set based on the system state, and performed. Following transition is used for performing the book-keeping of resetting the LSC chart copies.

- Messages have one or more committed middle locations, dependent on whether they are in simregion with condition that includes disjunctions. In case of a message without condition, one committed middle location is populated without an invariant. Exactly the same middle locations are populated as if it would be the standalone condition.

  If a condition in the simregion is a precondition, the sets of locations are populated twice. The first set is for specifying the progress upon observing the external message event. Another set is for specifying the chart progress when the standalone condition is evaluated.

  Synchronized action in *LSCSpec* that involves message, optionally condition and assignment, corresponds to a set of synchronized actions in UppAal model, where one of the actions is chosen from the set based on the system state and performed. The following transition is used for performing the book-keeping of resetting the LSC chart copies.

6.11 Populating the UPPAAL transitions for the cut progresses

For every middle location populated when analyzing the cut progress, two to three transitions are populated. The first transition originates at the location of the cut group of the source cut and terminates at the middle location.

The second transition originates at the middle location and terminates at the location of the cut group of the target cut.

The second transition has always the synchronization $ack?$ and guard $!resetting[ID][cID]$. This allows all the relevant TA progress in a single step and save the amount of synchronizations needed to put the network of the timed automata into their intended states. This becomes especially important after some UppAal locations belonging to the groups of cuts are committed.

The third transition has always the synchronization $ack?$ and guard $resetting[ID][cID]$. The corresponding entry in $resetting$ array may be set by $enterMain(ID, cID)$ in some other copy of the template which means one copy of iterative LSC chart entering mainchart. In this case, all other copies must reset. Thus the third transition starts at middle location and points to the initial location of the TA template.

Labeling of the transitions and their target cut groups are determined dependent on the LSC construct included in the progress:

- For the standalone assignments, the update section of the first transition is decorated with the expression of the assignment.
- For standalone conditions, the synchronization of the first transition is *_cond_?*. The second transition has an update labeled with expression from dependent assignment, if any. Update is not added to the second transition, if the middle location includes part of the negated condition's expression; this is LSC semantics of the failing conditions. The target location corresponds to the cut group where the cut would land upon evaluating the condition - passing, cold failing or hot failing situations are possible when evaluating the condition.
- The first transition corresponding to the message construct usually carries the message label. In case of attached assignment, rules of the standalone assignment apply if the assignment is at the end of message (postaction). Otherwise, the first transition update is labeled with assignment's expression.

  In case of a precondition, two sets of middle locations is populated. Transitions that point to the first set of locations, have the synchronization same as the message label; those pointing to the second set of locations have their synchronizations labeled with *_cond_?*. Their complement transitions from the middle locations are labeled with parts of the pre-negated condition expression. They point back to the location of the initial cut, thus forming a loop. The loop is needed to supplement the negative condition evaluation, so that the translated simregion with precondition does not constrain events more than expected.

The first transitions are optionally labeled with guards $canStart(ID, cID)$, and updates $start(ID, cID)$, $propPass(ID, cID)$, $propFail(ID, cID)$, $enterMain(ID, cID)$, $leaveChart(ID, cID)$, $resetChart(ID, cID)$ or $failHot(ID, cID)$ dependent on what execution path they represent and what kind of LSC chart has been translated.

6.12 Specification and property role charts in translation

LSC charts can perform one of the two functions: they can constitute the system specification or be used as the property to verify the model against.

The former group of charts are the specification role charts, the latter are the property role charts. Differences between these two roles during the translation are:

- Specification role charts have their UppAal locations optionally decorated with invariants $owns[ID][cID]$ or $!owns[ID][cID]$. The property-charts need only invariants $!owns[ID][cID]$ as they are only used to observe the events generated by implementation and environment.
- Specification role charts have some of their transitions in mainchart decorated with update $rt(ID, cID)$. The update is used to reduce the *token* variable and consequently resolve which copy of the template is supposed to own the mesage event. The property charts do not need these updates either, since they do not own events.
- Chart violation function differs for the two types of charts. The one for specification role charts ($failHot(ID, cID)$) records error(s) or prevents the move from being taken. The one for property role charts ($propFail(ID, cID)$) does not prevent the move from being taken - instead it sets the flag $u\_unsatisfied$ which indicates that the property role chart has been violated by the existing specification.
- For the property role charts of existential type, the flag $e\_satisfied$ is set upon the successful completion of the chart.

The property-charts have several considerations:

- Ignored and forbidden scopes always apply to the chart as it is observing the system behavior (operating over non-owned events only).
- Failure (hot violation) in mainchart of the universal chart, besides resetting the chart, results in setting the violation flag $u\_unsatisfied$.
- Maximal events in existential chart must be identified, so that their transitions can be decorated with function $propPass(ID, cID)$ which resets the chart and sets the $e\_satisfied$ flag.

6.13 Summary

The software pieces developed in the PhD project are

- extension of the LSC editor and file format, section 6.1
- translator from LSC to UppAal, formally introduced in chapter 5

The software developed allows to achieve the objective of the PhD project, namely testing the system implementations against their specifications defined by means of the LSC charts.

Actual testing is performed against the LSC charts translated into the UppAal automata. Moreover, the translation defined allows simulation and model checking of the LSC specifications in the tool UppAal. The properties in LSC can be translated into the observer automata and used to check the properties of the translated LSC specifications.

Simplified translation has been carried out solely for generating the observer automata from the LSC charts. Such translated LSC charts can be included into any of the UppAal specifications with minimal prerequisites (all synchronizations in such NTA must be broadcast). The simplified translation is not documented in the thesis as it maintains fewer functions than the translation chosen.

Chapter 7

Case studies

Several case studies have been carried out in order to evaluate how suitable the taken LSC formalism is for capturing the behavioral requirements. The models chosen are as follows:

ABP Alternating bit protocol. This has been one of the first communication protocols to define in LSC and translate into UppAal. The description of the protocol with its Promela code is originally presented in [Hol92]. The requirements have been defined based on the code and description, and formalized using the LSC. The specification then has been translated into UppAal and exercised for simulation and model checking.

SWP Sliding Window Protocol. The code and description of SWP, similarly to ABP, has been taken from [Hol92]. Formal model has been also built by means of LSC charts for that protocol, and translated into UppAal. It should be noted that formalization of these two protocols has largely influenced the semantics of LSC and the set of LSC constructs used. The further case studies, although more complex, have minor contribution to the LSC semantics definition.

ATM The automatic telling machine. This is an academic un-timed example taken from [Eri05]. Originally, the requirements and properties have been written in a formal language, and then the UppAal model built based on these. We have built the LSC model based on these requirements instead and translated the specification into the UppAal. The translated specification has performed slower than the model built originally in UppAal. However, we could express more properties in LSC than in UppAal that were used to validate the model.

Lamp The smart lamp model. Originally it has been the example of the timed UppaAal model that comes as a demo with the toolUppAal TRON [LMN04], [LMNS05]. Its requirements of behavior have not been given explicitly, therefore the UppAal model has been analyzed and the requirements extracted from the timed automata that constitute the model. The LSC specification has been built from those and translated into UppAal. The JAVA implementation of the smart lamp has been tested with UppAal TRON against the original model and the one translated from LSC. Same set of inputs and outputs have been observed during the test, when the conformance test was run against the LSC and UppAal specifications. The conformance test against the LSC specification did not terminate because of the size of the translated specification.

Mouse the intelligent mouse model. This has been another timed UppAal model coming as a demo with UppAal TRON. Similarly to the smart lamp model, specification has been derived from the intuitive requirements, captured in LSC and translated back into the UppAal. The translated model has been too large to have its properties checked in UppAal. However, it has performed well during the conformance test of the JAVA implementation of the intelligent mouse. The mutants of the implementation have been defined, and both LSC and UppAal specifications detected the same mutants.

DHCP The state machine of the DHCP [Dro97] client. The RFC document has been analyzed and the behavioral requirements addressing the client state changes and reactions to the inputs and timeouts have been captured in LSC scenarios. The specification then has been translated into UppAal, the Python adapter written for UppAal TRON to handle the real implementation of DHCP client which has been developed in Ericsson A/S (Telebit). The conformance test has been run against that implementation, and the test verdict obtained that the implementation conforms to the specification.

In all cases, the translation of the LSC specifications to UppAal models has been carried out automatically, using the own-developed translator $LSC2UPPAAL$.

The first two cases of communication protocols are straight-forward translation from the Promela code. Their yet Promela models have been verified using model checker SPIN, and translated models using UppAal just to show that both versions possess same properties. Thus it was not worth to treat them as separate case studies. The last four cases are presented extensively in this thesis, the ATM in section 7.1, the untimed Smart Lamp model has been presented in section 3.2, the timed model of intelligent mouse presented in section 7.2, and the DHCP client state machine in section 7.4.

7.1 Automatic Telling Machine

The ATM machine specification has been defined by means of Live Sequence charts. The case study has been carried out as the response to the [Eri05] where the system requirements and the properties have been defined by means of timed automata. The specification in LSC has then been translated into the UppAal timed automata and model checked against properties defined by means of LSC charts and also translated into the UppAal automata.

7.1.0.1 Actors and communication among them

Three actors of the system have been identified:

- Machine
- Customer
- Auditor

Actors communicate via events. Set of events is defined that are directed from Customer to Machine:

- coinIn - when the customer inserts a coin
- requestCan - when the customer requests for a can
- cancel - when the customer discards the transaction

Set of events is defined that are directed from Machine to Customer:

- coinOut - when machine returns one coin to the customer
- canOut - when machine sells the customer one can

Set of events is defined that are directed from Auditor to Machine:

- inventory - when auditor refills the cans and takes money for sold cans

Set of constants and variables is defined for the model. The constants are as follows:

- MAX_CANS = 5 is the capacity of the machine.
- MAX_COINS = 10 is the capacity of coins that the customer can insert into the machine before requesting a can.
- CAN_COST = 5 is the cost of one can in coins.

The integer variables are as follows:

- nCans is the amount of cans currently available at the machine (varies from 0 to MAX_CANS)
- nCoins is the amount of coins currently inserted by user (varies from 0 to MAX_COINS)
- totalCoins is the amount of coins collected inside the machine for selling the cans to the customers.

Requirements

1. The customer inserts a number of coins into the machine

   The requirement is re-stated as requirement (1): "The customer can insert a coin into the machine"

2. The (valid) coins are accepted until a maximum amount of pending cash is reached

   The requirement is re-stated as requirement (2a): "The customer is allowed to insert a coin when $nCoins < MAX\_COINS$". Another requirement (2b) describes that "whenever the customer inserts a coin, *nCoins* is increased by one". These two requirements allow to keep track of how many coins there are currently inserted by user.

3. The customer presses a button to state his intention of purchasing a can.

   The requirement is re-stated as requirement (3): "The customer can press a request button to request a can".

4. The machine returns the can and the change (and, 2 alternatives specified).

   The requirement is re-stated as a set of scenarios:

   (4a) whenever a customer presses a request button, a sequence of events is performed:

   (a) If amount of inserted money $nCoins$ is enough for a can ($ncoins \geq CAN\_COST$) and cans are available in the machine ($nCans > 0$), the can is returned, otherwise nothing happens

   (b) Remaining coins are returned to the user.

   The input from user and auditor is disabled during the return of the coins and optionally the can. In LSC, this is expressed by forbidding the inputs during returning of a can and coins.

   (4b) whenever a machine issues a can, *nCoins* gets decreased by CAN_COST, *nCans* gets decreased by one and *totalCoins* gets increased by CAN_COST.

5. An inventory can be carried out on the machine if none is currently trying to purchase a can.

   The requirement is re-stated as a set of scenarios:

   (5a) An inventory can be carried out on a machine

   (5b) Inventory is only allowed on a machine whenever *nCoins* is zero

(5c) Upon inventory, number of cans is increased to 5, and amount of stored coins for the sold cans is decreased by CAN_COST every time a new can is inserted.

6. When the customer presses the "cancel" button, the inserted coins are returned. The requirement is re-stated as a set of scenarios:

   (6a) The customer can press a "cancel" button

   (6b) Whenever the customer presses the "cancel" button, inserted coins $nCoins$ are returned. No interaction from user allowed while the cans are being returned.

7.1.0.2 Requirements captured as LSC scenarios

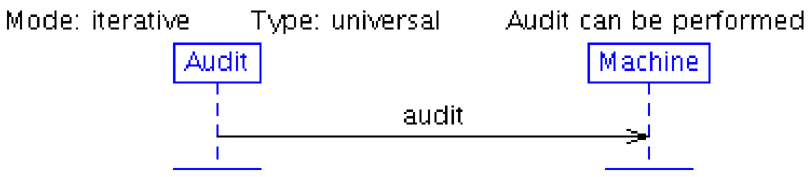

Figure 7.1: LSC chart representing the requirement (5a).

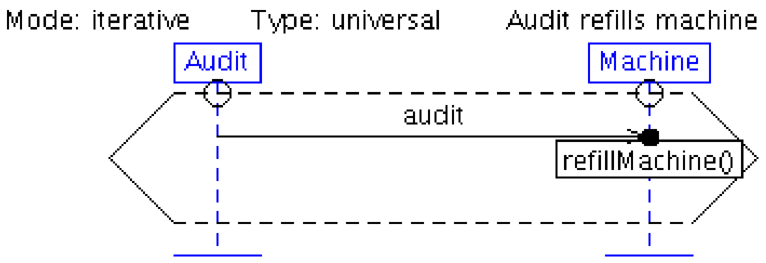

Figure 7.2: LSC chart representing the requirement (5c). Cans are refilled upon MAX_CANS, and CAN_COST coins are moved from temporary storage of coins to the storage for money collected upon each can inserted.

The function $RefillMachine()$ is declared as follows:

```
void refillMachine() {
while (nCans < 5) {
   nCans += 1;
   totalCoins -= CAN_COST;
  }
}
```

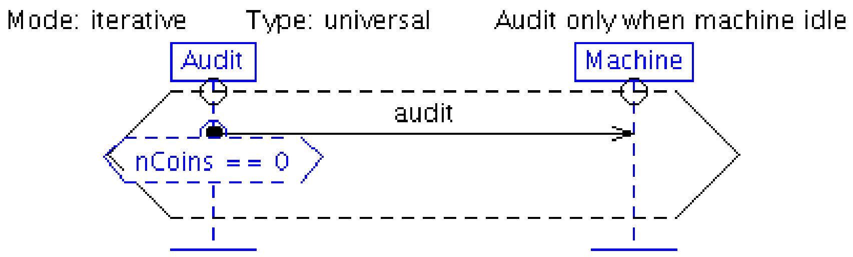

Figure 7.3: LSC chart representing requirement (5b): no pending requests (i.e. no coins inserted) must be while performing the audit.

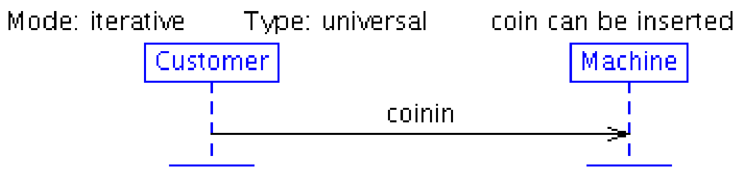

Figure 7.4: LSC capturing requirement (1).

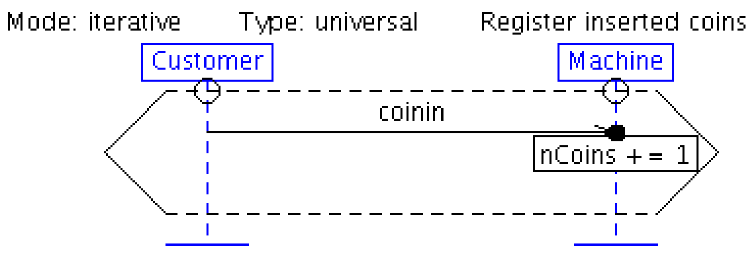

Figure 7.5: LSC specifying requirement (2b).

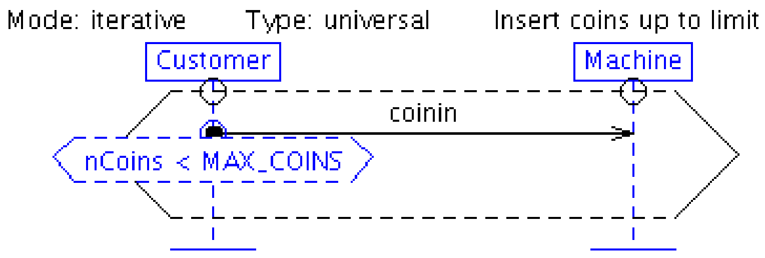

Figure 7.6: Restriction mentioned in requirement (2a). The storage must have some empty space left for new coin in order to accept it.

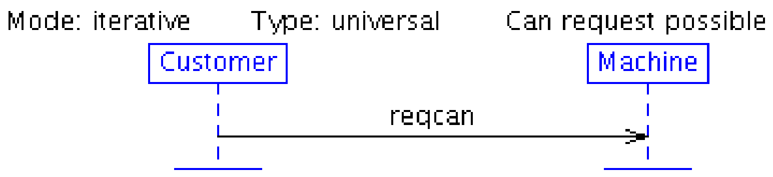

Figure 7.7: LSC chart representing the requirement (3).

7.1.0.3 Properties for the model by means of LSC charts

Properties to check the system against are presented in figures 7.13 to 7.21. Description of the properties follow with the figures.

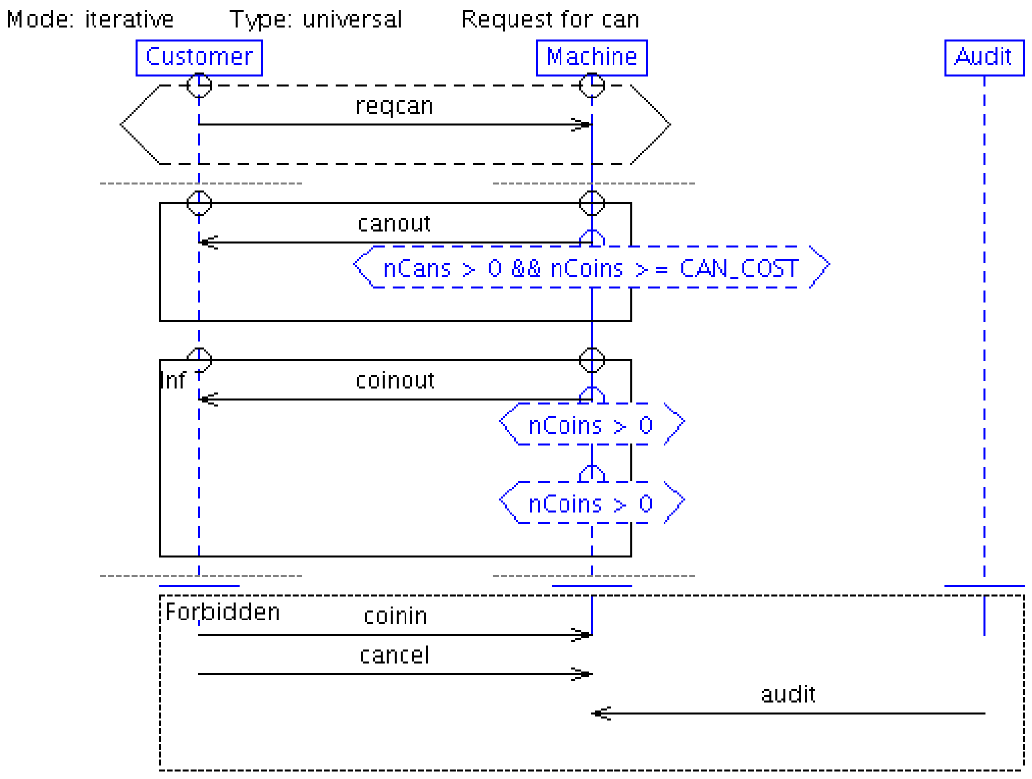

Figure 7.8: LSC chart representing the requirement (4a). LSC subchart construct allows continue execution outside it when inside some cold violation has occurred.

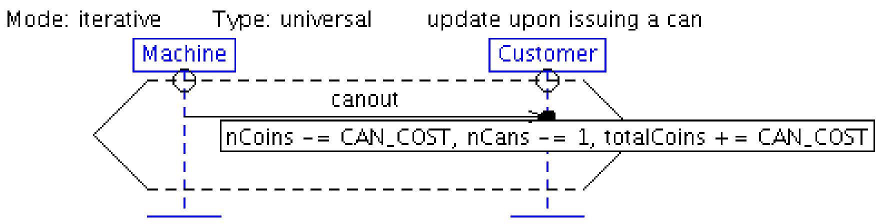
Mode: iterative
Type: universal
update upon issuing a can
Machine
Customer
canout
nCoins -= CAN_COST, nCans -= 1, totalCoins += CAN_COST

Figure 7.9: LSC chart specifying requirement (4b), i.e. upon output of the can, inserted CAN_COST coins need to be moved to another storage and amount of cans decreased.

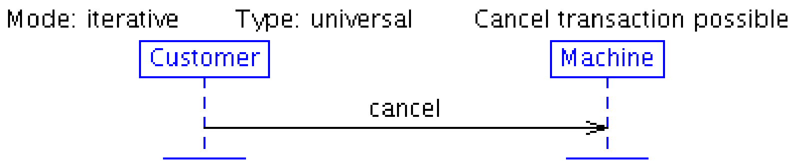

Figure 7.10: LSC chart representing the requirement (6a) - that the user can press the cancel button.

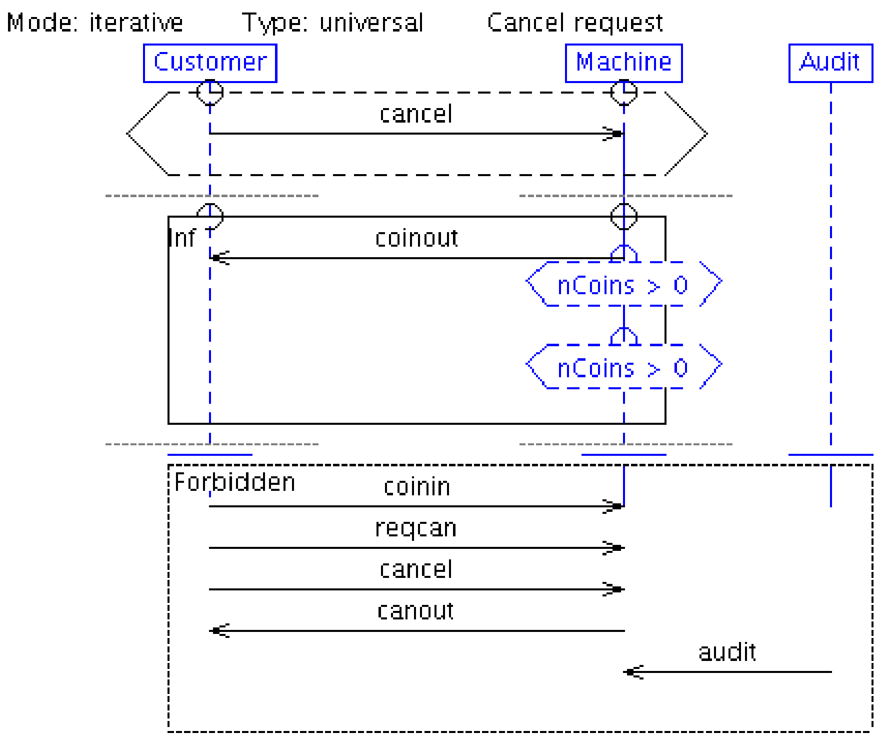

Figure 7.11: LSC chart representing requirement (6b) - what happens upon pressing the cancel button.

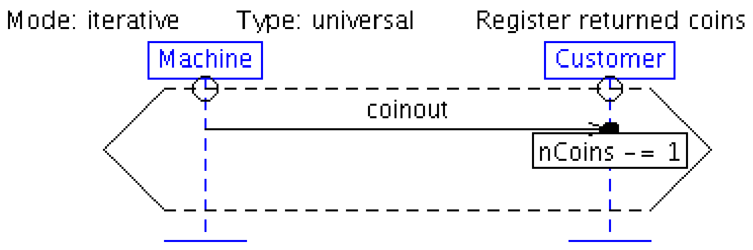

Figure 7.12: LSC chart representing what happens upon releasing a coin at the machine.

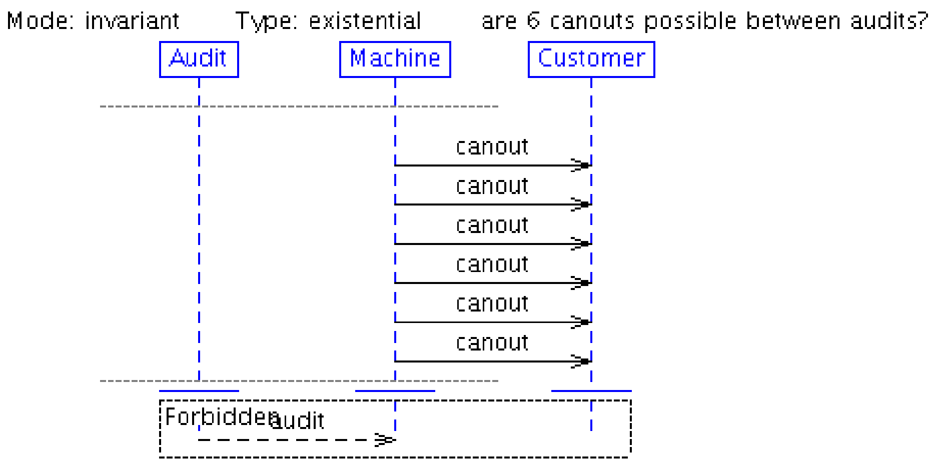

Figure 7.13: Existential chart expecting 6 canout actions in a row without the audit action in-between.

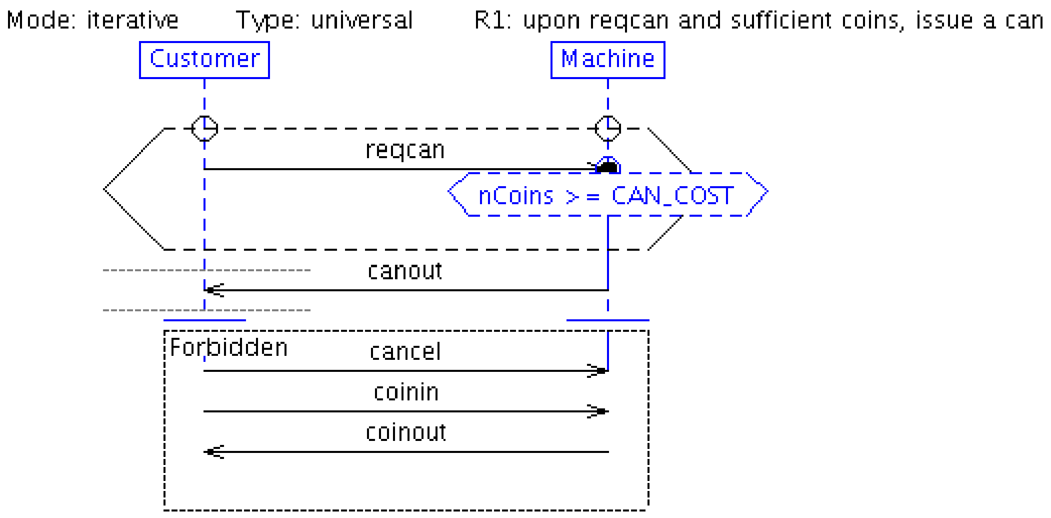

Figure 7.14: Universal chart encoding the property ”When number of inserted coins is sufficient, the can will always be delivered”.

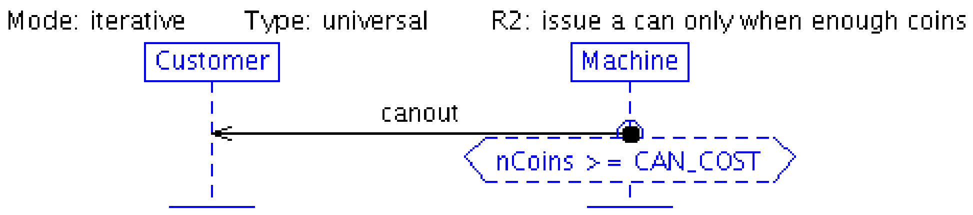

Figure 7.15: Universal chart encoding the property ”The can is issued only when number of coins is sufficient”.

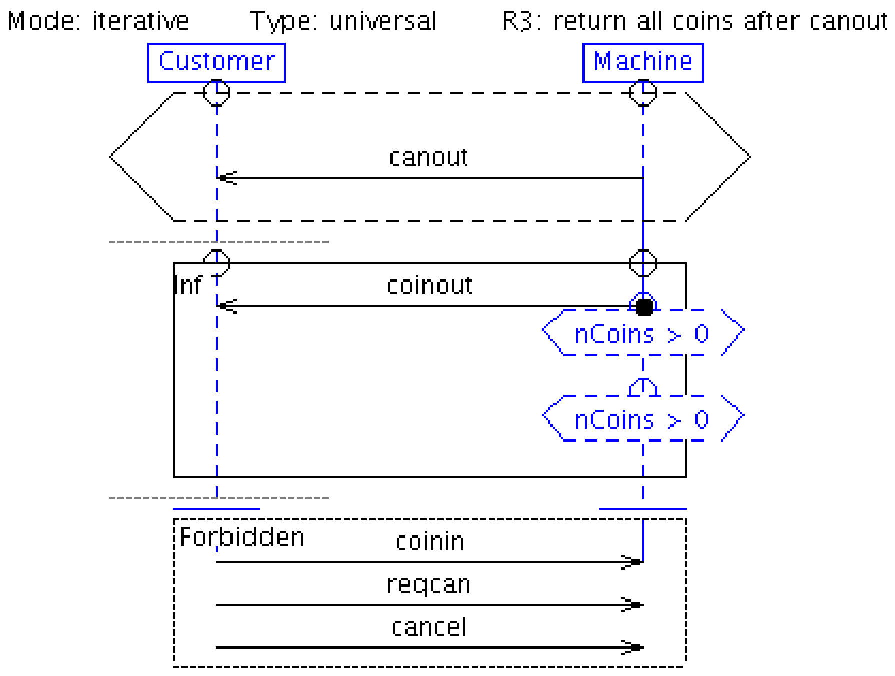

Figure 7.16: Universal chart encoding the property "All remaining coins are returned after the can is delivered to the customer". The loop construct is used to capture as many coinOut events as necessary.

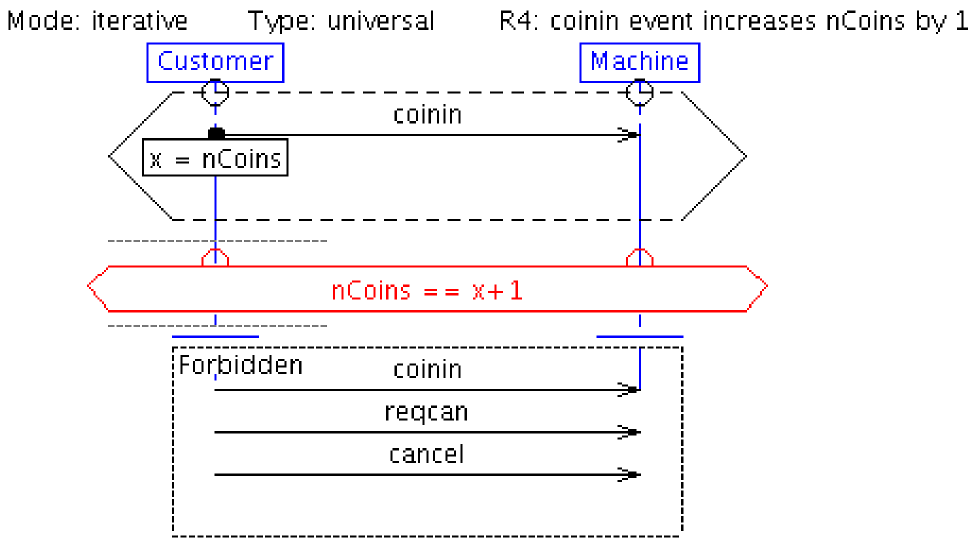

Figure 7.17: Universal chart encoding property "Every inserted coin increases the *nCoins* by one". The requirement is successfully captured because of the "as-soon-as-possible" semantics of the enabled conditions.

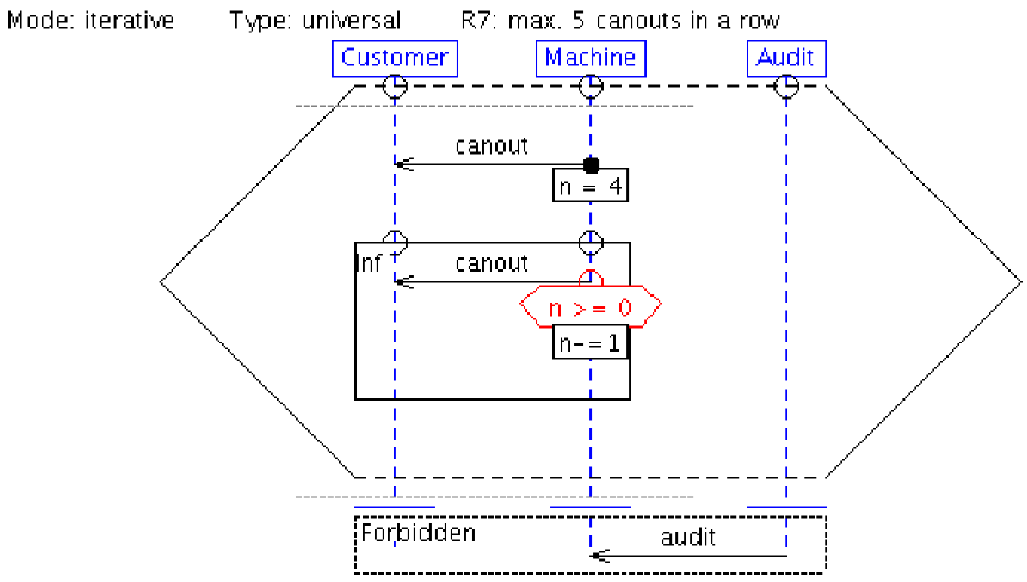

Figure 7.18: LSC capturing the property "Maximum five canout events can happen between audits". For successful translation, the chart has infinite loop registering the CAN_OUT events.

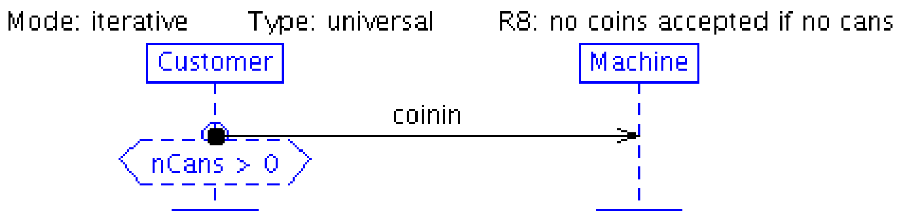

Figure 7.19: Property "No coins accepted if no cans available at the automaton" captured in LSC. The property is actually violated, as specification allows inserting coins while no cans is available at the machine.

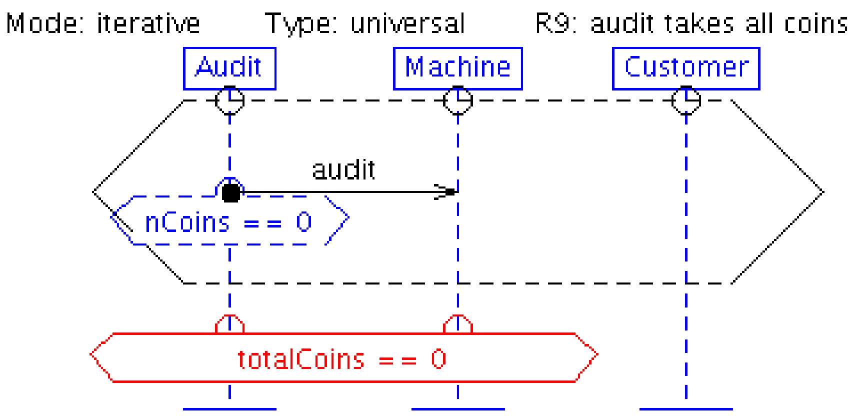

Figure 7.20: Property "Audit takes all coins" captured in LSC. Translation of the property to UppAal is straight=forward with ASAP condition evaluation semantics.

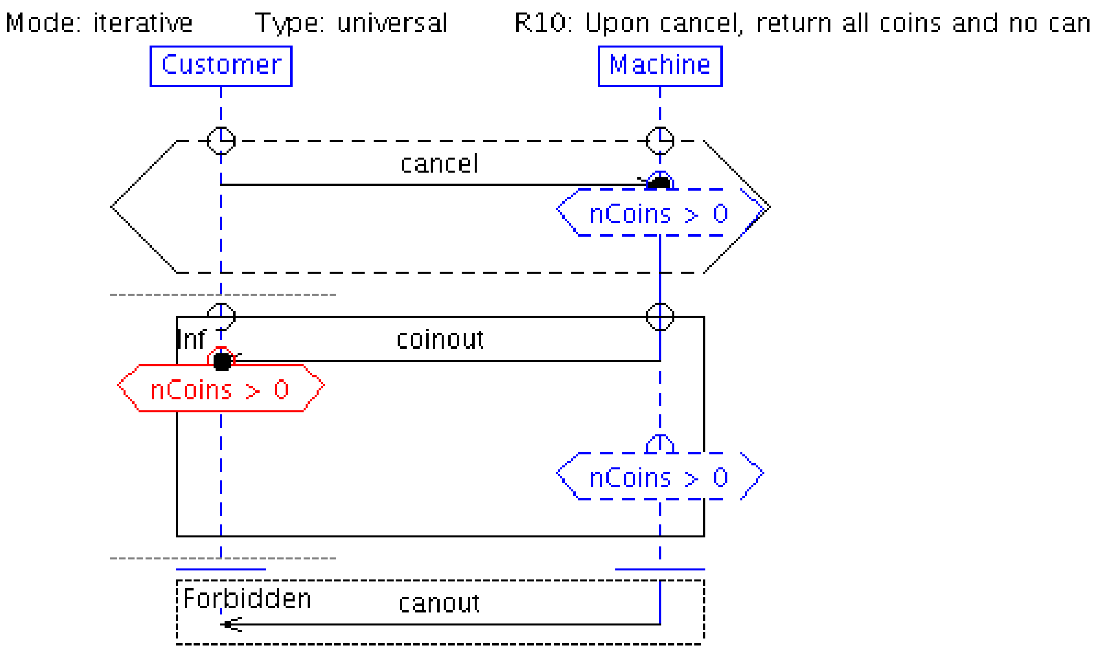

Figure 7.21: Property "Upon pressing the cancel button, the machine returns all coins and no cans" in LSC.

Property	Translated from LSC	Defined in UppAal
6 canout possible	108.19s, 99316KB	0.12s, 2856KB
R1	14.19s, 39096KB	-
R2	19.66s, 39668KB	-
R3	48.79s, 76612KB	-
R4	84.46s, 130032KB	-
R7	not finished	-
R8	17.49s, 39408KB	-
R9	21.36s, 39680KB	-
R10	46.31s, 58000KB	-

Table 7.1: Time and space used by the UppAal model translated from the LSC charts and the original UppAal model while checking the properties validating the model.

7.1.0.4 Results of model checking

One property was added at a time to the model, in order to keep the amount of generated states to minimum.

Property in figure 7.13 has not been satisfied. It indicates that no more cans are sold than inserted into machine by audit, i.e. behavior of the ATM machine is correct.

Property $R2$ has not been violated. It is defined in specification that the can is released only upon sufficient amount of coins inserted.

Property $R8$ is violated because the specification allows inserting coins into the machine, even when there are no cans in it to sell.

7.1.0.5 Performance

The UppAal-specific *memtime* utility has been used to measure time and space usage while checking against each property. The utility has then been used while model-checking the model of ATM constructed directly in UppAal. Ratio of time taken to check the property and memory used indicates how efficient the translation is. Results are presented in table 7.1.

Only a subset of properties in UppAal have their counterpart defined in LSC. Tests have been performed with respect to that subset.

Property "are 6 canout events possible in a row without audit event in between" is not satisfied. Memory used and time elapsed indicates that state space to traverse is much larger (the translated LSC charts result in more global variables and flags, also several copies of the template are maintained). The time elapsed to verify the property in translated LSC specification is proportionally longer. Cause for that is executing many functions and processing the variables that maintain LSC semantics.

Property R1 is not satisfied. The fact is not taken into account that the machine may be out of cans while requesting for one.

Property R2 is satisfied. Requesting enough coins is actually weaker than requesting both enough coins and cans at the machine.

Property R3 is satisfied. It is actually part of the LSC chart used in the specification, where response to *reqcan* is described.

Property R4 holds as well. Its reduced version is the chart in LSC specification which defines how *nCoins* are affected by the *coinin* event.

Property R7 has many instances and its verification could not complete on a notebook.

Property R8 fails. It is because according to specification, the customer can insert coins into machine which has no cans.

Property R9 passes. It proves indirectly that *totalCoins* do not obtain negative value. The specification by means of LSC charts is sound in this case.

Property R10 is satisfied. It mimics one of the requirements in the specification.

7.1.0.6 Conclusions

Several conclusions can be drawn from the example:

- LSC charts can be used both as the pieces of the system specification and the properties to check the system against. It is especially convenient for describing behavior by means of event sequences.
- State of the system can be observed between events thanks to ASAP condition evaluation semantics.
- Because of the ASAP evaluation semantics for conditions, loop constructs in the specification role charts can successfully synchronize and interact with those in the property role charts.

7.1.0.7 Capturing the ATM specification in LSC - comments

Capturing the requirements for particular model have revealed an important point regarding the condition evaluation semantics.

ASAP evaluation semantics for the conditions is necessary when passing data between the charts, i.e. using global variables. Side-effects are still possible when several conditions have assignments attached and are evaluated at a time. Overwriting of data at the same step should be detected early in the model or at the run-time.

7.2 MouseClick

A small example of the intelligent mouse is taken that comes as a demo with the tool UppAal TRON. The requirements for the model are formalized by means of the LSC charts.

Informally, intelligent mouse, given a sequence of clicks from the user, determines whether it has been a single or double click, and sends the outcome to the computer.

There are three actors in the system:

- User
- Mouse
- PC (personal computer)

Three kinds of messages are used in the specification, each corresponding to a real event:

- *click*, which corresponds to the user clicking the mouse
- *singleClick*, which corresponds to the mouse issuing the single click event to the PC
- *doubleClick*, which corresponds to the mouse issuing the double click event to the PC

The sequence of clicks is evaluated against their time of occurrence. Time in the model is measured in time units, where one time unit corresponds to one millisecond. Several time constants are used in the model:

- ϵ is the allowed deviation from the exact time because of the inaccurate hardware and / or software. The default value of ϵ will be 30 time units.
- T_u is the minimal delay between two user clicks. To simulate real user, T_u will be given a value of 75 time units.
- T_s is the threshold for the single click. When the time interval between current and the last clicks from the user is shorter than T_s (including deviation), the sequence of these two user clicks is recognized as double click, otherwise single click. Default value for T_s is 300 time units.
- T_m is the maximal transfer time when the message about single or double click must be sent from mouse to PC after determining its type. Default value for T_m is 50 time units.

The system can be described by the following requirements:

(1) The user may click the mouse in intervals of T_u and larger between the clicks

(2) Whenever the mouse detects a user click, it must issue the *singleClick* event within $T_s + T_u$ time units from that user click, unless *doubleClick* even is issued instead. The following *click* events do not affect the requirement for *singleClick*.

(3) Whenever the mouse detects two user clicks and the *singleClick* after the second *click* event, it means the extra *click* occurring in the situation described by requirement (2). Thus, an extra *singleClick* must be issued within $T_s + T_u$ from the last *click*.

(4) In case of two *click* events with no more than $T_s + \epsilon$ betwen them, the *doubleClick* event is issued not later than T_m time units after the last corresponding click. This can be interrupted by the *singleClick* if the mouse acts according to the requirement (2).

(5) In case of two *click* events, the *singleClick* event and the third *click* event which happens no later than $T_s + \epsilon$ time units after the second *click* event, the *doubleClick* event must be issued within T_m time units after the last *click*.

7.2.0.8 Analysis of requirements

If requirements (2) and (3) are treated as equal, it can be seen that decision whether to issue the *singleClick* or *doubleClick* is made inside the mouse. This means that the situation can occur when the *singleClick* is decided to issue by the mouse, and an additional *click* event is observed before the *singleClick* is actually issued.

If the requirement (2) has higher priority than (3), there still may occur situation where the *singleClick* must be issued until $T_s + T_m$, and there can be no *doubleClick* issued when the second *click* is observed after $T_s + \epsilon$ time units after the first one. Occurrence of *click* in the time window $]T_s + \epsilon, T_s + T_m]$ from the previous *click* must be recorded by means of variables, otherwise many scenarios would be needed to define the behavior of the mouse by means of its inputs (*click*) and outputs (*singleClick*, *doubleClick*).

In order to treat the situations when the next *click* appears before *singleClick* or *doubleClick*, the boolean variable *more* is introduced to record such an occurrence. No more than one such occurrence is possible if $T_m < T_u$, what is true with the predefined vales of these constants. When such inequality holds, there is no such situation that three *click* events in a row occur before the *doubleClick* event.

7.2.0.9 The UPPAAL model based on requirements

Based on the requirements of section 7.2.0.8, the UppAal model is built that is later used to compare the performance and error detection capability of the LSC-based model. The UppAal model consists of exactly three process templates that correspond to the processes of $User$, $Mouse$ and PC.

In the UppAal model, communication channel names and global variables are left exactly the same as defined for the LSC specification. The only difference is in the templates and communication channel types. All the communication channels are defined as binary since for each channel and sender process there is exactly one receiver process. In particular, *click* channel is directed from $User$ to $Mouse$, and *singleClick* with *doubleClick* channels are directed from the $Mouse$ to the $User$ process.

The UppAal model has been very helpful to understand the state machine based functioning of the smart mouse. The model has been simulated to find out the possible event

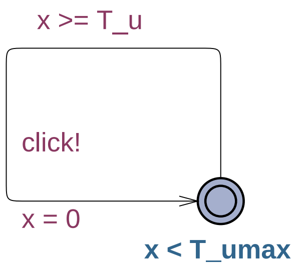

Figure 7.22: The UppAal process template of the *User*. The requirement not to issue the *click* events too often is implemented through the transition guard. Events are forced to happen at least every *T_umax* time units, this is implemented through the location invariant.

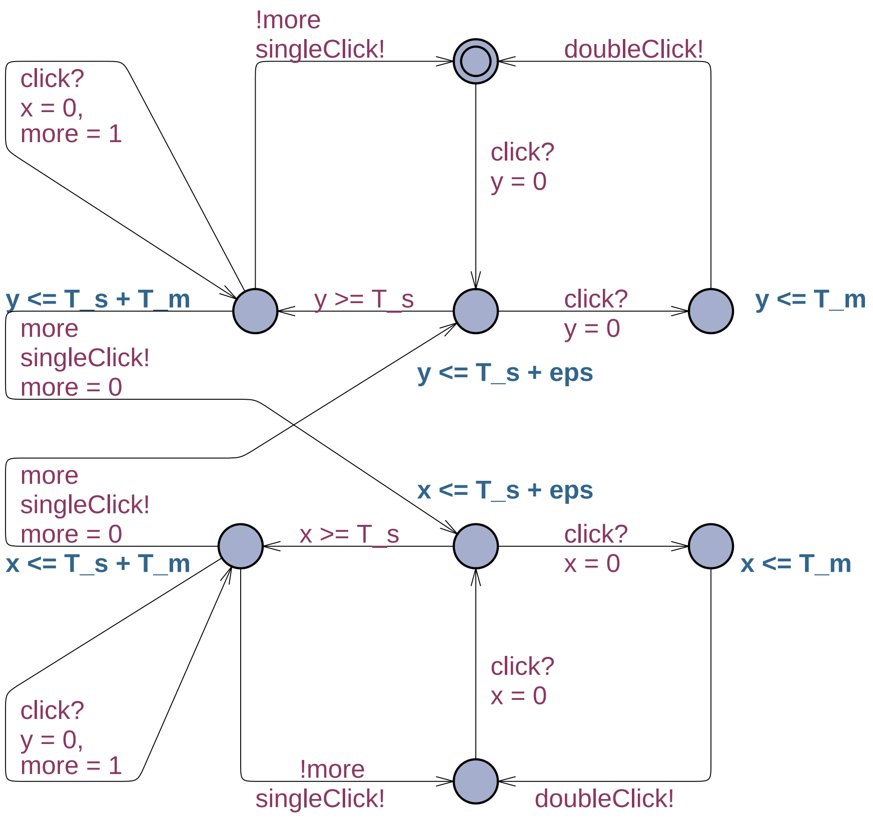

Figure 7.23: The UppAal process template of the *Mouse*. The upper part of the template is a trivial implementation of the reaction to the *click* event sequences that correspond to the single click or double click situations. The lower part is the mirror of the upper part. It is used when additional *click* event is observed when the *Mouse* has already decided what output it will issue. The additional *click* event is then taken into memory, its timer is started and the *more* flag is used. The process changes sides every time an additional *click* is observed.

sequences for the smart mouse.

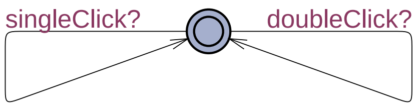

Figure 7.24: The UPPAAL process template of the *PC*. It does nothing but consume the *singleClick* and *doubleClick* events issued over binary communication channel.

7.2.0.10 Requirements captured as LSC scenarios

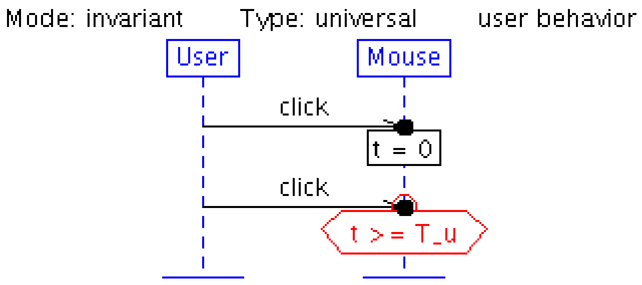

Figure 7.25: Requirement (1) captured in a LSC chart. The minimal user click interval is enforced by the hot condition which is satisfied only when the time between a click and the click afterwards is equal or larger than T_u. The LSC chart has no prechart, what means that the chart itself keeps activating from time to time.

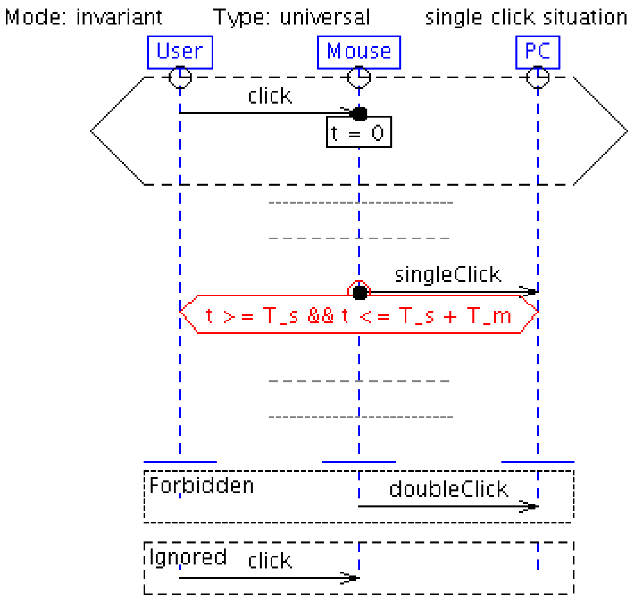

Figure 7.26: Requirement (2) captured in LSC chart.

It should be noted that the translated LSC specification does not have the invariants attached to the locations that correspond to the cuts. As a result of that, the sequence of

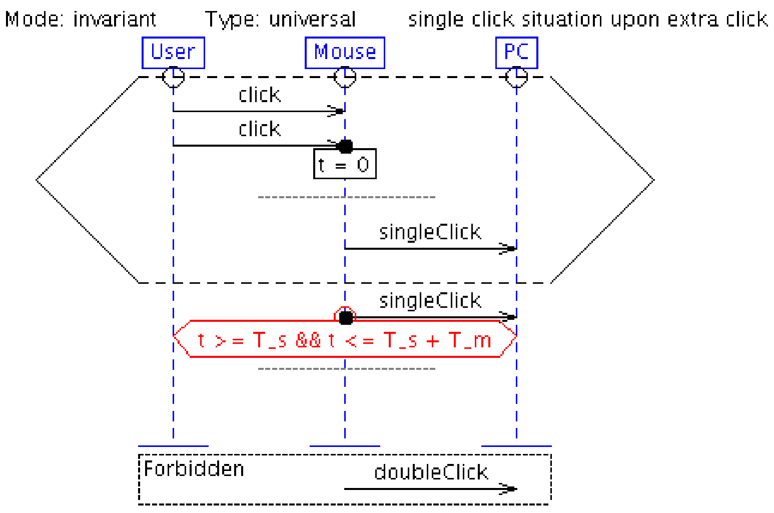

Figure 7.27: Requirement (3) captured in LSC chart.

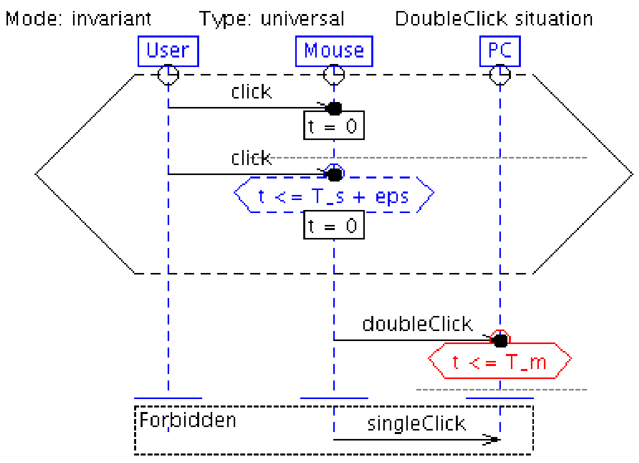

Figure 7.28: Requirement (4) captured in LSC chart.

events *click* a time t, another *click* at time $t_1 > t + T_s + \epsilon$ and the third *click* at time $t_2 > t + T_s + \epsilon + T_u$ can be performed on the translated model. However, this sequence puts the specification into deadlock, since the *singleClick* event has not been produced between the first and second occurrence of *click*. The UPPAAL based model of the smart mouse does not allow such combinations because of its invariants on certain locations.

The absence of invariants on the stable locations of the timed automata from translated LSC specification may seem a drawback. On the contrary, this provides an easy way to implement the quiescence mechanisms inside the model.

In spite of difference in the UPPAAL and translated LSC models, they both are equally suitable for the conformance test since it is the IUT which is responsible for producing the timely output *singleClick*.

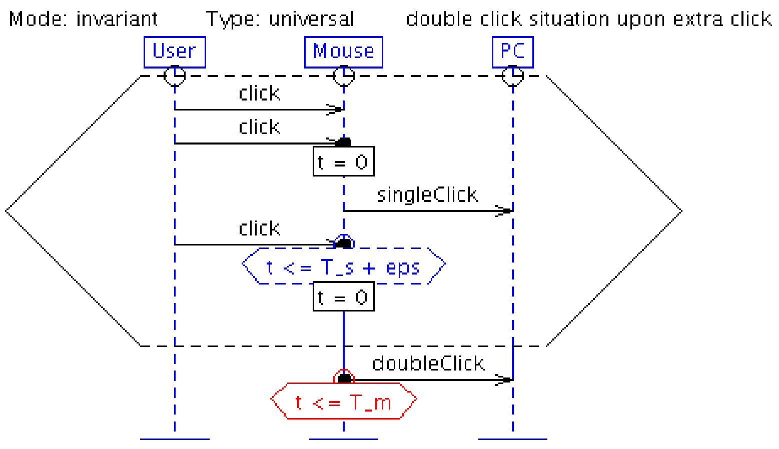

Figure 7.29: Requirement (5) captured in LSC chart.

7.2.0.11 Properties to check the model specified as the LSC scenarios

Model checking is the way to check whether defined model satisfies some properties. A set of properties has been defined for the mouse click model:

(1) *User can click the mouse 3 times before the mouse reacts to these clicks.* The property would expose a bug if satisfied, since the mouse must always send a single or double click event based on at most 2 user clicks.

(2) *One mouse click can provoke a single and then a double click events.* This property also describes unwanted behavior of the mouse.

(3) *One mouse click can provoke a single and then a single click events.*

(4) *One mouse click can provoke a double and then a double click events.*

(5) *One mouse click can provoke a double and then a single click events.*

Properties are defined by means of LSC charts as follows.

7.2.0.12 Results of model checking the specification against the property scenarios

Properties in general have taken much time to check, as no upper bound specified for the interval between 2 clicks.

Property (1) is possible to observe, this implies under-specification of the model. The double click-specifying LSC should strictly forbid subsequent clicks from the user after two clicks are made.

Properties (2)-(5) are never observed. It means, the mouse does not generate unexpected events.

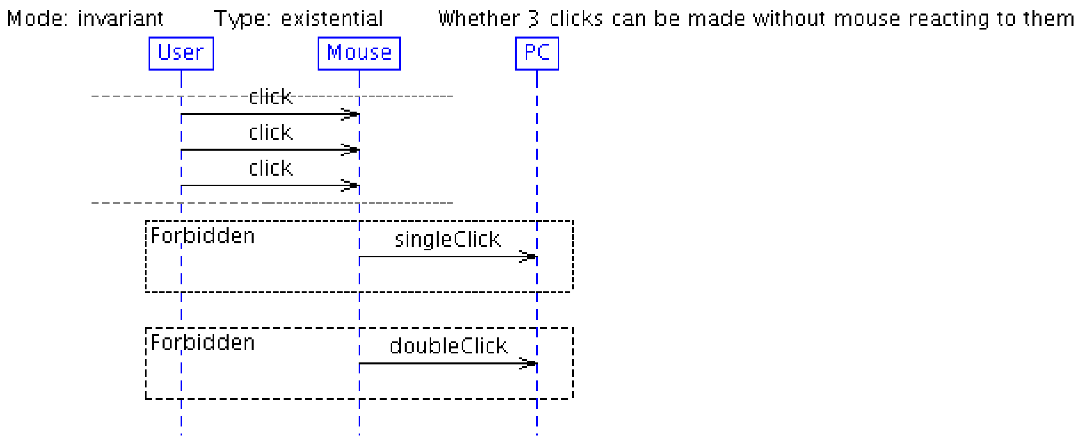

Figure 7.30: Property (1) captured as the LSC chart.

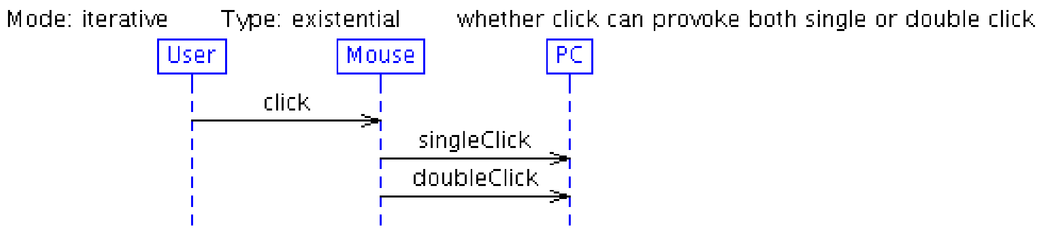

Figure 7.31: Property (2) captured as the LSC chart.

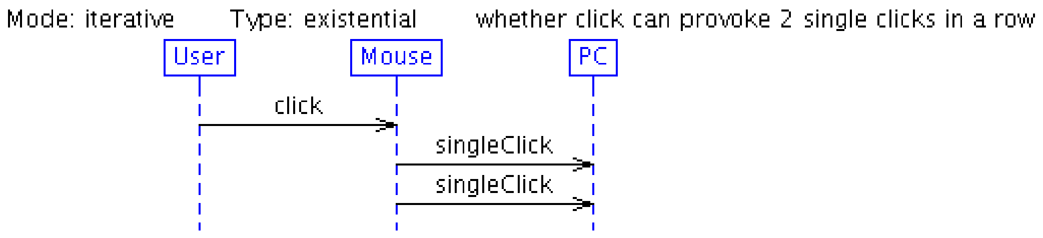

Figure 7.32: Property (3) captured as the LSC chart.

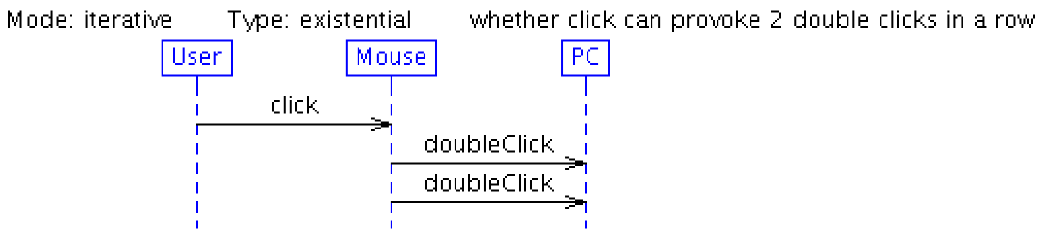

Figure 7.33: Property (4) captured as the LSC chart.

7.2.1 Implementation under test

Implementation under test is the JAVA application. There is a single actor, *Mouse*, running as a thread. The other actors are assumed to belong to the environment. In the experiment, the IUT runs in the same address space as the UPPAAL TRON tool, and input and output actions are communicated to and from the driver/adapter via two single place bounded buffers.

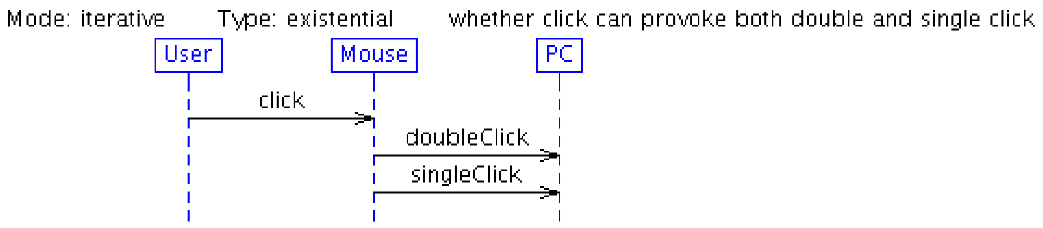

Figure 7.34: Property (5) captured as the LSC chart.

In order to check the error detection capability, a set of implementation mutations for smart mouse have been developed. In all the mutants, the constant T_m is zero, i.e. the mouse outputs the *singleClick* or *doubleClick* immediately after detecting the corresponding situation or a timeout. The mutants are defined as follows:

M0 , the original implementation, where T_s time units pause after the last *click* always triggers the *singleClick* immediately, otherwise *doubleClick* is triggered immediately. In other words, the timeout is exactly T_s time units.

M1 , similar to $M0$ but the timeout is $T_s + \epsilon$ time units instead of T_s

M2 , where timeout is $T_s - \epsilon$ time units instead of T_s

M3 , similar to $M0$ but the timeout is $T_s + 2 * \epsilon$ time units instead of T_s

M4 , where every tenth *singleClick* is substituted with *doubleClick*

M5 , where every tenth *doubleClick* is substituted with *singleClick*

The mutants are tested against both models, that defined in UppAal and another one, which has been originally defined in LSC and then translated. Mutants $M0$ and $M1$ are supposed to pass since they still act according to the rules of non-deterministic specification. The rest of mutants are inconsistent with the specifications and thus their conformance test against the specifications is supposed to fail.

7.3 Error detection capability

The purpose of testing the implementations for conformance against the models is dual. First, it is expected that the model is specific enough to detect disrepancies of the mutant behavior. Secondly, performance of the translated LSC specification is compared with that originally defined in UppAal.

Simulated clock has been used in experiments to allow for faster and more experiments and reduce potential problems with real- time clock synchronization between the TRON and IUT.

Experiment against the UppAal model is run 1000 times for the mutants against each of the two models. Maximal test duration against both models is set to 100.000 time units. In the UppAal model, the delay between the inputs *grasp* and *release* is encoded in the

Mutant	Min clicks	Avg clicks	Max clicks	Min TUs	Avg TUs	Max TUs
$M2$ UppAal	1	1,1	5	346,0	1108,9	3215
$M2$, LSC	1	1,8	11	270	1878,4	10755
$M3$, UppAal	1	3,2	20	414	2164,1	10207
$M3$, LSC	1	2,4	17	336	2331,7	14058
$M4$, UppAal	10	11,2	18	3309	7735,1	17969
$M4$, LSC	10	21,2	44	5521	18083,9	34419
$M5$, UppAal	69	179,2	359	42051	117849.0	243622
$M5$, LSC	22	37,6	65	11405	31128,9	64165

Table 7.2: Conformance test results of the smart mouse mutants M2-M6 against the corresponding translated LSC and UppAal models. Mutants M0 and M1 conform to specification and conformance test has 100% passed with them.

environment automaton. In the translated LSC specification there is no such restrictions on the inputs, equivalent restrictions are defined by means of the TRON command line arguments.

For each test run the number of communicated observable actions and total running time until the error is found is recorded. The minimum, maximum, and average running time and number of used input actions of the failing tests are summarized in table 7.2. Segmentation faults have been observed when running the tests, and they vary dependent on the machine where the tests are run. Tests resulting in segmentation faults are not included in the statistics.

No inconclusive tests have been observed during the test session. The mutants $M0$ and $M1$ passed all the tests against the UppAal and LSC specifications. Other mutants have failed all tests altogether, independently from the specification.

7.3.1 Test results

Minimal amount of clicks needed to reveal the mutants $M2$ and $M3$ are 1. For the mutant $M4$, 10 clicks are needed at least, each of them being responded by *singleClick*. Mutant $M5$ needs at least 20 clicks, that would result in 10 consecutive *doubleClick* situations, the tenth one being substituted inside IUT with a *singleClick*.

Several conclusions can be made based on the experiment results:

- For mutants $M2$ through $M4$ and ($M5$, LSC), the lower bound of the inputs until failure coincides with the fewest possible inputs needed to detect it. For the mutant ($M5$, UppAal) surprisingly, the lower bound of inputs until failure (69) is much larger than necessary (20). The minimal amount of clicks observed in the ($M5$, LSC) test setup is 22, what is close enough to the theoretic minimum (20).
- The average trace length to detect the fault does not differ more than two times for the mutants $M2$ through $M4$. The mutant $M5$ needs more than four times more inputs against the UppAal specification than the LSC specification.
- Judging by the average amount of inputs until mutant detection, there is no clear answer

whether the UppAal or LSC specification is better in particular conformance test. Such result is expected since the specifications capture the behavior of the same system in different ways; the UppAal model captures the state-based aspects of the smart mouse, while the LSC model, partially derived from the former, captures requirements by means of the input and output sequences.

7.3.1.1 Conclusions

Some observations should be made regarding the mode of the chart:

- When the universal chart consists of only prechart or a single construct like message, assignment, condition or simregion thereof, chart of iterative and invariant mode of the chart results in same behavior.

  Comments regarding the specification constituted by the LSC charts are as follows:

- The LSC charts have been easily used to capture behavioral requirements. For particular model, the requirements captured are rather simple, and the meaning of the LSC charts is obvious.
- Under-specification of the model has been observed during the model checking. The model checking against TCTL and LSC properties is thus necessary to validate the specifications, whenever the size of the specification allows that. The under-specification can be solved through slight alteration of a single LSC chart (the one which specifies the double-click minimal or maximal time). Easiness to alter and correct the specification through altering minimal amount of its LSC charts is an attractive feature of the LSC charts.
- LSC charts is a very flexible formalism when it comes to aggregation of the scenarios. Five simple scenarios in figures 7.25 to 7.29 constitute the specification.

  Regarding the conformance test against

7.3.1.2 Improvements

The model is semantically correct, but because open upper bound between clicks, its model checking is complicated. However, such a model is successfully used in simulation and on-line test.

7.4 Dynamic Host Control Protocol - the Client part

The Dynamic Host Control Protocol (DHCP) [Dro97] has been chosen as the implementation under test for several reasons. The client part of the protocol has been implemented in the Ericsson A/S (Telebit). It has been one of our tasks to write the manual test cases for some parts of the DHCP client state machine. As a result of that, we are aware of the manual test framework and the parts of the protocol that can have the automated test applied.

The framework of the manual test has been adapted to support the on-line conformance test. The specification of the DHCP client state machine has been defined in LSC and translated into the UppAal, while the requirements for the data part of the protocol have been largely omitted. The test itself has run and the implementation found compliant to the state-machine specification of the DHCP client.

No mutants of the DHCP client have been defined and no error detecting capabilities measured against the model of translated LSC charts. The reason for that is that only a part of the protocol specification has been modeled, and the model does not include requirements on time constraints. Suitability of the translated LSC specification with time constraints is discussed in the Intelligent Mouse case (section 7.2).

The framework of the manual test is introduced in section 7.4.1. It is described in section 7.4.2 about the parts of the test framework to be changed and aims defined to achieve by automatic testing. Implementation of the test framework is presented in section 7.4.4, and results of the automated test with the conclusions are in sections 7.4.4.1 and 7.4.5, respectively.

7.4.1 DHCPC test framework prior to automation

It is necessary to describe the framework that has been used prior to the test automation in Ericsson A/S (Telebit). The LSC charts based specification and the tool UppAal TRON have been added to the framework at some later point, and it is in our interest to compare the advantages and drawbacks of the changes.

The DHCP client is referred to as the DHCPC, which is the shorthand notation.

7.4.1.1 Test cases for the DHCPC

The test cases are written in Python. A number of the test cases are written, and they altogether constitute the test batch. When the batch is run, testcases get executed one by one, and at the end of the batch run, the number of passed and failed test cases is returned. In every test case, the DHCPC process is spawned from the executable, and communication takes place between the Python that interprets the test case code and the DHCPC.

7.4.1.2 Actors

There are only two actors in the famework, that are the DHCP client and its environment. The DHCPC process stands for the client, and the Python test script for its environment. The environment can simulate the DHCP server, other DHCP clients and arbitrary hosts with their IP addresses assigned.

7.4.1.3 Architecture of the tests

The architecture of the DHCPC tests is displayed in figure 7.35.

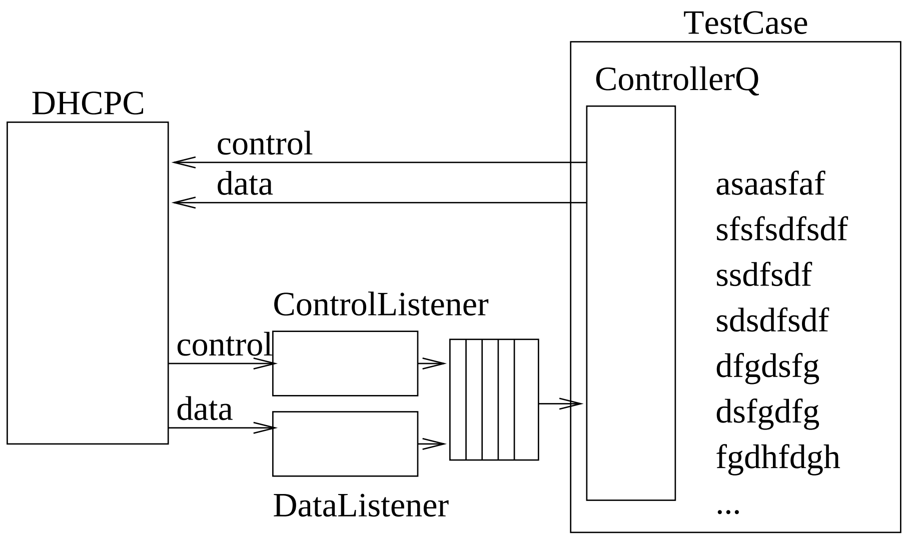

Figure 7.35: The DHCPC testing framework before introduction of test with UppAal TRON.

The following notations have been used in figure :

DHCPC is the executable that maintains the DHCP client functionality. It is implementation under test.

ControllerQ stands for the interface class between the test case and the DHCPC. The *ControllerQ* class written in Python defines the communication with the DHCPC executable through two interfaces. One interface (the so-called control interface) is used between the test case and the embedded Python module within DHCPC. Another one, called data interface, is between the test case and the lower part of DHCPC which is supposed to be a network interface. Communication takes place by means of messages through the file sockets.

data stands for the DHCP and other communication protocol packets that in real life are exchanged between DHCP server, DHCP client and other hosts. The data interface is used to transmit the data between the DHCPC and the test case.

control stands for the commands sent via the control interface directly to or from the Python module running inside the implementation. Typically, the DHCP client is started, reset and stopped that way. Some functions inside the DHCPC can also report its progress, yielding more information about the status of the DHCPC and thus easing the test process.

DataListener is the thread that collects the DHCP and other messages from the DHCPC, that are supposed to be sent through the ethernet interface in the real implementations. The messages are put into the queue for the further processing by the test script.

ControlListener is the thread that collects the control information from the DHCPC and adds it into the queue. A single queue is used both for the data and control information, to resolve order of its occurrence.

7.4.1.4 DHCP message data format

Communication between the Python test case and the DHCPC process has used the DHCP messages as one type of communication. There are several fixed parameters of the DHCP message:

Type the essential fixed parameter. The types of the DHCP mesages can be found in section 7.4.2.1. In most cases, correct DHCP messages have been constructed during the test and sent to the DHCPC. Sometimes they have some of their fields deliberately changed to check whether the DHCPC discards the improperly formed messages.

Format Messages of the same type have different formats dependent on the current state of the DHCP client. Using only the message type is not enough to deduce the state of the DHCPC client. To have a better view of the DHCP client's current state, additional checks are performed besides the message type analysis.

7.4.2 Guidelines for the automated test

There are several guidelines for the automated test:

- The DHCP client implementation is going to be tested against its state machine behavior. The requirements for the state machine will be collected from [Dro97] and the LSC specification composed of those.
- The inputs to the IUT will be the DHCP mesages and the user commands to start, close the DHCPC or renew the obtained IP address.
- The only outputs considered from the IUT will be the DHCP messages and the ARP message.
- Checks for the DHCP packet correctness will not be performed during the conformance test. However, ability to do this must be left and it must be easy to implement.
- The existing test framework is used as the starting point in construction of the test framework adapted for conformance test.

Inputs and outputs of the IUT are discussed in section 7.4.2.1. Implementation of the test framework is presented in section 7.4.4.

7.4.2.1 Modeling actors and communication among them

Simple setup with only one DHCP client and one DHCP server has been assumed during the previous DHCP tests, so does it remain in the automated tests. The system consists of three actors in total, namely the DHCP client, the DHCP server and the user that starts, stops the DHCP client and triggers its IP address renewal:

User the user that starts, stops the DHCP client and triggers its IP address renewal

Client the DHCP client

Server the DHCP server

The User is the only stateful actor in the system. It has the variable *cstate* which can be assigned one of the values similar to those defined in [Dro97]:

- S_OFF (initial state, when the Client is inactive)
- S_INIT
- $S\_SELECTING$
- $S\_REQUESTING$
- S_ARPING
- S_BOUND
- $S\_RENEWING$
- $S\_REBINDING$
- $S\_INITREBOOT$
- $S\_REBOOTING$

There is no direct communication between $User$ and $Server$ defined.

Communication $User$ - $Client$ is defined as one-way. It consists of the following messages:

u_start the $User$ activates the $Client$.

u_end the $User$ deactivates the $Client$.

u_renew the $User$ triggers the $Client$ to renew its lease of the IP address.

Traditionally, the test scripts have had the functionality of both the $User$ and $Server$.

Communication $Client$ - $Server$ is bi-directional.

Set of messages from $Client$ to $Server$ are available, and several versions of DHCPREQUEST message are defined as in the table 4 of RFC 2131 (DHCP):

- the DHCP-specific messages:

  DHCPDISCOVER , named $m\_discover$

  DHCPREQUEST issued from $S\_SELECTING$ or $S\_REQUESTING$ state, named $m\_request\_se$

  DHCPREQUEST issued from $S\_RENEWING$ state, named $m\_request\_ne$

  DHCPREQUEST issued from $S\_REBINDING$ state, named $m\_request\_bi$

  DHCPREQUEST issued from $S\_INITREBOOT$ or $S\_REBOOTING$ state, named $m\_request\_ir$

  DHCPDECLINE named $m\_decline$

  DHCPRELEASE named $m\_release$

  DHCPINFORM named m_inform

- Other protocol (ARP) messages. It is assumed that the *Server* is the only other entity on the local network, acting as all the other hosts and capable of receiving messages from the *Client*.

  ARP a la m_arp, address resolution protocol packet, used to probe whether any other host on the local network has same IP address as requested by the *Client*.

Set of messages from *Server* to *Client* are available:

- the DHCP-specific messages:

  DHCPOFFER , named m_offer

  DHCPACK , named m_ack

  DHCPNAK , named m_nak

- Other protocol (ARP) messages. It is assumed that the *Server* is the only other entity on the network, capable of receiving messages from the *Client*.

  RARP , named m_rarp, address resolution protocol packet, used to reply to the sender that a host with specified IP address exists on the network.

The messages are formed at the glue layer by the same functions of the class *ControllerQ* as used in the previously written test cases. The DHCPREQUEST messages are abstracted into four abstract messages, dependent on the message properties that uniquely reflect the state of the *Client*.

7.4.3 DHCP client test cases - requirements

In this section, the set of requirements are defined for the DHCP client state machine [Dro97]. The requirements are defined by means of scenarios that consist of inputs and outputs. The input and outputs themselves are abstracted DHCP messages, DHCP client control commands and auxiliary messages such as ARP or RARP messages, defined in section 7.4.2.1.

7.4.3.1 DHCPC client state machine

Figure 7.36 represents the DHCP client state machine. The picture is taken from the Online TCP/IP guide (http://www.tcpipguide.com).

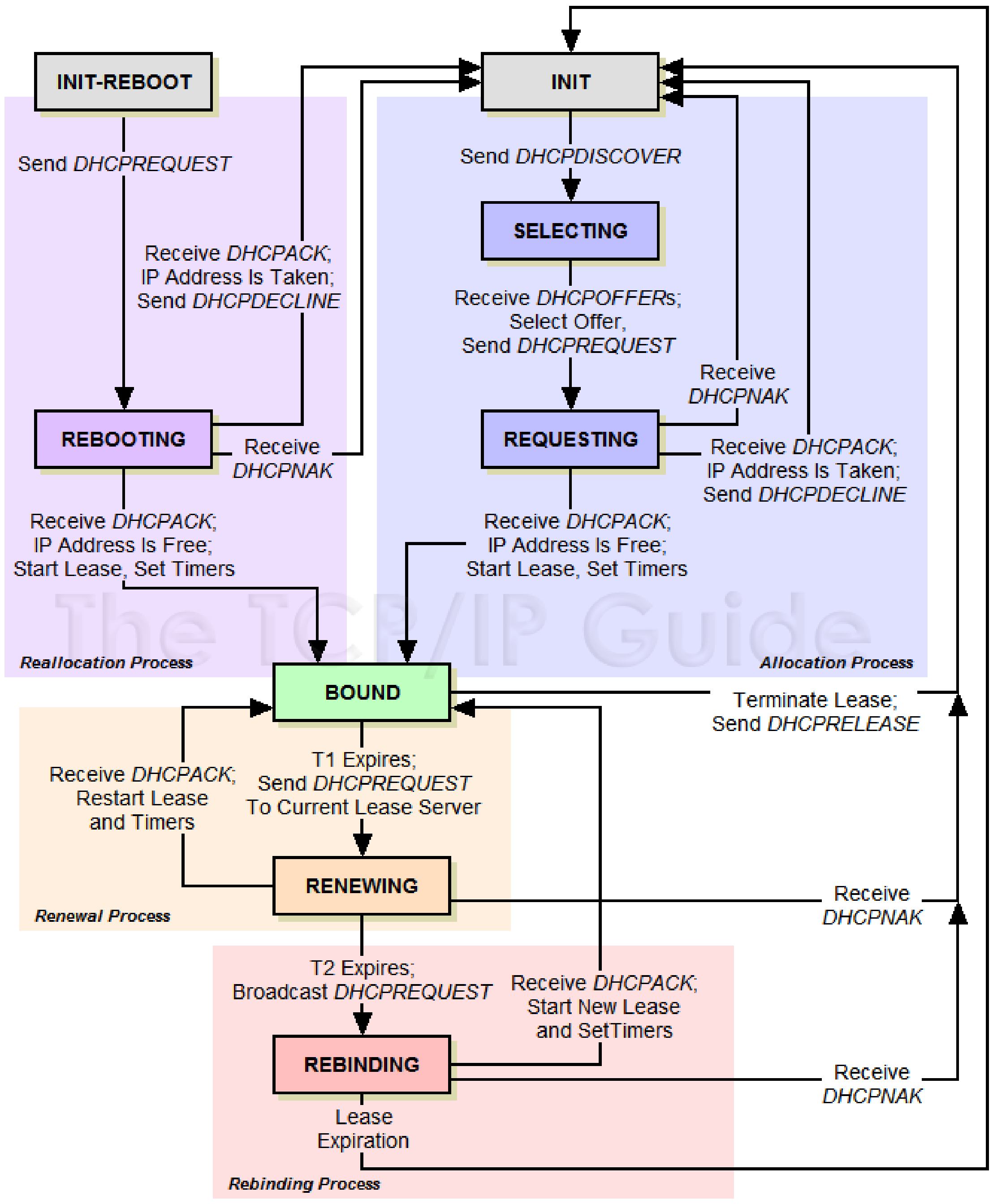
INIT-REBOOT
INIT
Send DHCPREQUEST
Send DHCPDISCOVER
SELECTING
Receive DHCPACK; IP Address Is Taken; Send DHCPDECLINE
Receive DHCPOFFERs; Select Offer, Send DHCPREQUEST
Receive DHCPNAK
REBOOTING
Receive DHCPNAK
REQUESTING
Receive DHCPACK; IP Address Is Taken; Send DHCPDECLINE
Receive DHCPACK; IP Address Is Free; Start Lease, Set Timers
Receive DHCPACK; IP Address Is Free; Start Lease, Set Timers
The TCP/IP Guide
Reallocation Process
Allocation Process
BOUND
Terminate Lease; Send DHCPRELEASE
Receive DHCPACK; Restart Lease and Timers
T1 Expires; Send DHCPREQUEST To Current Lease Server
RENEWING
Receive DHCPNAK
Renewal Process
T2 Expires; Broadcast DHCPREQUEST
Receive DHCPACK; Start New Lease and SetTimers
REBINDING
Receive DHCPNAK
Lease Expiration
Rebinding Process

Figure 7.36: The finite state machine of the DHCP client. The state *arping* is missing in the picture since this is an original state machine. The *arping* state is between the *requesting* and *bound* state, and transition from *arping* to *bound* occurs on time-out. Picture taken from the Online TCP/IP guide (http://www.tcpipguide.com).

Abbreviation of the DHCPC states and mesages Due to the number of states in DHCP client used and requirements used to define the behavior, conventions are necessary to refer conveniently to the requirements given the state and message and optionally other information.

The abbreviation of the DHCP client states are as follows:

OF for state S_OFF

IN for state S_INIT

SE for state $S\_SELECTING$

RE for state $S\_REQUESTING$

AR for state S_ARPING

BO for state S_BOUND

NE for state $S\_RENEWING$

BI for state $S\_REBINDING$

IR for state $S\_INITREBOOT$

RB for state $S\_REBOOTING$

Abbreviation of the DHCP, ARP messages and the control inputs for the DHCP client are given the following abbreviations:

ust for u_start

urn for u_renew

uen for u_end

dis for $m\_discover$

rse for $m\_request\_se$

rne for $m\_request\_ne$

rbi for $m\_request\_bi$

rrb for $m\_request\_ir$

dec for $m\_decline$

rel for $m\_release$

inf for m_inform

ofr for m_offer

ack for m_ack

nak for m_nak

arp for m_arp

rrp for m_rarp

7.4.3.2 Scope of the requirements

The requirements are limited to non-timed requirements. Primary focus of the requirements is to specify what outputs can be produced by the client from each of its states. The client is modeled as an input-enabled machine, and in practice it is indeed so.

The complete set of requirements used for test is supposed to describe what outputs the DHCP client can produce from particular state and what outputs are forbidden from their states. The requirements also describe into what state the DHCP client transits upon certain combination of inputs and outputs.

The timed requirements can be added at the later stage. This is easy to do because of the flexibility of LSC charts in capturing the behavioral requirements.

7.4.3.3 Groups of requirements

The requirements are supposed to dissect the DHCP client model as the Mealy machine into its constituting parts, that are states and transitions.

Having at hand a number of techniques and strategies of how to formalize the requirements, several subgroups of requirements for the DHCP client and its environment (the DHCP server and the $User$) will be presented. The subgroups are as follows:

- The behavior of the $User$
- The DHCP client allowed transitions and behavior
- Limitations on the DHCP client behavior
- The DHCP server allowed transitions and behavior
- Limitations on the DHCP server behavior

Allowed behavior of the $User$ The set of LSC charts describing the $User$ behavior is the simplest set, consisting of universal charts with single message event in each. The messages triggered by the user are *u_start*, *u_renew* and *u_end*. So are the charts labeled respectively:

ust_ , "the $User$ can trigger the event *u_start*"

urn_ , "the $User$ can trigger the event *u_renew*"

uen_ , "the $User$ can trigger the event *u_end*"

The chart labeled *ust_* is presented in figure 7.37. The LSC charts of the other two requirements only have different message label.

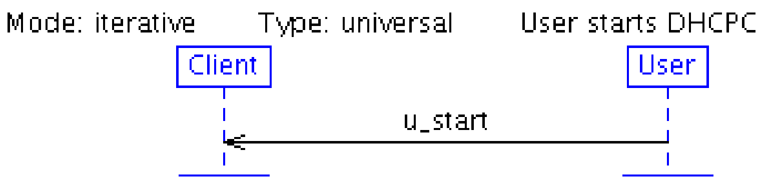

Figure 7.37: Requirement "the *User* can trigger the event *u_start*" captured as the LSC chart.

Allowed transitions and behavior for DHCP client This subgroup specifies what transitions and behavior is allowed by the DHCP client according to the DHCP client state machine. The subgroup involves sets of requirements based on:

1. What state the DHCP client transits to or what output it produces, given its current state and the input
2. What outputs from the DHCP client are enabled from particular state
3. What state the DHCP client transits to, given its current state and the produced output

The requirements of the first set would be depicted as the universal charts with the sim-region in the prechart. The predicate over the current state would be placed in postcondition, and update of the state would be placed at the postaction.

The notation of the chart is of the form *_msg_STATE*, where *msg* is the abbreviation of the message received or produced by the client, and the *STATE* is the state it transits to.

Another notation is *msg1_msg2_*, what means that upon receiving message *msg1*, the client issues the message *msg2*.

The states are not mentioned in the abbreviation, where the client resides at the point of receiving the message.

Requirements for the first set are as follows:

_ust_IN , "upon receiving the event *u_start*, the client moves to state *S_INIT*". The requirement holds in state *S_OFF*.

_urn_IR , "upon receiving the event *u_renew*, the client moves to state *INIT_REBOOT*". The requirement holds in states *S_BOUND*, *S_RENEWING* and *S_REBINDING*.

_uen_rel , "upon receiving the *u_end* message from *User*, the client must issue the *m_release* message to the server". The requirement holds in states *S_BOUND*, *S_RENEWING* and *S_REBINDING*.

_ofr_RE , "upon receiving the *m_offer* message, the client either transits to the state *S_REQUESTING* or remains in same state". The requirement holds in state *S_SELECTING*.

_nak_IN , "upon receiving the *m_nak* message, the client transits to the state *S_INIT*". The requirement holds in states *S_REQUESTING*, *S_RENEWING*, *S_REBINDING* and *S_REBOOTING*.

_ack_BO , "upon receiving the *m_ack* message, the client transits to the state *S_BOUND*". The requirement holds in states *S_REBINDING*, *S_RENEWING* and *S_REBOOTING*.

_ack_AR , "upon receiving the *m_ack* message, the client transits to the state *S_ARP*". The requirement holds in state *S_REQUESTING*.

_rrp_dec , "upon receiving the *m_rarp* message, the client issues the *m_decline* message". The requirement holds in state *S_ARPING*.

_dec_IN , "upon receiving the *m_decline* message, the client transits to the state *S_INIT*". The requirement holds in states *S_ARPING*.

_rrb_RB , "upon receiving the $m\_request_i r$ message, the client transits to the state *S_REBOOTING*". The requirement holds in state *S_INITREBOOT*.

The chart labeled *_ust_IN* is presented in figure 7.38. It is typically a message observed in prechart, and the state change occurs in the same simregion where the message is involved. This LSC chart is the template to specify the DHCP client state change upon receiving a message.

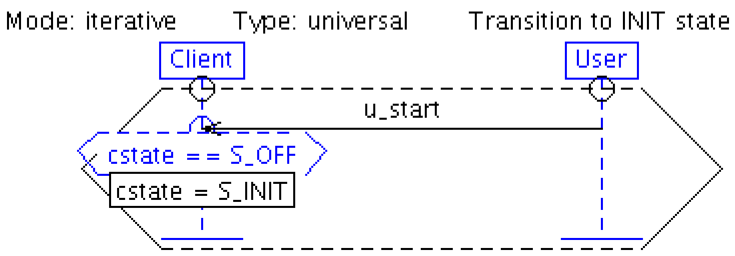

Figure 7.38: Requirement "upon receiving the event *u_start*, the client moves to state *S_INIT*" captured in the LSC chart.

The chart labeled *_uen_rel* is presented in figure 7.39. It is typically a message observed in prechart, and the response message occurring outside the prechart, namely meaning that the response message is obligatory. This LSC chart is the template to specify the DHCP client response with a message upon receiving a message.

The second set of requirements would be typically depicted as the universal charts without prechart. These charts would be activated and corresponding events triggered if no restrictions apply to them under current state of the client. The events would have preconditions such as the check over suitable states to emit the event.

The notation for the charts of this set are *STATE_msg_* what means that message *msg* can be emitted from state *STATE*. Also, notation *STATE1_STATE2* will be used what

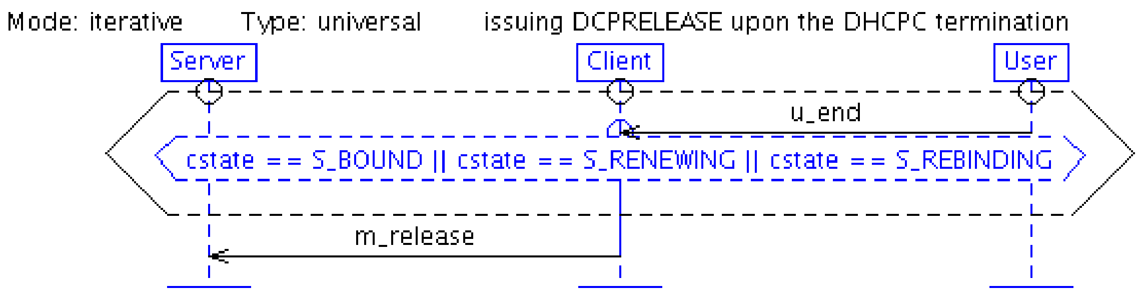
Mode: iterative
Type: universal
issuing DCPRELEASE upon the DHCPC termination
Server
Client
User
u_end
cstate == S_BOUND || cstate == S_RENEWING || cstate == S_REBINDING
m_release

Figure 7.39: Requirement "upon receiving the *u_end* message from *User*, the client must issue the *m_release* message to the server" captured in the LSC chart.

means silent transition from state $STATE1$ to state $STATE2$, which in reality happens upon time-out.

IN_dis_ , "the client can issue the *m_discover* message while in state *S_INIT* or *S_SELECTING*".

SE_rse_ , "the client can issue the *m_request_se* message while in state *S_REQUESTING*".

AR_arp_ , "the client can issue the *m_arp* message while in state *S_ARPING*.

AR_BO , "the client transits from state *S_ARPING* to the state *S_BOUND*". The requirement holds in state *S_ARPING*.

BO_inf_ , "the client can issue the *m_inform* message while in state *S_BOUND*.

BO_NE , "the client transits from state *S_BOUND* to the state *S_RENEWING*". The requirement holds in state *S_BOUND*.

NE_rne_ , "the client can issue the $m\_request_ne$ message while in state *S_RENEWING*".

NE_BI , "the client transits from state *S_RENEWING* to the state *S_REBINDING*". The requirement holds in state *S_RENEWING*.

BI_rbi_ , "the client can issue the $m\_request_bi$ message while in state *S_REBINDING*".

BI_IN , "the client transits from state *S_REBINDING* to the state *S_INIT*". The requirement holds in state *S_REBINDING*.

IR_rrb_ , "the client can issue the *m_request_ir* message while in state *S_INITREBOOT* or *S_REBOOTING*".

RB_IN , "the client transits from state *S_REBOOTING* to the state *S_INIT*". The requirement holds in state *S_REBOOTING*.

The chart labeled *IN_dis* is presented in figure 7.40. It is typically a message in mainchart, and the state from which the client cansend the message, is restricted by the condition in

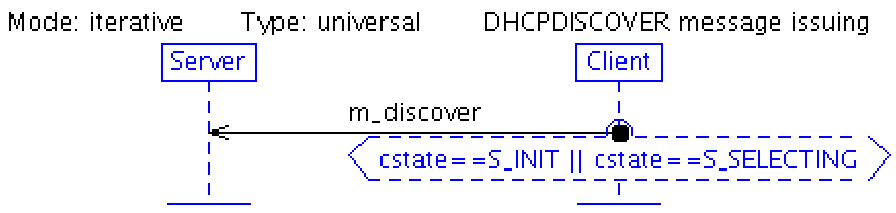
Mode: iterative
Type: universal
DHCPDISCOVER message issuing
Server
Client
m_discover
cstate==S_INIT || cstate==S_SELECTING

Figure 7.40: Requirement "the client can issue the *m_discover* message while in state *S_INIT* or *S_SELECTING*" captured in the LSC chart.

the simregion. This LSC chart is the template to specify the DHCP client message sending capabilities regarding its state.

The chart labeled *AR_BO* is presented in figure 7.41. It is a simregion in mainchart (although it can be in prechart, as there is no difference for the simregion without message). Simregion includes the cold condition that restricts the current state, and the assignment which updates the current state of the DHCP client. The LSC chart is the standard template to specify the silent state change (i.e. when no message is consumed or transmitted).

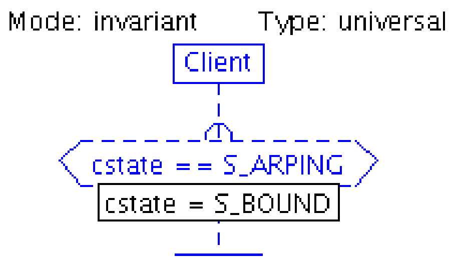

Figure 7.41: Requirement "the client transits from state *S_ARPING* to the state *S_BOUND*" captured in the LSC chart.

The third set of the requirements would be typically depicted as the universal charts with the simregion in the prechart. checking of the state and update of the state would be depicted as the postcondition and postaction, respectively.

Notation of the requirements is *_msg_STATE*, what means that upon issuing the message *msg*, the client transits to the state *STATE*.

_dis_SE , "upon issuing the *m_discover* message, the client transits to the state *S_SELECTING*". The requirement holds in state *S_INIT* and *S_SELECTING*.

_rel_OF , "upon issuing the *m_release* message, the client transits to the state *S_OFF*". The requirement holds in states *S_BOUND*, *S_RENEWING* and *S_REBINDING*.

The requirements of the third set are captured in the same way as the one captured in figure 7.38.

Limitations on DHCP client behavior One subgroup of the requirements describes the limitations on the DHCP client behavior. It cannot be easily split into the LSC chart sets because of variety of limitations. Here a number of LSC constructs comes into play, as opposed to single-simregion and optional prechart template used to capture the allowed behavior.

Allowed behavior of the DHCP server This subgroup specifies what transitions and behavior is allowed by the DHCP server according to the DHCP specification. The DHCP server is basically stateless, therefore its requirements result in simple form LSC charts. The subgroup involves set of requirements that follow the pattern of request-response. The charts are typically with the request message from the client in the prechart, and response (responses) in the mainchart, optionally in the If-Then-Else constructs to allow explicit non-determinism.

Notation of the requirements is $\_msg1\_msg2$, what means that upon receiving message $msg1$ from the client, the server responds with any of several outcomes (sending message or skipping the response), where message $msg2$ is among allowed responses.

The reaction of the DHCP server to the inputs from the client is as follows:

_arp_rrp , "upon receiving the m_arp message from $Client$, the server either issues the m_rarp message or remains silent".

_dis_ofr , "upon receiving the $m\_discover$ message from $Client$, the server either issues the m_offer message or remains silent".

_inf_ack , "upon receiving the m_inform message from $Client$, the server either issues the m_ack message, m_nak message or remains silent".

_rbi_ack , "upon receiving the $m\_request$ message from $Client$ in $S\_REBINDING$ state, the server either issues the m_ack message, m_nak message or remains silent".

_rne_ack , "upon receiving the $m\_request$ message from $Client$ in $S\_RENEWING$ state, the server either issues the m_ack message, m_nak message or remains silent".

_rse_ack , "upon receiving the $m\_request$ message from $Client$ in $S\_SELECTING$ state, the server either issues the m_ack message, m_nak message or remains silent".

_rrb_ack , "upon receiving the $m\_request$ message from $Client$ in $S\_INITREBOOT$ or $S\_REBOOTING$ state, the server either issues the m_ack message, m_nak message or remains silent".

The template for the DHCP server requirements can be the LSC chart in figure 7.42 labeled $\_arp\_rrp$. It has a message in the prechart, that corresponds to the message arriving from the client. Possible outcomes (reply messages) are presented in the mainchart afterwards, in the If-Then-Else constructs. If there are more than two different outcomes (including the silent discard of the message), the If-Then-Else structures can be nested.

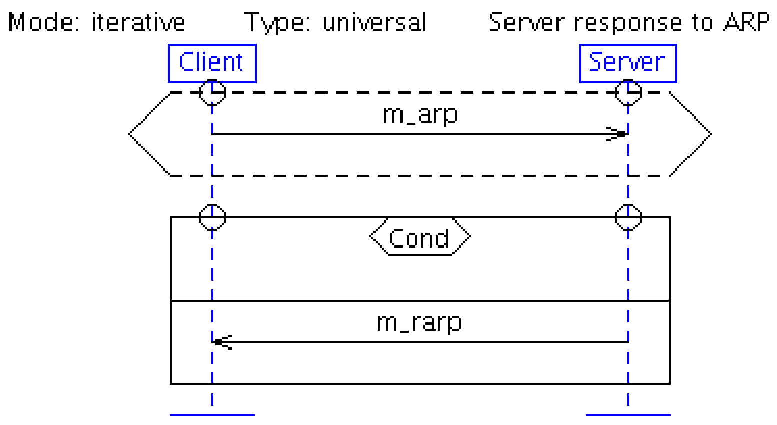

Figure 7.42: Requirement "upon receiving the *m_arp* message from *Client*, the server either issues the *m_rarp* message or remains silent" captured in the LSC chart.

7.4.4 DHCP client test cases - implementation

There has been an adapter developed to convert the abstract inputs and outputs in UppAal TRON into the real DHCP and control messages sent to or received from IUT. The adapter for the DHCP client has most of its functions taken from the *ControllerQ* class. The implementation scheme is displayed in figure 7.43.

One part of the adapter is *Abstractor*, which decides the message type based on the incoming packet and informs TRON about occurrence of such event. Another part is *Resolver*, which, upon reception of abstract inputs from UppAal TRON, constructs and issues the real DHCP packet or user command to DHCPC.

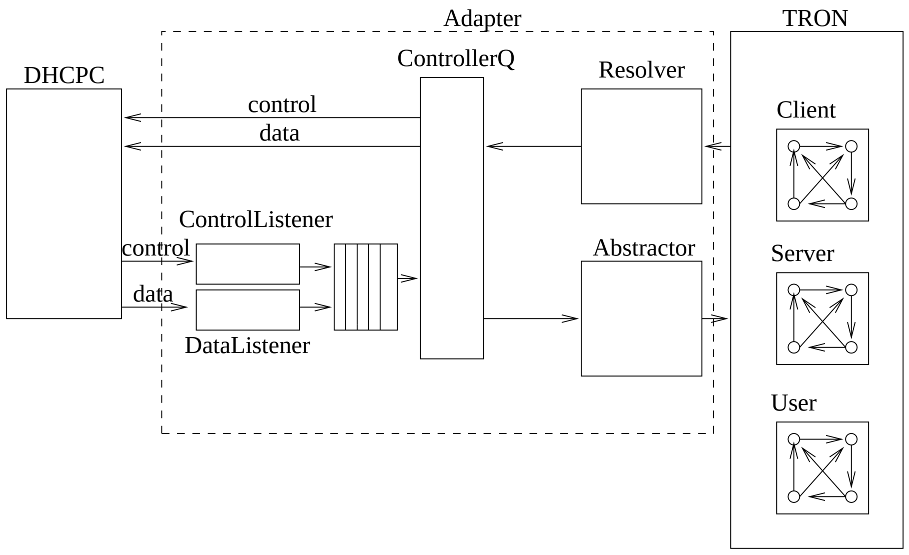

Figure 7.43: Testing framework after introduction of test with UppAal TRON

There are several structural differences between the test case-based (figure 7.35) and TRON-based version (figure 7.43) of DHCPC test framework:

- In the test case, a sequence of commands is executed. The UppAal TRON instead performs simulation of the *Client*, *Server*, *User* models to generate the commands on-the-fly.
- Interleaving between inputs and outputs of DHCPC is strictly defined by the sequence of commands in the test case. Their interleaving in UppAal TRON is non-deterministic in cases when both TRON and DHCPC issues an event or the packet. Priority of the channels or additional protocol must be implemented in order to ensure first-come, first-serve rule for the data generated at the DHCPC and TRON.
- More threads are used in case of TRON-based approach because one thread executing a sequence of commands is now replaced with at least two threads for bi-directional communication between TRON and adapter.

Shifting to TRON gives more flexibility during the system test:

- Adapter is written once, and does not depend on the changes in the model.
- Test cases are generated and executed from the model, instead of writing them manually. It does not mean that static test cases should be dropped, but instead the implementation can be tested against more patterns of behavior. Ultimately, the test event sequences that fail, should be reduced to the static test cases for the fast validation test, when the implementation changes.
- One or more of the actors in the system can have its behavior changed, without effect on the adapter or the implementation. In approach of test cases, that would result in some of the test cases becoming obsolete, and the coverage degree being decreased.
- Test suites can be generated that cover particular aspect of the behavior at the implementation or its environment in least number of moves, least time other useful criterion.
- Since behavior part is modeled and separated from data part, the stress-test and invalid combinations can easily be implemented by extending the environment model with an automaton which keeps issuing malformed packets. Collection of bad data can be generated using pairwise, or N-wise coverage of writing various (correct and incorrect) values into the fields in the test packet.

Several challenges are waiting after introducing the TRON-based test:

- First of all, abstract inputs and outputs of the model should be defined and described in the document. Changes in the inputs and outputs will result in changes of the existing test cases, what is time consuming and error-prone.
- Existing and generated static test cases should be kept in abstract form. This prevents them from changes in adapter details.
- Techniques of test case reduction to shortest failing sequence of events are known, and need to be implemented for TRON-generated event traces that lead to error.
- Generating the stand-alone test case from the TRON event trace must be implemented. This is complicated in some cases because of non-determinism.
- Convenient modeling tools for capturing requirements should be developed based on the existing prototypes.

Technical shortcomings of the current implementation:

- Response messages are generated based on the last message received from DHCP client. Queue of certain length should be maintained to store outputs of DHCP client, and response messages would be generated based on the entries from that queue instead.
- Some inputs to the IUT might become inadequate because its state has changed. Example: RARP message sent long time after ARP, where it would have no effect and would be disregarded.The challenge is to reflect the adequacy of inputs in the model. The forbidden and ignored elements are the way to introduce that in the model - and the additional charts with time-outs and combinations of events that render the specific event outdated.

7.4.4.1 Results of the DHCPC testing

Untimed on-line test has been performed on the DHCP client implementation developed at the Ericsson A/S (Telebit). No errors have been found that are related to the DHCP client state machine.

However, the on-line test has covered only the part of the DHCP client behavior. It has been the combination of the message m_nak and the state of the DHCP client being $S\_REBOOTING$, when the DHCP client must restart the configuration process. Since the DHCP server has been offering one and the same IP address, the same address is not accepted by the client because of requirement in [Dro97], section 3.3 : "If the client receives a DHCPNAK message, it cannot reuse its remembered network address". In the model, the DHCP server does not distinct between the IP addresses it is offering.

The presented situation shows that a serious attention must be paid to the data part of the protocol in order to construct the model capable of redundant test. It is not enough to consider only the behavior part of the communication protocols.

7.4.5 DHCP client test - conclusions

Tests of the DHCP client have given insight of how the tests should be performed and the validity of the test framework assured.

The test framework involves four parts: the set of LSC requirements that constitute the LSC specification, the UppAal TRON tool with the translated LSC model, the implementation (IUT) and adapters to enable communication between TRON and IUT.

The IUT has passed the conformance test with respect to the model defined by means of requirements from the corresponding RFC document.

The model that comprises the DHCP client state machine has not been detail enough to perform the tests to such an extent as the manual test cases have. The reason for that is the subtleties of the data part requirements for the protocol.

Suggestions from the test

A number of situations have been experienced during the test that are worth mentioning:

- Always review the event log files. The model is abstraction of the real behavior, and it typically misses the functionality related to the data part. Even if the automatic test runs for a long time without failing, it might be that the IUT is cycling in a small part of the state machine, like the DHCP client upon having its requested IP address rejected once, and not using it any more in spite of same addres offered repeatedly.

- The updated and translated model is supposed to have certain errors corrected. Make sure that the same translated model is used in the test and in the validation by using the UppAal model checking tool.

- Analyze the log files produced by the IUT under certain conditions, for example when the log file size stops increasing, but the test still runs. During lengthy tests with numerous inputs, memory leaks can cause the IUT deadlock, and it can remain undetected unless the response is expected within bounded time.

- Analyze the log files produced by the IUT, to assure that all relevant inputs/outputs have been abstracted or converted to real events or packets. The behavior of the model and abstract events trace is dependent on how accurately the relevant events are recognized and converted.

- Reuse the requirements. Flexible management of requirements, like in LSC Play Engine, allows composing as many test configurations as necessary, each focusing on several issues. Operations on the sets of requirements are much easier than modifying the model constructed directly in UppAal. This advantage must be exploited, making it easier to have the requirements sorted according to their functionality and reusing them, instead of creating new requirements every time. Typically the model-constituting requirements are split into 2 sets: the ones allowing the model behave in certain ways, and the ones limiting that behavior. The first group can be reused in most or all models, while the second group will typically include unique requirements (scenarios) that forbid or expose certain parts of the model.

- Check the translated models for reachability during their validation. This especially concerns the models of the stateful protocols.The diagnostic trace types such as shortest, fastest, random can already expose some unexpected behavior or underspecification of the model.

Chapter 8

Conclusions and future work

8.1 Contribution

Several contributions of the thesis should be emphasized:

- The set of LSC constructs has been determined that is sufficient to capture the informal textual requirements. The choice has been validated through examples and case studies (section 3.2 and chapter 7).

- Several flavors of LSC semantics have been analyzed and one of them chosen as a standard. The examples and case studies have been provided based on that flavor of semantics.

- The rules of translation from LSC to UppAal timed automata have been defined and the automatic translation has been implemented.

- It has been shown that testing with UppAal TRON is feasible using translated specifications. An industrial case-study and a small academic example has been taken to demonstrate the ability to test the implementations against their specifications originally defined in LSC.

- It has been demonstrated that the translated LSC charts can also serve as the properties to check the system against. Untimed specification of Automatic Telling Machine (ATM, section 7.1) has been taken as an example.

- The translated LSC specifications have been successfully loaded and simulated in UppAal family tools. The tools have served for the debugging purposes of the specifications and also provided model checking capabilities for smaller specifications.

The tasks defined as the aim of the thesis in the abstract have consequently been achieved.

8.2 Future work

8.2.1 Manipulation of the requirements sets

Currently, the LSC specification is translated to the UppAal model in a batch work, taken the set of LSC as specification and set of LSC as properties. Each configuration of the sets is performed in command mode and by editing the text files. The LSC charts are edited separately.

Managing the sets of LSC charts should be automated, and possibility preserved to edit the LSC chart in the same tool where the translation is performed. The existing tool chain consists of several different tools, that make the requirements debug or requirements analysis processes inconvenient. No immediate feedback is currently possible when a requirement or a set of requirements in LSC charts are altered, and typically several tools need to be launched in order to get such a feedback. Moreover, locally made changes in the final UppAal model are lost every time the new translation is performed.

To ease the construction of the specification of the systems by means of LSC, the multi-chart editor would be handy with functions such as on-the-fly translation to a UppAal model. An even more convenient alternative would be implementation of the simulator operating over LSC charts and the TRON counterpart operating over the LSC charts as well. In such a case, the developer would be free to interact with the LSC specification at a high level and easily debug it.

From the point of the requirements traceability, a starting step has been done, which is namely giving the LSC chart the description. However, this is the chart-centric approach, and it is not convenient to manipulate over requirements. Instead, the requirement should be the central entity, as the LSC charts are certain attempts to formalize certain aspects of the requirement. Changes of the requirement should be stored in the versioning system, and the scenarios related to each version and part of the requirement. This would make operation over the requirement sets more intuitive and easy. Besides, the knowledge could be collected about, how and why the chosen LSC formalism or its certain constructs contribute or obstruct the capturing of some requirement. Several types of the requirement traceability have been defined in [SS97], and these could be taken into account when advanced support over the requirements manipulation is implemented.

8.2.2 Enhanced translation from LSC to UPPAAL

In its current version, the translation heavily relies on global variables, flags and mechanisms of emulating the LSC semantics over UppAal.

There is space for improvements so that several properties or mechanisms are encoded more efficiently:

- Chart mode semantics is currently maintained by having several copies of the same template, and using global variables and functions to activate, deactivate these copies and interact among them. Let us call the existing templates the old ones, where one instantiation of a template corresponds to a copy of LSC chart, and let us assume an automaton (referred to as the new one), which has these several translated copies of the LSC chart aggregated inside. When the maximal number of active chart copies is bound all the time, several improvements can be achieved by using the new automata:

  – Several copies of the templates can be merged into one, which corresponds to a set of LSC charts; amount of timed automata in the NTA thus decreases significantly.
  – The deactivation, activation functions and variables specifying when to start or terminate a particular chart copy would become redundant.
  – Each stable location of new generated automaton would uniquely correspond to a set of cuts occupied by all the active old copies of a LSC chart.

- Event generating automaton and the token mechanism would become extraneous upon employing the automata translated from aggregate copies. Instead of the token mechanism to determine the message event owner among the LSC charts, the conventional synchronizations *chan*! and *chan*? could be used in aggregate automata. In such a way the message event owner would be trivially identified and the corresponding step easily selectable in the UppAal tool during simulation of the translated LSC specification.

  It would become easier to simulate such translated specifications in UppAal and integrate them into the models with natively developed templates with far less restrictions than currently. The main difference would be relaxed requirement to put all the automata under control of the token in selecting the active chart, and allowing synchronizations in automata with both "!" and "?" signs, as opposed to only "?" sign used currently.

- Instance abstraction and message event abstraction is available by means of using shared variables. The ways of encapsulating these abstractions should be considered, if the abstractions shall be used extensively. Encapsulation can be implemented in the UppAal side by means of explicit declaration of shared variables. From the LSC specification side, the graphic front-end could be alternatively adapted to specify these abstractions.

8.2.3 Model checking and testing in LSC

Having one of several flavors of LSC semantics fixed, the UppAal tool could be adapted to the operation over LSC instead of timed automata. Currently, the LSC charts are translated into the timed automata. On the one hand, this allows high flexibility to choose the translation and LSC semantics flavor according to whom the translation is performed. On the other hand, the chosen flavor of LSC semantics is already fixed and successfully used by means of translated LSC specifications. When sufficient amount of translated specifications with same LSC semantics exists, it is more costly to translate the specification into other formalism and use it elsewhere, than have tools that support the specifications in their original formalism.

Operating totally with LSC charts would have several advantages, like getting rid of the remaining TA mechanisms to emulate LSC semantics, and have the GUI editor dedicated to LSCs.

Translation from different LSC flavors to UppAal would be the missing link between these two formalisms.

In similar fashion as the UppAal TRON tool is derived from UppAal, the LSC TRON tool could be developed which maintains the LSC semantics and performs the conformance test of implementations according to TIOCO conformance relation against their specifications in LSC.

In Proceedings

[AFH91] Rajeev Alur, Tomas Feder, and Thomas A. Henzinger. The benefits of relaxing punctuality. In *PODC '91: Proceedings of the tenth annual ACM symposium on Principles of distributed computing*, pages 139–152, New York, NY, USA, 1991. ACM Press.

[BB04a] L. Brandán Briones and E. Brinksma. A test generation framework for quiescent real-time systems. In Brian Nielsen Jens Grabowski, editor, *Formal Approaches to Software Testing, FATES*, Linz, Austria, Sep 2004. Springer-Verlag GmbH.

[BB04b] Laura Brandán Briones and Ed Brinksma. A test generation framework for *quiescent* real-time systems. In Jens Grabowski and Brian Nielsen, editors, *FATES*, volume 3395 of *Lecture Notes in Computer Science*, pages 64–78. Springer, 2004.

[BBD$^+$04] Matthias Brill, Ralf Buschermöhle, Werner Damm, Jochen Klose, Bernd Westphal, and Hartmut Wittke. Formal verification of lscs in the development process. In Ehrig et al. [EDD$^+$04], pages 494–516.

[BDK$^+$04] Matthias Brill, Werner Damm, Jochen Klose, Bernd Westphal, and Hartmut Wittke. Live sequence charts: An introduction to lines, arrows, and strange boxes in the context of formal verification. In Ehrig et al. [EDD$^+$04], pages 374–399.

[BFG$^+$00] Marius Bozga, Jean-Claude Fernandez, Lucian Ghirvu, Susanne Graf, Jean-Pierre Krimm, and Laurent Mounier. IF: A validation environment for timed asynchronous systems. In *Computer Aided Verification*, pages 543–547, 2000.

[BG01] Annette Bunker and Ganesh Gopalakrishnan. Using live sequence charts for hardware protocol specification and compliance verification. In *HLDVT '01: Proceedings of the Sixth IEEE International High-Level Design Validation and Test Workshop (HLDVT'01)*, page 95, Washington, DC, USA, 2001. IEEE Computer Society.

[BGMO04] Marius Bozga, Susanne Graf, Laurent Mounier, and Iulian Ober. If validation environment tutorial. In Graf and Mounier [GM04], pages 306–307.

[BKSS06] Benedikt Bollig, Carsten Kern, Markus Schlütter, and Volker Stolz. MSCan - a tool for analyzing MSC specifications. In Holger Hermanns and Jens Palsberg, editors, *TACAS*, volume 3920 of *Lecture Notes in Computer Science*, pages 455–458. Springer, 2006.

[BT01] Ed Brinksma and Jan Tretmans. Testing transition systems: An annotated bibliography. In *MOVEP '00: Proceedings of the 4th Summer School on Modeling and Verification of Parallel Processes*, pages 187–195, London, UK, 2001. Springer-Verlag.

[CF05] Hana Chockler and Kathi Fisler. Temporal modalities for concisely capturing timing diagrams. In Dominique Borrione and Wolfgang J. Paul, editors, *CHARME*, volume 3725 of *Lecture Notes in Computer Science*, pages 176–190. Springer, 2005.

[EDD+04] Hartmut Ehrig, Werner Damm, Jörg Desel, Martin Große-Rhode, Wolfgang Reif, Eckehard Schnieder, and Engelbert Westkämper, editors. *Integration of Software Specification Techniques for Applications in Engineering, Priority Program SoftSpez of the German Research Foundation (DFG), Final Report*, volume 3147 of *Lecture Notes in Computer Science*. Springer, 2004.

[GM04] Susanne Graf and Laurent Mounier, editors. *Model Checking Software, 11th International SPIN Workshop, Barcelona, Spain, April 1-3, 2004, Proceedings*, volume 2989 of *Lecture Notes in Computer Science*. Springer, 2004.

[HKP04] David Harel, Hillel Kugler, and Amir Pnueli. Smart play-out extended: Time and forbidden elements. In *QSIC'04: Proceedings of the Quality Software, Fourth International Conference on (QSIC '04)*, pages 2–10, Washington, DC, USA, 2004. IEEE Computer Society.

[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 *ICSE '88: Proceedings of the 10th international conference on Software engineering*, pages 396–406, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[HP07] Anders Hessel and Paul Pettersson. A global algorithm for model-based test suite generation. In *Proceedings of Third Workshop on Model-Based Testing*. Electronic Notes in Theoretical Computer Science 16697, March 2007.

[HRD06] Jameleddine Hassine, Juergen Rilling, and Rachida Dssouli. Timed use case maps. In Reinhard Gotzhein and Rick Reed, editors, *SAM*, volume 4320 of *Lecture Notes in Computer Science*, pages 99–114. Springer, 2006.

[KT04] Moez Krichen and Stavros Tripakis. Black-box conformance testing for real-time systems. In Graf and Mounier [GM04], pages 109–126.

[KTWW06] Jochen Klose, Tobe Toben, Bernd Westphal, and Hartmut Wittke. Check it out: On the efficient formal verification of live sequence charts. In Thomas Ball and Robert B. Jones, editors, *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 219–233. Springer, 2006.

[KW01] Jochen Klose and Hartmut Wittke. An automata based interpretation of live sequence charts. In *TACAS 2001: Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 512–527, London, UK, 2001. Springer-Verlag.

[LMN04] K.G. Larsen, M. Mikucionis, and B. Nielsen. Online testing of real-time systems using UPPAAL. In *Formal Approaches to Testing of Software*, Linz, Austria, September 21 2004. Lecture Notes in Computer Science.

[LMNS05] Kim G. Larsen, Marius Mikucionis, Brian Nielsen, and Arne Skou. Testing real-time embedded software using uppaal-tron: an industrial case study. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 299–306, New York, NY, USA, 2005. ACM Press New York, NY, USA.

[MH06] Shahar Maoz and David Harel. From multi-modal scenarios to code: compiling LSCs into aspectj. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 219–230, New York, NY, USA, 2006. ACM Press.

[MHK02] Rami Marelly, David Harel, and Hillel Kugler. Multiple instances and symbolic variables in executable sequence charts. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 83–100, New York, NY, USA, 2002. ACM Press.

[MLN04] Marius Mikucionis, Kim G. Larsen, and Brian Nielsen. T-UPPAAL: Online model-based testing of real-time systems: tool demo. In *the 19th IEEE International Conference on Automated Software Engineering*, pages 396–397, Linz, Austria, September 24 2004.

[MNL03] M. Mikucionis, B. Nielsen, and K.G. Larsen. Real-time system testing on-the-fly. In *the 15th Nordic Workshop on Programming Theory*, number 34 in B, pages 36–38, Turku, Finland, October 29–31 2003. Åbo Akademi, Department of Computer Science, Finland. Abstracts.

[MNP06] Oded Maler, Dejan Nickovic, and Amir Pnueli. From mitl to timed automata. In Eugene Asarin and Patricia Bouyer, editors, *FORMATS*, volume 4202 of *Lecture Notes in Computer Science*, pages 274–289. Springer, 2006.

[NS01] Brian Nielsen and Arne Skou. Automated test generation from timed automata. In *TACAS 2001: Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 343–357, London, UK, 2001. Springer-Verlag.

[TB03] G. J. Tretmans and H. Brinksma. Torx: Automated model-based testing. In A. Hartman and K. Dussa-Ziegler, editors, *First European Conference on Model-Driven Software Engineering, Nuremberg, Germany*, pages 31–43, December 2003.

[Tre99] J. Tretmans. Testing concurrent systems: A formal approach. In J.C.M Baeten and S. Mauw, editors, *CONCUR'99 – 10^{th} Int. Conference on Concurrency Theory*, volume 1664 of *Lecture Notes in Computer Science*, pages 46–65, 1999.

[vL03] Axel van Lamsweerde. Goal-oriented requirements engineering: From system objectives to uml models to precise software specifications. In *ICSE*, pages 744–745. IEEE Computer Society, 2003.

[vO06] Michiel van Osch. Hybrid input-output conformance and test generation. In *FATES/RV*, pages 70–84, 2006.

[WRHM06] Michael W. Whalen, Ajitha Rajan, Mats P.E. Heimdahl, and Steven P. Miller. Coverage metrics for requirements-based testing. In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, pages 25–36, New York, NY, USA, 2006. ACM Press.

[WRYC04] Tao Wang, Abhik Roychoudhury, Roland H. C. Yap, and S. C. Choudhary. Symbolic execution of behavioral requirements. In *PADL*, pages 178–192, 2004.

Articles

[ACD93] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, 1993.

[AE03] D. Amyot and A. Eberlein. An evaluation of scenario notations and construction approaches for telecommunication systems development. *Telecommunication Systems*, 24:61–94(34), September 2003.

[BGM04] Annette Bunker, Ganesh Gopalakrishnan, and Sally A. McKee. Formal Hardware Specification Languages for Protocol Compliance Verification. *ACM Transactions on Design Automation of Electronic Systems*, 9(1):1–32, January 2004.

[BKO05] Victor Braberman, Nicolas Kicillof, and Alfredo Olivero. A scenario-matching approach to the description and model checking of real-time properties. *IEEE Trans. Softw. Eng.*, 31(12):1028–1041, 2005.

[CCI92] ITU-T CCITT. Recommendation z.100: Specification and description language (sdl). 1992. General Secretariat, Geneve, Switzerland.

[CHR91] Zhou Chaochen, C. A. R. Hoare, and Anders P. Ravn. A calculus of durations. *Inf. Process. Lett.*, 40(5):269–276, 1991.

[DH01] Werner Damm and David Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, July 2001.

[DSD07] Doron Drusinsky, Man-Tak Shing, and Kadir Alpaslan Demir. Creating and validating embedded assertion statecharts. *IEEE Distributed Systems Online*, 8(5), 2007.

[DW05] Werner Damm and Bernd Westphal. Live and let die: LSC-based verification of UML-models. *Science of of Computer Programming*, 55(1–3):117–159, March 2005.

[LPY97] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[PKA94] C. Potts, K.Takahashi, and A.I.Antón. Inquiry-based requirements analysis. *IEEE Software*, 11:21–32, March 1994.

Books

[HM03] David Harel and Rami Marelly. *Come, let's Play. Scenario-Based Programming using LSCs and the Play-Engine*. Springer-Verlag, 2003.

[Hol92] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1992.

[IT99] ITU-T. *ITU-T Recommendation Z.120: Message Sequence Chart (MSC)*. ITU-T, 1999.

[oEE90] Institute of Electrical and Electronics Engineers. *IEEE Std 610.12-1990:IEEE Standard Glossary of Software Engineering Terminology*. 1990.

[SS97] Ian Sommerville and Pete Sawyer. *Requirements Engineering: A Good Practice Guide*. John Wiley & Sons, Inc., New York, NY, USA, 1997.

InCollections

[BDK$^+$04] Matthias Brill, Werner Damm, Jochen Klose, Bernd Westphal, and Hartmut Wittke. Live Sequence Charts. Number 3147 in Lecture Notes in Computer Science, pages 374–399. Springer-Verlag, 2004.

Manuals

[Eri05] Ole Eriksen. *Formal Based Requirement Engineering of Embedded Software - an Example*. University of Aalborg, Esbjerg, 2005.

Master Thesis

[MS03] Marius Mikucionis and Egle Sasnauskaite. On-the-fly testing using UppAal. Master's thesis, Department of Computer Science, Aalborg University, Denmark, June 2003.

[RAJtJ04] Jens Gorm Rye-Andersen, Mads W. Jensen, René Gøttler, and Michael Jacobsen. Property extraction engine for lscs. Master's thesis, Department of Computer Science, Aalborg University, Denmark, June 2004.

PhD Thesis

[Bon05] Yves Bontemps. *Relating Inter-Agent and Intra-Agent Specifications (The Case of Live Sequence Charts)*. PhD thesis, Facultés Universitaires Notre-Dame de la Paix, Institut d'Informatique (University of Namur, Computer Science Dept), April 2005.

Miscellaneous

[BFdV+99] A. Belinfante, J. Feenstra, R. de Vries, J. Tretmans, N. Goga, L. Feijs, S. Mauw, and L. Heerink. Formal test automation: A simple experiment. In G. Csopaki, S. Dibuz, and K. Tarnay, editors, 12th Int. Workshop on Testing of Communicating Systems, 1999.

[Bra89] R. T. Braden. RFC 1122: Requirements for Internet hosts — communication layers, October 1989.

[Dro97] R. Droms. RFC 2131: Dynamic host configuration protocol, March 1997.

[Gro04] W3C Working Group. Web services architecture requirements, February 2004. W3C Working Group Note 11 February 2004.

[OMG02] OMG. Response to the OMG RFP for Schedulability, Performance and Time, v. 2.0. OMG document ad/2002-03-04, March 2002.

[PMBW00] Frederick Proctor, John Michaloski, Sushil Birla, and George Weinert. Analysis of behavioral requirements for component-based machine controllers. Proceedings of SPIE International Symposium on Intelligent Systems and Advanced Manufacturing, November 2000.

[Pos81] J. Postel. RFC 791: Internet Protocol, September 1981.

Technical Reports

[BBNS03] Mark Blackburn, Robert Busser, Aaron Nauman, and Bryan D. Stensvad. Defect identification with model-based test automation. Technical report, SAE 2003 World Congress & Exhibition, March 2003, Detroit, MI, USA, Session: Testing and Instrumentation, March 2003.

[MLN03] Marius Mikucionis, Kim G. Larsen, and Brian Nielsen. Online on-the-fly testing of real-time systems. Technical Report RS-03-49, BRICS, iesd, December 2003. 14 pp.

[ttg04] An expressive and implementable formal framework for testing real-time systems. Technical Report TR-2004-13, Verimag Technical Report, June 2004.