Zusammenfassung

Graphische Spezifikation von Kommunikationsabläufen mittels Szenarien erfreut sich großer Beliebtheit, vor allem wegen ihrer Einfachheit und Intuitivität. Die beiden in diesem Bereich existierenden Standards, Message Sequence Charts (MSC) und Sequence Diagrams (SD), schöpfen aufgrund von mangelnder Ausdruckskraft und fehlender semantischer Fundierung ihre Möglichkeiten im Hinblick auf den erzielbaren Nutzen nicht voll aus. Zudem sind sowohl MSCs als auch SDs in erster Linie auf den Einsatz in einem informellen, exemplarischen Kontext ausgerichtet. Ziel dieser Arbeit ist es, die Grundlage für einen weitergehenden Einsatz von Sequence Charts, insbesondere in der formalen Verifikation, zu schaffen. Ein Ansatz in diese Richtung sind die von Damm und Harel vorgestellten Live Sequence Charts (LSC).

Kernpunkt von LSCs ist die Unterscheidung zwischen möglichem und geforderten Verhalten. Auf oberster Ebene erlaubt dies die Spezifikation von einzuhaltenden Abläufen, zusätzlich zu exemplarischen Abläufen in MSCs und SDs. Weiterhin ist es in LSCs möglich Lebendigkeitseigenschaften zu formulieren, wie etwa das Ankommen einer Nachricht. Andere wichtige Neuerungen sind die Aufwertung von Bedingungen zu booleschen Ausdrücken, sowie die Möglichkeit, den Aktivierungszeitpunkt einer Chart mittels einer Bedingung zu charakterisieren.

Der von Damm und Harel vorgeschlagene Sprachumfang wird im Rahmen dieser Arbeit um weitere essentielle Konstrukte ergänzt, wie Zeitbedingungen, Gleichzeitigkeit von Ereignissen, sowie die Möglichkeit, die Aktivierung einer Chart durch Nachrichtensequenzen auszulösen. Des weiteren wird die Spezifikation von Annahmen über das Umgebungsverhalten innerhalb der LSC, die das gewünschte Systemverhalten ausdrückt, ermöglicht.

Neben der Festlegung des LSC-Sprachumfangs ist die Definition der formalen Semantik von LSCs zentraler Punkt dieser Dissertation. Die Semantik wird konstruktiv definiert durch Abbildung einer LSC in einen endliche Automaten (zeitbehaftete Büchi-Automaten), unter Berücksichtigung der in der LSC ausgedrückten partiellen Ordnung zwischen den einzelnen LSC Elementen.

Das Sprachdesign wird abgerundet durch eine Einordnung von LSCs in einen modellbasierten Entwicklungsprozeß, wobei das Ziel ist, Wissen in Form von einmal spezifizierten LSCs soweit möglich in späteren Phasen des Entwurfs wiederzuverwenden. Abschließend erfolgt eine Untersuchung der praktischen Anwendbarkeit der entwickelten Spezifikationssprache am

Beispiel der formalen Verifikation von STATEMATE-Modellen. Hierbei werden die zu überprüfenden Anforderungen als LSCs spezifiziert.

Abstract

Graphical description of message exchange by means of scenarios is a popular specification technique, especially due to the intuiveness provided. The prevalent standards in this area, Message Sequence Charts (MSC) und Sequence Diagrams (SD), are lacking both expressiveness and formal foundation. Additionally, they are used almost exclusively in an exemplary fashion describing typical system interactions. The goal of this thesis is the creation of a sound basis for the application of sequence charts to other, more advanced use cases like formal verification. An approach along this line of thought has been presented by Damm and Harel in the form of Live Sequence Charts (LSC).

The basic idea of LSCs is the distinction between possible and mandatory elements allowing, at the level of an entire chart, the specification of required behavior in addition to the sample interactions of MSCs and SDs. At a more fine-grained level this feature creates the possibility to express liveness properties like the mandatory receipt of a message. Other novel features are the upgrade of conditions to boolean expressions and the characterization — via an activation condition — of when the chart is to be activated.

This thesis extends the LSC language as proposed by Damm and Harel by several missing, but essential features like time constraints, simultaneity of events as well as activation of an LSC by a sequence of interactions. Another important new feature is the possibility to specify assumptions about environment behavior directly within the LSC describing the desired system behavior.

The definition of the formal semantics of LSCs is the second major part of this thesis. The semantics is defined in a constructive fashion by transforming an LSC into a timed Büchi automaton capturing the partial order prescribed by the LSC elements.

The language design is completed by a presenting a methodology which embeds LSCs into a model-based development process. The goal is to reuse knowledge about the system, recorded in the form of LSCs, in later phases of development. The rear of this thesis is brought up by an evaluation of the applicability of the developed specification language. The formal verification of STATEMATE models is chosen as a sample field of application, where the properties to be verified are specified by LSCs.

Acknowledgments

A major piece of work, like this thesis, is seldom created in isolation. At this point I therefore would like to gratefully acknowledge those people, which have helped me to achieve this success.

First of all I would like to thank Werner Damm, who provided me with the opportunity to write this thesis. I have greatly benefitted from his encouragement and the creative working environment he created. His knowledge and support have been an invaluable resource during the last five years. I furthermore thank Ernst-Rüdiger Olderog, who was kind enough to act as the second reviewer for my thesis. Additional thanks go to David Harel for the vivid discussions on issues of design and semantics of the LSC language.

Another key ingredient for the existence of this thesis is the working group in which I have been working in Oldenburg. I have had many interesting and helpful discussions with the people in this working group and have profited from their expertise and experience. I especially thank Hartmut Wittke, with whom I have shared many thoughts and ideas on the subjects of this thesis. He also contributed to a substantial degree in the integration of the LSC tools into the existing STATEMATE verification environment, thus helping to make the verification of LSCs become a reality. He was the most assidious proof readers, as well. I furthermore thank Tom Bienmüller and Alexander Metzner. They not only have contributed to this thesis by discussing semantical details, proof reading and general comments, but also shared an office with me. The congenial atmosphere of this "center of knowledge" was extremely motivating and enjoyable. Further thanks go to Martin Fränzle, who gave valuable comments on the automata-theoretic part, and Hans Jürgen Holberg, who provided me with feedback regarding the methodology and application of LSCs. I also thank Olaf Bär, Uwe Higgen and Rainer Koopmann, who carried out the initial implementation of the tools supporting the formal verifiction of LSCs. Moreover, I thank Andreas Thums for his collaboration on the STATEMATE case study.

Last, but certainly not least, I thank my parents for their support, not only in this endeavor. They have always encouraged me in what I was doing, for which I am deeply grateful.

Contents

Chapter 1

Introduction

The amount of modern products containing hard and software components is becoming larger and larger. The type of products containing computer technology ranges from coffee and washing machines to cars and trains. These computer systems are embedded into a physical environment, which provides input data to these *embedded controllers* and which is in turn affected by the embedded controller. An airbag controller for instance reads sensor data, which indicate if a crash has occurred, and activates the firing capsule in this case. A train control system, as another example, uses various sensors to determine the position and velocity of a train, computes the maximum speed for the current position and checks, if the train is going too fast.

Not only does the number of products containing embedded controllers increase, but also the number of embedded controllers within a single product: a modern high-end car e.g. may contain up to seventy embedded controllers. This proliferation of embedded controllers is accompanied by a rising demand for more functionality entailing more complex embedded controllers. Following the general trend there is also a rising number of embedded controllers performing *safety-critical* tasks. The incorrect computation or supervision of a train's speed e.g. may lead to a derailed train involving severe damage to people and material.

The development of embedded controllers today typically starts with a requirement specification document written in natural language. Such informal specifications have the inherent danger of being ambiguous, inconsistent and incomplete — especially since such documents can consist of several hundred pages — and lead to errors and incompatibilities, which often are detected only later in the design process. The later an error is found, the more costly — both in terms of money and time — it is to remove, since

each step back in the development cycle means that the following steps have to be re-taken as well. In order to reduce the overall development costs it is thus imperative to uncover errors as early as possible.

Validation that the embedded controller conforms to the requirements specified initially is carried out by *testing*, i.e. by applying inputs to the controller and observing, if the correct outputs are produced. The testing process today is mostly carried out manually by test engineers, which rely on their experience and intuition for finding good test cases covering the relevant parts of the design. Clearly, the informal and often inconsistent process of embedded controller development today is not an optimal base for producing correct controllers in the most economic fashion, especially when considering safety-critical applications and the increasing complexity of the tasks, which are to be performed by the embedded controllers.

In order to guarantee the correctness and reliability of safety-critical electronic control units one prominent proposition is the adoption of an appropriate *development process*. Such a process structures the development into different phases, defines the activities to be performed in the phase, which documents are to be produced, etc. Examples for development processes are the V-model [ESt97], which has been defined for developing software for the German armed forces, and the CENELEC[1] norm EN 50128 [CEN01], which regulates the development of software for railway applications.

One approach, which addresses the abovementioned deficiencies of ambiguity and inconsistency, is known as *model-based development process*. The central idea is to construct an abstract model of the embedded controller capturing the requirements of the initial textual specification. Depending on the concrete needs and focus of the developed embedded controller different aspects — like e.g. functionality and decomposed structure — are reflected in the model. Modern CASE-tools[2] like e.g. STATEMATE [HLN+90] or various UML[3] tools [OMG01] provide support for such a development process and also offer graphical representations for the modeling of the embedded controller.

The abstract models allow the user to formalize the textual requirements and thus to more easily detect inconsistencies, ambiguities and incomplete specifications. Several tools, e.g. STATEMATE, additionally offer simulation

[1] Comité Européen de Normalisation Electrotechnique
[2] Computer Aided Software Engineering
[3] Unified Modeling Language

capabilities allowing to directly observe and influence the dynamical behavior of the model. This feature is referred to as *executable specification* and enables the user to gain a good understanding of how the embedded controller behaves dynamically. Moreover, it allows to change or add features and assess their impact without much effort or risk. If the embedded controller comprises several (logical or physical) components, it is possible to examine their combined dynamic behavior before a concrete implementation is available yielding a *virtual integration*. A model-based development hence allows to assess the functionality of the designed embedded controller at a very early stage in the design process removing ambiguities and inconsistencies, which would otherwise have potentially led to errors discovered only in later phases. The abstract representation constructed serves as a *reference model* (also called *golden device*) for the later stages of the development. Possible applications in this direction are for instance the automatic generation of code from the reference model and deriving test vectors for unit or integration testing.

Formal Verification

The model-based development process tackles the problem of ambiguous and inconsistent requirement specifications. The correctness of the designed embedded controller wrt. to its requirements is another vital property, which has to be guaranteed especially for safety-critical applications. In view of the increasing embedded controller complexity it is clear that the traditional testing approach is not sufficient to ensure correctness under *all* circumstances. The number of possible combinations of input stimuli and possible sequences thereof is too large to be tested in its entirety for industrial-sized embedded controllers. Testing thus considers only a finite set of test cases chosen by the test engineers.

In recent years *formal verification* has been developed in order to guarantee correctness under all circumstances. Formal verification entails a formal mathematical proof that a model satisfies the specified requirements. In combination with the model-based development process formal verification demonstrates the correctness of the model wrt. the specified requirements lending more weight to the reference model. In this way the requirement that "*a train never passes a not secured crossing*" can be proven for all possibilities of sequences of input stimuli.

Within the field of formal verification, there are three major approaches: *theorem proving*, *model checking* and *bounded model checking*. The basic idea of theorem proving is to support the user in constructing a proof calculus, which demonstrates the validity of the specified requirement. Theorem provers, like e.g. the popular PVS [OS97], require a large amount of user interaction and expert knowledge, since they provide computer-based support for the formal reasoning, which has to be carried out by the user.

Model checking is a fully automatic technique, which has been invented independently by two research groups (Clarke and Emerson [CE81] on the one hand and Quielle and Sifakis [QS82] on the other). Today's model checkers use a symbolic representation (see [BCM$^+$92, McM93]) of the model for greater efficiency, so that this technique is called *symbolic model checking*. In the remainder we use the term 'model checking' instead of symbolic model checking for simplicity's sake.

A model checker requires two inputs: the model to be examined and the requirement to be proven. The former input is given as a *Finite State Machine* (FSM), which formally describes the behavior of the model. The requirement is stated in *temporal logic* [Eme90, MP92], which adds temporal operators to the standard boolean operators '$\wedge$', '$\vee$' and '$\neg$'.

The model check algorithm determines, if the model satisfies the specified property under all circumstances, i.e. all possible sequences of input combinations are examined. The result is either 'true', if there is no way to violate the property, or 'false' otherwise. In the latter case the model checker produces a *counter example*, also called *error path* or *witness*, showing the sequence of input stimuli, which lead to the violation of the requirement. The counter example can be examined by the verifier and thereby gives valuable insight to the cause why the property does not hold. This feature is another reason why model checking has become more and more popular in recent years.
The details of the symbolic model checking are not relevant in this work and can be found in [BCM$^+$92], [McM93] or [CGP99].

The strategy of bounded model checking [BCCZ99] is to examine the model up to a depth k starting from the initial state and check, if the property is violated in this part of the model. If a violation is detected, a counter example is returned. If no violation is found, the result is inconclusive, since a violation might exist at a depth greater than k. The bound k is hence increased and the property is checked again. The actual checking procedure for each incomplete model FSM of depth k is formulated as a task for so-called *SAT-checkers*, which have been developed to efficiently solve propositional

satisfiability problems [DP60].

The crucial point is to know when to stop increasing the bound, i.e. when an increased depth does not add new states, which have not been examined before. This maximal bound is called *diameter* and depends on both the model and the property to be checked. The diameter of a given proof can be computed, but in practice this is efficiently possible only for small models, so that in practical applications k is provided by the user or given a default value. Thus, the part of bounded model checking, which is applied in practice, is an incomplete method.

Simplified Property Specification

Temporal logic formulas expressing real world requirements can quickly become fairly complex and hard to understand. Correct specification of non-trivial properties in terms of temporal logic requires considerable expert knowledge. Several approaches exist, which try to provide other, simpler means to specify properties. This is one step in order to enable non-expert users to employ formal verification techniques in practice, the other step being an easy and preferably automatic way to construct an FSM. This section presents the approaches dealing with the property side.

Two fundamental ideas are distinguished: one restricting the user to chose a *specification pattern* or *template* from a set of often recurring patterns and instantiate its parameters with the concrete model elements in order to specify the desired requirement. This method bases on the observation that often requirements are identical, except for the concrete variables used. The analysis of typical requirements thus leads to the identification of recurring patterns, which are offered as templates for requirement specification. For each such pattern the formal basis, e.g. temporal logic, is defined once and the user only chooses an appropriate pattern without having to wrestle with the low-level particulars. Examples for different pattern libraries are the ones compiled by Dwyer et al. [DAC98, DAC99], which allow patterns to be instantiated for a number of formalisms, e.g. LTL (linear time temporal logic) and CTL (computation tree logic) formula or Quantified Regular Expressions, and the one by Bitsch [Bit00, Bit01]. The STATEMATE Verification Environment (see below) also offers a library of specification patterns [OSC02a].

A variant of the pattern-based approach is the use of a restricted subset of natural language, which can be transformed into a temporal logic formula.

Holt and Klein [HK99, Hol99] e.g. use a subset of the English language in order to specify CTL formulas and Ruf et al. [FMR00] combine the structured natural language approach with specification patterns by allowing the user to construct sentences from preselected natural language fragments and instantiating parameters.

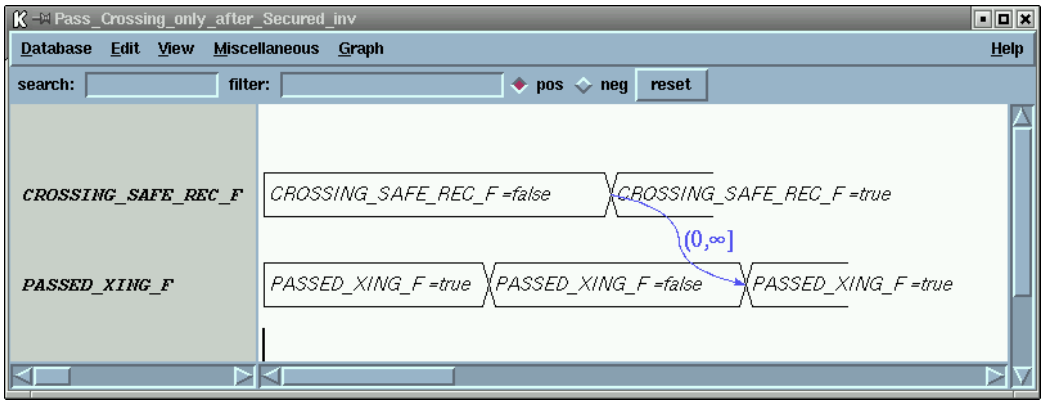


Figure 1.1: Symbolic Timing Diagram example

Using graphical notations to visualize temporal logic formula is the second method of simplifying the specification. Timing diagrams used in hardware design are the base for the first such graphical specification languages. Most interesting from our perspective are the *Symbolic Timing Diagrams* (STD) developed by Schlör, which first appeared in the early 1990's [SD93]. The definition of the semantics of LSCs presented in later chapters of this thesis has been influenced to a large degree by the semantics given for STDs in [Sch00]. STDs extend standard timing diagrams by allowing to specify qualitative time constraints between value changes (called *events*), of signals (called *waveforms*). Figure 1.1 shows an example STD with two waveforms, three events and one constraint. A later extension of STDs adds quantitative time constraints [Fey96, FJ97].

STDs are a state-based formalism, which are best suited for the specification of black-box requirements, i.e. consider only the external interface of a component, but also support compositional reasoning. Other approaches using timing diagrams as property specifications exist, e.g. [Kut94], [Fis99] or [AEKN00]. *Constraint Diagrams* [Die96] are a similar formalism, which allows real-time specifications and is based on the Duration Calculus.

Specification of Communication Properties

The specification formalisms in the preceding section have been developed with a single component in mind. The expressivity of most approaches also allows to state properties about several components, but they are not tailored to this particular use case. With respect to the increasing number of embedded controllers, which are used today and which often also exchange information and commands among each other, an appropriate graphical formalism is needed in order to meet the changed demands. Ideally, such a formalism is not limited to being a graphical front-end for temporal logic, but is suited also for other use cases, as elaborated below.

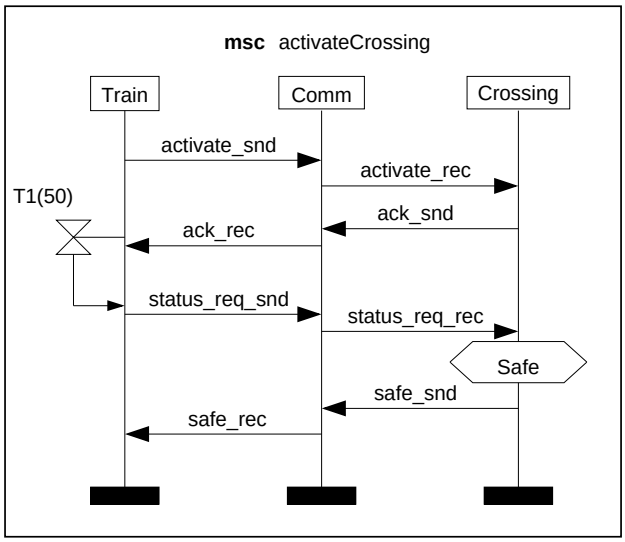


Figure 1.2: MSC example

Message Sequence Charts (MSCs) [IT93] have been used for specifying communication behavior for some time, predominantly in the development of telecommunication systems, and thus are a good candidate for such a formalism. Figure 1.2 shows an example for an MSC describing the message exchange necessary for the securing of a crossing. The three vertical lines, called *instances*, represent the communicating entities: the train, the communication channel and the approached crossing. Messages are depicted by arrows between the instances, the condition `Safe` indicates that the crossing is in a safe state, and the hour-glass symbol represents a timer.

This example demonstrates the intuitiveness of these basic MSC constructs, which motivates their application for the requirement capture in the early development phases, where they are used to document typical interactions, often referred to as *scenarios*. This is also the major use case for *Sequence Diagrams* (SDs), which are a very similar graphical description within the UML and are applied to the same end there. Today, such scenarios serve two purposes: gaining a better understanding of the behavior of the developed application, and documentation. They often also record simulation or test traces. More advanced use cases are conceivable, however, which reuse the early scenarios in later stages of the development process and thus provide an added value. The following use cases show great promise:

Model Synthesis Starting from a set of scenarios, which identify the entities comprising the system under design (SUD) and describe their typical interactions, a first cut of a model is synthesized. From the communication behavior shown in the MSCs or SDs a preliminary model structure and behavioral description is derived, which can be extended manually.

Existential Check Once a model exists it can be checked, if the functionality specified by the early scenarios is possible in the current model, i.e. if it is able to fulfill each behavior described in a scenario at least once. A failed check indicates a fundamental error. This check serves as an early and easy to use debugging aid.

Model Testing When a largely stable model exists and a simulator is available, MSCs/SDs can serve as watchdogs monitoring a user-driven simulation session. Deviations from the specified scenarios are detected and reported. Additionally MSCs/SDs can be used to *drive* the simulation without user interaction by providing the required input stimuli and observing, if the expected model reactions ensue. This use case is another step toward a reference model and is also ideally suited for regression testing, where a set of MSCs/SDs are re-run after a change to the model in order to ensure that the basic original functionality is still guaranteed.

Formal Verification MSCs/SDs can be used to state communication protocols between different entities and employ them for formal verification.

Test Vector Generation Existing scenarios from earlier phases in the development or also newly generated ones can be used to automatically generate test vectors for integration testing of several communicating embedded controllers.

All of the abovementioned use cases demand a formal foundation in order to be realized by corresponding tools. Neither MSCs nor SDs fully comply with this demand. For MSCs a formal semantics exists, but several important issues are covered only inadequately or not at all. Liveness properties, e.g. that a message must arrive at its destination, can for instance not be expressed in the formal semantics of MSCs. Only safety properties are expressible, i.e. that the receipt of a message may only occur after sending. For Sequence Diagrams no formal semantics has been defined so far. Moreover, both languages are lacking expressiveness wrt. to the envisioned advanced use cases. For the verification and model testing use cases e.g. it is vitally important to know when the specified communication behavior should be observed, i.e. when the chart is *active*. A more detailed introduction and criticism of MSCs and SDs in this respect is presented in chapter 3. A in-depth treatment of the state-of-the-art regarding the abovementioned advances use cases is given in the chapters dealing with the individual language constructs of LSCs.

In summary we can state that in general the intuitiveness and visual appeal of MSCs and SDs is very well suited for the abovementioned use cases, but they are lacking expressiveness and an adequate formal semantics. This is where Live Sequence Charts, the central subject of this thesis, come into play. Damm and Harel noted the potential of MSCs and SDs to become more than scenario descriptions, if given a sound and more expressive basis, and proposed Live Sequence Charts (LSCs) in [DH98] as an extension of MSCs and SDs, which addresses these shortcomings. This work is the starting point for the present thesis, where the basic ideas of Damm and Harel are rendered more precise and treated more completely than in [DH98].

The fundamental idea of LSCs is the distinction between *mandatory* and *possible* behavior, where conventional MSCs and SDs are considered as consisting of possible elements only and mandatory elements constitute the enhancements of expressiveness offered by LSCs. This concept is applied to almost all language constructs: entire charts, instance lines, messages, conditions, etc. On the chart level this allows the distinction between *existential* and *universal* LSC specifications, the former being the scenario view of MSCs

and SDs (there *exists* a run, which conforms to the chart) and the latter allowing the specification of protocols, which have to be obeyed by *all* runs of a system. For instance lines and messages mandatory means that *liveness* properties can be specified, i.e. points along an instance line must be reached and messages must be received once sent. This important feature is also the source of the name of Live Sequence Charts. The expressive power of LSCs is additionally enhanced by truly supporting conditions by associating them with a boolean expression (making them "*first-class citizens*" in the words of [DH98]), instead of the informal treatment in MSCs or their absence in SDs. Conditions in LSCs are not limited to single points in time, but may constrain a number of contiguous time points, in which case they are called *local invariants*. Additionally, LSCs allow to specify the activation point of a chart by an *activation condition*. The full set of features is explained in more detail in chapters 4 - 9.

The goal of this thesis is the development of a language for the easy, intuitive, graphical specification of interactions between communicating entities. This language are Live Sequence Charts. The first task in this respect is the definition of the language constructs required to express the properties necessary for a more prominent role of sequence charts[4] in the development process. This involves a non-trivial trade-off between retaining as much intuitiveness as possible on the one hand and providing as much expressive power as necessary on the other. The second major task is the definition of a suitable formal semantics, which unambiguously expresses the meaning of all features and their combinations and thus allows the automated processing of LSCs.

Another point is essential in order to successfully apply any formalism or method in general, and LSCs in particular, in the real world: An indication has to be given in which phases of the development the formalism in question should be applied and to which end. Answering this question is the third goal of this thesis. The final task is the evaluation of the LSC language by applying it to one of the abovementioned advanced use cases: formal verification. The evaluation should demonstrate, if the expressiveness is sufficient and if the semantics is appropriate and useful in practice. This proof of concept will be done by using LSCs for property specification for the formal verification of STATEMATE designs.

[4]We will use the term 'sequence chart' to denote the sum of all dialects, be it MSCs, SDs, LSCs, ...

Formal Verification of Statemate Designs

This section briefly introduces the STATEMATE tool and notations and also gives a short overview over the STATEMATE *Verification Environment* (STVE), into which the tools dealing with LSCs are prototypically integrated.

Statemate

STATEMATE is a CASE-tool, distributed by I-Logix, Inc., USA, which allows to build an abstract model of an SUD, e.g. an embedded controller. It offers to capture several views onto the SUD, the most important ones being the functional decomposition and the behavioral description. Other views like the modeling of continuous aspects or physical distribution of components are possible as well; the details are described e.g. in [HLN+90] or [HP96]. Here only the former two aspects are explained as the STVE is based on these.

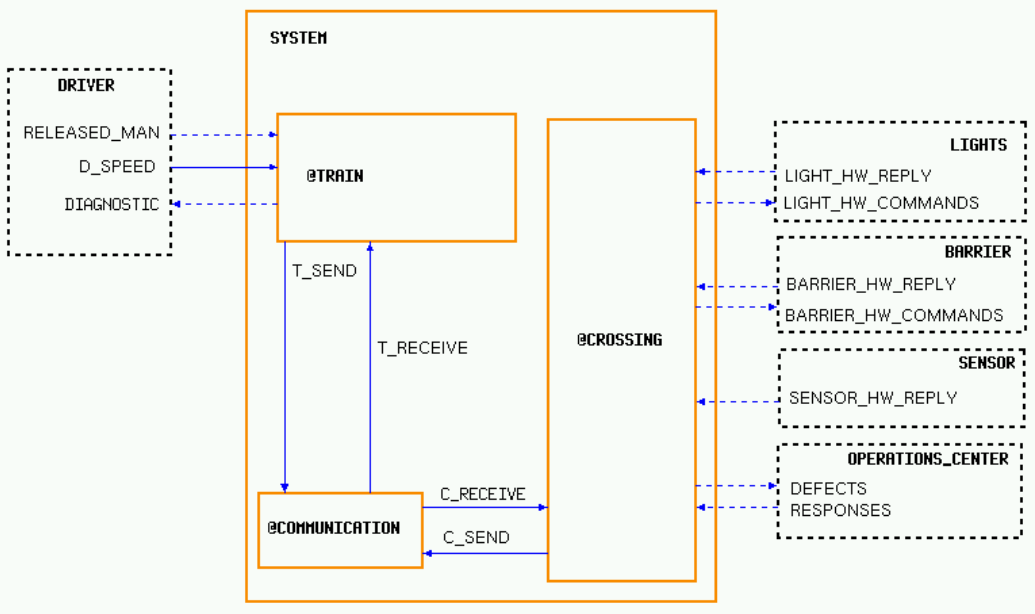


Figure 1.3: Top level Activity Chart

Activity Charts

The functional decomposition of an SUD is modeled by *Activity Charts*, whereas the behavioral description is given by *Statecharts*. Figure 1.3 on the preceding page shows an example for a top-level Activity Chart for the train control system, which is discussed in more detail in chapter 2. Each functional unit, called *activity* in STATEMATE, is represented by a solid line box, e.g. `TRAIN` or `CROSSING` in figure 1.3. The environment is represented by external activities depicted by dashed line rectangles, e.g. `DRIVER` or `BARRIER`, which provide input stimuli or accept outputs of the model. Activities can be structured into a hierarchy, where each Activity Chart represents one level of the hierarchy. The top-level Activity Chart in figure 1.3 for instance contains the three activities shown, which in turn are further decomposed into other Activity Charts, indicated by the '`@`' in front of the activity name. The activities `TRAIN` and `CROSSING` are truly decomposed further as shown in figures 2.3 on page 28 and 2.10 on page 37 in section 2.2, whereas the Activity Chart for `COMMUNICATION` contains only the behavioral description of this component.

On each level of hierarchy a *control activity* can be specified, which is responsible for controlling, i.e. starting, stopping, etc., the other activities present. The control activity is depicted by solid line box with rounded corners (e.g. `SPEED_CONTROL_CTRL` in figure 2.3 on page 28). If no control activity is specified, as is the case in the Activity Chart in figure 1.3, all activities at this level of hierarchy are activated at system start. A control activity located on a level without any other activities, e.g. `SPEED_CONTROL_CTRL`, is one of the possibilities offered by STATEMATE for the specification of behavior.

Information exchange between activities is depicted by arrows leading from the sender to the receiver. The user can distinguish *data* and *control flows*, represented by solid, resp. dashed arrows. Several individual communications leading from one sender to the same receiver can be combined into a single arrow, called an *information flow*.

Statecharts

STATEMATE offers several concepts for the specification of the behavior of an activity. The most important one are Statecharts [HP96, HN96], which can be roughly characterized as automata extended by parallelism and hierarchy. Figure 1.4 on the facing page shows an example for a Statechart,

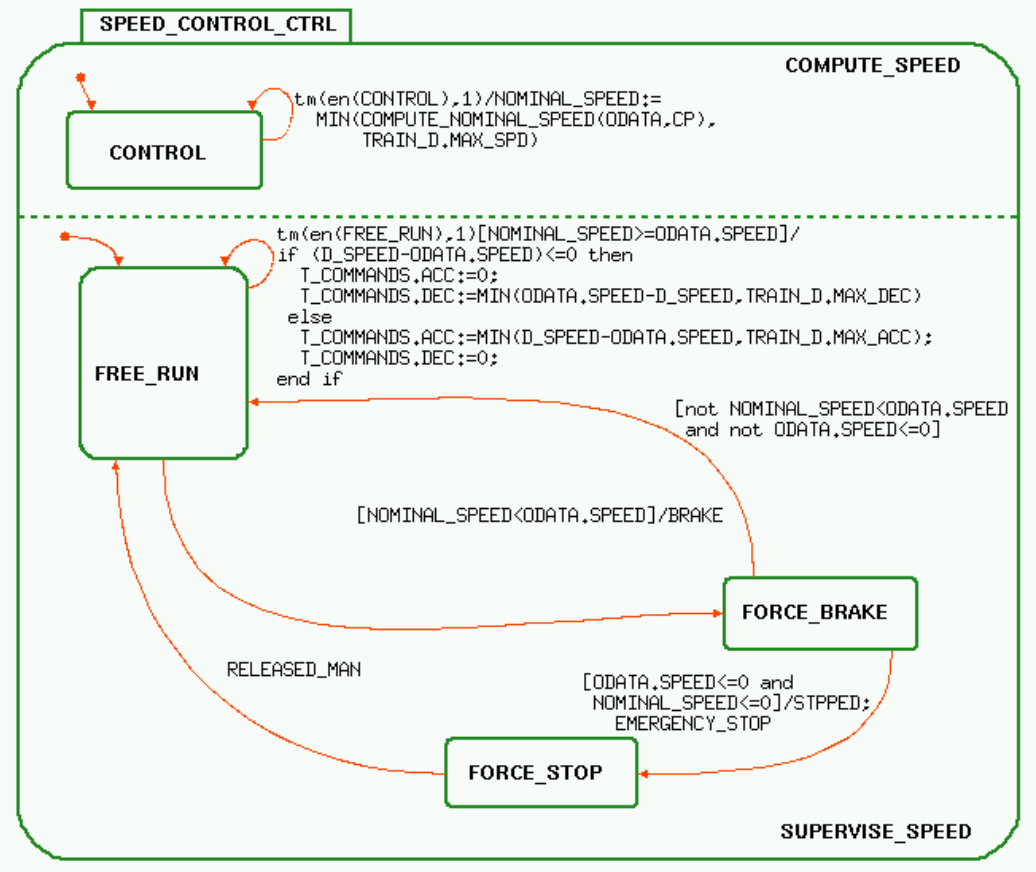


Figure 1.4: Statechart example

again taken from the model of the train control system. The Statechart `SPEED_CONTROL_CTRL` is split into two parallel parts, called *AND-States* in STATEMATE terminology, indicated by the dashed line separating sub-states `COMPUTE_SPEED` and `SUPERVISE_SPEED`. The latter is further structured into three sub-states, so-called *OR-states*. States, which are not decomposed, are called *basic states*. All sub-states of an AND-state are active, when the governing AND-state is active, i.e. both `COMPUTE_SPEED` and `SUPERVISE_SPEED` are active when `SPEED_CONTROL_CTRL` is active, whereas only exactly one of the OR-states at each level of hierarchy may be active at a time, i.e. if `SUPERVISE_SPEED` is active either `FREE_RUN` or `FORCE_BRAKE` or `FORCE_STOP` is active. The entire Statechart is active, if the activity it is contained in is active.

When a Statechart is active it can react on changes to system variables by taking transitions between states. Each transition is annotated by

$$trigger[guard]/action$$

consisting of a *trigger*, which determines when the transition is possibly enabled, a boolean expression *guard*, which further restricts the enabledness of the transition, and an *action* part containing the ensuing consequences. A transition is *enabled*, if the source state is active, the trigger is observed and the guard evaluates to true. An enabled transition need not fire, since there may be other transition, which are enabled concurrently. There are priority rules determining which enabled transition is actually fired, but not all cases can be covered by these rules, so that non-deterministic situations can arise. If a transition is fired, control changes to the target state and the actions are executed, which can consist of the generation of events, variable assignments, control commands for other activities, e.g. `st!(TIMER)` or `sp!(TIMER)` for starting, resp. stopping activity `TIMER`.

Which state of a Statechart or decomposed state is active initially is determined by a *default transition*, which is graphically depicted as a transition without a source state; state `FREE_RUN` e.g. is initially activated in figure 1.4.

Variables in STATEMATE are typed and the user may choose from several data types, one of the most important ones being `Event`, which is a boolean signal visible for one *step* (execution cycle). Other available types are *condition* and *data items*, the latter comprising integer, real, etc.; see [HP96] for more details. Other modeling constructs will be introduced by example when presenting the STATEMATE for the train control case study in section 2.2.

Statemate Simulation Semantics

Part of the STATEMATE tool is a simulator, which allows the interactive execution of the model. Simulation runs can be recorded in *Simulation Control Programs* (SCP) to be replayed later. The simulator supports two execution models: *synchronous* and *asynchronous semantics*, which differ in the underlying time model and the points in time when the embedded controller communicates with its environment.

In the synchronous semantics the model accepts inputs from the environment every step, whereas in the asynchronous semantics new inputs are consumed only when all computations in reaction to the previous inputs have been completed, i.e. a stable state is reached, where no further transition can

be fired without new input stimuli. In the asynchronous semantics reaction to one set of input stimuli thus may entail several internal steps, which do not consume time; time only passes when the system has reached a stable state and synchronizes with its environment. The transition from one stable state to the next is called a *superstep*. The asynchronous semantics is often also referred to as *superstep semantics*, the synchronous one as *step semantics*.

Statemate Verification Environment

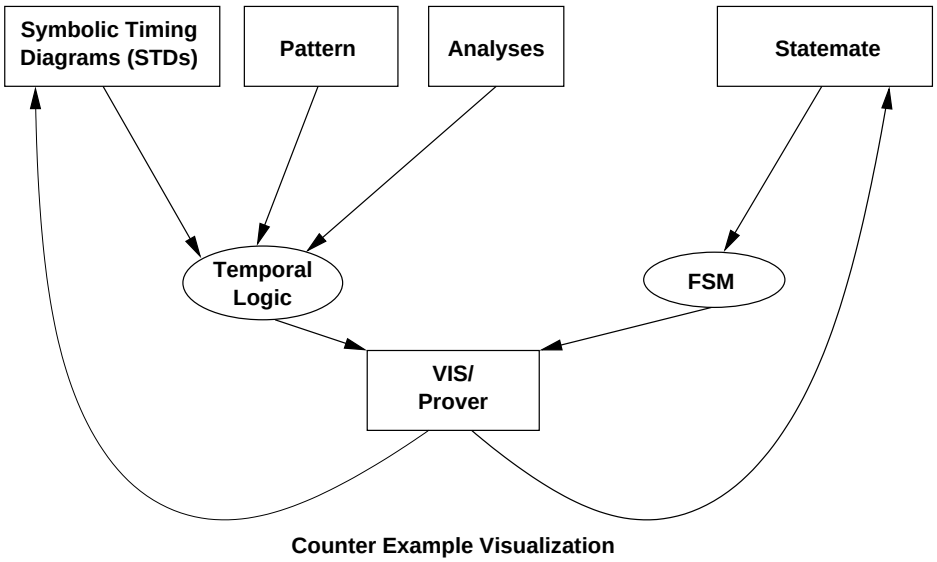


Figure 1.5: Overview over the STATEMATE Verification Environment

The STATEMATE Verification Environment (STVE) has been developed jointly by the embedded systems division at OFFIS e.V.[5] and the University of Oldenburg in order to provide easy to use formal verification capabilities to users, which have no intensive training in formal methods. Part of the techniques and formalisms described in the remainder of this section form a commercial product offering, which is marketed by OSC Embedded Systems AG, Oldenburg, Germany, and I-Logix, Inc., USA. This section gives an overview over the techniques integrated in the commercial product and also other concepts, which are currently not part of the product. We subsume the

[5]Oldenburger Forschungsinstitut für Informatikwerkzeuge und -systeme

entire set of features and techniques under the term STATEMATE Verification Environment. Figure 1.5 on the page before shows the general organization of the STVE.

The offered techniques are grouped into different skill levels ranging from analyses, which can be employed by ordinary designers familiar with STATEMATE to full-fledged property specification and verification capabilities. The increase of expert knowledge required is accompanied by an increase of expressive power: the more knowledge is needed to apply a technique, the more complex properties can be expressed.

The techniques offered are grouped into three categories: robustness checks, pattern-based verification and STD-based verification. The robustness checks are simple, but formal analyses, which can be used by a typical STATEMATE user. They comprise checks/analyses for

- non-deterministic situations:
  - concurrently enabled transitions
  - multiple writer (two or more activities simultaneously write a data item)
  - read-write hazards (a data item is read and written simultaneously)
  - range violations (a data item is assigned an out-of-range value)
- reachability
  - reachability of basic states
  - reachability of state configurations (sets of basic states)
  - reachability of transitions
  - reachability of specific values for data items

Note that the check for a non-deterministic choice between concurrently enabled transitions only reports those situations, which are not already resolved by the STATEMATE priority rules for transitions. These analyses are intended to be used for debugging purposes as the model is developed by answering questions like: "*Are all states of the model reachable?*", "*Is it possible to observe value '7' at output o1?*", or "*Are there situations, where —*

after applying the priority rules — more than one transition is concurrently enabled?".

There are two core verification engines which can be used alternatively: the VIS model checker ([Gro96a, Gro96b]) and a bounded model checker based on the SAT checker ProverCL. By exploiting the counter example generation capabilities of the VIS witnesses are produced, which lead into exactly those situations, which have been checked by the robustness analysis, provided such a situation exists. The general goal of these analyses thus is *falsification*, i.e. the expectation is that a witness exists, e.g. how to reach a certain basic state. This can be exploited by using *reachability-based model checking with early termination*[6] [Gro96a, Gro96b]. Instead of employing the standard backward-oriented fix-point iteration (see e.g. [CGP99]) this strategy checks the formula while performing a forward-oriented reachability analysis. If the formula does not hold, the reachability analysis is aborted (early termination), because a counter example has been found. If the formula indeed fails, this strategy generally performs better than the fix-point iteration, since only part of the reachable states need to be considered.

Falsification is also the prime use case for the practical application of bounded model checking [BCCZ99], which is very efficient for these cases. When a checked property does not hold, SAT-checkers, which form the core of bounded model checkers, are typically more efficient than standard model checkers. The STVE therefore offers bounded model checking via integration of the SAT solver ProverCL [SS98] of Prover Technologies, Sweden, instead of the VIS for robustness checks and other situations when a counter example is expected. Note that both reachability-based and bounded model checking are limited to check invariant formulas of the form '$\mathsf{AG}(p)$'.[7]
More information on the robustness checks of the STVE are found in [BBHW00, BDW00, OSC02b, BDKW01].

The pattern-based verification is a oriented towards certification, instead of falsification, i.e. the specified properties are expected to hold on the model. The STVE offers a library of pre-defined parameterized patterns, which allow to express a set of typical properties. Each pattern is instan-

[6]The term "reachability" used here is a different one than the one used in the above checks. The term here refers to reachable states in the finite state machine (FSM) generated for the STATEMATE design, whereas reachability as used above refers to STATEMATE items (basic states, transitions, etc.).

[7]The bounded model checking procedure described in [BCCZ99] also considers other formulas, but in practice globally formulas are used almost exclusively.

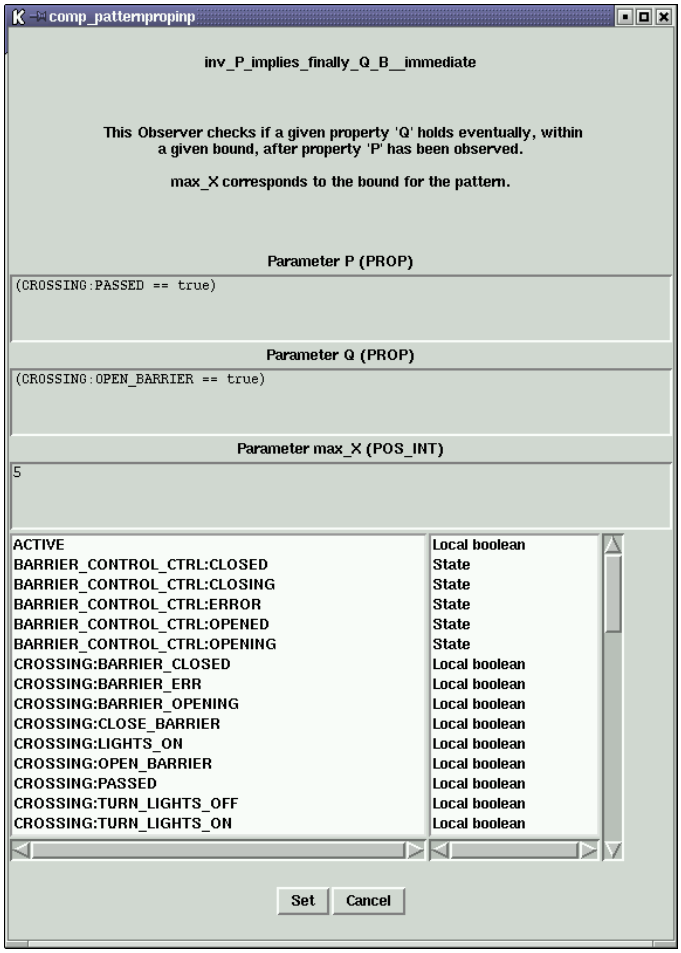


Figure 1.6: Pattern instantiation example

tiated by supplying concrete STATEMATE expressions. An example pattern is e.g. `P_implies_finally_Q_B`, which can be used to express the property "*After the train has passed the crossing, the command for the opening of the barriers must be given within 5 steps.*". Figure 1.6 shows the parameter instantiation dialog for this property. The signal from the pass sensor is mapped to `P`, the open command for the barrier is mapped to `Q` and `B`, the upper bound for the time passed between the two signals, is mapped to 5.

The expressive power of this approach is on the one hand limited by the pre-defined set of patterns and on the other hand by the fact that the patterns for efficiency reasons have been designed to be used with the reachability-

based method. This entails that, in addition to safety properties, bounded liveness properties may be expressed as the example pattern demonstrates, but no unbounded liveness requirements.
More information on the STVE patterns is available in [OSC02a].

If more flexibility in stating requirements is desired or unbounded liveness properties are to be expressed, Symbolic Timing Diagrams (STDs [Sch00, FJ97]) are offered, which allow to specify completely user-defined properties. Figure 1.1 on page 6 shows an example STD, which has been specified for the crossing component of the train control system. The requirement is formulated over the interface objects `CROSSING_SAFE_REC_F` and `PASSED_XING_F` and expresses that the crossing may only be passed by the train after it has indicated its safe state. Properties stated as STDs are checked using the standard fix-point iteration-based model check algorithm. More information on the application of STDs within the STVE can be found in [BW98, BBD+99, DDK99, KM00, DK01].

The overview shown in figure 1.5 on page 15 illustrates the general organization of the STVE. The STATEMATE design to be verified is transformed automatically into a finite state machine (right hand side of figure 1.5) and the property to be checked is translated into a temporal logic formula (left half of figure 1.5). The translation of STDs is split into two phases: first a symbolic automaton is derived for an STD, which in turn is transformed into a temporal logic formula.

If a property specified as an STD or pattern is violated by the model or a witness for a robustness check is found, the (bounded) model checker generates an error path, which can be visualized and examined in two ways. The preferred and most natural way is to translate it into an SCP in order to execute in the STATEMATE simulator. Alternatively the error path can be visualized as an STD.

For more information about the different formalisms and techniques contained in the STVE the reader is referred to the following references: [Bro99] describes the FSM generation for STATEMATE designs, [BDW00, BDKW01, BBHW00, OSC02b] provide more details about the analyses, the pattern-based approach is described in [OSC02a], and information on formal verification of STATEMATE designs using STDs can be found in [BW98, DDK99, KM00, DK01]. Details about the technologies underlying the STVE are contained in [BBD+99, BBB+99, Bie03, Wit03].

Organization of this Thesis

Chapter 2 introduces the train control case study, which serves as a running example throughout this thesis. This case study deals with a radio-based signaling system for the control of level crossings. This chapter contains a general introduction and a detailed description of the STATEMATE model, which will also be used for obtaining experimental results in chapter 11.

The language of Live Sequence Charts is motivated by both the visual appeal and lack of expressiveness and formal rigor of Message Sequence Charts and UML's Sequence Diagrams as has been expounded above. Chapter 3 gives an overview over the two sequence charts dialects, describing the major features, historical development and discussing the shortcomings wrt. the advanced use cases briefly presented above. The complete set of features of the LSC language, along with the definition of the formal semantics, is presented incrementally in the following chapters. Chapter 4 begins with the basic LSC features, whose motivation, graphical representation and informal meaning are described.

The semantics of an LSC is defined in terms of an automaton. The relation between the embedded controller being developed (the SUD) and the LSC specification is established by considering runs of the system and determining, if they are accepted by the automaton. The SUDs, whose properties are to be specified by LSCs, operate for an indeterminate amount of time (theoretically forever), so that the automata used for the semantics definition must be able to deal with infinite runs. Chapter 5 introduces a suitable automata format, derived from *Büchi automata* and also defines a corresponding timed variant thereof. Defining the semantics of LSCs in terms of Büchi automata allows to easily derive temporal logic formulas due to the well-known relationship between (a sub-class of) Büchi automata and LTL. The temporal logic formulas are essential for later formal verification activities.

Chapter 6 defines a formal syntax for the LSC constructs introduced in chapter 4, upon which the algorithm for the generation of the automaton operates. The basic algorithm defined here is extended in the following chapters. The complete semantics of an LSC is then defined on the basis of the generated automaton incorporating the activation and quantification information.

Chapter 7 extends the basic features by additionally considering timing constraints and extending the automaton generation algorithm to produce

a timed automaton. Chapter 8 deals with a feature, which is essential for the use case of formal verification: assumptions. Since the environment of an SUD is already part of the LSC, in form of one or more environment instances, assumptions about the expected environment behavior are easily specified within an LSC by using elements on dedicated instances.

Chapter 9 enhances the activation information not only allowing to consider one point in time, via the activation condition, in order to determine if an LSC is to be activated or not. Additionally a sequence of messages forming a *pre-chart* may now trigger the activation of the actual LSC. The pre-chart semantics is again defined in terms of an automaton.

Chapter 10 addresses the third task specified above and proposes a methodology of how LSCs can be embedded into a model-based development process. The focus is on a re-use of LSCs from early stages in the design process.

The practical application of LSCs to the use case of formal verification is presented in chapter 11. The STATEMATE Verification Environment is covered in more detail and the integration of the LSC tools is described. The major part of this chapter is taken up by the experimental results. Chapter 12 concludes this work with a summary and identification of directions for future work.

Appendix A summarizes all LSCs used in the verification of the train control system, including the corresponding automata. Appendix B lists the contents of all information flows used in the STATEMATE model for the train control system and appendix C contains the grammar of the textual LSC representation.

Chapter 2

Sample Application: A Radio-based Signaling System

This chapter introduces the case study, which will be used as a running example throughout this thesis. Section 2.1 gives a short introduction to the general subject before the STATEMATE model of the case study is presented in section 2.2. This STATEMATE model will also be used to evaluate the concepts and tools described in the remainder of this work; cf. chapter 11.

2.1 General Description

The control of level crossings, crossings for short, is currently carried out by wayside hardware, for example sensors, which announce an approaching train to a crossing, signals, which for instance indicate the status of the crossing to the train driver, etc. This solution is rather inflexible, since the hardware is permanently installed and must be able to handle different trains varying in speed, length, etc. Another drawback is the high amount of maintenance involved to keep signals, sensors and wiring operational. International rail traffic additionally raises demands for more flexibility, since almost every European country uses different signaling technology, so that trains can not easily cross borders. This has prompted railway companies to look for better, more flexible and efficient solutions for the control of trains, crossings, etc. There exist efforts on the European level with the objective of harmonizing and facilitating international rail traffic in Europe: the ERTMS/ETCS (European Rail Traffic Management System/ European Train Control System)

will use radio transmission, among other measures, for the communication between trains and the operations center (then called *radio block controller*). The German railway company Deutsche Bahn investigated a more advanced concept, which proposes to use a radio connection also for the communication between train and crossing and points. This allows more flexibility inasmuch as each train can contact a crossing depending on its specific information, a slow train would e.g. announce itself later than a faster train. This solution also entails lower maintenance effort, since the involved components are located on the train and directly at the crossing, instead of being dispersed along the track.

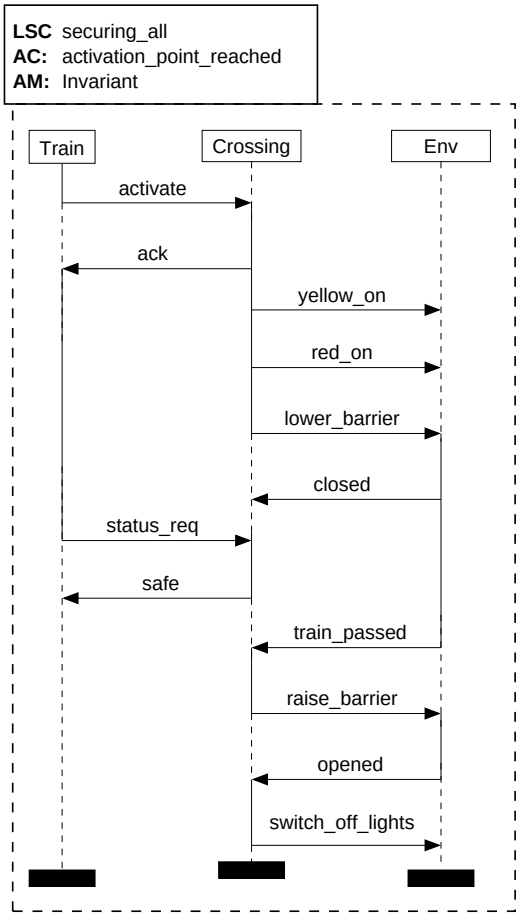


Figure 2.1: Existential LSC showing the typical interaction between train and crossing

There are several different strategies used in the traditional control of crossings by means of wayside equipment, which serve as blueprints for the radio-based approach. This case study considers the radio-based crossing control according to the *guarding signal* strategy. Figure 2.1 shows an existential LSC depicting the typical interaction between train and crossing for this strategy. Even without a detailed understanding of all the features of the LSC language, the graphical nature of this representation is sufficient to be used as an illustration of the protocol between train and crossing.

Once the activation point is reached, indicated by the second row in the LSC header, the train activates the crossing. The activation point is the latest point at which the train can initiate the securing of the crossing, if it is to be passed without braking. In the hard-wired control this point was given by a fixed sensor, which sent a signal to the crossing. In the radio-based version this point can be determined dynamically. The exact position of this point must consider the delays for communication setup, message transmission, the time necessary to secure the crossing, an additional safety interval and the speed and position of the train. The crossing acknowledges the receipt of the activation request and starts the securing procedure by switching on first the yellow and then the red light of the traffic lights in order to warn the car traffic. Then the command to close the barriers is given and once the barriers are indeed closed, the crossing is in a secured state.

After the amount of time has passed, which in ordinary circumstances is needed to secure the crossing, the train requests a status report from the crossing. Here, the crossing responds with the report *safe*. Should the crossing not be in a safe state when the status request arrives, no response is given. In the hard-wired control this corresponds to the crossing setting the guarding signal to *go* in the safe case, and leaving it set to *stop* in the unsafe case.

A pass sensor determines when the train has passed the crossing and sends a corresponding signal to the crossing controller. Then the crossing is returned into its normal state, i.e. the barriers are opened again and the traffic lights are switched off.

Before a train can approach and contact a crossing, it needs to know the position of the crossing. This information is stored in a *track chart* along with other details about the track, like maximal velocity, position of crossings, points, stations, etc. Once the train has been granted movement authorization for a track segment, it looks up the relevant information in the track chart and places *control points* at all potentially dangerous points,

which can e.g. be track segments with a lowered maximally allowed speed, crossings, points, stations, or the end of the assigned track segment. With each control point a target speed is associated, which must be observed; for not secured crossings or not set points for instance this speed is zero, so that the train e.g. has to stop in front of a crossing. Control points due to crossings, points, stations or the end of the assigned track segment require some action on the part of the train: a crossing needs to be secured before it can be passed, a switch must be set to the right track, the train should stop at a station it is supposed to service, and movement authorization for a subsequent track segment must be requested before reaching the end of the currently assigned segment.

The control points are considered by the train in the calculation of the maximal velocity for each point in time. This speed profile is the basis for the speed supervision, which controls that the train does not go faster than allowed by track, train and control points. Once a control point has become irrelevant, e.g. because a crossing has been secured and it is no longer necessary that the train stops, it is disregarded for the maximal speed computation and thus no longer restricts the train's velocity.

This case study focuses on one aspect of the entire set of tasks necessary for radio-controlled train operation: the control of level crossings. Points, stations, etc. are neglected and it is assumed that movement authorization has been granted. The considered type of crossing guards a single track and its barriers cover only one side of the street.

2.2 STATEMATE Model

The STATEMATE model of the radio-controlled crossing bases on a model, which has been developed in [KT00], but has been slightly adapted to our needs in this thesis. Earlier versions have been partly described in [DDK99, KM00, DK01]. The model presented here uses the asynchronous simulation semantics of STATEMATE, a variation using the synchronous semantics exists as well, but is not described in detail here, since the differences are only minor.

2.2.1 Activity Chart `SYSTEM`

Figure 2.2 on the facing page shows the top level Activity Chart (`SYSTEM`) of the radio-based crossing control. The two main activities are `TRAIN` and

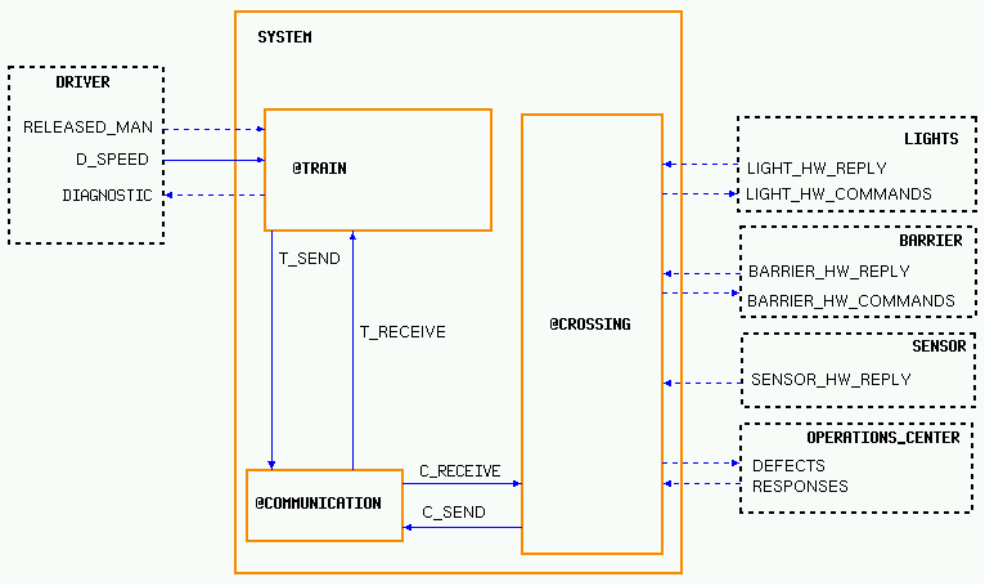


Figure 2.2: Top level Activity Chart

CROSSING and the activity COMMUNICATION connecting them. All three internal activities are each further described in a separate Activity Chart as indicated by the '@' in front of the activity name. The environment of the modeled part of the crossing control system is given by the external activities, indicated by the dashed borders. On the train side the environment consists of the console interface to the train driver (activity DRIVER) and on the crossing side there are the sensors and actuators for the peripheral elements. Additionally there is the operation center, to which defect messages are directed and which can send responses to these. The DRIVER determines by D_SPEED the desired train speed and may re-start the train after an emergency stop by pressing the manual release button (RELEASED_MAN). The information flow DIAGNOSTIC contains two events for the indication of the train stopping in front of a not secured crossing and for the passing of the crossing, respectively.

The activity TRAIN (cf. section 2.2.2 on the next page) implements only those parts of the train, which are necessary for the protocol considered here. These are the control of the speed and applying the brake if needed, the odometer for keeping track of the position and speed, and the major task: the activation of the crossing.

The implementation of a crossing in activity CROSSING (cf. section 2.2.4 on page 37) contains sub-controllers for all peripheral elements (lights, barrier, pass sensor) as well as the overall control for securing the crossing.

The information flow between TRAIN and CROSSING is established by means of activity COMMUNICATION (cf. section 2.2.3 on page 36). The information exchange takes place via the information flows T_SEND and T_RECEIVE between TRAIN and COMMUNICATION, respectively C_SEND and C_RECEIVE between CROSSING and COMMUNICATION. The signals sent in T_SEND and C_SEND are relayed by COMMUNICATION to the corresponding receive channel (C_RECEIVE, resp. T_RECEIVE). The exact contents of information flows is listed in appendix B.

2.2.2 Activity Chart TRAIN

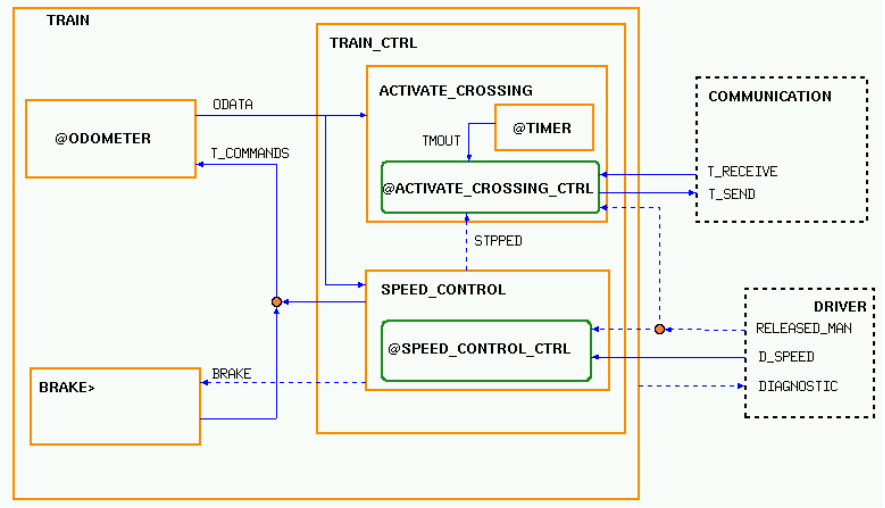


Figure 2.3: Activity Chart TRAIN

The train consists of several sub-components as shown in figure 2.3. The activity TRAIN_CTRL forms the core of the train comprising the two functionalities of communication with the crossings (ACTIVATE_CROSSING) and supervising the train's speed (SPEED_CONTROL). The activity ODOMETER keeps track of the train's speed and position and the BRAKE slows down the train, if necessary.

Statechart SPEED_CONTROL_CTRL

The behavior of activity SPEED_CONTROL is described by Statechart SPEED_CONTROL_CTRL shown in figure 2.4, which consists of two parallel sub-states: COMPUTE_SPEED computes the maximally allowed speed (NOMINAL_SPEED), whereas SUPERVISE_SPEED checks if the current train speed is below the maximum.

The computation of the NOMINAL_SPEED is done every super-step utilizing a timeout event and the implicit *entered* event for state CONTROL: en(CONTROL). The actual computation is done by function COMPUTE_NOMINAL_SPEED, which takes the dynamic train data (ODATA) and the next control point (CP) as parameters. A control point may be set, i.e. it must be observed and the target speed at its position is thus zero, or deleted, i.e. it need not be observed any more and the target speed at its position is set to the maximal value for this track segment. All control points in a track segment are set by default when movement authorization is granted to the train for this segment.

The function COMPUTE_NOMINAL_SPEED checks, if control point CP still needs to be observed and if so, determines the maximal speed depending on the train's current distance to the control point:

```
if CP.ALREADY_REGARDED==true then
  RET:=30
else
 if TRAIN.POS>=CP.POS then
    RET:=0
  else
    DIST:=CP.POS-TRAIN.POS;
    if DIST<=10 then RET:=0
     else if DIST<=30 then RET:=5
       else if DIST<=50 then RET:=10
         else
         if DIST<=80 then RET:=20
           else RET:=30
           end if
         end if
      end if
    end if
```

```
  end if
end if;
return(RET)
```

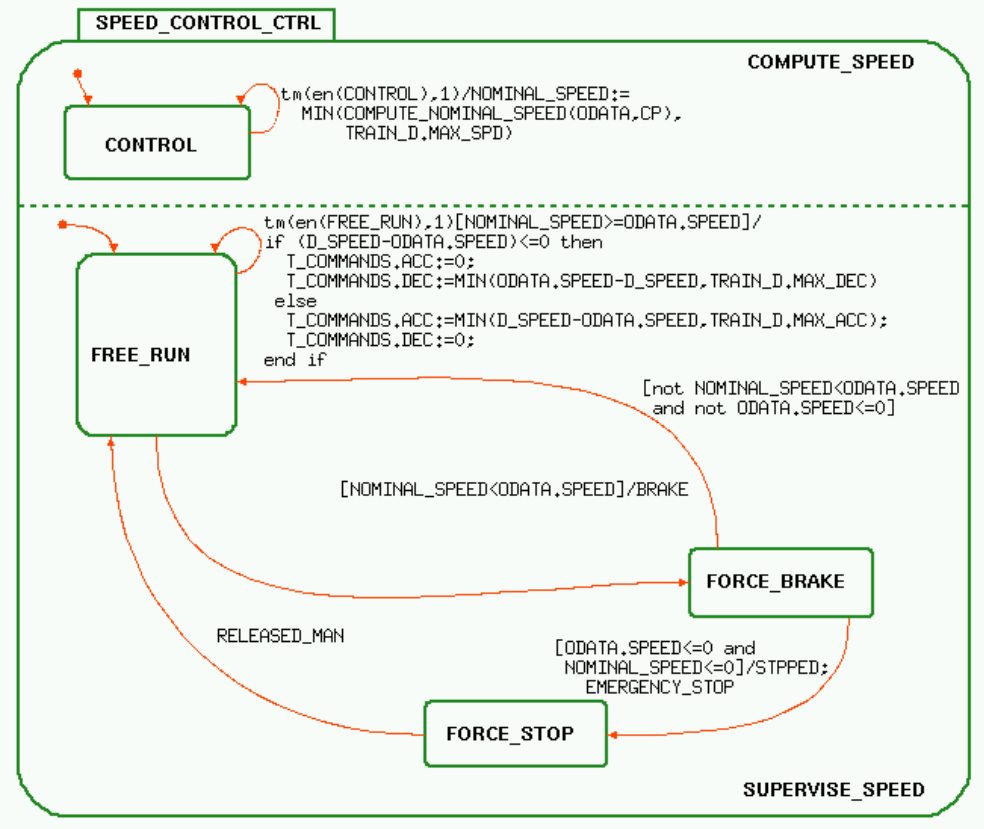


Figure 2.4: Statechart SPEED_CONTROL

The resulting speed is the maximal speed for the current position and can not be larger than the maximal speed allowed for this track segment. Taking the minimum of the computed value and the train's maximal speed yields the overall maximal speed NOMINAL_SPEED.

The lower sub-state checks if the current train speed (ODATA.SPEED) is within the allowed range. If so, the Statechart stays in state FREE_RUN and computes the new acceleration and deceleration values depending on the speed set by the driver (D_SPEED). This adjustment is performed by the self loop on FREE_RUN as long as the train does not exceed its allowed speed

as computed by sub-state COMPUTE_SPEED (condition part of the transition annotation of the self loop). The implemented algorithm is fairly simple: If the speed set by the driver is lower than the current train speed, the train is decelerated by the difference, otherwise it is accelerated. If the two speeds are equal, the train is neither accelerated nor decelerated.

If the train speed is greater than the allowed speed, state FREE_RUN is exited, FORCE_BRAKE is entered and the brake is activated. Once the speed is less than the maximal speed again and the train has not stopped, the Statechart returns to FREE_RUN. If the NOMINAL_SPEED and the actual train speed are both zero, the train has stopped in front of a still active control point, state FORCE_STOP is entered and SPEED_CONTROL_CTRL emits the event STPPED to ACTIVATE_CROSSING_CTRL and EMERGENCY_STOP to the driver. Note that only those stops are covered by STPPED, which result from NOMINAL_SPEED dropping to zero. If the train stops for other reasons, it may resume its course on its own without intervention by the driver. In the former case, however, the driver has to manually confirm that the crossing may be passed safely (RELEASE_MAN) in order to continue. SPEED_CONTROL_CTRL then switches back into the normal behavior of state FREE_RUN.

Statechart ACTIVATE_CROSSING_CTRL

The activity ACTIVATE_CROSSING (cf. figure 2.3 on page 28) contains the Statechart ACTIVATE_CROSSING_CTRL shown in figure 2.5 on the next page, which handles the communication between train and crossing, and a timer, which is used to supervise that the train reaches a secured crossing in time. The communication behavior of the train is described by ACTIVATE_CROSSING_CTRL. Once the train reaches the activation point — indicated by the condition V_ACTIVATION_POINT_P[1] — it changes its state from IDLE to WF_CROSSING_SAFE, which realizes both the setup of the communication channel and the protocol between the train and the crossing. The detailed behavior is shown in Statechart WF_CROSSING_SAFE in figure 2.6.

On entering WF_CROSSING_SAFE the communication channel with the crossing is setup. Once the connection has been established, the train requests the securing of the crossing by emitting the event

[1]Note that the conditions marked by the suffix _P represent procedures, which are not included in the model. The conditions thus represent the result of the procedures. Condition V_ACTIVATION_POINT_P abstracts from the computation of the activation point for a crossing.

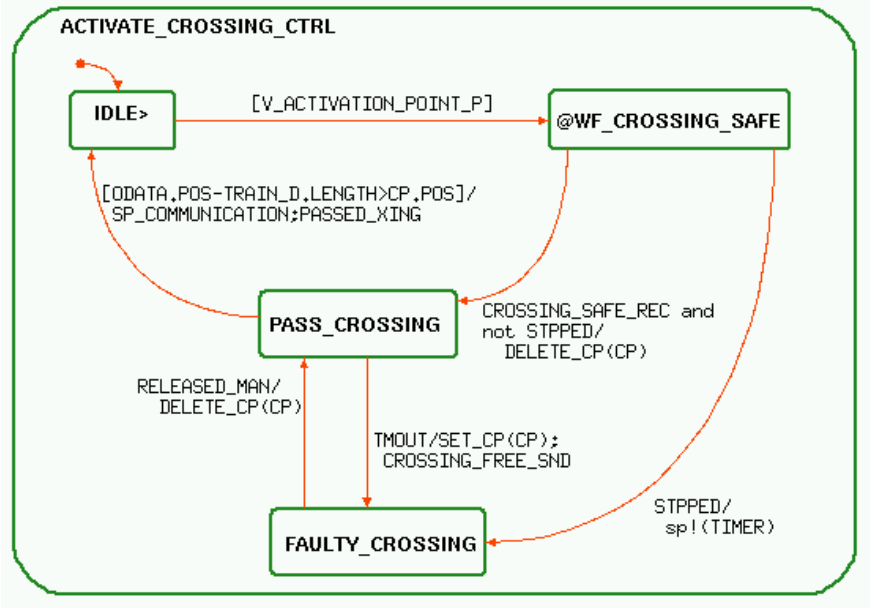


Figure 2.5: Statechart `Activate_Crossing`

`ACTIVATE_CROSSING_SND`. On receiving the acknowledgment from the crossing (`ACK_REC`), the train sends the status request (`STATUS_RQ_SND`) after waiting for the amount of time needed by the crossing to carry out the securing procedure. This is the crossing closing time `CCT`. Simultaneously with the status request a timer is started, which supervises that the train reaches the crossing in time (see below). In the last state of `WF_CROSSING_SAFE` the status message of the crossing is awaited.

Returning to `ACTIVATE_CROSSING_CTRL`, first the normal case is considered, i.e. the report indicates a successful securing of the crossing (`CROSSING_SAFE_REC`). State `PASS_CROSSING` is entered and the control point is deleted by function `DELETE_CP`. When the train now passes the crossing – represented by its control point – the communication channel is closed (`SP_COMMUNICATION`) and the train is ready for the next crossing.

During the securing procedure two error situations can arise: First, there may be problems at the crossing, so that it is not in a safe state when the status request arrives. In this case it does not answer and therefore the train cannot delete the control point with the result that it stops in front of the crossing. This is indicated by Statechart `SPEED_CONTROL_CTRL`

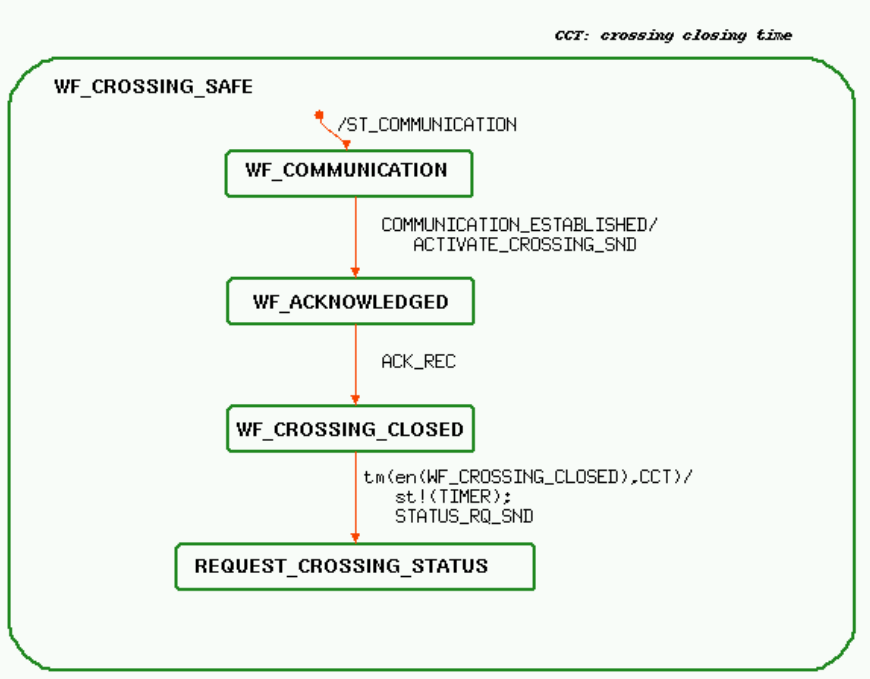


Figure 2.6: Statechart `WF_CROSSING_SAFE`

with the event `STPPED` triggering the transition from `WF_CROSSING_SAFE` to `FAULTY_CROSSING`. Since the train has stopped already, the timer becomes irrelevant and is stopped. In this situation the driver has to manually confirm that the crossing can be safely passed (`RELEASE_MAN`) entering state `PASS_CROSSING` and deleting the control point in order to allow the train to pass it.

The second error situation arises when the crossing has answered the status request but the train is unable to pass it within the maximal barrier closed time. This is indicated by the event `TMOUT` sent by the timer resulting in changing from state `PASS_CROSSING` to `FAULTY_CROSSING`, setting the control point again via the function `SET_CP` and notifying the crossing that the train will not reach it in time (event `CROSSING_FREE_SND`). Again this situation has to be resolved by the driver.

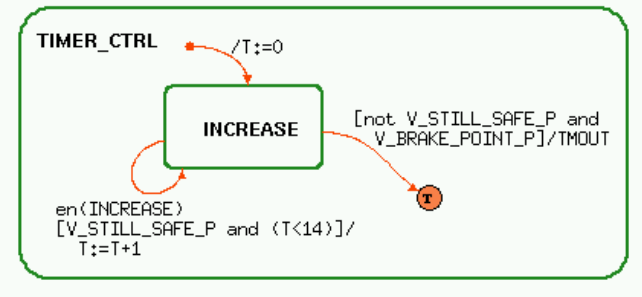


Figure 2.7: Statechart TIMER_CTRL

Statechart TIMER_CTRL

The timer depicted by Statechart TIMER_CTRL[2] in figure 2.7 supervises that the train reaches an already secured crossing before the maximal barrier closed time elapses, i.e. the maximum amount of time that a crossing may stay closed without a train passing it. The reason for this upper limit is that car drivers tend to become impatient, if no train passes after some time, and start to drive around the barriers[3], which creates a highly dangerous situation. A timeout (TMOUT) is generated when this property is violated.

The timer implements a simple counter which increments counter variable T. The timeout is generated depending on two conditions, V_STILL_SAFE_P and V_BRAKE_POINT_P, which again represent the result of procedures:

V_STILL_SAFE_P indicates, if the train will pass an already *secured* crossing within the maximum barrier closed time. The computation of the value of this condition depends on the current train speed and position.

V_BRAKE_POINT_P indicates, if the train has reached the braking point, i.e. the point where it would have to start braking in order to safely stop in front of the crossing. This computation also depends on the current train speed and position.

The behavior suggested by the transition to the termination connector in figure 2.7 is thus the following: If the train reaches its braking point and is unable wrt its maximally allowed speed to reach the crossing before the

[2]The Activity Chart TIMER contains only the Statechart TIMER_CTRL, so that it is omitted here.

[3]Recall that the barriers only cover half of the street on each side of the crossing.

maximum barrier closed time elapses, the timeout is generated and the timer is stopped.

Odometer and Brake

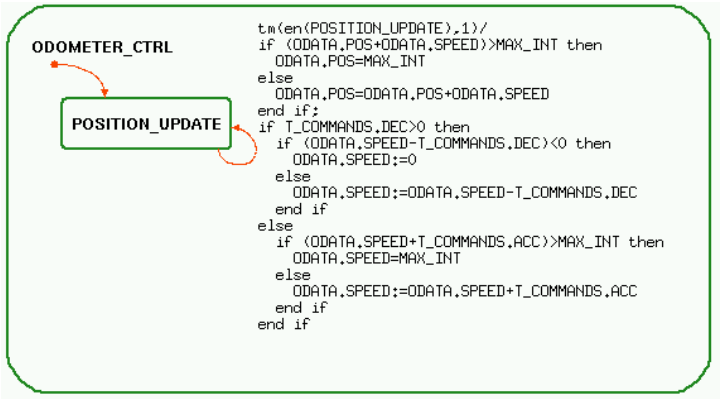


Figure 2.8: Statechart ODOMETER_CTRL

The activity ODOMETER is responsible for the determination of the speed and position of the train; its behavior is given by Statechart ODOMETER_CTRL shown in figure 2.8. The current speed is added to the current position yielding the new position and the new speed is computed by either subtracting the current deceleration value or adding the acceleration value. The activity BRAKE is activated whenever the train exceeds its maximal speed (cf. Statechart SPEED_CONTROL_CTRL in figure 2.4 on page 30). Once activated the brake decelerates as fast as possible, until the speed is within the legal range again or the train is stopped:

```
BRAKE/
    T_COMMANDS.ACC:=0;
    T_COMMANDS.DEC:=TRAIN_D.MAX_DEC
```

Note, however, that the brake is only activated upon violation of the maximal speed, but not when the driver requests a lower speed than the maximally allowed one. The second case is handled by the self loop of state FREE_RUN in Statechart SPEED_CONTROL_CTRL. Also note that there is no interference between the two cases, since the brake is only activated when FREE_RUN is left.

2.2.3 Activity Chart COMMUNICATION

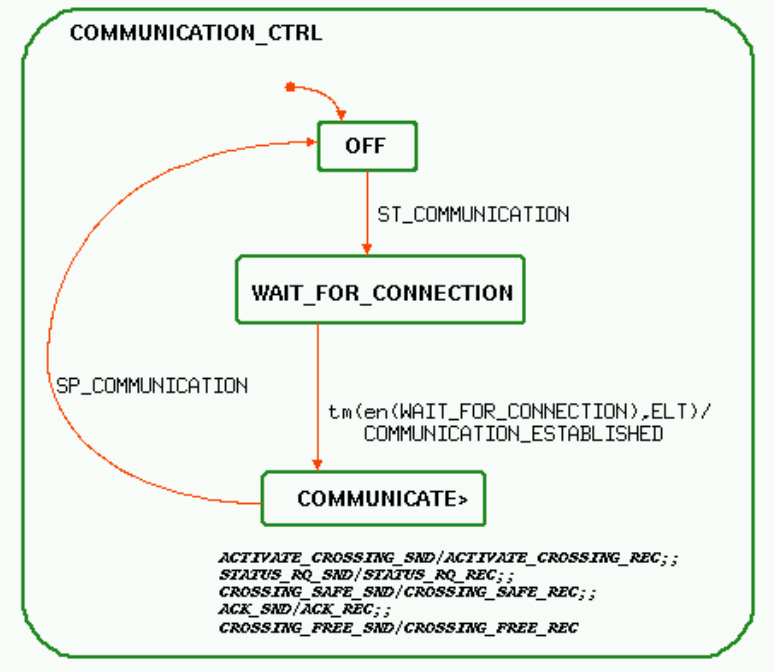


Figure 2.9: Statechart COMMUNICATION_CTRL

The activity COMMUNICATION represents the radio link between TRAIN and CROSSING. It is explicitly modeled in order to provide the possibility to simulate errors when establishing the connection and to be able to consider communication delays. The current implementation of the STATEMATE model however assumes that the communication setup is always possible and that there is no transmission delay.

The train initiates the connection setup by ST_COMMUNICATION whereupon control in Statechart COMMUNICATION_CTRL changes to state WAIT_FOR_CONNECTION (see figure 2.9). After the time necessary to setup the connection (establishing lag time, ELT) has passed, the radio connection is ready and an acknowledgment is sent to the train (COMMUNICATION_ESTABLISHED).

The further interactions take place instantaneously, i.e. requests and acknowledgments are passed on without delay. The original events in T_SEND resp. C_SEND are indicated by the suffix _SND, which becomes _REC as the

events are relayed to the corresponding receiving channels C_RECEIVE resp. T_RECEIVE. The exact contents of the information flows is listed both as a comment in Statechart COMMUNICATION_CTRL and in appendix B and appear both on the crossing and train side (cf. sections 2.2.2 on page 28 and 2.2.4).

The termination of the communication between TRAIN and CROSSING is triggered by the train via SP_COMMUNICATION which leads to an immediate abortion of the communication. This is modeled by changing into the state OFF in the COMMUNICATION_CTRL.

2.2.4 Activity Chart CROSSING

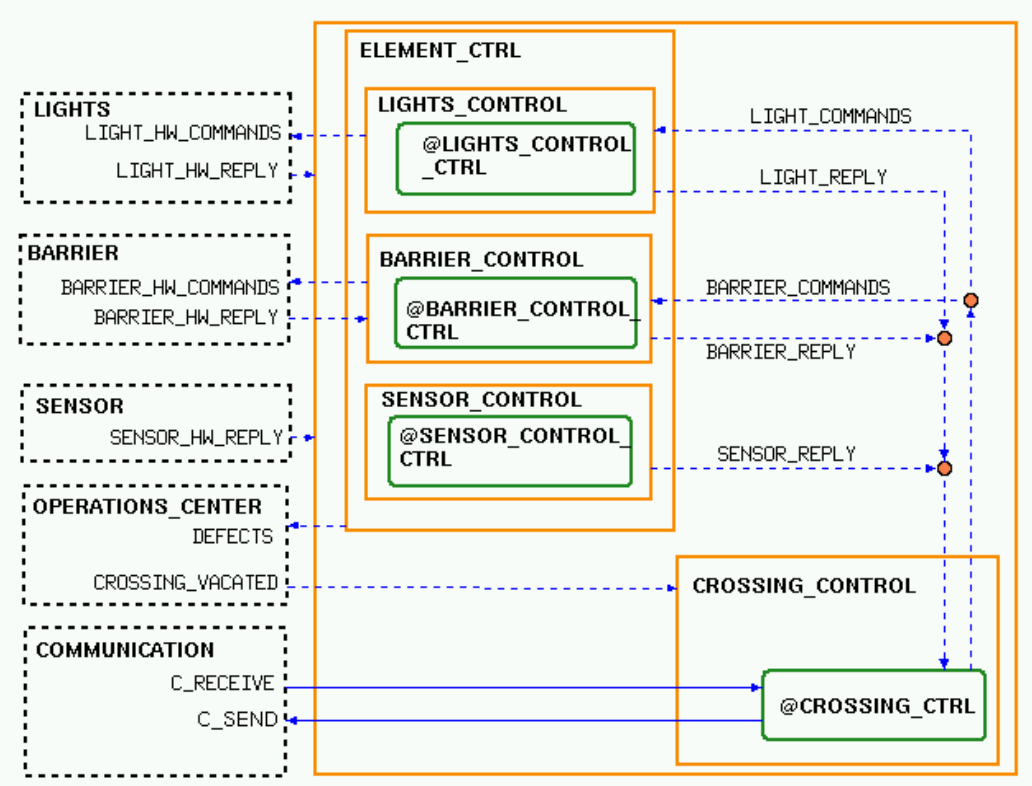


Figure 2.10: Activity Chart CROSSING

The activity CROSSING shown in figure 2.10 contains the overall software control of the crossing (CROSSING_CONTROL) responsible for the coordination of the whole securing process and the software control (ELEMENT_CTRL)

for the involved peripheral elements consisting of LIGHTS_CONTROL, BARRIER_CONTROL, and SENSOR_CONTROL.

Statechart CROSSING_CONTROL

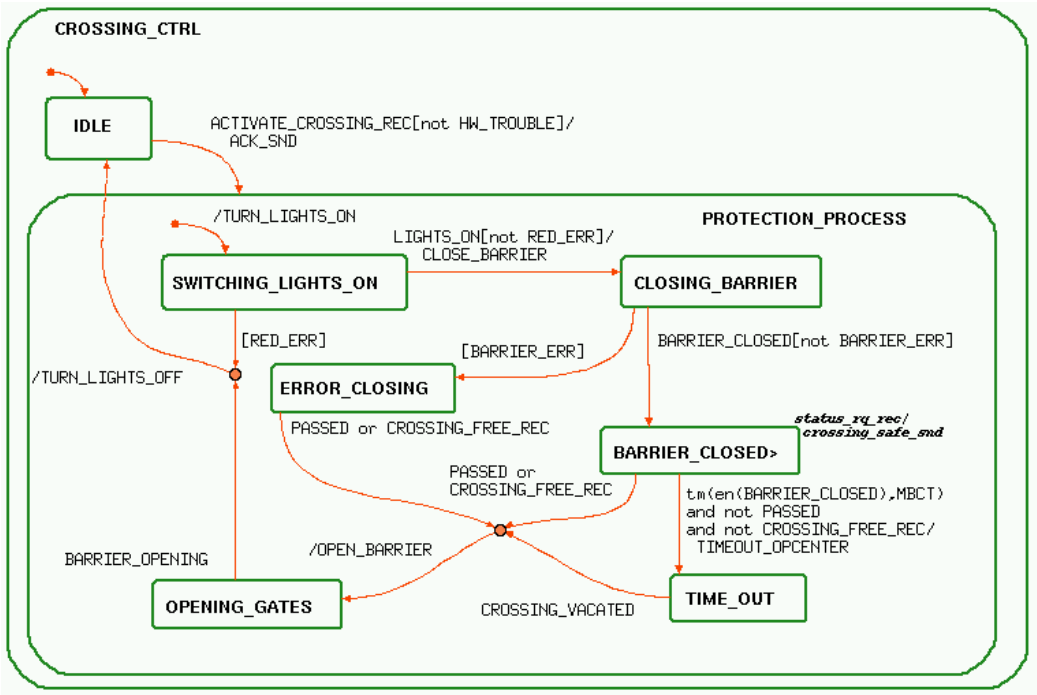


Figure 2.11: Statechart CROSSING_CTRL

The function of this component is the coordination of all measures while securing or unsecuring the crossing. The initial state IDLE remains activated until the crossing controller receives the signal ACTIVATE_CROSSING_REC; thereupon state PROTECTION_PROCESS is entered, which encapsulates all further states. The securing procedure is only begun, if the all peripheral elements of the crossing are operational. This is indicated by the condition not HW_TROUBLE on the transition from IDLE to PROTECTION_PROCESS. HW_TROUBLE is a shorthand notation for (not RED_ERR) and (not BARRIER_ERR) and (not SENSOR_ERR).

Upon entering the PROTECTION_PROCESS the state SWITCHING_LIGHTS_ON is activated and the event TURN_LIGHTS_ON is sent in order to activate

LIGHTS_CONTROL. The overall control remains in this state until either receiving the event LIGHTS_ON or a red light malfunction is detected (RED_ERR), both events originating in activity LIGHTS_CONTROL. A red light failure results in an abortion of the rest of the protection process, since it is too dangerous to close the barriers without warning by a light signal. The crossing controller thus returns to the IDLE state.

If the traffic lights have been successfully switched on, the lowering of the barriers is initiated by sending event CLOSE_BARRIER to BARRIER_CONTROL. If the barrier is closed successfully, the event BARRIER_CLOSED is emitted by the barrier controller and the crossing enters its corresponding state. If the barrier fails to close or does not close in time, BARRIER_ERR is set to true and the error state ERROR_CLOSING of CROSSING_CTRL is entered. Note that this failure does not result in a complete abortion of the securing procedure, but that the traffic lights remain on. The reason is that (according to German law) the crossing is considered secured, if the lights are red, i.e. the barriers are only an additional safety measure, since red lights are more easily overlooked by cars than a closed barrier. The crossing nevertheless does *not* answer the status request after a barrier failure for the same safety considerations.

Once state BARRIER_CLOSED has been reached, the crossing is in a safe state and awaits the status request of the train. Only in this state the status request is answered by event CROSSING_SAFE_SND, which is generated by the state's static reaction (represented by the '>' after the state name; the contents of the static reaction is given by the comment next to state BARRIER_CLOSED in figure 2.11).

The overall controller stays in state BARRIER_CLOSED until the train has passed (indicated by event PASSED of the SENSOR_CONTROL), the train informs the crossing that it will not reach it in time (CROSSING_FREE_REC; cf. section 2.2.2) or the maximum barrier closed time (MBCT) has expired. The last possibility ensures that a crossing does not remain closed too long, if e.g. the pass sensor is defect and the train has already passed the crossing without the crossing noticing. In this case the TIME_OUT state is entered and a corresponding event (TIMEOUT_OPCENTER) is sent to the operations center, which then determines if and when the crossing can return to its unsecured state.

Once the train has passed the crossing or announced by the free message that it will not reach it in time, the crossing controller sends the command to open the barriers (OPEN_BARRIER) to the barrier controller. This is done even when the barriers have malfunctioned during the previous closing. Once the barriers leave their lower position indicated by event BARRIER_OPENING,

the lights are switched off and the Statechart CROSSING_CTRL returns to its initial state and waits for the next train. Note that the crossing is returned to its open state as soon as the barriers are opening, i.e. it is not necessary that the barriers are open all the way in order to switch off the traffic lights.

Statechart LIGHTS_CONTROL_CTRL

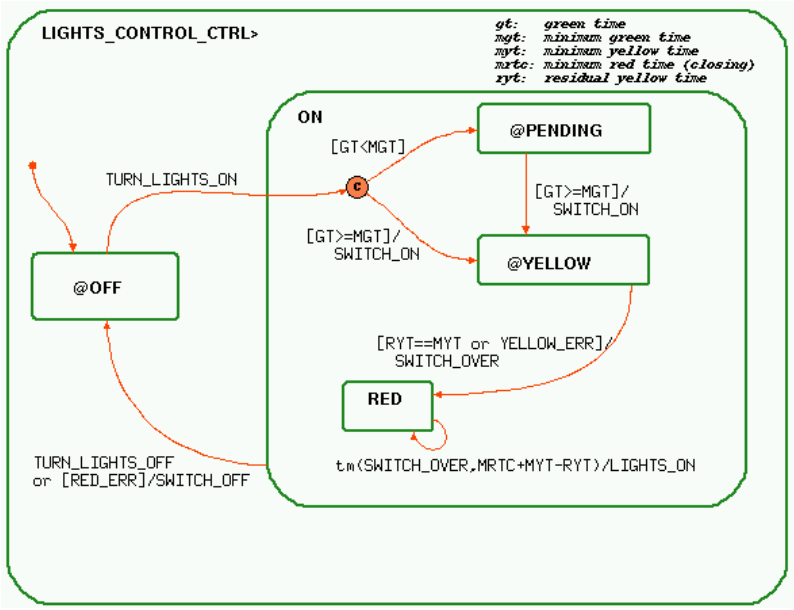


Figure 2.12: Statechart LIGHTS_CONTROL_CTRL

The Statechart of the controller for the traffic lights contains two fundamental states as shown in figure 2.12: the states OFF and ON. Entering OFF (see figure 2.13) the SWITCH_OFF signal is sent to the hardware to ensure that all lights are turned off and counter GT is reset. The timer counts the *green time*, i.e. the time the traffic lights have been off since their last activation. This is required in order to ensure that the *minimum green time* (MGT) is respected, i.e. there must be a certain amount of time between two subsequent activations, where the crossing can be passed by car traffic. Similar to the timer in the train (cf. section 2.2.2), GT is incremented every superstep. Once the MGT is reached no further increment is necessary.

The reaction to the command to switch on the lights (TURN_LIGHTS_ON) differs depending on the value of GT: If it has reached the MGT, the yellow

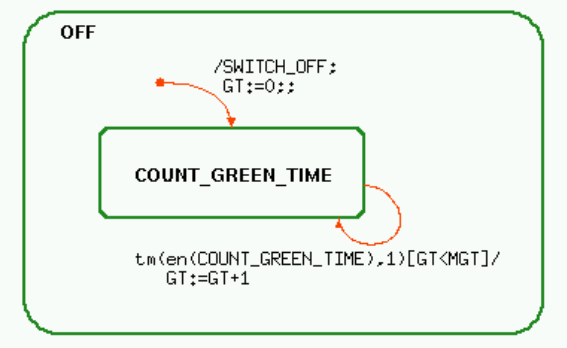


Figure 2.13: Statechart OFF

light is switched on immediately (event SWITCH_ON). If the minimum green time has not yet passed, the activation of the yellow light is delayed and the PENDING state is entered, which continues counting the green time. This state is identical to OFF and is thus not shown explicitly here. Once the MGT has been reached the switch on command is issued.

The YELLOW state, which is similar to the states OFF and PENDING, counts the *elapsed yellow time* (EYT), which measures the period of time the yellow light is on. Once the yellow light has been on for the required amount of time, the light changes to red (event SWITCH_OVER). If the yellow light is not operational or fails during operation, indicated by the condition YELLOW_ERR, the red light is switched on immediately and left on for the combined amount of required yellow and red time.

The red light must be on for a certain amount of time, the red time (maximum red time (closing), MRTC), before the barriers may be lowered. Therefore the timeout event on the self loop on state RED waits until the red time has passed before sending the event, that the traffic lights have been switched on successfully (LIGHTS_ON). In the case of a yellow light failure the remaining yellow time (RYT) is added to the red time. The lights controller returns to its initial state when either the command to switch off the lights is received from the crossing controller or the red light fails, indicated by the condition RED_ERR.

The static reactions contained in LIGHTS_CONTROL_CTRL send a defect message to the operations center when either a yellow or red light failure occurs (events YELLOW_DEFECT, resp. RED_DEFECT).

Statechart `BARRIER_CONTROL_CTRL`

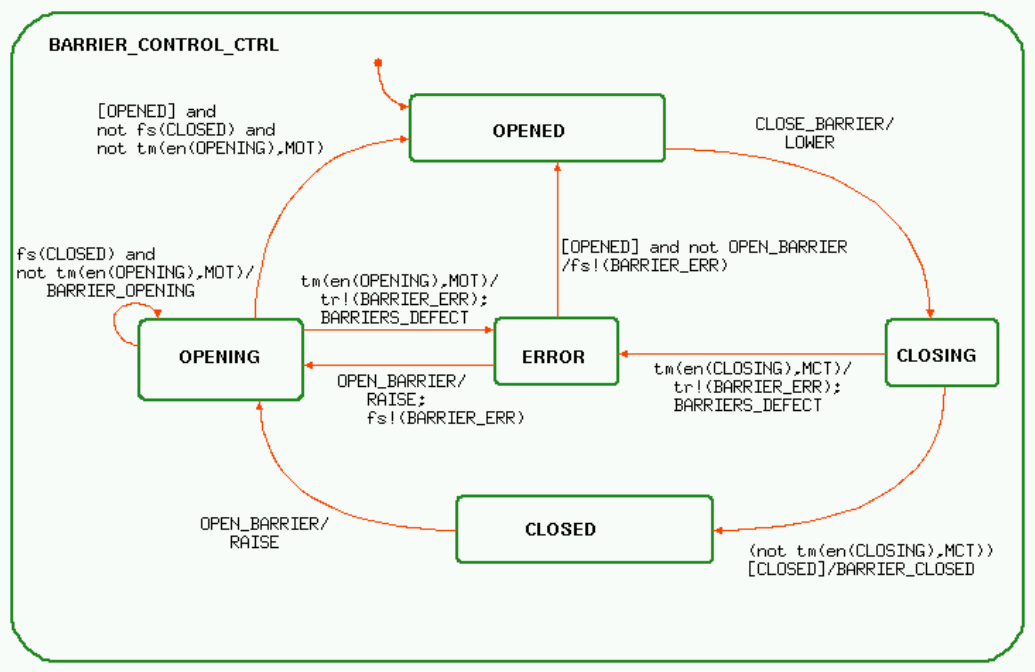


Figure 2.14: Statechart `BARRIER_CONTROL_CTRL`

The barrier controller shown in figure 2.14 starts in the `OPENED` state and upon receiving the command to close the barriers (`CLOSE_BARRIER`) from the crossing controller sends the corresponding event `LOWER` to the barrier actuators and waits for the result in state `CLOSING`. If the barriers close in time, i.e. reach their lower end position, indicated by condition `CLOSED`, within the *maximum closing time* `MCT`, the corresponding state is entered and the command for the opening of the barriers (`OPEN_BARRIER`) is awaited. If the barriers are closing too slowly or not at all, the timeout event for the `MCT` is observed, the error state is entered and a barrier error is reported both to the crossing controller (via condition `BARRIER_ERR`) and to the operations center (via event `BARRIERS_DEFECT`).

Both in the `ERROR` and `CLOSED` state the barrier controller reacts to the opening command by sending the event `RAISE` to the barrier actuators and changes to state `OPENING`. The falling edge of condition `CLOSED` (`fs(CLOSED)`) on the self loop on state `OPENING` indicates that the barriers

have left their lower end position, which is reported to the overall controller by event BARRIER_OPENING. Once the barriers have reached their upper end position, indicated by condition OPENED, the barrier controller returns to its initial state.

The opening of the barriers has to occur within a certain time, the *maximum opening time* MOT, which typically is identical to the maximum closing time. In the case of a barrier failure the same behavior as for the closing case is observed. The error state can also be left again, if the barriers are opened successfully.

Statechart SENSOR_CONTROL_CTRL

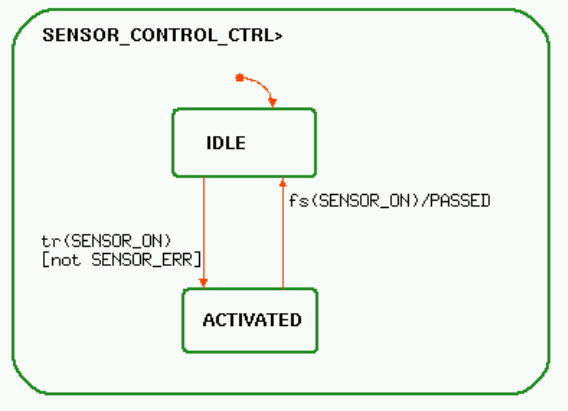


Figure 2.15: Statechart SENSOR_CONTROL_CTRL

The controller for the pass sensor shown in figure 2.15 waits for a train to activate the sensor, indicated by the rising edge of condition SENSOR_ON (tr(SENSOR_ON)). The sensor is not activated, if an error has been detected (condition SENSOR_ERR). Once the train leaves the sensor area, indicated by fs(SENSOR_ON), the passing of the train is reported via event PASSED to the crossing controller.
The static reaction of SENSOR_CONTROL_CTRL reports a failure of the sensor to the operations center via event SENSOR_DEFECT.

Chapter 3

Message Sequence Charts and Sequence Diagrams

In the development of computer systems visual languages are becoming increasingly popular due to their graphical appeal, as developers are more inclined to draw diagrams in order to design a system than to write cryptic lines of code. Especially the telecommunications domain has been using visual languages for many years. In this field the language of *Message Sequence Charts*, which captures information exchange in communication systems, has its origin. It has been adopted in other fields as well in order to specify message exchange between entities. Most notable is the inclusion of an object-oriented variant of Message Sequence Charts, called *Sequence Diagrams*, into the UML standard. In this chapter we will give an introduction to both sequence chart variants, starting with Message Sequence Charts in section 3.1 and followed by Sequence Diagrams in section 3.2, and highlight their inadequacies barring a more prominent role in the design process. These shortcomings have led to the definition of a variant which addresses and remedies these deficiencies: Live Sequence Charts, which are introduced in chapter 4.

3.1 Message Sequence Charts

Message Sequence Charts (MSCs) are standardized and maintained by the International Telecommunications Union (ITU). Prior to their standardization there existed several precursors, like Time Sequence Diagrams, Arrow

Diagrams, and many more. We will not discuss these variants here, but only deal with MSCs themselves. A comprehensive account of pre-MSC sequence charts can be found in [GRG93a] or [Ren99].

3.1.1 MSC-93

MSCs were first called *Extended Sequence Charts*, an auxiliary notation for SDL (Specification and Description Language, [IT88][1]), a graphical language for the specification of system structure and behavior (typically telecommunication systems), which is maintained by the ITU as well. It was realized that the Extended Sequence Charts were a valuable addition to the state-based view of SDL, which resulted in the adoption of Message Sequence Charts as ITU Recommendation (Z.120 [IT93]) in 1993.

In this first revision of the MSC standard, MSC-93, individual MSCs are collected in *MSC documents*, although no further relation is implied by grouping them together. Each MSC has two dimensions: the horizontal or structural one, where the different entities participating in the chart appear, and the vertical or temporal one, which indicates the passing of time from top to bottom of an MSC. Figure 3.1 shows the basic MSC features.

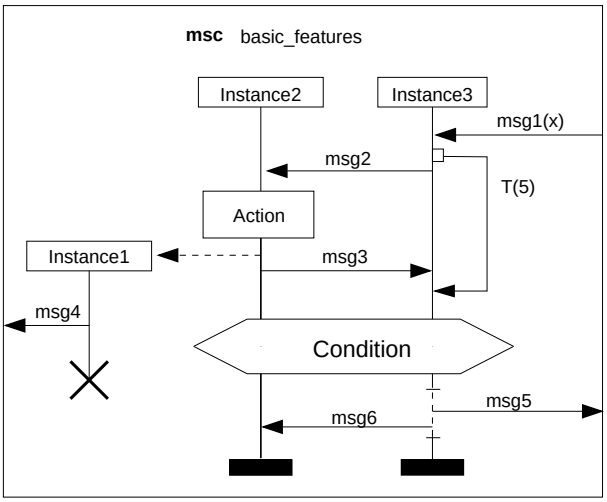


Figure 3.1: Basic MSC features

[1]By now there exist three newer versions of the SDL standard, the most recent one being SDL-2000 [IT00].

The entities involved in the communication are represented by vertical lines called *instances*, which consist of an instance head symbol (empty rectangle) carrying the instance name, the instance axis and an instance end symbol (filled rectangle); cf. figure 3.1. Due to the close relation between MSCs and SDL instances often represent entities of the SDL design, like processes, blocks, etc., although this is not required. On the instance axis the actions carried out by the entity represented by the instance appear; these actions are called *events* and can e.g. be messages, conditions or timers. The events of one instance axis are totally ordered from top to bottom unless they are contained in a coregion. An ordering between events on different instances is enforced only by messages; see below for more detailed descriptions of these concepts.

The environment is represented by the frame enclosing the MSC. Note that the instance head and end do *not* correspond to the creation and destruction of the model entity represented by the instance. They just mark the points in time where the corresponding design entities start, resp. stop to take part in the communication described in the MSC. There exist separate constructs for expressing creation and destruction of entities (see below).

Information exchange between entities is represented by messages, which are depicted by arrows; all communication is asynchronous. As shown in figure 3.1, messages can be exchanged between two instances or between an instance and the environment. Each message consists of two distinct events: a send event and a receive event. Messages are the only way to establish an ordering between events of two different instances, since obviously a message must be sent before it can be received. A message has a name and may also carry parameters, cf. `msg1` in figure 3.1. Only one message may be sent or received by each instance at one point in time.

Conditions are used to represent system states, which are described textually. They are depicted by hexagonal shapes and can refer, i.e. be attached, to one or more instances. Instance axes of instances, which are not participating in a condition, but which cross the condition symbol are drawn through the hexagon, whereas the axes of participating instances are suspended. In figure 3.1 e.g. the condition refers to both `Instance2` and `Instance3`. Conditions can be used to define possible continuations of MSCs: An MSC ending with a condition `connected` e.g. can be continued in another MSC, which starts with the same condition.

MSCs also allow to express time properties in the form of timers, with three possible actions:

1. setting a timer to a value
2. resetting the timer
3. observing a timeout

Timer set events are depicted by a small square attached to an instance and may be named; cf. `T` in figure 3.1. The duration of the timer can optionally be specified in parenthesis. A timer set event is always connected to either a timer reset or timeout event by means of a vertical line which emanates from the timer set square, runs in parallel to the instance axis and is connected to the instance axis once again by an arrow, which indicates the timeout or reset event. A timeout event is rendered as a solid arrow, a reset event by a dashed arrow. Figure 3.1 on page 46 shows an example for a timeout event.

Process creation is depicted by a dashed message arrow leading from the creating instance to the created one. Process termination is depicted by a large X terminating the instance line; cf. `Instance1` in figure 3.1 for an example. Local actions, i.e. activity internal to an entity, are depicted by a rectangle, which contains a textual description of the action.

The total order among the events on one instance may be suspended by a *coregion*. All events within a coregion are completely unordered, i.e. they may occur in any order, but *not* simultaneously. A coregion is depicted by a dashed segment of an instance axis. In figure 3.1 the send events of the messages `msg5` and `msg6` are contained in the coregion of `Instance3` and may therefore be sent in any order.

The only structuring mechanism offered by MSC-93 is the refinement of an instance into sub-entities, called *decomposition*. This is not depicted graphically, but just indicated by the key word *decomposed* followed by the name of the chart, which shows the refined structure and behavior, below the instance head.

Several different approaches to define a semantics for MSC-93 have been undertaken: automata-based [LL92b, LL92a, LL95], Petri-net-based [GRG93b] and by process algebra [MR94]. A short overview of these approaches can again be found in [GRG93a]. From these semantics the one based on process algebra was chosen to become the official semantics published by the ITU in [IT95].

3.1.2 MSC-96

Three years after approval of the first version, a new revision of MSCs was accepted by the ITU: MSC-96 [IT96b]. The major change from MSC-93 to MSC-96 was the introduction of a number of structuring mechanisms on two different levels. First it has to be noted that the new revision distinguishes *Basic MSCs (BMSCs)*, meaning individual charts (MSCs as defined in MSC-93, with a few extensions), from *High-level-MSCs (HMSCs)*, which describe the organization or interplay of several charts. Thus HMSCs constitute one level of structuring (inter-chart structure), the other level being within BMSCs (intra-chart structure) by use of inline expressions and sub-charts.

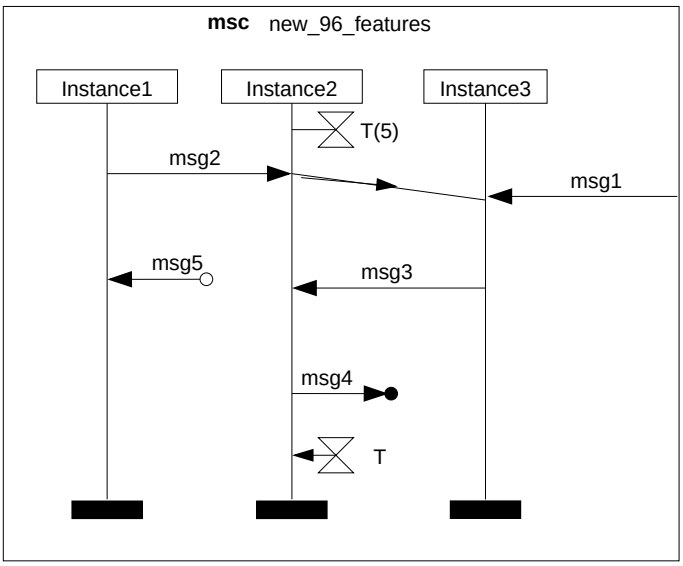


Figure 3.2: New MSC-96 features

For BMSCs a few non-structuring changes were introduced as well. There is a new type of message, incomplete messages, i.e. either the sender of the message may be omitted, resulting in a *found* message, or the receiver, resulting in a *lost* message. The graphical representation of lost and found messages is an arrow which ends or starts at a circle instead of an instance; `msg4` in figure 3.2 is an example of a lost, `msg5` of a found message.

MSC-96 also provides the possibility to impose an order on otherwise unordered events, either on separate instances or within a coregion. This *generalized ordering* is represented graphically by an arrow whose head is

not positioned at the end but in the middle. Figure 3.2 shows an example of ordering two otherwise unrelated message receive events on different instances: The receipt of `msg2` should precede the receipt of `msg1`.

The appearance of timer has been altered in the new standard as well: the old timer set symbol has been replaced by a more intuitive hour-glass symbol, the timer reset symbol by an X similar to the process termination symbol, but connected to the instance axis by a vertical line, and a timeout is now indicated by an hour-glass symbol which is connected via an arrow to the instance axis. Moreover, different timer events related to the same timer can now be decoupled from each other – i.e. they need no longer be connected by a vertical line, but are identified by the timer name – and may thus appear in different charts. A timer e.g. may be set in one chart and the corresponding timeout may be observed in another MSC. Figure 3.2 shows the setting of timer `T` and the respective timeout event.

Intra-chart Structure

The intra-chart category of structuring mechanisms offered by MSC-96 contains two new possibilities to structure Basic MSCs, in addition to the instance decomposition of MSC-93. The first of those are *inline expressions*, the second are *MSC references*. The former are represented graphically by a box, which encloses those instances and events, which take part in the inline expression. Inline expressions come in different flavors indicated by a key word in the upper left corner of the box:

loop: Describes the iteration of the communication sequence shown in the box. Both bounded and unbounded iterations are allowed; for bounded ones a lower and an upper bound may be specified. Figure 3.3 shows an example where `msg1` is send and received at least 2 and at most 7 times.

alt: Several alternative continuations of the chart can be specified. Which alternative is taken is determined by which event is observed first after entering the inline expression. In case of a common prefix among the alternatives, the common prefix is executed first and the first differing event determines the alternative. The different alternatives are separated by dashed horizontal lines; see figure 3.4 on page 52 for an example. Here there are three possible continuations after the sending

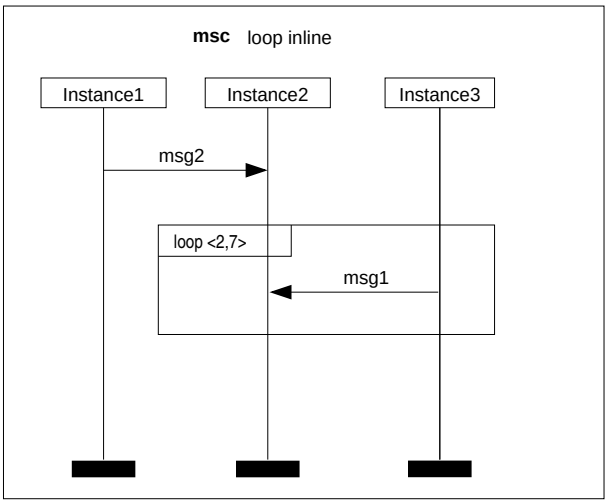


Figure 3.3: loop inline expression

and receipt of `msg1`: `Instance2` either sends `msg2` to `Instance1`, `msg3` to `Instance3` or `msg4` to `Instance3`.

opt: This inline expression describes optional behavior and has only one section (like the loop inline expression). It is a short-hand notation for an alternative inline expression with two alternatives, where the second one is empty.

par: The sections of this inline expression are executed in parallel. The graphical representation is identical to the alternative inline expression, except that the key word is `par` instead of `alt`.

exc: This inline expression describes an exception and contains only one section. Either the events of the inline expression are executed and the MSC is exited afterwards, or the inline expression is skipped and the rest of the MSC is executed normally. It is a short-hand notation for an alternative inline expression with two alternatives, where the first one is the exception and the second one the remainder of the MSC.

Each inline expression may be shared by any number of instances in the MSC, only the exception inline expression must cover all instances. Instances which are not sharing an inline expression may exchange messages with instances within by using *gates*.

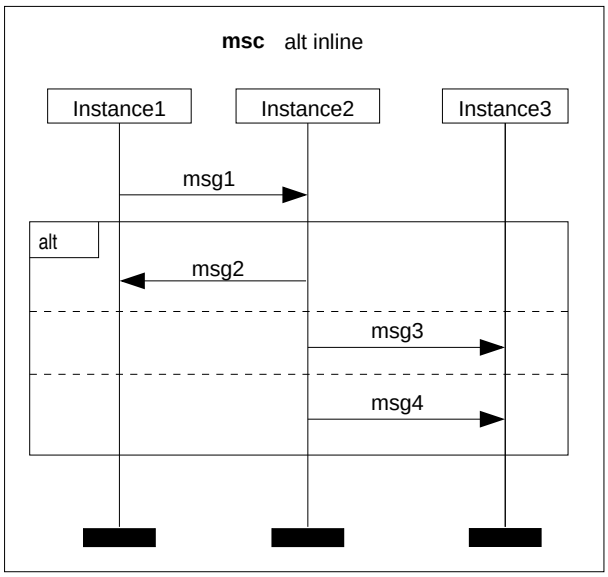


Figure 3.4: loop alternative expression

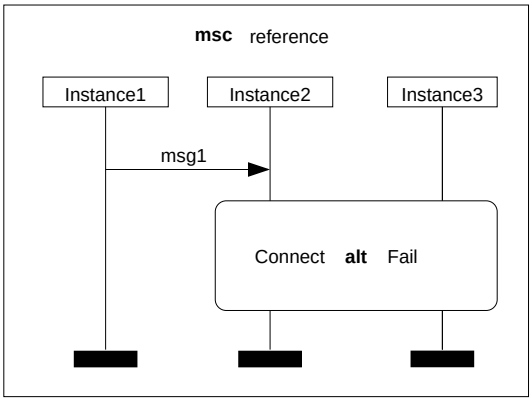


Figure 3.5: MSC reference example

The second intra-chart structuring element is a sub-chart construct, called *MSC reference* in MSC-96. A reference stands for other BMSCs or collections of other BMSCs. Several BMSCs may be part of a reference expression, which can be constructed using the same operators as for inline expressions plus one additional operator for sequential composition (*seq*). The difference between inline expressions and MSC references is that the latter are (a set of) BMSCs

themselves which can be reused several times, similar to procedure calls in programming languages. In order to make this concept more flexible it is also allowed to substitute elements in a reference (like instances or messages) when plugging an MSC reference into a chart, analogous to passing parameters to functions or procedures in programming languages. Instances outside the reference may communicate with instances within via gates.

Graphically references are represented as boxes with round corners, which contain the name(s) of the MSCs which are referenced. The composition of several MSCs is not expressed graphically but by using corresponding key words. Figure 3.5 shows an example of an MSC reference with two alternatives, either the first MSC `Connect` is entered or the second one (`Fail`).

Inter-chart Structure

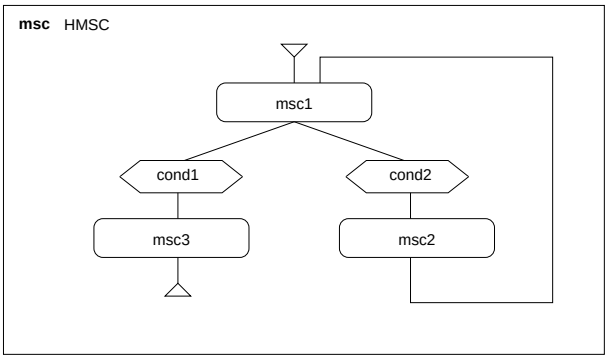


Figure 3.6: Highlevel MSC

On this level only one structuring mechanism is offered by MSC-96: *High-level MSCs (HMSCs)*, which allow to graphically define how MSCs may be combined, i.e. they allow to draw a graph which shows all possible execution orders for a number of MSCs. The nodes of this graph are either other HMSCs, conditions or MSC references, i.e. (collections of) BMSCs. The MSCs of the nodes can be combined by using the operators for MSC references. Conditions may be used in addition to references as nodes. Each HMSC must contain at least one start and one end node – represented by a downward-pointing and upward-pointing triangle, respectively. Graphically, sequential composition is depicted by connecting the nodes in question by a flow line. Alternatives are represented by several flow lines leading from one node to a set of nodes. Parallel composition is depicted by having two or more start and end symbols in a HMSC. Figure 3.6 shows an example where first MSC `msc1` is executed and afterwards, depending on the evaluation of the two conditions, either `msc2` or `msc3`. If the right alternative is taken, the initial MSC is entered again and so forth.

For MSC-96 an extensive description of the static semantics exists [IT96a],

but the definition of the dynamic semantics required more effort and time. It is a remodeled and extended variant of [IT95], i.e. it is again defined in terms of process algebra. Initially worked out in [Ren99], it was approved by the ITU as [IT98] in 1998. In the meantime between the publication of MSC-96 and the publication of its semantics, other suggestions for a semantics were made, based on Petri-Nets [Hey00] [KPE00], rewriting logic [Kos97], Duration Calculus [GDO98] and streams [Krü00]; there also exists an alternative process algebra characterization [GHRW98, GHN+98]. These works mostly do not cover the complete set of MSC-96 features, but concretize different open points of MSCs, like time and sequential composition.

3.1.3 MSC-2000

The latest revision of the MSC standard, MSC-2000 [IT99], was published at the end of 1999. While the major additions of MSC-96 were structuring mechanisms, MSC-2000 focuses on extending MSCs with object-oriented features and adding data. The MSC document containing individual MSCs has been extended to include several object-oriented features, like inheritance or virtual references, plus a number of declarations (messages, instances, ...) in the style of programming languages. These new organizational features are mostly non-graphical.

MSC-2000 also allows the specification of method calls and replies, similar to Sequence Diagrams, although they do not need to be synchronous (cf. section 3.2). The section of the instance which is carrying out the method call, is indicated by broadening the instance axis into a thin, filled rectangle, identical to the activation in SDs (see section 3.2). Moreover, the instance axis of the calling and waiting instance is marked by an empty rectangle. The method call is additionally distinguished from normal, asynchronous messages by the keyword `call`. Every method call must be matched by corresponding reply message represented by a dashed arrow; see figure 3.7 for an example. In addition to the normal, asynchronous messages of the former versions of the MSC standard MSC-2000 thus offers the possibility of synchronous and asynchronous method calls.

MSC-2000 allows the user to declare a data language to be included, so that variables of this data domain can be used in the MSCs. This is particularly useful in conjunction with conditions, where it is now possible to associate boolean expressions with a condition; they are called *guarding*

conditions[2]. But the standard also puts a restriction on the allowed usage of guarding conditions: They are associated with a scope, which can be either an inline expression or an entire MSC, and may only be used at the beginning of their respective scope. Guarding conditions are identified by the keyword `when`; cf. figure 3.7.

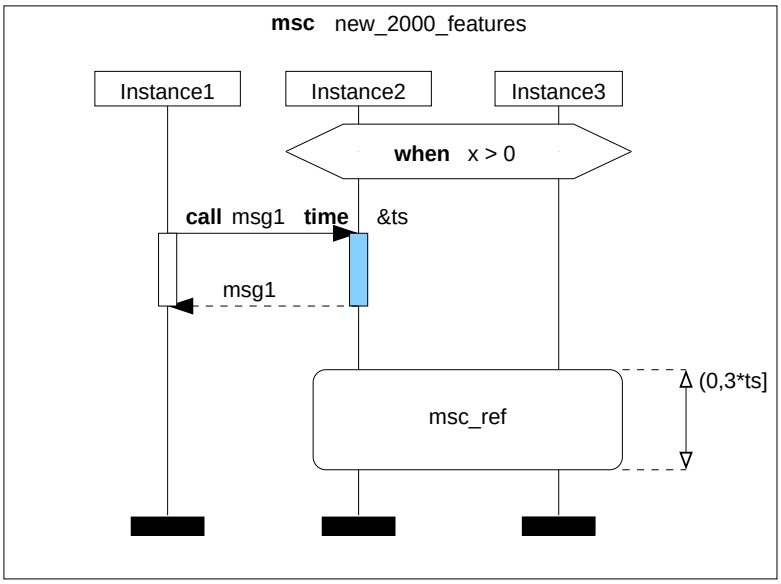


Figure 3.7: MSC-2000 features

MSC-2000 also offers more possibilities of specifying time constraints. Apart from timers, timing requirements can now be expressed by intervals as well. Such a time interval may connect any two events on one instance or constrain the execution of an MSC reference. Both timing intervals and timers can have upper and lower bounds and arbitrary time expressions are allowed as bounds. In addition, it is possible to reference a global clock by time stamps which can be referenced in later timing expressions. An interval is depicted graphically by an double-headed arrow whose end points are connected to the constrained events by a dashed line; the time constraint is given in interval notation. Time stamps are only represented textually by

[2]To be precise: conditions are called *guarding* if they restrict the further execution of the MSC. This restriction may be expressed by a simple label as for conditions in MSC-96 or by an expression of the data language. Thus all boolean conditions are guarding, but not all guarding conditions are boolean.

the keyword `time`. Figure 3.7 contains an example of a relative time stamp `&ts` which measures the time needed to execute the method call `msg1`. The execution time for the MSC reference `msc_ref` is then constrained to be at most three times as long as the execution time of the method call.
So far no formal semantics has been published for MSC-2000 by the ITU. A first attempt at defining part of the semantics has been undertaken in [JP01].

3.1.4 Shortcomings of MSCs

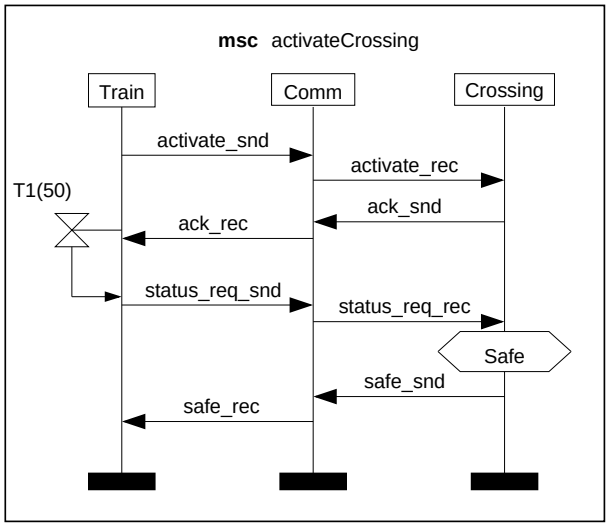


Figure 3.8: MSC for the securing procedure

At this point we would like to assess the language offered by MSCs with respect to their usefulness and suitability in the development process. The usage to which MSCs have been mostly put is as documentation, either in the early phases of system development as sample scenarios or to record test or simulation runs. But sequence charts have the potential for other, additional usages in the design process, like e.g. graphical property specification for formal verification, on which we will elaborate in chapter 10. This requires an unambiguous and formally based language, requirements which the MSC language is lacking for the most part.

We use the MSC in figure 3.8 in the following to illustrate the major points of criticism. This MSC shows the interaction between train and crossing for the securing of the crossing (cf. chapter 2). The train issues an activation

command, which is immediately acknowledged, and after waiting for 50 units of time, it requests the status of the crossing. If the crossing has been secured successfully, it answers by sending the corresponding message. All messages are exchanged via radio link represented by the communication component.

The formal semantics given by the ITU ([IT95, IT98]) defines the semantics only in terms of allowed sequences of events, without indicating, if the behavior specified by an MSC is mandatory or not. The major points of criticism are detailed in the following:

1. An MSC shows only one sample run of the system, one scenario. The MSC in figure 3.8 thus expresses that it is possible for a train to order a crossing to be secured. But the intended meaning of this MSC is not to specify a possible, sample behavior, but rather to express a mandatory protocol between train and crossing: *Every* train should contact *every* crossing along its way and secure it, i.e. every system run should conform to the behavior specified in the MSC.

2. An MSC does not state explicitly *when* the behavior it describes should be observed, i.e. there is no indication of when the MSC should be activated. People familiar with the application may know that the train should activate the crossing once it reaches the corresponding activation point, but the MSC does not reflect this hidden knowledge. It would e.g. allow the train to activate *all* crossings along its way at the start of its trip or even when it has already passed the crossing.

   The guarding data conditions of MSC-2000 can be used to specify the activation point of an MSC by placing them at the beginning of an MSC and expressing the system state characterizing the activation point by the boolean expression, although sequences of messages, which activates a chart, can not be specified.

3. The MSC semantics offers no distinction, whether progress is enforced or not. The intention of the MSC in figure 3.8 e.g. is that the crossing *does* send the safe message, if it has been secured. Thus progress along the instance line should be enforced, but the semantics in [IT95] only define permitted sequences of events; the occurrence of an event can not be enforced. Likewise, we do expect all messages, which are sent in our example MSC, to arrive at their destination. This too can not be expressed according to the MSC semantics. Using the terms coined

by Lamport [Lam77] MSCs can only express *safety* (nothing bad ever happens), but not *liveness* properties (something good will happen eventually).

Bounded liveness is theoretically expressible by using timers, but since timer durations are not covered by the semantics (see below), this is only true on an informal level.

4. MSCs do not allow more than one event to happen at the same time. In the example MSC we would like to express that 50 time units pass between the reception of the acknowledgment and the sending of the status request, i.e. we want the timer to start counting when the acknowledgment arrives and the timeout to occur simultaneously with the sending of the status request. Likewise, the condition expressing the safe state of the crossing should be evaluated when the status request arrives at the crossing, i.e. the crossing is required to be in the secured state at this point. Neither can be expressed in MSCs.

   Simultaneity is now available for timing annotations as the example in figure 3.7 shows, but it is still not possible to enforce e.g. simultaneousness of a message and a condition or of several messages.

5. Conditions in MSCs have no formal semantics, but are either glue points, which indicate possible concatenations of MSCs, or "guard" alternatives in HMSCs. In the words of [IT95]: "*The semantics of a chart containing conditions is simply the semantics of the chart with the conditions deleted from it.*". This is obviously not the way to treat conditions from a more formal point of view. In MSC-2000 conditions have been upgraded to some degree due to the introduction of data, which allow boolean conditions. But these guarding conditions do not solve the problem completely, since they are restricted to be placed at the beginning of their scope. It is thus not possible to have boolean conditions in arbitrary locations of a chart, so the extension for conditions as offered by MSC-2000 is not completely satisfactory. Even more so, since no semantics for the latest revision of the standard is available so far.

   The missing semantics also entail that the meaning of a violated condition is unclear: is this an error or not? In the example MSC the

meaning of the condition is: *If* the crossing is safe, then the corresponding message must be sent. This means that the satisfaction of the condition is not required; a violation should therefore not result in an error.

6. The treatment of time is only rudimentary, since quantitative timing is not covered by the semantics, i.e. timer durations are ignored. Only the correct sequence of timer events, resp. intervals is enforced.

3.2 Sequence Diagrams

Sequence Diagrams (SDs) are one of the many diagrams offered by the Unified Modeling Language (UML) [OMG01]. We will not give a detailed introduction to the complete UML here, but rather focus on Sequence Diagrams. For a comprehensive description of the UML and all its diagrams see [OMG01] or [JBR99].

The features of SDs are mostly derived from BMSCs, i.e. there are no operators for organizing collections of SDs similar to HMSCs, which may change in the newest version of the UML (2.0) currently under development. SDs are usually viewed in isolation as being possible scenarios of the system being developed. SDs may be associated with Use Case Diagrams, which provide an abstract view of the general functionality of a system. In this case SDs are examples and concretizations of the functionality depicted in the use case.

The basic SD features have been adopted from MSC-93, although the terminology is a different one. Instances are called *object life lines* as they represent objects (or actors) in the UML world; messages are called *stimuli* and may be synchronous (*operations*) or asynchronous (*signals*). Operations correspond largely to the MSC-2000 feature of method calls, although in MSCs method calls need not be synchronous, whereas in SDs they always are. The reply message can be drawn explicitly, but is not required. Optionally SDs allow to express the flow of control through the life lines: the part of the life line where the corresponding object is active, i.e. computing, is indicated by broadening the life line into a narrow rectangle called *activation*. This feature has been partially adopted for method calls in MSC-2000, but in MSCs only the segment of the instance which is executing body of the method is indicated by a rectangle. SDs allow the use of activation also outside of

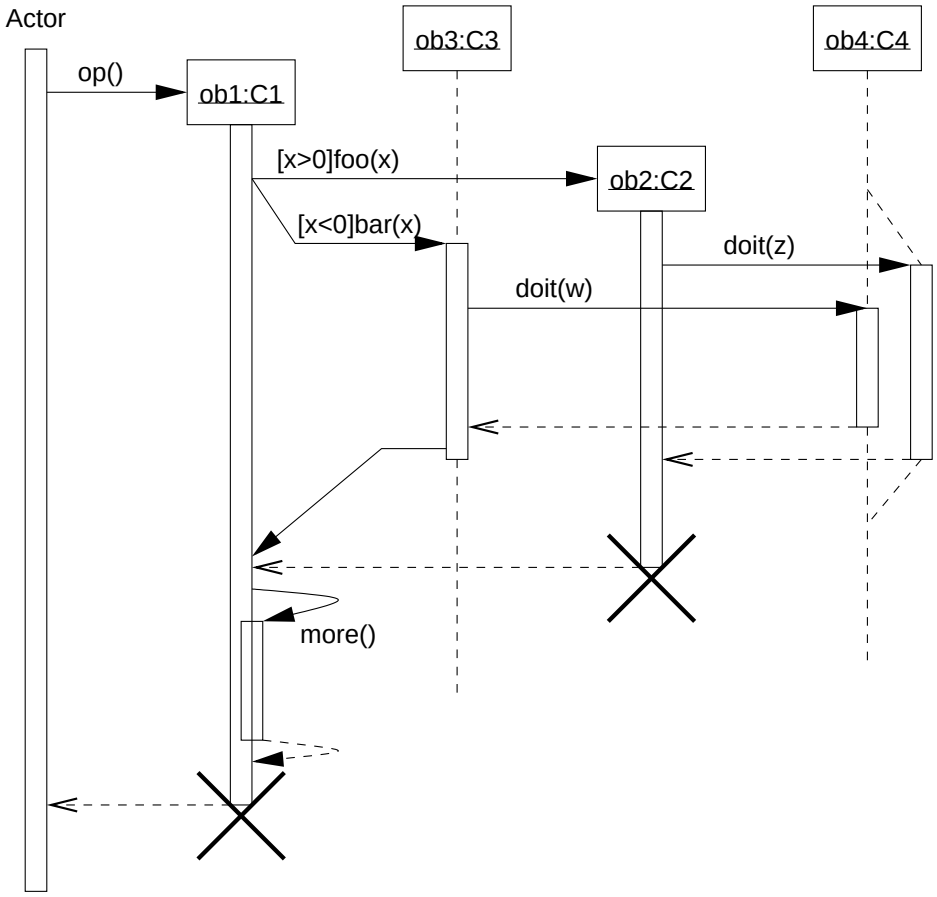


Figure 3.9: Sequence Diagram example (taken from [OMG01])

method bodies, i.e. here the active part(s) of objects in an SD can be highlighted in a more general fashion (cf. figure 3.9, which has been taken from [OMG01]) where objects `ob1` and `ob2` are shown as active, although they are currently not servicing operation calls).

Graphically life lines are depicted by vertical dashed lines or thin rectangles, if the activation is shown. Operations are represented by solid arrows with a filled arrow head, return messages by dashed arrows with a stick arrow head. Signals are depicted by solid arrows with stick or half stick arrow heads. As seen in figure 3.9, creation operations are represented by normal operation arrows in contrast to MSCs, where creation messages are depicted by dashed arrows. Object destruction is represented by a large X, just as

process termination in MSCs. Figure 3.9 also shows that stimuli may be guarded, so that it is possible to specify alternatives. This corresponds to an alternative inline expression or a reference in MSCs. A duplication of life lines is offered to express alternatives or concurrent activities (e.g. `ob4` in figure 3.9).

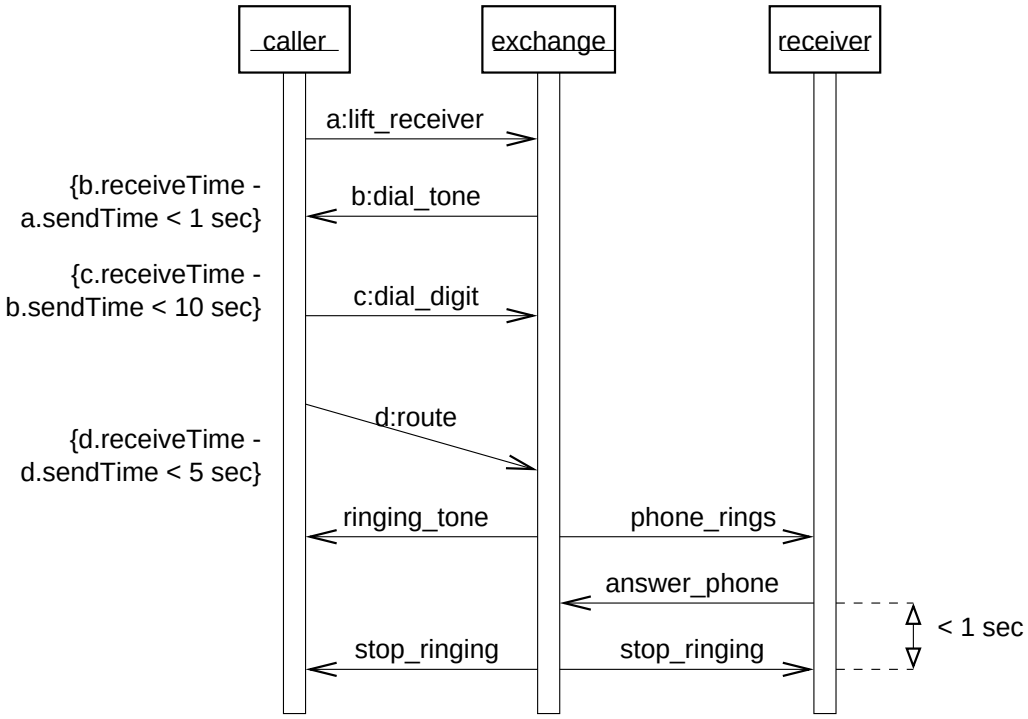


Figure 3.10: Sequence Diagram example with time (taken from [OMG01])

SDs also offer timing constraints, which may be stated either graphically or textually. The graphical representation is similar to the time intervals of MSC-2000, except that the annotation is not necessarily in interval format, but may be expressed textually as well, see figure 3.10 (also taken from [OMG01]). In order to use the textual representation it is necessary to tag the stimuli with a symbolic name, by which they can be referenced in the constraint. Specific time points of a stimulus can be defined and referenced, e.g. `a.sendTime` in figure 3.10. The actual constraint is written in curly braces and can be expressed in any language, the one predefined by the UML standard for this purpose is the *Object Constraint Language* (OCL), see [OMG01].

The UML standard defines a semantics for SDs in terms of the UML meta model, which eventually is defined in terms of itself, but until now no official *formal* semantics for either the complete UML or SDs is available, although several propositions have been made for different parts of the UML, some of which are shortly presented in the following. Several works have taken a meta-modeling approach, trying to define a common domain in which the different diagram types of the UML can be mapped: see e.g. works by the *precise UML group*[3] [EFLR99, EBF+98, EK99] and [GR99]. [RACH00] present an algebraic semantics for class and state diagrams, [LP99] define an operational semantics for state diagrams, including consistency checks by model checking, [LMM99a, LMM99b] follow a very similar approach. [FHD+99] translate SDs into timed automata in order to check their timing consistency, but do not consider the complete set of features. [DJPV03] define a semantics for a kernel UML language, which covers class and state diagrams and also takes the action language into account.

Shortcomings of Sequence Diagrams

The points of criticism raised for MSCs in section 3.1.4 hold as well for SDs, as they are existential in nature, make no statement about the activation and do not allow the distinction whether progress is enforced or not. Simultaneousness is expressible for timing annotations by the interval notation or by the timing labels. Conditions in the sense of MSCs do not exist in SDs, the only construct coming close are the guard expressions on messages (cf. figure 3.9). These allow to express simultaneousness, but cover only conditions on the sender's side. Simultaneous sending and/or receipt of several messages is not discussed in [OMG01]. The major drawback of SDs, however, is the absence of a formal semantics.

So in conclusion we can say that neither MSCs nor SDs provide the expressiveness and formal rigor, which is needed for sequence charts to be used for more advanced use cases. This motivates the introduction of a sequence chart dialect which remedies these shortcomings: Live Sequence Charts.

[3] www.cs.york.ac.uk/puml

Chapter 4

Live Sequence Charts: The Kernel Language

This chapter presents the key elements of the Live Sequence Chart (LSC) language. The shortcomings of the MSCs and SDs, which have been detailed in the previous chapter, motivated Werner Damm and David Harel to propose an improved sequence chart dialect, which resolves these problems: LSCs. The first ideas appeared as [DH98] in 1998 (an abridged version was published as [DH99]); a newer revised edition was published in 2001 as [DH01]. The base for the present work is the core feature set as put down in [DH98], which is still incomplete regarding the expressivity of the envisioned language. Some features like real-time are not treated in detail, others like simultaneous regions, local invariants or pre-charts have been omitted completely. The need for these constructs has quickly become apparent in our work resulting in this extension of the feature set given in [DH98]. Some of these novel features, like simultaneous regions and pre-charts, have already been adopted by Damm and Harel in the latest version of the paper [DH01].

The general guideline for the definition of the LSC language was to raise the expressiveness of sequence charts to the level required for the application as a formal specification technique, but still retain the intuitiveness and visual appeal of MSCs and SDs.

In this chapter we will first present the basic graphical elements, like instances, messages, etc., of which an LSC is comprised, with the terminology being largely derived from MSCs. In the second part we will go beyond the existential view and also introduce a first possibility to characterize the activation point for a chart.
The semantics of the elements described in this chapter follow in chapter 6.

4.1 Basic LSC Features

The basic idea of LSCs is to allow a distinction between *mandatory* and *possible* behavior, i.e. most LSC elements can be designated to belong to either one category or the other. This distinction is also expressed graphically, which contributes largely to the easy understanding of LSC specifications. Mandatory elements are depicted by solid lines, possible ones by dashed lines. The basic features described in this section are:

- *instances*, which represent the participants for the communication,
- *messages* sent between the instances,
- *temperatures* indicating progress along instances and messages,
- *conditions* describing particular states of the instances at a certain point in time,
- *local invariants*, which are conditions which hold for a period of time
- *coregions* and *simultaneous regions*, which express ordering information on instances.

4.1.1 Instances and Messages

Instances and messages are the elementary building blocks of LSCs. The graphical representation for instances has been adopted from MSCs, i.e. LSC instances consist of an instance head carrying the instance name, an instance axis and an instance end, as the example LSC in figure 4.1 shows. As for MSCs and SDs, the horizontal dimension is the structural dimension and the vertical dimension corresponds to the time dimension.

Deviating from MSCs we depict the environment by an instance of its own rather than the border of the LSC. When using LSCs for formal verification, an explicit environment instance offers the possibility of expressing assumptions on the behavior of the environment within the LSC by employing the same elements as for the other instances (see chapter 8 for details). Graphically, environment instances are denoted by a shaded instance head symbol; see figure 4.1.

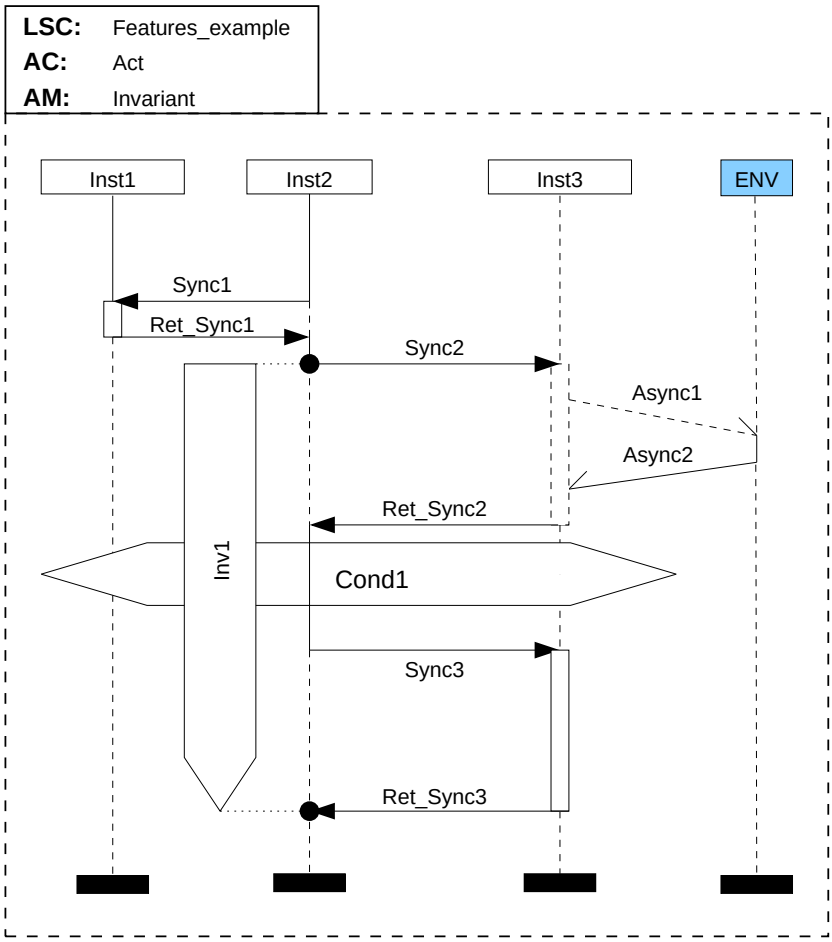
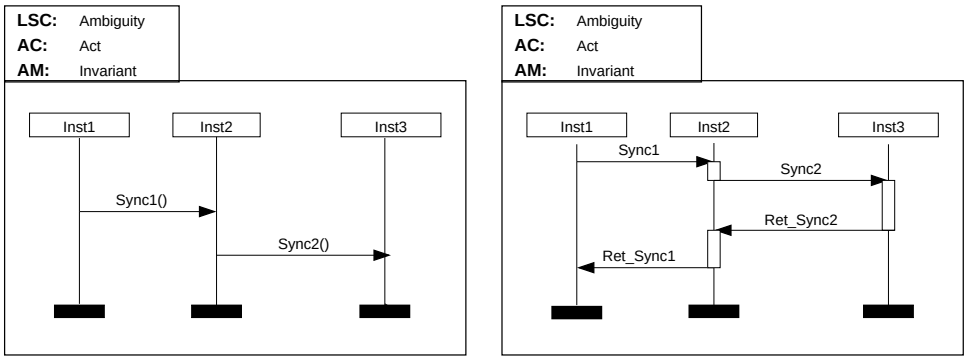


Figure 4.1: Kernel LSC example

Message Types

Concerning messages we consider two kinds: asynchronous and instantaneous ones. This is a deviation from [DH01], which distinguishes between synchronous and asynchronous communication. The characteristic of synchronous communication is that the sender blocks until the receiver is ready to receive the message. Once the receiver is willing to accept the message, the communication is carried out. What is *observable* in this case thus is simply that the information exchange takes place, i.e. the message is observed, sending and receiving are simultaneous. In order to clearly distinguish these

concepts, we rather use the term instantaneous than synchronous. Since the time dimension in LSCs is the vertical one, a delay on the senders side between trying to send a message and acceptance of this request by the receiver should be expressed on the instance axis as a progress requirement. The same line of thought has been used in the context of Message Flow Graphs by Ladkin and Leue [LL95].



(a) ambiguous LSC (b) possible resolution

Figure 4.2: Ambiguity example

Operation calls, e.g. in UML or method calls in MSC-2000, are thus represented by two instantaneous messages: the method call and the return message. For method calls in LSCs we require the return message to be included explicitly, because otherwise confusion arises whether messages and other elements following a method call receipt are part of the method body. Consider the LSC in figure 4.2(a). It is not immediately clear whether operation call `Sync2` should be considered as taking place after the completion of the preceeding operation call or if it is part of the body of operation `Sync1`. Requiring the user to explicitly include the return message clarifies the situation. Figure 4.2(b) shows how the second interpretation would look like. Consequently we require to explicitly include the return message for each operation invocation.
[DH01] does not explicitly address the question of method calls or returns.

For the graphical representation of messages we use an SD-like notation: asynchronous messages are visualized by half stick arrows, instantaneous messages by arrows with solid heads; see figure 4.1. Instantaneous messages

have to be drawn horizontally to indicate simultaneity of sending and receiving, while asynchronous ones are drawn slanted to indicate the passage of time between sending and receipt. Note that both types of messages consist of a sending and receiving event. The pairing of operation call and return is indicated graphically by widening the instance axis on the receiver side into a thin rectangle which marks the operation body, as for method calls in MSC-2000; see figures 4.1 and 4.2(b) for examples. A graphical representation of activations, i.e. explicitly showing the flow of control by enlarging the corresponding instance axis segments, as done in SDs is not provided, since this is not the focus of LSCs.

Progress

One deficiency of both MSCs and SDs is their inability to enforce progress, as mentioned in section 3.1.4. LSCs overcome this drawback by associating a *temperature* with both locations[1] and messages. The temperature can be either *hot* or *cold*, the former indicating that progress is enforced. The analogy here is that one cannot remain at a hot location for an infinite amount of time, because then one would burn ones feet. This obviously requires that a hot location has to be left, i.e. the following location has to be reached. At a cold location one can stay forever without harming ones feet, i.e. the following location need not be reached. In terms of messages this means that a hot message has to be delivered, whereas a cold message may be lost along the way. Progress information is thus expressed by the temperatures of the messages and along the instance lines. The meaning of the temperatures is summarized in table 4.1.

	hot	cold
location	location has to be left i.e. next locations has to be reached	may stay infinitely long at location
message	message has to be received once sent	message may be lost

Table 4.1: Temperatures for messages and locations

[1]Locations are those points on an instance axis, where some event is attached, e.g. sending or receipt of a message, conditions, etc. In chapter 6 we give a formal definition of a location.

Graphically hot temperatures are represented by solid lines, cold ones by dashed lines. This means that e.g. a hot location is depicted by a solid instance axis segment, which starts at this location and ends either at the next cold location or the instance end. In figure 4.1 for example the location of the sending of message `Sync2` is cold and the next location (the receipt of the return message) is hot, so that the instance axis segment in between them is dashed. Note that also the rectangle representing operation bodies is rendered either solid or dashed depending on the location temperatures. Analogously are messages depicted by solid or dashed arrows depending on their temperature; see figure 4.1 for examples. Note that the completion of operation calls is indicated by location temperatures, not by the message temperature. The request for operation `Sync2` in figure 4.1 thus must be received, but the operation need not be completed.

It is recommended that only those locations are annotated with a hot temperature, whose instance is responsible for achieving the progress. The receiving instance of a method call e.g. is responsible for generating a corresponding return message, so only this location is assigned a hot temperature, not the sending one. Conforming to this convention yields LSCs, which show a focus of control, similar to the activation of object life lines in SDs.

We will gradually transform the MSC of figure 3.8 on page 57, which we used to illustrate the shortcomings of MSCs and SDs in chapter 3, into an LSC as we introduce the LSC features. Figure 4.3 shows the first step of this transformation with temperatures added for locations and messages. Note that all final location temperatures are cold, because once all events on an instance axis have been observed, no progress need be enforced. All messages are instantaneous, since we are considering a STATEMATE model, where no asynchronous communication is possible.

4.1.2 Conditions

In order to make statements about the state of the system boolean conditions referring to attributes or data items of the involved entities are used. Note that we are using symbolic names instead of concrete expressions in our LSC examples, since the unwinding algorithm presented in chapter 6 operates on propositions rather than concrete model elements. Graphically, conditions are represented as in MSCs by an elongated hexagon (see figure 4.1 or figure 4.3).

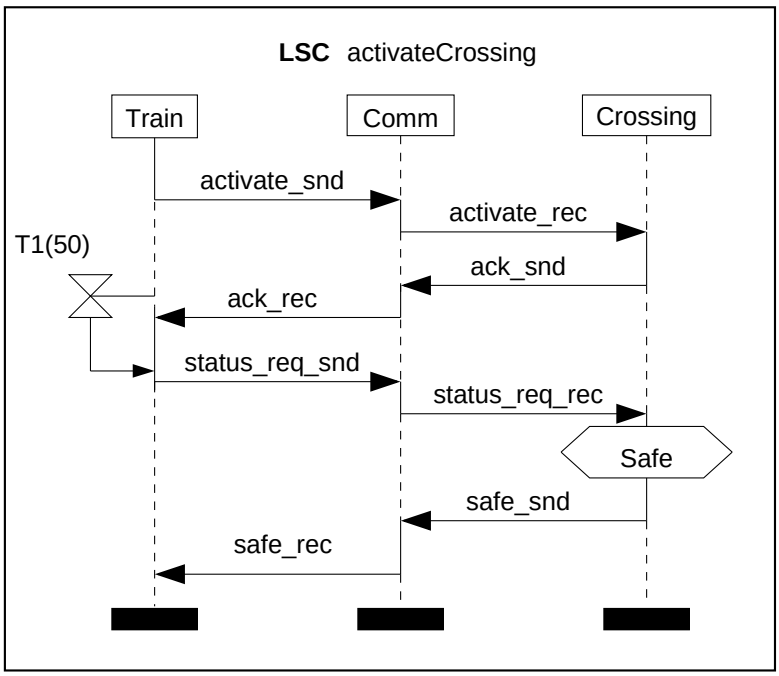


Figure 4.3: Crossing activation LSC, first step

Conditions also come in two variants: mandatory and possible. This is a second enhancement compared to MSCs in addition to making all conditions boolean, which is aimed at answering the question raised in chapter 3.1.4 as one of the points of criticism: What is the meaning of a condition, which is not satisfied? A violated condition could either be an error or it could mean that the considered run, which violates the condition, is not a relevant one for this scenario, so that the violation and the remainder of the chart should be disregarded.

LSCs allow both answers. A mandatory condition *must* be satisfied, i.e. the boolean expression associated with it has to hold; violation of the condition is an error. Possible conditions do not generate an error when they are not satisfied, but merely constitute an exit from the enclosing LSC.

Mandatory conditions are denoted by solid lines (e.g. `Cond1`) and possible ones by dashed lines. Note that we deviate here from [DH01] inasmuch as we do not use temperatures to distinguish mandatory and possible conditions. Temperatures – as they are used for messages and locations – express liveness. The mode of a condition on the other hand indicates the consequence of a

violation of that condition, but this does not entail liveness, i.e. a mandatory condition does not force us to progress in the LSC. In order to have a cleaner separation of concerns we use the term *condition mode* in this thesis. Note that conditions involving more than one instance specify a synchronization barrier, since all participating instances have to be ready to evaluate the condition simultaneously.

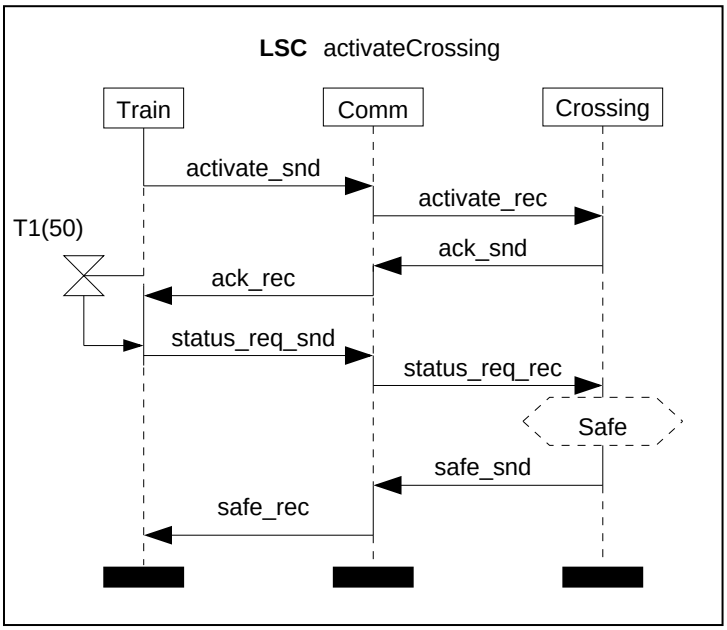


Figure 4.4: Crossing activation LSC, second step

Figure 4.4 shows the second transformation step for the example MSC. As noted in the criticism section (section 3.1.4), the statement in this example is that if the crossing is safe, the remainder of the chart, transmission of the corresponding messages, has to be observed. Thus the `Safe` condition has mode possible.

4.1.3 Local Invariants

Conditions constrain attributes or data items of entities at one point in time, but often it is desired to express validity of a condition over a period of time. In the example in figure 4.4 for instance, an alternative and more restrictive

specification of the condition might require the crossing to remain safe until the `safe` messages has been transmitted to the train. This observation motivates the introduction of a fitting feature: *local invariants*, which come in two flavors: possible and mandatory, with the same interpretation as for conditions. They are not part of the LSC language definition of [DH01], where only normal conditions are covered.

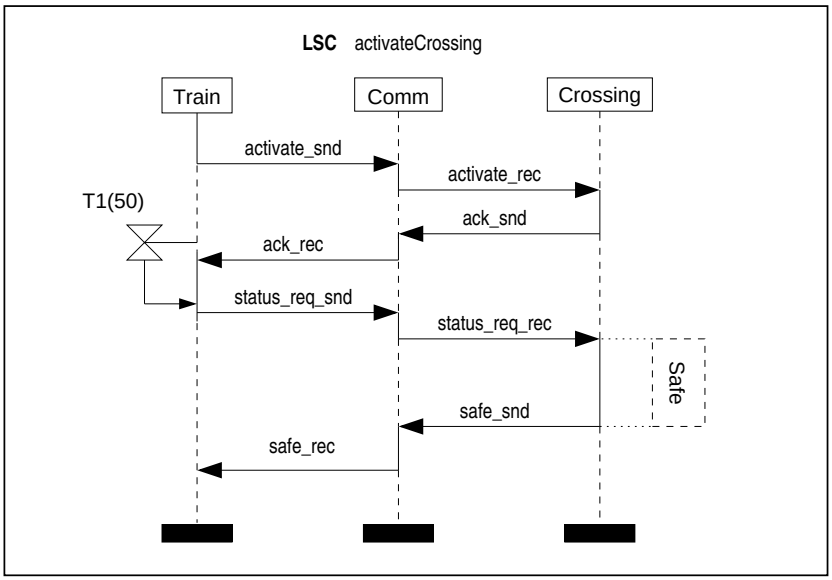


Figure 4.5: Crossing activation LSC, third step

Since local invariants cover a period of time, they need reference points for start and end. They should thus always be bound to observable events, i.e. message sending or receipt, or timeouts or timing intervals (cf. chapter 7). For each reference point the issue of inclusion needs to be resolved: Should the scope of a local invariant include its reference points or not? Both alternatives, inclusion and exclusion of the reference point, are needed as will become apparent in the examples in later chapters, therefore for each reference point both alternatives can be specified.

Local invariants starting at an instance head can only be exclusive, since activation is separated from the LSC body, which is facilitated the definition of the activation point of the LSC (cf. section 6.3). Inclusiveness, if desired, can thus be expressed by adding the concerned local invariants to the activation condition.

Graphically, local invariants are depicted by a condition symbol, which is rotated by 90°. Inclusion of a reference point is indicated by making the corresponding end of the condition symbol planar. Figure 4.5 shows that condition `Safe` has been extended to a local invariant, where both start and end reference points are included in the scope. Figure 10.5 on page 233 shows an example for a local invariant starting at the instance head.

Local invariants are expressible neither in classical MSCs nor in SDs nor in [DH01], though in UML there exists the possibility of specifying such invariants textually by OCL expressions. These are not part of the SD, however, rather than being constraints on the class, i.e. they always affect all objects of a class under all circumstances given in the OCL expression and are not local to an object of a particular SD.

4.1.4 Simultaneous Regions and Coregions

We now have assembled all basic elements of our kernel LSC language. The default ordering of these basic elements is one after the other from top to bottom along the instance axis. Ordering between instances is induced only by messages and conditions ranging over more than one instance. *Simultaneous regions* allow to group several elements, which should be observed at the same time. This is essential for determining reference points for conditions, local invariants and timer. This feature is a true addition compared to classical MSCs and SDs, where simultaneousness is forbidden or not considered. Neither is simultaneity considered in the first version of LSCs as introduced in [DH98]. The newest version of this paper, [DH01], however, follows our suggestion and contains this important feature. Simultaneous regions remedy the fourth point of criticism of chapter 3.1.4. Graphically they are represented by enlarging the location in question into a small filled circle; see figure 4.6 or 4.1 on page 67 for an example.

Figure 4.6 shows the MSC example with simultaneous regions in the desired places. Both the timer events, which will be discussed in detail in chapter 7, and the local invariant events are connected to their respective reference points, so that the LSC now expresses the intended behavior — apart from activation and universal interpretation. Note that in simultaneous regions defined on instance heads, only exclusive local invariant starts are permitted, since all other elements are part of the activation condition.

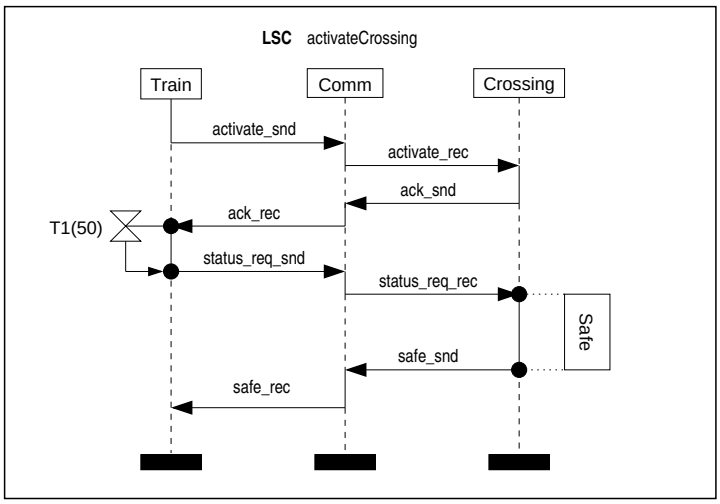


Figure 4.6: Crossing activation LSC, fourth step

A coregion is used to indicate that no ordering is imposed on the events it contains, i.e. they may occur in any order. This corresponds to the classical MSC view of a coregion with the exception that – as a consequence of the simultaneous region construct – we also allow events in a coregion to take place simultaneously.

Coregions are represented graphically by a dotted line running in parallel to the instance axis. This differs from the representation in MSCs, where they are depicted as dashed portions of the instance axis. This visualization clashes with the one for cold locations, therefore this alternative representation has been chosen. Figure 4.7 on the following page shows a coregion example as well as further examples for simultaneous regions.

4.2 Activation and Quantification

In the preceding section the graphical elements describing the communication behavior of several interacting entities are presented. This section adds information about when this behavior should be observed and whether it specifies a sample behavior or a protocol to be obeyed, addressing items 2 and 1, respectively of the points of criticism raised in chapter 3.1.4.

The quantification information represents the distinction between mandatory and possible behavior on the chart level. The sample-run or scenario

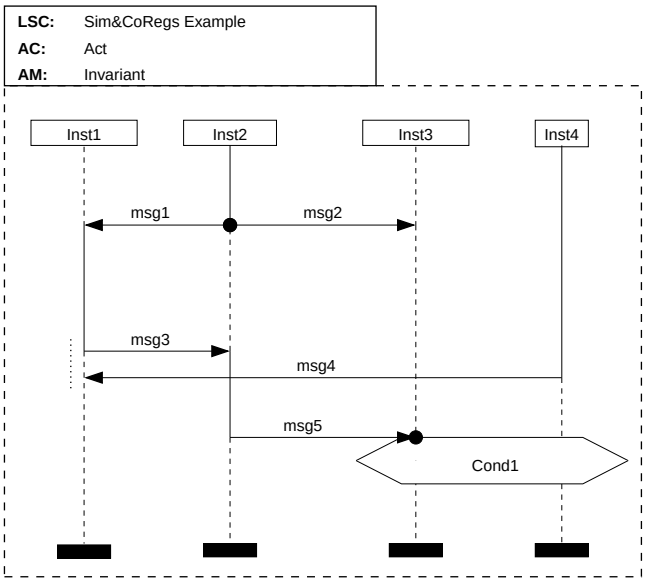


Figure 4.7: Simultaneous and coregion example

view of MSCs and SDs, i.e. the interpretation that there *exists* a run, which fulfills the LSC, is covered by the possible mode, which we call *existential*. The mandatory mode, which is missing in MSCs and SDs as laid out in section 3.1.4 on page 57, expresses that the behavior specified in the LSC must be fulfilled by *all* runs, for which reason it is called the *universal* view. Graphically, the quantification information is depicted by the border style of the LSC: a solid border indicates a universal chart, a dashed border an existential one.

For universal LSCs it is vital to be able to characterize the activation point. If every run has to fulfill the universal LSC, it must be possible to state at which point(s) of the run the LSC should be considered, otherwise the behavior of the entire system has to be specified in one LSC, which is clearly undesirable. The activation point of an LSC is characterized by two complementary concepts: *activation condition* and *activation mode*. The activation condition is a boolean condition, which expresses the activation point for a chart. The activation mode specifies, how often an LSC should be activated and is a new addition compared to [DH98, DH01] and has been adopted from Symbolic Timing Diagrams (STDs) [Sch00]. The offered modes

(a) initial

(b) invariant

(c) iterative

Figure 4.8: Activation Modes

are *initial, invariant* and *iterative*[2]. The initial mode indicates that the LSC is activated at system start only, i.e. it is intended to describe a start-up or initialization sequence.

Figure 4.8(a) illustrates the initial mode; the boxes represent possible activations of the LSC, with black boxes indicating *incarnations*, i.e. activated instantiations, of one LSC, which are actually activated according to the activation mode and grey ones indicating incarnations which are not activated according to the activation mode. The figure shows that an initial LSC is activated only once, at system start and other possible activations are disregarded.

The other two modes indicate that the LSC is activated whenever the activation condition holds. The difference between an invariant and an iter-

[2]Note that STDs do not support the iterative mode.

ative LSC is that the first one allows a reactivation of the LSC while another incarnation of the same chart is still active. i.e. an invariant LSC allows the activation of another incarnation, if the activation condition of a chart holds again, while the first incarnation is still active. Iterative LSCs allow only one incarnation of the chart at a time, i.e. if the activation condition does hold again while the first incarnation is active, no new one will be incarnated.

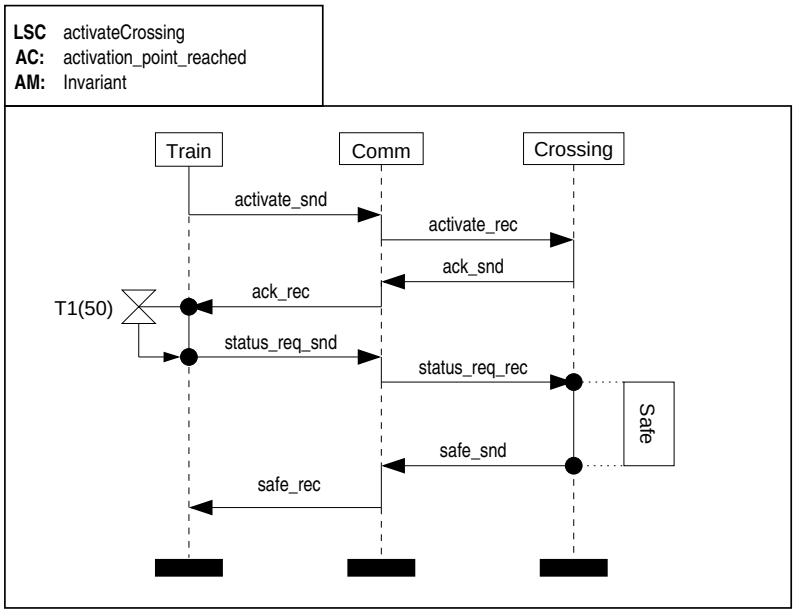


Figure 4.9: Crossing activation LSC, last step

This distinction is best illustrated by figures 4.8(b) and 4.8(c). Note that both modes also allow the LSC to be activated at system start, provided the activation condition holds in the initial state. The invariant mode allows the overlapping activation of several incarnations of the same LSC (two in the example in figure 4.8(b)). This is prohibited by the iterative mode, which is illustrated in figure 4.8(c). Here the overlapping activation of the LSC is shadowed by the previous incarnation and therefore disregarded as indicated by the grey third rectangle in figure 4.8(c). This entails that the use of the iterative mode should be exercised with caution, since forcing an LSC to be iterative when the model in reality allows the reactivation may shadow some incarnations. For model checking this means in the worst case, that the shadowed incarnation detects a violation of the model, but this

counter example is never found, because this incarnation is shadowed by the previous incarnation, which is not violated. The use of iterative LSCs for property specification for formal verification is thus prohibited and will not be considered in our application example. For other use cases, e.g. testing, however, the iterative mode does make sense.

Activation mode and condition are not depicted graphically, but are expressed textually at the top of an LSC together with the name, using the acronyms `AM` and `AC` respectively. For examples for activation mode, condition and quantification see figures 4.1, 4.7 and 4.9. Figure 4.9 shows the final result of the stepwise transformation of the MSC of figure 3.8 into an LSC.

Chapter 5

Automata-Theoretic Foundation

The base for the definition of the formal semantics of LSCs is a variation of timed Büchi automata. We chose Büchi automata for several reasons: First, we use LSCs to describe the communication behavior of reactive systems, i.e. systems which must be able to accept and react to input signals at any time. These systems are typically designed to operate forever, at least theoretically, which means that runs of reactive systems are infinite. Therefore classical finite automata are not sufficient for our purpose. Büchi automata ([Tho90]) are one possible solution as they accept infinite words. The second reason for choosing Büchi automata over other variants of automata on infinite words is that there is a close relationship between Büchi automata and linear time temporal logic (LTL) [Tho90, Eme90]. Since the main application field for LSCs in this thesis is property specification for formal verification, this is a major advantage. The third reason is that Büchi automata allow to easily express liveness properties via their acceptance criterion.

This line of reasoning has also been employed in the definition of the formal semantics of Symbolic Timing Diagrams (STDs) by Schlör in [Sch00], where the semantics of STDs are given in terms of a Büchi automaton variant. This variant, called *Symbolic Automaton* in [Sch00], extends classical Büchi automata by allowing expressions over an assertion language as transition labels. Since this extension is a vital addition for our semantics definition of LSCs, as will become apparent later, we are using this type of automaton as well and further extend it to be able to cope also with *quantitative* timing, which is not considered in [Sch00].

Section 5.1 starts with a short introduction to classical Büchi automata, followed by the extension to the timed variant in section 5.2. In section 5.3 we present the extension of non-timed Symbolic Automata as defined in [Sch00] and add time, effectively merging Symbolic Automata and timed Büchi automata.

5.1 Büchi-Automata

Büchi automata ([Tho90]) are able to deal with words of infinite length, in contrast to finite automata, which accept only finite words. Their expressiveness is identical to ω-regular expressions, which are an extension of finite regular expressions. Extending the Kleene star the ω indicates unbounded repetition: a^ω for instance represents an infinite sequence of a's. The definition of ω-regular languages is in fact given by the relation to Büchi automata: A language is called ω-regular iff it is accepted by some Büchi automaton.

The acceptance criterion for Büchi automata (and other automata on infinite words) needs to take the infiniteness of the words into account. Informally the set of accepting states of a Büchi automaton is defined by those states, which are visited infinitely often. These states are also called *fair* states. In the context of acceptance of infinite words we will in the remainder only speak of fair states, reserving the term *accepting state* for finite words. For automata generated from LSCs we will often use the term *final state*, which marks the complete traversal of the LSC. The term final state is not synonymous to fair state, even though it is always contained in the set of fair states as we will see later. For more information on other automata on infinite words, which differ in the definition of the acceptance condition, see [Tho90] or [AD94].

The following definition introduces non-deterministic Büchi automata following [Tho90] and [AD94]. The goal of our semantics is to generate deterministic Büchi automata from LSCs, as they are easier to handle for practical applications; this is not possible in all cases as will be seen later (see e.g. section 6.2.4 on page 118).

5.1 Definition ((Non-deterministic) Büchi Automaton)
A (non-deterministic) Büchi automaton $\mathcal{BA}$ is a tuple

$$\mathcal{BA} = (\Sigma, Q, q_0, \longrightarrow, F), \text{where}$$

- Σ is a finite alphabet of input symbols,
- Q is a finite set of states,
- $q_0 \in Q$ is the initial state,
- $\longrightarrow \subseteq Q \times \Sigma \times Q$ is the transition relation. A transition $(q, \sigma, q') \in \longrightarrow$ represents the change from state q to state q' on input symbol σ. We typically write a transition in the form $q \xrightarrow{\sigma} q'$,
- $F \subseteq Q$ is the set of fair states.

Let $\sigma = \sigma_0\sigma_1 \ldots$ be an infinite word over alphabet $\Sigma, \sigma \in \Sigma^\omega$. A *run* r of $\mathcal{BA}$ over σ is a sequence of transitions

$$r : q_0 \xrightarrow{\sigma_0} q_1 \xrightarrow{\sigma_1} \ldots \text{ , such that}$$

$\forall i \geq 0 : (q_i, \sigma_i, q_{i+1}) \in \longrightarrow$ (the target state of each transition is the source state of the following transition.

Let $inf(r) \subseteq Q$ denote the set of states of $\mathcal{BA}$ which are visited infinitely often by run r, i.e. $inf(r)$ consists of those states $q \in Q$ such that $q = q_i$ for infinitely many $i \geq 0$. The language accepted by $\mathcal{BA}$ is defined as:

$$\mathcal{L}(\mathcal{BA}) := \{\sigma = \sigma_0\sigma_1 \cdots \in \Sigma^\omega \mid \exists r = q_0 \xrightarrow{\sigma_0} q_1 \xrightarrow{\sigma_1} \cdots : inf(r) \cap F \neq \emptyset\},$$

■

Informally the Büchi acceptance criterion says, that those runs are accepted by a Büchi automaton, which visit some state $q \in F$ infinitely often. The language $\mathcal{L}(\mathcal{BA})$ accepted by $\mathcal{BA}$ consists of those words $\sigma \in \Sigma^\omega$, for which there is an accepting run.

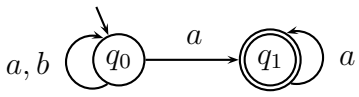


Figure 5.1: example Büchi automaton

EXAMPLE 5.1 (BÜCHI AUTOMATA)
Figure 5.1 shows an example of a Büchi automaton. The initial state is marked by a transition without a source state and fair states by a double border. In this example there is only one fair state, q_1. The automaton accepts the language $(a + b)^* a^\omega$. ■

5.2 Timed Büchi Automata

When the occurrence times of the letters of words are important, timed languages are used and consequently a corresponding type of automaton is needed: a *timed automaton*. We only consider timed Büchi automata here, there also exist variants for finite automata and other timed automata on infinite words (see [AD94]). For the definition of timed Büchi automata we loosely follow the lead of [AD94] here, inasmuch as we associate an occurrence time to each symbol of a word, yielding timed words. In contrast to [AD94], however, we only consider discrete time instead of dense time as this is sufficient for our purpose. Time is represented by a sequence of time values from the set of non-negative natural numbers: $\tau_i \in \mathbf{N}$, which has to satisfy two constraints: time only advances and time never stands still:

5.2 DEFINITION (TIMED WORD)
A *time sequence* $\tau = \tau_0\tau_1\tau_2\ldots$ is an infinite sequence of time values $\tau_i \in \mathbf{N}$ for which the following holds:

1. $\tau_i < \tau_{i+1}, \forall i \geq 0$ (τ is strictly monotonically increasing).
2. $\forall t \in \mathbf{N}\ \exists i \geq 1 : \tau_i > t$ (time always progresses)

A *timed word* over an alphabet Σ is then a pair (σ, τ), where $\sigma = \sigma_0\sigma_1\ldots$ is an infinite word and $\tau = \tau_0\tau_1\ldots$ is a time sequence. The time value τ_i denotes the occurrence time of input symbol σ_i. ■

In the untimed case the behavior of the automaton depends only on the input symbols, i.e. being in some state the next states of an automaton are determined by the current input symbol. In order for an automaton to also accept timed words it needs a means to count time, since the choice of the next states also depends on the occurrence time of the input symbols in question. This is realized by *clocks*, which can be set to zero on any transition

of the automaton and count the time since their last reset. This allows for the introduction of clock constraints on transitions – i.e. a transition may only be taken, if all its clock constraints are satisfied – forcing the input word to obey certain timing requirements. Thus time is introduced into an automaton by adding a (finite) set of clocks and augmenting transitions by clock resets and clock constraints formulated over this set of clocks.

Before giving the formal definition of a timed automaton it is necessary to determine which are legal expressions for clock constraints and how the value of the clocks is resolved. For our purposes it is sufficient to allow comparison of clock values to constants, conjunction and disjunction of clock constraints.

5.3 Definition (Clock Constraint)
Let C be a set of clocks. A *clock constraint* $\gamma \in \Phi(C)$ is defined as

$$\gamma := \epsilon \mid x < c \mid x \leq c \mid x = c \mid x > c \mid x \geq c \mid \gamma_1 \wedge \gamma_2 \mid \gamma_1 \vee \gamma_2,$$

where $x \in C$ and $c \in \mathbf{N}$. ϵ represents the empty clock constraint.

A *clock interpretation* ν assigns to each clock $x \in C$ a value of the time domain:

$$\nu : C \longrightarrow \mathbf{N}$$

The set of all clock interpretations is defined as I.
The valuation of a clock constraint is defined by

$$[\![\cdot]\!](\cdot) : \Phi(C) \times I \longrightarrow \mathbf{B}$$

$$\begin{aligned}
[\![\epsilon]\!](\nu) &:= true \\
[\![x < c]\!](\nu) &:= \nu(x) < c \\
[\![x \leq c]\!](\nu) &:= \nu(x) \leq c \\
[\![x > c]\!](\nu) &:= \nu(x) > c \\
[\![x \geq c]\!](\nu) &:= \nu(x) \geq c \\
[\![x = c]\!](\nu) &:= \nu(x) = c \\
[\![\gamma_1 \wedge \gamma_2]\!](\nu) &:= [\![\gamma_1]\!](\nu) \wedge [\![\gamma_2]\!](\nu) \\
[\![\gamma_1 \vee \gamma_2]\!](\nu) &:= [\![\gamma_1]\!](\nu) \vee [\![\gamma_2]\!](\nu)
\end{aligned}$$

Let $\nu + t$ be the clock interpretation, which adds t to all clocks: $\nu + t := \forall x \in C : (\nu + t)(x) = \nu(x) + t$. ■

In the following the definition of a timed Büchi automaton is given by extending definition 5.1 on page 82. The set of clocks used in the clock constraints of the automaton is added and the transitions between states are augmented by clock resets and constraints. The acceptance criterion is adjusted accordingly, so that fair states are defined in terms of timed words.

5.4 Definition (Timed Büchi Automaton)
A *timed Büchi automaton* $\mathcal{TBA}$ is a tuple

$$\mathcal{TBA} := (\Sigma, Q, q_0, C, \longrightarrow, F), \text{where}$$

- Σ is a finite alphabet of input symbols,
- Q is a finite set of states,
- $q_0 \in Q$ is the initial state,
- C is a finite set of clocks
- $\longrightarrow \subseteq Q \times \Sigma \times \mathcal{P}(C) \times \Phi(C) \times Q$ is the transition relation. A transition $(q, \sigma, \rho, \gamma, q') \in \longrightarrow$ represents the change from state q to state q' on input symbol σ. The set $\rho \in \mathcal{P}(C)$ indicates which clocks are reset when taking the transition and γ is a clock constraint over C, which has to be fulfilled.
- $F \subseteq Q$ is the set of fair states.

Let $(\sigma, \tau) = (\sigma_0, \tau_0)(\sigma_1, \tau_1)\ldots$ be a timed word over Σ. A *timed run tr* of a timed Büchi automaton $\mathcal{TBA}$ over timed word (σ, τ) is an infinite sequence of pairs q_i, ν_i, where $q_i \in Q$ is the i-th state of the automaton along the run and $\nu_i \in I$ is the clock interpretation in this state:

$$tr : (q_0, \nu_0) \xrightarrow[\tau_0]{\sigma_0} (q_1, \nu_1) \xrightarrow[\tau_1]{\sigma_1} (q_2, \nu_2) \xrightarrow[\tau_2]{\sigma_2} \ldots, \text{with}$$

- $\forall x \in C : \nu_0(x) = 0$, (initially all clocks are zero)
- $\forall i \geq 1 : \nu_i := \nu_{i-1} + \tau_i - \tau_{i-1}$

- $\forall i \geq 0\, \exists\, (q_i, \sigma_i, \rho_i, \gamma_i, q_{i+1}) \in \longrightarrow\colon\ [\![\gamma_i]\!](\nu_i) = true \wedge \forall x \in \rho_i : \nu_{i+1}(x) = 0 \wedge \forall x \notin \rho_i : \nu_{i+1}(x) = \nu_i(x) + \tau_i - \tau_{i-1}$
  (the target state of each transition is the source state of the following transition, the transition respects all its clock constraints, resets all appropriate clocks and all clocks, which are not reset, correctly advance the time).

Let $inf(tr) \subseteq Q$ denote the set of states of $\mathcal{TBA}$ which are visited infinitely often by timed run tr, i.e. $inf(tr)$ consists of those states $q \in Q$ such that $q = q_i$ for infinitely many $i \geq 0$. The language accepted by $\mathcal{TBA}$ is defined as the set of timed words, for which there is an accepting run of $\mathcal{TBA}$:

$$\mathcal{L}(\mathcal{TBA}) := \{(\sigma, \tau) \mid \exists tr = (q_0, \nu_0) \xrightarrow[\tau_0]{\sigma_0} (q_1, \nu_1) \xrightarrow[\tau_1]{\sigma_1} \cdots : inf(tr) \cap F \neq \emptyset\}$$

■

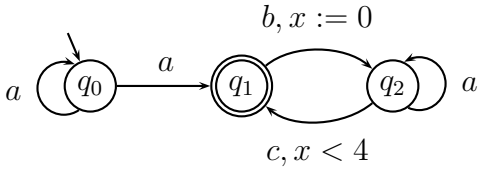


Figure 5.2: example timed Büchi automaton

Example 5.2
Figure 5.2 shows an example of a timed Büchi automaton, which accepts the language $\{(\sigma, \tau) \mid \sigma \in a^+(ba^*c)^\omega \wedge \forall i\, \exists j > i : \sigma_i = b \Rightarrow \sigma_j = c \wedge \tau_j < \tau_i + 4\}$.
■

5.3 Symbolic Automata

The automata described so far operate on single input symbols only, since they allow but one element of Σ per transition. In order to be able to describe the communication behavior of a system, it is necessary to allow more than one observation at a time. Symbolic Timing Diagrams faced the same

problem, which in [Sch00] is solved by the extension of Büchi automata to *symbolic automata*, where a run is extended to a *computation sequence* referring to valuations of system variables and formulas are allowed as transition annotations in the automaton. We adopt this strategy for the LSC semantics definition and therefore use symbolic automata.

[Sch00] does not consider quantitative timing for STDs, so that symbolic automata are untimed. We consequently extend the definition of symbolic automata to also encompass (discrete) time, similar to timed Büchi automata. First, we extend the notion of a timed word to a timed symbolic word (called a computation sequence in [Sch00]).

5.5 Definition (Timed Symbolic Word)
Let V be a set of typed variables, DOM the domain of variables $v \in V$ and $\theta : V \longrightarrow DOM$ a valuation, which assigns to each $v \in V$ a value of its domain. Furthermore let τ be a time sequence.
A *symbolic word* then is an infinite sequence $\theta = \theta_0\theta_1\theta_2 \ldots$
A *timed symbolic word* then is a pair $(\theta, \tau) = (\theta_0, \tau_0)(\theta_1, \tau_1)(\theta_2, \tau_2) \ldots$, where τ_i denotes the occurrence time of valuation θ_i. ■

In order to refer to the occurrence of several communication events it is necessary to extend the timed Büchi automata to allow formulas:

5.6 Definition (Formula over Σ)
Let $\Sigma \subseteq V$ be a set of boolean variables. A *formula* ψ over Σ is a boolean expression produced by the following rules:

$$\psi := \sigma \mid \neg\psi \mid (\psi) \mid \psi_1 \wedge \psi_2 \mid \psi_1 \vee \psi_2$$

with $\sigma \in \Sigma$.
The set of all formulas over Σ is denoted by $BExpr_\Sigma$. ■

5.7 Definition (Validity of a formula)
Let θ be a valuation and ψ a formula. The validity of ψ with respect to θ, denoted $\theta \models \psi$ is defined as follows:

$$\begin{aligned}
\theta \models \sigma &:= \theta(\sigma) = true \\
\theta \models \neg\psi &:= not\ \theta \models \psi \\
\theta \models (\psi) &:= \theta \models \psi \\
\theta \models \psi_1 \wedge \psi_2 &:= \theta \models \psi_1\ and\ \theta \models \psi_2 \\
\theta \models \psi_1 \vee \psi_2 &:= \theta \models \psi_1\ or\ \theta \models \psi_2
\end{aligned}$$

■

We can now define in extension of definition 5.4 on page 86 a timed symbolic automaton:

5.8 Definition ((Non-deterministic) Timed Symbolic Automaton) A *timed symbolic automaton* $\mathcal{TSA}$ is a tuple

$$\mathcal{TSA} := (\Sigma, Q, q_0, C, \longrightarrow, F),$$

where

- Σ is a finite alphabet of input symbols (variables),
- Q is a finite set of states,
- $q_0 \in Q$ is the initial state,
- C is a finite set of clocks
- $\longrightarrow \subseteq Q \times BExpr_\Sigma \times \mathcal{P}(C) \times \Phi(C) \times Q$ is the transition relation. A transition $(q, \psi, \rho, \gamma, q') \in \longrightarrow$ represents the change from state q to state q' while satisfying formula ψ. The set $\rho \in \mathcal{P}(C)$ indicates which clocks are reset when taking the transition and γ is a clock constraint over C, which has to be fulfilled.
- $F \subseteq Q$ is the set of fair states.

A *timed run tsr* of a timed symbolic automaton $\mathcal{TSA}$ over a timed symbolic word (θ, τ) is an infinite sequence of pairs (q_i, ν_i), where $q_i \in Q$ is the i-th state of the automaton along the run, $\nu_i \in I$ is the clock interpretation in this state:

$$tsr : (q_0, \nu_0) \xrightarrow[\tau_0]{\theta_0} (q_1, \nu_1) \xrightarrow[\tau_1]{\theta_1} (q_2, \nu_2) \xrightarrow[\tau_2]{\theta_2} \ldots, \text{with}$$

- $\forall x \in C : \nu_0(x) = 0$, (initially all clocks are zero)
- $\forall i \geq 1 : \nu_i := \nu_{i-1} + \tau_i - \tau_{i-1}$
- $\forall i \geq 0 \, \exists \, (q_i, \psi_i, \rho_i, \gamma_i, q_{i+1}) \in \longrightarrow : \; \theta_i \models \psi_i \wedge [\![\gamma_i]\!](\nu_i) = true \wedge \forall x \in \rho_i : \nu_{i+1}(x) = 0 \wedge \forall x \notin \rho_i : \nu_{i+1}(x) = \nu_i(x) + \tau_i - \tau_{i-1}$
  (the target state of each transition is the source state of the following transition, the boolean expression annotating the transition is evaluated to true, the transition respects all its clock constraints, resets all appropriate clocks and all clocks, which are not reset, correctly advance the time).

Let $inf(tsr) \subseteq Q$ denote the set of states of $\mathcal{TSA}$ which are visited infinitely often by timed run tsr, i.e. $inf(tsr)$ consists of those states $q \in Q$ such that $q = q_i$ for infinitely many $i \geq 0$. The language accepted by $\mathcal{TSA}$ is defined as:

$$\mathcal{L}(\mathcal{TSA}) := \{(\theta, \tau) \,|\exists \; tsr = q_0 \xrightarrow[\tau_0]{\theta_0} q_1 \xrightarrow[\tau_1]{\theta_1} q_2 \xrightarrow[\tau_2]{\theta_2} \cdots : inf(tsr) \cap F \neq \emptyset\}$$

■

The definition of the semantics for pre-charts requires a finite automaton (cf. chapter 9), so that here a finite variant of timed symbolic automaton is introduced as well. This *timed finite automaton* is similar to a timed symbolic automaton, except for the definition of acceptance.

5.9 Definition (Timed Finite Automaton)
A *timed finite automaton* $\mathcal{TFA}$ is a tuple

$$\mathcal{TFA} := (\Sigma, Q, q_0, C, \longrightarrow, F),$$

where

- Σ is a finite alphabet of input symbols (variables),
- Q is a finite set of states,
- $q_0 \in Q$ is the initial state,
- C is a finite set of clocks

- $\longrightarrow \subseteq Q \times BExpr_\Sigma \times \mathcal{P}(C) \times \Phi(C) \times Q$ is the transition relation. A transition $(q, \psi, \rho, \gamma, q') \in \longrightarrow$ represents the change from state q to state q' on formula ψ. The set $\rho \in \mathcal{P}(C)$ indicates which clocks are reset when taking the transition and γ is a clock constraint over C, which has to be fulfilled.

- $F \subseteq Q$ is the set of accepting states.

Let (θ, τ) be a finite timed word: $(\theta, \tau) = (\theta_0, \tau_0) \dots (\theta_n, \tau_n)$. A *finite timed run* tfr of a timed finite automaton $\mathcal{TFA}$ over (θ, τ) is a finite sequence of pairs (q_i, ν_i), where $q_i \in Q$ is the i-th state of the automaton along the run, $\nu_i \in I$ is the clock interpretation in this state:

$$tfr : (q_0, \nu_0) \xrightarrow[\tau_0]{\theta_0} \dots \xrightarrow[\tau_n]{\theta_n} (q_{n+1}, \nu_{n+1}), \text{with}$$

- $\forall x \in C : \nu_0(x) = 0$, (initially all clocks are zero)

- $\forall 1 \leq i \leq n+1 : \nu_i := \nu_{i-1} + \tau_i - \tau_{i-1}$

- $\forall 0 \leq i \leq n \, \exists \, (q_i, \psi_i, \rho_i, \gamma_i, q_{i+1}) \in \longrightarrow : \, \theta_i \models \psi_i \wedge [\![\gamma_i]\!](\nu_i) = true \wedge \forall x \in \rho_i : \nu_{i+1}(x) = 0 \wedge \forall x \notin \rho_i : \nu_{i+1}(x) = \nu_i(x) + \tau_i - \tau_{i-1}$
  (the target state of each transition is the source state of the following transition, the boolean expression annotating the transition is evaluated to true, the transition respects all its clock constraints, resets all appropriate clocks and all clocks, which are not reset, correctly advance the time).

A finite timed word is accepted by $\mathcal{TFA}$, **iff** $q_{n+1} \in F$. The language accepted by $\mathcal{TFA}$ is defined as:

$$\mathcal{L}(\mathcal{TFA}) := \{(\theta, \tau) \mid \exists \, tfr = (q_0, \nu_0) \xrightarrow[\tau_0]{\theta_0} \dots \xrightarrow[\tau_n]{\theta_n} (q_{n+1}, \nu_{n+1}) : q_{n+1} \in F\}$$

■

Relation between Büchi Automata and Linear Temporal Logic

There is a well-known relation between ω-regular languages, i.e. Büchi automata, and linear time temporal logic (LTL): star-free ω-regular languages are identical in expressiveness to LTL [Eme90]. Schlör [Sch00] constructively

shows that this property holds for symbolic automata by defining a translation from symbolic automata to LTL. Wittke [Wit03] additionally considers STDs with quantitative timing constraints. The results of these works form the base for the application of LSCs to formal verification considered later in this thesis. The formal basis for the LSC semantics are symbolic automata, however, the reader is referred to the above works for a description of the translation to LTL. Note that both of these approaches require that the automaton to be translated contains at most trivial loops, i.e. self loops on states. Automata fulfilling this property are called *partially ordered* in [Sch00] and *flat* in [Wit03]. The algorithm for unwinding LSCs into timed symbolic automata ensures that this requirement is respected as will become apparent later.

Chapter 6

Semantics of the LSC Kernel Language

This chapter introduces the semantics of the LSC features presented in chapter 4. The definition of the semantics is split into two phases: the definition of the semantics for the kernel behavior of an LSC in terms of a symbolic automaton, and the treatment of quantification, activation condition and mode, since these bits of information can be handled much more efficiently after the core automaton has been generated. Especially the invariant mode would complicate the automaton generation procedure considerably, since a reactivation of the automaton must be possible in every state. This would entail a very complex construction algorithm, since for every state of the core automaton the product of the core automaton with itself would have to be computed. Thus we will first generate only the core automaton, disregarding possible reactivations.

The procedure of deriving an automaton from an LSCs, which we call *unwinding*, is inspired by the semantics definition for Symbolic Timing Diagrams presented in [Sch00], which has been taken as a blue print for the semantics definition in [DH01]. The semantics given in [DH01], however, use a symbolic transition system, the so-called *skeleton automaton*, as a semantical framework. The unwinding algorithm presented in this chapter operates on a propositional level in order to be independent of the particulars of a specific modeling language and its expression language.
A preliminary and incomplete version of the formal semantics presented in this chapter has been published in [KW01].

Before giving the details of the unwinding algorithm we define the formal

syntax of LSC elements in section 6.1, which is the starting point for the translation from an LSC into a timed symbolic automaton presented in section 6.2. Since time is introduced later in chapter 7, the automaton does not contain clocks or clock constraints. Activation and Quantification information is incorporated in section 6.3. We conclude this chapter with a review of related work in section 6.4.

6.1 Formal Syntax

This section defines the syntax of the LSC elements introduced in chapter 4. This definition is the base for all further extensions in the following chapters. An LSC consists of several components: the body of the LSC — i.e. the instances and events defined on it —, the activation condition and mode, the quantification and the pre-chart, which is not discussed here, but is dealt with in detail in chapter 9. There is also a (possibly empty) set of assumptions linked to an LSC, which is discussed in chapter 8. In the remainder of this section the focus is on the body of the LSC, the other information is dealt with in section 6.3.

6.1 Definition (LSC)
An LSC is a tuple $L = (l, assumptions, ac, pch, amode, quant)$, with

- l : the body of LSC
- $assumptions$: a set of assumptions
- ac : the activation condition
- pch : the pre-chart
- $amode$: the activation mode
- $quant$: the quantification

■

An LSC body consists of a number of instances, which are collected in the set $Inst(l)$. In the following let l denote the body of an LSC L, and let $i \in Inst(l)$ denote some instance of l. The basic blocks ($atoms$) of which an LSC body is comprised are the following:

- instance heads
- instance ends
- sending a message
- receiving a message
- condition atom (local to one instance)
- start of a local invariant
- end of a local invariant

The atoms carry the progress information of each instance (cf. section 6.2.1 on page 103) and are organized according to their positioning on the instance axes. The atoms are thus instance-wise collected in sets:

- $Msgsnd(i)$: set of message send atoms
- $Msgrcv(i)$: set of message receive atoms
- $Conds(i)$: set of condition atoms[1]
- $LI\_starts(i)$: set of start atoms of local invariants
- $LI\_ends(i)$: set of end atoms of local invariants

There is only one instance head atom, denoted by $\bot_i$, and one instance end atom, denoted by $\top_i$, for each instance i, so that it is not necessary to have sets for these atoms on the instance level. Collecting all atoms of the LSC body according to type yields the following sets:

- $Instheads(l) := \bigcup_{i \in Inst(l)} \{\bot_i\}$
- $Instends(l) := \bigcup_{i \in Inst(l)} \{\top_i\}$
- $Msgsnd(l) := \bigcup_{i \in Inst(l)} Msgsnd(i)$
- $Msgrcv(l) := \bigcup_{i \in Inst(l)} Msgrcv(i)$

[1]Note that a condition atom is local to one instance. A condition shared by several instances consists of one condition atom on each participating instance.

- $Conds(l) := \bigcup_{i \in Inst(l)} Conds(i)$
- $LI\_starts(l) := \bigcup_{i \in Inst(l)} LI\_starts(i)$
- $LI\_ends(l) := \bigcup_{i \in Inst(l)} LI\_ends(i)$

Collecting all atoms of an instance, resp. of the entire body yields the sets:

$$\begin{aligned} Atoms(i) := \quad & \{\bot_i\} \cup \\ & Msgsnd(i) \cup \\ & Msgrcv(i) \cup \\ & Conds(i) \cup \\ & LI\_starts(i) \cup \\ & LI\_ends(i) \cup \\ & \{\top_i\} \end{aligned}$$

$$\begin{aligned} Atoms(l) := \quad & Instheads(l) \cup \\ & Msgsnd(l) \cup \\ & Msgrcv(l) \cup \\ & Conds(l) \cup \\ & LI\_starts(l) \cup \\ & LI\_ends(l) \cup \\ & Instends(l) \end{aligned}$$

The atoms of each instance are ordered from top to bottom as drawn in the LSC[2] and thus have a graphical position, given by the function $position(a)$ for $a \in Atoms(i)$.

6.2 Definition (Atom Position)
Let $a, a', a'' \in Atoms(i)$ and $a \prec_i a'$ denote the order of a and a' as drawn on instance i. The function $position(\cdot)$ assigns a natural number to each atom $a \in Atoms(l)$ of LSC body l. This number is called *position of a*.

$$position : Atoms(l) \longrightarrow \mathbf{N}_0$$

$$position(a) := \begin{cases} 0 & \text{if } \forall a' \neq a : a' \not\prec_i a \\ 1 + position(a') & \text{if } a' \prec_i a \wedge \neg\exists a'' \in Atoms(i) : \\ & \qquad a' \prec_i a'' \prec_i a \end{cases}$$

■

[2]This total order can be disrupted by a coregion as discussed later.

Remark 6.1 (Atom Positions)
The definition of the position of an atom does not assign a unique number to each atom, but allows several atoms of one instance to share one position. This is necessary in order to correctly describe simultaneous regions (see below), which are characterized by the fact that more than one atom occupies the same position. ■

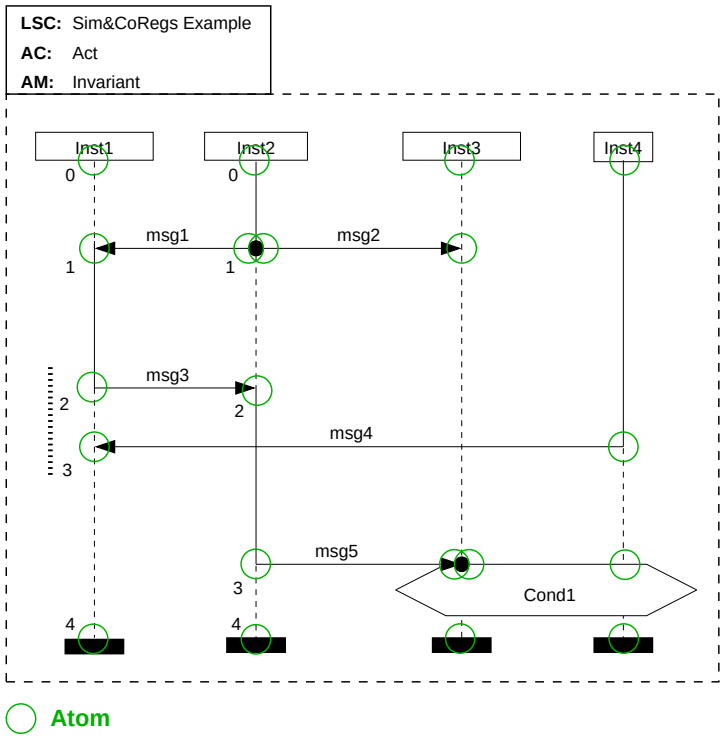


Figure 6.1: LSC with highlighted atoms

Example 6.1 (Positions of Atoms)
Figure 6.1 shows the example LSC from figure 4.7 with atoms highlighted. Note that the two simultaneous regions each contain two atoms: the sending atom of message `msg1` and the sending atom of message `msg2`, and the receipt atom of message `msg5` and one condition atom of condition `Cond1`, respectively. It also illustrates that conditions may consist of more than one atom, `Cond1` e.g. is comprised of two condition atoms, one on `Inst3` and one on `Inst4`. For instances `Inst1` and `Inst2` we also give the atom positions. ■

Atoms $a \in Atoms(i)$ are grouped into *clusters*, which are characterized by the fact that all atoms contained in them are observed simultaneously and therefore have the same position. This concept is needed to express simultaneous regions.

6.3 Definition (Cluster)
The set of clusters of an instance $i \in Inst(l)$ is defined by the maximal set

$$Clusters(i) := \{cl \subseteq Atoms(i) \mid \forall a, a' \in cl : \ position(a) = position(a')\}.$$

The set of clusters of an LSC body l is defined by

$$Clusters(l) := \bigcup_{i \in Inst(l)} Clusters(i).$$

The position function is extended to cover clusters as well:

$$position(cl) := position(a), a \in cl, cl \in Clusters(l).$$

■

Example 6.2 (Clusters & simultaneous regions)
Figure 6.2 illustrates the cluster concept. Most of the clusters contain only one atom, only the two clusters, which correspond to simultaneous regions contain two atoms each. Also note that the positions of the clusters and the atoms contained in them are identical. ■

Remark 6.2 (Clusters)
Note that each atom is contained in exactly one cluster, but that each cluster may contain several atoms. The latter case only arises, if the cluster corresponds to a simultaneous region; atoms outside of a simultaneous region result in a singleton cluster. ■

The clusters on each instance as defined so far are totally ordered; as introduced in chapter 4, coregions allow to suspend this total order. A coregion cr is thus a set of unordered clusters and contains all those clusters, which are covered by the dotted line next to the instance axis; see figure 6.1 on the page before for an example. Each coregion consists of a set of unordered clusters: $cr := \{cl_1, \ldots, cl_n\} \subseteq Clusters(i)$. The set of all coregions of instance i is given by $Coregions(i)$ and the set of all coregions of an LSC body

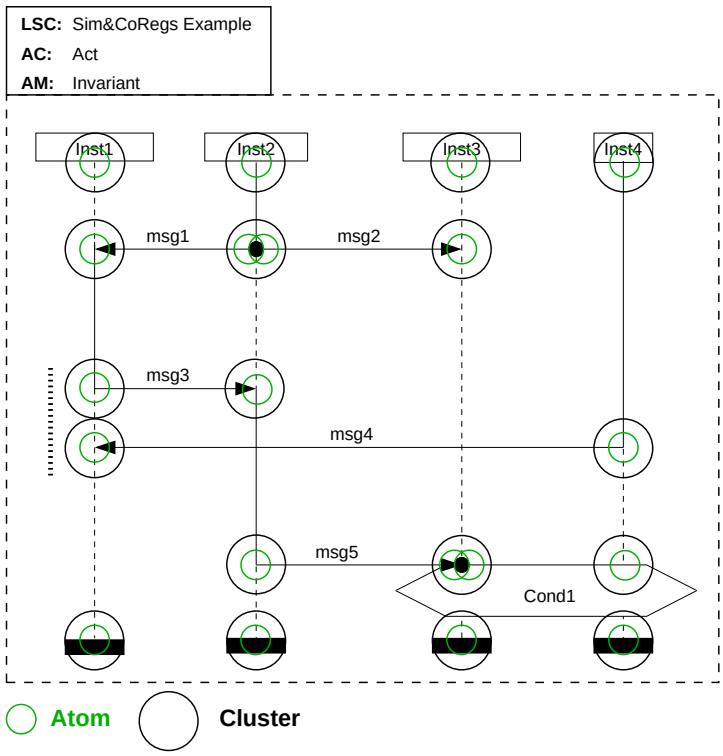


Figure 6.2: LSC with highlighted atoms and clusters

l by $Coregions(l)$. For each cluster $cl \in Cluster(l)$ we define the function $coreg(cl)$, which returns the coregion cl is part of:

$$coreg(cl) := \begin{cases} \emptyset & , \text{if} \neg\exists cr \in Coregions(l) : cl \in cr \\ cr & , \text{else} \end{cases}$$

The position does not reflect the relaxed ordering requirement imposed by a coregion: the sending of `msg3` and the receipt of `msg4` on `Inst1` in figure 6.1 on page 97 are still ordered according to their positions. In order to correctly capture the coregion semantics a *logical* position, called *location* is needed:

6.4 Definition (Location)
The *location* of a cluster $cl \in Clusters(i)$ is given by the function

$$location(cl) := \begin{cases} position(cl), & \text{if } coreg(cl) = \emptyset \\ min(\{position(cl_i) \mid cl_i \in coreg(cl)\}), & \text{if } coreg(cl) \neq \emptyset \end{cases}$$

$Locations(i) := \{\ location(cl) \mid cl \in Clusters(i)\}$ is the set of locations of instance i and $Locations(l) := \bigcup_{i \in Inst(l)} Locations(i)$ the set of all locations of LSC body l. ■

Remark 6.3 (Locations)

- *The locations are unique only on one instance, since location(·) relies on position(·).*
- *The location of a cluster is equal to its position, unless it is part of a coregion.*
- *The location of a cluster, which is not part of a coregion is unique on its instance.*
- *All clusters in a coregion have the same location, but different positions.*
- *Clusters on instances without coregions are totally ordered.*
- *Clusters on instances with coregions are partially ordered.*

■

Example 6.3 (Locations)
Figure 6.3 on the next page illustrates the location concept. Notice that the location of a cluster is identical to its position, except for clusters in coregions. Here all clusters in the coregion share one location. ■

The atoms introduced above are local to one instance. In order to be able to refer to the graphical elements used in the LSC it is necessary to establish the connection between atoms, which belong to the same LSC element but are located on different instances. Each message e.g. is made up of two atoms, the send and the receive atom, which are located on different instances. Likewise, conditions, which involve more than one instance are comprised by one condition atom on each participating instance. The relation between such atoms is established by the identifier, which designates the graphical element. For an LSC body l there are thus the following disjunct sets of unique identifiers and the following functions, which associate an atom with its identifier:

- the set of instance identifiers in l : $Instances(l)$

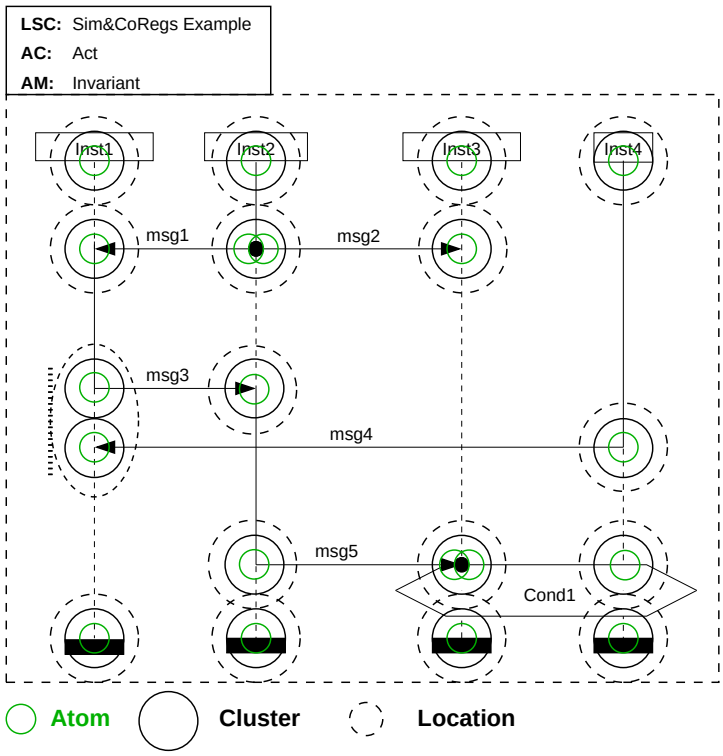


Figure 6.3: LSC with highlighted atoms, clusters and locations

- the set of message identifiers in $l : Messages(l)$
- the set of condition identifiers in $l : Conditions(l)$
- the set of identifiers of local invariants in $l : Local\_Invariants(l)$

$$instID : Inst(l) \longrightarrow Instances(l)$$
$$msgID : Msgsnd(l) \cup Msgrcv(l) \longrightarrow Messages(l)$$
$$condID : Conds(l) \longrightarrow Conditions(l)$$
$$liID : LI\_starts(l) \cup LI\_ends(l) \longrightarrow Local\_Invariants(l)$$

The identification of the send and receive part of a message is achieved by the $msgID(\cdot)$ function. In the transition annotations of the automaton generated from an LSC body (cf. section 6.2) it is necessary to distinguish

send and receive atoms, therefore the set of message labels $MsgLabels(l)$ is introduced, including the function $msgLabel(m)$, which associates a message label with each $m \in Msgsnd(l) \cup Msgrcv(l)$. Sending of message $m1$ is represented by the label $!m1$, receipt by $?m1$.

$$msgLabel : Msgsnd(l) \cup Msgrcv(l) \longrightarrow MsgLabels(l)$$

$$msgLabel(m) := \begin{cases} !msgID(m) & \text{if } m \in Msgsnd(l) \\ ?msgID(m) & \text{if } m \in Msgrcv(l) \end{cases}$$

Furthermore is it necessary to access the properties of messages, conditions and local invariants. For messages the relevant properties are temperature and type (instantaneous or asynchronous), for conditions and local invariants the mode (mandatory or possible). The progress information along each instance is associated with the atoms of this instance, so that the domain of function $temp(\cdot)$ below is the union of message identifiers and atoms. The properties for messages and conditions on the other hand pertain to the LSC elements, represented by the corresponding identifiers, and not the atoms. The situation for the mode of local invariants is identical to that of conditions, i.e. this information is tied to the LSC element. The information, if a local invariant atom is included in the simultaneous region, which contains it, on the other hand pertains to the individual atom, so that the *incl* function is defined on the set of local invariant atoms rather than on the set of identifiers.

$$temp : Messages(l) \cup Atoms(l) \longrightarrow \{hot, cold\}$$

$$sync\_type : Messages(l) \longrightarrow \{async, instant\}$$

$$mode : Conditions(l) \cup Local\_Invariants(l) \longrightarrow \{mandatory, possible\}$$

$$incl : LI\_starts(l) \cup LI\_ends(l) \longrightarrow \mathbf{B}$$

The treatment of liveness requirements in the formal semantics in the next section is tied to locations, so that the atom temperature has to be lifted to the location level. An intermediate step to achieve this is to first lift it to the Cluster level, which in fact determines the temperature of simultaneous regions. Analogously to the cut temperature computation in section 6.2.3 on page 117 one hot atom overrides the temperature of all other atoms in a simultaneous regions.

$$temp : Clusters(l) \longrightarrow \{hot, cold\}$$

$$temp(cl) := \begin{cases} cold, & \text{if } \forall a \in cl : temp(a) = cold \\ hot, & \text{else} \end{cases}$$

The temperature function is extended analogously to locations, so that coregions can be treated correctly. Since no ordering exists between the atoms of a coregion, no progress between them can be enforced, so that a single temperature is associated with a coregion. In the remainder we only apply the $temp(\cdot)$ function to message identifiers and locations, so that no confusion should arise.

$$temp : Locations(l) \longrightarrow \{hot, cold\}$$

$$temp(loc) := \begin{cases} cold, & \text{if } \forall cl \in Clusters(l) : location(cl) = loc \wedge \\ & \qquad temp(cl) = cold \\ hot, & \text{else} \end{cases}$$

6.2 Formal Semantics

In this section we describe the generation of the automaton from an LSC. We first construct the basic structure and then discuss several semantical issues, which influence the final automaton, before completing the procedure.

6.2.1 Basic Automaton Construction

The unwinding approach has been adopted from the definition of the semantics of Symbolic Timing Diagrams [Sch00, Fey96, FJ97]. The general idea is to define a cut through the LSC starting at the top and moving this cut downward until we reach all instance ends, while respecting the partial order imposed by the LSC. Cuts become states in the automaton and the transition relation of the automaton encodes the successor relation among the cuts. Ordering among the atoms in an LSC is induced by the following rules:

1. atoms (clusters) along each instance axis are totally ordered (unless they are part of a coregion)

2. a message has to be sent before it can be received

3. conditions ranging over several instances (*shared conditions*) enforce synchronization between the involved instances

These rules determine when an LSC element is *enabled*, i.e. when it is ready to be unwound:
An atom is enabled when

1. all its predecessors along the instance axis have already be unwound
2. the corresponding message send atom has already been unwound (if the atom is a message receive atom of an asynchronous message) or is being unwound simultaneously (if the atom is part of an instantaneous message)
3. all other condition atoms belonging to the same condition are also enabled (if the atom is a shared condition)

The first rule demands a computation of all predecessors on the instance axis for each atom. It is actually sufficient to only conduct a local computation of the *immediate* predecessor of each atom, since this relation is transitive: If the immediate predecessor of a cluster *cl* has been unwound, then immediate predecessor of *cl* must have been unwound, and so on. The formal definition of the function computing the immediate predecessor is:

6.5 Definition (Immediate Predecessor)

$$predecessor : Clusters(i) \longrightarrow \mathcal{P}(Clusters(i))$$

$$predecessor(cl) := \begin{cases} \emptyset & , \text{if} \exists\ a \in cl : a = \bot_i \vee a = \top_i \\ CL & , \text{else} \end{cases}, \text{ where}$$

$$CL := \{cl' \in Clusters(i) \mid location(cl') < location(cl) \wedge \neg\exists\, cl'' \in Clusters(i) : location(cl') < location(cl'') < location(cl)\}$$

■

The function $predecessor(cl)$ looks along one instance for the location, which is immediately above the cluster *cl* and returns the clusters bound to that location. If the element bound to the predecessor location is not a

coregion, the returned set of clusters contains a single cluster. Otherwise all clusters of the coregion are returned. For clusters containing instance head or end atoms no predecessor is returned, since an instance head does not have a predecessor and instance end atoms are not unwound by the algorithm (see below).

The second and third rule above involve more than one instance and thus require to also take clusters of other instances into account. The message send and receive atom for an instantaneous message have to be unwound simultaneously, and all condition atoms of a condition must be unwound at the same time. This synchronization of clusters is expressed by the relation $\approx$.

6.6 Definition (Equivalence of Clusters)
Let $cl, cl' \in Clusters(l)$. The relation $\approx$ on $Clusters(l)$ is defined as:

$$\begin{aligned} cl \approx cl' \Leftrightarrow \exists\ a \in cl\ \exists\ a' \in cl' : \\ cl = cl' \vee sharedConds(a, a')\ \vee instMsgs(a, a') \\ \vee transitivity(a, a') \end{aligned}$$

where

$$\begin{aligned} sharedConds(a, a') &:= a, a' \in Conds(l)\ \wedge\ condID(a) = condID(a') \\ instMsgs(a, a') &:= a \in Msgsnd(l)\ \wedge\ a' \in Msgrcv(l) \\ &\quad \wedge\ msgID(a) = msgID(a') \\ &\quad \wedge\ sync\_type(msgID(a)) = instant \\ transitivity(a, a') &:= a, a' \in syncAtoms(l) \\ &\quad \wedge \exists cl'' \in Clusters(l)\ \exists a_i, a_j \in cl'' : |cl''| > 1 \\ &\quad \wedge (sharedConds(a, a_i) \vee instMsgs(a, a_i)) \\ &\quad\quad \vee (sharedConds(a', a_j) \vee instMsgs(a', a_j)) \\ syncAtoms(l) &:= \{a \in Atoms(l) | a \in Conds(l) \vee \\ &\quad (a \in Msgsnd(l)\ \vee\ a \in Msgrcv(l)) \\ &\quad \wedge sync\_type(msgID(a)) = instant\} \end{aligned}$$

The classes defined by $\approx$ are called *simultaneous classes*. $SimClasses(l) := \{\ scl\ \subseteq\ Clusters(l)\ |\ \forall\ cl, cl' \in scl : cl \approx cl'\ \}$ is the set of simultaneous classes of l. ■

Remark 6.4 (Single atoms)
For single atoms, e.g. the sending of an asynchronous message, the corresponding simultaneous class is a singleton set of clusters, which in turn is a singleton set of atoms. ■

The simultaneous classes are the basic elements, which the unwinding algorithm operates on. Similar to the case for clusters it is necessary to know the predecessor(s) of a simultaneous class in order to determine, if it is enabled or not. This gives rise to the definition of the function $prerequisite(\cdot)$, which uses the $predecessor(\cdot)$ function defined for clusters:

6.7 Definition (Simultaneous Class Prerequisite)

$$prerequisite : SimClasses(l) \longrightarrow \mathcal{P}(SimClasses(l))$$

$$prerequisite(scl) \quad := \quad \begin{cases} \emptyset & , if\ \exists\ cl \in scl : \exists a \in cl : a \in Instheads(l) \\ SCL & , else \end{cases},$$

where

$$\begin{aligned} SCL := \{scl' \in & Sim\_Classes(l) \mid \\ & \exists cl \in scl\ \exists cl' \in scl' : (cl' \in predecessor(cl) \vee \\ & \exists a \in cl\ \exists a' \in cl' : a \in Msgrcv(l) \wedge \\ & a' \in Msgsnd(l)\ \wedge msgID(a) = msgID(a') \wedge \\ & sync\_type(msgID(a)) = async)\} \end{aligned}$$

■

The $prerequisite(\cdot)$ function is similar to the $predecessor(\cdot)$ function. It looks for the immediate predecessors of all clusters contained in the considered simultaneous class and collects the simultaneous classes for all predecessor clusters. If the current simultaneous class contains an asynchronous message receive atom, the simultaneous class containing the corresponding message send atom is added as well. It returns the empty set, if the considered simultaneous class contains an instance head atom.

Example 6.4
Figure 6.4 illustrates the concepts of positions, clusters, locations and SimClasses. We have omitted the SimClasses for instance heads and ends for

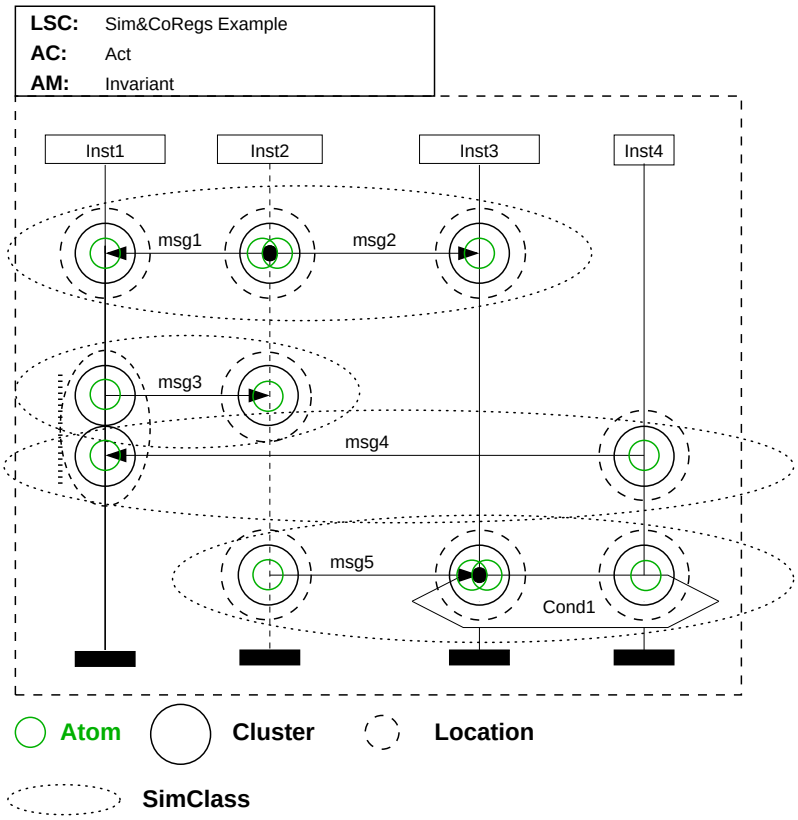


Figure 6.4: LSC with positions, locations and SimClasses

readability's sake; in these cases the SimClasses contain only one cluster, see table 6.1.

The predecessor computation is obvious, we therefore concentrate on the prerequisites. Atoms are named after their corresponding identifiers, message send atoms are characterized by prefixing the identifier with a '!', message receive atoms with a '?'. Table 6.1 shows the simultaneous classes and their prerequisites for this example.

The multitude of braces in the prerequisites is due to the nested sets. A single atom is not enclosed in braces (e.g. ?msg1), but is enclosed in a cluster, which is a set of atoms (e.g. {?msg1}). Simultaneous regions result in clusters with more than one element (e.g. {!msg1,!msg2}). The clusters are part of a SimClass, which is a set of clusters (e.g. {{!msg3}, {?msg3} }). When considering the prerequisites another level of braces is added, since

SimClass	Prerequisite
$\{\{\bot_{Inst1}\}\}$	$\emptyset$
$\{\{\bot_{Inst2}\}\}$	$\emptyset$
$\{\{\bot_{Inst3}\}\}$	$\emptyset$
$\{\{\bot_{Inst4}\}\}$	$\emptyset$
{ {?msg1},{!msg1,!msg2},{?msg2} }	{ $\{\{\bot_{Inst1}\}\}$, $\{\{\bot_{Inst2}\}\}$, $\{\{\bot_{Inst3}\}\}$ }
{ {!msg3}, {?msg3} }	{ {{?msg1},{!msg1,!msg2},{?msg2}} }
{ {?msg4}, {!msg4} }	{ {{?msg1},{!msg1,!msg2},{?msg2}}, $\{\{\bot_{Inst4}\}\}$ }
{ {!msg5}, {?msg5, cond1} }	{{{!msg3},{?msg3}},{{?msg4},{!msg4}}}
$\{\{\top_{Inst1}\}\}$	$\emptyset$
$\{\{\top_{Inst2}\}\}$	$\emptyset$
$\{\{\top_{Inst3}\}\}$	$\emptyset$
$\{\{\top_{Inst4}\}\}$	$\emptyset$

Table 6.1: Prerequisites for the LSC in figure 6.4

prerequisites are sets of SimClasses. Notice that all elements of the coregion share part of their prerequisites, which is due to the fact that coregions are treated as a single location. ■

A *cut* through the LSC is used to keep track of the progress of the unwinding procedure. It represents the borderline between already unwound elements and those which still have to be considered. The elements directly below the cut are those, which are currently enabled. Cuts are represented by a tuple containing one cluster of each instance:

6.8 Definition (Cut)
A cut $Cut \subseteq Clusters(i_1) \times \cdots \times Clusters(i_n)$, for $Inst(l) = \{i_1, \ldots, i_n\}$ is a tuple $(cl_{i_1}, \ldots, cl_{i_n}),\ cl_{i_j} \in Clusters(i_j),\ 1 \leq j \leq n = |Inst(l)|$.

Let $Cuts(l)$ be the set of all possible cuts of LSC body l. ■

The unwinding algorithm requires a number of other auxiliary sets and constructs, which are collected in a tuple called *phase*. Each phase characterizes one unwinding step and consequently corresponds to a state in the resulting automaton. Possible transitions from one phase to another correspond to transitions of the automaton.

6.9 Definition (Phase)
A phase is a tuple $Phase := (Ready, History, Cut)$, with

- $History \subseteq SimClasses(l)$: the set of simultaneous classes which have already been unwound
- $Ready \subseteq SimClasses(l)$: the set of simultaneous classes, which are currently enabled to be unwound
- Cut : the current cut as defined in definition 6.8.

Let $Phases(l)$ be the set of all phases of LSC body l. ■

Since several simultaneous classes may be enabled concurrently, e.g. due to a coregion, each ready set may contain more than one simultaneous class. In this case all combinations of elements of the ready set lead to valid successor phases and thus have to be taken into consideration. All possible combinations correspond to the powerset of the ready set; the set of simultaneous classes, which is currently chosen to be unwound, is called the *fired set* $Fired \subseteq \mathcal{P}(SimClasses(l))$. Thus, for each phase $Phase_i$ there exist k fired sets $Fired_{i_k}$, with $k = |\mathcal{P}(Ready_i)|$. Let $Firedsets(l)$ be the set of all fired sets in l.

Example 6.5 (Fired set)
Let the current ready set consist of simultaneous classes a and b, thus $Ready_i = (a, b)$. The resulting fired sets are $Fired_{i_1} = \emptyset, Fired_{i_2} = \{a\}, Fired_{i_3} = \{b\}$, and $Fired_{i_3} = \{a, b\}\}$. ■

The unwinding algorithm starts at the top of the LSC body and computes the initial phase $Phase_0$, which is given by

$$Phase_0 = (Ready_0, History_0, Cut_0),$$

with

$$
\begin{aligned}
Ready_0 = \Big\{\ & scl \in SimClasses(l) \mid \\
& prerequisite(scl) \in \Big\{\mathcal{P}\Big(\{scl' \in SimClasses(l) \mid \\
& \quad \forall cl \in scl' : \forall a \in cl : \\
& \quad\quad a \in (Instheads(l) \cup LI\_starts(l))\}\Big) \setminus \emptyset\Big\}\ \Big\} \\
History_0 = \Big\{ & \bigcup_{i \in Inst(l)} \{\{\bot_i\}\}\Big\} \\
Cut_0 = & (\bot_1, \ldots, \bot_n)
\end{aligned}
$$

Remark 6.5 (Initial ready set)
The instance heads are not unwound explicitly, but are rather the starting point of the algorithm. The simultaneous classes, which are initially enabled are consequently those, which have only instance head in conjunction with local invariant start atoms as prerequisites. Recall that the only atoms allowed in simultaneous regions at the instance head are local invariant starts; timer set atoms and timing intervals are permitted as well, cf. chapter 7. The instance heads themselves are accordingly excluded from the ready set by removing the empty set from the computed prerequisites. ■

Starting with the initial phase the construction of the automaton considers every phase and computes the fired set(s) for it. For each phase a state is generated in the automaton and for each fired set of the current phase the successor phase is computed and the corresponding state is generated. For each fired set a transition is inserted, which is annotated with the simultaneous class(es) of the fired set. The successor phases for a phase is computed by the function $Step(\cdot, \cdot)$:

6.10 Definition (Step)

$$Step : Phases(l) \times Firedsets(l) \longrightarrow Phases(l)$$

$$Step(Phase_i, Fired_{i_k}) = Phase_j,$$

where

$$
\begin{aligned}
History_j &:= History_i \cup Fired_{i_k} \\
Ready_j &:= \Big\{ scl \in SimClasses(l) \setminus \Big\{ \bigcup_{i \in Inst(l)} \{\{\top_i\}\} \Big\} \mid \\
&\qquad \forall scl' \in prerequisite(scl) : scl' \in History_j \wedge scl \notin History_i \Big\} \\
Cut_j &:= (cl'_1, \ldots, cl'_n),
\end{aligned}
$$

with

$$
cl'_t = \begin{cases} cl''_t & \exists\ f \in Fired_{i_k}\ \exists\ scl \in f : cl''_t \in scl \\ cl_t & else \end{cases}, t = 1, \ldots, n
$$

■

Remark 6.6 (Empty fired set)
The $Step(\cdot,\cdot)$ function returns the identity for the empty fired set, which corresponds to a self loop in the automaton. ■

The $Step(\cdot,\cdot)$ function is applied to all phases until the entire LSC body has been unwound, i.e. until the final phase is reached:

$$
Phase_{final} = (Ready_{final}, History_{final}, Cut_{final}),
$$

with

$$
\begin{aligned}
Ready_{final} &= \emptyset \\
History_{final} &= SimClasses(l) \setminus \Big\{ \bigcup_{i \in Inst(l)} \{\{\top_i\}\} \Big\}
\end{aligned}
$$

Remark 6.7 (Final phase)
The final ready set is always empty, since those simultaneous classes, which consist of instance end atoms have been excluded from the ready set in definition 6.10.
The final cut contains those simultaneous classes, which include the penultimate atom for each instance. ■

EXAMPLE 6.6 (PRIMITIVE UNWINDING STRUCTURE)
Application of the $Step(\cdot,\cdot)$ function to the LSC in figure 6.4 on page 107 results in the (incomplete) automaton shown in figure 6.5 on page 114. The sets of the corresponding phases are listed in the following:

- q_0 :
  - $History_0 = \{\{\{\bot_{Inst1}\}\}, \{\{\bot_{Inst2}\}\}, \{\{\bot_{Inst3}\}\}, \{\{\bot_{Inst4}\}\}\}$
  - $Cut_0 = (\{\bot_{Inst1}\}, \{\bot_{Inst2}\}, \{\bot_{Inst3}\}, \{\bot_{Inst4}\})$
  - $Ready_0 = \{\{\{?msg1\}, \{!msg1, !msg2\}, \{?msg2\}\}\}$
- q_1 :
  - $History_1 = History_0 \cup \{\{\{?msg1\}, \{!msg1, !msg2\}, \{?msg2\}\}\}$
  - $Cut_1 = (\{?msg1\}, \{!msg1, !msg2\}, \{?msg2\}, \{\bot_4\})$
  - $Ready_1 = \{\{\{!msg3\}, \{?msg3\}\},\ \{\{?msg4\}, \{!msg4\}\}\}$
- q_2 :
  - $History_2 = History_1 \cup \{\{\{!msg3\}, \{?msg3\}\}\}$
  - $Cut_2 = (\{!msg3\}, \{?msg3\}, \{?msg2\}, \{\bot_4\})$
  - $Ready_2 = \{\{\{?msg4\}, \{!msg4\}\}\}$
- q_3 :
  - $History_3 = History_1 \cup \{\{\{!msg4\}, \{?msg4\}\}\}$
  - $Cut_2 = (\{?msg4\}, \{!msg1, !msg2\}, \{?msg2\}, \{!msg4\})$
  - $Ready_2 = \{\{\{?msg3\}, \{!msg3\}\}\}$
- q_4 :
  - $History_4 = History_2 \cup \{\{\{!msg4\}, \{?msg4\}\}\}$
  - $Cut_4 = (\{?msg4\}, \{?msg3\}, \{?msg2\}, \{!msg4\})$
  - $Ready_4 = \{\{\{!msg5\}, \{?msg5, cond1\}\}\}$
- q_5 :
  - $History_5 = History_1 \cup$
    $\{\{\{!msg3\}, \{?msg3\}\}, \{\{!msg4\}, \{?msg4\}\}\}$

  - $Cut_5 = (\{!msg3, ?msg4\}, \{?msg3\}, \{?msg2\}, \{!msg4\})$
  - $Ready_5 = \{\{\{!msg5\}, \{?msg5, cond1\}\}\}$

- q_6 :
  - $History_6 = History_3 \cup \{\{\{!msg3\}, \{?msg3\}\}\}$
  - $Cut_6 = (\{!msg3\}, \{?msg3\}, \{?msg2\}, \{!msg4\})$
  - $Ready_6 = \{\{\{!msg5\}, \{?msg5, cond1\}\}\}$

- q_7 :
  - $History_7 = History_4 \cup \{\{\{!msg5\}, \{?msg5, cond1\}\}\}$
  - $Cut_7 = (\{?msg4\}, \{!msg5\}, \{?msg5, cond1\}, \{!msg4\})$
  - $Ready_7 = \emptyset$

- q_8 :
  - $History_8 = History_5 \cup \{\{\{!msg5\}, \{?msg5, cond1\}\}\}$
  - $Cut_8 = (\{?msg4\}, \{!msg5\}, \{?msg5, cond1\}, \{!msg4\})$
  - $Ready_8 = \emptyset$

- q_9 :
  - $History_9 = History_6 \cup \{\{\{!msg5\}, \{?msg5, cond1\}\}\}$
  - $Cut_9 = (\{?msg4\}, \{!msg5\}, \{?msg5, cond1\}, \{!msg4\})$
  - $Ready_9 = \emptyset$

A closer look at the automaton reveals some redundant states and transitions: The phases for states q_7, q_8, q_9 are identical and could therefore be represented by a single state. The same is true for the phases for states q_4, q_5, q_6, with the exception that their cuts differ. The cuts are not directly relevant for the $Step(\cdot, \cdot)$ function, thus these states could be merged into one as well. The rule for being able to combine states is thus that they have identical history and ready sets.

Note that identifiers rather than message labels have been used for the transition annotations in the automaton in figure 6.5, since all messages in the LSC are instantaneous and hot. The correct annotation for the transition

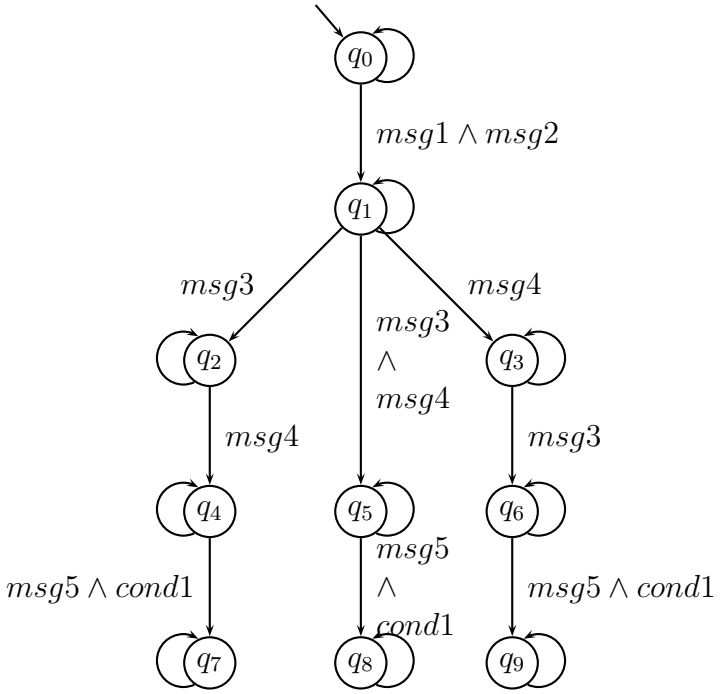


Figure 6.5: Incomplete automaton for LSC `Sim&Coregs Example`

from q_1 to q_2 would for instance be $!msg3 \wedge ?msg3$. In the remainder of this thesis we will use the abbreviation by message identifier for hot instantaneous messages. ■

In order to generate more concise automata, a new state will only be generated, if no equivalent state exists already. Even with this optimization the automaton is still incomplete, since e.g. the self loops carry no annotation. This and other issues, which influence the completion of the automaton, are discussed in the following sections.

6.2.2 Self Loop Annotation

The unwinding structure as it is so far generated by the $Step(\cdot, \cdot)$ function provides annotations only for transitions between different states. The self

loops which exist for every state are not annotated, since no simultaneous classes are unwound. An empty annotation corresponds to *true* and thus makes the resulting automaton highly non-deterministic.

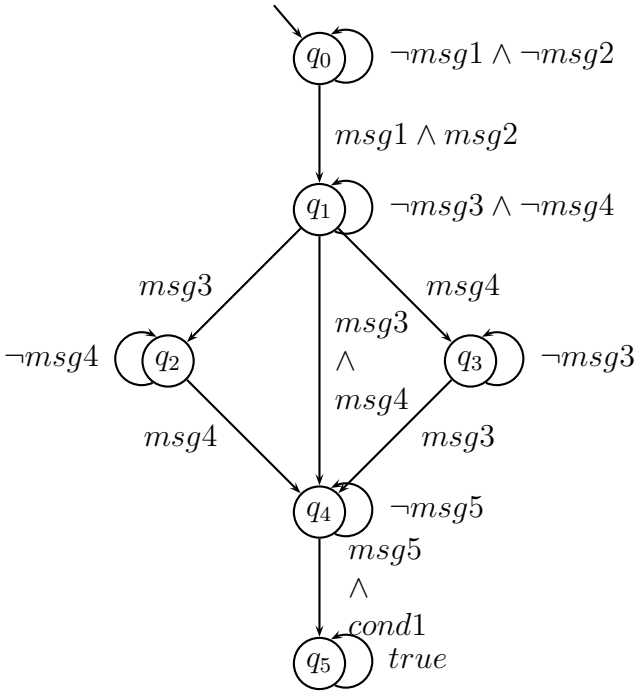


Figure 6.6: Incomplete, optimized automaton for LSC `Sim&Coregs Example` using weak interpretation

The question of how to annotate the self loops is influenced by the question, if duplicate messages are allowed, i.e. should it be considered an error, if a message, which is contained in an LSC body, is observed more than once during the activation of the LSC? Is e.g. the crossing in figure 4.9 on page 78 permitted to send the message `safe_snd` twice? The interpretation assumed in [DH01] is that duplicate messages result in an error and are thus prohibited. Applying this *strict* interpretation to the incomplete automaton generated so far results in an automaton, where every self loop is annotated with the conjunction of the negation of all messages of the LSC; all non-self-loop messages similarly need to be extended by the conjunction of the

negation of all messages of the LSC, which are not unwound in the current step. It is not immediately clear, if this interpretation is too restrictive.

This motivates a second interpretation, which we call *weak* and which allows duplicate messages as long as the messages contained in the LSC are observed also at the correct times. This corresponds to annotating the self loops in the automaton only with the conjunction of the negation of all messages, which are in the ready set associated with the current state.

Both interpretations remove the non-determinism in the automaton due to the empty self loop annotation (see example 6.7). The annotation for the self loop of the final state is identical for both variants: $true$, because once the behavior specified in the LSC has been observed entirely no further restriction applies to the system. Since the choice of an adequate interpretation of an LSC also depends on the specific field of application, we will return to this issue when evaluating the LSC language in chapter 11.

Example 6.7 (Interpretations)
Figure 6.6 and 6.7 show the weak, resp. strict interpretation for the (optimized) automaton for LSC `Sim&Coregs Example`. For readablilty's sake the self loop annotation for the strict alternative is shown as sc, which stands for $\neg msg1 \wedge \neg msg2 \wedge \neg msg3 \wedge \neg msg4 \wedge \neg msg5$.

■

Figure 6.6 on the preceding page exemplifies that in case of the weak interpretation, special care has to be taken with concurrently enabled SimClasses, which result from either a coregion or elements, which are completely independent in the LSC body, e.g. two conditions consisting of only one condition atom and being defined on separate instances. As illustrated by the annotations of the transitions leaving state q_1 in figure 6.6, the annotations of the transitions leaving q_1 are not mutually exclusive, i.e. there is the possibility of non-determinism. Assume q_1 has been reached and $msg3$ and $msg4$ are observed simultaneously. The next state could be any of q_2, q_3 or q_4. In order to remove this source of non-determinism the transitions, which do not contain all elements of the ready set, need to be additionally annotated with the negation of the remaining elements. In figure 6.6 this means that the transition from q_1 to q_2 is annotated with $msg3 \wedge \neg msg4$ and the one to q_3 with $msg4 \wedge \neg msg3$. This is not necessary for the strict interpretation, since it already forbids the occurrence of *all* other messages of the LSC.

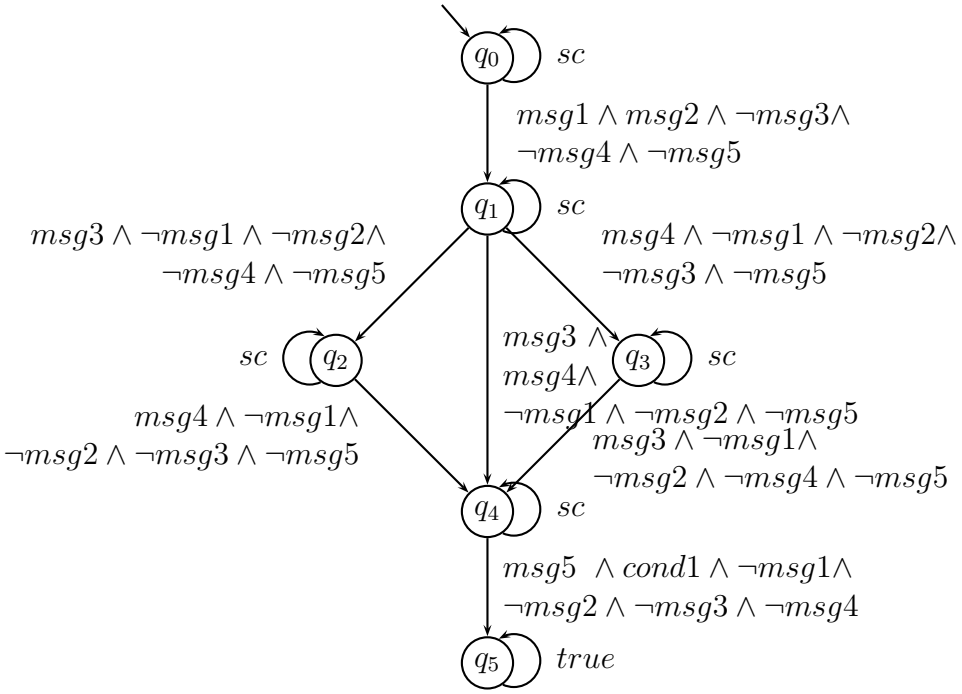


Figure 6.7: Incomplete, optimized automaton for LSC `Sim&Coregs Example` using strict interpretation

6.2.3 Location Temperatures

So far the location temperatures, i.e. the progress requirements on the instance axes, have not been considered. This information is collected by the cuts, which indicate up to which locations the unwinding has progressed. The local liveness information of each instance, given by the location temperature, is used to compute the global liveness requirements, which is expressed by the cut temperature defined below.

6.11 Definition (Cut Temperature)

$$cut\_temp : Cuts(l) \longrightarrow \{hot, cold\}$$
$$cut\_temp(cut) := \begin{cases} hot & \text{if } \exists\ cl_j \in cut : temp(location(cl_j)) = hot \\ cold & \text{else} \end{cases},$$

for $cut = (cl_1, \ldots, cl_n), 1 \leq j \leq n$. ■

Thus phases containing a hot cut mean that the corresponding states in the automaton have to be left within a finite amount of time — analogously to hot locations on an instance. A cold cut indicates that the corresponding state need not be left. In conjunction with the self loop this means that such a state is fair in the sense of the Büchi acceptance criterion. Note that the final and the exit state (see below) are always fair, since no further requirements are posed by the LSC once it has been completed or exited.

6.2.4 Semantics of Conditions

Up to now conditions are only partly treated by the unwinding procedure. The unwinding so far only covers the case that the condition is fulfilled, violations are not considered; cf. figures 6.6 on page 115 and 6.7 on the preceding page. For mandatory conditions no further steps are necessary, since evaluating a condition to false results in a non-accepting run (there exists no transition in the automaton for this case). As illustrated by the automata in figures 6.6 and 6.7 condition `Cond1` must be satisfied, when message `msg5` occurs. For possible conditions, whose violations should result in an exit from the LSC, a special *exit state* is introduced, which is entered whenever a possible condition is violated. This state is similar to the final state inasmuch as no further restrictions apply after entering this state, i.e. the self loop annotation is *true* as well; see figure 6.9(a) on page 120 for an example[3].

All conditions, possible and mandatory ones, are disregarded by both the weak and the strict interpretation discussed above, because they are only required to hold at one specific point in time; other points in time are not constrained. Since conditions typically retain their value for a span of time,

[3]Exit and final state can effectively be merged as a further optimization of the symbolic automaton. In the remainder we will nevertheless use separate exit states in order to increase readability.

it is undesirable to forbid the evaluation to true at other times than the specified one. Conditions are thus not treated like messages, i.e. they are not included in the self loop annotations, etc.

The fact that conditions usually retain their value for more than one point in time gives rise to the question, when a condition should be evaluated. This issue is tied to the annotation of the concerned self loops and is covered in the remainder of this section, starting with possible conditions.

Evaluation of Possible Conditions

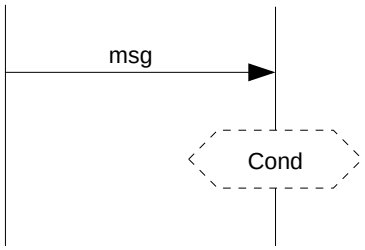
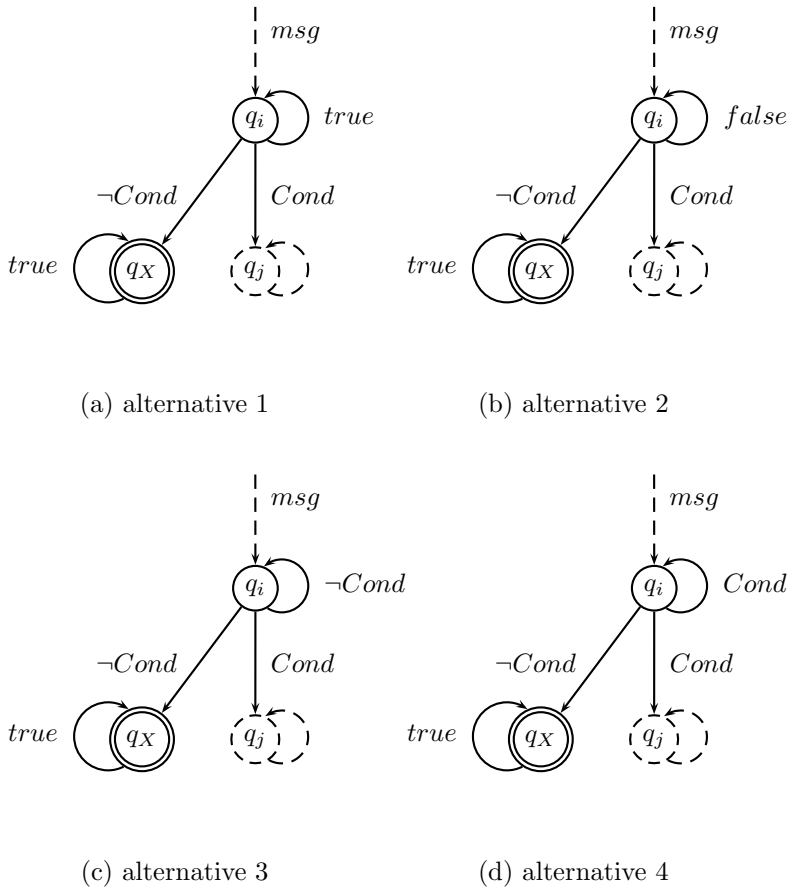


Figure 6.8: LSC fragment for the evaluation of possible conditions

If a condition is bound to one or more messages via a simultaneous region, as is e.g. the case in the LSC in figure 6.1 on page 97, the evaluation point is clear: it is given by the occurrence time of the message(s). The same is true if the condition is bound to a timeout or end of a timing interval (cf. chapter 7). If the condition is isolated, i.e. not part of a simultaneous region or contained in a simultaneous region with no messages, however, it is not clear when it should be evaluated. The point of evaluation is influenced by the annotation of the self loop at the state(s), where the condition is enabled, i.e. contained in the ready set.

For the illustration of the different alternatives we use the LSC fragment shown in figure 6.8. Figures 6.9(a) - 6.9(d) show automata fragments demonstrating the possible solutions, dashed parts indicating glue points to the remaining parts of the automaton. The automata fragments differ only in the self loop annotation.

Figure 6.9(a) shows the automaton fragment, which is generated by the algorithm so far (assuming weak interpretation). The self loop does not carry any annotation, which is equivalent to *true*, since no message is involved in the unwinding of `Cond`. Regardless of the truth value the automaton may



(a) alternative 1

(b) alternative 2

(c) alternative 3

(d) alternative 4

Figure 6.9: Alternatives for the evaluation point for possible conditions

remain in state q_i, the automaton is thus non-deterministic and the evaluation point arbitrary (once `msg` has been observed).

A second possible annotation of the self loop is shown in figure 6.9(b). It is the converse of the first alternative and forces the immediate evaluation of the condition, since the automaton must not remain in q_i. This automaton fragment is deterministic. The third possibility depicted in figure 6.9(c) annotates the self loop with the negation of the condition and once more yields a non-deterministic automaton. This alternative forces the automaton to leave q_i and reach q_j when the condition is evaluated to true. As long as the condition does not hold there is a non-deterministic choice between staying

in q_i and moving into the exit state q_X. The fourth alternative is illustrated by figure 6.9(d) and is the converse of the preceeding solution annotating the self loop with the condition itself. This solution is non-deterministic as well. Before assessing the alternatives and choosing the default solution for the automaton construction we consider the above possibilities in conjunction with mandatory conditions.

Evaluation of Mandatory Conditions

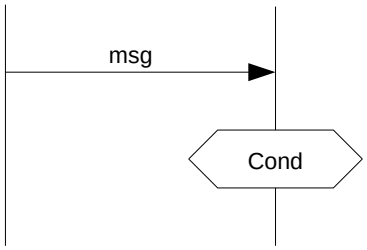
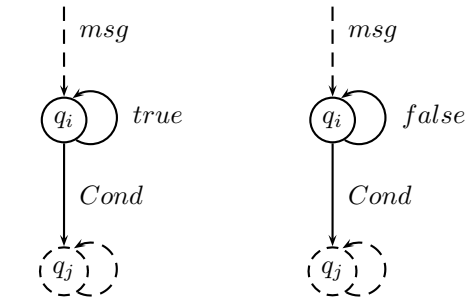


Figure 6.10: LSC fragment for the evaluation of mandatory conditions

The situation is slightly different for mandatory conditions. Figure 6.10 shows the LSC fragment for a mandatory condition, figures 6.11(a) - 6.11(d) show the automata fragments for the different alternatives. Since `Cond` now is a mandatory condition, there is no exit state for these fragments. The consequences only change for alternative 3 (figure 6.11(c)), the other alternatives are essentially not concerned by the altered mode. Solution 3 differs inasmuch as the resulting automaton is no longer non-deterministic as for a possible condition. This solution now prohibits to detect a violation of the condition, since an evaluation of `Cond` to false only results in taking the self loop.[4] Therefore alternative 3 is no viable solution.

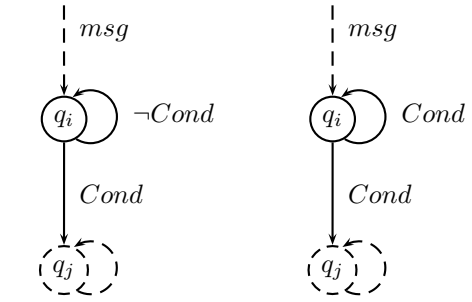
The remaining alternatives are not ideal solutions either: Possibility 1 and 4 produce non-deterministic automata, which should be avoided, if possible; alternative 2 is too restrictive. The optimal solution is to unambiguously determine the point of evaluation by tying the condition to a reference point. For the moment the only reference point available are message atoms, which are grouped with the condition in a simultaneous region as illustrated e.g. in

[4]The only violation, which could be detected, is a violation of the progress requirement, if the cut corresponding to state q_i is hot and the condition never evaluates to true. This is independent of the condition mode, however.



(a) alternative 1

(b) alternative 2



(c) alternative 3

(d) alternative 4

Figure 6.11: Alternatives for the evaluation point for mandatory conditions

figure 6.1 on page 97 and the associated automata in figures 6.6 on page 115 and 6.7 on page 117. Timeout atoms and end points of timing intervals introduced in chapter 7 qualify as reference points as well. We thus strongly recommend the provision of such a reference point.

For conditions without a reference point alternative 1 is the best solution, even though it results in a non-deterministic automaton. It allows the most flexibility, however, since all other situations described above can be

expressed by using appropriate local invariants, etc. This is not the case for the other alternatives.

Remark 6.8 (Tying possible conditions to messages)
If a possible condition has a reference point in the form of a message, this message must also be included in the annotation of the transition, which leads into the exit state. Assume for instance that the mode of `Cond1` *in the LSC in figure 6.1 were possible. Then there would be a transition from state q_4 to the exit state annotated by "$\neg Cond1 \wedge msg5$" in the automata in figures 6.6 and 6.7.* ■

6.2.5 Message Temperatures

So far message temperatures have been assumed to be hot, for which the semantical treatment is clear: if the send atom has been observed, the receive atom must also be observed. The intuitive semantics of a cold message is that it need not be received. For instantaneous messages the situation is clear: Since sending and receipt happen simultaneously, only the send atom is unwound. For asynchronous messages, however, there exist different options how to formally define the semantics:

- **Alternative 1:** The receiving instance is not allowed to progress beyond the receiving atom for this message, i.e. no atoms below the receiving atom may be observed before the message arrives.

- **Alternative 2a:** The receiving instance is allowed to progress beyond the receiving atom without actually receiving the message. If the message does arrive later on, the LSC is exited without generating an error, similar to a violated cold condition.

- **Alternative 2b:** The receiving instance is allowed to progress beyond the receiving atom without actually receiving the message. If the message does arrive later on, this constitutes an error, similar to a mandatory condition.

- **Alternative 2c:** The receiving instance is allowed to progress beyond the receiving atom without actually receiving the message and a late arrival of the message is ignored.

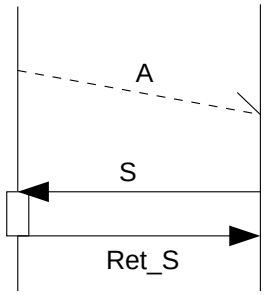


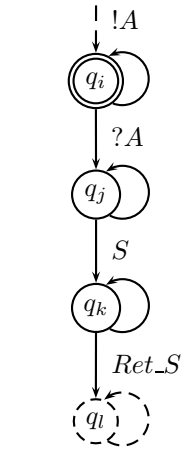
Figure 6.12: LSC fragment for the illustration of different alternatives for cold asynchronous messages

Figure 6.12 shows an LSC fragment with a cold asynchronous message and figure 6.13 on the facing page illustrates the different alternatives (disregarding self loop annotations for readability). Alternative 1 means that the state after receipt of `A` becomes a fair state regardless of the temperature of the associated cut; see figure 6.13(a). Thus message `A` blocks the remainder of the LSC, if it is never received.

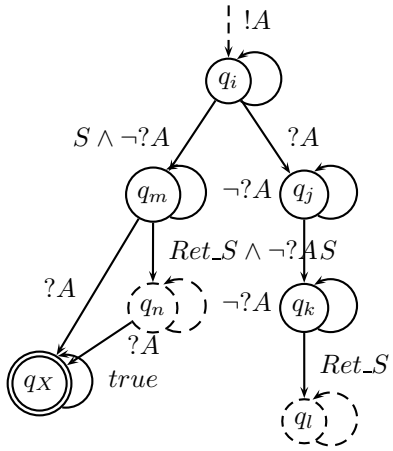
Alternative 2a (illustrated in figure 6.13(b)) necessitates a duplication of part of the automaton structure, since for all elements following after the cold message receipt it is necessary to know, if the `A` has been received, i.e. if a late receipt of `A` has to result in an exit or not. If `S` occurs before the receipt of `A`, the left branch in the automaton is taken, and if then `A` arrives belated, this results in a transition to the exit state. In order to guarantee this behavior all transitions, including self loops, of the left-hand branch of the automaton in figure 6.13(b) have to be extended by $\neg?A$.

Alternative 2b is similar to 2a (see the automaton in figure 6.13(c)), with the exception that no transitions to the exit state are required, since late arrivals of a cold asynchronous message constitute an error. The branch in the automaton is necessary for this alternative as well, since a late arrival of `A` must be detectable.

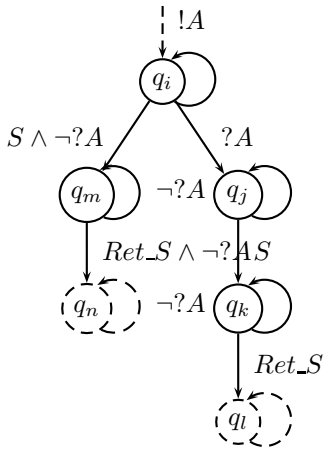
The last alternative (see the automaton in figure 6.13(d)) allows either to observe the cold asynchronous message receipt and the succeeding atom in the specified order or to move on if the succeeding atom is observed before the cold message arrives. In the latter case a late arrival of the cold message is not treated in a special way as by the two preceeding options. This is achieved by inserting an additional transition starting at the state where the receipt is awaited and ending in the state, where the succeeding element has been un-



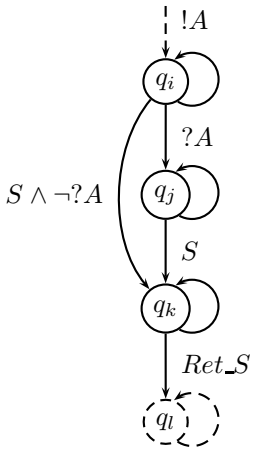
(a) automaton for alternative 1



(b) automaton for alternative 2a



(c) automaton for alternative 2b



(d) automaton for alternative 2c

Figure 6.13: Automata fragments for the different alternatives for cold asynchronous messages

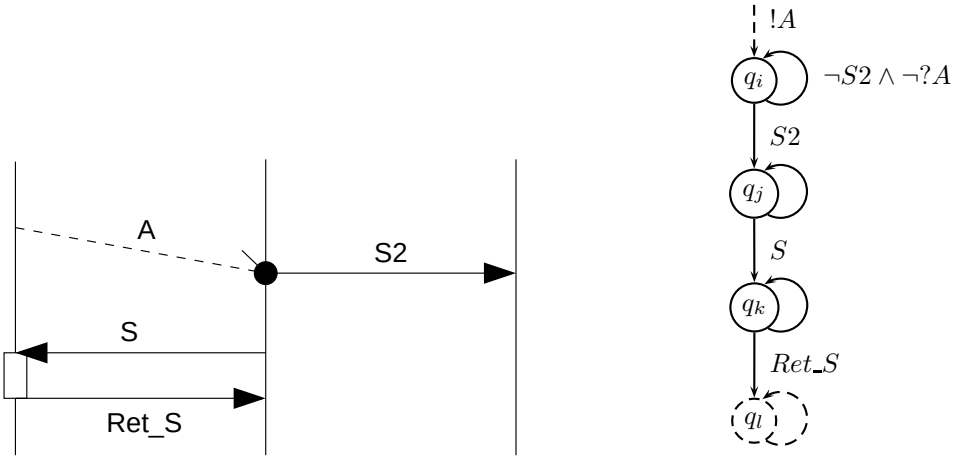
wound (cf. figure 6.13(d)), thus skipping the the cold asynchronous message receipt. The transition annotation is identical to the one of the succeeding transition. For the weak interpretation the skipping transition additionally has to be annotated with the negation of the asynchronous receipt in order to avoid non-determinism.

Alternatives 2a and 2b interfere with the interpretation: The weak interpretation, which explicitly allows duplicate messages, is implicitly overridden by restricting the occurrence of late cold asynchronous message receipt. The strict interpretation already forbids other occurrences of $?A$ than the ones specified in the LSC. In this case the consequence of alternative 2a is an implicit weakening of the strict interpretation due to additional exits. Such implicit and hidden effects must be avoided, since they result in unintuitive charts. Both options are better expressed explicitly by using appropriate possible and mandatory local invariants.

We hence choose the last alternative (2c), because it is the only solution, which is completely orthogonal to other LSC elements and concepts; all other alternatives are expressible by other LSC elements and concepts. Fair states in the automaton are determined by the cut temperature, i.e. location temperatures. Alternative 1 would introduce a second, redundant and potentially contradictory way to specify a cold cut, which overrides the location temperatures. This situation can and should be expressed by a hot asynchronous message and cold cut, which contains the receipt. Solution 2c also avoids the duplication of parts of the automaton, which is necessary for options 2a and 2b yielding a more concise automaton yielding a more concise automaton.

Cold Asynchronous Messages in Simultaneous Regions

The previous discussion has only taken into account single cold asynchronous message receipts, but not those which are part of a simultaneous region. The latter case comprises two sub-cases: the only message atoms in the simultaneous region are cold asynchronous message receipts, or there are also other message atoms present. The first sub-case is slightly more restrictive than for isolated receipts, because either *all* cold asynchronous messages contained in the simultaneous region are received at the same time, or *none* arrives. The semantics thus is: If one message arrives, all other messages must arrive simultaneously as well. If not all messages are observed simultaneously, the consequences depend on the interpretation. This case is already covered



(a) LSC fragment

(b) corresponding automaton fragment

Figure 6.14: Cold asynchronous message receipt within a simultaneous region

by the procedure for isolated receipts described above.

The other sub-case requires a different treatment, since those message atoms, which are not cold asynchronous receipts, need to be observed and thus may not be skipped. Consider e.g. the LSC fragment in figure 6.14(a): Even though `A` need not be received, `S2` must be sent before `S` may be sent. Thus q_j — in contrast to the situation in figure 6.13(d) — must not be skipped, if `S` is observed before the receipt of `A`. If `S2` occurs, either `A` arrives simultaneously or not, thus the automaton in figure 6.14(b) for the LSC fragment advances from q_i to q_j on observing `S2`, regardless of the receipt of `A`, because:

$$(S2 \wedge \neg ?A) \vee (S2 \wedge ?A) \equiv (S2 \wedge (\neg ?A \vee ?A)) \equiv \ S2$$

After observing `S2` the arrival of `A` is not restricted (unless done explicitly by an appropriate local invariant or the strict interpretation is used). Note however that it is an error, if `A` arrives before `S2` is observed. If it is received, it must arrive simultaneously with the sending of `S2`. This is expressed by retaining $?A$ in the stable condition of q_i.

6.2.6 Local Invariants

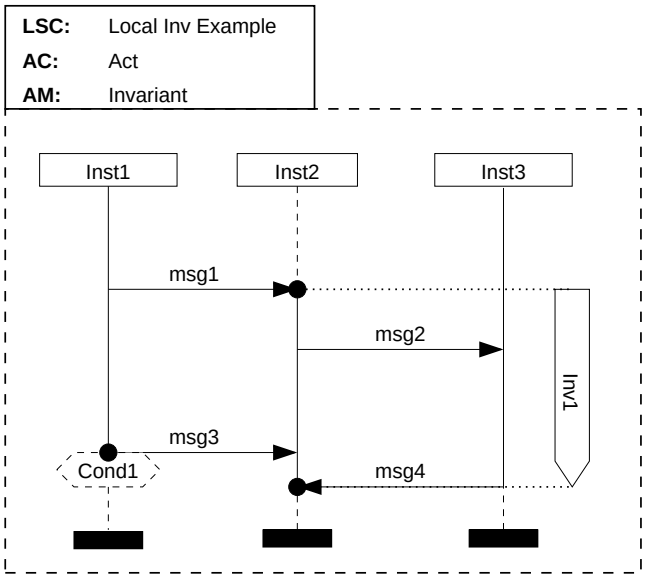


Figure 6.15: LSC with local invariant and condition

The treatment of local invariants is similar to conditions, only the longer duration has to be handled differently. For mandatory local invariants this means that all transitions, including self loops, between the unwinding of the local invariant start and the local invariant end have to be annotated by the corresponding identifier. Possible local invariants additionally require to provide transitions to the exit state from all states, which are covered by the local invariant. Each such transition is annotated with the negation of the local invariant identifier.

Example 6.8
Figure 6.16 shows the automaton for the LSC in figure 6.15 using the weak interpretation for better readability. Since the start of `Inv1` includes its reference point (message `msg1`), the first transition to be annotated with the local invariant is the one from q_1 to q_2. The end does not include its reference point (message `msg4`), thus the self loop on state q_4 is the last transition, which is annotated. If the start had not been included, the first transition to be annotated would have been the self loop on q_2. If the end had been included, the last transition annotated with the local invariant would have been the one from q_4 to q_5.

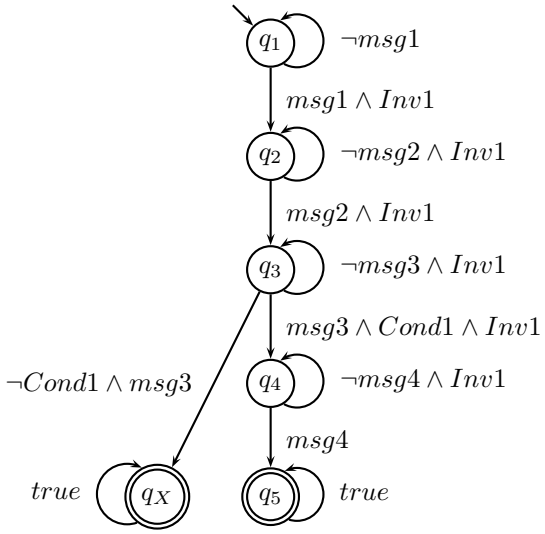


Figure 6.16: Automata for mandatory local invariant in LSC `Local Inv Example` (weak interpretation)

The automaton in figure 6.17 illustrates the case, where the mode of `Inv1` is possible. From every state within the scope of `Inv1` a transition to the exit state is added and annotated with the negation of the local invariant. Note that start and end require special attention. The transition from q_1 to the exit state needs to be annotated with the message corresponding to `Inv1`'s reference point. This is necessary because the local invariant is active[5] only once `msg1` has been observed. Otherwise it would be possible to enter the exit state before `msg1` occurred. Similarly must the annotation of the transition from q_4 to the exit state be extended by the negation of the reference point, because the scope of `Inv1` does not include `msg4`. A violation of `Inv1` thus is only relevant before `msg4` is observed. Excluding `msg1` from the scope of `Inv1` results in omitting the transition from q_1 to the exit state. Inclusion of `msg4` results in annotating the transition from q_4 to the exit state by $\neg Inv1$ only. ■

[5]We call a local invariant *active*, if its start atom has been unwound, but not its end atom.

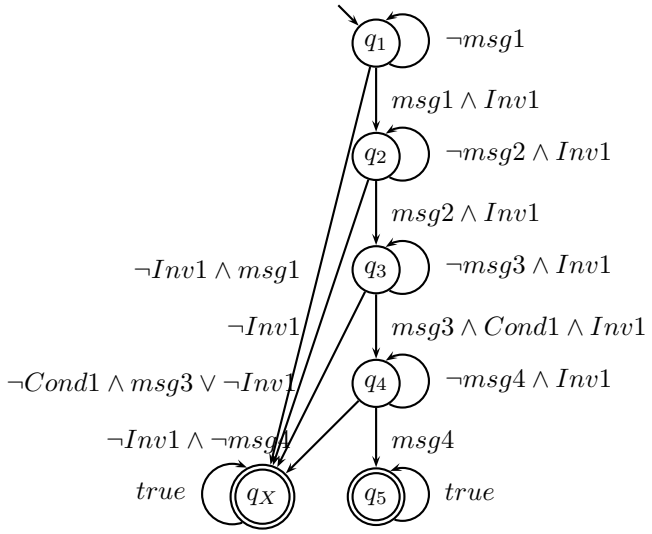


Figure 6.17: Automata for possible local invariant in LSC `Local Inv Example` (weak interpretation)

Condition and Local Invariant Priorization

Figures 6.15, 6.16 and 6.17 also illustrate the interplay of local invariants and conditions. The LSC in figure 6.15 contains a potential conflict between the possible condition `Cond1` and the mandatory local invariant `Inv1`: If `Cond1` is evaluated to $false$ and at the same time `Inv1` is violated, should this be considered an error or should the LSC be exited? The same problem exists for the converse situation, when both a mandatory and possible condition are present, or when both a mandatory and possible local invariant are active.

Since possible conditions and local invariants are intended to explicitly weaken the requirement specified in the LSC, we prioritize the violation of possible conditions or local invariants over mandatory ones. In the automaton in figure 6.16 this is expressed by the annotation of the transition to the exit state, which is taken, if the condition is violated, regardless of the evaluation of the local invariant.

6.2.7 The Unwinding Algorithm

The unwinding algorithm needs as input an LSC body l, including the sets and functions defined in the preceding sections, and the interpretation (weak or strict). The result is a timed symbolic automaton $\mathcal{TSA} = (\Sigma, Q, q_0, C, \longrightarrow, F)$. Since no timing information is considered at this point, the set of clocks is empty for the moment and all transitions will carry no clock reset or constraints. Some auxiliary functions are needed, which are indicated by small capitals font and whose behavior is described below. Algorithm 6.1 shows the main routine of the unwinding procedure.

Algorithm 6.1, lines 1 – 16: Initialization First the stable condition *sc* is constructed in line 2, which is only possible beforehand, if the interpretation is strict, because only then are all stable conditions identical[6]. The auxiliary function NEG_CONJUNCT takes a set of identifiers and constructs the conjunction of the negation of all elements of this set. If the set contains only one element, only the negation of this element is returned. Thus NEG_CONJUNCT($\{msg1, msg2\}$) produces $\neg msg1 \wedge \neg msg2$, NEG_CONJUNCT($\{msg1\}$) yields $\neg msg1$.

The set of properties, from which the transition annotations are built, is made up of the identifiers used in the LSC (line 4). Note that message labels are used instead of message identifiers, because for asynchronous messages there need to be two annotations for each message, one for the sending and one for the receipt. In addition to the identifiers also constants $true$ and $false$ are allowed. The set of clocks is empty (line 5). *phases* in line 6 is the set of phases and is initialized to the initial phase $Phase_0$ (cf. definition 6.9 on page 109), which also becomes the first state in Q and the initial state (lines 7 and 8). Function STATE establishes the relation between automaton states and phases by assigning a unique state name to a phase, e.g. q_0 for the initial phase. If the phase has already been assigned a name, the existing one is returned. If the initial cut is cold, then the set of fair states is initialized with Q (line 10), which at this point only contains q_0, otherwise F is empty (line 12). Note that the exit node is always included to keep the algorithm simple. In practice it is only added when needed. The transition relation is initialized with the self loop of the exit state in line 14.

[6]The stable conditions may be not identical in the end due to local invariants, but the base remains the same.

Algorithm 6.1 Unwinding Algorithm

1: **if** $interpretation$ = strict **then**
2: $sc :=$ NEG_CONJUNCT($MsgLabels(l)$)
3: **end if**
4: $\Sigma := MsgLabels(l) \cup Conditions(l) \cup Local\_Invariants(l) \cup \{true, false\}$
5: $C := \emptyset$
6: $phases := \{Phase_0\}$
7: $Q := \{\text{STATE}(Phase_0), q_X\}$
8: $q_0 :=$ STATE($Phase_0$)
9: **if** $cut\_temp(Cut_0) = cold$ **then**
10: $F := Q$
11: **else**
12: $F := \{q_X\}$
13: **end if**
14: $\longrightarrow := \{(q_X, true, \emptyset, \epsilon, q_X)\}$
15: $poss\_invs_0 := \{li \in Local\_Invariants(l) \mid \exists scl\, \exists cl \in scl\, \exists a, a' \in cl :$
$a \in Instheads(l) \wedge a' \in LI\_starts(l) \wedge liID(a') = li \wedge mode(a') = possible\}$
16: $mand\_invs_0 := \{li \in Local\_Invariants(l) \mid \exists scl\, \exists cl \in scl\, \exists a, a' \in cl :$
$a \in Instheads(l) \wedge a' \in LI\_starts(l) \wedge liID(a') = li \wedge$
$mode(a') = mandatory\}$
17: **while** $phases \neq \emptyset$ **do**
18: **let** $ph = (Ready_i, History_i, Cut_i) \in phases$
19: **if** $Ready_i = \emptyset$ **then**
20: $F := F \cup \{\text{STATE}(ph)\}$
21: $\longrightarrow := \longrightarrow \cup \{(\text{STATE}(ph), true, \emptyset, \epsilon, \text{STATE}(ph))\}$
22: **else**
23: GENERATE_EXITS($Ready_i, ph$)
24: **for all** $Fired_{i_k} \in \mathcal{P}(Ready_i)$ **do**
25: $successor := Step(ph, Fired_{i_k})$
26: **let** $successor = (Ready_j, History_j, Cut_j)$
27: **if** $successor = ph$ **then**
28: INSERT_SELF_LOOP($ph, Ready_i$)
29: **else**
30: $TransitionAnnot :=$CONSTRUCT_TRANSITION($ph, successor,$
$Ready_i, Fired_{i_k}$)
31:
32: **if** $\exists q \in Q$ *with* $\text{STATE}^{-1}(q) = (Ready_x, History_x, Cut_x) : Ready_i = Ready_x \wedge History_i = History_x$ **then**
33: $successor := \text{STATE}^{-1}(q)$
34: **else**
35: $Q := Q \cup \{\text{STATE}(successor)\}$
36: $phases := (phases \setminus \{ph\}) \cup \{successor\}$
37: **if** $cut\_temp(Cut_i) = cold$ **then**
38: $F := F \cup \{\text{STATE}(ph)\}$
39: **end if**
40: **end if**
41: $\longrightarrow := \longrightarrow \cup$
$\{(\text{STATE}(ph), TransitionAnnot, \emptyset, \epsilon, \text{STATE}(successor))\}$
42: **end if**
43: INSERT_SKIP_TRANSITION($TransitionAnnot, ph, successor$)
44: **end for**
45: **end if**
46: **end while**

Two sets are associated with each phase $Phase_i$: $poss\_invs_i$ and $mand\_invs_i$, which contain the identifiers of all possible, resp. mandatory local invariants, which are active in this phase. The initial sets $poss\_invs_0$ and $mand\_invs_0$ are initialized with those local invariants, whose start atoms are attached to an instance head (lines 15 and 16).

Algorithm 6.1, lines 18 – 47: Main Part The remainder of algorithm 6.1 iterates over the list of accumulated phases and computes the successors for each by applying the step function of definition 6.10 on page 110. As long as the list of phases is not empty (line 17) a phase ph is taken from the list (line 18) and unwound.

Algorithm 6.1, lines 19 – 21: Final State Line 19 checks, if the current phase is the final phase, which is indicated by an empty ready set. The final phase need not be unwound using the step function, since it does not have a successor. Thus it is sufficient to add the name corresponding to the phase to the set of fair states[7] (line 20) and to insert the *true*-annotated self loop to the transition relation (line 21). Note the empty clock reset list and clock constraint.

Algorithm 6.1, lines 23 – 46: Computing Successors If the current phase is not the final one, the step function needs to be applied for all fired sets $Fired_{i_k}$ computed for ready set $Ready_i$ (lines 24 – 26). Before the successors are computed the transition to the exit state is generated, if needed, in line 23 by the function GENERATE_EXITS, which is shown in algorithm 6.2 on page 135.

If the successor is identical to the current phase, this means that the self loop for the corresponding state must be generated, which is handled by function INSERT_SELF_LOOP (line 28) and which is described in algorithm 6.3 on page 137. If the successor phase is different from the current phase, the annotation of the transition to the successor phase is generated by function CONSTRUCT_TRANSITION in line 30; for the behavior of this function refer to algorithm 6.4 on page 138.

Lines 32 and 33 implement the optimization introduced in section 6.2.1: If there already is an equivalent state in Q, this state is used as successor (line 33), otherwise a new state is generated and inserted into Q (line 35),

[7] Recall that the final state is always fair.

the old phase is removed from the *phases* list and the new phase is appended (line 36), and the state is added to the set of fair states, if the corresponding cut is cold (lines 37, 38). The function STATE^{-1} is the inverse of function STATE, i.e. it returns the phase associated with a state in Q.

In line 41 the transition is generated and added to $\longrightarrow$. Line 43 contains the treatment for cold asynchronous message receipts, if they require the inserting of a skipping transition as described in section 6.2.5 on page 123. This is handled by function INSERT_SKIP_TRANSITION, which is described in algorithm 6.5 on page 139.

Algorithm 6.2 on the next page creates the transition to the exit state for each unwound phase, i.e. it is executed only once for each phase. It consists of three major parts: checking for possible conditions and local invariants in simultaneous regions (lines 2 – 25), handling possible conditions, which are not part of a simultaneous region (lines 27 – 29) and generating the transition to the exit state (lines 30 – 32). The annotation for the transition to the exit state is initialized to *true* in line 1.

Algorithm 6.2, lines 2 – 5: Possible Conditions in Simultaneous Regions First, the simultaneous regions contained in the current ready set are checked for possible conditions. The auxiliary function SIMCLASSES(SIMREGS($Ready_i$)) returns the set of simultaneous classes in $Ready_i$, which contain a simultaneous region, i.e. the set of clusters, which contain more than one atom each. The auxiliary function POSS_CONDIDS(*scl*) similarly returns the set of condition atom identifiers, which are contained in SimClass *scl* and whose mode is *possible*. If such identifiers exist, i.e. the set returned by POSS_CONDIDS(*scl*) is not empty (line 3), the conjunction of the negation of these identifiers is disjunctively added to the annotation of the transition to the exit state *ExitAnnot* (line 4) conjoined with the conjunction of all message identifiers contained in the same simultaneous region. The auxiliary function MSGLABELS(*scl*) returns the set of labels of messages contained in SimClass *scl*. This implements the semantics for possible conditions within simultaneous regions described in remark 6.8 on page 123: the evaluation point of a condition bound to a message via a simultaneous region is given by the occurrence of this message. The procedure described here is the generalization to a number of possible conditions and several messages contained in a simultaneous region. Note that the elements within one simultaneous region are assembled conjunc-

Algorithm 6.2 Generate Exits($Ready_i, ph$)

```
 1: ExitAnnot := false
 2: for all scl ∈ SimClasses(SimRegs(Ready_i)) do
 3:    if Poss_CondIDs(scl) ≠ ∅ then
 4:       ExitAnnot := ExitAnnot ∨
              (Neg_Disjunct(Poss_CondIDs(scl))∧
                Conjunct(MsgLabels(scl)))
 5:    end if
 6:    for all lis ∈ LIStarts(Ready_i) do
 7:       if lis ∈ scl ∧ mode(liID(lis)) = possible then
 8:          if incl(lis) then
 9:             ExitAnnot := ExitAnnot ∨ (¬liID(lis)∧
                    Conjunct(MsgLabels(scl)))
10:          else
11:             ExitAnnot := ExitAnnot ∨ ¬liID(lis)
12:          end if
13:       end if
14:    end for
15:    for all lie ∈ LIEnds(Ready_i) do
16:       if lie ∈ scl ∧ mode(liID(lie)) = possible then
17:          if incl(lie) then
18:             ExitAnnot := ExitAnnot ∨ (¬liID(lie)∧
                    Neg_Conjunct(MsgLabels(scl)))
19:          else
20:             ExitAnnot := ExitAnnot ∨ ¬liID(lie)
21:          end if
22:       end if
23:    end for
24: end for
25:
26: for all pc ∈Poss_CondIDs(Ready_i)\
    Poss_CondIDs(SimClasses(Ready_i) do
27:    ExitAnnot := ExitAnnot ∨ ¬pc
28: end for
29:
30: if ExitAnnot ≠ false then
31:    ⟶:=⟶ ∪ {(State(ph), ExitAnnot, ∅, ϵ, q_X)}
32: end if
```

tively, but that several such regions are joined disjunctively, since violations of distinct possible conditions (and local invariants) must each individually lead to an exit.

Algorithm 6.2, lines 6 – 14: Possible Local Invariant Starts The auxiliary function LISTARTS($Ready_i$) returns the set of local invariant start atoms of $Ready_i$. Only the possible ones, which are contained in the current simultaneous region, are of interest here (line 7). If they also include the simultaneous region they are bound to (line 8), the exit transition needs to be annotated not only with the negation of the local invariant, but also with the message identifiers of their simultaneous region as described in example 6.8 on page 128 (line 9). If they are non-inclusive the negation of the local invariant identifier is sufficient (line 11).

Algorithm 6.2, lines 15 – 24: Possible Local Invariant Ends A similar treatment is necessary for possible local invariant ends, since they also need a special treatment, if they include the simultaneous region they are bound to. As explained in example 6.8 on page 128 it is necessary to additionally annotate the negation of the local invariant identifier with the *negation* of the messages contained in the currently considered simultaneous region (line 18).

Algorithm 6.2, lines 26 – 28: Isolated Possible Conditions All possible conditions, which are not part of a simultaneous region (line 26) require only their disjunctive inclusion into the annotation of the exit transition (line 27).

Algorithm 6.2, lines 30 – 32: Exit Transition Generation The generation of a transition to the exit state is only necessary, if any possible condition or local invariant is present in the current ready set, indicated by a non-empty, i.e. other than $false$, $ExitAnnot$ (line 30). If this is the case, the transition to the exit state is added to the transition relation (line 31).

Algorithm 6.3 Algorithm 6.3 on the next page generates the self loop for a state and adds it to the transition relation. For simplicity's sake the two sets of active local invariants are joined in line 1. If the interpretation is strict, then the base stable condition has already been computed and only the conjunction of the active local invariants needs to be added (line 3). For the weak interpretation it has to be checked, if there are any messages in the ready set (line 5). The auxiliary function MSGLABELS is an extension of the corresponding auxiliary function defined in algorithm 6.2. If $Ready_i$ contains

Algorithm 6.3 Insert Self Loop($ph, Ready_i$)

```
 1: let invs_i = poss_invs_i ∪ mand_invs_i
 2: if interpretation = strict then
 3:    ⟶:=⟶ ∪ {(State(ph), sc ∧ Conjunct(invs_i), ∅, ϵ, State(ph))}
 4: else
 5:    if MsgLabels(Ready_i) ≠ ∅ then
 6:       ⟶:=⟶ ∪ {(State(ph),
             Neg_Conjunct(MsgLabels(Ready_i)) ∧ Conjunct(invs_i),
             ∅, ϵ, State(ph))}
 7:    else
 8:       ⟶:=⟶ ∪
          {(State(ph), true ∧ Conjunct(invs_i), ∅, ϵ, State(ph))}
 9:    end if
10: end if
```

at least one message, the conjunction of the negation of their identifiers is used for the transition annotation, in addition to the conjunction of active local invariants (line 6). If no messages are present in $Ready_i$ the annotation is simply *true* plus the conjunction of active local invariants (line 8).

Algorithm 6.4 Algorithm 6.4 on the following page constructs the annotation for the transition corresponding to the currently unwound fired set and computes the new sets of active local invariants. The starting point for these sets, $poss\_invs_j$ and $mand\_invs_j$, are the current sets (lines 1, 2). First the local invariants, whose end atoms are unwound in this step have to be considered, since these local invariants must not appear in the transition annotation and thus have to be removed first. This is done by function Handle_Local_Invariant_Ends (line 4), which returns the first part of the transition annotation and adjusts the set of active local invariants and whose behavior is detailed by algorithm 6.6. Lines 6 – 8 introduce some abbreviations for the set of all active local invariants, the set of identifiers of cold message receipts contained in the current fired set, and the set of message labels contained in simultaneous regions of $Fired_{i_k}$. The auxiliary function SimRegs($Fired_{i_k}$) yields the simultaneous regions contained in the fired set, i.e. those clusters, which are comprised of more than one atom, ColdMsgRcvLabels($Fired_{i_k}$) returns the set of message labels of cold asynchronous message receipt atoms in $Fired_{i_k}$. If cold message receipts are present in the current fired set, the transition annotation is extended appropriately by the function Generate_Main_Annotation (line 10), which is presented

Algorithm 6.4 *transAnnot* Construct Transition($ph, successor, Ready_i, Fired_{i_k}$)

```
 1: mand_invs_j := mand_invs_i
 2: poss_invs_j := poss_invs_i
 3:
 4: transAnnot := HANDLE_LOCAL_INVARIANT_ENDS(true,
                  Fired_{i_k}, Ready_i, j)
 5:
 6: let invs_j = poss_invs_j ∪ mand_invs_j
 7: let cmr = COLDMSGRCVLABELS(Fired_{i_k})
 8: let msr = MSGLABELS(SIMREGS(Fired_{i_k}))
 9: if cmr ≠ ∅ then
10:    transAnnot :=GENERATE_MAIN_ANNOTATION(Fired_{i_k}, cmr,
           msr, invs_j, transAnnot)
11: end if
12: if interpretation = strict then
13:    transAnnot := transAnnot ∧
           NEG_CONJUNCT(MsgLabels(l) \ MSGLABELS(Fired_{i_k}))
14: end if
15: if |SimClasses(Ready_i)| > 1 ∧ interpretation = weak then
16:    transAnnot := transAnnot∧
           NEG_CONJUNCT(MSGLABELS(Ready_i \ MSGLABELS(Fired_{i_k})))
17: end if
18:
19: transAnnot :=HANDLE_LOCAL_INVARIANT_STARTS(transAnnot,
        Fired_{i_k}, Ready_i, j)
20:
21: return transAnnot
```

in detail in algorithm 6.8 on page 140.

In case of the strict interpretation the transition must contain the conjunction of the negation of all message labels of the LSC, which are not unwound in the current step, which is done in line 13. If the current ready set contains more than one SimClass, which means that either (part of) a coregion is unwound or independent LSC elements, the transitions leaving the current state *ph* have to be made deterministic — as motivated in section 6.2.2 on page 116 — by adding the negation of all elements, which are not unwound, but are in the ready set (line 16).

Finally, the local invariant starts contained in the current fired set have to be considered and added to the transition annotation. This is handled by function HANDLE_LOCAL_INVARIANT_STARTS in line 19, which is described

by algorithm 6.7 on the following page.

Algorithm 6.5 Insert Skip Transition($TransitionAnnot, ph, successor$)

```
1: for all (q_x, ψ, STATE^{-1}(ph)) ∈⟶ do
2:    let (Ready_x, History_x, Cut_x) = STATE^{-1}(q_x)
3:    if COLDMSGRCVS(Ready_x) =ATOMS(Ready_x) then
4:       ⟶:=⟶ ∪ (q_x, TransitionAnnot∧
                 NEG_CONJUNCT(COLDMSGRCVLABELS)(Ready_x),
                 STATE(successor))
5:    end if
6: end for
```

Algorithm 6.5 Algorithm 6.5 generates the transitions, which skip the receipt of cold asynchronous message receipts as explained in detail in section 6.2.5. The strategy is to examine all transitions leading to the state, for which a successor has just been computed (line 1), and find those states immediately preceding the one for the current phase ph, which only contain cold message receipt atoms (line 3). The auxiliary functions COLDMSGRCVS($Ready_x$), resp. ATOMS($Ready_x$) return the set of cold message receive atoms, resp. all atoms of the ready set. These states are the ones from which a skipping transition originates and thus a correspondingly annotated transition is added to $\longrightarrow$ (line 4). As described in section 6.2.5 the annotation consists of the annotation of the currently considered transition (from STATE$^{-1}(ph)$ to STATE$^{-1}(successor)$).

Algorithm 6.6 $TransAnnot$ Handle Local Invariant Ends($TransAnnot, Fired_{i_k}, Ready_i, j$)

```
1: for all lie ∈ LIENDS(Ready_i) do
2:    if incl(lie) then
3:       TransAnnot := TransAnnot ∧ liID(lie)
4:    end if
5:    if mode(liID(lie)) = mandatory then
6:       mand_invs_j := mand_invs_j \ {liID(lie)}
7:    else
8:       poss_invs_j := poss_invs_j \ {liID(lie)}
9:    end if
10: end for
11: return TransAnnot
```

Algorithm 6.6 Algorithm 6.6 on the preceding page conjunctively adds the identifiers of all local invariant end atoms to the annotation of the current transition (line 3), if they are inclusive (line 2). Non-inclusive ones are automatically handled by not being in the set of active local invariants. All are removed from the respective sets for the successor phase (lines 5 – 8).

Algorithm 6.7 $TransAnnot$ Handle Local Invariant Starts($TransAnnot, Fired_{i_k}, Ready_i, j$)

```
 1: for all lis ∈ LIStarts(Ready_i) do
 2:    if incl(lis) then
 3:       TransAnnot := TransAnnot ∧ liID(lis)
 4:    end if
 5:    if mode(liID(lis)) = mandatory then
 6:       mand_invs_j := mand_invs_j ∪ {liID(lis)}
 7:    else
 8:       poss_invs_j := poss_invs_j ∪ {liID(lis)}
 9:    end if
10: end for
11: return TransAnnot
```

Algorithm 6.7 Algorithm 6.7 is the dual of algorithm 6.6 and likewise adds inclusive local invariants to the current transition (lines 2, 3) and adds all unwound local invariants to the respective sets of active local invariants for the next phase (lines 5 – 9).

Algorithm 6.8 *transAnnot* Generate Main Annotation($Fired_{i_k}, cmr, msr, invs_j, transAnnot$)

```
1: let msgs = MsgLabels(Fired_{i_k}) \ ColdInstLabels(Fired_{i_k})
2:
3: if (cmr ⊂ msr) then
4:    transAnnot := transAnnot ∧ Conjunct((msgs \ cmr)
         ∪ CondIDs(Fired_{i_k}) ∪ invs_j)
5: else
6:    transAnnot := transAnnot ∧ Conjunct(msgs ∪
         CondIDs(Fired_{i_k}) ∪ invs_j)
7: end if
8: return transAnnot
```

Algorithm 6.8 Algorithm 6.8 on the facing page constructs the main part of the transition annotation and also takes care of cold message receipts, except for the skipping transition, which is handled by function INSERT_SKIP_TRANSITION described in algorithm 6.5 on page 139. Cold instantaneous messages are taken care of in line 1, where they are removed from the set of labels of fired messages; the auxiliary function COLDINSTLABELS($Fired_{i_k}$) returns the labels of all cold instantaneous messages in the current fired set.

Then it is checked, whether asynchronous cold message receipts have to be considered for the transition annotation. They can be neglected, if they are contained in a simultaneous region, which also contains other messages (cf. page 126). This is expressed by the strict subset requirement in line 3. In this case the cold message receipts (cmr) are disregarded for the transition annotation (line 4), otherwise they are included (line 6). In both cases the condition identifiers and the active local invariants are added, if present.

6.3 Activation and Quantification

The symbolic automaton generated by the unwinding algorithm in the preceding section defines the behavior of the LSC body, disregarding the activation point, mode and quantification information. In order to define the semantics of the entire LSC the system, whose behavior is specified by the LSC, is needed; we call this the *reference system* of the LSC. In this section we introduce an abstract representation of such a reference system, which is used to define the complete semantics of an LSC.

6.3.1 Reference System

Informally, a system consists of a set of functional units, which are hierarchically organized, i.e. there are several levels of hierarchy, each consisting of a set of functional units, which may be decomposed into sub-units and so on. Each functional unit has a set of *ports*, which form its interface, the interfaces of different units are connected by *channels*. Information from unit to unit is passed along the channels, which can be delaying. Each unit also implements a behavior, whose concrete realization is unimportant for this thesis. Figure 6.18 shows an example with three levels of hierarchy. Functional units are depicted as boxes, channels by lines between units, interfaces by channels

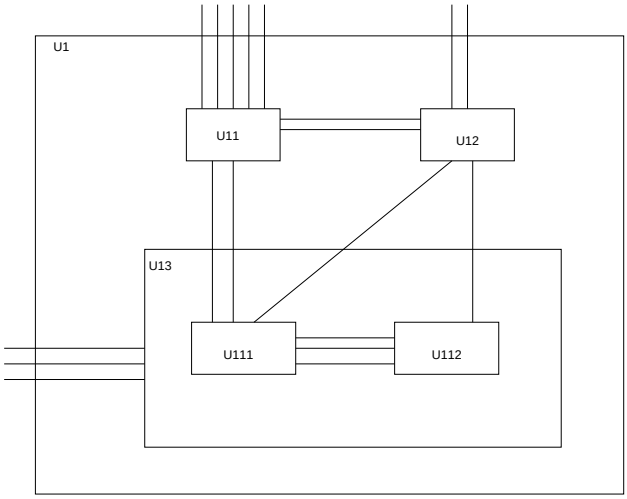


Figure 6.18: Example for a hierarchical system

connecting to a unit, and hierarchy by the nesting of boxes.

6.12 Definition (Functional Unit)
Let T be a set of types. A *functional unit* FU is a tuple

$$FU = (Ports, SubFUs, Channels)$$

where

- $Ports$ is the set of typed, directed ports, which makes variables of FU visible outside of FU
- $SubFUs = (FU_1, \ldots, FU_n)$ is the set of functional sub-units, which are contained in FU
- $Channels$ is the set of channels connecting the functional sub-units with each other and with the ports of FU

Each channel $ch \in Channels$ connects two ports: $ch = (src\_port, dest\_port)$, where src_port is the source port of the channel and $dest\_port$ the destination port. The port direction is given by the function $dir : Ports \longrightarrow in, out$[8]. The type of a port is given by the function: $port\_type : Ports \longrightarrow T$. ■

[8]Inout ports can be modeled by two separate sets of channels and source and destination ports.

6.13 Definition (Well-formedness of a Functional Unit)
A functional unit FU is well-formed ***iff***

1. $\forall ch \in Channels : port\_type(src\_port) = port\_type(dest\_port)$ (Ports must be type compatible.)
2. $\forall ch \in Channels : dir(src\_port) \neq dir(dest\_port)$ (A channel may only connect an in port with an out port and vice versa.)
3. $\forall ch = (src\_port, dest\_port), ch' = (src\_port', dest\_port') \in Channels : dest\_port \neq dest\_port'$ (No multiple senders.)

■

6.14 Definition (Reference System)
A *reference system* S is a well-formed functional unit: $S = (Ports, SubFUs, Channels)$. ■

Each functional unit contains internal variables, which may be made visible to other units or the outside world via the units' ports. The ports are typed according to the type of the variable, which is associated with the port. The concrete types and nature of the variables is again abstracted from for our purposes. The observables of the system thus are those variables, which are made available via ports. The domain of variables, and hence of the ports, of reference system S is denoted by DOM_S. Valuation

$$\theta : (Ports \cup Ports_{FU_1} \cup \cdots \cup Ports_{FU_n}) \longrightarrow Dom_S$$

assigns to each port $p \in Ports \cup Ports_{FU_1} \cup \cdots \cup Ports_{FU_n}$ the value of the variable associated with p. A sequence of such valuations makes up a run of reference system (cf. section 5.3):

6.15 Definition (Reference System Run)
Let $Props_S := \{p \rhd v \mid p \in Ports \cup Ports_{FU_1} \cup \cdots \cup Ports_{FU_n}, v \in DOM_S \vee v \in\in Ports \cup Ports_{FU_1} \cup \cdots \cup Ports_{FU_n}, \rhd \in \{=, <, >, \leq, \geq\}\}$ be the set of all propositions over ports of reference system $S = (Ports, SubFUs, Channels)$ and let $\tau = \tau_0\tau_1\tau_2 \ldots$ be a time sequence.

A *timed run* tr of reference system S is an infinite sequence of pairs $(\theta, \tau) : tr = (\theta_0, \tau_0)(\theta_1, \tau_1)(\theta_2, \tau_2) \ldots$
The i-th valuation of timed run tr is denoted by $tr_i = (\theta_i, \tau_i)$ and the suffix of a timed run tr, which starts at the i-th valuation is denoted by $\overrightarrow{tr_i} =$

$(\theta_i, \tau_i)(\theta_{i+1}, \tau_{i+1})\ldots$ The finite segment of a timed run, starting at the i-th valuation and ending at the j-th, is denoted by $tr_i^j = (\theta_i, \tau_i)\ldots(\theta_j, \tau_j)$
The satisfaction of a formula $\psi \in BExpr_{Props}$ over basic propositions over ports of S by the i-th valuation of tr is denoted by $tr_i \models \psi$.
$Runs(S)$ is the set of all timed runs of reference system S. ■

6.3.2 Complete Semantics

The complete semantics of an LSC is defined in terms of the runs, which are produced by the system and accepted by the automaton of the LSC. The unwinding algorithm operates on the identifiers, which are associated with the unwound LSCs elements, and therefore the generated symbolic automaton is annotated with formulas over these identifiers. The final step in relating the LSC to the reference system is taken by mapping identifiers to concrete design elements.

Since environment instances have to be mapped to the reference system itself — which represents the border between the environment and the actual system under development — the function $envInst(\cdot)$ is introduced. It returns $true$, if an instance is an environment instance and $false$, if it is a normal instance:

$$envInst : Instances(l) \longrightarrow \mathbf{B}$$

The *mapping function map* for an LSC L and a reference system S assigns a functional unit to each instance identifier, and a proposition over ports of S to each message label and condition and local invariant identifier:

- $map(i) = FU_j$, for $i \in Inst(l)$ with $envInst(i) = false, FU_j \in SubFUs, 1 \leq j \leq n$
- $map(i) = FU$, for $i \in Inst(l)$ with $envInst(i) = true$
- $map(label) = p$, for $label \in MsgLabels(l) \cup Conditions(l) \cup Local\_Invariants(l), p \in Prop_S := \{p \rhd v \mid p \in Ports \cup Ports_{FU_1} \cup \cdots \cup Ports_{FU_n}, v \in DOM_S \vee v \in\in Ports \cup Ports_{FU_1} \cup \cdots \cup Ports_{FU_n}, \rhd \in \{=, <, >, \leq, \geq\}\}$

Message labels should be mapped to ports of suitable types, suitable in this context meaning event-based, since messages in LSCs express dynamic

information exchange. Mapping a message e.g. to an integer port is typically not appropriate; mapping a message to a value change of an integer port is suitable. Complementarily, should conditions be mapped to static expressions, as they represent properties which are stable for several points in time. Local invariants belong to both categories, since they express both static and dynamic properties. Consequently, their mapping may consist of event-based and static parts. The concrete definition of $map(\cdot)$ depends on the reference system; see section 6.3.3 on the following page for an example.

For a symbolic automaton $\mathcal{TSA}$ we thus denote the substitution of all identifiers by their corresponding proposition by $map(\mathcal{TSA})$, the accepted language is consequently denoted by $\mathcal{L}(map(\mathcal{TSA}))$.

The exact definition of the relation between both types of runs depends on the activation mode and quantification. Initial LSCs are activated at system start, whereas invariant and iterative LSCs are activated whenever the activation condition is true. Iterative LSCs allow only one incarnation of an LSC at a time, i.e. such an LSC may not be reactivated.

Recall from definition 6.1 on page 94 the formal LSC definition: $L = (l, assumptions, ac, pch, amode, quant)$. We assume here that the set of assumptions is empty and no pre-chart is specified in order to focus on the LSC constructs presented in this chapter. The semantics is extended to deal with assumptions and pre-charts in chapters 8 and 9, respectively.

6.16 Definition (Satisfaction of an LSC)
Let $L = (l, \emptyset, ac, \epsilon, amode, quant)$ be an LSC, $\mathcal{TSA}_l$ the timed symbolic automaton generated for LSC body l by the unwinding algorithm, and S the corresponding reference system.

L is *existentially satisfied by* S, denoted $S \models_\exists L$, ***iff*** $quant = existential \wedge \exists tr \in Runs(S) : tr \models_\exists L$, where

$$tr \models_\exists L \textbf{\textit{iff}} \begin{cases} tr_0 \models ac \wedge \\ \qquad \overrightarrow{tr_1} \in \mathcal{L}(map(\mathcal{TSA}_l)) & amode = initial \\ \exists i : tr_i \models ac \wedge \\ \qquad \overrightarrow{tr_{i+1}} \in \mathcal{L}(map(\mathcal{TSA}_l)) & amode = invariant \\ \exists i : tr_i \models ac \wedge \neg active(L) \wedge \\ \qquad \overrightarrow{tr_{i+1}} \in \mathcal{L}(map(\mathcal{TSA}_l)) & amode = iterative \end{cases}$$

L is *universally satisfied by* S, denoted $S \models_\forall L$, ***iff*** $quant = universal \wedge \forall tr \in Runs(S) : tr \models_\forall L$, where

$$tr \models_\forall L \textbf{\textit{iff}} \begin{cases} tr_0 \models ac \Rightarrow & \\ \quad \overrightarrow{tr_1} \in \mathcal{L}(map(\mathcal{TSA}_l)) & amode = initial \\ \forall i : tr_i \models ac \Rightarrow & \\ \quad \overrightarrow{tr_{i+1}} \in \mathcal{L}(map(\mathcal{TSA}_l)) & amode = invariant \\ \forall i : tr_i \models ac \wedge \neg active(L) \Rightarrow & \\ \quad \overrightarrow{tr_{i+1}} \in \mathcal{L}(map(\mathcal{TSA}_l)) & amode = iterative \end{cases}$$

with
the predicate $active(L)$ is *true*, if there is another active incarnation of L, which has not yet reached its final or exit state.

■

Note that for the universal satisfaction of an LSC a satisfaction of the activation condition only *implies* the satisfaction of the LSC body. This is due to the fact that in the universal quantification *all* states of all runs have to be checked for activation: a conjunction as in the case of existential satisfaction would require every state of every run to satisfy the activation condition, which is not the intended semantics. The consequence of the implication is that an LSC may be universally satisfied, even though it is never activated. For practical applications it is thus recommended to check for a satisfied universal LSC, if it is activated at least once. One such check is e.g. to check the universal LSC for existential satisfaction.

6.3.3 Implications on the Interpretation

The types of the ports of the reference system have an impact on the interpretation of the LSC, since the LSC messages, conditions and local invariants use these ports. We will use STATEMATE as example in this section, but the argument is valid independently of a particular modeling language. We thus first give an informal overview of the mapping of LSC elements to STATEMATEelements: the functional units are activities, so that instances are mapped to activities and environment instances to environment activities. The channels are represented by information, control and data flows, and the ports by the data items and events contained in them.

Conditions should therefore be mapped to data items and conditions, whereas messages should be mapped to events, explicit or derived ones. The latter type of events are derived from data items and conditions by observing actions on these. For data items events expressing the writing or change of a data item are available — e.g. `changed(int_x)` which is true when the value of integer data item `int_x` changes. For conditions derived events for observing the falling and rising edge are available.

EXAMPLE 6.9
In LSC `securing_yerr` in figure A.33 on page 326 message `switch2yellow` is e.g. mapped to event `SWITCH_ON` (`SWITCH_ON = TRUE`) and message `opening` is mapped to condition `CLOSED` (`false(CLOSED)`, falling edge of `CLOSED`). ■

For the strict interpretation only certain types are usable, since duplicate messages are forbidden. Assume e.g. that a message is mapped to a port `cond` of type (STATEMATE) condition and the expression `cond = true`. Since typically `cond` is true for more than a single point in time, the corresponding LSC is immediately violated in the strict interpretation. The same holds for integer data types and other data items. Thus, such data types must not be used in the strict interpretation, unless it can be guaranteed, that during the activation of the LSC(s) in question the expected value is observed only at one single point in time, which is generally not the case.

A similar situation arises when messages are mapped to often recurring events, like e.g. in STATEMATE implicit signals indicating that a data item has been written or changed. Assume for instance that `int_x` is an integer and that it is incremented every two steps. Mapping a message to the port representing the implicit signal `changed(int_x)` results in immediate violations in the strict interpretation. Consequently, no two messages of an LSC may be mapped to the same port or expression when this LSC is to be interpreted strictly. Furthermore, in order to be able to use the strict interpretation every message must be mapped to an expression, which is completely disjunct from expressions other messages of the same LSC are mapped to. Consider for instance two messages `msg1` and `msg2`, which are both mapped to the same port, say integer `int_x` and expressions `int_x < 7` and `int_x > 2`, respectively. Since the truth valuations of the two expressions are not disjunct, both expressions become true when `int_x` is e.g. set to 4, immediately violating an LSC, which contains both messages and is interpreted strictly. For the weak interpretation there is no restriction on which ports to choose for a message, since duplicate messages are allowed.

Therefore caution has to be exercised, if the strict interpretation is to be used. Otherwise unexpected or unwanted violations of the LSC will result.

Another source of undesired behavior are cold cuts. In the weak interpretation cold cuts should be used with care, because once a cold cut is reached the LSC is essentially deactivated. The message(s) expected at this point are either observed and the LSC continues or they are never observed without any further restrictions (from the interpretation) on the subsequent behavior of the system. Cold cuts in combination with the strict interpretation are also potentially dangerous, since if the system never exhibits behavior, which prompt the automaton to leave the state corresponding to the cold cut, all messages contained in the LSC are forbidden forever. This may not be the behavior intended by the specifier, so cold cuts should be used with care in any case.

6.3.4 Well-formedness Rules

In conclusion we here summarize the well-formedness rules, which have been identified in this chapter and in chapter 4. Henceforth we assume that all LSCs are well-formed.

1. Instantaneous messages may not cross other instantaneous ones.
2. For each method call there is a return message and vice versa.
3. On an instance axis which has emitted a method call there are no messages either sent or received between the call and the return.
4. Every local invariant start or end atom must be encapsulated in a simultaneous region.
5. Simultaneous regions involving an instance head may only contain the start of a local invariant.
6. Every condition should be bound to at least one hot message.

6.4 Related Work

In this section we relate our approach to other work from the literature, which either extend the feature set as given by MSCs or SDs and/or provide formal characterizations for standard or extended sequence charts. Chapter 4 has already highlighted the points, where the language of LSCs presented in this thesis differs from the one proposed in [DH98, DH01]. In this section we thus focus on the differences of the formal semantics.

The semantics of LSCs as defined in [DH01] are similar to our approach due to the characterization by a symbolic transition system called *skeleton automaton*, which is vaguely similar to our symbolic automaton, but differs in definition of acceptance of a run. It is encoded into different parts of the skeleton automaton and is defined on the one hand by reaching either the final or the exit state and on the other hand by stopping somewhere along the way as long as all progress requirements, which are collected in separate set, have been met. Our notion of acceptance in contrast hinges only on one concept: the set of fair states of the generated symbolic automaton.

The skeleton automaton includes an explicit error state, called *abort*, which is implicit in the symbolic automaton. The abort state does not capture all violations of the LSC, since unfulfilled liveness requirements are detected via a different mechanism, a set of promises.

Whereas the automaton of [DH01] defines a pure interleaving semantics, the symbolic automaton additionally allows in certain circumstances more than one instance to advance in a step. For concurrently enabled LSC elements the latter solution is more appropriate, since simultaneity is explicitly supported by simultaneous regions.

[DH01] only considers the strict interpretation, which is not formalized. All messages appearing in an LSC are collected in a list, which may additionally contain other messages, which are forbidden during activation of the LSC (*forbidden messages*). Such additional forbidden messages are not explicitly treated in our approach, but can be easily added by using mandatory local invariants. This moreover allows the selective prohibition of messages within specific segments of the LSC only. Cold asynchronous message receipts are treated identically to our approach, i.e. they are ignored.

The evaluation point for isolated conditions is not considered in detail in [DH01], it is rather left open, when such a condition is evaluated, which corresponds to the default solution of our approach.

Bontemps [Bon01] gives a semantics definition for LSCs, which closely follows [DH01], but constructs a finite automaton. The semantics cover only a subset of the LSC features leaving out conditions, local invariants, simultaneous regions and time. In addition to existential and universal quantification [Bon01] considers also a third possibility, called *no-stories*, which simply is the negation of the existential quantification.

There are other approaches, which aim at defining a richer, more expressive variant of MSCs or a suitable formal semantics. Krüger [Krü00] takes up some of the ideas presented in [DH98] and enhances (a subset of) MSC-96 with a number of concepts; a formal semantics based on streams is also provided. Among the new concepts is the possibility to specify quantification information, i.e. to designate MSCs as existential or universal, in the same fashion as in [DH98]. In addition to these two modes (called interpretations in [Krü00]) an exact mode, which forbids all behavior not explicitly specified in the MSC, and the negation of an MSC are introduced.

There is no possibility to distinguish between mandatory and possible elements within the chart in this approach; all messages must be observed, i.e. all locations and messages are hot in LSC terminology. The underlying communication paradigm is synchronous, i.e. there is no distinction between the points of sending and receiving of a message, although a potential interpretation for asynchronous message exchange is offered, which assumes that messages are buffered by the channels they are sent on, similar to an event queue in UML. Conditions are not given a semantics directly, but can be modeled by *guarded MSCs*, i.e. MSCs, which are guarded by a boolean condition. Since the final MSC may be composed from smaller parts, which are MSCs themselves, this concept can be used to give meaning to conditions and also to model activation conditions. The use of an activation condition is not required though, so that in general the point of activation is not resolved. If an MSC starts with a guard, this would correspond to the invariant activation mode; other activation modes are not considered.

Local invariants, simultaneous regions and an activation mode are not covered in [Krü00]. Duplicate messages are not constrained, except in the exact interpretation. Thus for all modes except the exact the interpretation — in our terminology — is weak, whereas the constraint imposed by the exact mode is more restrictive than our strict interpretation, since only those system behaviors are allowed, which are explicitly specified in the LSC. This level of strictness is unnecessary for our approach, since we do not aim at specifying the entire behavior of a system in one LSC.

The (to our knowledge) first to bring together liveness and MSCs were Ladkin and Leue [LL92b, LL92a, LL95], who define a formal, automata-based semantics for part of MSC-93. Their approach to derive an automaton from an MSC is similar to the one presented in this chapter, although the set of features covered is much smaller, since they only consider messages.[9] A construct similar to a cut, called *global system state*, and a transition relation on these states is used to generate the automaton. Liveness is added to the generated automaton by manually designating states as fair in the sense of Büchi automata. These liveness properties are not expressible in the graphical MSC representation, but added only as a finishing touch on the generated automaton.

Firley et al. [FHD+99] translate Sequence Diagrams with timing annotations into timed automata in order to check the SDs for timing consistency using the UPPAAL model checker [LPY97]. Inspired by [DH99] they distinguish between mandatory and optional behavior on the chart level, which nevertheless differs from the LSC concept of quantification. There exists also a third type of behavior, called *If-Then-Behavior*, which is similar to pre-charts and therefore covered in section 9.3 on page 218.

The goal in [FHD+99] is to check, if a given SD is consistent with respect to its timing constraints. This property only requires *one* run of the system, which completely traverses the entire SD. The quantification, in our terminology, is thus always existential. Given the fact that no liveness properties, no conditions and no local invariants can be specified within the considered SDs, the notion of existential satisfaction is slightly different than in our approach. [FHD+99] always require the complete traversal of the SD, which is not necessarily required in our existential interpretation, where the LSC is also fulfilled, if a possible condition or local invariant is evaluated to false or we get stuck in a cold cut forever. The timed automata constructed in [FHD+99] thus do not contain an exit state; in contrast to our timed symbolic automata they do have an explicit error state, however.

The distinction between mandatory and optional SD behavior corresponds rather to strict, resp. weak interpretation of LSCs. The issue of activating the SD is addressed partly, again depending on the SD behavior. Mandatory SDs are activated when the first message is observed, optional and If-Then-SDs are activated non-deterministically on observation of the

[9] Asynchronous ones in the MSC-93 case, but also synchronous ones in the more general case of Message Flow Graphs, which are similar to MSCs.

first message.[10] Since the quantification is inherently existential in this approach, there exists no possibility for multiple or concurrent activations of an SD. The non-deterministic activation for optional and If-Then-SDs, however, allows to consider SD incarnations other than the first.

Within an SD there is no distinction between mandatory and possible behavior, so that no progress is enforceable. The timing constraints in a mandatory SD allow to express bounded liveness requirements, however, but in the other SD types progress is not required. Conditions and local invariant are not covered and messages are limited to synchronous ones, which additionally have to totally ordered.

Padilla and Jonsson [JP01] define a formal semantics for MSC-2000, which is given as an *abstract execution machine*, which is constructed according to a set of production rules from an MSC and is intended to serve either as a monitor or generator of test sequences. They consider boolean conditions, which has become possible due to the introduction of data in MSC-2000. The question of what the consequence of a violated condition is, is resolved by waiting for the rising edge. The issues of activation, simultaneity and progress are not covered.

There are several approaches, which are concerned with adding timing information to MSCs and SDs. These approaches allow to specify a limited version of liveness by placing upper or exact bounds on the occurrence times of messages, etc. Typically it is not clear, if the elements constrained by the time bound must occur within the allotted time or whether, if they occur, it has to be within the time bound. Some also allow to use ∞ as an upper bound, so that real liveness can be specified, although the semantics provided are of varying degrees of formalization. We will discuss these approaches in more detail in the section 7.3 on page 167.

[10] Activation on observation of the first message works in this case, because only SDs with totally ordered messages are considered.

Chapter 7

Adding Time

Being able to specify time constraints in property specifications is a key point, especially but not exclusively in the field of safety-critical systems, where a timely behavior is essential. One essential property of an airbag controller e.g. is that it activates the firing capsule within a certain amount of time after a crash has been detected. As presented in chapter 3 both MSCs and SDs provide constructs for the expression of time like timer or timing intervals, although the semantics, if existing, do not treat time quantitatively, as criticized in sections 3.1.4 and 3.2.

LSCs allow the specification of time constraints either in form of an MSC-style timer or in interval notation, with a lower and an upper bound. The semantics of the time constraints is defined by using a timed symbolic automaton instead of an untimed one and associating a clock with each time constraint. Transitions containing elements constrained by a timer or timing interval are annotated with a corresponding clock constraint.

This chapter is structured in the following manner: section 7.1 introduces the time constraints used in LSCs, the formal syntax and semantics are given in section 7.2. This chapter concludes with a comparison to the other works dealing with time in sequence charts in section 7.3.

7.1 Time Constraints in LSCs

Formulating requirements about the behavior of a system over time is essential for most modern computer systems and even more so for safety-critical systems. The LSC in figure 4.9 on page 78 shows an example from our train

systems application: the train is required to issue a status request to the crossing a certain amount of time after receiving the acknowledgment for the activation message.

As presented in chapter 3 several constructs for expressing time requirements are available in MSCs and SDs: MSCs offer timers and in MSC-2000 also an interval notation and time stamps (absolute and relative) are introduced, SDs provide an interval-like graphical notation and textual capabilites for the specification of timing requirements. The timing constructs chosen for LSCs are based on MSC-96-style timers and the interval notation. The intuitiveness of the graphical representation of timers and timing intervals in MSCs motivated us to use these constructs.

The graphical representation of timers is identical to the one given in MSC-96 and MSC-2000, i.e. the setting of a timer is represented by an hour glass symbol, which is annotated by a name and a duration; a timeout symbol is represented by an hour glass symbol, which is connected to the instance axis by an arrow; a timer reset is represented by a large X, which is connected to the instance axis by a simple line. The relation between a timer set and its corresponding query (timeout or reset) is established either by a vertical line, which connects the two symbols or by using the same name for setting the timer and the query. In the former case the second hour glass symbol can be omitted for a timeout. The second alternative (identification via name) offers more flexibility in placing timer atoms and thus increases readability. Figure 7.1 gives an example. `T1` shows a timeout, which is connected to its timer setting by a vertical line; `T2` shows an example, where the timer reset is identified with its corresponding timer set by the name instead of a line. Note that resetting a timer implies that in the meantime the timer has not expired, i.e. no timeout has occurred. As for MSCs there may be at most one timer reset or timeout per timer set and they are confined to one instance.

Timing intervals express quantitative local liveness properties, since they refer to neighboring atoms. They are used to give both a minimum and a maximum delay between two directly consecutive atoms. The delimiting atoms (or rather clusters) can either be located on the same instance axis, one directly after the other, or be the sending and receipt of an asynchronous message. They deviate in this respect from the intervals available in UML SDs and MSC-2000, which can connect arbitrary clusters. These more general intervals can, however, be modeled by timers and can thus be seen as shorthand notations. We therefore restrict the timing features in this thesis to the basic ones introduced above.

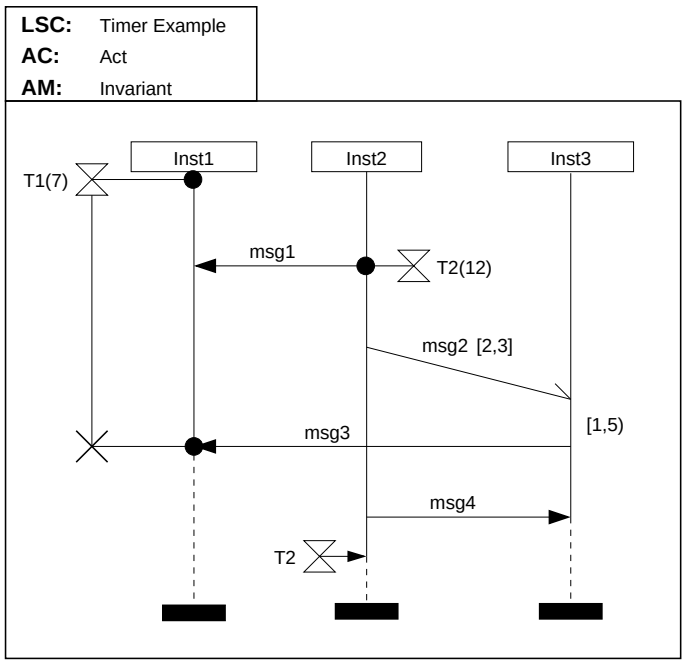


Figure 7.1: Examples of LSC timing constraints

The interval notation has already been used in Real-Time Symbolic Timing Diagrams [Fey96, FJ97] and also been proposed for MSC-96 in [AHP96, BAL97a, BAL97b, LL99a, GDO98]. Timing constraints in STDs are also used to express unbounded liveness requirements by using and excluding ∞ as an upper bound; this is catered for by location and message temperatures in LSCs. The intervals are placed next to the instance axis between the two locations which delimit them or are attached to the identifier of the constrained asynchronous message. Two types of parenthesis are available: one to indicate inclusion of a bound ('[' or ']') and one for exclusion ('(' or ')') yielding closed ('$[n, m]$'), half-open ('$[n, m)$' or '$(n, m]$') and open ('(n, m)') intervals. Obviously the upper bound must be greater or equal to the lower bound: $n \leq m$. Figure 7.1 shows two examples: the transmission of `msg2` should take at least 2 and at most 3 time units and after receiving `msg2`, `msg3` should be sent at least 1 time unit later, but before 5 time units pass.

Timer set atoms should be bound to (a set of) message(s) via a simultaneous region, because they need reference points, similar to local invariants,

as pointed out in our criticism of MSCs in section 3.1.4 on page 57. The most typical use case for timers is that they are set when some event is observed, e.g. the receipt of a message, and some reaction is expected within a certain span of time. The situation is slightly different for timeouts and timer resets: Timeouts are observable themselves, so that they need no reference point, but can rather act as reference points themselves. They can thus be used to delimit local invariants, conditions and other timer sets, and of course messages. Timer resets can not be used as reference points, since they do not indicate a specific point in time. They may nevertheless be used in isolation, since this allows the expression of upper bounds: Since they indicate that the associated timer has not yet expired, this means that the occurrence times of all atoms between the timer set and reset occur before the timer expires. The reference points of timing intervals are clear, since they are always bound to two adjacent locations.

As figure 7.1 illustrates for `T1`, timer sets can be associated with an instance head in order to express timing constraints, which measure time from the point of activation. Timing intervals may also be attached to instance heads just like they are attached to other locations.

Timing intervals, due to being constrained to two adjacent locations, are intended for local constraints, whereas timers are intended to range over larger parts of an instance. Timers ranging over several instances are generally conceivable, but require more care, since it is not immediately clear, if the timer is always set before it is queried. We thus constrain the use of timer to one instance.

7.2 Formal Semantics

Before the unwinding algorithm is adapted to handle timer and timing intervals in section 7.2.2 these constructs are added to the abstract syntax representation in section 7.2.1.

7.2.1 Formal Syntax

Analogous to the atoms defined in section 6.1 on page 94 we introduce the following sets for the timer atoms of LSC body l and instance $i \in Instances(l)$, respectively:

- $Timer\_Set(i), Timer\_Set(l)$: sets of timer set atoms

- $Timer\_Reset(i), Timer\_Reset(l)$: sets of timer reset atoms
- $Timeouts(i), Timeouts(l)$: sets of timeout atoms

We collect all timer related atoms in the sets

$$Timer(i) := Timer\_Set(i) \cup Timer\_Reset(i) \cup Timeouts(i)$$

$$Timer(l) := Timer\_Set(l) \cup Timer\_Reset(l) \cup Timeouts(l)$$

These sets have to be added to the set of atoms of an instance, resp. entire LSC body:

$$\begin{aligned} Atoms(i) := \quad & \{\bot_i\} \cup \\ & Msgsnd(i) \cup \\ & Msgrcv(i) \cup \\ & Conds(i) \cup \\ & LI\_starts(i) \cup \\ & LI\_ends(i) \cup \\ & Timer(i) \cup \\ & \{\top_i\} \end{aligned}$$

$$\begin{aligned} Atoms(l) := \quad & Instheads(l) \cup \\ & Msgsnd(l) \cup \\ & Msgrcv(l) \cup \\ & Conds(l) \cup \\ & LI\_starts(l) \cup \\ & LI\_ends(l) \cup \\ & Timer(l) \cup \\ & Instends(l) \end{aligned}$$

As for most other atoms it is necessary to be able to identify timers and map timer atoms to their corresponding identifier. The set of timer identifiers is given by $TimerIDs(l)$. Moreover, the duration of a timer needs to be known. Timing intervals are not handled by atoms of their own, since they are not observable entities to be unwound. Thus mappings from locations and asynchronous message sendings to timing intervals are defined, which provide the information on upper and lower bounds and if they are inclusive. Timing intervals are attached to the location of the *first* cluster. There are thus the following functions:

$$timID : Timer(l) \longrightarrow TimerIDs(l)$$

$$duration : TimerIDs(l) \longrightarrow \mathbf{N}$$

$$lowBound : Locations(l) \cup AsyncMsgSnd(l) \longrightarrow \epsilon \cup \mathbf{N}$$

$$lowIncl : Locations(l) \cup AsyncMsgSnd(l) \longrightarrow \mathbf{B}$$

$$upBound : Locations(l) \cup AsyncMsgSnd(l) \longrightarrow \epsilon \cup \mathbf{N}$$

$$upIncl : Locations(l) \cup AsyncMsgSnd(l) \longrightarrow \mathbf{B}$$

The set $AsyncMsgSnd(l)$ is the set of all asynchronous message send atoms: $\{m \in Msgsnd(l) \mid sync\_type(msgID(m))) = async\}$. The functions $lowBound(\cdot)$ and $upBound(\cdot)$ give the lower, resp. upper bound of a timing interval. They return ϵ, if there is no timing interval associated with the location or message. The functions $lowIncl(\cdot)$ and $upIncl(\cdot)$ return $true$, if the respective bound is inclusive.

Well-formedness Rules

1. Timer set atoms must be bound to a cluster of an observable atom.

2. A timer must be set before it expires or is reset.

3. For each timer set atom there is at most one reset or timeout atom.

4. The upper bound of a timing interval must be greater or equal to its lower bound.

The first rule is motivated by the fact that — similar to local invariants— there has to be a reference point for the start of a timer, as has been remarked in section 3.1.4 on page 57. Note that a timeout atom can be a reference point, since it identifies a single point in time, whereas a timer reset can not. Timer resets may be used in isolation in order to specify a lower bound.

7.2.2 The Timed Unwinding Algorithm

So far the semantics of the presented LSC elements were expressible in an untimed symbolic automaton, whereas now we need the time properties introduced in section 5.3. The timed unwinding algorithm extends the one presented in section 6.2.7 by incorporating the timing features introduced above and producing a timed symbolic automaton. The general idea is to associate a clock with every timer and timing interval, which is set to zero when the corresponding timer is set or the location of the timing interval is reached. When the timer is either reset or a timeout is observed or the location following the one with the timing interval is reached, a clock constraint is placed on the corresponding transitions in the automaton, which reflects the nature and duration of the timer or timing interval. For a timeout atom this means that the value of the associated clock must be equal to the duration of the timer. For a timer reset atom the clock must be strictly less than the timer duration, because no timeout has been observed so far, i.e. the clock must not have reached its maximal value. For timing intervals the clock must be equal to or greater than the lower bound and less than or equal to the upper bound depending on the type of interval (open, closed). Note that timing intervals and timer atoms which are bound via a simultaneous region to some observable do not alter the structure of the TBA. Only when a timer atom occurs without a simultaneous region the structure is changed; then a new node and a new transition are inserted.

Recall from definition 5.8 on page 89 that a set C of clocks is needed for a timed symbolic automaton. We use variables $z_0, z_1, \ldots$ to denote clocks, with the number of clocks for an automaton being the sum of timer set atoms plus locations, which carry a timing interval annotation, plus the number of asynchronous message send atoms, which are guarded by an interval.

Time Constraints and Exits

Clock constraints do not only influence the transitions, on which they appear, they also have an impact on transitions to the exit state, which start in the same state as the constrained transition. Consider e.g. the LSC in figure 7.2 on the following page and the corresponding automaton shown in figure 7.3 on the next page. The clock z_0 associated with the timing interval is reset when observing $msg1$ and the transition from q_1 to q_2 is constrained by a clock constraint corresponding to the interval. Additionally, the tran-

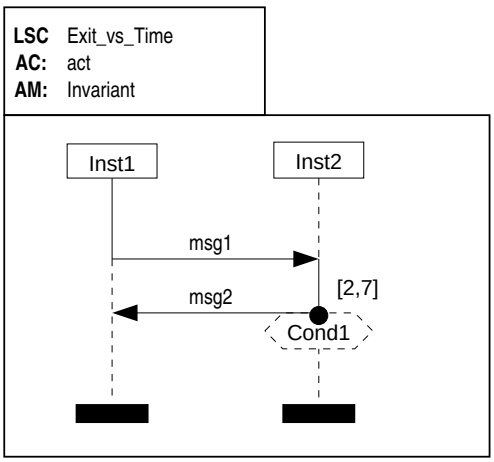


Figure 7.2: LSC illustrating the relation between timing constraints and possible conditions

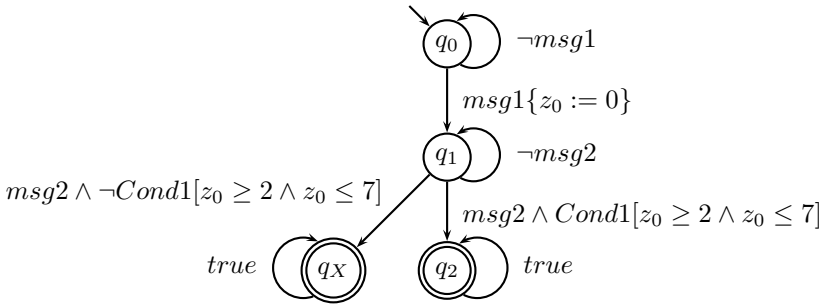


Figure 7.3: Automata for LSC `Exit_vs_Time` (weak interpretation)

sition for the violation of condition $Cond1$ must be annotated by the clock constraint as well. Otherwise it would be possible to enter the exit state from q_1, if $msg2$ occurs outside of the timing interval. We could e.g. wait for 10 time units in state $q1$, then observe $msg2$ while $Cond1$ is evaluated to false and take the transition into the exit state. In order to avoid this incorrect behavior the timing constraint for $msg2$ must also be attached to the transition leading to the exit state. This rule applies to all situations, where there is both a clock constrained transition to a non-exit successor state and a transition to the exit state, i.e. whenever possible conditions and local invariants are overlapping with timers or timing intervals.

Timed Unwinding Algorithm

Algorithm 7.1 shows the timed version of algorithm 6.1 on page 132, which generates a timed symbolic automaton $\mathcal{TSA} := (\Sigma, Q, q_0, C, \longrightarrow, F)$ as given by definition 5.8 on page 89. The core of the algorithm is identical to algorithm 6.1, therefore only the extensions for the treatment of time are discussed in detail here. The set C of clocks is defined in line 5, with k being the maximum number of clocks necessary for the LSC under consideration, i.e. $k := |TimerSets(l)| + |\{loc \in Locations(l) \,|\, lowBound(loc) \neq \epsilon\}| + |\{ams \in AsyncMsgSnd(l) \,|\, lowBound(ams) \neq \epsilon\}|$. The set of clocks, which are reset in this unwinding step, is computed by the function COMPUTE_RESETS (line 31), and the clock constraints for the current transition are computed by the function COMPUTE_CONSTRAINTS (line 32), which are described in detail by algorithms 7.3, resp. 7.4. The computed clock resets and constraints are added to the transition in line 42. Self loops do not have any timing constraints or clock resets, thus the corresponding parts of the transition remain empty (line 21). Note that the function inserting the skipping transition for cold asynchronous message receipts in line 44 is extended by the set of clock resets and the current clock constraints, since these have to be applied to the skipping transition as well.

The algorithms, which construct the annotation of the transition to the successor node (CONSTRUCT_TRANSITION, HANDLE_LOCAL_INVARIANT_ENDS, HANDLE_LOCAL_INVARIANT_STARTS, GENERATE_MAIN_ANNOTATION), need not be altered and are thus not repeated here.

The algorithm for the computation of the transition to the exit state needs to be slightly adjusted as shown in algorithm 7.2 on page 163. For the transition to the exit state for violated possible conditions and local invariants the clock constraints of *all* normal transitions leaving the current state need to be applied as well (cf. page 159 above). Note that the this entails the computation of the clock constraints for the *entire* ready set by function COMPUTE_CONSTRAINTS (see algorithm 7.4 on page 165 below) in line 30 and adding them to the transition to the exit state in line 33. Since both the ready set and the fired sets are sets of SimClasses the same function can be used. On this transition no clock needs to be reset, so the set of clock resets is empty.

Algorithm 7.3 collects the clock resets for all timer sets and starting intervals, which are unwound in the current step. Clocks are reset for each

Algorithm 7.1 Timed Unwinding Algorithm

1: **if** $interpretation$ = strict **then**
2: $sc :=$ NEG_CONJUNCT($MsgLabels(l)$)
3: **end if**
4: $\Sigma := MsgLabels(l) \cup Conditions(l) \cup Local\_Invariants(l) \cup \{true, false\}$
5: $C := \{z_0, z_1, \ldots, z_k\}$
6: $phases := \{Phase_0\}$
7: $Q := \{\text{STATE}(Phase_0), q_X\}$
8: $q_0 := \{\text{STATE}(Phase_0)\}$
9: **if** $cut\_temp(Cut_0) = cold$ **then**
10: $F := Q$
11: **else**
12: $F := \{q_X\}$
13: **end if**
14: $\longrightarrow := \{(q_X, true, \emptyset, \epsilon, q_X)\}$
15: $poss\_invs_0 := \{li \in Local\_Invariants(l) \mid \exists scl\, \exists cl \in scl\, \exists a, a' \in cl :$
$a \in Instheads(l) \wedge a' \in LI\_starts(l) \wedge liID(a') = li \wedge mode(a') = possible\}$
16: $mand\_invs_0 := \{li \in Local\_Invariants(l) \mid \exists scl\, \exists cl \in scl\, \exists a, a' \in cl :$
$a \in Instheads(l) \wedge a' \in LI\_starts(l) \wedge liID(a') = li \wedge$
$mode(a') = mandatory\}$
17: **while** $phases \neq \emptyset$ **do**
18: **let** $ph = (Ready_i, History_i, Cut_i) \in phases$
19: **if** $Ready_i = \emptyset$ **then**
20: $F := F \cup \{\text{STATE}(ph)\}$
21: $\longrightarrow := \longrightarrow \cup \{(\text{STATE}(ph), true, \emptyset, \epsilon, \text{STATE}(ph))\}$
22: **else**
23: GENERATE_EXITS($Ready_i, ph$)
24: **for all** $Fired_{i_k} \in \mathcal{P}(Ready_i)$ **do**
25: $successor := Step(ph, Fired_{i_k})$
26: **let** $successor = (Ready_j, History_j, Cut_j)$
27: **if** $successor = ph$ **then**
28: INSERT_SELF_LOOP($ph, Ready_i$)
29: **else**
30: $TransitionAnnot :=$CONSTRUCT_TRANSITION($ph, successor, Ready_i, Fired_{i_k}$)
31: $ClkResets :=$ COMPUTE_RESETS($Fired_{i_k}$)
32: $ClkConstr :=$ COMPUTE_CONSTRAINTS($Fired_{i_k}$)
33: **if** $\exists q \in Q$ *with* $\text{STATE}^{-1}(q) = (Ready_x, History_x, Cut_x) : Ready_i = Ready_x \wedge History_i = History_x$ **then**
34: $successor := \text{STATE}^{-1}(q)$
35: **else**
36: $Q := Q \cup \{\text{STATE}(successor)\}$
37: $phases := (phases \setminus \{ph\}) \cup \{successor\}$
38: **if** $cut\_temp(Cut_i) = cold$ **then**
39: $F := F \cup \{\text{STATE}(ph)\}$
40: **end if**
41: **end if**
42: $\longrightarrow := \longrightarrow \cup \{(\text{STATE}(ph), TransitionAnnot, ClkResets, ClkConstr, \text{STATE}(successor))\}$
43: **end if**
44: INSERT_SKIP_TRANSITION($TransitionAnnot, ph, successor, ClkResets, ClkConstr$)
45: **end for**
46: **end if**
47: **end while**

Algorithm 7.2 Generate Exits($Ready_i, ph$)

```
 1: ExitAnnot := false
 2: for all scl ∈ SimClasses(SimRegs(Ready_i)) do
 3:    if Poss_CondIDs(scl) ≠ ∅ then
 4:       ExitAnnot := ExitAnnot ∨
             (Neg_Disjunct(Poss_CondIDs(scl))∧
               Conjunct(MsgLabels(scl)))
 5:    end if
 6:    for all lis ∈ LIStarts(Ready_i) do
 7:       if lis ∈ scl ∧ mode(liID(lis)) = possible then
 8:          if incl(lis) then
 9:             ExitAnnot := ExitAnnot ∨ (¬liID(lis)∧
                   Conjunct(MsgLabels(scl)))
10:          else
11:             ExitAnnot := ExitAnnot ∨ ¬liID(lis)
12:          end if
13:       end if
14:    end for
15:    for all lie ∈ LIEnds(Ready_i) do
16:       if lie ∈ scl ∧ mode(liID(lie)) = possible then
17:          if incl(lie) then
18:             ExitAnnot := ExitAnnot ∨ (¬liID(lie)∧
                   Neg_Conjunct(MsgLabels(scl)))
19:          else
20:             ExitAnnot := ExitAnnot ∨ ¬liID(lie)
21:          end if
22:       end if
23:    end for
24: end for
25:
26: for all pc ∈Poss_CondIDs(Ready_i) \ Poss_CondIDs(SimClasses(Ready_i)) do
27:    ExitAnnot := ExitAnnot ∨ ¬pc
28: end for
29:
30: ClkConstr := Compute_Constraints(Ready_i)
31:
32: if ExitAnnot ≠ false then
33:    ⟶:=⟶ ∪ {(State(ph), ExitAnnot, ∅, ClkConstr, q_X)}
34: end if
```

timer set atom occurring in the fired set and every location and asynchronous message send atom, which carry a non-empty time interval. The auxiliary functions TimerSetIDs($Fired_{i_k}$), Locs($Fired_{i_k}$) and AsyncMs-

Algorithm 7.3 *ClkResets* Compute Resets($Fired_{i_k}$)

```
 1: for all timset ∈ TimerSetIDs(Fired_{i_k}) do
 2:    ClkResets := ClkResets ∪ {AssignClk(timset)}
 3: end for
 4:
 5: for all loc ∈ Locs(Fired_{i_k}) do
 6:    ClkResets := ClkResets ∪ {AssignClk(loc)}
 7: end for
 8:
 9: for all ams ∈ AsyncMsgSnds(Fired_{i_k}) : lowBound(ams) ≠ ε do
10:    ClkResets := ClkResets ∪ {AssignClk(ams)}
11: end for
```

gSnds($Fired_{i_k}$) yield all timer identifiers, all locations of clusters, resp. all asynchronous message send atoms of the fired set. The auxiliary function AssignClk() assigns a unique, unused clock name from C to each timer set and each location or asynchronous message send atom, which are guarded by a time interval.

Algorithm 7.4, lines 1 – 7: Timer Treatment Algorithm 7.4 constructs and collects all clock constraints, which are unwound in the current step. As seen in algorithm 7.2 on the preceding page this function is used to compute the clock constraints for the entire ready set. In this first part of the algorithm the clock constraints for timeout and timer resets are constructed. The auxiliary functions TimerResetIDs($Fired_{i_k}$) and TimeoutIDs($Fired_{i_k}$) return the identifiers of all timer reset, resp. timeout atoms, which are currently unwound. The auxiliary function ClkName() looks up the clock name, which has been assigned to the timer by function AssignClk above and which has to be used in the clock constraint for the timer reset or timeout. For a timer reset the only statement, which can be made about the clock, is that the duration of the corresponding timer has not yet elapsed, i.e. that presently the value of the associated clock is strictly less than the duration of the timer (line 2). For timeout atoms the constraint is more specific, since the timer expires at this moment, so that the clock must be equal to the duration of the associated timer (line 6).

Algorithm 7.4, lines 9 – 33: Timing Interval Treatment This part of the algorithm constructs the clock constraints for timing intervals,

Algorithm 7.4 *ClkConstr* Compute Constraints($Fired_{i_k}$)

for all $timres \in$ TimerResetIDs($Fired_{i_k}$) **do**
  $ClkConstr := ClkConstr \wedge$ ClkName($timres$) $<$ $duration(timres)$
end for

for all $to \in$ TimeoutIDs($Fired_{i_k}$) **do**
  $ClkConstr := ClkConstr \wedge$ ClkName(to) $=$ $duration(to)$
end for

for all $loc \in$ PredecLocs($Fired_{i_k}$) **do**
  if $lowIncl(loc)$ **then**
    $ClkConstr := ClkConstr \wedge$ ClkName(loc) $\geq$ $lowBound(loc)$
  else
    $ClkConstr := ClkConstr \wedge$ ClkName(loc) $>$ $lowBound(loc)$
  end if
  if $upIncl(loc)$ **then**
    $ClkConstr := ClkConstr \wedge$ ClkName(loc) $\leq$ $upBound(loc)$
  else
    $ClkConstr := ClkConstr \wedge$ ClkName(loc) $<$ $upBound(loc)$
  end if
end for

for all $amr \in$ AsyncMsgRcvs($Fired_{i_k}$) : $\exists ams \in AsyncMsgSnd(l)$: $msgID(amr) = msgID(ams) \wedge lowBound(ams) \neq \epsilon$ **do**
  if $lowIncl(ams)$ **then**
    $ClkConstr := ClkConstr \wedge$ ClkName(ams) $\geq$ $lowBound(ams)$
  else
    $ClkConstr := ClkConstr \wedge$ ClkName(ams) $>$ $lowBound(ams)$
  end if
  if $upIncl(ams)$ **then**
    $ClkConstr := ClkConstr \wedge$ ClkName(ams) $\leq$ $upBound(ams)$
  else
    $ClkConstr := ClkConstr \wedge$ ClkName(ams) $<$ $upBound(ams)$
  end if
end for

which are currently unwound. The auxiliary function PredecLocs($Fired_{i_k}$) yields for all locations in the current fired set the set of their immediate predecessor locations, which are annotated by a timing interval: PredecLocs($Fired_{i_k}$) := $\{loc \in Locations(l) \mid \exists scl \in Fired_{i_k} \exists cl \in scl : loc \in predecessor(location(cl)) \wedge lowBound(loc) \neq \epsilon\}$. The lower and upper bounds are constructed, depending on them being inclusive or exclusive.

Note that for timing intervals two terms are added to the clock constraint, one for the lower and one for the upper bound.

A similar procedure is carried out for asynchronous messages, which are constrained by a timing interval. For each receipt of an asynchronous message, which is annotated by an interval (line 22), a corresponding clock constraint is added, the exact form again depending on the type of bound and if it is inclusive or not.

Algorithm 7.5 Insert Skip Transition($TransitionAnnot, ph, successor, ClkResets, ClkConstr$)

```
1: for all (q_x, ψ, ρ, γ, STATE^{-1}(ph)) ∈⟶ do
2:    let (Ready_x, History_x, Cut_x) = STATE^{-1}(q_x)
3:    if COLDMSGRCVS(Ready_x) =ATOMS(Ready_x) then
4:       ⟶:=⟶ ∪ (q_x, TransitionAnnot, ClkResets, ClkConstr, STATE(successor))
5:    end if
6: end for
```

Algorithm 7.5 shows the adapted version of the function INSERT_SKIP_TRANSITION, which inserts the skip transition for cold asynchronous message receipts. Since the skipping transition is derived from the currently constructed transition, the current clock resets and constraints are applied also to the skip transition (line 4).

EXAMPLE 7.1
Figure 7.4 shows the automaton for the LSC from figure 7.1; setting a clock is represented by a corresponding expression in curly braces, whereas clock constraints are shown in square brackets. Clock z_0 is associated with `T1`, z_1 with `T2`, z_2 with the interval on `msg2` and z_3 with the interval between the receipt of `msg2` and the sending of `msg2` on instance `Inst3`. Note that z_0 is not reset explicitly, because all clocks are initially zero as given in definition 5.8 on page 89.

The clock constraint between states q_3 and q_4 is the conjunction of the two constraints generated for `T1` and the timing interval on `Inst3`. The final transition is only annotated with *true*, since no observable atoms are unwound. It is nevertheless constrained by the clock constraint for the timer reset of `T2`. ■

The definition of satisfaction of an LSC, definition 6.16 on page 145, does not need to be changed, since it already considers timed runs and timed symbolic automata.

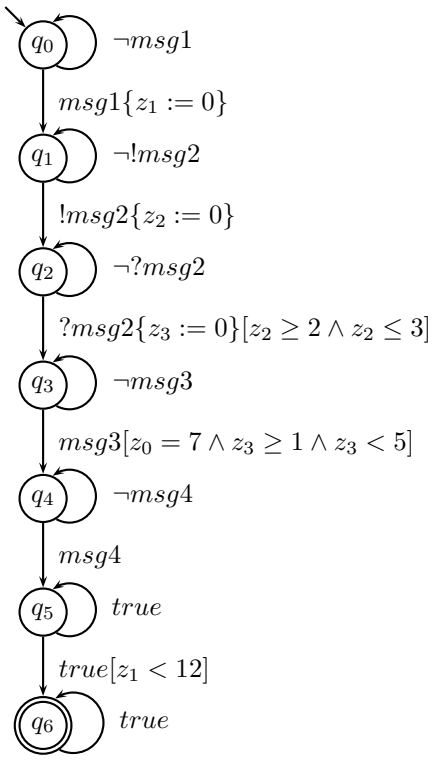


Figure 7.4: Automaton for LSC in figure 7.1 using weak interpretation

7.3 Related Work

Time constraints in sequence charts have been addressed rather uniformly by other approaches. In the field of MSCs almost all researchers dealing with time add the timing interval, often also called delay interval, to the standard MSC timers. It has to be noted, though, that all these publications are prior to the MSC-2000 standard, which provides a richer set of timing features (cf. section 3.1.3). All approaches, which deal with timed SDs only use the textual capabilities to express timing constraints, since they are more general and more expressive than the one graphical notation. Surprisingly no one seems to have missed a more general graphical way to specify timing constraints in SDs.

The original LSC paper [DH01] does not consider time at all. Harel and Marelly [HM02] extend LSCs by an explicit clock and special variables of

domain time. Time variables are assigned the current clock value and can be queried within conditions. Their approach is similar to the one we propose for specifying timing constraints in terms of supersteps in STATEMATE's asynchronous semantics (cf. section 11.3), except that their counter is unbounded, which is unsuitable for model checking. By using conditions [HM02] are able to encode complex timing constraints more concisely. The distinction between mandatory and possible conditions allows to specify assumptions about the time behavior of the environment with the same effect as placing timing constraints on the environment axis in our LSCs (cf. chapter 8). However, we feel that our graphical representation of timing constraints is more intuitive.

Alur et al. [AHP96] extend MSC-96 by time intervals on subsequent locations and (asynchronous) messages, but do not provide a formal semantics for either untimed or timed MSCs. Infinity is allowed as an upper bound, thus allowing to specify unbounded liveness, which is covered by location and message temperatures in our approach. The focus of this article is the analysis of MSCs, including the checking for timing consistency. The used method is a conversion of the MSC into a graph, whose nodes are the events (atoms in LSC terminology) of the MSC. Edges are inserted for all timing annotations, with unannotated messages and instance segments being assigned the default interval $(0, \infty)$, and weighted according to the upper bounds for the occurrence of each pair of events. Timing consistency is then checked by computing negative cost cycles in this graph, where the existence of a negative cost cycle between two events means that there is no run, which can fulfill the timing constraints imposed by the MSC. This kind of check corresponds to the existential verification of LSCs (cf. chapter 10).

Leue and Ben-Abdallah [BAL97a, BAL97b] use the same approach as [AHP96] for basic MSCs and extend it to HMSCs by checking every path, i.e. every possible concatenation of basic MSCs in the HMSC graph, for timing consistency. All three approaches also consider other consistency checks more concerned with implementation issues, e.g. detection of process divergence, influence of queuing strategies, etc.

Grabowski et al. [GDO98] also extend MSC-96 by time intervals and define the semantics in terms of Duration Calculus formulae via translation of MSCs into a set of Constraint Diagrams. They also allow to express unbounded liveness by assuming a default interval of $(0, \infty)$ between ordered events. The time model is dense time.

Lilius and Li [LL99a] do not provide a formal semantics for MSCs, but rather check, if an MSC is timing consistent or not. Each timing constraint, expressed by a timer or an interval, is translated into a linear inequality, thus yielding a set of linear inequalities. The problem of checking, if all timing annotations of the MSC are consistent, is then reduced to finding a solution for this group of linear inequalities, which can be solved efficiently by linear programming. The MSC is timing consistent, if a solution exists. [LL99a] deal with HMSCs in the same manner as [BAL97a, BAL97b]. The same approach is applied to Sequence Diagrams in [LL99b].

Firley et al. [FHD+99] also propose a timing consistency check for SDs via translation into timed automata, which can be analyzed by the UPPAAL model checker. The timing annotations used are the textual ones proposed in the UML standard [OMG01], i.e. labels are used to identify message send and receive events and from these labels textual constraints are formulated. The timing constraints result in clocks, which are reset and queried, in the timed automaton, similar to timed symbolic automata. As detailed in section 6.4 on page 149 UPPAAL is employed to find out, if it is possible to reach the final state of the timed automaton constructed from the SD. This procedure corresponds to the formal verification of existential LSCs we propose.

Seemann and von Gudenberg [SvG98] also use SD-style textual timing constraints and check SDs for timing consistency, but use the negative cost cycle procedure of [AHP96].

Chapter 8

Integrated Assumption Treatment

Modern computer systems typically are not stand-alone installations, but are deployed in a certain context. The train control system which serves as our running example e.g. relies on input from sensors like the information that the train has passed the crossing. On the other hand it controls other entities external to the currently developed ones, like the train brake or the barrier. Thus a system is developed with this context or *environment* in mind. Typically, assumptions about the possible values and combination of input signals in this environment are made. For the crossing we e.g. assume that the barrier can not be simultaneously open and closed. The environment can represent a number of things:

- Other soft- or hardware components, which are either part of the developed system or are completely external. The communication channel for instance belongs to the environment, if we look at the train in isolation, but it still belongs to the system under design. The operation center on the other hand is an external environmental component.
- Sensors or actuators which gather information, resp. affect the real world.
- Persons, which interact with the system.

When verifying a system under design, the model checker plays the role of the environment and supplies the inputs of the model. Since the model

checker does not know anything about the particular application, which is being developed, it may apply input combinations, which cannot occur in the real environment of the system under development, i.e. which violate the assumptions made by the designers about the deployment context. An unsuccessful verification can thus result from unexpected behavior of the environment or it is a real error in the design. The former type is often very easy for the model checker to cause by not supplying the required inputs or supplying them at the wrong time. Such violations are typically the first ones, which are produced by the model checker, but the ones of real interest are of the latter type. Since the model checker is unaware of the assumptions, which have been made about the system's environment, they have to be made explicit in order to be respected. This is called *assumption/commitment* or also *assume/guarantee* style specification (cf. [Jos93]), with the property to be checked being the commitment.

One of the advantages of LSCs is that expected environment behavior can be included in the chart itself by using dedicated environment instances as described in section 4.1.1 on page 66. On such an instance all features offered by LSCs can be used and thus assumptions about the environment can be formulated and exploited. Environment instances can be handled in two ways: Their treatment can be incorporated into the normal unwinding of the commitment LSC or the assumption part of the commitment LSC can be extracted into a separate, explicit assumption. We call the former *internal assumptions* and the latter *extracted assumptions*. A third type of assumptions is also considered in this chapter: assumptions, which the user specifies separately from the commitment LSC, i.e. which do not use the environment instances of the commitment LSC. We subsume extracted and user specified assumptions under the term *external assumptions*.

It has to be noted, that internal assumptions are — strictly speaking — not proper assumptions, but rather have the same semantical effect. Technically, assumptions are typically handled by intersecting the runs allowed by the assumptions with the runs allowed by the model; see e.g. [Jos93]. Assumptions thus restrict the runs a model can perform and consequently the number of possible interactions for the model checker. Internal assumptions do not decrease the number of allowed runs of the model, but rather deactivate the LSC (via an exit), if a run does not conform to the expectation specified on the environment axis. An actual restriction of system runs can be achieved by using extracted assumptions.

Internal assumptions are presented in section 8.1, while external assumptions are discussed in section 8.2, with extracted assumptions being the focus in section 8.2.1 and user specified ones in section 8.2.2. The semantics of an LSC with associated external assumptions is defined in section 8.3. This chapter closes with a review of related work in section 8.4.

8.1 Internal Assumptions

Internal assumptions aim at not allowing the model checker to violate an LSC specification by simply applying unexpected inputs. The expected environment behavior is specified by the user on the dedicated environment instances using the normal LSC features (cf. section 4.1.1 on page 66). The way to achieve this goal is to treat wrong behaviors of the environment not as errors, but as a deactivation of the LSC, i.e. if the environment does not conform to the expected behavior (as given by the elements on the environment instance axes), this is treated as an exit from the LSC, similar to possible conditions or possible local invariants. Thus the model checker has to fulfill the assumptions expressed on the environment instances in order to progress through the automaton and possibly find a real error in the model.

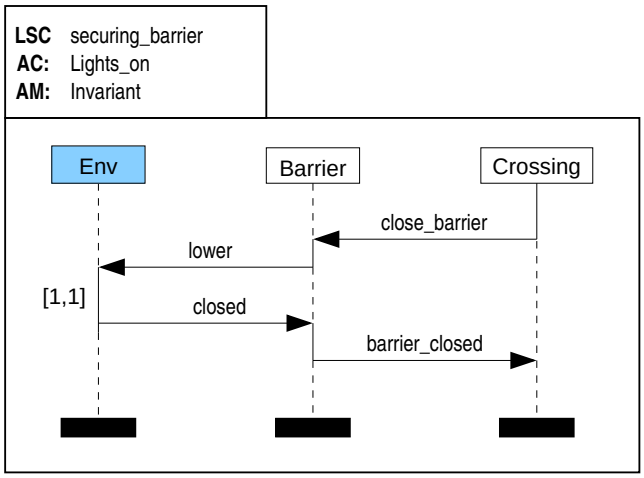


Figure 8.1: LSC for the Crossing component specifying the barrier closing

Figure 8.1 shows an example from the train control system case study: the protocol for successfully closing the barrier. Once the lights have been turned

on (`AC: Lights_on`) the crossing controller instructs the barrier controller to close the barrier, which in turn sends the signal to the barrier actuator, which is part of the environment. As is indicated by the hot part of the environment axis and the attached timing interval, the barrier is expected to be closed after one time unit and this result must be reported back to the crossing controller. The environment axis thus expresses the assumption that the barrier is operational and closes in time. Making the LSC robust against a misbehaving environment in this case means that the LSC is deactivated, if the `closed` message does not arrive at all or not within the given interval (which would typically be captured in a seprate LSC). The special treatment necessary for each LSC element defined on the environment instance is listed in the following:

instantaneous messages Need no special treatment, since it is either observed or not. The occurrence of an instantaneous message can be required only by a hot location temperature. In the strict interpretation messages sent by the environment must not result in an error, if they occur at a time when they are not supposed to. The correct treatment rather is to exit from the LSC. This means that for each state in the automaton, where such a message is not supposed to be observed, a transition to the exit state must be added.

asynchronous messages sent from the environment Need no special treatment. As for instantaneous messages the location temperature indicates, if a message must be sent. The message temperature is always hot, since an assumption about a message, which need not arrive, is useless. In the strict interpretation the same treatment as for instantaneous messages is necessary.

asynchronous messages received by the environment The message temperature is set to cold, because the environment can not be forced to receive the message. In the strict interpretation the same treatment as for the previous message types is necessary.

conditions All conditions on the environment instance are treated as possible ones, since the environment can not be forced to satisfy a given condition. Thus, exits are inserted into the automaton for all conditions defined on the environment axis regardless of their mode.

local invariants They are treated as possible ones, analogously to conditions.

timer and timing intervals Timing constraints of the environment, which are violated, are treated as exits from the LSC. This means that additional transitions to the exit node are inserted into the automaton and are annotated by the negation of the constraints.

Recall that timing intervals of the form $[n, m]$ constrain two clusters, e.g. two messages, and translate into a clock constraint of the form $[z_3 \geq n \wedge z_3 \leq m]$ in the automaton. This timing annotation is violated, if the second atom, e.g. $msgX$ occurs either too early or too late, which translates into the following annotation for the transition to the exit state: $msgX[z_3 < n \vee z_3 > m]$. Open and half-open timing intervals are treated analogously.

Timeouts result in clock constraints of the form $[z_1 = dur]$, where dur is the duration given for the timer. Assuming that the timeout is bound to some message $msgX$ this leads to an annotation of the corresponding exit transition of $msgX[z_1 < dur \vee z_1 > dur]$. Timer resets result in clock constraints of the form $[z_1 < dur]$, which leads to an annotation of the corresponding exit transition of $msgX[z_1 \geq dur]$.

location temperatures Location temperatures are set to cold, because progress of the environment can not be enforced. The system under development can e.g. not force the environment to respond to a request; this requires an explicit assumption (see section 8.2.1 on page 195 below).

simultaneous regions A simultaneous region is violated, if not all messages of this region are observed simultaneously; other LSC elements are already taken care of by the procedures described above. Thus, whenever only a strict subset of messages of the simultaneous region is observed, this should result in an exit, if one of the missing messages should have been sent by the environment. The same treatment is necessary for simultaneous regions, which are not defined on the environment axis, but which contain messages sent by the environment.

coregions No special treatment necessary.

8.1.1 Adjustment of the Formal Semantics

Since the treatment of LSC elements differs depending on the type of instance axis (environment or normal) they appear on, this information has to be made available to the unwinding algorithm. This is done by the function $envInst(\cdot)$, which returns $true$, if an instance is an environment instance and $false$, if it is a normal instance:

$$envInst : Instances(l) \longrightarrow \mathbf{B}$$

For convenience's sake similar functions are defined for atoms, resp. clusters of each instance:

$$isEnv : Atoms(i) \longrightarrow \mathbf{B}$$

$$isEnv(a) := \begin{cases} true & \text{if } envInst(i) = true \\ false & \text{else} \end{cases}, a \in Atoms(i)$$

$$isEnv : Clusters(i) \longrightarrow \mathbf{B}$$

$$isEnv(cl) := \begin{cases} true & \text{if } \exists a \in cl : isEnv(a) = true \\ false & \text{else} \end{cases}, cl \in Clusters(i)$$

Since hot location temperatures on the environment instance must be treated as cold, the computation of the cut temperature has to be adjusted accordingly. This ensures that the model checker can not violate the property simply by doing nothing, provided progress only hinges on the environment instance(s). If progress is additionally required by the commitment part of the LSC, then this may still result in a violation depending on the system behavior. This exemplifies the benefit of assigned a hot temperature only to those locations of an LSC, which are actually responsible for progressing.

Thus the function $cut\_temp(\cdot)$ (cf. definition 6.11 on page 117), which computes the cut temperature and thus determines the set of fair states, is extended in the following manner: A hot location temperature is only regarded, if the location is not situated on the environment instance.

8.1 Definition (Cut Temperature)

$$cut\_temp : Cuts(l) \longrightarrow \{hot, cold\}$$

$$cut\_temp(cut) := \begin{cases} hot & \text{if } \exists\ cl_j \in cut : temp(location(cl_j)) = hot \\ & \qquad \wedge \neg isEnv(cl_j) \\ cold & \text{else} \end{cases},$$

for $cut = (cl_1, \ldots, cl_n), 1 \leq j \leq n$. ■

8.1.2 Unwinding Algorithm for Internal Assumptions

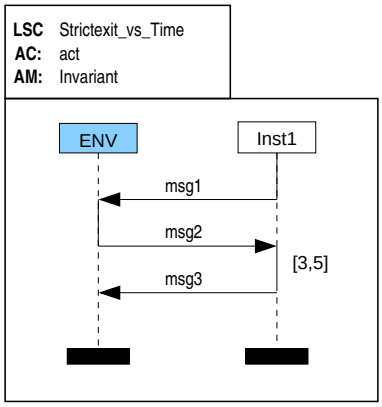


Figure 8.2: LSC illustrating the relation between timing constraints and possible conditions

The correct treatment of the other elements described above requires the adjustment of the unwinding algorithm, which is presented in the following. Special care has to be taken for exits caused by environment messages in the unwinding, similar to the case of possible conditions and local invariants, which are covered by a timing annotation (see page 159). The LSC shown in figure 8.2 and the corresponding TSA (figure 8.3) illustrate the problem. The occurrence of `msg3` is constrained to at least three and at most five time units after observing `msg2`. The treatment of internal assumptions prescribes that the two transitions to the exit state are added. The automaton shown in figure 8.3 is incorrect inasmuch as a second receipt of `msg2` after more

than five time units, assuming `msg3` has not been observed within this time, results in satisfaction of the LSC, even though the timing constraint for `msg3` is violated. Once the upper limit given by the timing constraint has been reached without observing the expected message, an out of turn receipt of messages sent by the environment should thus not be interpreted as an exit but as an error, since at this point it is impossible to still satisfy the specified property.

This problem is overcome by adding the upper bound of the timing constraint to the transition to the exit state. The lower bound must not be used, since observing the environment message before the expected message must not result in an error.

Algorithm 8.1 on the facing page shows the main part, which is largely unaffected by the adjustments. The special treatment for local invariants on the environment axis is done in line 15 and 16, where the initial sets of active possible and mandatory local invariants are computed. Local invariants defined on the environment axis are added to the possible set in line 15, regardless of their mode. The mandatory set is consequently restricted to those local invariants, which are specified on a normal instance (line 16). Additionally note that the changed cut temperature computation takes effect in lines 9 and 38.

Algorithm 8.2 on page 180, which takes care of the transitions to the exit state, is primarily affected by the adjustments. Because mandatory con-

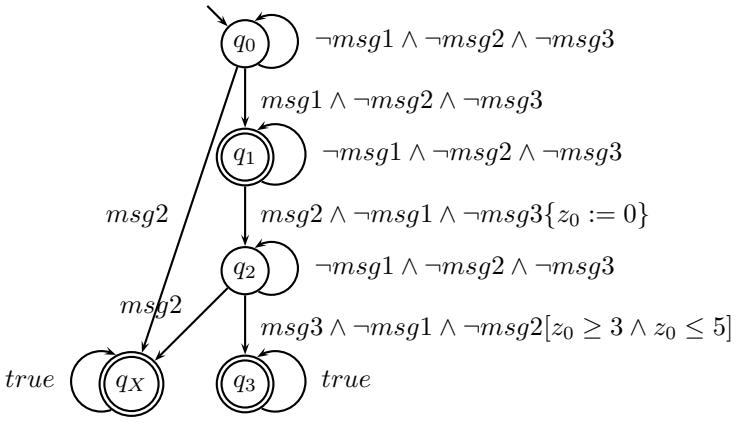


Figure 8.3: Automata for LSC `Strictexit_vs_Time` (strict interpretation)

Algorithm 8.1 Timed Unwinding Algorithm (Internal Assumptions)

1: **if** $interpretation$ = strict **then**
2: $sc :=$ NEG_CONJUNCT($MsgLabels(l)$)
3: **end if**
4: $\Sigma := MsgLabels(l) \cup Conditions(l) \cup Local\_Invariants(l) \cup \{true, false\}$
5: $C := \{z_0, z_1, \ldots, z_k\}$
6: $phases := \{Phase_0\}$
7: $Q := \{\text{STATE}(Phase_0), q_X\}$
8: $q_0 := \{\text{STATE}(Phase_0)\}$
9: **if** $cut\_temp(Cut_0) = cold$ **then**
10: $F := Q$
11: **else**
12: $F := \{q_X\}$
13: **end if**
14: $\longrightarrow := \{(q_X, true, \emptyset, \epsilon, q_X)\}$
15: $poss\_invs_0 := \{li \in Local\_Invariants(l) \mid \exists scl\, \exists cl \in scl\, \exists a, a' \in cl :$
$a \in Instheads(l) \wedge a' \in LI\_starts(l) \wedge liID(a') = li \wedge (mode(a') = possible \vee isEnv(a'))\}$
16: $mand\_invs_0 := \{li \in Local\_Invariants(l) \mid \exists scl\, \exists cl \in scl\, \exists a, a' \in cl :$
$a \in Instheads(l) \wedge a' \in LI\_starts(l) \wedge liID(a') = li \wedge$
$mode(a') = mandatory \wedge \neg isEnv(a')\}$
17: **while** $phases \neq \emptyset$ **do**
18: **let** $ph = (Ready_i, History_i, Cut_i) \in phases$
19: **if** $Ready_i = \emptyset$ **then**
20: $F := F \cup \{\text{STATE}(ph)\}$
21: $\longrightarrow := \longrightarrow \cup \{(\text{STATE}(ph), true, \emptyset, \epsilon, \text{STATE}(ph))\}$
22: **else**
23: GENERATE_EXITS($Ready_i, ph$)
24: **for all** $Fired_{i_k} \in \mathcal{P}(Ready_i)$ **do**
25: $successor := Step(ph, Fired_{i_k})$
26: **let** $successor = (Ready_j, History_j, Cut_j)$
27: **if** $successor = ph$ **then**
28: INSERT_SELF_LOOP($ph, Ready_i$)
29: **else**
30: $TransitionAnnot :=$CONSTRUCT_TRANSITION($ph, successor,$
$Ready_i, Fired_{i_k}$)
31: $ClkResets :=$ COMPUTE_RESETS($Fired_{i_k}$)
32: $ClkConstr :=$ COMPUTE_CONSTRAINTS($Fired_{i_k}$)
33: **if** $\exists q \in Q$ *with* $\text{STATE}^{-1}(q) = (Ready_x, History_x, Cut_x) : Ready_i = Ready_x \wedge History_i = History_x$ **then**
34: $successor := \text{STATE}^{-1}(q)$
35: **else**
36: $Q := Q \cup \{\text{STATE}(successor)\}$
37: $phases := (phases \setminus \{ph\}) \cup \{successor\}$
38: **if** $cut\_temp(Cut_i) = cold$ **then**
39: $F := F \cup \{\text{STATE}(ph)\}$
40: **end if**
41: **end if**
42: $\longrightarrow := \longrightarrow \cup \{(\text{STATE}(ph), TransitionAnnot, ClkResets,$
$ClkConstr, \text{STATE}(successor))\}$
43: **end if**
44: INSERT_SKIP_TRANSITION($TransitionAnnot, ph, successor, ClkResets, ClkConstr$)
45: **end for**
46: **end if**
47: **end while**

Algorithm 8.2 Generate Exits($Ready_i, ph$)

```
 1: ExitAnnot := false
 2: for all scl ∈ SimClasses(SimRegs(Ready_i)) do
 3:   if (Poss_CondIDs(scl)∪Mand_Env_CondIDs(scl)) ≠ ∅ then
 4:     ExitAnnot := ExitAnnot ∨
            (Neg_Disjunct(Poss_CondIDs(scl) ∪ Mand_Env_CondIDs) ∧
              Conjunct(MsgLabels(scl)))
 5:   end if
 6:   for all lis ∈ LIStarts(Ready_i) do
 7:     if lis ∈ scl ∧ (mode(liID(lis)) = possible ∨ isEnv(lis)) then
 8:       if incl(lis) then
 9:         ExitAnnot := ExitAnnot ∨ (¬liID(lis)∧
                Conjunct(MsgLabels(scl)))
10:       else
11:         ExitAnnot := ExitAnnot ∨ ¬liID(lis)
12:       end if
13:     end if
14:   end for
15:   for all lie ∈ LIEnds(Ready_i) do
16:     if lie ∈ scl ∧ (mode(liID(lie)) = possible ∨ isEnv(lie)) then
17:       if incl(lie) then
18:         ExitAnnot := ExitAnnot ∨ (¬liID(lie)∧
                Neg_Conjunct(MsgLabels(scl)))
19:       else
20:         ExitAnnot := ExitAnnot ∨ ¬liID(lie)
21:       end if
22:     end if
23:   end for
24:   if EnvMsgs(scl) ≠ ∅) ∧ |MsgLabels(scl)| > 1 then
25:     ExitAnnot := ExitAnnot ∨ Handle_Env_SimReg(scl)
26:   end if
27: end for
28: for all pc ∈ (Poss_CondIDs(Ready_i) ∪ Mand_Env_CondIDs(Ready_i)) \
    (Poss_CondIDs(SimRegs(Ready_i)) ∪ Mand_Env_CondIDs(SimRegs(Ready_i))) do
29:   ExitAnnot := ExitAnnot ∨ ¬pc
30: end for
31:
32: Add_Env_Msg_Exits(Ready_i, ph)
33:
34: ClkConstr := Compute_Constraints(Ready_i)
35: if ExitAnnot ≠ false then
36:   ⟶:=⟶ ∪ {(State(ph), ExitAnnot, ∅, ClkConstr, q_X)}
37: end if
38:
39: Handle_Env_Timer(ph, Ready_i)
40: Handle_Env_Timing_Intervals(ph, Ready_i)
```

ditions on the environment axis should be treated as possible ones, these have additionally to be taken into account, when generating the exit transitions for cold conditions. Mandatory conditions specified within simultaneous regions on the environment instance — given by the auxiliary function Mand_Env_CondIDs — are treated as possible conditions by adding them

to the set of possible conditions in lines 3 and 4 of algorithm 8.2. Solitary mandatory environment conditions are analogously added to the corresponding sets of solitary possible conditions in line 28.

Local invariants require a similar treatment, which is carried out in lines 7 and 16 by considering not only the mode of a local invariant, but also if it is defined on the environment instance using the $isEnv(\cdot)$ function.

Lines 24 and 25 are concerned with the treatment of simultaneous regions, which contain messages sent by the environment. Such a simultaneous region can only be violated by the environment, if it contains at least two messages (second part of the conjunction in line 24) and one of them is sent by the environment (first part of the conjunction). The auxiliary function EnvMsgs(scl) computes the message atoms, which are either sent by the environment or are asynchronous receipts of environment messages, i.e. it sums up all messages of the environment, which can interfere with the specified property, if they do not occur when expected. All other combinations of LSC elements in a simultaneous region are already handled by other parts of this algorithm (one message and a number of conditions by lines 3 and 4, local invariants by lines 6 to 20, timer and timing intervals by lines 39 and 40). The generation of this part of the exit transition annotation is done by function Handle_Env_SimReg, which is described in algorithm 8.5 on page 183. The treatment of messages sent by the environment in the strict interpretation is done by function Add_Env_Msg_Exits (line 32), whose details are given by algorithm 8.3.

Algorithm 8.3 Add Env Msg Exits ($Ready_i, ph$)

```
 1: ExitAnnot := true
 2: if interpretation = strict then
 3:    for all m ∈ (EnvMsgSndLabels(l) \ MsgLabels(Ready_i)) do
 4:       ExitAnnot := ExitAnnot ∨ m
 5:    end for
 6: end if
 7: ClkConstr := Compute_Constraints_Upper_Bounds(Ready_i)
 8:
 9: if ExitAnnot ≠ true then
10:    ⟶:=⟶ ∪ (State(ph), ExitAnnot, ∅, ClkConstr, q_X)
11: end if
```

Algorithm 8.4 *ClkConstr* Compute Constraints Upper Bounds($Ready_i$)

```
 1: for all timres ∈ TimerResetIDs(Ready_i) do
 2:    ClkConstr := ClkConstr ∧ ClkName(timres) < duration(timres)
 3: end for
 4:
 5: for all to ∈ TimeoutIDs(Ready_i) do
 6:    ClkConstr := ClkConstr ∧ ClkName(to) = duration(to)
 7: end for
 8:
 9: for all loc ∈ PredecLocs(Ready_i) do
10:    if upIncl(loc) then
11:       ClkConstr := ClkConstr ∧ ClkName(loc) ≤ upBound(loc)
12:    else
13:       ClkConstr := ClkConstr ∧ ClkName(loc) < upBound(loc)
14:    end if
15: end for
16:
17: for all amr ∈ AsyncMsgRcvs(Ready_i) : ∃ams ∈ AsyncMsgSnd(l) :
    msgID(amr) = msgID(ams) ∧ lowBound(ams) ≠ ε do
18:    if upIncl(ams) then
19:       ClkConstr := ClkConstr ∧ ClkName(ams) ≤ upBound(ams)
20:    else
21:       ClkConstr := ClkConstr ∧ ClkName(ams) < upBound(ams)
22:    end if
23: end for
```

Lines 2 to 4 of algorithm 8.3 handle exits in the strict interpretation, which are due to messages sent by the environment at a wrong point in time. $EnvMsgSndLabels(l)$ is the set of all messages, which are sent by the environment: $EnvMsgSndLabels(l) := \{msglabel \in MsgLabels(l) \mid \exists m \in Msgsnd(l) : isEnv(m) \wedge msgLabel(m) = msglabel\}$. Only those message labels must be included in the exit transition annotation, which are not enabled in the current state, i.e. which are not in the current ready set. Thus, these message labels are removed from the environment message labels in line 3. All other messages sent by the environment are added disjunctively to the exit transition annotation (line 4).

Function Compute_Constraints_Upper_Bounds in line 7 collects the upper bounds of the clock constraints, which appear on non-exit transitions in the current unwinding step, and returns their conjunction. The details of this function are given by algorithm 8.4; the algorithm is identical to the algorithm computing the clock constraints for the normal transitions

(cf. algorithm 7.4 on page 165), except that the lower bounds are not considered. The remainder of algorithm 8.3 inserts the transition to the exit state, if needed. Note that this transition, if present, is separate from the one, which is computed in algorithm 8.2, because the transition, which is generated by algorithm 8.2 may in general contain clock constraints with upper and lower bounds, whereas the transition generated by algorithm 8.3 must not contain lower bounds. Since transition annotations and clock constraints can not be mixed arbitrarily, a separate transition is required (see the explanation of algorithm 8.6 below for details).

Algorithm 8.5 *ExitAnnot* Handle Env SimRegs(*scl*)

```
 1: let msgpset = 𝒫(MsgLabels(scl))
 2: for all msgs ∈ msgpset do
 3:    if msgs ≠ ∅ ∧ msgs ≠ msgpset ∧ EnvMsgs(MsgLabels(scl) \ msgs) ≠ ∅ then
 4:       ExitAnnot := ExitAnnot ∨ (Conjunct(msgs)∧
              Neg_Conjunct(MsgLabels(scl) \ msgs))
 5:    end if
 6: end for
 7: return ExitAnnot
```

Algorithm 8.6 Handle Env Timer($ph, Ready_i$)

```
 1: for all timres ∈ TimerResets(Ready_i) : isEnv(timres) = true do
 2:    ⟶:=⟶ ∪(State(ph), ¬ StableCond(ph), ∅,
          ClkName(timres) ≥ duration(timres), q_X)
 3: end for
 4:
 5: for all to ∈ Timeouts(Ready_i) : isEnv(to) = true do
 6:    let scl = SimClass(to)
 7:    if scl ≠ ∅ then
 8:       exit_annot := Conjunct(MsgLabels(scl)
 9:    else
10:       exit_annot := true
11:    end if
12:    ⟶:=⟶ ∪(State(ph), exit_annot, ∅, ClkName(to) < duration(to), q_X)
13:    ⟶:=⟶ ∪(State(ph), ¬ StableCond(ph), ∅,
          ClkName(to) > duration(to), q_X)
14: end for
```

Exits due to violations of timing constraints by the environment are handled by the functions Handle_Env_Timer and Handle_Env_Timing_Intervals, which are presented in algorithm 8.6, resp. algorithm 8.7 on the next page.

Algorithm 8.7 Handle Env Timing Intervals($ph, Ready_i$)

1: **for all** $loc \in$ PredecLocs($Ready_i$) **do**
2: **let** $loc' =$ Succ(loc)
3: **let** $scl =$ SimClass(loc')
4: **if** $\exists cl =$ Cluster(loc') : $isEnv(cl) = true$ **then**
5: $exit\_annot :=$ Conjunct(MsgLabels(scl))
6: **if** $lowIncl(loc)$ **then**
7: $ExitConstr :=$ ClkName(loc) $< lowBound(loc)$
8: **else**
9: $ExitConstr :=$ ClkName(loc) $\leq lowBound(loc)$
10: **end if**
11: $\longrightarrow := \longrightarrow \cup$(State($ph$), $exit\_annot, \emptyset, ExitConstr, q_X$)
12:
13: **if** $upIncl(loc)$ **then**
14: $ExitConstr := ExitConstr \vee$ ClkName(loc) $> upBound(loc)$
15: **else**
16: $ExitConstr := ExitConstr \vee$ ClkName(loc) $\geq upBound(loc)$
17: **end if**
18: $\longrightarrow := \longrightarrow \cup$(State($ph$), $\neg$ StableCond(ph), $\emptyset, ExitConstr, q_X$)
19: **end if**
20: **end for**
21:
22: **for all** $amr \in$ AsyncMsgRcvs($Ready_i$) : $\exists ams \in AsyncMsgSnd(l) : msgID(amr) = msgID(ams) \wedge lowBound(ams) \neq \epsilon \wedge (isEnv(amr) = true \vee isEnv(ams) = true)$ **do**
23:
24: **if** $isEnv(amr) = true$ **then**
25: **let** $scl =$ SimClass(amr)
26: **else**
27: **let** $scl =$ SimClass(ams)
28: **end if**
29: $exit\_annot :=$ Conjunct(MsgLabels(scl))
30: **if** $lowIncl(ams)$ **then**
31: $ExitConstr :=$ ClkName(ams) $< lowBound(ams)$
32: **else**
33: $ExitConstr :=$ ClkName(ams) $\leq lowBound(ams)$
34: **end if**
35: $\longrightarrow := \longrightarrow \cup$(State($ph$), $exit\_annot, \emptyset, ExitConstr, q_X$)
36:
37: **if** $upIncl(ams)$ **then**
38: $ExitConstr := ExitConstr \vee$ ClkName(ams) $> upBound(ams)$
39: **else**
40: $ExitConstr := ExitConstr \vee$ ClkName(ams) $\geq upBound(ams)$
41: **end if**
42: $\longrightarrow := \longrightarrow \cup$(State($ph$), $\neg$ StableCond(Stateph), $\emptyset, ExitConstr, q_X$)
43: **end for**

Algorithm 8.5 on page 183 generates the part of the exit transition annotation, which deals with simultaneous regions, which depend on messages sent by the environment. If the environment does not send the messages it is supposed to send within the considered simultaneous region[1], this should result in a transition to the exit state. This means that all combinations of messages, which are part of the simultaneous region, have to be considered. Therefore the powerset of all messages in the current simultaneous region is computed in line 1. Each set of the powerset corresponds to one combination of messages and is interpreted here as the occurrence of the messages contained in the set. All messages, which are missing from the set, i.e. the complementary set, is interpreted as messages, which are not observed.

The empty set and the set containing all messages can be ignored, as they correspond to the self loop, resp. the good case. Since only violations caused by the environment should lead to an exit, it is sufficient to add a corresponding annotation, if one of the messages sent by the environment is missing (line 3). If this is the case, the observed messages ($msgs$) are recorded positively in the exit transition annotation and all other messages in a negated form (line 4).

Algorithm 8.6 on page 183 constructs the exits for timing constraints, which are due to timeouts and timer resets and which are violated by the environment. The first part (lines 1 to 2) considers violated timer resets, the second part (lines 5 to 13) violated timeouts. The transition annotation is the negation of the stable condition and the clock constraint for the exit transition is the negation of the normal constraint for a timer reset (line 2; also cf. algorithm 7.4 on page 165). The annotation is the negation of the stable condition rather than the single label of the message to which the timer refers, because otherwise in the strict interpretation the occurrence of any message contained in the LSC would lead to an undesired error, once the upper bound given by the timer reset is violated. The auxiliary function StableCond(ph) returns the annotation of the self loop on the state corresponding to the current phase.

The treatment of violated timeouts is different inasmuch as it requires two separate transitions to the exit state, each carrying one part of the negation of the original clock constraint. The computation of the transition annota-

[1]We are using the term *simultaneous region* here out of convenience, although we are actually dealing with the SimClass, which contains the simultaneous region under consideration.

tion for the first transition (lines 6 to 10) covers the early occurrence of the constrained messages. The auxiliary function SimClass(to) yields the SimClass, which contains the current timeout atom (line 6). The annotation then carries the conjunction of all messages occurring in the considered SimClass (line 8) or simply $true$, if the timeout is isolated (line 10). A separate second condition is needed, because a different transition annotation is required for the case that the expected messages arrive too late or not at all (line 13). We refer the reader to example 8.1 on page 189 for a detailed justification and explanation why the second, different annotation is needed and why timer resets are treated in the way described above.

Algorithm 8.8 *transAnnot* Construct Transition($ph, successor, Ready_i, Fired_{i_k}$)

```
1:  mand_invs_j := mand_invs_i
2:  poss_invs_j := poss_invs_i
3:
4:  transAnnot := HANDLE_LOCAL_INVARIANT_ENDS(true,
            Fired_{i_k}, Ready_i, j)
5:
6:  let invs_j = poss_invs_j ∪ mand_invs_j
7:  let cmr = COLDMSGRCVLABELS(Fired_{i_k}) ∪
              ENVHOTASYNCRCVLABELS(Fired_{i_k})
8:  let msr = MSGLABELS(SIMREGS(Fired_{i_k}))
9:  if cmr ≠ ∅ then
10:   transAnnot :=GENERATE_MAIN_ANNOTATION(Fired_{i_k}, cmr, msr, invs_j,
          transAnnot)
11: end if
12: if interpretation = strict then
13:   transAnnot := transAnnot ∧
          NEG_CONJUNCT(MsgLabels(l) \ Fired_{i_k})
14: end if
15: if |SimClasses(Ready_i)| > 1 ∧ interpretation = weak then
16:   transAnnot := transAnnot∧
          NEG_CONJUNCT(MSGLABELS(Ready_i \ Fired_{i_k}))
17: end if
18:
19: transAnnot :=HANDLE_LOCAL_INVARIANT_STARTS(transAnnot,
        Fired_{i_k}, Ready_i, j)
20:
21: return transAnnot
```

Different transition annotations, which are associated with distinct clock constraints, always require separate transitions. Assume for instance that there are two messages, $msgX$ and $msgY$, with clock constraints $clkconstrX$, resp. $clkconstrY$. Considered in isolation the transition annotation would be $msgX[clkconstrX]$, resp. $msgY[clkconstrY]$ or more explicitly $msgX \wedge clkconstrX$, resp. $msgY \wedge clkconstrY$. Using a single transition for both messages including clock constraints, i.e. disjunctively adding annotations and constraints would result for instance in $(msgX \vee msgY) \wedge (clkconstrX \vee clkconstrY)$. This is incorrect, since the association of each message to its constraint is lost. The correct annotation should read $(msgX \wedge clkconstrX) \vee (msgY \wedge clkconstrY)$ and can only be expressed by two separate transitions; cf. example 8.1.

Algorithm 8.7 on page 184 constructs the exit transitions for violated timing intervals. The basic procedure is similar to the one used in algorithm 8.6 on page 183, although the concrete execution is slightly different. As in algorithm 7.4 on page 165 the function PredecLocs($Ready_i$) yields for all locations in the current ready set the set of their immediate predecessor locations, which are annotated by a timing interval (line 1). In line 2 the location, which marks the end of the timing interval, i.e. the successor of loc, is computed by the auxiliary function Succ(loc), which is used to find the SimClass containing this location (line 3). Line 4 determines if there is a cluster (given by the auxiliary function Cluster(loc'); line 4) defined on the environment instance in this location. The SimClass is needed for the transition annotation (line 5).

Since a timing interval is always bound to two locations, the annotation cannot be empty as for timer resets or timeouts. Similar to the treatment of violated timeouts, two separate transitions to the exit state are needed (see example 8.1 on page 189 for the details). The clock constraint for the case where the lower bound is violated is constructed in lines 6 to 9, depending on the nature of the interval (open, half-open, closed). The constraint is the negation of the normal constraint, as for timer resets and timeouts. The corresponding transition is added to $\longrightarrow$ in line 11. The transition for the case where the upper bound is violated is constructed analogously in lines 13 to 18. The treatment for timing intervals of asynchronous messages is analogous; the transition annotation depends on the message atom, which is defined on the environment instance (lines 22 to 42).

Algorithm 8.8 on the facing page shows the adjusted version of the algorithm for the construction of the transition to the successor state. Only a

minor adjustment is necessary here: the hot asynchronous messages, which are received by the environment (given by the auxiliary function EnvHotAsyncRcvLabels ($Fired_{i_k}$)) are added to the cold asynchronous messages in line 7, thus treating them as cold ones.

Algorithm 8.9 *TransAnnot* Handle Local Invariant Ends($TransAnnot, Fired_{i_k}, Ready_i, j$)

```
1:  for all lie ∈ LIEnds(Ready_i) do
2:    if incl(lie) then
3:      TransAnnot := TransAnnot ∧ liID(lie)
4:    end if
5:    if mode(liID(lie)) = mandatory ∧ ¬isEnv(lie) then
6:      mand_invs_j := mand_invs_j \ {liID(lie)}
7:    else
8:      poss_invs_j := poss_invs_j \ {liID(lie)}
9:    end if
10: end for
11:
12: return TransAnnot
```

Algorithm 8.10 *TransAnnot* Handle Local Invariant Starts($TransAnnot, Fired_{i_k}, Ready_i, j$)

```
1:  for all lis ∈ LIStarts(Ready_i) do
2:    if incl(lis) then
3:      TransAnnot := TransAnnot ∧ liID(lis)
4:    end if
5:    if mode(liID(lis)) = mandatory ∧ ¬isEnv(lis) then
6:      mand_invs_j := mand_invs_j ∪ {liID(lis)}
7:    else
8:      poss_invs_j := poss_invs_j ∪ {liID(lis)}
9:    end if
10: end for
11:
12: return TransAnnot
```

The algorithms for handling local invariants, algorithm 8.9 and algorithm 8.10, also require only a minor adjustment. Those mandatory local invariants, which are defined on the environment instance, need to be added to, resp. subtracted from the set of possible local invariants instead of mandatory ones. Thus, local invariants are only added to/subtracted from the set

Algorithm 8.11 Insert Skip Transition($TransitionAnnot, ph, successor, ClkResets, ClkConstr$)

```
1: for all (q_x, ψ, ρ, γ, STATE^{-1}(ph)) ∈⟶ do
2:    let (Ready_x, History_x, Cut_x) = STATE^{-1}(q_x)
3:    let cmsgs = COLDMSGRCVLABELS(Ready_x) ∪
                  ENVHOTASYNCRCVLABELS(Ready_x)
4:    if cmsgs =ATOMS(Ready_x) then
5:       ⟶:=⟶ ∪ (q_x, TransitionAnnot, ClkResets,
                  ClkConstr, successor)
6:    end if
7: end for
```

of mandatory local invariants, if they are not defined on the environment instance (line 5 in both algorithms).

Algorithm 8.11, which computes the skipping transition for cold asynchronous messages, also has to be altered, so that it considers hot asynchronous messages received by the environment as cold ones as well. This is done in line 3.

EXAMPLE 8.1

Figure 8.4 on the next page shows the automaton for the LSC in figure 8.1 on page 173 for the weak interpretation, which treats the hot part and timing interval of the environment instance correctly. Note that state q_2 is fair, because the hot temperature of the location on the environment instance is disregarded. This disallows the model checker to violate the LSC by never sending the `closed` message. Assuming that the LSC specification holds up to this point, intuitively stated, the model checker now has to send the `closed` message in order to proceed to the lower part of the automaton of figure 8.4 and check for a possible violation there.

Even though the model checker can not violate the LSC specification by never sending `closed`, it still can send the message at a wrong point in time, e.g. after 4 time units, thus violating the timing interval. In order to prohibit this undesired behavior, the transitions from q_2 to the exit state have been added. The reason for the two separate transition becomes apparent in the strict interpretation only.

Figure 8.5 on the next page shows an incorrect automaton for the strict interpretation; for readability the self loop annotations have been abbreviated to *sc*, which stands for $\neg close\_barrier \wedge \neg lower \wedge \neg closed \wedge \neg barrier\_closed$. In this automaton transitions to the exit state have been added for each

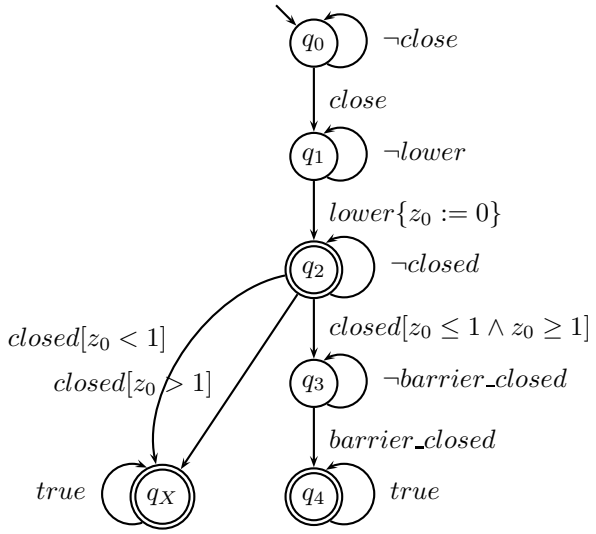


Figure 8.4: Automaton for LSC `securing_barrier` with special treatment of environment elements included (weak interpretation)

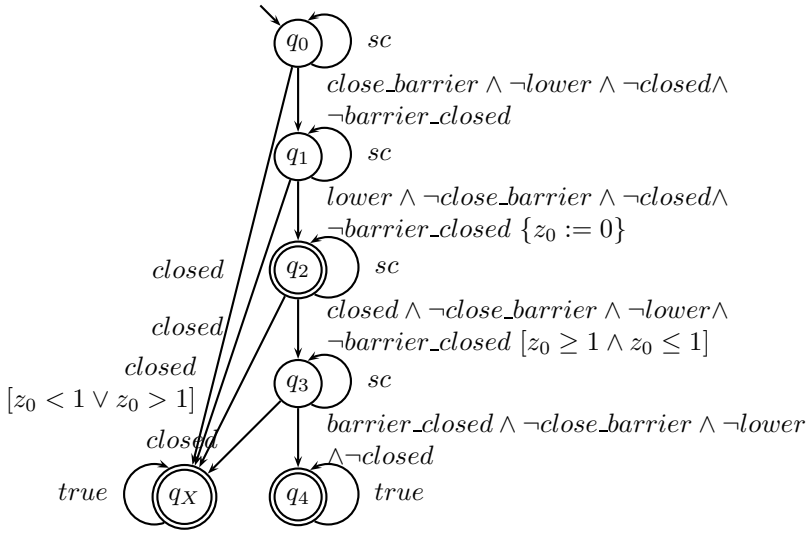


Figure 8.5: Incorrect automaton for LSC `securing_barrier` with special treatment of environment elements included (strict interpretation)

state, in which the `closed` message should not occur (q_0, q_1, and q_2). Note that the transition from q_2 is due to the timing interval and *not* because of the strict interpretation; without the timing interval this transition would be missing.

This transition is the incorrect part of the automaton, since it should be represented by two separate transitions as in the automaton of figure 8.4. It is used here to demonstrate that the straight-forward way to represent a violation of the timing interval by simply negating the original clock constraint as done here, is insufficient when using the strict interpretation. Consider the situation that the automaton in figure 8.5 has progressed to state q_2, i.e. that we are waiting to observe `closed` within the given time bound. In the weak interpretation there was no problem, if the environment never sent `closed`, since q_2 is an fair state. In the strict interpretation the case is different, because the stable condition expressed by the self loop annotation can be violated by other messages. Assume e.g. that the automaton in figure 8.5 has remained in state q_2 for five time units and now we observe `barrier_closed`: We have a violation of the property even though the environment did not obey the (internal) assumption expressed by the timing constraint.

In order to prohibit this undesired behavior by the model checker, we exploit the knowledge that after the time given by the upper bound of the timing interval has passed, the LSC can not be fulfilled anymore regardless of the occurrence of the required message. Once more time has passed than allowed by the upper bound, without observing the expected message from the environment (`closed` in our example), in the strict interpretation the occurrence of any message, which is present in the LSC, causes an error as a late effect of the not sent environment message. Consequently a violation of the stable condition of the state, where the environment message is expected, should lead to an exit, if this occurs after the time given by the upper bound has elapsed. In our example this results in a transition to the exit state with annotation $\neg sc[z_0 > 1]$ or more explicitly: $close\_barrier \vee lower \vee closed \vee barrier\_closed[z_0 > 1]$.

A second transition to the exit state is required for a violation of the lower bound, since here a different annotation and a different clock constraint is required. Before the time for the lower bound elapses only the occurrence of the environment message must result in an exit. Should any of the other messages in the LSC be observed in this time span, the consequence must be an error (in the strict interpretation). In our example here the annotation of the first transition thus reads $closed[z_0 < 1]$. This strategy is correct for the

weak interpretation as well, as the automaton in figure 8.4 illustrates.

This treatment covers all cases: If the expected message arrives too early, the first transition is taken. If it is observed within the time bound given, the normal transition is taken to the successor state in the automaton. If the message arrives too late, the second exit transition is taken. If it never arrives and the stable condition holds forever, we remain in this state. If it never arrives and the stable condition is violated after the time given by the upper bound has passed, again the second exit transition is taken. If the stable condition is violated at any other time, an error is indicated.

Figure 8.6 shows the corrected version of the automaton for LSC `securing_barrier` in the strict interpretation. Note that the special considerations above also apply to other timing constraints, which involve the violation of an upper bound, i.e. timeouts and timer resets as shown in algorithm 8.6 on page 183.

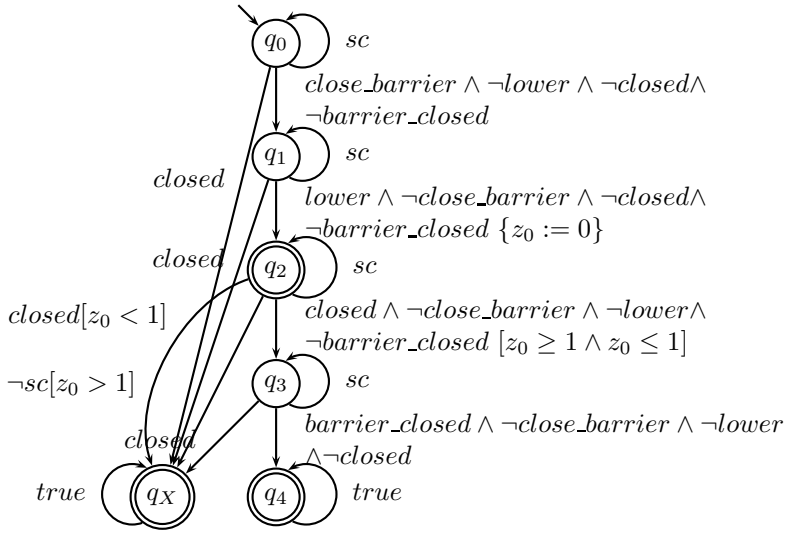


Figure 8.6: Correct automaton for LSC `securing_barrier` with special treatment of environment elements included (strict interpretation)

■

Example 8.2

Figure 8.7 on the facing page shows another example illustrating the adjusted unwinding algorithm. The corresponding symbolic automaton, using

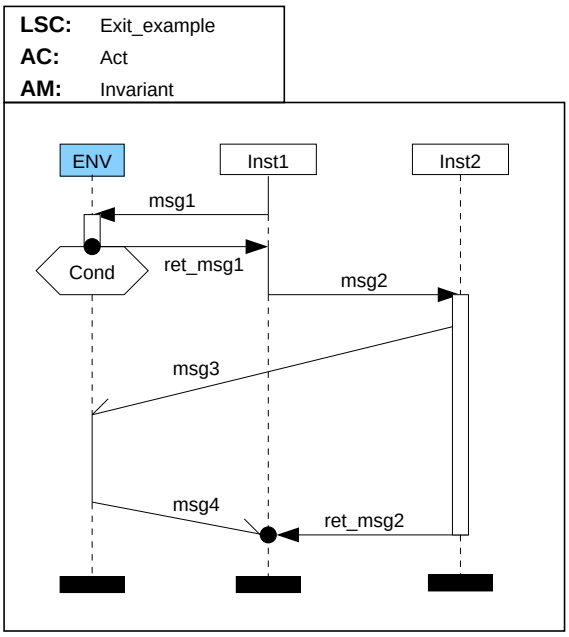


Figure 8.7: Example LSC for the treatment of environment violations

weak interpretation, is shown in figure 8.8 on the next page. q_1 is a fair state, because the temperature of the second location on the environment instance (receipt of `msg1`) is treated as cold, thus resulting in a cold cut. The temperature of the penultimate environment location (receipt of `msg3`) is also treated as cold, but does not result in a cold cut and a fair state q_4, because the location of `Inst2`, which is part of the corresponding cut, is hot. The mandatory condition `Cond` on the environment instance is treated like a possible one resulting in a corresponding transition from q_1 to the exit state. The asynchronous message `msg3` is treated like a cold one, since it is not guaranteed that the environment will receive it. This means that a skipping transition is inserted from q_4 to q_6. q_6 also shows an example for the treatment of simultaneous regions containing messages sent by the environment: If `msg4` does not arrive at `Inst1` at the correct time, i.e. when method call `msg2` returns, the automaton moves into the exit state. ■

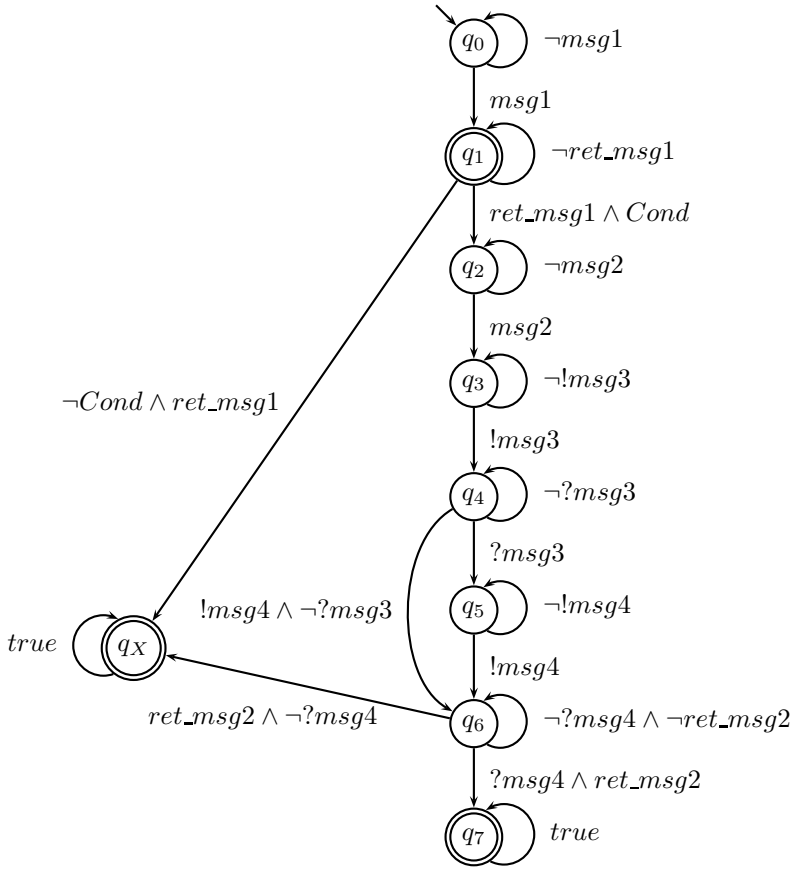


Figure 8.8: Automaton for the LSC of figure 8.7

8.2 External Assumptions

The previous section has illustrated that the treatment of most LSC elements, which occur on the environment instance, can be integrated into the unwinding algorithm and that there is one case, where this is not possible: liveness (hot location temperatures). In order to require the environment to send a message, an external assumption has to be employed, which actually restricts the behavior of the environment. Since the assumptions about the environment behavior are already present in the LSC in form of the LSC elements specified on the environment axis, this information can also be exploited for the automatic derivation of external assumptions, i.e. assumptions which actually enforce the behavior specified on the environment instance.

Section 8.2.1 describes how this is realized by extracting an assumption LSC from the originally specified commitment LSC, whereas section 8.2.2 introduces external assumption LSCs, which are explicitly specified by the user.

8.2.1 Extracting Assumptions from LSCs

The assumptions extracted from a commitment LSC are represented by an LSC as well, although of a slightly restricted type. Such *assumption LSCs* contain only two instances, one for the environment and one for the system under development. This is sufficient, since assumptions are concerned with the items observable at the interface between the system and its environment. The environment can only observe and influence the system behavior via this interface, all internal actions are hidden from it. The extraction thus involves the hiding of internal communication and internal elements, i.e. only those elements of the commitment LSC are extracted into the assumption LSC, which either have originally been specified on the environment instance or are messages leading to or coming from the environment. The environment instance axis of the commitment LSC is consequently retained in the assumption LSC and all other elements are projected onto the external interface of the system represented by the system instance axis.

For the message atoms, which are projected onto the system axis, the question of their ordering arises. In the case of instantaneous messages the position of the complementary message atom on the system instance is easily determined, since the positions of both the sending and receiving atom and cluster are identical. In the case of asynchronous messages the position of the complementary message atoms is more difficult to determine, because the positions of one can not be derived from the position of the other. The observance of the complementary atoms, however, is not necessary in the assumption LSC, since only the message atoms on the environment instance carry relevant information. The fact that messages received have been sent earlier by the system or that messages sent to the system are eventually received is non-essential in this view. Messages received by the environment provide ordering information for the sending of messages and validity of conditions and local invariants. But the concrete ordering of the atoms/clusters on the system instance is irrelevant in the assumption, so that the position of the complementary message atoms is in this case copied from the corresponding atom on the environment axis and all clusters of the system axis are placed into a coregion, if the environment instance contains at least one asyn-

chronous message atom. Care has to be taken when an asynchronous message atom is part of a simultaneous region on the environment axis. In this case the complementary asynchronous message atoms must not be placed in a simultaneous region on the system axis, since no simultaneity between sending and receipt is enforced for asynchronous messages. The complementary message atom is then placed either in a not already occupied cluster above the current one for complementary message sends or below for receipts. Instantaneous messages are not affected, because their point of observation is still uniquely determined by the position of the corresponding atom on the environment axis.

The role of the environment axis in an assumption LSC differs from the one in a commitment LSC. In the former the behavior specified on the environment axis has to be guaranteed, whereas it is only expected in the latter. The environment axis in an external assumption is therefore not treated differently than a normal instance axis. The elements used on the system axis of an extracted assumption are needed by the environment to determine when to send which messages, etc. Therefore only messages, simultaneous regions and coregions will appear on the system axis of an extracted assumption. Hence all location temperatures on the newly introduced system instance are cold, because liveness requirements on the system are expressed in the commitment LSC.
Assumption extraction is performed for each environment instance axis of the commitment LSC, i.e. for each environment instance an assumption LSC is derived.

Assumption Extraction Algorithm

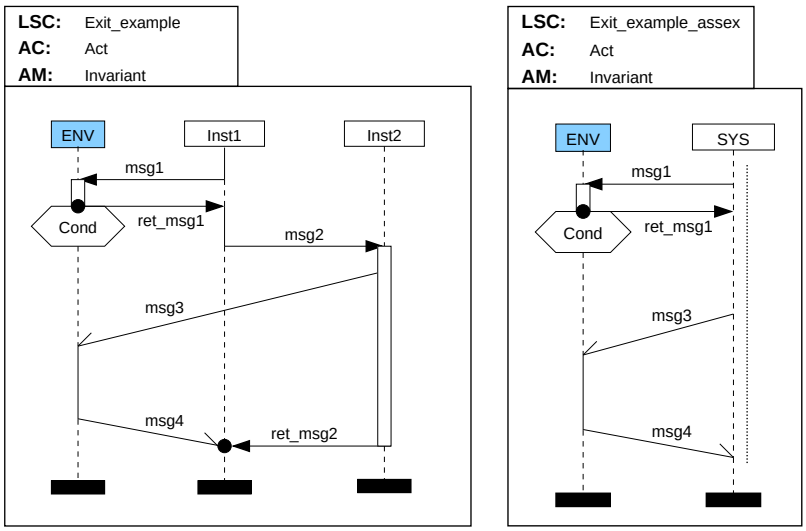
The extracted assumption LSCs obviously share the activation information of the commitment, i.e. take over the activation mode and activation condition. The quantification is always universal, because assumptions express universal constraints on the behavior of the environment.

Recall from definition 6.1 on page 94 that an LSC is a tuple $L = (l,\ assumptions,\ ac,\ pch,\ amode,\ quant)$. The (possibly empty) set $assumptions$ contains the assumption LSCs linked to commitment LSC l. The extraction algorithm thus constructs from a commitment LSC $L_{comm} = (l_{comm}, assumptions, ac, pch, amode, quant)$ a new LSC $L_{ass} = (l_{ass}, \emptyset, ac, pch, amode, universal)$ for each environment instance of L_{comm} and adds it to the set of assumptions of L_{comm}: $L_{comm} = (l_{comm}, assumptions \cup \{L_{ass}\}, ac, pch, amode, quant)$.

Algorithm 8.12 Assumption Extraction

1: **for all** $i_{env} \in Inst(l_{comm}) : envInst(i_{env}) = true$ **do**
2: $Inst(l_{ass}) := \{i_{env}, i_{sys}\}$
3: $Atoms(l_{ass}) := Atoms(i_{env}) \cup \{\bot_{i_{sys}}, \top_{i_{sys}}\}$
4: $async\_msgs := false$
5:
6: **for all** $cl \in$ SortByPos($Clusters(i_{env})$) **do**
7: **for all** $a \in cl : a \in Msgsnd(i_{env})$ **do**
8: **let** $m \in Msgrcv(l_{comm}) : msgID(m) = msgID(a)$
9: $Msgrcv(i_{sys}) := Msgrcv(i_{sys}) \cup \{m\}$
10: **if** $sync\_type(msgID(m)) = async$ **then**
11: $async\_msgs := true$
12: **if** $|cl| > 1$ **then**
13: $position(m) :=$ InsertBelow($position(a)$)
14: **else**
15: $position(m) := position(a)$
16: **end if**
17: **end if**
18: **end for**
19:
20: **for all** $a \in cl : a \in Msgrcv(i_{env})$ **do**
21: **let** $m \in Msgsnd(l_{comm}) : msgID(m) = msgID(a)$
22: $Msgsnd(i_{sys}) := Msgsnd(i_{sys}) \cup \{m\}$
23: **if** $sync\_type(msgID(m)) = async$ **then**
24: $async\_msgs := true$
25: **if** $|cl| > 1$ **then**
26: $position(m) :=$ InsertAbove($position(a)$)
27: **else**
28: $position(m) := position(a)$
29: **end if**
30: **end if**
31: **end for**
32: **end for**
33: **end for**
34:
35: **if** $async\_msgs$ **then**
36: $Coregions(i_{sys}) := Clusters(i_{sys}) \setminus \{\{\bot_{i_{sys}}\}, \{\top_{i_{sys}}\}\}$
37: **else**
38: $Coregions(i_{sys}) :=$ CopyCoregs($Coregions(i_{env})$)
39: **end if**
40: **for all** $loc \in Locations(i_{sys})$ **do**
41: $temp(loc) := cold$
42: **end for**

Algorithm 8.12 on the page before gives the details how the assumption LSC body l_{ass} is derived from the commitment LSC body l_{comm}. A separate assumption is extracted for each environment instance of the commitment LSC (line 1). In line 2 the set of instances for the assumption LSC is constructed by taking over the chosen environment instance from the commitment LSC and adding a new instance i_{sys} for the system under development. In line 3 the atoms on the environment instance of the commitment are copied into the new LSC and instance head and end atoms are added for the system instance. The indicator for the presence of asynchronous messages is initialized to false in line 4. Then each cluster of the environment instance is examined (line 6) and for each message send atom (message receive atom) a corresponding receive atom (send atom) is added to the system instance. In order to be able to correctly determine the positions of asynchronous message atoms, whose complementary atoms on the environment axis are part of a simultaneous region, the clusters are examined from top to bottom. Therefore the clusters are first sorted according to their position by auxiliary function SortByPos().



(a) commitment LSC

(b) extracted assumption

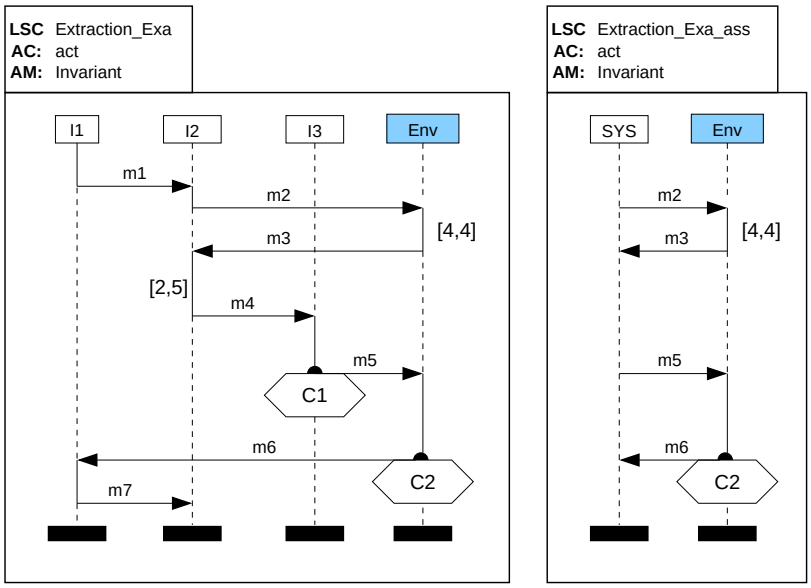
Figure 8.9: Assumption extraction example

For each message send atom the corresponding complementary receive atom (m in line 8) is copied from the commitment LSC and added to the message receive atoms of the system instance (line 9). If the synchronization type of the message is asynchronous, more actions ensue. First, the occurrence of an asynchronous message is recorded in line 11, then the position of the complementary atom is determined. If the asynchronous message atom on the environment instance is part of a simultaneous region, indicated by the current cluster containing more than one atom (line 12), the position is set to an unoccupied value, which is lower than the position of the sending atom on the environment (using auxiliary function InsertBelow (), line 13). Otherwise the position is copied from the message atom on the environment instance (line 15). Message receive atoms on the environment instance are treated analogously in lines 20 - 28.

If the extracted assumption LSC contains asynchronous messages, the clusters of the system axis, save the ones containing the instance head and end atom, are placed into a coregion (line 36). If no asynchronous messages are present, the coregions on the environment axis are transferred to the system axis via auxiliary function CopyCoregs() in order to correctly capture the coregion semantics for instantaneous messages (line 38). In lines 40 and 41 the temperatures of location on the system axis are set to cold.

Example 8.3 (Assumption Extraction)
Figure 8.9 on the facing page gives an example for the assumption extraction algorithm. Figure 8.9(a) shows the LSC from figure 8.7 on page 193, which we use as the commitment LSC in this example, figure 8.9(b) shows the result of the assumption extraction. The two internal messages `msg2` and `ret_msg2` have disappeared, only those messages, which involve the environment, are retained in the extracted assumption. The asynchronous messages `msg3` and `msg4` have been projected onto the system axis and the simultaneous region containing `msg3` and `ret_msg2` disappeared, since the latter message is not observable at the interface. The entire system instance axis is covered by a coregion due to the presence of the two asynchronous messages in the assumption LSC.

Figure 8.10 on the following page shows a second example, which more noticeably illustrates the hiding of internal LSC elements by the extraction algorithm. Only the timing interval and the condition `C2` on the environment instance remain, the internal timing constraint and the internal condition `C1` vanish as well as all internal messages (`m1, m4, m7`). Note in both examples



(a) commitment LSC

(b) extracted assumption

Figure 8.10: Assumption extraction example

that the system instance is entirely made up of cold locations. ■

8.2.2 User Specified Assumptions

In addition to assumptions, which are either internal to the commitment LSC or extracted from it, user specified external assumptions are needed as well. Since internal and extracted assumptions discussed in the previous sections share the activation information of their respective commitment LSC, they are bound to its activation scope, i.e. they are only effective, if the commitment LSC is activated. There are situations, where this is not sufficient: An invariant LSC may e.g. depend on the correct initialization of the model, or the behavior of the environment is to be constrained before the commitment LSC is activated. The latter requirement can also be realized by using an appropriate pre-chart, but using separate assumptions can be easier in some cases. See section 11.3 for concrete examples.

User specified assumptions are identical to extracted assumptions, except that they are not derived automatically from the commitment LSC. They hence contain exactly two instance axes, environment and system, since the environment can only observe communication at the interface to the system. Similarly, only messages, simultaneous regions and coregions should be used on the system axis and all location temperatures should be cold.

8.3 Semantics of External Assumptions

The semantics of internal assumptions is already taken care of by the modified unwinding algorithm presented in section 8.1. The semantics of external assumptions is expressed as a restriction of the system under design, since they disallow certain runs of the system. Technically these restrictions need not only concern the behavior of the environment, i.e. constrain input signals, but may in general restrict any variable of the system under design. Great care has to be exerted when constraining system variables in assumptions, because prohibiting — via assumptions — certain runs of the system, which are actually possible in the system, means that model check results become untrustworthy. In the worst case a violation of a property, which has been specified as an LSC, is only possible in the part of the system behavior, which has been disallowed by such an assumption, so that the property seems to hold, but is in reality not guaranteed when considering the complete system. For this reason we only allow assumptions about inputs in the practical application of LSCs.

Since assumptions, both extracted and user specified, are expressed as LSCs, their semantics is defined by unwinding them into a timed symbolic automaton, as for commitment LSCs. Because assumptions express the expected environment behavior, i.e. are formulated from the environment's point of view, there is no treatment for internal assumptions in the unwinding algorithm, i.e. both instances are treated as normal ones.

It is therefore necessary to know, whether an LSC is an assumption or a commitment. This is indicated by the *usagemode*, which can have the values *comm* or *ass*. The usage mode is an additional input for the unwinding algorithm, which comes into play when determining which LSC elements should be treated as belonging to the environment: in a commitment LSC these are the elements defined on the environment instance, in an assumption LSC no special treatment of these elements is necessary. Thus only the

$isEnv(\cdot)$ function for atoms has to be adjusted:

$$isEnv : Atoms(i) \longrightarrow \mathbf{B}$$

$$isEnv(a) := \begin{cases} true & \text{if } usagemode = comm \wedge envInst(i) \\ false & \text{else} \end{cases}, a \in Atoms(i)$$

Thus the LSC semantics including assumptions are given in the following by extending definition 6.16 on page 145:

8.2 Definition (Satisfaction of an LSC with Assumptions)
Let $L_{comm} = (l_{comm}, \{L_{ass_1}, \ldots, L_{ass_n}\}, ac, \epsilon, amode, quant)$ be a commitment LSC with assumption LSCs $L_{ass_1}, \ldots, L_{ass_n}$, let $\mathcal{TSA}_{l_{comm}}, \mathcal{TSA}_{l_{ass_1}}, \ldots, \mathcal{TSA}_{l_{ass_n}}$ be the timed symbolic automata generated for the LSC bodies $l_{comm}, l_{ass_1}, \ldots, l_{ass_n}$, and let S be the corresponding reference system.

L_{comm} is *existentially satisfied by* S, denoted $S \models_\exists L_{comm}$, ***iff***
$quant = existential \wedge \exists tr \in (Runs(S) \downarrow (L_{ass_1}, \ldots, L_{ass_n})) : tr \models_\exists L_{comm}$

L_{comm} is *universally satisfied by* S, denoted $S \models_\forall L_{comm}$, ***iff***
$quant = universal \wedge \forall tr \in (Runs(S) \downarrow (L_{ass_1}, \ldots, L_{ass_n})) : tr \models_\forall L_{comm}$

with
$Runs(S) \downarrow (L_{ass_1}, \ldots, L_{ass_n}) := \{tr' \in Runs(S) \,|\, tr' \models_\forall L_{ass_1} \wedge \ldots \wedge tr' \models_\forall L_{ass_n}\}$ and $tr \models_\exists L$ and $tr \models_\forall L$ as given in definition 6.16 on page 145. ■

The major difference between definition 8.2 and definition 6.16 lies in the set of (timed) system runs, which are allowed to be checked against the commitment LSC. Here the number of possible system runs is restricted to the ones, which fulfill the assumptions, expressed by $Runs(S) \downarrow (L_{ass_1}, \ldots, L_{ass_n})$. This corresponds to the standard treatment of assumptions, where the system runs are intersected with the runs allowed by the assumptions [Jos93]. Definition 6.16 corresponds to the special case, where the assumptions list of the commitment LSC is empty: an empty assumption does not constrain any system run.

8.4 Related Work

To our knowledge there are no other approaches, which deal with integrated assumptions in sequence charts. Krüger [Krü00] considers external assumptions, which are not represented graphically, however. Assumptions appear in the context of the decomposition of a global specification, given in terms of an extended MSC (MSC-96), into local specifications of the participating instances. For each local specification the other instances of the MSC are part of its environment and therefore the desired behavior of this instance is only guaranteed, if the other instances conform to their role in the global specification. No concrete application, like e.g. model checking, is presented, however.

The basic approach is similar to our extraction procedure: One instance is singled out and the message exchanges of the entire MSC are projected onto the interface between the chosen instance and the remaining instances. But whereas in our case the commitment part is the entire LSC and the assumption part becomes another LSC, the result of [Krü00] is not an MSC, but rather sets of textual properties for both the commitment and the assumption part. The properties are further divided into a safety and a liveness part, while in our approach there is only one assumption containing all individual properties. In contrast to our work [Krü00] does not allow the possibility to specify assumptions about the environment of an entire chart.

Chapter 9

Upgrading Activation: Pre-charts

So far the activation of an LSC is guarded by activation condition and mode. It turns out that these constructs do not always sufficiently characterize the activation point of a property specification. Often it is necessary to know more about the history of a run, before being able to decide, whether the LSC should be activated. There may be e.g. more than one way for a run to arrive at a certain system state (characterized by a condition), but the LSC should only be activated, if the run has followed a specific "route". We are for instance only interested in activating an LSC, when no errors have occurred so far. This motivates the introduction of pre-charts which allow to specify a prefix of a run acting as a trigger for the actual LSC.

The formal semantics of a pre-chart is defined by an altered unwinding algorithm yielding a second automaton, in addition to the one for the LSC itself. The pre-chart automaton acts as a filter on the system runs letting only those pass, which fulfill the pre-chart.

Section 9.1 introduces the pre-chart concept in detail, whereas the formal semantics is presented in section 9.2. As usual we close this chapter with a discussion of related work in section 9.3.

9.1 Pre-charts

This chapter deals once more with the issue of when an LSC is activated. So far two means for specifying the activation point have been presented in

section 4.2 on page 75: activation condition and activation mode. The former specifies a single point in time, at which the LSC is activated. Depending on the activation mode the LSC may be activated a number of times, if this point is reached more than once. Such a single point in time is not always sufficient for triggering an LSC, since there may be several distinct possibilities to reach a system state, where the activation condition holds. Consider for instance the LSC in figure 9.1, which repeats the LSC for successfully closing the barrier of the train control system of figure 8.1 on page 173. There are two basic ways to arrive at the state, where the signal `lights_on` is sent: the barrier was operational in the previous securing cycle or there was an error, either during closing or opening. In the latter case the securing procedure for the next train will not be able to successfully secure the crossing, i.e. the protocol shown in the LSC in figure 9.1 does not hold. In order to successfully verify that the barrier is closed, the prerequisite is thus that the barrier has been successfully opened during the previous securing procedure. This requires that a *sequence* of messages must be observed, i.e. several points in time, instead of only one.

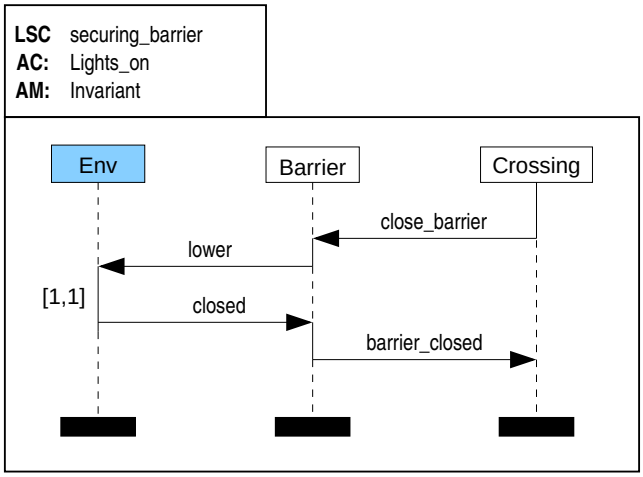


Figure 9.1: LSC for the Crossing component specifying the barrier closing

Pre-charts take care of this requirement by allowing to specify a prefix or history, which must be fulfilled by a run in order to activate the LSC. A pre-chart is essentially an LSC, i.e. all language constructs can be used in a pre-chart, but its semantics is different, since the message sequence of the pre-chart is not *required* to hold in the system, but rather must be observed before activating the actual LSC.

Pre-charts do not replace the activation condition, but extend it; the activation condition in the presence of a pre-chart indicates the starting point of the prefix. The pre-chart may be empty as has been assumed in the previous sections. The informal semantics of an LSC with pre-chart is consequently: If the activation condition holds *and* afterwards the pre-chart is *completed*, then the LSC is activated. The key point here is that the pre-chart has to be traversed completely in order to activate the LSC. This explicitly disallows activation of the actual LSC, if the pre-chart is exited, e.g. due to a violated possible condition. Thus, only when the entire prefix described in the pre-chart has been observed is the LSC considered. A violation of the pre-chart consequently is not a violation of the actual LSC, but rather means that the LSC is not activated.

Graphically pre-charts are represented by a large possible condition symbol covering the pre-chart elements. The pre-chart is placed above the actual LSC. Figure 9.2 shows LSC `securing_barrier` (cf. figure 9.1), extended by an appropriate pre-chart. The new activation condition is the opening command for the barrier actuator (`raise_barrier`). Since the goal is to require a successful opening of the barrier in order to check the LSC describing the closing procedure for the following train, the environment must send the messages `opening` and `open` as shown at the beginning of the pre-chart. The time bounds between the instance head and `opening`, resp. `opening` and `open` ensure that both barrier and crossing controller return to their initial states: The barrier controller in the STATEMATE model (cf. section 2.2.4) must receive the `open` message at most two steps after the event to raise the barrier (`RAISE`) has been issued in order to return to its initial state (`OPENED`). Because the barrier has to first leave its closed position (event `BARRIER_OPENING`, to which the `opening` message is mapped) before it can reach its upper end position, the tight timing as given on the environment axis of the LSC in figure 9.2 is necessary. Once the barrier and crossing controller have returned to their respective initial states, the message originally activating the LSC in figure 9.1, `lights_on`, should be sent triggering the activation of the actual LSC. Another requirement for the prefix is that the

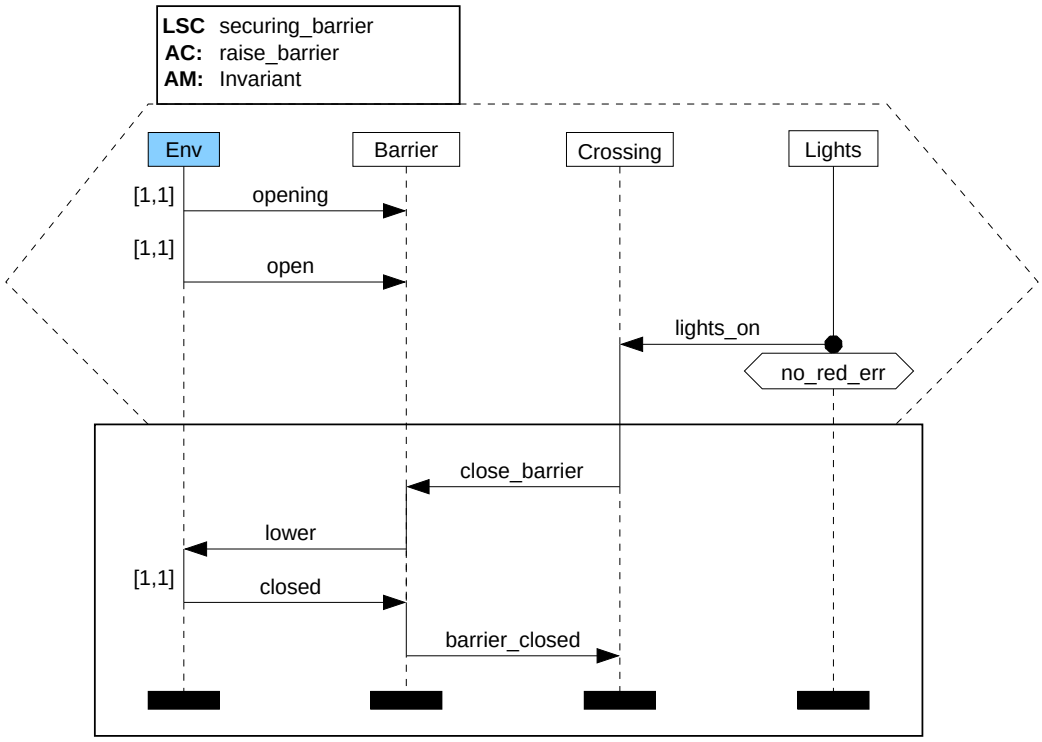


Figure 9.2: LSC `securing_barrier` with pre-chart

traffic lights are fully operational, because otherwise the crossing controller will not send the closing command to the barrier controller. This is in part ensured by the fact that `lights_on` occurs, which can only happen, if the lights are functioning up to this point. The condition `no_red_err` is needed to ensure that the traffic lights remain operational also when the `lights_on` message is sent. Thus the message, which activated the LSC without pre-chart (figure 9.1), still plays this role as it has moved to the end of the pre-chart.

The early versions of the original LSC paper, [DH98, DH99], did not consider pre-charts, in the latest version of the paper [DH01] the authors follow our suggestion and include pre-charts, although the details are not spelled out.

9.2 Formal Semantics

The semantics of an LSC with pre-chart hinges on the connection of pre-chart and LSC. The LSC must be activated *immediately* after the pre-chart has been fulfilled, analogously to the case where only an activation condition is present. Once the desired prefix has been observed the control lies in the hands of the actual LSC, which determines which communications are allowed and if progress is enforced. Since the semantics of both individual charts are given as automata, the issue to be solved is the connection between the two automata.

One way to effect the connection of pre-chart and LSC is by concatenation of the respective automata, which is proposed by [DH01] in order to describe the semantics of an LSC with pre-chart. The pre-chart automaton needs to be modified in order to correctly capture the semantics: Since the purpose of a pre-chart is to observe an activating prefix, violations of the pre-chart must not result in a violation of the LSC and exits from the pre-chart must not activate the actual LSC. Both situations do not reflect a complete traversal of the pre-chart and should thus have no influence on the actual LSC. The generation of the pre-chart automaton as described in [DH01] consequently merges the error state with the exit state in order to cater for the first requirement stated above, and merges the exit states of pre-chart and LSC during the concatenation process in order to ensure the second requirement.

The semantics of an LSC including its pre-chart are thus informally defined as (disregarding the activation mode for the moment): If a run satisfies the activation condition, then in the remainder it must be accepted by the concatenated automaton.

The automaton construction described informally in [DH01] is correct only for the universal quantification; existential LSCs require a different treatment, which is omitted in [DH01]. Consider for instance the automaton in figure 9.3, which shows how the concatenation proposed in [DH01] looks like for LSC `securing_barrier` of figure 9.1 on page 206. The states of the pre-chart part of the automaton are all accepting, since progress violations in the pre-chart should not result in a violation of the LSC. For the same reason is condition `no_red_err` treated as possible instead of mandatory. This automaton allows violations only once the pre-chart has been fulfilled, all possibilities for a violation of the pre-chart have been removed, similar to the treatment of internal assumptions (cf. section 8.1 on page 173). The depicted automaton is correct only in the universal quantification. For the

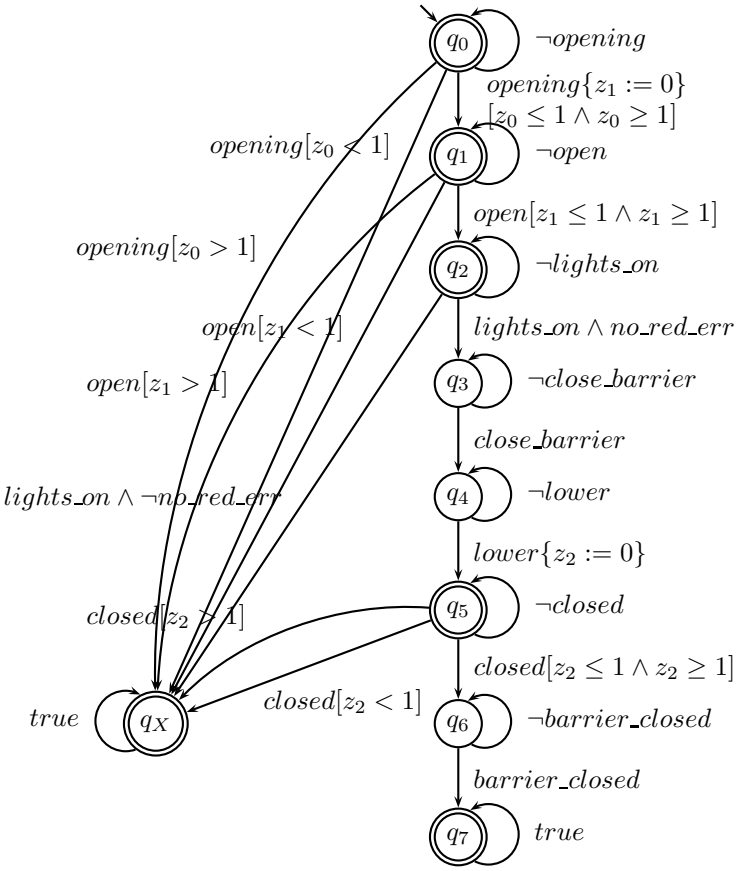


Figure 9.3: Concatenated automaton for LSC `securing_barrier` for universal quantification

existential quantification there must be at least one run, which satisfies the LSC. This requires that the LSC is activated at least once, a case not covered by the automaton in figure 9.3. The concatenated automaton accepts not only those runs satisfying the LSC, but also those, which violate the pre-chart or cause exits from it. A run for instance, which shows all the messages of the pre-chart at their allotted places and times, but violates condition `no_red_err` when `lights_on` is observed, is accepted by this automaton, even though the LSC is not activated. It would thus be possible that an LSC is existentially satisfied, even though it is never activated.

In order to correctly reflect the semantics a different automaton would have to be generated for the pre-chart, which — instead of being as fair,

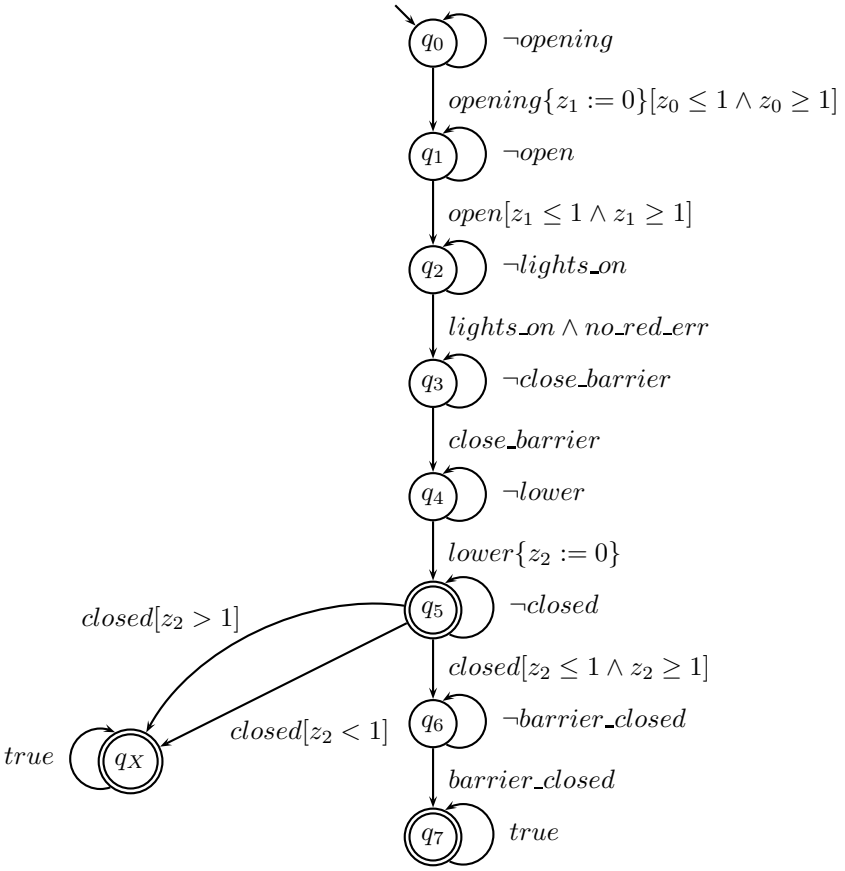


Figure 9.4: Concatenated automaton for LSC `securing_barrier` for existential quantification

i.e. accepting, as possible as the one shown in figure 9.3 — needs to be as unfair as possible in order to reach the actual LSC part of the concatenated automaton, which is shown in figure 9.4. Removing the states belonging to the pre-chart from the set of fair states in combination with omitting all transitions to the exit state in the pre-chart ensures that the pre-chart must be traversed. All runs which do not fulfill the pre-chart result in an error, which is inconsequential in the existential quantification as long as there is one run, which activates and satisfies the LSC.

The drawback of this approach is thus that a different automaton construction is required depending on the quantification. In order to avoid this unnecessary dependency, we use a different approach, which does not rely

on the quantification. Our approach builds on the fact that a pre-chart describes only a finite sequence of messages: from the time when the activation condition holds until the pre-chart is traversed completely. This means that a finite (timed) run and consequently a finite (timed) automaton are sufficient to recognize an activating prefix. The only accepting state of this finite automaton is the final state representing the complete traversal of the pre-chart.

In definition 9.1 on page 215 below we will see that pre-charts are treated uniformly in the semantics, independent of the quantification, thus effecting a simpler treatment of pre-charts in the semantics. The pre-chart automaton is separated from the LSC automaton and treated in the same manner as the activation condition. For universally quantified LSCs this results in the following informally stated implication: If the activation condition holds *and* the pre-chart is fulfilled, then the LSC is considered. Our approach thus yields a clearer separation of concerns between activation and satisfaction of an LSC. The concatenation approach of [DH01] mixes both aspects and results in a more complex semantics.

Since pre-charts only *observe*, if the desired prefix is part of a run, there is no need for internal assumptions, i.e. to treat environment axes in a special way for the unwinding. The effect of internal assumptions is guaranteed by the fact that the actual LSC is only activated once the complete pre-chart has been traversed. Thus, no exit state is needed.

The treatment of extracted assumptions for the LSC does not need to be adjusted in the presence of a pre-chart. The required synchronization with the activation of the actual LSC is already incorporated into the assumption extraction described in section 8.2.1 on page 195. Recall that for the assumption extraction an assumption LSC $L_{ass} = (l_{ass}, \emptyset, ac, pch, amode, universal)$ is generated for a commitment LSC $L_{comm} = (l_{comm}, assumptions, ac, pch, amode, quant)$. In particular, the extracted assumption inherits the pre-chart of its commitment LSC. The correct activation of the extracted assumption is thus guaranteed.

9.2.1 Pre-chart Unwinding Algorithm

Since the environment axis does not need to be treated specially for pre-charts, the pre-chart unwinding algorithm is similar to algorithm 7.1 on page 162 for timed LSCs. Algorithm 9.1 generates a timed finite automaton $\mathcal{TFA}_{pch} = (\Sigma, Q, q_0, C, \longrightarrow, F)$ from a pre-chart *pch* (cf. definition 5.9 on page 90).

Algorithm 9.1 on the following page shows the main part of the pre-chart unwinding algorithm. Note that the cut temperature is disregarded for the determination of the set of accepting states and no exit state is added. The set of accepting states is consequently initialized to the empty set (line 9). The treatment of local invariants does not distinguish possible and mandatory ones, since the mode is irrelevant in a pre-chart. Thus the set of active possible local invariants is initialized to the empty set (line 10) and all local invariant starts are treated like mandatory ones (line 11). Line 15 contains the definition of the set of accepting states, which contains only the final state indicated by the empty ready set. Since this is a finite automaton, there is no need for a self loop on the final state. Reaching this state is sufficient to determine the acceptance of a timed run.

Since no exit state exists, there is no need to generate exit transitions, so function GENERATE_EXITS() is eliminated completely. The remaining functions of the pre-chart unwinding algorithm do not touch the acceptance criterion and are therefore identical to the ones found in section 7.2.2.

Note that pre-chart and LSC have separate scopes due to being of a different nature. The pre-chart consequently possesses a separate set of identifiers, i.e. in the strict interpretation only the messages occurring in the pre-chart are thus restricted in the corresponding automaton, but not the ones occurring in the actual LSC. If a restriction of messages occurring in the LSC is desired in the pre-chart, this can be done selectively by the user by adding appropriate local invariants.

9.2.2 Pre-charts Semantics

In this section we extend the definition of satisfaction of an LSC as given by definition 8.2 on page 202 in section 8.3 to LSCs with pre-charts. Note that a pre-chart is not concerned by the quantification or activation mode, since this information pertains to the complete LSC: activation condition, pre-chart and LSC, as is seen in the definition below. The interpretation

Algorithm 9.1 Pre-chart Unwinding Algorithm

if *interpretation* = strict **then**
  $sc :=$ NEG_CONJUNCT($MsgLabels(pch)$)
end if
$\Sigma := MsgLabels(pch) \cup Conditions(pch) \cup Local\_Invariants(pch) \cup \{true, false\}$
$C := \{z_0, z_1, \ldots, z_k\}$
$phases := \{Phase_0\}$
$Q :=$ {STATE($Phase_0$)}
$q_0 :=$ {STATE($Phase_0$)}
$F := \emptyset$
$poss\_invs_0 := \emptyset$
$mand\_invs_0 := \{li \in Local\_Invariants(l) \mid \exists scl\, \exists cl \in scl\, \exists a, a' \in cl :$
  $a \in Instheads(l) \wedge a' \in LI\_starts(l) \wedge liID(a') = li$
while $phases \neq \emptyset$ **do**
  let $ph = (Ready_i, History_i, Cut_i) \in phases$
  if $Ready_i = \emptyset$ **then**
    $F :=$ {STATE(ph)}
  else
    for all $Fired_{i_k} \in \mathcal{P}(Ready_i)$ **do**
      $successor := Step(ph, Fired_{i_k})$
      let $successor = (Ready_j, History_j, Cut_j)$
      if $successor = ph$ **then**
        INSERT_SELF_LOOP($ph, Ready_i$)
      else
        $TransitionAnnot :=$CONSTRUCT_TRANSITION($ph, successor,$
          $Ready_i, Fired_{i_k}$)
        $ClkResets :=$ COMPUTE_RESETS($Fired_{i_k}$)
        $ClkConstr :=$ COMPUTE_CONSTRAINTS($Fired_{i_k}$)
        if $\exists q \in Q$ *with* STATE$^{-1}(q) = (Ready_x, History_x, Cut_x)$: $Ready_i = Ready_x \wedge History_i = History_x$ **then**
          $successor :=$ STATE$^{-1}(q)$
        else
          $Q := Q \cup$ {STATE($successor$)}
          $phases := (phases \setminus \{ph\}) \cup \{successor\}$
        end if
        $\longrightarrow := \longrightarrow \cup \{($STATE($ph$)$, TransitionAnnot, ClkResets,$
          $ClkConstr,$ STATE($successor$))$\}$
      end if
      INSERT_SKIP_TRANSITION($TransitionAnnot, ph, successor,$
        $ClkResets, ClkConstr$)
    end for
  end if
end while

affects both the pre-chart and the LSC as well, although no messages of the pre-chart are constrained in the actual LSC and vice versa.

9.1 Definition (Satisfaction of an LSC with Pre-chart)
Let $L_{comm} = (\ l_{comm},\ \{\ L_{ass_1},\ \ldots,\ L_{ass_n}\ \},\ ac,\ pch,\ amode,\ quant\ )$ be a commitment LSC with assumption LSCs $L_{ass_1}, \ldots, L_{ass_n}$, let $\mathcal{TSA}_{l_{comm}}, \mathcal{TSA}_{l_{ass_1}}, \ldots, \mathcal{TSA}_{l_{ass_n}}$ be the timed symbolic automata generated for the LSC bodies $l_{comm}, l_{ass_1}, \ldots, l_{ass_n}$, let $\mathcal{TFA}_{pch}$ be the timed finite automaton generated for pre-chart pch, and let S be the corresponding reference system.

L_{comm} is *existentially satisfied by* S, denoted $S \models_\exists L_{comm}$, ***iff***

$$quant = existential \wedge \exists tr \in (Runs(S) \downarrow (L_{ass_1}, \ldots, L_{ass_n})) : tr \models_\exists L_{comm},$$

with

$$tr \models_\exists L_{comm} \textbf{\textit{iff}} \begin{cases} \exists j \geq 0 : tr_0 \models ac \wedge & amode = initial \\ \quad tr_1^j \models pch \wedge & \\ \quad \overrightarrow{tr_{j+1}} \in \mathcal{L}(map(\mathcal{TSA}_{l_{comm}})) & \\ \exists i\, \exists j \geq i : tr_i \models ac \wedge & amode = invariant \\ \quad tr_{i+1}^j \models pch \wedge & \\ \quad \overrightarrow{tr_{j+1}} \in \mathcal{L}(map(\mathcal{TSA}_{l_{comm}})) & \\ \exists i\, \exists j \geq i : tr_i \models ac \wedge & \\ \quad \neg active(L_{comm}) \wedge & amode = iterative \\ \quad tr_{i+1}^j \models pch \wedge & \\ \quad \overrightarrow{tr_{j+1}} \in \mathcal{L}(map(\mathcal{TSA}_{l_{comm}})) & \end{cases}$$

L_{comm} is *universally satisfied by* S, denoted $S \models_\forall L_{comm}$, ***iff***

$$quant = universal \wedge \forall tr \in (TRuns(S) \downarrow (L_{ass_1}, \ldots, L_{ass_n})) : tr \models_\forall L_{comm},$$

with

$$tr \models_\forall L_{comm} \textit{iff} \begin{cases} \forall j \geq 0 : tr_0 \models ac \wedge & amode = initial \\ \quad tr_1^j \models pch \Rightarrow & \\ \quad \overrightarrow{tr_{j+1}} \in \mathcal{L}(map(\mathcal{TSA}_{l_{comm}})) & \\ \forall i \, \forall j \geq i : tr_i \models \ ac \wedge & amode = invariant \\ \quad tr_{i+1}^j \models pch \Rightarrow & \\ \quad \overrightarrow{tr_{j+1}} \in \mathcal{L}(map(\mathcal{TSA}_{l_{comm}})) & \\ \forall i \, \forall j \geq i : tr_i \models ac \wedge & \\ \quad \neg active(L_{comm}) \wedge & amode = iterative \\ \quad tr_{i+1}^j \models pch \Rightarrow & \\ \quad \overrightarrow{tr_{j+1}} \in \mathcal{L}(map(\mathcal{TSA}_{l_{comm}})) & \end{cases}$$

with

- $tr_{i+1}^j \models pch$ denoting the satisfaction of pch by tr_{i+1}^j :
  $tr_{i+1}^j \models pch$ **iff** $tr_{i+1}^j \in \mathcal{L}(map(\mathcal{TFA}_{pch}))$,
- $(Runs(S) \downarrow (L_{ass_1}, \ldots, L_{ass_n}))$ as given by definition 8.2 on page 202, $active(L)$ and $\mathcal{L}(map(\mathcal{TSA}_l))$ as given by definition 6.16 on page 145.

■

For existential satisfaction of an LSC all three components must hold, i.e. for at least one run the activation condition must be evaluated to true and the pre-chart must be fulfilled and the LSC must be satisfied. For universal satisfaction of an LSC the pre-chart is placed on the left-hand side of the implication: *If* the activation condition holds *and* the pre-chart is fulfilled, *then* the actual LSC must be satisfied. For existential satisfaction the implication is simply exchanged for a conjunction. The pre-chart is thus treated in an identical manner to the activation condition.

The transition from the pre-chart to the LSC upon complete traversal is achieved using index j, which marks the valuation of timed run tr reaching the final state in the pre-chart automaton and thus the transition into the LSC. The next valuation $j + 1$ is the first step of run tr, which is within

the scope of the actual LSC. Note that $j = i$ in case of an empty (i.e. non-existent) pre-chart, so that the run segment tr^i_{i+1} is empty as well, which means that $tr^i_{i+1} \models pch$ is trivially true. Definition 8.2 hence is a special case of the complete definition given here.

Note that the quantification of j conforms to the overall quantification of the LSC. In the existential case only one such j need exist in order to fulfill the LSC. In the universal case a universal quantification of j allows non-determinism in the pre-chart automaton, which can for instance arise due to isolated conditions (cf. section 6.2.4 on page 118). The use of non-deterministic pre-charts is strongly discouraged, however.

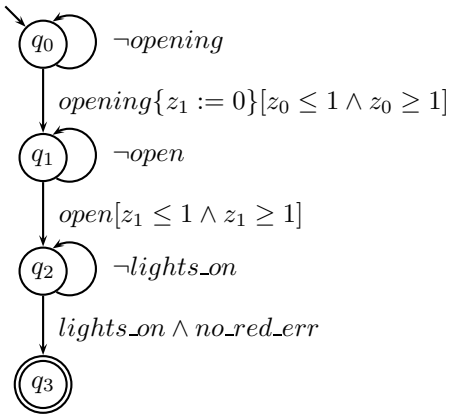


Figure 9.5: Finite automaton for pre-chart of LSC `securing_barrier` using weak interpretation

Example 9.1 (Pre-chart)
Figure 9.5 shows the finite timed automaton for the pre-chart of LSC `securing_barrier` from figure 9.1 on page 206. The only accepting state is the final state q_3. Note that there is no self loop on the final state due to the finite nature of the automaton and that the elements specified on the environment instance are not treated as internal assumptions. The treatment of the other elements and timing constraints in this finite automaton is identical to the one for timed symbolic automata.

Figure 9.6 on the next page shows the timed symbolic automaton for the actual LSC of figure 9.1. Note that the LSC automaton respects the special nature of elements defined on the environment instance, in this case

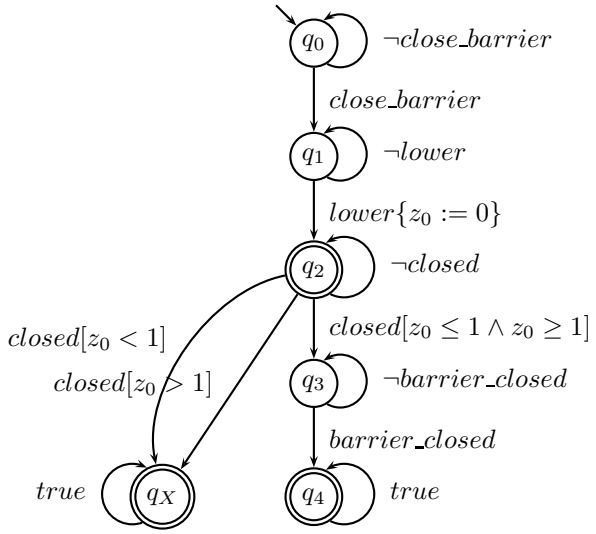


Figure 9.6: Automaton for LSC `securing_barrier` using weak interpretation

the `closed` message and the associated timing interval, whereas the pre-chart automaton includes no special treatment for elements specified on the environment instance.

■

9.3 Related Work

We have already discussed in detail the pre-charts approach proposed in [DH01] and the differences to our own in the motivation of our semantics definition in section 9.2 on page 209 and thus do not repeat the argumentation here.

The pre-charts considered by Bontemps in [Bon01] conform to the treatment presented in [DH01], i.e. pre-chart and LSC automaton are concatenated. This only works in [Bon01], because only messages and hot temperatures are considered, otherwise the abovementioned problems wrt. the concatenation approach would arise.

Krüger [Krü00] (cf. section 6.4 on page 149) proposes a composition operator called *trigger composition*, which shows some traits of a pre-chart. This operator joins two MSCs: $\alpha \mapsto \beta$. Its meaning is: If the behavior specified by MSC α has been observed, then the behavior specified by MSC β

is inevitable. This notion is similar to the pre-chart concept presented in this section, but not identical as the trigger composition operator expresses rather a liveness requirement than a history of behavior, since β need not start immediately after the completion of α. This is a key property of pre-charts, which can not be expressed by the MSC language presented in [Krü00]. The trigger composition rather corresponds to a liveness requirement, i.e. a hot cut, in LSCs.

Firley et al. [FHD+99] (cf. section 6.4 on page 149) present a pre-chart-like concept as well, called *If-Then-Behavior*, which is applied to SDs. It corresponds more closely to pre-charts than the trigger composition, since the second SD is entered immediately after the first one is completed. The connection between the first and second SD is established by concatenation, which does not provoke the problem with the quantification described in section 9.2, since SDs are viewed only existentially in [FHD+99].

Chapter 10

Embedding LSCs into the Development Process

The practical usefulness of a language not only depends on its immediate properties like features, graphical representation or expressiveness, but also on guidelines how and when to apply it. The users need to know for which tasks in the design process the language is intended and how it should be applied to get the best results. Hence, in addition to the description of the language and its meaning (syntax and semantics) a methodology should be provided, which indicates what the primary use cases of this language are. After introducing the language features and defining their semantics in the preceding chapters, this chapter presents such a methodology for LSCs on the basis of a model-based development process. There are several standardized process models, like the waterfall model [Roy70], spiral model [Boe88] or V-model [ESt97], to name but a few.

Regardless of the particular process model there are recurring common phases. Since we do not want to focus on a specific development process, we look at an abstract process model, which consists of these typical, recurring development phases. In section 10.1 we thus introduce such an abstract process model and its phases. We assume that the process is model-based and focus on the phases on the use case of formal verification and the phases affected thereby. Section 10.2 outlines our methodology by embedding LSCs into this abstract development process. We conclude this chapter with a review of related work in section 10.3.

10.1 Abstract Development Process

Phase 1: Initial System Analysis Phase embedded controller development typically starts with a textual requirements specification, which describes the basic functionality using natural language. It also contains non-functional constraints like timing requirements, side conditions due to limited resources, interoperability demands, etc. For the documentation of the major functionality Use Case Diagrams are often used.

Phase 2: Architectural Design Phase The Architectural Design Phase is concerned with structuring the system under design (SUD). Based on the results of the previous phase the SUD is decomposed in a top-down manner, iteratively breaking the complete system down into sub-components. Simultaneously the interfaces, including the interface between the system itself and its environment, and the connections between the different sub-systems are defined. In a model-based development process the functional decomposition of the system can e.g. be represented by classes (if for instance the UML is employed) or activities (if STATEMATE is used). The system structure, including connections between entities, is then for example modeled by Class, Collaboration or Sequence Diagrams or Activity Charts. The result is a model of the hierarchical system architecture with defined interfaces and connections.

Phase 3: Behavioral Design Phase Assuming a model-based process, the detailed behavior of each sub-system is modeled in this phase. This model can be analyzed, simulated and verified, depending on the tool support. For STATEMATE designs the behavioral specification is given as Statecharts, in the UML State Diagrams are typically used. The goal of this phase is a reference model, which has been thoroughly analyzed and implements the key properties specified in the requirements document. It thus serves as a reference for the following phases, which is the most important advantage of a model-based development process.

Phase 4: Implementation Phase This phase entails the actual implementation of the SUD in hard- and software. The tasks to be performed here are thus primarily the partitioning into software and hardware parts and the derivation of code (manually or automatically). The model, which

is the outcome of the previous phase, serves as a reference against which the implementation is measured. The particular steps in this phase strongly depend on the type of application being developed and the tools used; several modeling tools e.g. offer automatic code generation capabilities.

Phase 5: Test Phase The goal of this phase is to ensure that the implementation generated in the previous phase is free of any major errors and meets the initially specified requirements. This entails first testing each sub-system in isolation (*module test*) and later in conjunction with other subsystems (*integration testing*). For embedded controllers prototypes, realized either in hard- or software, are often hooked up to the physical environment they are built for (*hardware* or *software in the loop testing*). The final part of this stage is the *acceptance test*, where the conformance of the complete system to the customer's requirements is demonstrated. Note that in most process models this phase consists of several individual phases.

10.2 Advanced Use Cases for LSCs

In this section we outline the methodology for embedding LSCs into a model-based development process, with special attention to the use case of formal verification. We use the abstract development process introduced in the previous section as a guideline through the methodology. In the Initial System Analysis Phase LSCs are typically not used, since LSCs are not intended for specifying non-functional requirements, except timing information. Moreover, the structure of the SUD is only developed in the following phase, which constitutes the first use case for LSCs:

10.2.1 Capturing Typical System Interactions

At the beginning of Architectural Design Phase existential LSCs are used for the illustration of the major use cases, similar to the use Sequence Diagrams are put to in the UML. During elaboration of the use cases in the form of LSCs the primary participants are identified and become instances in the LSC. In this manner a first view of the system architecture evolves and thus the first level of hierarchy is created.

By allowing to capture the typical communication scenarios as LSCs the designer is supported in identifying the entities needed for the realization

of the major functionality and in determining their interfaces. This process is iterated as the entities identified at the top level are further decomposed yielding a hierarchical structuring of the SUD where at each level the typical interactions are specified by a set of existential LSCs.

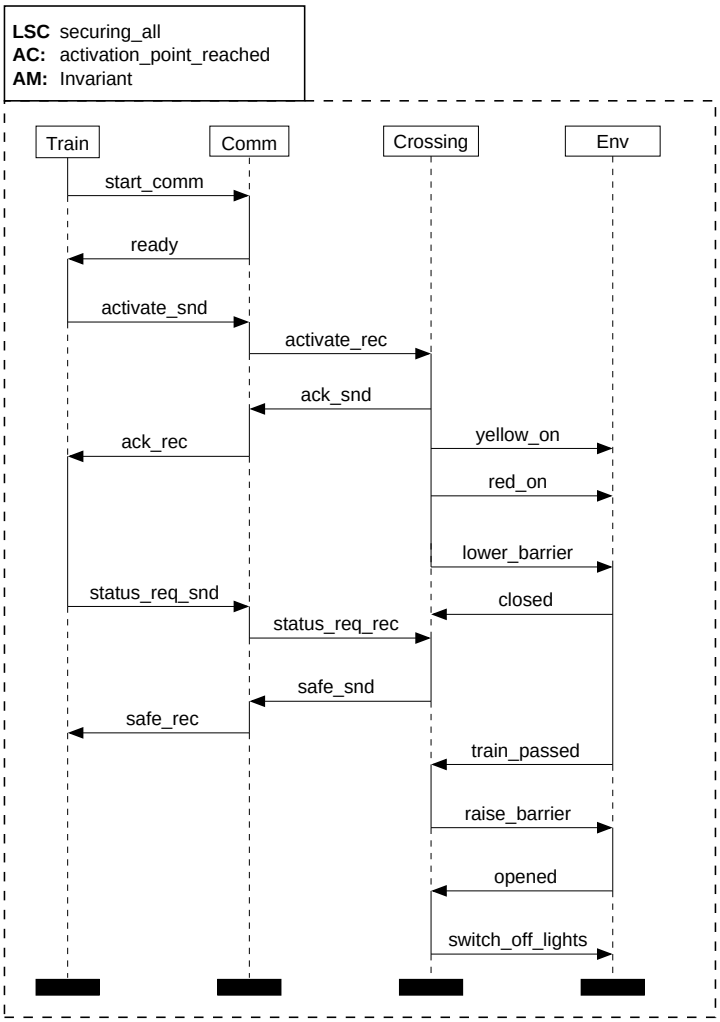


Figure 10.1: Existential LSC showing the desired interaction on the top level of the train control system (Architectural Design Phase)

The scenarios expressed show overall interaction sequences, i.e. the LSCs are extensive and contain a substantial number of messages. A typical approach for instance is to first draw an LSC depicting the good case, the

desired behavior, and then diversify this scenario by considering error cases and alternate behaviors. Figure 10.1 on the preceding page e.g. shows the good case for the top level of the train control system; see section A.1.1 for an extensive set of existential LSCs for the top level.

LSCs in this phase consist mainly of messages, some conditions and only occasionally of local invariants or timing annotations. Mandatory elements and hot temperatures are prevailing, because due to the existential quantification only one run showing the interactions specified need exist. This run should show all of the specified behavior, however, not only parts of it, i.e. exits due to violated possible conditions or local invariants or getting stuck in a cold cut are not desired here. Activation is guarded in the majority of cases by activation conditions only, since the knowledge about the system's details are not needed for the description of typical interactions and may not even be available at this point. For similar reasons coregions are used in cases, where the exact ordering is unimportant or unknown at the moment. Moreover, environment instances are not distinguished from ordinary instances, since assumptions (internal or external) are irrelevant in the existential view.

Complementary to the description of expected interactions existential LSCs can also be employed for the specification of undesired communication sequences, i.e. *negative scenarios*, which should never be observed. This usage becomes especially useful in the subsequent phase when existential verification is available. Then the absence of such undesired behavior can be proven.

Description of typical interactions is also the prime use case for standard sequence charts, although the purpose is mostly documentary. LSCs improve on this informal usage of sequence charts in two ways. On the one hand implicit information is made explicit by increasing the expressiveness, e.g. by allowing to specify the activation point or to group several elements together in simultaneous regions. The methodology presented here on the other hand aims at reusing the existential LSCs in later phases (see below), thereby raising their value above a mere documentary level.

Example 10.1
Figures 10.2 and 10.3 show examples of existential LSCs for the crossing: Depicted are the good case (no errors occur, the crossing is secured; figures 10.2) and the error case, where the train does not reach the crossing

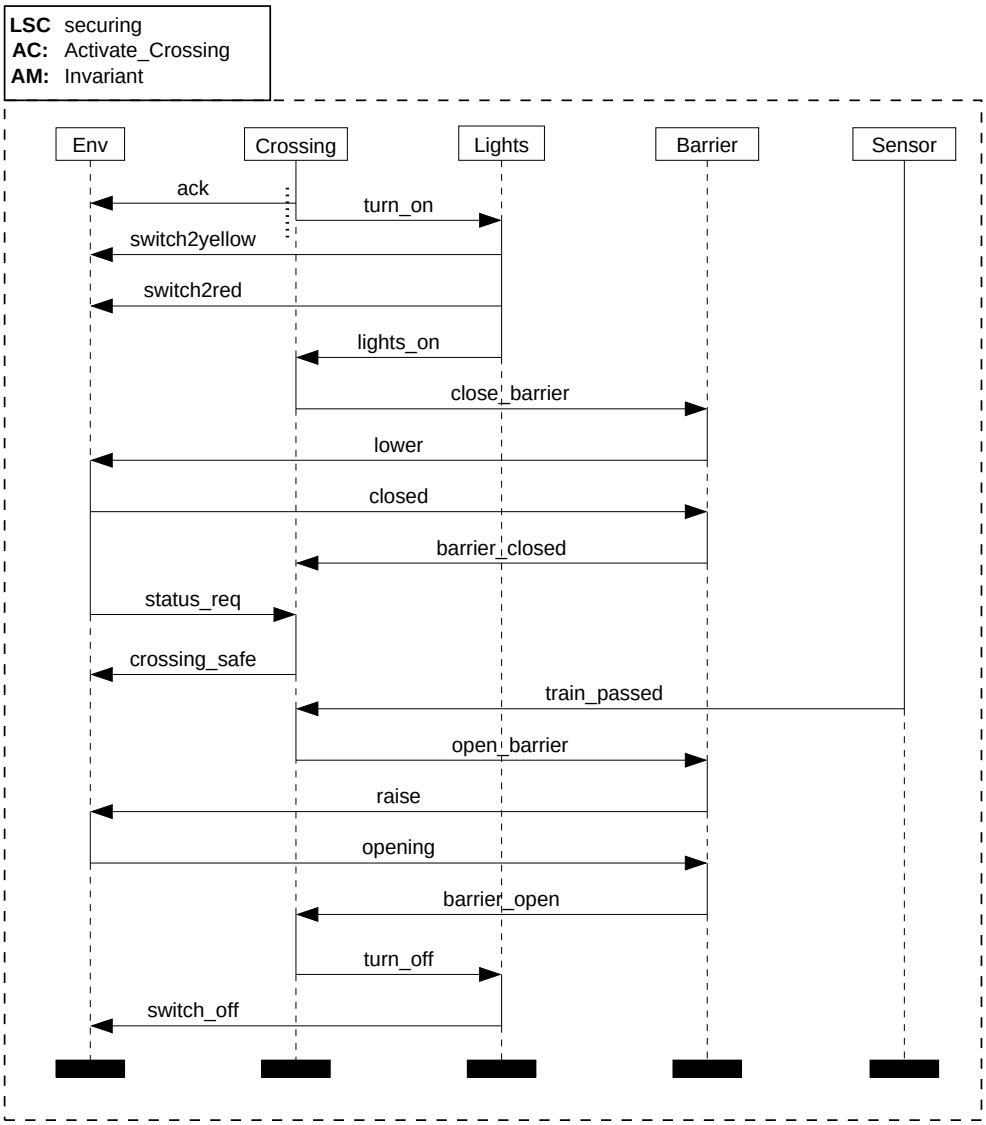


Figure 10.2: Existential LSC showing the desired interaction at the crossing (Architectural Design Phase)

in time and sends the release message (`free`), which results in reversing the securing process by starting to open the barrier (figure 10.3). Note that the environment axis is not distinguished from the other instances. These two examples, in addition to the LSC depicted in figure 10.1 on page 224, show

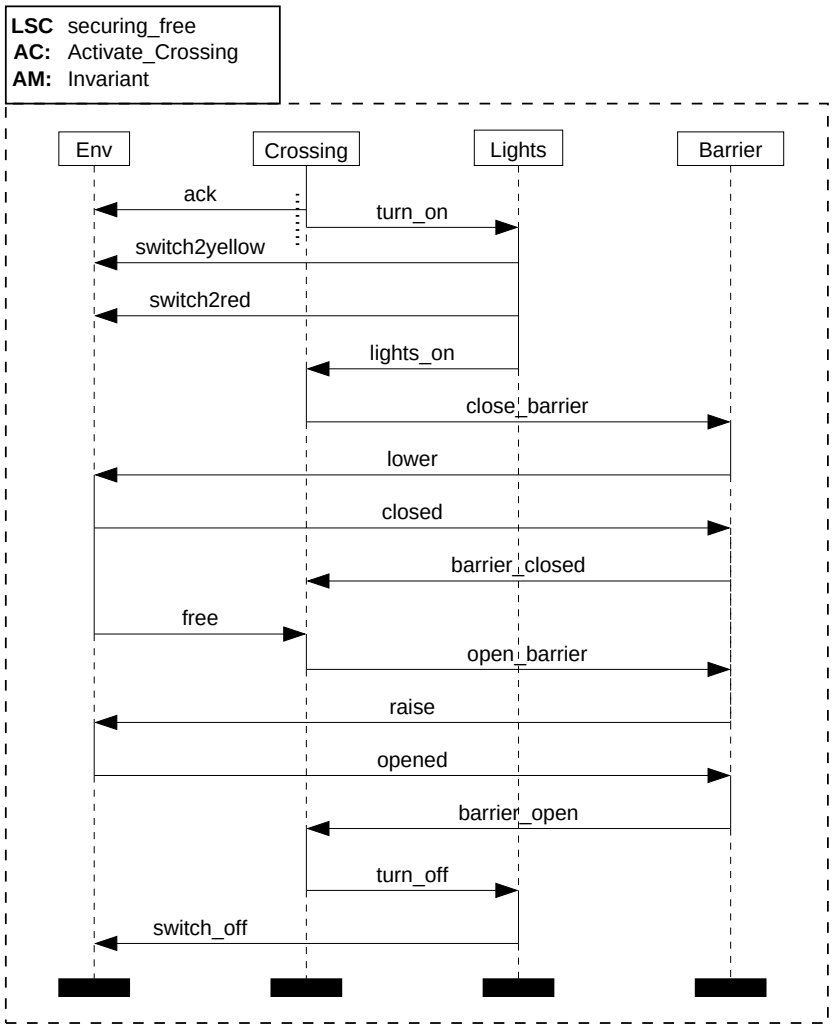


Figure 10.3: Existential LSC for the crossing showing an error case (Architectural Design Phase)

that LSCs in the Architectural Design Phase can become quite large and that the prevailing feature used are messages. Also note that the exact ordering between the acknowledge sent to the train (`ack`) and the command to turn on the traffic lights (`turn_on`) is not expressed, since this detail is not yet worked out. ■

10.2.2 Debugging by Existential Verification

At the end of the Architectural Design Phase the system architecture is known and for each hierarchy level a set of existential LSCs has been drawn to capture the typical communications between (sub-)systems/entities at this level. The next step is to define the behavior of each entity and thus of the complete system, which is done in the Behavioral Design Phase. In a model-based development process the goal is to produce a reference model for the subsequent phases. Steps toward this goal are simulation, analysis and formal verification. We present those related to LSCs in this and the following section.

Once a suitable portion of the behavior has been modeled, the existential LSCs specified in the previous phase can be reused for verification. Since these LSCs are existential, it is checked if the part of the system constructed so far, is indeed capable of performing the typical communication sequence at least once. Or in other words: Is there at least one run, which conforms to the specified scenario? We call this use case *existential verification*. This check can be applied as the model is built, i.e. the model need not be complete. The developer decides if and when existential verification is to be applied, which LSCs are to be checked, etc.

Existential verification can hence be employed as a debugging aid early in the development in order to verify that the current state of the SUD model is able to perform the basic interactions as specified by the existential LSCs. The LSCs created in the previous phase can be used directly without any modifications, allowing a facile reuse.

Negative scenarios are verified in the same way as normal existential LSCs, only the result is interpreted differently. If a negative scenario is proven to be possible in the model, this indicates an error, since the *absence* of the specified interaction sequence is expected. Checking negative scenarios is thus performed at the same stage as universal verification, i.e. at the end of the Behavioral Design Phase when the model has reached a mature and stable state.

10.2.3 From Scenarios to Protocol Specifications

Modularization of Existential LSCs

Existential verification, potentially in conjunction with simulation and analysis of the model, increases the confidence in the correctness of the model being built compared to a traditional design process. These measures do not ensure, however, that a requirement is fulfilled under all circumstances, since not all possible cases are covered. This can be achieved by formally verifying key properties specified as universal LSCs, which is applied towards the end of the Behavioral Design Phase, i.e. when the model is mature and stable.

The question at this point is where do these universal LSCs come from. The existential LSCs specified in the Architectural Design Phase already contain these key properties. The aim is thus to reuse these LSCs for the specification of the properties to be model checked. Since such existential LSCs tend to be rather large as the LSCs in figures 10.1 - 10.3 illustrate (see also sections A.1.1, A.2.1 and A.3.1), the relevant protocols have to be identified and extracted.

The procedure for producing smaller, more concise universal LSCs from extensive existential LSCs is guided by identifying fragments belonging together, i.e. forming one sub-protocol, one property. A first indication in this direction is gained by recognizing identical or similar parts in a set of existential LSCs. The existential LSCs produced in the preceding phase are not completely disjunct, but partially overlapping, since an LSC showing a good case, is often complemented by others, which describe potential error situations. The LSCs in figure 10.2 on page 226 and figure 10.3 on page 227 for instance are largely identical. At the developers discretion the overlapping parts are factored out yielding a number of shorter, core communication sequences. We call this extraction of universal protocols from an existential LSC *modularization*.

Strengthening the LSCs

Before the extracted sub-protocols can be verified they have to be adjusted to the changed usage. In a first step the quantification is changed to universal and environment instances, if present, are distinguished from normal instances in order to enable the consideration of internal and extracted assumptions. Generally, other modifications have to be applied as well to enable formal verification of the extracted universal LSCs. We refer to this

second part of the transformation from existential scenarios to universal protocol specifications as *strengthening*. The strengthening activities can also be viewed as a generalization, since general behavior is deduced from particular runs.

The typical interaction sequences created in the previous phase can usually be expressed by basic LSC features, mostly messages. As the aim now is to prove that the specified interaction is executed under all circumstances, more information and precision is needed, requiring more advanced features. Mandatory conditions or timing constraints, are e.g. added and indetermined ordering of messages expressed e.g. by coregions is made more precise by requiring a specific order or employing simultaneous regions.

In order to prove that the model conforms to the behavior specified by a universal LSC, generally the environment actions have to be restricted in addition to specifying more precise charts. The assumptions effecting these restrictions can be grouped into two categories: Assumptions ensuring reactions of the environment and assumptions expressing case distinctions. The former category comprises hot locations and timing annotations on the environment axis. The latter category of assumptions is used to focus on a specific behavior of the SUD. In order to successfully verify that the barriers are closed, the barriers have to be operational, i.e. the system behavior has to be restricted to this case. Conditions and local invariants are typically used for this purpose. Both internal and external assumptions can be used, although it is recommended to use internal assumptions if possible, since they are more intuitive due to the fact that they are contained directly in the commitment LSC.

Activation conditions of universal LSCs often need to be made more precise, i.e. more restricted, in order to rule out undesired activations of the LSC. For some charts it will become apparent that more information about the history of a run is needed in order to determine, if the LSC should be activated. Activation conditions are thus in such cases extended to pre-charts.

The process of model checking the resulting LSC specifications typically requires several iterations, in which the mentioned strengthening actions are performed incrementally. First the commitment side of an LSC is strengthened in conjunction with some basic restrictions of the environment. If the specified property does not hold, there is either an error in the model or the LSC is not precise enough. In the first case the model is corrected and the proof is executed again. Each change to the model moreover requires

a re-assessment of the other LSCs, i.e. it has to be determined, if they are affected by the altered model behavior. In the second case the LSC has to be analyzed and corrected, e.g. by adding an assumption.

Care has to be taken when specifying assumptions in order to be consistent with the initial requirements. Before an assumption is added to a commitment LSC the designer has to determine that the assumption is valid, i.e. conforms to the initial requirements document and is justified by the environment.

The result of this phase is a model, which serves as a reference for the further steps in the development process. The verification activities described above are an important step towards such a reference model, in addition to other measures like simulation and analysis. The reference model constitutes an early, virtual integration of the components of which the SUD is comprised. Existential and universal LSC-based verification allows to check the integration and interplay of the individual components prior to their implementation.

Note that verification does not ensure the complete correctness of the model, but only wrt. the specified properties. If the specified and verified properties cover all the requirements stated in the initial document or if those requirements are complete themselves, has to be determined in a different way. Such considerations are beyond the scope of this thesis.

Example 10.2 (Modularization and Strengthening)
In this example we illustrate part of the modularization and strengthening process described above at the level of the crossing component of the train control system. For the strengthening we will not present all intermediate versions of the universal LSCs, but rather only show the final one.

The LSCs in figures 10.2 on page 226 and 10.3 on page 227 show two of the existential LSCs for this component, more can be found in section A.3.1 in the appendix. LSC `securing` (figure 10.2) shows the good case, the remaining ones describe error scenarios, which are identical in large parts. A natural modularization is to separate the activities necessary for a train approaching the crossing from those for a train, which has already passed the crossing. This cuts off the lower part of the LSCs, from message `open_barrier` on downward; figure 10.4 shows the resulting already strengthened, universal LSC.

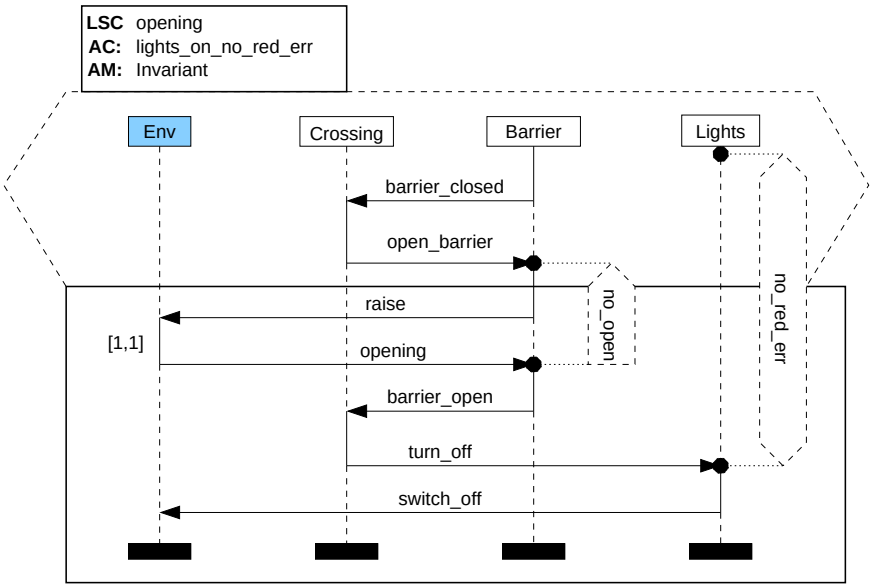


Figure 10.4: Universal LSC for the opening of the barrier

The strengthening of this LSC for the successful verification required adding of some assumptions (in the form of local invariants and timing intervals) and the extension of the activation condition to a pre-chart. The pre-chart is needed in order to express that the barrier has really been closed for the approaching train; otherwise the shown protocol can not be guaranteed. The barrier should remain closed the whole time after it has been lowered. This is expressed by local invariant `no_open`, which starts right after activation of the LSC and ends simultaneously to the barrier leaving its lower end position. The message for the opening of the barrier (`open_barrier`) triggers the activation of the actual LSC. It is additionally necessary to require that the lights are operational the whole time, because a failure occurring before the command to turn off the lights (`turn_off_lights`) is sent results in a premature switching off of the traffic lights. This would entail that the liveness property, which requires the lights controller to send the switch off signal to the physical lights once it receives the corresponding command can not be fulfilled. The last assumption to be made is that the barrier functions correctly and starts opening within the allotted time, which is expressed by the timing interval on the environment axis.

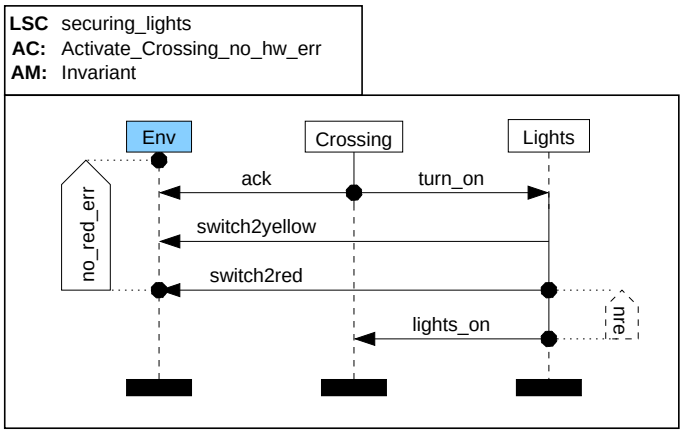


Figure 10.5: Universal LSC for the turning on of the traffic lights

The part of the scenarios describing the securing procedure before the train arrives at the crossing can be broken down into several sub-protocols. The first such piece is the turning on of the lights, which is identical in the LSCs of figures 10.2, A.38, A.39, A.41 and A.40 and which is only slightly modified in the remaining existential LSC shown in figure 10.3. The next sub-protocol covers the closing of the barrier and is not present in all LSCs, but in the ones shown in figures 10.2, 10.3, A.33 and A.41.

Figure 10.5 shows the universal LSC for the turning on of the traffic lights, again already strengthened. For this requirement to hold the physical parts of the crossing have to be operational when the activation request arrives from the train, otherwise no action on part of the crossing controller will ensue. This means that the activation condition has to be extended, which is reflected by the new condition identifier. We furthermore assume that the traffic lights are operational at least until the red light has been on for the required amount of time, which is indicated by the two local invariants `no_red_err` and `nre`. The third part of the LSC, which has been strengthened, is the coregion covering the messages `ack` and `turn_on`, which has become a simultaneous region due to more detailed knowledge about the system.

Figure 10.6 shows the universal LSC for the closing of the barrier. Here the LSC itself is strengthened by adding the mandatory condition that the crossing has to be in a safe state one step after the barrier controller reports the closing of the barriers. Additionally a timing interval is added to the

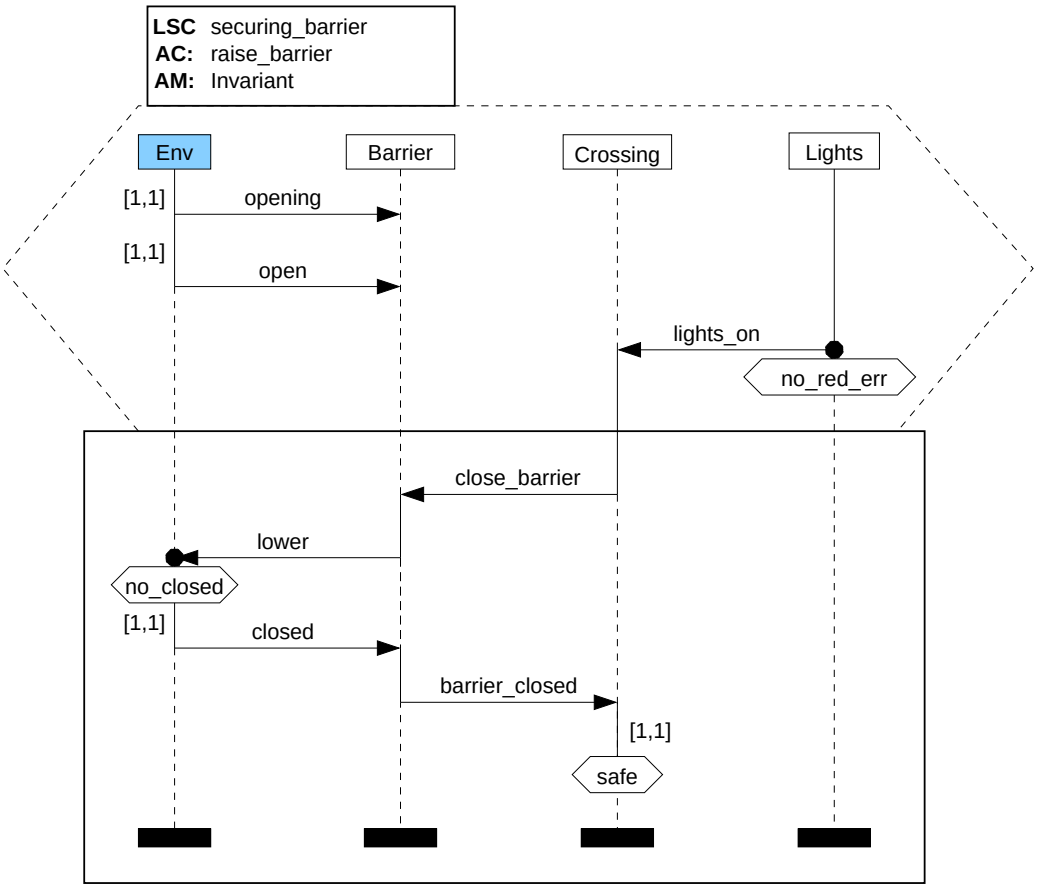


Figure 10.6: Universal LSC for the closing of the barrier

environment instance in order to ensure that the `closed` message is sent in time. Again the activation condition has to be extended to a pre-chart in order to be able to successfully verify this LSC; see the description accompanying figure 9.2 on page 208 in section 9.1 for a detailed explanation and justification of the pre-chart elements.

Since the pre-chart of this LSC requires that there has been a successful opening of the barrier during the previous securing procedure, the requirement shown in figure 10.6 covers only barrier closing processes *after* the first train has passed the crossing. Therefore a separate, initial LSC covering this case has to be additionally verified. It contains only the LSC itself and is shown in figure 10.7. Note that the activation condition is empty, i.e. true,

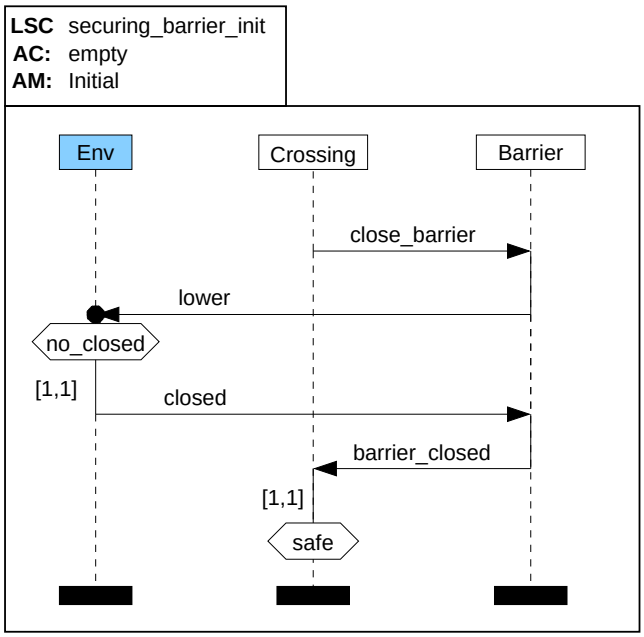


Figure 10.7: Initial universal LSC for the closing of the barrier

since no restriction is imposed on the system start.

The remaining parts of the existential LSCs, which are not covered by a universal LSC are either black-box requirements (the scenario for timeout in figure A.41 on page 334 only concerns the crossing controller and the environment) or belong on a different level of hierarchy (the scenarios for answering or not answering the status request and sending the free message). ■

10.2.4 Test Vector Generation

The reference model developed in the Behavioral Design Phase is ideally suited to derive test vectors for the Test Phase. LSCs created in earlier phases can therefore be reused for the automatic generation of test vectors for integration testing. They have been specified wrt. to the reference model and have already been used in the verification of the virtual integration, so that it is guaranteed that they specify relevant test cases.

10.3 Related Work

Damm and Harel [DH01] outline some use cases for LSCs using a UML-based development process as an example. They also advocate the use of existential LSCs in order to add more contents to Use Case Diagrams and further develop those LSCs during the object analysis phase, which corresponds to the Architectural Design Phase and partly the Behavioral Design Phase. They propose testing and formal verification activities as well, which largely coincide with the ones presented here, since both are inspired by the same activities at OFFIS. While advocating a transition from existential to universal LSCs to be used for formal verification, no details on how this transition is to be achieved are offered.

In [KL01, BKL01] we have presented an LSC-based approach for testing UML designs modeled with Rhapsody. The basic idea is that during a simulation session a watchdog is generated, which monitors if a set of user-specified properties, given in the form of LSCs, is satisfied. The simulation can either be conducted by the user or it can be driven automatically according to the specified LSCs exploiting the activation information and designated environment instances offered by LSCs. In the latter case the required inputs are generated according to the LSCs.

Once a set of LSCs has been compiled it can be employed for regression testing, i.e. if the model is changed, e.g. by adding new functionality, this set of LSCs can be rerun in order to check, if the extensions have impaired the original functionality. The LSCs used in this simulation-based testing are also ideal candidates for the derivation of test vectors for integration testing.

In [KRK02] we have investigated a similar approach, which deals with simulation-based testing of system-level hardware designs, implemented in SystemC. Here LSCs serve as a graphical front-end for LTL formulas, which are monitored in parallel to the simulation of a SystemC implementation of the SUD by the checker presented in [RHTR01].

Harel and Marelly have proposed a different use case for LSCs described in [Har00, HKMP02, HM02, MHK02]. The basic idea of this approach, called *Play-In/Play-Out*, is to play in the desired interactions in the Architectural Design Phase and use LSCs to record them. The key point is that no model representation exists at this time, so that the play in procedure is carried out via a mock-up graphical user interface. Once a set of LSCs (universal and existential) has been recorded in this way they can be used as behavior specifications, which monitor a user-guided simulation (*play-out*). This ap-

proach is intended to be used at the beginning of the development process to generate the basic system interactions in an easy and intuitive way. The view of this approach is an operational one, whereas the purpose of LSCs as presented here is a denotational description of the behavior of an SUD.

[HKMP02] additionally use model checking in order to check the consistency of several universal LSCs. This check is also used to find a sequence of steps (a superstep), which leads to another stable system state, i.e. a situation where new inputs are required in order for the system to advance further. This approach is similar to our existential verification, except that [HKMP02] consider several LSCs and are limited to one superstep.

Another advanced use case for LSCs, described by Harel and Kugler in [HK01, HK02], is bridging the gap between requirements, specified by LSCs, and a behavioral model by automatically synthesizing a first-cut model from LSCs. [HK02] outline an algorithm to automatically generate state charts from LSCs in an object-oriented system. Similar approaches exist for MSCs as well: Leue et al. [LMR98] synthesize ROOM state charts from MSCs, Krüger et al. derive Statecharts from MSCs in [KGSB99], and Krüger generates finite state machines from extended MSCs in [Krü00]. In the telecommunications field there are several approaches dealing with synthesis of an SDL model from MSCs, e.g. [RKG97] or [Man01].

Bunker and Gopalakrishnan [BG01, BG02] use LSCs for the specification and formal verification of a hardware protocol. The properties to be verified are derived by hand from the requirements document and a set of simple proof obligations is derived manually from the LSC. The application used as an example is small enough, so that a direct creation of universal LSCs is possible and no need for an elaborate design process is considered.

Mauw et al. [MRW00] present a methodology for the application of MSCs, which is not limited to a specific process, but is described in terms of an abstract design flow. They also advocate the use of MSCs in the early phases in order to capture the typical use cases, which is the classical field of application for MSCs. Other use cases are the recording and monitoring of simulation runs, reusing MSCs as test vectors and recording execution traces. No formal treatment of MSCs is assumed and the only (informal) verification activity considered is comparison between MSCs from different phases. The exact nature or basis of such comparisons is not presented and neither is the concrete relation between the MSCs from different phases, which in [MRW00] have different views (black-box or grey-box) depending on the phase.

The embedding of MSCs into a development process presented in [MRW00] is, in summary, less tight than our proposal for LSCs given in this chapter. We achieve a greater reuse of LSCs from previous in later phases of the development process, thereby reducing the development effort necessary and gaining a better integration of LSCs, especially across phase boundaries. The enhanced expressiveness and formal foundation of LSCs yields even more added value, in particular wrt. formal verification.

Existential verification of MSCs against an SDL model is offered by some SDL tools Telelogic e.g. offers a check to determine if a SDL model contains the communications shown in an MSC ([Ek98]). There are also numerous approaches to using MSCs for the specification of test cases. We refer the interested reader to [MRW00].

Chapter 11

Assessment of the LSC Language

In this chapter we explore the applicability and usability of the language of LSCs and the proposed methodology as presented in the previous chapters. We take a closer look here at the advanced use case of specification of properties for formal verification; more concretely the application field is the formal verification of STATEMATE designs. The sample application used is the train control system introduced in chapter 2. The LSCs presented in this chapter have been developed according to the methodology described in chapter 10. First existential LSCs have been specified, which are used for existential verification and from which universal ones have been derived as expounded in section 10.2.3.

The technical basis for the evaluation presented here is the STATEMATE Verification Environment (STVE), which has been presented in the chapter 1 on page 15. Section 11.1 outlines the prototypical integration of the tools supporting LSCs into the STVE (subsection 11.1.1) and deals with the particulars of specifying properties for STATEMATE designs in the synchronous (subsection 11.1.2) and asynchronous (subsection 11.1.3) simulation semantics. Section 11.2 presents the the existential and universal LSCs specified for the train control system and section 11.3 presents the experimental results. Section 11.4 concludes this chapter with an overall assessment of the LSC language.

11.1 Property Specification for Statemate

11.1.1 Integration of LSCs into the STVE

The integration of LSCs into the STVE follows the same path as STDs, since the semantical basis for both formalisms are symbolic automata. Where STDs are intended for the specification of black-box properties of single entities, LSCs aim at specifying grey-box properties, which involve the interaction of several entities. LSCs additionally focus on liveness properties as this is an inherent feature of the language, although both safety and liveness is expressible in both formalisms. LSCs are better suited for specifying communication properties due to their event-based nature, where STDs are a state-based formalism. Wrt. categorization of property specification possibilities LSCs are located at the same level as STDs.

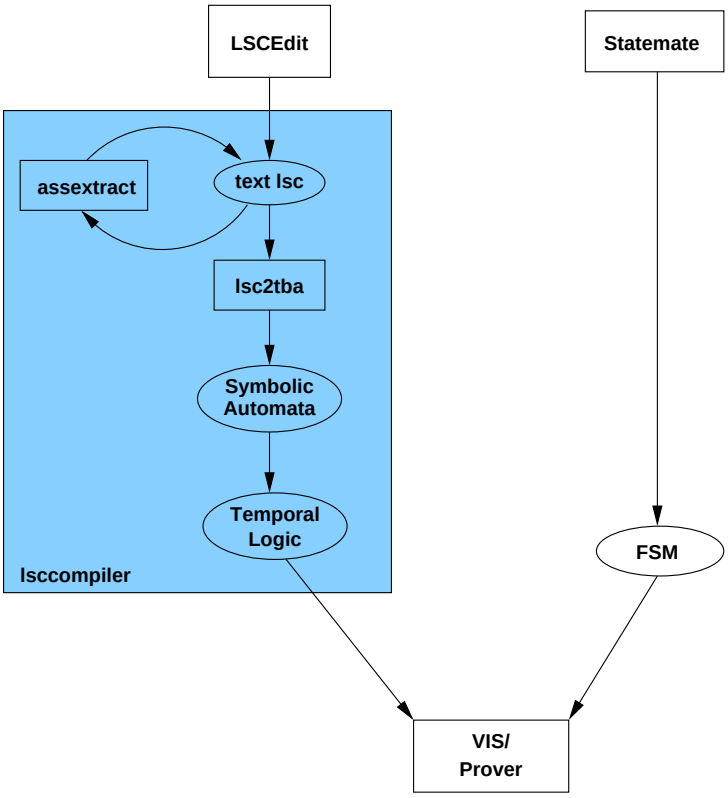


Figure 11.1: Integration of LSCs into the STVE

Figure 11.1 gives an overview of the LSC tools and their interplay with the rest of the STVE; rectangles in this picture represent tools and ellipses formats. An editor, `LSCEdit`, is available to draw and manipulate LSCs, which are stored in a textual syntax derived from the textual representation of MSC-96; see appendix C for the grammar of the textual LSC representation. The textual LSC is the basis for all further treatment of the LSCs. If applicable, the tool `assextract` performs the extraction of assumptions as described in section 8.2.1 on page 195 and adds the resulting assumption LSC(s) to the textual representation of the commitment LSC. The tool `lsc2tba` implements the unwinding algorithms for assumptions, commitments and pre-charts as defined in the preceding chapters. Both tools are integrated into the `lsccompiler`, indicated by the shaded part of figure 11.1, which controls the entire generation of a temporal logic formula from an LSC, thereby comprising both LSC-specific tools and tools, which are already part of the STVE and are responsible for the generation of temporal logic formula from symbolic automata and the treatment of assumptions, which are already part of the STVE. The tasks performed by `lsccompiler` for a commitment LSC L are thus:

1. Extract assumptions from L using `assextract`, if necessary.
2. Unwind all assumptions associated with L using `lsc2tba`.
3. Perform STVE assumption treatment using symbolic automata generated from assumptions.
4. Unwind pre-chart *pch* of L using `lsc2tba`, if necessary.
5. Unwind L using `lsc2tba`.
6. Generate temporal logic formula from the symbolic automaton for L.

The `lsccompiler` produces the formats and files required by the STVE. The subsequent steps necessary for the verification of LSCs (restriction of the system runs to the ones conforming to the assumptions, observance of the prefix given by the pre-chart and the actual model checking run) are thus already part of the existing STVE machinery and are not presented here. A description of the technical details of the STVE is out of the scope of this thesis and can be found in [Wit03].
Note that both the `lsccompiler` and `lsc2tba` are parameterized to support both the weak or the strict interpretation.

11.1.2 Property Specification for Synchronous Models

This subsection deals with the particulars of STATEMATE designs, which are modeled using the synchronous simulation semantics. Recall that in this semantics the SUD accepts inputs and produces outputs in every step (execution cycle) and time passes between two steps. Note that the treatment of time is *not* part of the STATEMATE semantics, but is left to the tools operating on the STATEMATE design like e.g. the simulator (cf. [HN96]). Time annotations consequently are interpreted in terms of steps by the STVE and in LSCs as well, i.e. timers and timing intervals in LSCs refer to steps.

There are two other peculiarities, which affect property specification with LSCs: All communication in STATEMATE step models is instantaneous and there is no notion of a lossy channel, so that messages can not be lost. LSCs specified for a STATEMATE model therefore only contain hot instantaneous messages. Note that this is also true for asynchronous models, because only a single controller (embedded controller) at a time can be modeled in STATEMATE. If several embedded controllers are contained in one STATEMATE model, they are treated as on single controller, i.e. the communications between individual controllers are considered to be internal, i.e. are step-based.

Notice that all messages occurring in the LSCs for the train control system (cf. section 11.2) refer to events in order to use them for both the weak and the strict interpretation (cf. the remarks on page 146). This includes both normal STATEMATE events and derived events like changed and written events for data items and rising or falling edges of conditions.

11.1.3 Property Specification for Asynchronous Models

The asynchronous simulation semantics of STATEMATE proves to be more complicated than the step semantics. Recall that in the asynchronous (or superstep) semantics the modeled controller accepts inputs from the environment, which trigger a series of internal steps necessary to produce the outputs, and new inputs are accepted when the controller once again reaches a stable state. The transition from one stable state to the next is a superstep. All internal steps do not consume time; time is advanced when the system reaches a stable state, i.e. after completion of the current superstep. Figure 11.2 on the next page illustrates the concepts of steps and supersteps: the circles represent global system states, large circles denote stable states,

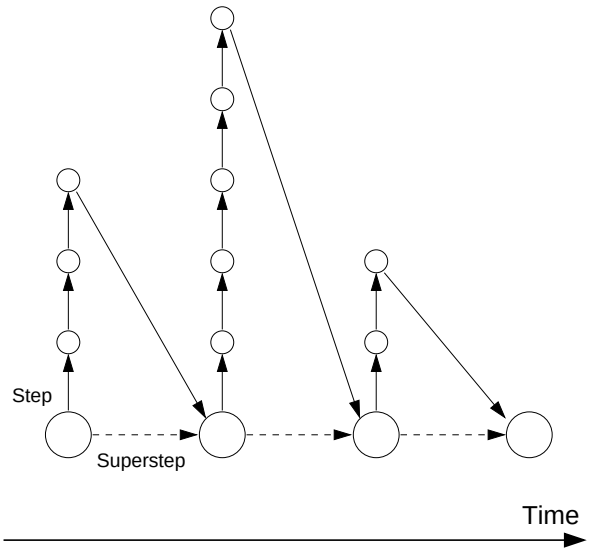


Figure 11.2: Abstract representation of steps and supersteps

i.e. states, where no further progress is possible without new input stimuli from the environment. The solid arrows between states indicate steps, the dashed ones supersteps. Note that the number of steps in a superstep varies and that consequently a superstep is a derived term, i.e. it is defined in terms of the individual steps leading to a stable situation. Such a stable state need not exist, however, i.e. the model may be divergent. For more information on the asynchronous simulation semantics see [Bro99, DJHP98].

This definition of the superstep semantics has implications on the question of which communications are observable, i.e. which model elements are available for the specification of properties. Strictly following the asynchronous semantics would mean that only those communications are observable, which occur at a superstep at the interface of the entire embedded controller. This corresponds to a pure black-box view onto the top level as all internal communications are hidden. When considering a single controller, such a view is in most cases not sufficient, because the internal steps determine if and how a stable state is reached, i.e. they are an integral part of the behavior. Since only a single embedded controller can be modeled in STATEMATE, its internal behavior can typically not be disregarded. For LSCs, moreover, a black-box view is inappropriate, since they are intended for specifying properties involving several components, i.e. a grey-box view.

The basic granularity of observability in the superstep semantics consequently still is a step. This creates the paradoxical situation that sequentiality of internal communications must be expressible, while assuming that all internal communication occurs at the same point in time. Messages in LSCs thus refer to both internal and external communication raising the question what timing annotations in LSCs should refer to: steps or supersteps? We have decided to let them still refer to steps, since this is the only consistent choice. Steps constitute the finest level of granularity and also form the semantical foundation (cf. [DJHP98]), with supersteps being derived from this base.

Supersteps can be referenced in LSCs, since the model representation in the STVE includes an output (event *stable*), which indicates a stable state, i.e. a superstep boundary. Quantitatively dealing with supersteps is often desired, since this allows to state requirements on timed behavior as time is associated with supersteps. Recall, however, that time itself is not part of the formal semantics of STATEMATE, so that we choose to allow the formulation of timing constraints in terms of supersteps by counting them.

In the remainder of this subsection we investigate how timing constraints referring to supersteps can be specified in LSCs. Two approaches are conceivable: explicit usage of the signal indicating stabilization of the model in LSCs and introduction of a dedicated counter into the model, which counts supersteps and can be referred to in the LSCs.

The first alternative requires the user to explicitly indicate the points, where a stable state in the model is reached, by including the *stable* event in the LSC. By explicitly referring to this event in an LSC specification it is possible to count supersteps. This *explicit enumeration* approach is illustrated in figure 11.3 on the facing page, which shows the adaption of LSC `securing_yellow_err` (cf. figure 11.13 on page 259) to the superstep semantics. Figure 11.3 depicts the situation where the yellow light is out of order, but the red light is still operational and is switched on immediately after detecting the failure of the yellow light. The red light has to be on for the combined duration required for yellow and red light.

The *stable* event is represented by mandatory conditions rather then messages, since several identically mapped messages would bar the usage of such LSCs in the strict interpretation. Note that the required six[1] supersteps be-

[1]The differing number compared to the LSC for the step semantics, which requires seven steps (cf. figure 11.13 on page 259), is due to the fact that the delay between occurrence

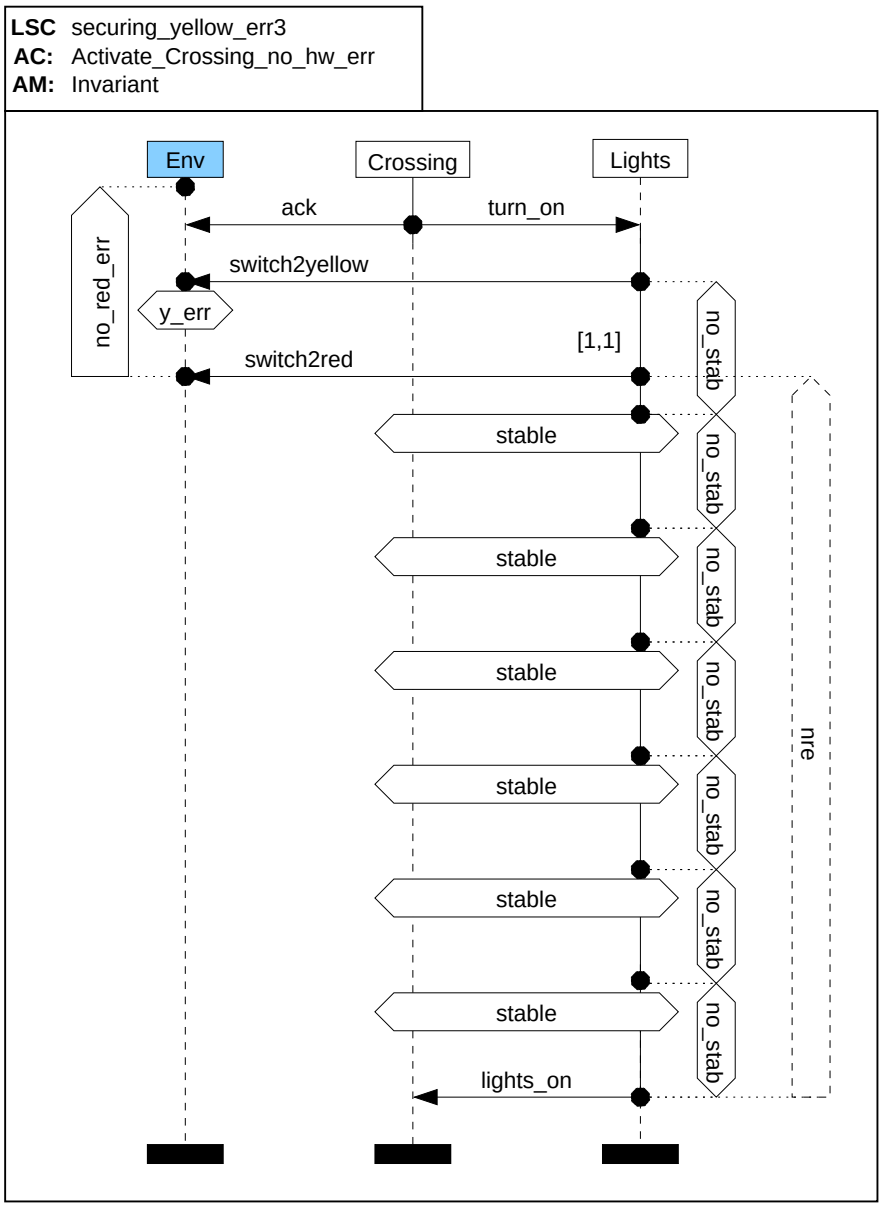


Figure 11.3: Example for counting supersteps by explicit enumeration

and observation of the timeout event is a step delay. Therefore in LSCs specified for the superstep version of the train control case study slight the numbers used in timing annotations can differ compared to the LSCs specified for the step version.

tween the command to switch on the yellow light and the indication that the lights are on are enumerated explicitly. In between two *stable* events no other stable indication is allowed, which is expressed by the mandatory local invariants `no_stab`. The first, resp. last local invariant enforce that `switch2yellow` and `switch2red` occur within the same superstep, resp. that `lights_on` is observed in the sixth superstep.

The advantages of this approach are that all relevant information is expressed explicitly in the LSC and that no additional complexity is introduced, since only information is used, which is already available (the *stable* event). Its disadvantages are that drawing such LSCs, especially those containing many or large timing constraints, is tedious and not very user-friendly, and that not all timing constraints can be expressed in this manner. Exact bounds can be specified as shown in figure 11.3 and lower bounds are expressible by placing the corresponding number of `stable` conditions and `no_stab` invariants between the points, which are to be separated by the lower bound. Upper bounds on the other hand can not be expressed by this approach, because this requires to explicitly specify all possible points of occurrence of the constrained message (after one superstep, after two superstep, etc.). While the enumeration of the *stable* events would still be possible with the aid of a coregion ranging over the message and all `stable` conditions representing the upper bound, the local invariants forbidding the occurrence of additional `stable` conditions can not be formulated here, since their start and end points can not be guaranteed to occur in the correct order within the coregion. Moreover, *all* supersteps, regardless of the occurrence time of the constrained message, are always required to be observed before any subsequent message may occur. This is generally not desired.

The second alternative entails adding a counter to the model, which counts *stable* events, i.e. supersteps. The counter counts modulo a certain value, and the current value of this counter is an output of the model and can thus be referred to in an LSC. Figure 11.4 illustrates how this counter can be utilized to specify timing constraints[2]. The beginning of the constrained LSC part is marked by a possible condition, where the current value of the counter (`cnt`) is stored in a special variable (`X` in this case). This variable is a *flexible specification variable* [ACS99, Wit03], i.e. a variable, which can assume an arbitrary value of its domain in each step. Technically a flexible

[2]Note that for this example we explicitly show the corresponding boolean expressions for the conditions and the local invariant referring to the superstep counter for clarity.

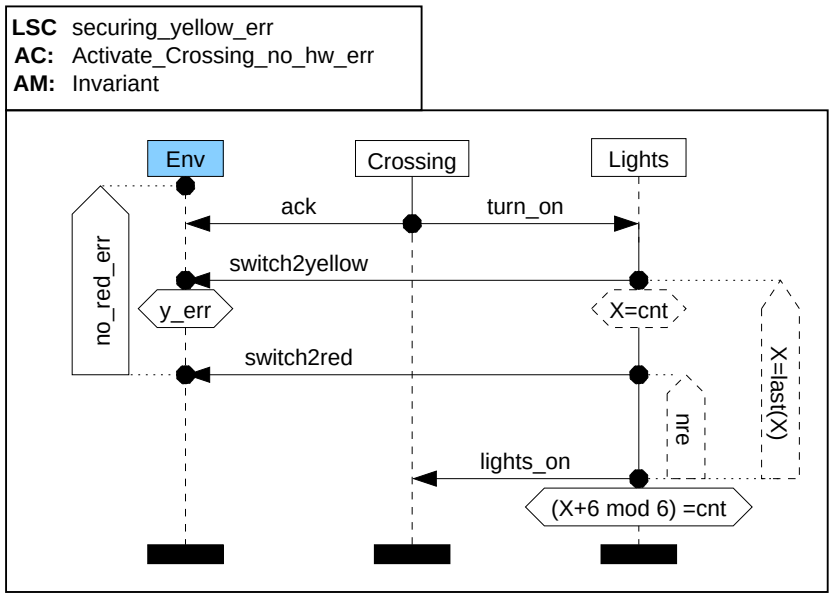


Figure 11.4: Example for counting supersteps by querying a counter within the model

specification variable is a free input, to which the model checker may assign a value in each step. The possible condition in this case ensures that `X` stores the current value of the superstep counter: the model checker will assign the correct value in order to be able to advance through the LSC. The possible local invariant `(X = last(X)` ensures that the flexible specification variable retains the value it has stored until the end of the timing constraint. Otherwise the model checker would assign a different value to `X` in order to violate the following mandatory condition expressing the timing constraint. In this example the condition is attached to message `lights_on` expresses that the counter has advanced six supersteps since switching on the yellow light.

The counter approach remedies the drawbacks of the explicit enumeration strategy, since expressing superstep timing constraints in this manner is more user-friendly and the expressiveness of the constraints is not restricted. The disadvantage of this approach is the additional complexity, which is introduced into the model by the counter.

Both approaches have additional disadvantages concerning assumptions. For both approaches no internal assumption treatment is offered automatically as in the case where no superstep timing constraints are present. In

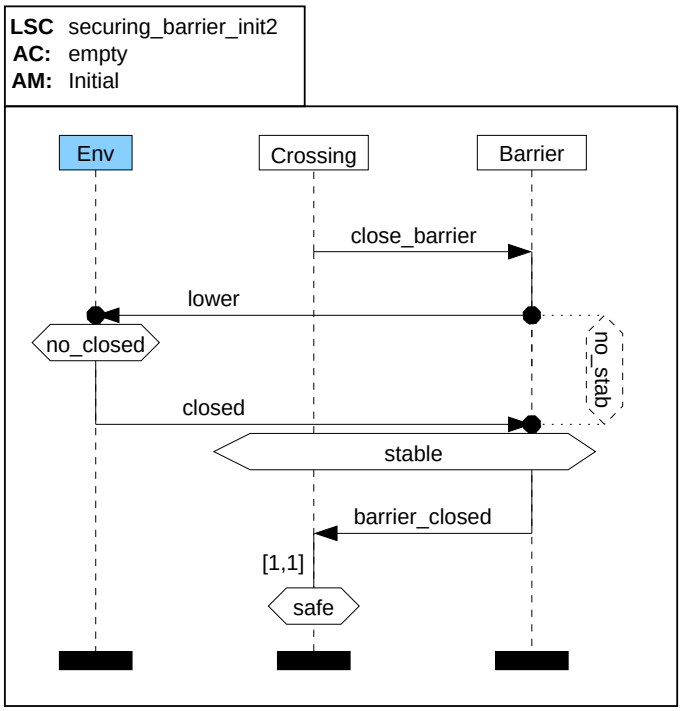


Figure 11.5: Initial universal LSC for the closing of the barrier, specified for the superstep semantics using the explicit enumeration of supersteps

both cases it is possible for the user to specify the commitment LSC in such a way that the same effect results, however. Figure 11.5 demonstrates how this is achieved with the explicit enumeration approach using the example of LSC `securing_barrier_init` (cf. figure A.57 on page 349). Here the combination of possible local invariant and mandatory condition enforces the occurrence of `closed` after exactly one superstep similar to the LSC shown in figure 11.3. The possible mode of the local invariant is responsible for the internal assumption effect, since the LSC is exited, if *stable* occurs without simultaneously observing `closed`. Note that timing interval between `barrier_closed` and `safe` is still expressed in terms of steps, because both are to be observed in the same superstep, but are sequentially ordered at the step level.

Figure 11.6 on the next page shows the same LSC as figure 11.5, but using the superstep counter. In order to use the existing internal assumption treatment, the condition expressing the timing constraint has to be connected to

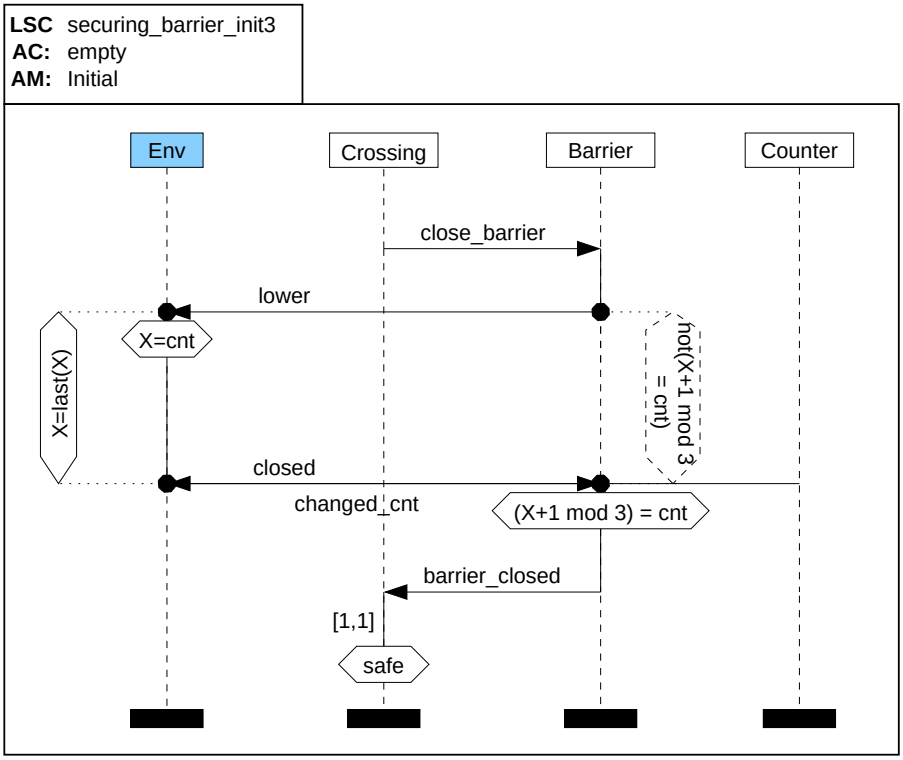


Figure 11.6: Initial universal LSC for the closing of the barrier, specified for the superstep semantics using the superstep counter

a message originating in the model, which has to be related to the timing constraint. The implicit event indicating a change in the value of the superstep counter (`changed_cnt`) ideally fits these requirements. In addition to the conditions and local invariants required for this approach as described above another local invariant has to be employed, which ensures that the condition containing the timing constraint is evaluated at the *first* occasion where the modulo counter reaches the desired value. Otherwise the environment would be allowed to send `closed` not only one superstep after observing `lower`, but also after four, seven, etc. supersteps. This is achieved by local invariant `not((X+1 mod 3) = cnt)`. Note that this approach requires the counter to be represented in the structural description of the model, since it must explicitly be included in the LSC as an instance. This is currently not automatically possible in the STVE, but can be done manually by explicitly modeling the counter in STATEMATE.

Assumption extraction is in general possible for the counter approach, provided that the counter instance can be referenced in the LSC and all local invariants and conditions are specified on the environment instance. Extracting assumptions does not work for the explicit enumeration approach, because the *stable* event is represented by a condition and not a message. A condition can only restrict the point in time when an associated (via a simultaneous region) message is generated, but the occurrence time of such a message can not be enforced.

Both approaches can be used for user-specified assumptions, since here only the interface between the considered part of the SUD and its environment is visible. This entails that the outputs for both the global *stable* event and the counter variable are accessible and can thus be used as message annotations.

Specifying timing constraints in terms of supersteps is, in summary, more complicated and more restricted than expressing them for the synchronous semantics. This, however, is *not* due to lacking expressiveness of the LSC language, but is caused by the idiosyncrasies of the asynchronous simulation semantics of STATEMATE. The measures presented in this subsection are an attempt to sensibly specify properties with LSCs in this setting. In the remainder we will consider both possibilities of expressing superstep timing constraints if applicable.

11.2 Specification of the LSCs for the Train Control System

The LSCs presented in this section have been developed according to the specification methodology given in chapter 10. Starting in the Architectural Design Phase the LSCs are developed in a top-down manner, i.e. beginning with the top level activity of the STATEMATE model (`SYSTEM`) and descending to the lower levels of hierarchy. We first specified the properties for the step model and reused these LSCs also for the superstep model, if possible, i.e. if no timing annotations were used. If timing annotations occurred in an LSC, altered superstep versions were created using the two approaches discussed in section 11.1.3 where applicable. We consequently for each level of hierarchy first present the LSCs for the step model and then those LSCs, which were adjusted for the superstep semantics.

The existential LSCs specified for `SYSTEM` are collected in appendix A.1.1. They show the basic interactions between the activities identified at this level (`TRAIN`, `COMMUNICATION` and `CROSSING`; cf. section 2.2.1) and their environment. The LSC in figure A.1 on page 298 shows the good case, where the crossing is secured as desired and the train passes it without needing to stop. The remaining LSCs show different error situations: figure A.2 and A.3 depict the consequences of a yellow, resp. and red light failure, figure A.5 show a failure of the barrier, figure A.7 exemplifies the situation, where the train realizes that it will not reach the crossing in time and thus sends the freeing message, and figure A.8 on page 305 finally depicts a timeout at the crossing, e.g. resulting from a defect pass sensor. The LSCs describing the red light and barrier failure have been characterized in two slightly different ways: One only describing the desired interactions (figures A.3 and A.5), the other additionally prohibiting the sending of the safe message in response to the status request (figures A.4 and A.6). The LSC dealing with exceeding the maximum barrier closed time also comes in two variants, one with (figure A.8) and one without a timer (figure A.9) enforcing the MBCT in the LSC.

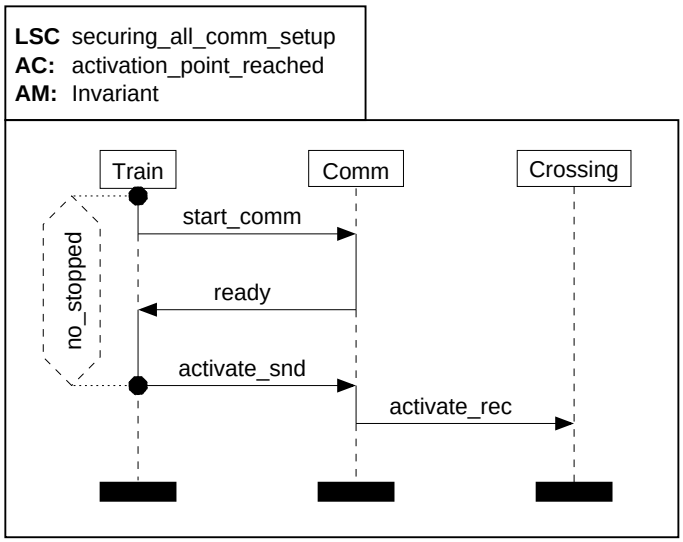


Figure 11.7: Universal LSCs for establishing the communication channel

The modularization and strengthening of these existential LSCs results in the universal LSCs shown in figures 11.7 - 11.10. Figure 11.7 shows the common prefix of all existential LSCs, the establishing of the communication

channel between train and crossing. Since this part of the protocol is assumed to be failsafe in the model, the only strengthening measure needed is to exclude that the train stops due to a not secured crossing before even the activation command has been transmitted. Such a behavior would indicate a severe problem in the speed supervision or localization components or activation point calculation of the train, which are assumed to work correctly here. The corresponding assumption is being taken care of by local invariant `no_stopped` on the `Train` instance.

This commitment LSC requires two external assumptions. The first inhibits the arbitrary reactivation of the LSC by ensuring that the activation condition (reaching the activation point), which is indicated by a message sent by the environment, does not occur again as the train is already approaching a crossing. The corresponding assumption LSC is shown in figure A.24 on page 318. Note that this property is not expressible by an internal assumption, i.e. a local invariant forbidding the recurrence of the activation condition. The effect of such a local invariant would be that the first, correct activation of the LSC is terminated by a recurring activation condition, but since the securing procedure has been initiated within the model, the corresponding protocol would be executed correctly, showing the desired communication sequence. The still active second incarnation, however, would not recognize this, because it has been activated out of turn and did not observe the prefix of the current message exchange. Therefore an external assumption, either extracted or user-specified, is necessary. In this case a user-specified assumption is employed, because there is no environment instance present in the commitment LSC.

The second external assumption LSC (`no_activation_point_ass`) shown in figure A.26 on page 319 is necessary in order to restrict the occurrence of the activation condition to a point in time, where the system is operational, i.e. after the initialization step. Since the first message (`start_comm`) must occur eventually due to the hot instance head location, it is essential that the commitment LSC is not activated in step zero, where in real life no activation point can be reported. The assumption LSC consequently forbids the initial report of reaching the activation point and therefore also its activation mode is initial. This assumption is again not expressible within the commitment LSC, since the activation modes of the commitment LSC is invariant, whereas a restriction of the initial state is required as an assumption. Note that the assumption LSC effectively contains no messages, since only the initial state is of interest, which is restricted by the activation condition.

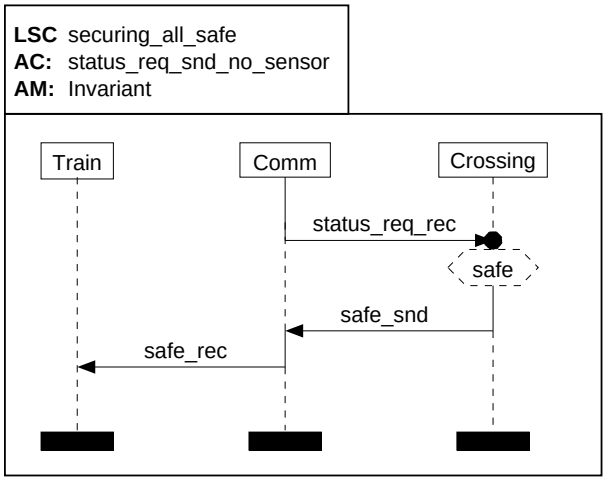


Figure 11.8: Universal LSC for a positively answered status request

The remaining three LSCs consider the different possibilities after the activation message has been transmitted, the other parts of the existential LSCs of section A.1.1 are properties of the crossing and thus are covered by LSCs for this component. Figure 11.8 depicts the message sequence for a successfully secured crossing, the crossing consequently is expected to be in a safe state when the status request arrives. Apart from adding the possible condition to ensure that the crossing indeed is in a safe state when the status request is received, it is necessary to ensure that the pass sensor is functioning correctly and does not prematurely indicate that the train has already passed the crossing. This is achieved by extending the activation condition forbidding the report of a train on the sensor.

Figure 11.9 depicts the situation, where the status request arrives when the crossing has not reached the safe state, indicated by the possible condition. Until the train has passed the crossing, no safe report from the crossing is permitted as expressed by the mandatory local invariant `no_safe_msg`. Therefore the train must stop (message `stopped`) and the driver must authorize the continuation of the journey (message `release`) before being allowed to pass the crossing.

Figure 11.10 on page 255 shows the situation where the train realizes that it will not reach the crossing in time, i.e. before the maximum barrier closed time is reached, and sends the free message to inform the crossing. Here two strengthening extensions are necessary. The first is the addition of the

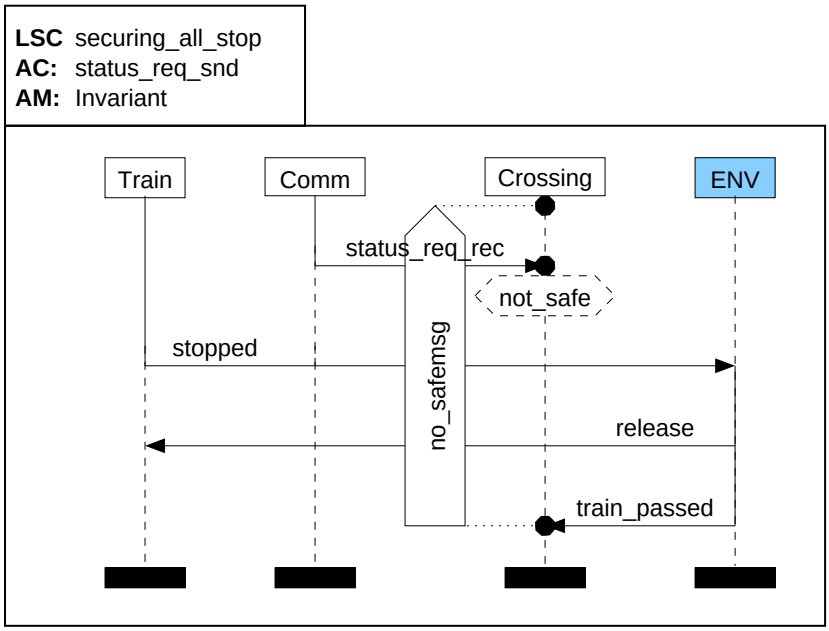


Figure 11.9: Universal LSC for a not answered status request

possible condition `safe`, which expresses the assumption that the crossing is still secured when the free message arrives, since otherwise no reaction on part of the crossing would ensue; cf. section 2.2.4. The activation condition is furthermore extended to a pre-chart, because in order for the train to send the free message the timer within the train first must generate its timeout event (cf. figure 2.7 on page 34). This requires that the corresponding inputs (`V_STILL_SAFE_P` and `V_BRAKE_POINT_P`) have to be set correctly before the maximum barrier closed time elapses, which is represented by message `ETA_violation`.

The last LSC (USAF1 in table 11.1) has to be adjusted for the superstep semantics due to the timing interval in the pre-chart. LSC USAF2 contains an approximation of the original time constraint for the explicit enumeration approach, since the constraint used in USAF1 specifies an upper bound, which is not expressible with this approach. An exact bound is used in USAF2 instead; in order to complexity for verification reasonable a bound of 15 has been chosen. USAF3 expresses the same (exact) bound, but uses the superstep counter. USAF4 also employs the counter approach, but expresses the original timing interval `[1,40]`.

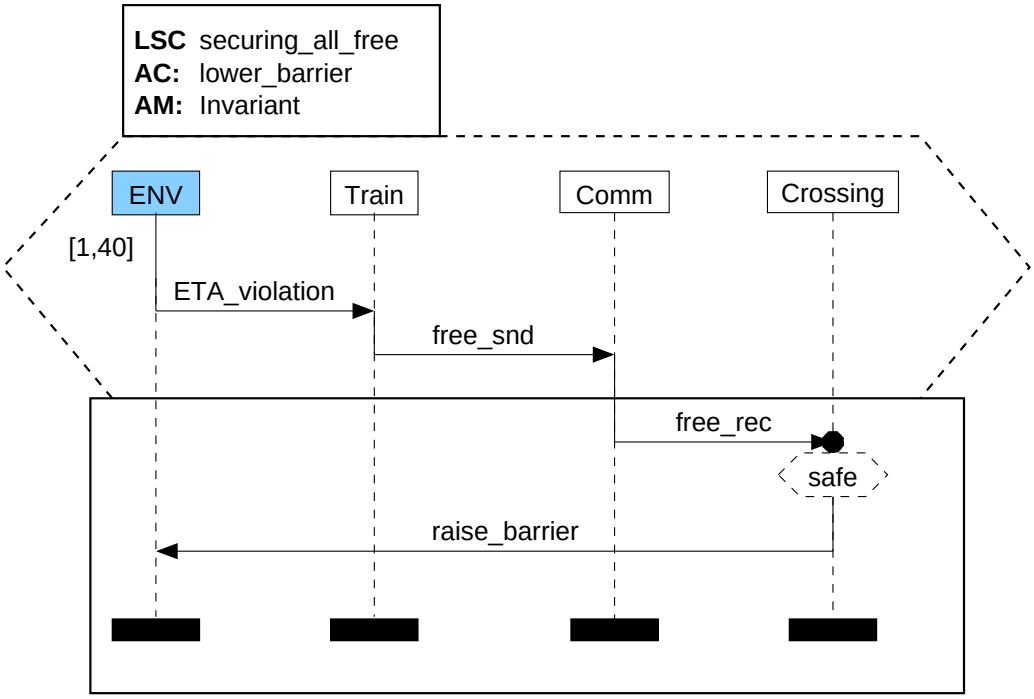


Figure 11.10: Universal LSC for the sending of the free message

Table 11.1 on the following page sums up all LSCs specified for the top level of the train control case study and assigns a property name to each commitment LSC, which will be used in the remainder of this chapter. Names for existential (universal) LSCs start with an 'E' ('U') and are followed by an abbreviation of the LSC name and a number to distinguish different variations of an LSC. Both variants (with and without the local invariant forbidding the sending of the safe message) of the existential LSCs for a red light and barrier failure are included; the two LSCs (with and without timer) for the timeout scenario are present as well. It also shows the external user-specified assumption LSCs linked to each commitment LSC and the last column contains the figure number for the LSC for easier reference. Note that no figure is included for USAF2, since the LSC would be too large to be represented in a readable way.

Moving down one level in the Activity Chart hierarchy there are two activities, which are further structured: `Train` and `Crossing`. The train activity mostly contains data communications and computations, whereas the focus

Property	Commitment LSC	Assumption LSCs	Figure
ESSA1	securing_all		A.1
ESSAY1	securing_all_yerr		A.2
ESSAR1	securing_all_red_err		A.3
ESSAR2	securing_all_red_err2		A.4
ESSAB1	securing_all_barr_err		A.5
ESSAB2	securing_all_barr_err2		A.6
ESSAF1	securing_all_free		A.7
ESSAT1	securing_all_timeout		A.8
ESSAT2	securing_all_timeout2		A.9
USACS1	securing_all_comm_setup	correct_activation_point_ass no_activation_point_ass	A.10
USAS1	securing_all_safe		A.14
USAST1	securing_all_stop		A.16
USAF1	securing_all_free		A.18
USAF2	securing_all_free2		—
USAF3	securing_all_free3		A.20
USAF4	securing_all_free4		A.22

Table 11.1: Properties for activity `SYSTEM`

of model checking STATEMATE designs is the verification of the control part of the model. Therefore the LSCs in figures A.27 - A.29 in appendix A.2 on page 320 only cover these aspects of the train. Figure A.27 on page 320 shows the good case (ETSC1), where the crossing is secured successfully, and is identical to the good case on the top level (ESSA1) as far as the train is concerned. Two LSCs showing error cases exist: one describing the situation, where the train has to stop in front of a not secured crossing (ETSCF1, figure A.28 on page 321), and the other considering the case, where the train does not reach the crossing in time and the internal timer consequently sends a timeout to `ACTIVATE_CROSSING` (ETFC1, figure A.29 on page 322). In the latter LSC the timer, which is part of activity `ACTIVATE_CROSSING` receives the indication that the train has reached the last position, where it could still stop in front of the crossing, and is not able to pass the crossing before the maximum barrier closed time elapses (message `ETA_violation` in figure A.29). This results in the sending of the timeout signal to `ACTIVATE_CROSSING`, which in turn sends the free message to the crossing.

Modularization of the existential LSCs yields only a single universal LSCs, since most of the communication is carried out between activity `ACTIVATE_CROSSING` and the communication component and thus is already adequately covered by LSCs on the level of `SYSTEM`. The message exchange described in LSC ETFC1, however, pertains to this level of decomposition and is therefore extracted into a universal LSC, which is shown in figure 11.11. It has already been strengthened by adding several assumptions in the form of local invariants in addition to extending the activation condition to a pre-chart.

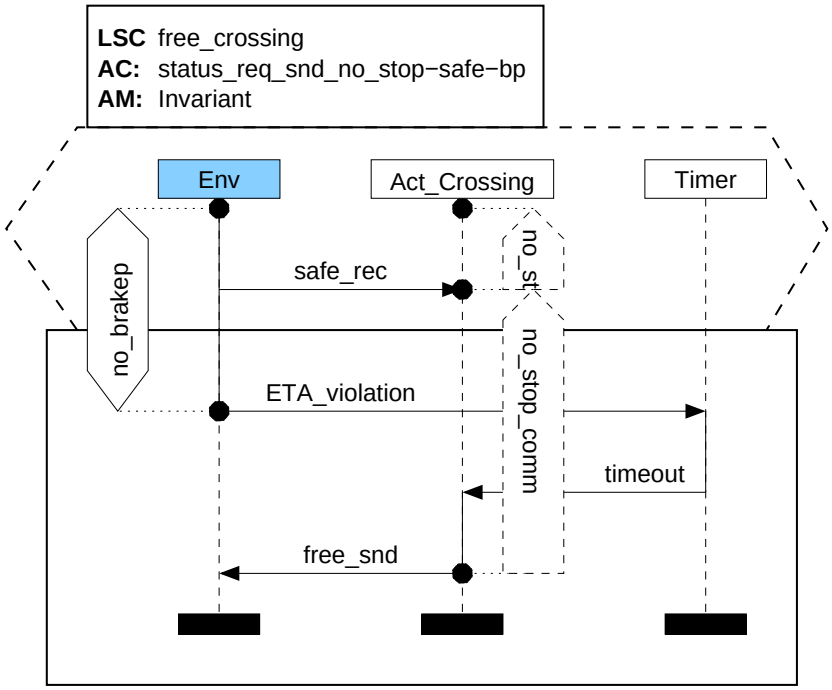


Figure 11.11: Universal LSC for the sending of the free message

The pre-chart expresses that a precondition for the sending of the free message is that the crossing has reported status safe, because otherwise a timeout would be meaningless (cf. section 2.2.2). The local invariant `no_brakep` expresses the assumption that the point which indicates the last chance for stopping in front of the crossing (condition `V_BRAKE_POINT_P`; cf. figure 7.1 on page 155) is not signaled repeatedly by the environment. Local invariant `no_st` stands for the assumption that the train does not stop due

to a not secured crossing before receiving the safe report of the crossing and thus expresses the assumption that the computation of the activation point is correct. Both these local invariants must already hold when the status request is sent, so that they are included also in the activation condition. Additionally it has to be guaranteed that the status report from the crossing does not arrive prematurely, i.e. simultaneously with the sending of the request, which is prohibited in the activation condition as well. Local invariant `no_stop_comm` is needed to express the fact that the train does not pass the crossing (indicated by terminating the connection with the crossing, event `SP_COMMUNICATION`) before the free message has been sent.
Table 11.2 shows the LSCs specified for the train.

Property	Commitment LSC	Assumption LSCs	Figure
ETSC1	securing_crossing		A.27
ETSCF1	securing_crossing_fail		A.28
ETFC1	free_crossing		A.29
UTFC1	free_crossing		A.30

Table 11.2: Properties for activity `TRAIN`

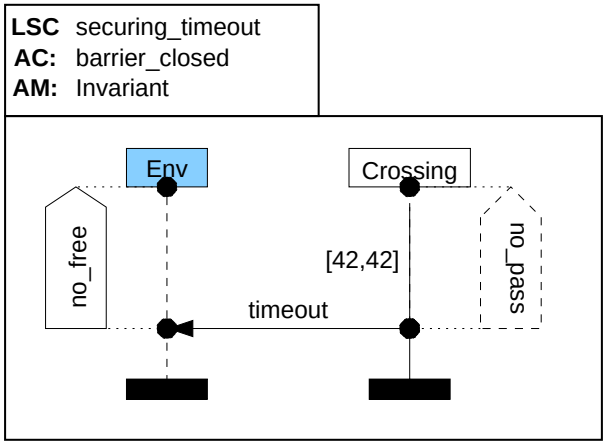


Figure 11.12: Universal LSC for exceeding the maximum barrier closed time

For the crossing activity the existential LSCs and their modularization has already been presented to a large extent in example 10.2 on page 231 in section 10.2.3. The complete set of existential and universal LSCs for the

crossing is found in appendix A.3.1 resp. A.3.2, the used assumption LSCs are collected in appendix A.3.3. In addition to the universal LSCs presented in 10.2.3 two additional LSCs are presented in figures 11.12 and 11.13. The first LSC expresses the requirement that the crossing controller should report an exceeding of the maximum barrier closed time, if the train has not passed or freed the crossing within that time. Note that the timing constraint in the step semantics is two steps larger (42, instead of the MBCT value of 40) due to the additional delay steps (one due to the delay for observing the entering event and one due to delay for observing the timeout event, cf. section 2.2.4).

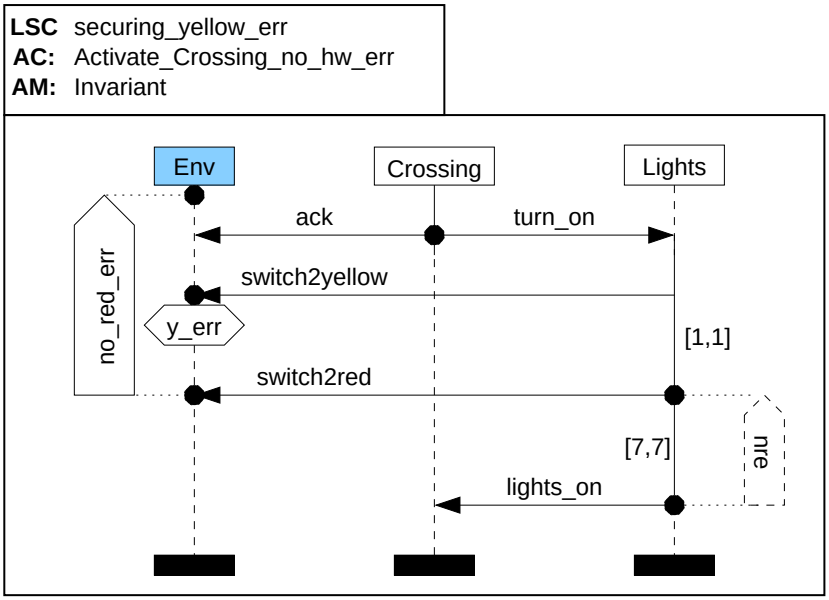


Figure 11.13: Universal LSC for a defect of the yellow light

Figure 11.13 depicts the situation where the yellow light is broken, but the red light is still operational and is switched on immediately after detecting the failure of the yellow light (cf. figure 11.3 on page 245). The red light has to be on for the combined duration required for yellow and red light. Two strengthening assumptions are required for this property, both of which are expressed internally: A successful switching on of the lights entails that the red light does not fail (cf. LSC `securing_lights`; figure A.55 on page 348) and the yellow light should indeed be out of order as expressed by mandatory condition `yerr`.

Property	Commitment LSC	Assumption LSCs	Figure
ECS1	securing		A.32
ECSY1	securing_yellow_err		A.33
ECSR1	securing_red_err		A.34
ECSR2	securing_red_err2		A.36
ECSR3	securing_red_err3		—
ECSR4	securing_red_err4		A.37
NECSR1	securing_red_err_neg		A.35
ECSB1	securing_barr_err		A.38
ECSB2	securing_barr_err2		A.39
ECSF1	securing_free		A.40
ECST1	securing_timeout		A.41
ECST2	securing_timeout2		A.42
ECST3	securing_timeout3		—
ECST4	securing_timeout4		—
UCSL1	securing_lights	correct_activate_ass no_activation_ass	A.55
UCSBI1	securing_barrier_init		A.57
UCSBI2	securing_barrier_init2		A.59
UCSBI3	securing_barrier_init3	prompt_closed_ass	A.61
UCSB1	securing_barrier		A.63
UCSB2	securing_barrier2		A.66
UCSB3	securing_barrier3	prompt_closed_ass	A.69
UCSTO1	securing_timeout		A.72
UCSTO2	securing_timeout2		—
UCSTO3	securing_timeout3		—
UCSTO4	securing_timeout4		A.74
UCYE1	securing_yellow_err	correct_activate_ass	A.76
UCYE2	securing_yellow_err2	correct_activate_ass	A.78
UCYE3	securing_yellow_err3	correct_activate_ass	A.80
UCYE4	securing_yellow_err4	correct_activate_ass	A.82
UCO1	opening		A.43

UCO2	opening	correct_open_ass correct_closed_ass correct_activate_ass functioning_lights_ass	A.49
UCO3	opening2		A.45
UCO4	opening3	prompt_open_ass	A.47
UCO5	opening4	correct_open2_ass correct_closed_ass correct_activate_ass	A.51
UCO6	opening5	prompt_open_ass correct_closed_ass correct_activate_ass	A.53

Table 11.3: Properties for CROSSING

The complete set of LSCs for the crossing is summed up in table 11.3. Again two variants of the existential LSCs for a red light and barrier failure as well as for exceeding the maximum barrier closed time are included. Additionally a negative scenario (NECSR1) is specified in figure A.35, which shows the situation that a failure of the red light has occurred and that the subsequent status request is nevertheless answered by the safe message. Since the crossing can not be secured after a red light failure, this LSC expresses an undesired communication sequence.

The existential LSCs describing the failure of the red light as specified above can not be reused in the superstep semantics (see the following section for explanation), therefore LSCs ECSR3 and ECSR4 have been added. Since the timing annotations used in ECST1 in the step semantics are not directly transferable to the superstep semantics, ECST3 and ECST4 are used to investigate the effect of the two alternatives for specifying superstep timing constraints. ECST3 uses the explicit enumeration approach, which is possible here since the timing constraint specifies an exact bound, and ECST4 queries the freshly introduced superstep timer.

Regarding universal LSCs there were also some adaptations necessary in order to correctly express the desired properties for the superstep model. The initial LSC describing the successful closing of the barriers (UCSBI1) is the first to be adjusted. The LSC using the enumeration approach (UCSBI2) has already been presented in figure 11.5 on page 248 and is specified in

such a way that the effect of an internal assumption is achieved (cf. the explanation accompanying figure 11.5). The counter approach is not applicable in this case, since no internal assumption treatment can be effected (cf. explanation accompanying figure 11.6 on page 249). Therefore the user-specified assumption LSC `prompt_closed_ass` (figure A.95 on page 378) is linked to the original commitment LSC (without the timing interval) yielding property UCSBI3 (figure A.61 on page 353). Since the timing interval specifies an exact bound and requires only one superstep delay, we use the explicit inclusion of the *stable* event in this assumption. The invariant LSC `securing_barrier` (UCSB1) is treated analogously resulting in LSCs UCSB2 for the enumeration approach and UCSB3 using the superstep counter (for the timing intervals in the pre-chart) and assumption `prompt_closed_ass` (for the timing interval in the commitment LSC).

The LSCs dealing with the timeout due to the elapsed maximum barrier closed time (UCSTO1) and considering a failure of the yellow light (UCYE1) both exist also in a version, which omits the timing intervals: UCSTO2 and UCYE2, respectively. UCSTO1 and UCYE1 moreover result in two variations for the superstep semantics as well: UCSTO3 and UCYE3 for the enumeration and UCSTO4 and UCYE4 for the counter approach.

The LSC specifying the return of the crossing to its idle state after a train has passed comes in two versions: one using a pre-chart (UCO1, cf. figure 10.4 on page 232) and one expressing the restrictions of the pre-chart by a set of user-specified assumptions (UCO2). Each version results in two LSCs in the asynchronous simulation semantics, one using explicit enumeration, the other an assumption (`prompt_open_ass`, similar to the one used for UCSBI3, cf. figure A.97 on page 378) for the specification of the timing constraint on the environment. UCO3 (figure A.46 on page 339) and UCO4 (figure A.48 on page 341) thus both come equipped with a pre-chart, UCO5 (figure A.52 on page 345) and UCO6 (figure A.54 on page 347) use the assumptions linked to UCO2 instead; UCO3 and UCO5 use the enumeration approach, whereas UCO4 and UCO6 additionally link the assumption LSC `prompt_open_ass`. Since this assumption covers the behavior specified by assumption LSC `correct_closed_ass`, the latter is substituted by `prompt_open_ass` in UCO6, yielding a total of four assumptions.

11.3 Verification Results

11.3.1 General Considerations

For the verification of existential LSCs the chosen strategy is to produce a witness for the LSC, rather than just reporting true or false as a result. A witness is more useful to the designer than the statement that there exists a run satisfying the LSC, since it can be viewed as a timing diagram or also executed in the STATEMATE simulator, thus allowing to examine the witness and compare it to the original existential LSC. The goal of verifying existential LSCs is consequently falsification, so that for this use case both the reachability-based approach and bounded model checking can be employed. For universal LSCs only the standard model checking technique can be used.

If the existential verification of an LSC fails, no indication is given by the model checker as to what the problem is, which is an inherent problem, since the counter example is the entire model. In this case the strategy is to manually shorten the LSC in order to find out which part of the LSC is not satisfiable. We expect this situation to be a rare case, however.

Note that the strategy for the witness generation differs from the formal semantics of existential satisfaction (cf. definition 9.1 on page 215). The witness is generated for one legal run, if one exists: the complete traversal of the LSC, whereas the formal semantics also allows runs, which exit due to violated possible conditions or local invariants and which get stuck in a cold cut. We feel that a complete traversal is much more useful and informative and therefore only consider an LSC existentially verified, if all communications have been observed.

For the verification of the LSCs specified for the train control system we have in some cases used an optimization called *data abstraction* offered by the STVE. In real life models control and data are not clearly separated as the radio-based crossing control application shows. Since the data part of a model typically heavily contributes to the overall complexity, it is desirable to exclude it from the model, if possible. Data abstraction allows to selectively remove data parts from the model, constructing an abstract model. Note that not the entire variable is removed by this approach, but some information about it is retained in the abstract model. There are several strategies, which govern how much and which information is kept. One possibility e.g. is to convert a data item into an input, which simplifies calculations involving this

data item while at the same time loosing accuracy. The selection of which variables should be abstracted is made by the user.

Data abstraction is an over-approximating technique, i.e. in the abstract model more behavior is allowed than in the original model. This entails that a property, which is proved on the abstract model, is true in the original one as well. A property, which is violated on the abstract model, on the other hand may still be satisfied in the original one, since the violation might be due to the added behavior. Therefore data abstraction can not be employed for falsification, i.e. existential verification in our context.
For more information on abstraction techniques see [BBD+99] and [Bie03].

In the following the model checker run times for the LSCs listed in tables 11.1, 11.2 and 11.3 are presented. All results have been produced on a SUN Blade 1000 equipped with a 750 Mhz UltraSparc processors and 2.5 GB RAM running Solaris 8. The VIS version used is 1.3. The run times are pure processing time used by the model checker, compilation times for the generation of the FSM and the formula are not included. The run times are measured in seconds and a timeout of five hours is enforced, after which the model check run is aborted. The used commitment and user-specified assumption LSCs and the corresponding symbolic automata are collected in appendix A. All universal LSCs have additionally been checked to ensure that they are activated at least once.

The results are considered separately for existential and universal LSCs in sections 11.3.2 and 11.3.3, which are each split into subsections for the assessment of LSCs for the step and superstep version of the train control system model.
Note that for all model check runs using the standard fix-point algorithm the reachability computation on the FSM level has been activated, because without it the proofs took decidedly more time.

11.3.2 Existential Verification Results

Synchronous Semantics

The verification times for the top level (`SYSTEM`) are shown in tables 11.4 on the facing page for the weak interpretation and 11.5 on page 266 for the strict interpretation. Both tables present the run times in seconds for the three different technologies: standard model checking (column headed by *Time mc*), reachability-based model checking (column *Time rb*) and bounded

model checking (column *Time bmc*). Additionally the length of the witness measured in number of steps, if generated, is listed in the last column. A time value in a field indicates that the corresponding model check technique produced a witness in the time given, fields marked by the word *timeout* indicate that the verification time exceeded the five hour time limit. If no witness exists, the time value is marked by †, e.g. for negative scenarios or unsuccessfully verified positive scenarios. Note that each technique generates the shortest error path possible, so that the trace length is identical for all three verification strategies.

Both tables for `SYSTEM` show that the model is quite complex, since only two proofs yield a witness within the allotted time. The complexity is mainly caused by the large integer data items used for the speed and location determination in the train and the operations on these. Every step involves the computation of new acceleration, speed and position values for the train, the calculation of the current maximum speed and a comparison between maximal and actual speed value, with each computation operating on at least two integer variables. This computational complexity in conjunction with the model size (see table 11.21 on page 280), results in the very low number of witnesses on the top level.

Property	Figure	Time mc	Time rb	Time bmc	Trace length
ESSA1	A.1	timeout	timeout	timeout	—
ESSAY1	A.2	timeout	timeout	timeout	—
ESSAR1	A.3	timeout	timeout	1417.9 s	23
ESSAR2	A.4	timeout	timeout	973.9 s	23
ESSAB1	A.5	timeout	timeout	timeout	—
ESSAB2	A.6	timeout	timeout	timeout	—
ESSAF1	A.7	timeout	timeout	timeout	—
ESSAT1	A.8	timeout	timeout	timeout	—

Table 11.4: Verification run times for existential LSCs on `SYSTEM` (weak interpretation, step semantics)

The two LSCs, for which a witness was generated, are consequently the least complex ones in terms of number of messages: ESSAR1 and ESSAR2. It seems that the bounded model checker is more efficient when the property to be checked is rather restrictive. This is indicated on the one hand by the fact that in both interpretations the witness for the LSC with the local in-

Property	Figure	Time mc	Time rb	Time bmc	Trace length
ESSA1	A.1	timeout	timeout	timeout	—
ESSAY1	A.2	timeout	timeout	timeout	—
ESSAR1	A.3	timeout	timeout	855.4 s	23
ESSAR2	A.4	timeout	timeout	728.2 s	23
ESSAB1	A.5	timeout	timeout	timeout	—
ESSAB2	A.6	timeout	timeout	timeout	—
ESSAF1	A.7	timeout	timeout	timeout	—
ESSAT1	A.8	timeout	timeout	timeout	—

Table 11.5: Verification run times for existential LSCs on `SYSTEM` (strict interpretation, step semantics)

variant, which explicitly prohibits the sending of the safe message (ESSAR2), is generated faster than for the LSC without this invariant. On the other hand the run times for the strict interpretation are lower than for the weak interpretation.

This supposition is at first glance disproven by the results on the train (see tables 11.6 and 11.7 on the facing page), where the run times for the strict interpretation increase for the bounded approach in all three cases. The worse run times are at least in part due to the longer error traces in the strict interpretation, however. The shorter witnesses for the weak interpretation result from the fact, that the bounded model checker is able to take 'shortcuts' by generating input events simultaneously to the outputs which should provoke these inputs.

Property	Figure	Time mc	Time rb	Time bmc	Trace length
ETSC1	A.27	timeout	timeout	147.2 s	16
ETSCF1	A.28	timeout	timeout	389.4 s	18
ETFC1	A.29	timeout	timeout	130.8 s	19

Table 11.6: Verification run times for existential LSCs on `TRAIN` (weak interpretation, step semantics)

For the train again only the bounded model checking technique produced any witness at all. In contrast to the `SYSTEM` level all existential LSCs specified could be successfully verified here due to both the reduced model and the smaller LSCs. The run time for LSC ETSCF1 in both interpretations is

Property	Figure	Time mc	Time rb	Time bmc	Trace length
ETSC1	A.27	timeout	timeout	283.0 s	18
ETSCF1	A.28	timeout	timeout	1054.2 s	19
ETFC1	A.29	timeout	timeout	344.5 s	21

Table 11.7: Verification run times for existential LSCs on `TRAIN` (strict interpretation, step semantics)

significantly larger than for the other two LSCs, because ETSCF1 requires the train to stop in front of the crossing, which entails more precise actual and nominal speed computations.

Property	Figure	Time mc	Time rb	Time bmc	Trace length
ECS1	A.32	750.9 s	15.4 s	228.9 s	31
ECSY1	A.33	7547.3 s	13.7 s	248.1 s	31
ECSR1	A.34	30.4 s	7.0 s	1.4 s	14
ECSR2	A.36	45.7 s	8.0 s	1.5 s	14
NECSR1	A.35	85.6 s	11.6 s	—	34
ECSB1	A.38	1379.5 s	15.4 s	367.4 s	31
ECSB2	A.39	1092.5 s	12.0 s	252.3 s	31
ECSF1	A.40	488.4 s	13.4 s	190.3 s	29
ECST1	A.41	13516.3 s	130.5 s	timeout	71
ECST2	A.42	8453.2 s	40.0 s	timeout	71

Table 11.8: Verification run times for existential LSCs on `CROSSING` (weak interpretation, step semantics)

Tables 11.8 and 11.9 on the following page show the results for the existential verification of the LSCs specified for the `CROSSING` activity. The reachability-based strategy is much faster in all cases than the normal model checking procedure. For the weak interpretation and the standard model check procedure the proof for ECSY1 is significantly more complex than for the almost identical LSC ECS1. The effect of the additional condition `yellow_err` seems to be very detrimental. The LSC describing a failure of the red light (ECSR1) is considerably smaller than ECS1 and its proof consequently has a decidedly lower model check time.

Property	Figure	Time mc	Time rb	Time bmc	Trace length
ECS1	A.32	776.4 s	16.4 s	174.2 s	31
ECSY1	A.33	47.8 s	22.8 s	188.2 s	31
ECSR1	A.34	21.3 s	5.5 s	1.3 s	14
ECSR2	A.36	29.2 s	7.9 s	1.3 s	14
NECSR1	A.35	7.8 s†	8.6 s†	—	—
ECSB1	A.38	453.6 s	17.8 s	84.5 s	31
ECSB2	A.39	111.3 s	15.7 s	91.9 s	31
ECSF1	A.40	75.1 s	17.1 s	104.0 s	29
ECST1	A.41	10837.2 s	163.9 s	13442.9 s	71
ECST2	A.42	466.0 s	43.7 s	10809.4 s	71

Table 11.9: Verification run times for existential LSCs on `CROSSING` (strict interpretation, step semantics)

The effect of the addition of the local invariant forbidding the sending of the safe message in variations ECSR2 and ECSB2 is not uniform: For ECSR2 it causes a run time increase, whereas the verification time for ECSB2 decreases. The impact of the local invariant is seemingly dependent on the LSC complexity: for a lower original complexity (ECSR1) the verification time increases, while it decreases for high original complexity (ECSB1). This effect is visible most pronouncedly for the standard model check procedure and to a lesser degree for the other techniques.

The LSC describing the opening of the crossing due to receiving the free message from the train (ECSF1) is slightly simpler than ECS1 and thus requires only about two thirds of the time necessary to prove ECS1. The proof for the LSC concerning the detection of a timeout at the crossing (ECST1) is very complex because of the large timers both in the model and in the LSC, so that here the largest verification time is observed. LSC ECST2, which does not contain the timer measuring the maximum barrier closed time, performs better. The timeout event within the model still causes a substantial verification time, however.

In the reachability-based strategy the model check times in the weak interpretation are mostly of the same order of magnitude. Interestingly, the detrimental effect of the local invariant in LSC ECSY1 disappears, so that the run times for the proofs for ECS1 and ECSY1 are almost identical. The other trends observed for the standard model check procedure remain in

effect. ECSR1 is still proven in less time than any of the other LSCs, even though the advantage is not as great, and ECST1 remains the most complex proof task. The omission of the timer again has a beneficial effect compared to ECSTO1 without reaching the low run times of the other LSCs.

The bounded model check approach performs rather poorly compared to the reachability-based technique. Only for the smallest LSCs (ECSR1 and ECSR2) it produces a small gain, for all other proof tasks the run times are much larger and the proofs for the LSCs involving the observing of the maximum barrier closed time do not even yield a result within the allotted time. Except for ECST1 and ECST2 the run times are still significantly better than for the standard model check procedure, however. The key point for the behavior of the bounded model check approach is the length of the witness: Short witnesses are easily found by bounded model checking, whereas long ones require an increased amount of time, since more iterations of bounded-FSM generation and SAT-checker runs are necessary. This effect can be observed for all verification runs for existential LSCs in the step semantics. For the crossing the fastest times for bounded model checking are achieved for LSCs resulting in the shortest witnesses and the worst results are seen for LSCs producing the longest error paths, e.g. ECST1. For the train and the complete system the witnesses found by bounded model checking are relatively short, compared to the trace lengths for most LSCs on the crossing level.

The overall effect of the strict interpretation (table 11.9) on the normal model checking procedure is beneficent, all run times improve save the one for ECS1, which slightly increases. For this technique the more restrictive formula limits the search space for the backward-oriented model check algorithm, so that it is kept more closely to the desired path, i.e. the witness. This effect is visible most pronouncedly for the proof of LSC ECSY1, whose run time is reduced by a factor of almost 160.

For the reachability-based method the change in interpretation is almost unnoticeable in the run times, since the forward-oriented technique does not have as much variability in finding the witness as the backward strategy, because it starts forward from the initial state. The verification times for the reachability-based method consequently almost uniformly increase slightly due to the more complex formula.

The effect of the strict interpretation for the bounded model checking approach is again beneficial, since each message is explicitly contained in all transitions of the automaton and therefore also in the resulting formula, which facilitates the task of the SAT-checker of finding a proposition valuation satisfying the formula. The verification times for almost all LSCs decrease consequently and even the proofs for ECST1 and ECST2 yield a result within the time limit.

The LSC specifying the negative scenario (NECSR1, see figure A.35 on page 328) demonstrates that for this use case the strict interpretation is compulsory, because an unexpected witness is found in the weak interpretation (cf. 11.8 on page 267). The generated witness requires two trains passing the crossing: the first one advancing the LSC to the point where the failure of the red light occurs and then stopping in front of the crossing, the second one arriving after the red light has been repaired and properly receiving the safe message in reply to its status request. The negative scenario checked, however, refers to a single securing procedure, i.e. one train, which is only enforced by the strict interpretation. The verification of NECSR1 in the strict interpretation is successful as table 11.9 on page 268 shows: no witness is found, i.e. it is impossible that a crossing sends the safe message, even though the red light has failed and the crossing is not secured. Note that bounded model checking can not be used for existential verification of negative scenarios, because the expectation here is that no witness exists, i.e. the basic use case is not falsification.

Asynchronous Semantics

Property	Figure	Time mc	Time rb	Time bmc	Trace length
ETSC1	A.27	timeout	timeout	9641.1 s	31
ETSCF1	A.28	timeout	timeout	10532.9 s	37
ETFC1	A.29	timeout	timeout	timeout	—

Table 11.10: Verification run times for existential LSCs on `TRAIN` (weak interpretation, superstep semantics)

For existential verification in the superstep semantics a general increase of the model checker run times is observed; the greater complexity is also reflected by the model sizes (cf. tables 11.21 - 11.26. On the `SYSTEM` level

Property	Figure	Time mc	Time rb	Time bmc	Trace length
ETSC1	A.27	timeout	timeout	2082.6 s	33
ETSCF1	A.28	timeout	timeout	timeout	—
ETFC1	A.29	timeout	timeout	3738.9 s	37

Table 11.11: Verification run times for existential LSCs on `TRAIN` (strict interpretation, superstep semantics)

all attempts at existential verification timed out, so that no results table is given for this level. On the `TRAIN` level again only the bounded model checking approach produced any results within the allotted time as tables 11.10 and 11.11 show. The model check times have increased significantly — even timing out on two occasions — due to the longer witnesses. As has already been apparent for the step semantics the strict interpretation is generally beneficial for bounded model checking, even though ETSCF1 shows a contrary effect.

Property	Figure	Time mc	Time rb	Time bmc	Trace length
ECS1	A.32	timeout	39.9 s	2539.2 s	45
ECSY1	A.33	timeout	42.7 s	2550.1 s	45
ECSR1	A.34	—	60.2 s†	—	—
ECSR3	—	timeout	13.2 s	22.2 s	21
ECSR4	A.37	timeout	16.6 s	17.7 s	21
ECSB1	A.38	timeout	43.8 s	2732.9 s	48
ECSB2	A.39	timeout	49.2 s	2874.1 s	48
ECSF1	A.40	timeout	35.1 s	1238.3 s	41
ECST2	A.42	timeout	552.8 s	timeout	124
ECST3	—	timeout	4174.4 s	timeout	124
ECST4	—	timeout	timeout	timeout	—

Table 11.12: Verification run times for existential LSCs on `CROSSING` (weak interpretation, superstep semantics)

The results for the `CROSSING` level mostly retain the relations between the proofs, which have been observed in the step semantics, as tables 11.12 and 11.13 demonstrate. The standard model check procedure is outperformed by the other two approaches in all cases, not producing any result within the time limit. The best performance is once more shown by the reachability-

Property	Figure	Time mc	Time rb	Time bmc	Trace length
ECS1	A.32	timeout	54.9 s	1999.5 s	45
ECSY1	A.33	timeout	42.1 s	1835.0 s	45
ECSR3	—	timeout	15.6 s	12.7 s	21
ECSR4	A.37	timeout	15.8 s	12.8 s	21
NECSR1	A.35	timeout	54.6 s†	—	—
ECSB1	A.38	timeout	42.0 s	1620.9 s	48
ECSB2	A.39	timeout	47.7 s	1559.6 s	48
ECSF1	A.40	timeout	35.5 s	1325.4 s	41
ECST2	A.42	timeout	427.1 s	timeout	124
ECST3	—	timeout	1314.1 s	timeout	124
ECST4	—	timeout	6439.2 s	timeout	124

Table 11.13: Verification run times for existential LSCs on `CROSSING` (strict interpretation, superstep semantics)

based strategy, which always produced a result, often within one minute. The bounded model checker can only compete with this for short witnesses (ECSR3 and ECSR4).

Note that the LSCs ECSR1 and ECSR2 could not be reused from the step semantics without change, even though they contain no timing annotations. As the result for ECSR1 in table 11.12 exemplifies, no witness exists for this LSC (the same is true for ECSR2). The problem is that the message switching on the red light can never be observed simultaneously to a red light failure in the superstep semantics. Since the failure indication (`RED_ERR`, cf. figure 2.12 on page 40) is an input, it has to be asserted at the beginning of a superstep. As soon as this error is indicated, however, the lights controller returns to its idle state without trying to switch on the red light. In the corresponding LSCs the condition `red_err` therefore must not be placed into a simultaneous region with message `switch2red` resulting in LSCs ECSR3 and ECSR4 (figure A.37 on page 330). The negative scenario NECSR1 is only checked in the strict interpretation here, since the weak one is inadequate for this use case as explained above.

As in the synchronous semantics the execution times of all existential LSCs without timing information are within the same order of magnitude for the reachability-based method. Only ECST2, which indirectly refers to the timeout event in Statechart `CROSSING_CTRL` (cf. figure 2.11 on page 38)

sticks out by requiring a significantly greater amount of time to be verified. The verification runs for the two LSCs containing superstep timing annotations (ECST3 and ECST4) require significantly more time than the untimed variant (ECST2), with the enumeration approach performing better, both in the strict and weak interpretation Only the reachability-based strategy produced any witness at all and only failed to do so for the counter LSC (ECST4) in the weak interpretation. The bounded model check procedure did not produce a result for any of the timeout LSC variations due to the length of the witness. The witnesses for both ECST3 and ECST4 are generated much faster in the strict interpretation.

11.3.3 Universal Verification Results

Synchronous Semantics

		weak		strict	
Property	Figure	Result	Time (abs)	Result	Time (abs)
USACS1	A.10	true	131.5 s	true	3607.2 s
USACS2	A.12	true	386.1 s	true	79.5 s
USAS1	A.14	true	41.4 s	true	44.0 s
USAST1	A.16	—	timeout	—	timeout
USAF1	A.18	true	71.4 s	true	138.2 s

Table 11.14: Verification run times for universal LSCs on `SYSTEM` (step semantics)

Table 11.14 shows the verification times and results for the universal LSCs of the top level (Activity Chart `SYSTEM`), which have been introduced in section 11.2. Both the weak and strict interpretation have been considered. All proofs use the data abstraction capabilities of the STVE abstracting from the data items, which primarily cause the complexity (`ODATA` and `NOMINAL_SPEED`). These data items are abstracted for all proof tasks, except USAST1 (see below). Without data abstraction no results are produced within the time limit on both the system and train level.

Using data abstraction the model size is reduced significantly (cf. table 11.21 on page 280) and LSCs USAS1 and USAF1 are proved fairly fast. The proof for USAF1 takes more time due to the additional complexity

caused by the timing annotation and the timer within the train, which governs the generation of the free message. The proof for the communication setup (USACS1) takes longer because of the more complicated automaton (cf. figure A.11 on page 307). The strict interpretation has a very detrimental impact on this property provoking a twenty times greater verification time. Interestingly, this is reversed when slightly altering the LSC as USACS2 (shown in figure A.12 on page 308) demonstrates: Here the local invariant covers one more message and consequently one more state and transition in the resulting automaton (figure A.13). The verification time for USACS2 compared to the one for USACS1 is almost tripled in the weak interpretation, but 45 times lower in the strict.

USAST1 proves to be too complex. The problem in this case is that the data abstraction used in the other proofs can not be applied here. For the other proof tasks those data items have been abstracted, which deal with the train's speed and position. This is impossible for USAST1, because the exact representation of this information is vital for the verification of this property. If we want to guarantee that in case of an error the train stops before reaching the crossing, its position and speed must be known accurately.

		weak		strict	
Property	Figure	Result	Time (abs)	Result	Time (abs)
UTFC1	A.30	true	2.6	true	2.1

Table 11.15: Verification run times for universal LSCs on `TRAIN` (step semantics)

Table 11.15 shows the result for the universal LSC specified for the train, which is proved very quickly. Due to the small model size (see table 11.22 on page 280) there is only a minute difference between the times for weak and strict interpretation. Again data abstraction (on `ODATA` and `NOMINAL_SPEED`) has been employed.

The universal LSCs specified for the crossing are all verified successfully as table 11.16 shows. Note that at this level no data abstraction has been necessary. The model checker run times are all within the same general range. For the weak interpretation the run time for UCYE1 is significantly greater than for UCSL1, which is identical, except for the possible condition `yellow_err` added in UCYE1. This increase is only marginally influenced by the timing intervals as the results for UCYE2 show, where the timing

Property	Figure	weak		strict	
		Result	Time	Result	Time
UCSL1	A.55	true	15.3 s	true	12.4 s
UCSBI1	A.57	true	22.6 s	true	9.4 s
UCSB1	A.63	true	30.6 s	true	21.4 s
UCYE1	A.76	true	42.2 s	true	47.0 s
UCYE2	A.78	true	36.7 s	true	47.4 s
UCO1	A.43	true	25.0 s	true	58.7 s
UCO2	A.49	true	59.2 s	true	54.0 s
UCSTO1	A.72	true	17.3 s	true	16.1 s
UCSTO2	—	true	8.2 s	true	7.9 s

Table 11.16: Verification run times for universal LSCs on `CROSSING` (step semantics)

intervals have been omitted. The verification time for UCSTO1 is fairly small considering the fact that both the LSC and the model contain a large timer. Omitting the timer in the LSC (USCTO2) has a beneficial effect as already observed in the existential verification. The proof for UCO2, which expresses the requirements of the pre-chart of UCO1, by four user-specified assumptions performs clearly worse than the one for UCO1 in the weak interpretation. This is caused by the added complexity due to having to observe several assumptions, i.e. restrictions on the runs the system may take.

The effect of the strict interpretation is not uniform as the last column in table 11.16 shows. There are slight performance gains for some LSCs (UCSL1, UCSBI1, UCSB1, UCO2, UCSTO1) and losses for others (UCYE1, UCYE2, UCO1), although the order of magnitude of the verification times remains unchanged.

Asynchronous Semantics

As for existential LSCs the run times generally increase for universal LSCs in the superstep semantics as well in this model. Table 11.17 shows the results for the LSCs specified on the `SYSTEM` level. As in the step semantics all proofs at this level employed data abstraction. In addition to the data items already abstracted in the step model also all non-essential timers resulting

		weak		strict	
Property	Figure	Result	Time (abs)	Result	Time (abs)
USACS1	A.10	true	7136.6 s	true	8024.4 s
USACS2	A.12	true	1840.9 s	true	1507.7 s
USAS1	A.14	—	timeout	—	timeout
USAST1	A.16	—	timeout	—	timeout
USAF2	—	true	1559.8 s	true	2296.2 s
USAF3	A.20	true	14126.1 s	true	11812.9 s
USAF4	A.22	—	timeout	—	timeout

Table 11.17: Verification run times for universal LSCs on `SYSTEM` (superstep semantics)

from timeout expressions in the Statecharts have been abstracted here, since these have shown to exact a high price in terms of run time: no results were obtained without abstracting these timers.

The LSCs USACS1, USACS2, USAS1 and USAST1 are identical to the ones checked in the synchronous semantics, since no timing annotations are used. The difference between USACS1 and USACS2 is again observable, although the relation between the run times is not the same as in the step semantics (cf. table 11.14 on page 273). Here USACS2 clearly performs better in both interpretations. The trend of USACS2 benefitting from the strict interpretation, while USACS1 is affected detrimentally, which has been observed in the synchronous semantics, is still visible, but not as pronouncedly as before.

The verification runs for USAS1 did not yield a successful result within the time limit, which is rather unexpected, since the proof tasks for this LSC had the lowest run times in the synchronous semantics. The reason, in combination with the generally increased complexity, is that for USAS1 more timers must be retained in the model than e.g. for USACS1 and USACS2. USAST1 again is too complex to be verified successfully.

Comparing both approaches for expressing superstep timing constraints for the LSC specifying the sending of the free message (USAF2 for the enumeration and USAF3 for the counter approach) shows that the counter approach is clearly inferior to superstep enumeration in this case. In both interpretations the verification time is significantly larger. The enumeration approach also seems fairly efficient compared to USACS2. Note, however,

that USAF2 and USAF3 use a exact bound of 15 instead of an upper bound of 40 as in the LSC for step semantics (USAF1); cf. section 11.2. USAF4 specifies the correct upper bound using the counter approach[3], but can not be proved successfully due to the large counter.

		weak		strict	
Property	Figure	Result	Time (abs)	Result	Time (abs)
UTFC1	A.30	true	5.3 s	true	5.5 s

Table 11.18: Verification run times for universal LSCs on `TRAIN` (superstep semantics)

Property UTCF1 on the `TRAIN` level is unaffected by the added complexity of the asynchronous semantics (see table 11.18), since the model is still small enough. For UTCF1 the same variables have been abstracted as in the step semantics.

The experimental results for the universal LSCs specified at the crossing level are shown in tables 11.19 (weak) and 11.20 (strict). Due to the detrimental effect of the timers, which are introduced by timeout events, these have been abstracted when possible. Tables 11.19 and 11.20 show the results for both the original (column *Time*) and abstract model (column *Time (abs)*). The verification times are always worse in the original model, often dramatically so. We therefore mainly consider the results on the abstract model below. For most proofs the trend shown by on the original model corresponds to those on the abstract one.

The verification times are still fairly low for most proofs thanks to the data abstraction. As already observed in the step semantics, the condition `y_err`, which is the only difference between UCSL1 and UCYE2, causes a run time increase in both interpretations, more pronouncedly so in the strict. Regarding the initial LSCs for the description of the correct closing of the barriers (UCSBI2 for the enumeration and UCSBI3 for the counter approach) the model checker run times are better for the enumeration approach in both interpretations.

For the invariant version of the protocol for closing the barriers (UCSB2, UCSB3) the enumeration technique performs significantly better in both interpretations as well. The situation is different for the LSCs describing the

[3]Recall that upper bounds can not be expressed by enumeration.

		weak		
Property	Figure	Result	Time	Time (abs)
UCSL1	A.55	true	4487.8 s	14.0 s
UCSBI2	A.59	true	111.4 s	10.8 s
UCSBI3	A.61	true	1081.2 s	20.6 s
UCSB2	A.66	true	97.3 s	21.5 s
UCSB3	A.69	true	timeout	89.0 s
UCYE2	A.78	true	1406.1 s	26.5 s
UCYE3	A.80	true	timeout	1640.0 s
UCYE4	A.82	true	timeout	179.6 s
UCO3	A.45	true	119.3 s	12.1 s
UCO4	A.47	true	2096.1 s	14.6 s
UCO5	A.51	true	timeout	6178.1 s
UCO6	A.53	true	timeout	73.2 s
UCSTO2	—	true	59.0 s	26.6 s
UCSTO3	—	true	2300.7 s	625.5 s
UCSTO4	A.74	true	timeout	240.1 s

Table 11.19: Verification run times for universal LSCs on `CROSSING` (superstep semantics)

successful switching on of the traffic lights, even though a failure of the yellow light has occurred (UCYE3 for the enumeration approach and UCYE4 foe the superstep counter). The proofs for both LSCs perform decidedly worse than either the untimed variant (UCYE2) or UCSL1, which contains neither time constraints nor the condition `y_err`, but is otherwise identical. The counter approach is significantly faster than explicit enumeration in the weak interpretation, but performs worse, if the LSC is interpreted strictly.

Yet another behavior is observed for the LSCs specifying the generation of the timeout signal sent to the operations center, UCSTO3 using enumeration and UCSTO4 using the superstep counter. Here the counter approach shows lower run times in both interpretations, with almost no difference between strict and weak for both proofs. The proof tasks for both LSCs need clearly more time than the untimed LSC UCSTO2, however.

The effect on the LSCs specifying the correct opening of the crossing once the train has passed, which are using the pre-chart, UCO3 (enumeration) and UCO4 (counter), are almost identical, only in the strict interpretation does

		strict		
Property	Figure	Result	Time	Time (abs)
UCSL1	A.55	true	7144.0 s	17.8 s
UCSBI2	A.59	true	70.7 s	7.7 s
UCSBI3	A.61	true	7114.6 s	13.7 s
UCSB2	A.66	true	2302.6 s	19.7 s
UCSB3	A.69	true	timeout	119.7 s
UCYE2	A.78	true	2186.9 s	58.2 s
UCYE3	A.80	true	timeout	751.1 s
UCYE4	A.82	true	timeout	1395.8 s
UCO3	A.45	true	144.2 s	13.7 s
UCO4	A.47	true	1329.0 s	28.5 s
UCO5	A.51	true	timeout	1304.0 s
UCO6	A.53	true	timeout	49.5 s
UCSTO2	—	true	58.2 s	26.4 s
UCSTO3	—	true	timeout	608.0 s
UCSTO4	A.74	true	17261.5 s	243.1 s

Table 11.20: Verification run times for universal LSCs on `CROSSING` (superstep semantics)

UCO4 take slightly longer. The situation for UCO5 and UCO6 is different: Here the superstep enumeration within the commitment results in a tremendous increase of the verification time compared to UCO3, whereas the exchanged assumption causes only a mild increase.

The following tables sum up the proofs including LSC names, used external assumptions and model sizes for each proof. There are separate tables for synchronous and asynchronous semantics. Note that the model sizes for proofs of the universal LSCs for the system and train level all give the size of the abstract model.

11.4 Assessment of LSCs

In this section we draw conclusions regarding the experimental results presented in the preceding section, the different choices (weak vs. strict, enumeration vs. counter approach, etc.) and the usefulness of the LSC language in

Property	Commitment LSC	Assumption LSCs	Model size
ESSA1	securing_all		216/35
ESSAY1	securing_all_yerr		200/35
ESSAR1	securing_all_red_err		214/33
ESSAR2	securing_all_red_err2		214/33
ESSAB1	securing_all_barr_err		218/34
ESSAB2	securing_all_barr_err2		218/34
ESSAF1	securing_all_free		201/35
ESSAT1	securing_all_timeout		217/34
ESSAT2	securing_all_timeout2		217/34
USACS1	securing_all_comm_setup	correct_activation_point_ass no_activation_point_ass	96/27
USACS2	securing_all_comm_setup2	correct_activation_point_ass no_activation_point_ass	96/27
USAS1	securing_all_safe		95/27
USAST1	securing_all_stop		188/40
USAF1	securing_all_free		97/28

Table 11.21: List of properties for activity `SYSTEM` (step semantics)

Property	Commitment LSC	Assumption LSCs	Model size
ETSC1	securing_crossing		119/25
ETSCF1	securing_crossing_fail		119/25
ETFC1	free_crossing		120/25
UTFC1	free_crossing		26/13

Table 11.22: List of properties for activity `TRAIN` (step semantics)

general. Before presenting the detailed conclusions, we give a brief overview over the major points:

- Formal verification using LSCs is feasible.
- The reachability-based strategy shall be used for the verification of existential LSCs.
- The strict interpretation shall be used. The corresponding preconditions must be observed.

Property	Commitment LSC	Assumption LSCs	Model size
ECS1	securing		84/15
ECSY1	securing_yellow_err		84/15
ECSR1	securing_red_err		80/13
ECSR2	securing_red_err2		81/13
NECSR1	securing_red_err_neg		81/13
ECSB1	securing_barr_err		84/14
ECSB2	securing_barr_err2		85/14
ECSF1	securing_free		83/14
ECST1	securing_timeout		84/14
ECST2	securing_timeout2		84/14
UCSL1	securing_lights	correct_activate_ass no_activation_ass	82/12
UCSBI1	securing_barrier_init		80/13
UCSB1	securing_barrier		81/13
UCYE1	securing_yellow_err	correct_activate_ass	81/12
UCYE2	securing_yellow_err2	correct_activate_ass	81/12
UCO1	opening		80/13
UCO2	opening	correct_open_ass correct_closed_ass correct_activate_ass functioning_lights_ass	81/13
UCSTO1	securing_timeout		79/12
UCSTO2	securing_timeout2		79/12

Table 11.23: List of properties for `CROSSING` (step semantics)

- Internal and user-specified assumptions are needed. Extracted assumptions are not needed.

- When possible the enumeration approach shall be used for specification of timing constraints in the asynchronous semantics.

- Data abstraction is useful for the reduction of complexity, but has to be used with care.

- Timing constraints increase complexity.

Property	Commitment LSC	Assumption LSCs	Model size
ESSA1	securing_all		286/43
ESSAY1	securing_all_yerr		286/43
ESSAR3	securing_all_red_err3		283/41
ESSAR4	securing_all_red_err4		283/41
ESSAB1	securing_all_barr_err		287/44
ESSAB2	securing_all_barr_err2		287/44
ESSAF1	securing_all_free		287/43
ESSAT2	securing_all_timeout2		286/39
USACS1	securing_all_comm_setup	correct_activation_point_ass no_activation_point_ass	77/42
USACS2	securing_all_comm_setup2	correct_activation_point_ass no_activation_point_ass	77/42
USAS1	securing_all_safe		148/39
USAST1	securing_all_stop		252/48
USAF2	securing_all_free2		118/36
USAF3	securing_all_free3		125/40
USAF4	securing_all_free4		129/42

Table 11.24: List of properties for activity `SYSTEM` (superstep semantics)

Property	Commitment LSC	Assumption LSCs	Model size
ETSC1	securing_crossing		163/25
ETSCF1	securing_crossing_fail		163/25
ETFC1	free_crossing		164/26
UTFC1	free_crossing		46/14

Table 11.25: List of properties for activity `TRAIN` (superstep semantics)

Existential Verification

Of the three strategies employed in the verification of existential LSCs the reachability-based one is the most promising method to be employed for existential verification of LSCs. In almost all cases it out-performs the other two approaches. Only when the model is very complex and the witness is relatively short the bounded model checker yields better results. The reachability-based strategy should thus be used as a default for this use case. Since bounded model checking performs well for short to medium length witnesses, the backup strategy is to use this technique, if the model is too

Property	Commitment LSC	Assumption LSCs	M. size	M. size (abs)
ECS1	securing		109/18	
ECSY1	securing_yellow_err		109/18	
ECSR3	securing_red_err3		104/16	
ECSR4	securing_red_err4		105/16	
NECSR1	securing_red_err_neg		105/16	
ECSB1	securing_barr_err		108/19	
ECSB2	securing_barr_err2		109/19	
ECSF1	securing_free		108/19	
ECST2	securing_timeout2		109/19	
ECST3	securing_timeout3		110/17	
ECST4	securing_timeout4		121/20	
UCSL1	securing_lights	correct_activate_ass no_activation_ass	105/12	76/14
UCSBI2	securing_barrier_init2		106/13	57/28
UCSBI3	securing_barrier_init3	prompt_closed_ass	106/13	57/28
UCSB2	securing_barrier2		106/13	74/28
UCSB3	securing_barrier3	prompt_closed_ass	110/15	78/30
UCYE2	securing_yellow_err2	correct_activate_ass	104/12	71/16
UCYE3	securing_yellow_err3	correct_activate_ass	106/12	75/13
UCYE4	securing_yellow_err4	correct_activate_ass	112/16	83/13
UCO3	opening2		105/13	60/28
UCO4	opening3	prompt_open_ass	105/13	60/28
UCO5	opening4	correct_open2_ass correct_closed_ass correct_activate_ass	106/13	61/28
UCO6	opening5	prompt_open_ass correct_closed_ass correct_activate_ass	105/13	61/28
UCSTO2	securing_timeout2		103/12	74/15
UCSTO3	securing_timeout3		104/12	75/25
UCSTO4	securing_timeout4		115/18	86/31

Table 11.26: List of properties for `CROSSING` (superstep semantics)

complex *and* the expected length of the witness is short.

Interpretation

Our expectation regarding the effect of the strict compared to the weak interpretation was that the larger formula resulting from the former would lead to noticeably increased model checker run times. The experimental results show that this expectation is fulfilled for only a small number of properties, e.g. USACS2 in the step semantics or UCYE4. In the majority of cases the verification times increased only slightly or even decreased. A possible reason for this effect could be that the inclusion of all message labels in every transition of the symbolic automaton constitutes almost an invariant property resulting in a formula, which is easier to check, since all contained propositions are similar. This supposition is supported by LSC USACS2: the extension of the local invariant to cover one more message in comparison to USACS1 yields a more uniform automaton (cf. figures A.11 on page 307 and A.15 on page 309) and a reduced run time in the strict interpretation.

Our decision for a default interpretation goes in favor of strict interpretation. This choice is motivated by several reasons. First, there is the issue of witnesses of differing lengths, which has become apparent on the train level. Even though the short-cuts taken in the weak interpretation in these cases do not constitute errors, witnesses showing the correct sequence of cause and effect better conform to the real world and the intuition of the user. The second argument for the strict interpretation are negative scenarios, which require this interpretation in order not to generate undesired witnesses. A similar problem — the satisfaction of an LSC by two trains — can also arise for ordinary existential LSCs. Here the strict interpretation is better suited as well, since then no witness is generated in this case, which directly indicates that there is a problem. The third reason for choosing the strict interpretation is that it better fits the intuition of the user. Additionally the performance figures do not bar the use of the strict interpretation as stated above.

Having made this decision in favor of the strict interpretation, let us emphasize again, however, that there are preconditions, which must be fulfilled (cf. also the remarks on page 146):

1. The message mappings must be unique.

2. There must be only one active incarnation of the same LSC at each point in time.

Assumptions

The conclusion for assumptions is that internal assumptions are very useful and intuitive, since they are specified directly within the commitment LSC using the same elements, which make up the commitment. User-specified assumptions are likewise indispensable (cf. e.g. USACS1 or USL1). Extracted assumptions were not needed for the successful verification of the LSCs specified. In retrospective, we expect them to be used only rarely, since they are intended to be used mostly for enforcing unbounded liveness requirements on the environment. This will typically not be sufficient to guarantee liveness properties on the commitment side, because for embedded controllers a response from the environment often must occur within a certain time frame. For this reason there is e.g. no unbounded liveness requirement expressed on an environment instance in the LSCs for the radio-based crossing control system. The barriers for instance have to be closed within a certain amount of time in order for the crossing to operate correctly.

Statemate Semantics

With respect to the superstep semantics we can state that for our case study the verification times are always worse than in the step semantics. This statement can not be generalized, however, since the model at hand is in effect not very modular, i.e. there is a large degree of interdependence between different activities (cf. [Bie03]). Thus, most of the synchronization between all activities, which is necessary for the stabilization of the complete part of the model considered in the verification, is contained in the verified model, increasing the complexity of the verification task. More modular models are expected to show better performance figures. Regardless of the modularity, however, it can be said that timeout events, and presumably also scheduled actions, have a detrimental impact on the model check performance.

Enumeration vs. Superstep Counter Approach

Regarding the issue of which approach to expressing timing constraints in the superstep semantics should be used, no clear trend can be discerned. On the complete model USAF2 and USAF3 show that the counter approach performs much worse than explicit enumeration, which is corroborated by UCSB2 and UCSB3 on the crossing level. In both these examples the timing constraints are located in the pre-chart, whereas a contrary effect is observed for UCSTO3 and UCSTO4, where the timing constraint is part of the commitment. UCYE3 and UCYE4, where the constraint is also located in the actual LSCs, show a mixed picture, which in the weak interpretation is identical to the behavior observed for UCSTO3 and UCSTO4, but is reversed in the strict interpretation.

From a performance point of view therefore the explicit enumeration approach is to be used for timing constraints, if it can be applied. This is not always possible, since this approach can only express lower and exact bounds. In these cases, however, it is a viable optimization; otherwise the generally applicable counter approach must be used.

Abstraction

For universal LSCs the verification complexity can be reduced by applying data abstraction as done for the proofs on the system and train level and on crossing in the superstep semantics. It has proven to be very useful as the experimental results show, especially on the crossing component in the superstep semantics (cf. tables 11.19 on page 278 and 11.20), but it has to be noted that this technique has some limitations. The selection of which variables should be abstracted is done by the user and requires a thorough understanding of the model. Care has to be taken not to abstract a variable, whose precise value is essential for the verified property. This is even more important for the verification of liveness properties, since typically concrete and precise values of involved variables must be available in order to eventually produce a certain result. When abstracting variables in the superstep semantics additional care has to be exercised, because not only those variables, whose value actually contributes to the checked property, have to be handled with caution, but also those, which influence the stabilization of the model. Abstracting a variable of the latter type results in a diverging model, which then does guarantee no liveness requirement.

Timing Annotations

The conclusion regarding timing constraints, irrespective of the semantics, is that they result in a more complex verification task. The effect is rather mild in the step semantics, where in the best case there is little difference in the performance between the timed and untimed variant of the checked LSC (e.g. UCYE1 vs. UCYE2 in the step semantics) and in the worst case the verification time is increased by factor four (ECST1 vs. ECST2). In the superstep semantics there is a substantial increase, however, ranging from roughly six times to over sixty times longer run time for the timed variants. Only those timing annotations should thus be used in LSCs, which are absolutely necessary, in order to increase performance. The same advice can be given regarding timers in the STATEMATE model, i.e. timeout events and scheduled actions, which also have a detrimental impact on model checker run times, especially in the superstep semantics.

Activation

The possibility to explicitly characterize the activation point of a chart is a key benefit of LSCs, especially for formal verification. For the experimental results the invariant activation mode has been used predominantly, although the initial mode has proven useful in a number of occasions, e.g. to restrict the initial system state via initial assumptions or to cover the first occurrence of a protocol (UCSBI1 - UCSBI3). Pre-charts have also demonstrated that they are a profitable feature for the easy and intuitive specification of a required history of a desired protocol (cf. LSCs UCSB1 - UCSB3 and UCO1, UCO3, UCO4). Care has to be taken with pre-charts as well as assumptions, however, that they are compatible with the original requirements.

Mandatory vs. Possible

The distinction between mandatory and possible elements in an LSC massively enhance the expressiveness of LSCs compared to MSCs or SDs. As the experimental results show the distinction of existential and universal charts is an integral element for the smooth integration of LSCs into a model-based development process. The capability of discriminating mandatory and possible conditions and local invariants is indispensable for expressing requirements and internal assumptions. In this context simultaneous regions have proven

to be extremely useful for tying together messages and conditions and for providing reference points for timers and local invariants.

Hot and cold location temperatures allow to intuitively mark the instances, which are responsible for progress. No assessment of message temperatures and asynchronous messages can be done, since all communication in STATEMATE is instantaneous and messages can not be lost. But we expect cold messages to be useful in rare cases only, especially when considering formal verification. Typically either both sending and receipt of a message is of interest or the message is omitted altogether.

Methodology

Taking a more general view on LSCs we can state that the graphical specification of interactions between communicating entities (activities in this case) is extremely helpful in the early phases of the design process in order to thoroughly understand a system's basic functionality. Even though LSCs are not necessarily required for this purpose — MSCs and SDs can be used as well — the increased expressiveness of LSCs adds more substance to early scenarios by allowing to state information more explicitly. A very important feature in this context is the activation condition, which requires the designer to explicitly characterize the situation triggering the scenario which is described. Another key advantage of LSCs over the other two sequence chart dialects are local invariants, which allow to forbid the occurrence of messages in a specific part of the chart, e.g. LSCs ECSR2 or ECSB2, which explicitly contain the prohibition of sending the safe message after a failure of the red light or the barrier. Such side conditions are only implicitly contained in SDs or MSCs, by not including them. The enhanced expressiveness of LSCs thus allows to make property specifications more precise, while at the same time retaining the intuitiveness and visual appeal of standard sequence charts.

The subsequent phases of the development process capitalize both on the additional precision and information contained in the early existential LSCs and on the formal semantics. Existential verification reuses the early scenario and provides valuable feedback to the designer in the form of early indications that the model contains the basic communication behavior as specified in the existential LSCs. Existential verification profits from the explicitly specified side conditions, because the checking is more focused.

Even though existential verification did yield few results for the top level of the train control system model, we still feel that this technique is valu-

able in general, since it allows to check for the existence of deep interaction sequences, thus exploiting one of the major advantages of model-based development process: virtual integration. The complexity problems are to a degree inherent due to the long message sequences, which consequently require a deep exploration of the model. Chapter 12 offers some suggestions, how to reduce the verification complexity for both existential and universal LSCs.

The modularization of existential LSCs into universal ones and the subsequent strengthening of the latter allows a further reuse of the existential LSCs created in earlier phases of development. This smooth transition to universal protocol specifications is another very important benefit offered by LSCs and the accompanying methodology.

For the formal verification of universal LSCs the capability to express assumptions about environment behavior directly within the LSCs has proven to be extraordinarily useful. From the user's point of view it is convenient and natural to specify assumptions not only in the same graphical formalism, but within the same chart. Especially internal assumptions play a key role in the strengthening of universal LSCs: most assumptions used in the verification of the LSCs for the train control system have been expressed internally. An equally positive effect can be asserted for pre-charts, which allow an easy and intuitive specification of a prefix of the actual LSC. The experimental results show that the use of pre-charts is superior to employing user-specified assumptions both wrt. ease of use and efficiency (cf. LSCs UCO1 -UCO5).

Expressiveness

Compared to the patterns provided by the STVE LSCs offer more expressiveness like true liveness, local invariants, pre-charts and an arbitrary number of messages, conditions, etc. Another advantage of LSCs is their visual appeal and intuitiveness, whereas the patterns are represented in a non-graphical way. The downside to the enhanced expressiveness, however, is the decreased performance when model checking LSCs. Additionally more expert knowledge is required to specify properties as LSCs than as patterns.

STDs in comparison to LSCs are similar in terms of expressiveness and the required expert knowledge, but are focused on single components and are a state-based formalism, whereas LSCs are event-based. Protocols involving several components are expressible in STDs, but separate STDs, one per component, are needed and a rather complicated compositional verification

approach must be employed. LSCs are better suited to describe communication sequences involving several components due to their event-based nature. Long interaction sequences is moreover more naturally and more easily specified with LSCs.

A development process based on the STVE and LSCs should use all available analyses and checks at the appropriate time. While the model is being constructed the STVE robustness checks are employed, possibly in conjunction with existential verification of LSCs once an adequate part of the behavior has been modeled. Patterns are then used to verify simple properties exploiting their greater efficiency, and STDs and universal LSCs are used for the specification and verification of more complex properties. STDs are best suited for black-box requirements, whereas LSCs are ideal for liveness properties involving more than one component (grey-box requirements).

It has to be noted, though, that the successful verification of universal LSCs can be time consuming due to the potential iterations of (unsuccessful) verification run, problem analysis and changing the LSC or the model. For obtaining the experimental results for universal LSCs presented in the previous section up to three iterations were necessary. This drawback of LSCs does not outweigh the advantages like intuitiveness and enhanced expressive power, especially since each iteration gives the specifier more knowledge about the behavior of the considered (sub-)system, which can be reused when verifying other, similar LSCs.

In conclusion we can say that LSCs are very well suited for the specification of communication protocols, especially those involving liveness requirements. The use case of formal verification, which has been considered in detail here, seems a promising field of application, in particular in combination with the associated methodology, which eases the task of property specification by partly reusing LSCs, which have been created in earlier phases of the development process. Both the universal and existential verification show promise to become valuable additions to a model-based development process. The experimental results presented in this thesis serve as a proof of concept, but also indicate that the model check performance has to be improved before LSC-based formal verification can take root in everyday industrial development.

Chapter 12

Conclusion and Outlook

The incentive of this thesis has been the upgrade of standard sequence charts in order to play a more prominent role in the development of electronic control systems. In the introduction we have outlined the most important advanced use cases, like formal verification and automatic test vector generation, which transcend the applications of todays sequence charts: documentation of typical interactions and visualization of test or simulation traces. A detailed look at the two standardized sequence charts, Message Sequence Charts and UML's Sequence Diagrams, revealed that these, in their present form, are not well suited to meet the challenges posed by the envisioned advanced use cases. Both MSCs and SDs lack expressivity and a sound formal base, which motivates the definition of the language of Live Sequence Charts carried out in this work involving the fixing of the language features and their syntax as well as providing their formal semantics. We have performed the definition of both syntax and semantics incrementally, starting with the core features and step by step adding more advanced concepts.

The kernel language of LSCs comprises the basic elements, which make up a chart like instances, messages and conditions. We have defined the semantical basis in a constructive way by translating an LSC into a variation of a timed Büchi automaton. The relation to the system, whose behavior is described by the LSC, is established by relating the runs of the system to those accepted by the automaton. We have extended this definition of the basic features bit by bit to include the more advanced concepts of time, pre-charts and assumptions.

With the syntax and semantics of LSCs in place we have added the third vital building block: an application methodology, which outlines how and

where the created language should be employed. Here we laid special emphasis on the seamless reusability of LSCs and the use case of formal verification. We concluded with the experimental results we have obtained in the formal verification of a radio-based train control system, which has already served as a running example throughout this work. The practical application of LSCs to the advanced use case of formal verification has demonstrated the usefulness and viability of the features and semantics of LSCs, even though some complexity problems were encountered.

Outlook

The high model checking complexity observed for a number of the LSCs in the experimental results section is somewhat inherent. The purpose of LSCs is to specify the communication behavior between several components, so that components can not be considered in isolation. This entails a non-trivial base complexity depending on the number and size of the involved components. The reduction of complexity wrt. model checking is thus an important direction for future work.

For universal LSCs data abstraction has already been applied in several of the proofs in section 11.3. This optimization requires a high degree of user interaction and knowledge about the system, since the variables to be abstracted have to be chosen by the user. Bienmüller [Bie03] presents an approach, which does not require the user to select the variables to be abstracted, but tries to determine them automatically. At first a very coarse abstraction is used and it is checked, if the property holds. If this is not the case the abstraction is refined and the procedure is iterated until a true result is obtained or no further abstractions are possible. The entire process is guided by a heuristic, which combines standard backward cone-of-influence (COI) computation and a similar forward-oriented strategy. The backward-COI depends on the outputs used in the checked property, whereas the forward-COI regards the inputs. In [Bie03] this approach has successfully been applied to properties specified as patterns in the STVE.

This automated abstraction technique seems promising to help reduce the complexity for verification of universal LSCs and additionally increase user-friendliness. The applicability of this approach hinges on the possibility to automatically determine which propositions of the property are inputs and which are outputs, which is easy for patterns. In order to transfer this

technique to LSCs as well, information about inputs and outputs used in the chart must be made available to the tool performing the automated abstraction.

Another possibility for complexity reduction is the automatic decomposition of grey-box requirements specified as LSCs into local black-box properties, which can be proven on the involved components. Key idea here is that the decomposition already guarantees one of the tasks, which have to be performed for compositional reasoning: the proof that the sum of the local requirements indeed implies the global grey-box requirement. Questions to be investigated for this approach comprise:

- Is a decomposition possible for invariant LSCs? Due to overlapping incarnations it is not clear, if the decomposed view correctly reflects the global behavior.
- What is the best representation for the decomposed local requirements? They could e.g. be represented as LSCs, STDs, TSAs, temporal logic or yet another format.
- How are the decomposed properties triggered/activated? Is the activation condition used for triggering all local requirements? Is the synchronized activation necessary and if so, how is it achieved?
- How should pre-charts be treated?
- Are there restrictions on the features, which may be used in the global LSC? Can e.g. synchronization for shared conditions be guaranteed?

For the verification of existential LSCs a possible optimization is the partitioning of the entire LSC into smaller segments, each of which forms a separate verification task. The idea hence is to first find a witness for the initial segment, record the current global state of the FSM once it has been found and use this state as a fresh start point — forgetting the first witness — for the generation of the witness for the second segment and so on. In this way long existential LSCs can be broken down into smaller fragments, which are checked decidedly faster. The witness for the complete LSCs then has to be assembled from the witnesses for the fragments.

Regarding a further extension of the set of LSC features it is interesting to consider the inclusion of structuring elements, like e.g. sub-charts and loops.

These constructs have been mentioned in the original LSC paper [DH98], albeit without a formal definition. The use of such operational constructs for formal verification purposes, where the manner of specification of interactions is rather declarative, is doubtful. For other purposes, e.g. testing, they can be useful, however. An exhaustive treatment of sub-charts encompasses also the relation between elements within and outside of the sub-chart and if an interference between internal and external elements should be allowed, e.g. messages crossing the sub-chart boundary. Furthermore bounded and/or unbounded loops should be considered in this context as well as instance refinement similar to instance decomposition in MSCs.

Another direction of further work deals with the improvement of property specification for STATEMATE models using the superstep semantics. The view offered by STATEMATE does not lend itself to property specification with LSCs as the evaluation has shown. STATEMATE in the asynchronous semantics allows to only model one embedded controller, whose internals are given by the Activity Charts and Statecharts thereby mixing internal and external communication. A more natural view wrt. LSCs would be to model *several* embedded controllers and only allow to refer to messages sent between them, thus disregarding any internal communication of the individual embedded controllers. Fränzle et al. [FNMD03] propose such an interpretation and provides an according formal semantics. On this basis applications better suited to property specification with LSCs are conceivable like system-level testing, i.e. testing the interplay of several embedded controllers.

Other application contexts for LSCs are possible as well, like e.g. UML, which already contains Sequence Diagrams. While the LSC language as presented in this work considers the basic elements, which are necessary in order to specify protocols as LSCs also within a UML context, the concrete relations between LSC and model elements still have to be defined. Whereas this mapping is fairly simple and obvious in the case of STATEMATE, more effort has to be invested for the practical application of LSCs in the UML world. A UML model is generally not as persistent as a STATEMATE model, which does not change its structure during its lifetime. One of the key features of object-orientation is the dynamic creation and destruction of objects, which entails that the structure of the model and the relation between objects in the model change over time as new objects are created, old ones are destroyed and associations between objects are redirected. There are additional questions to be resolved, like e.g. how sending and receipt of an event are to be interpreted in the UML model in view of an event queue, or the mapping

of LSC instances to model objects in the presence of relative identifications, e.g. of the form `crossing->itsTrain`.

We have undertaken a first attempt at providing such a relation in [KW02], albeit without referring to a formal representation of a UML model. In the context of ongoing efforts at the University of Oldenburg and OFFIS, Damm and Westphal [DW03] provide a more precise and formal definition of the relation of LSC to model elements basing on the semantics for a subset of the UML presented in [DJPV03].

The highly dynamic nature of UML models makes the formal verification a challenging field of research involving the definition of a formal semantics, the investigation of new verification strategies in order to cope with changing and even unbounded models, and the development of a fitting format for specifying properties. LSCs seem a very natural choice for the last task and it will be interesting to see how this area evolves.

Appendix A

LSCs for the Radio-based Train Control System

This chapter collects the LSCs, which have been specified and verified for the train control system. For universal and assumption LSCs additionally the generated timed symbolic automata are included. Some LSCs have been omitted when they were too large to be included (e.g. due to a large timing constraint in the superstep semantics using the explicit enumeration approach).

A.1 LSCs for System

A.1.1 Existential LSCs

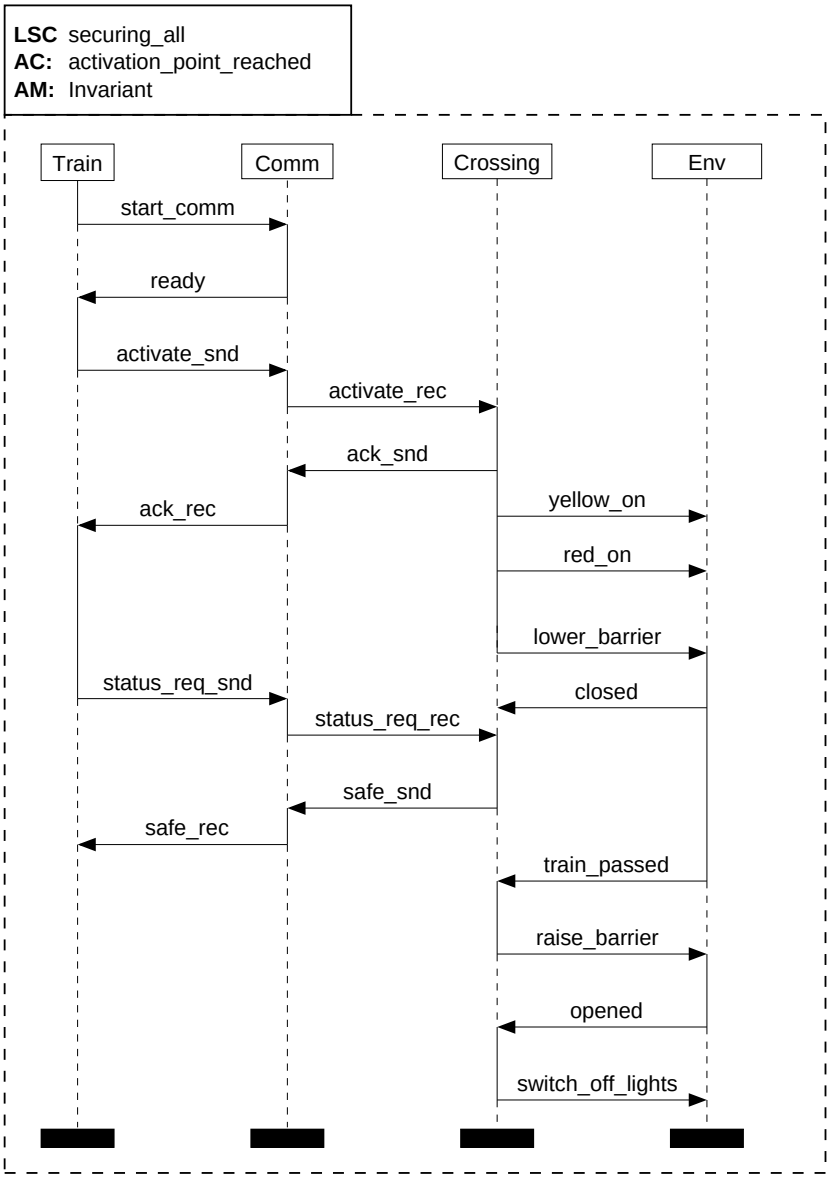


Figure A.1: Existential LSC showing the good case

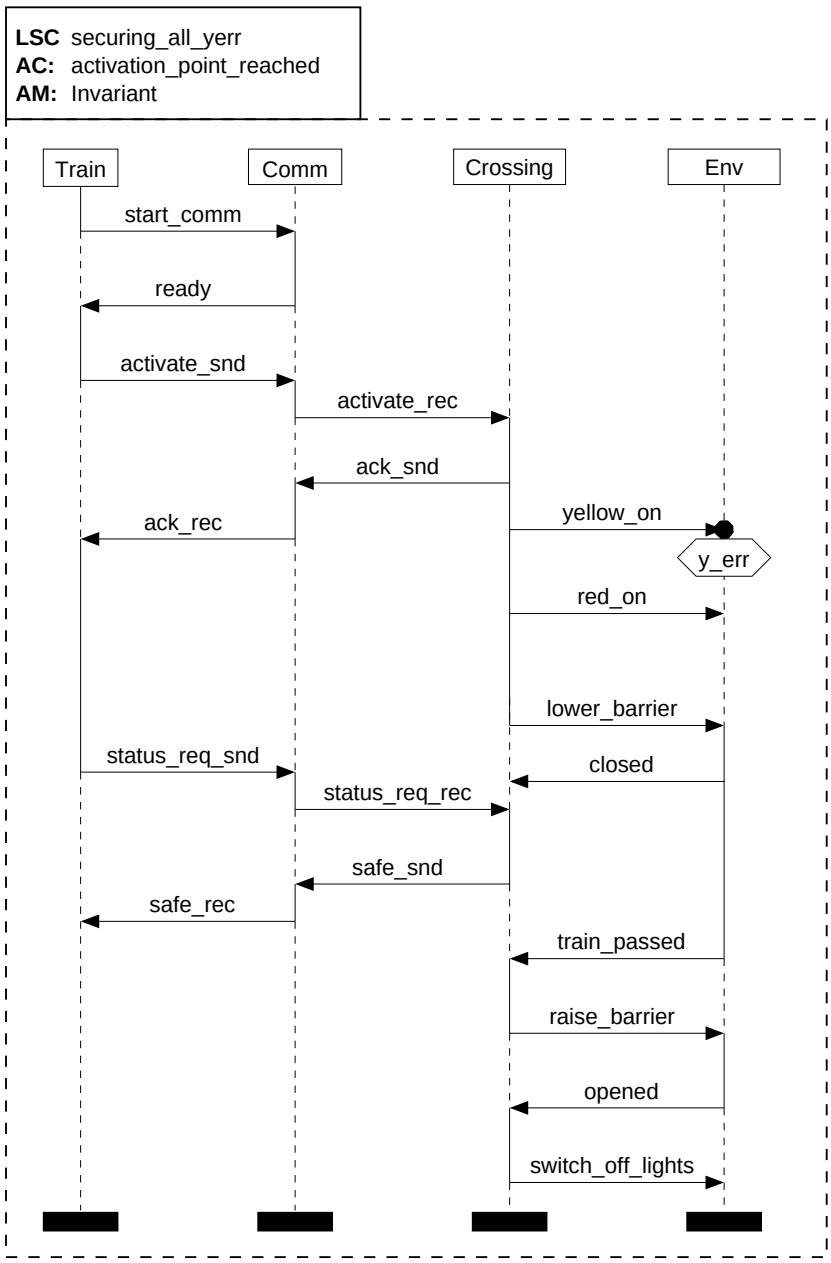


Figure A.2: Existential LSC showing a failure of the yellow light

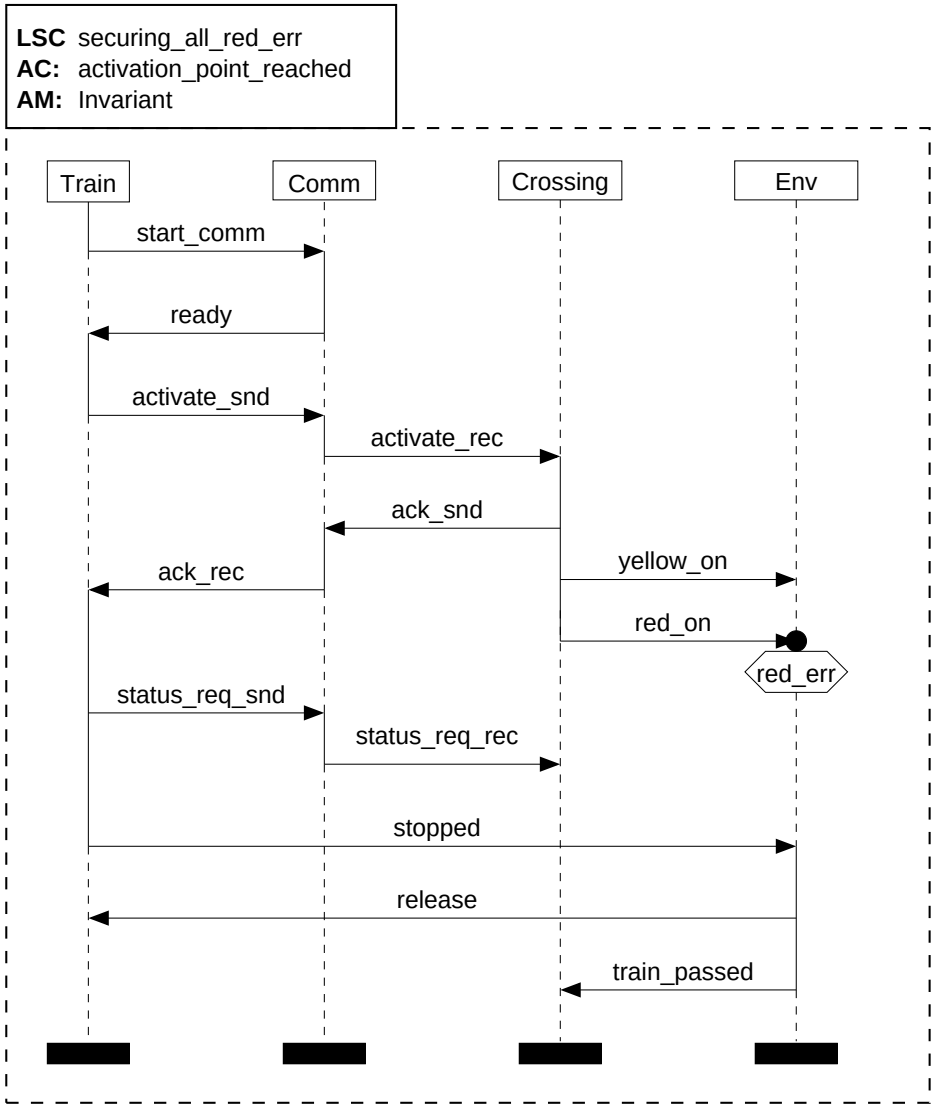


Figure A.3: Existential LSC showing a failure of the red light without explicit prohibition of the safe message

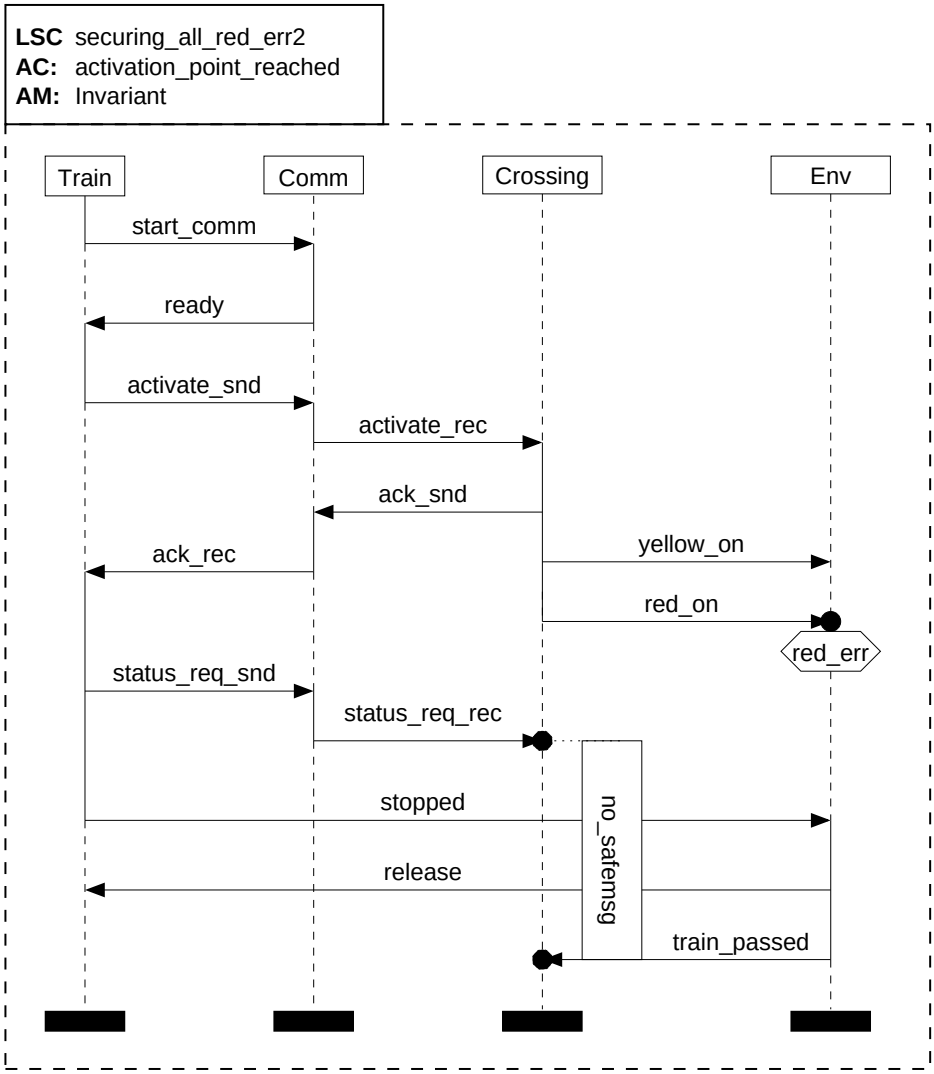


Figure A.4: Existential LSC showing a failure of the red light with explicit prohibition of the safe message

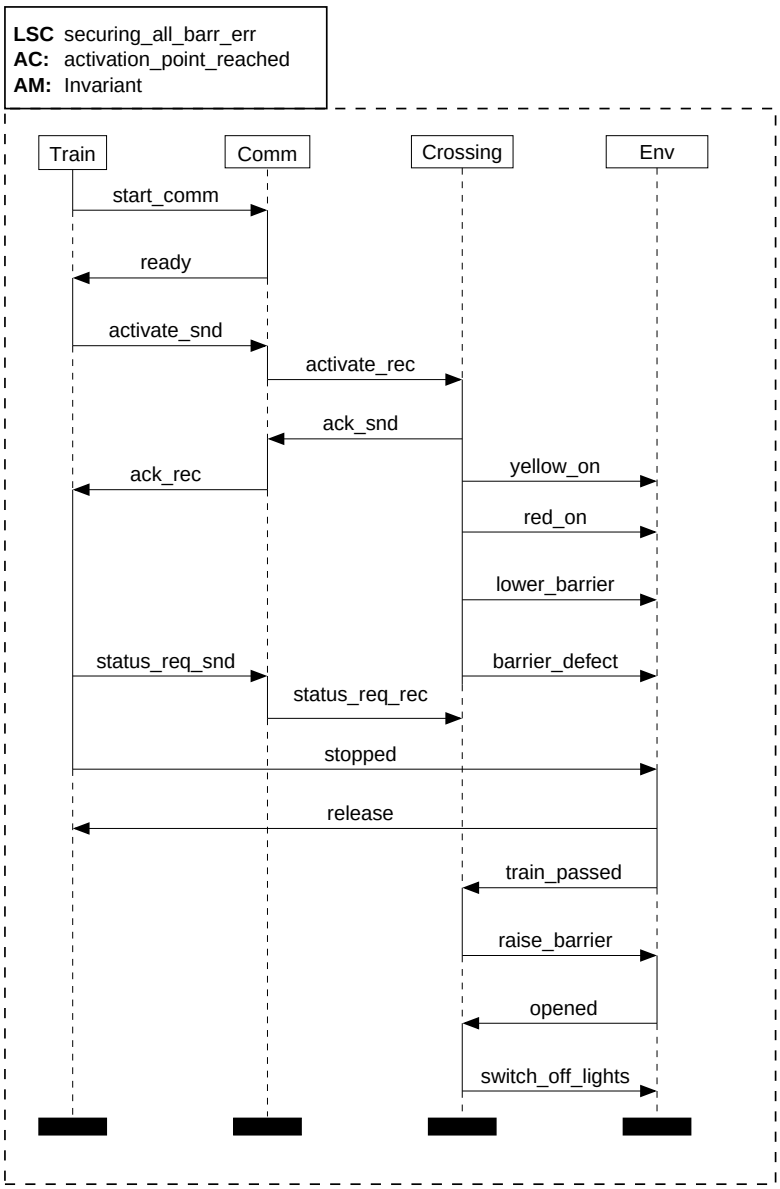


Figure A.5: Existential LSC showing a failure of the barrier without explicit prohibition of the safe message

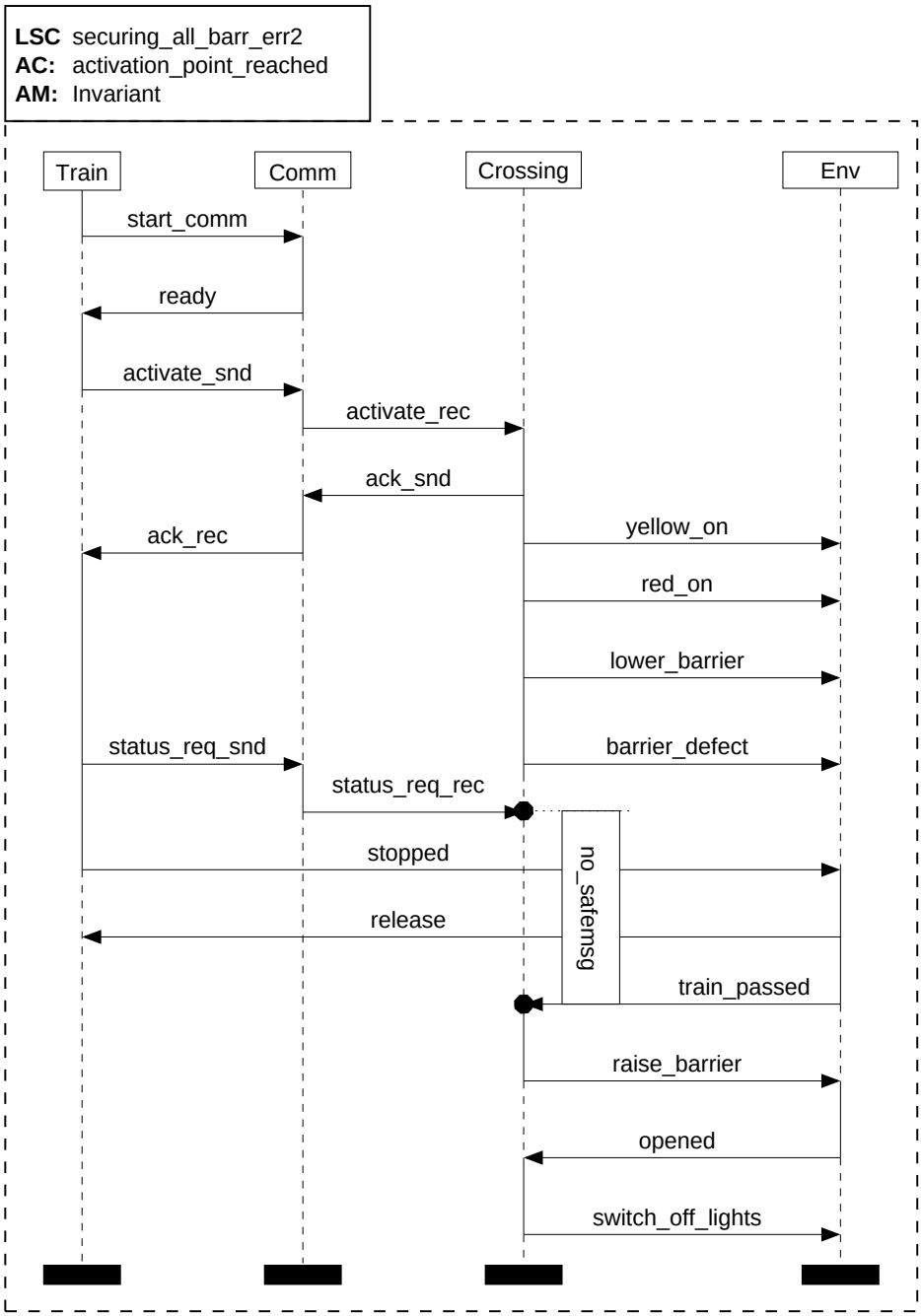


Figure A.6: Existential LSC showing a failure of the barrier with explicit prohibition of the safe message

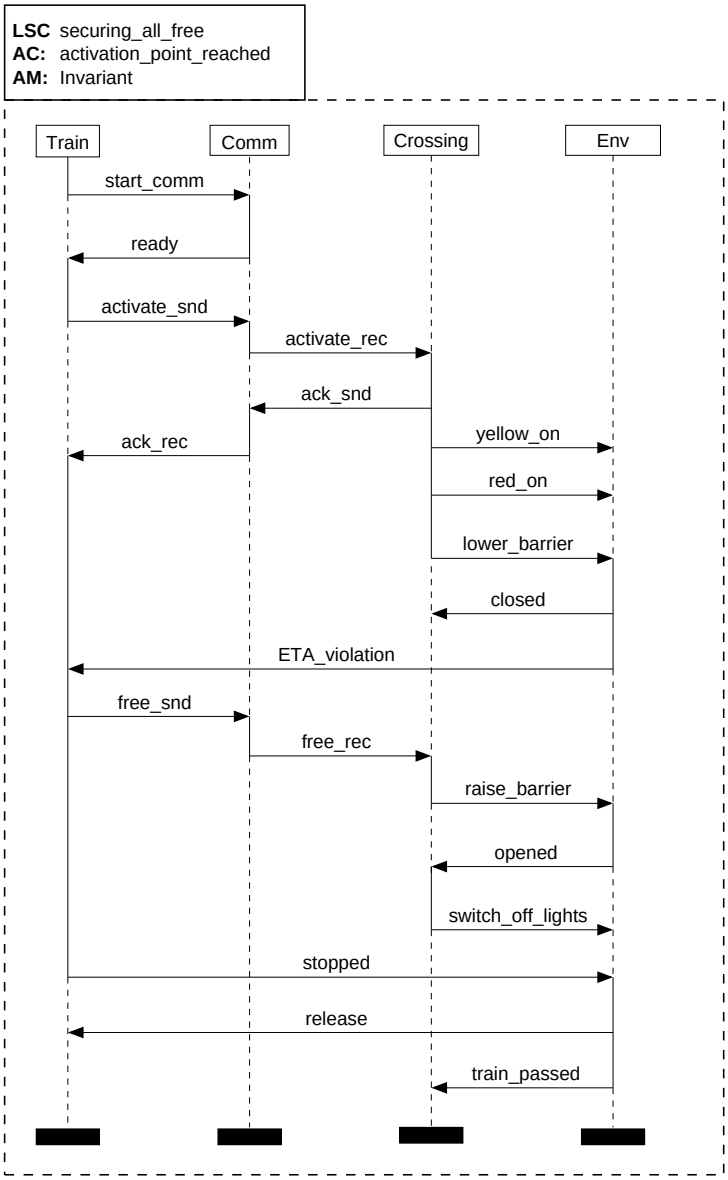


Figure A.7: Existential LSC showing the train sending the free message

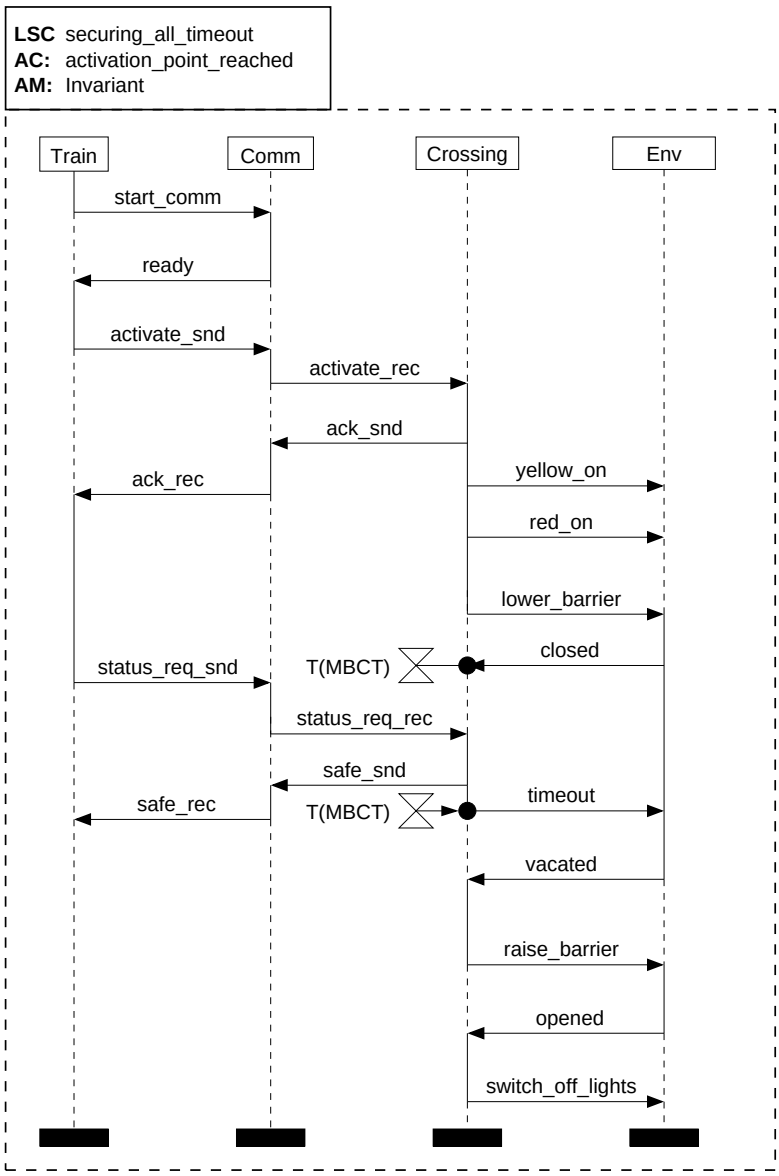


Figure A.8: Existential LSC showing a timeout at the crossing with timer

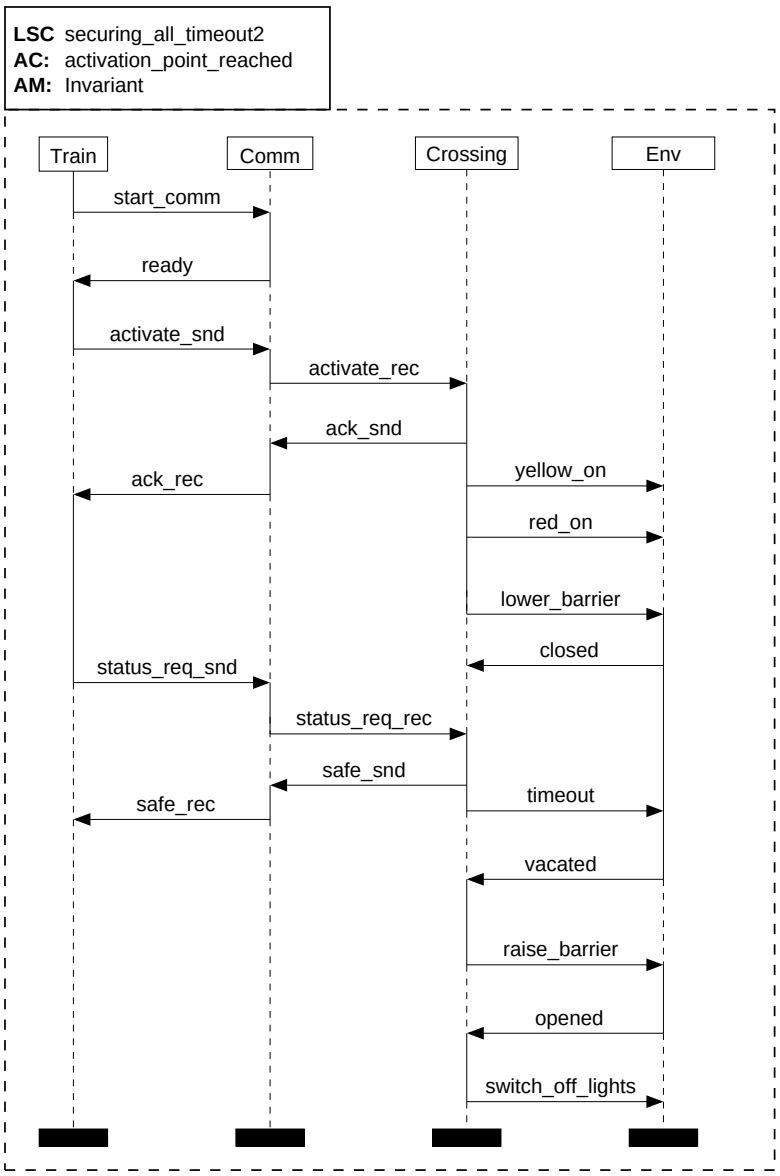


Figure A.9: Existential LSC showing a timeout at the crossing without timer

A.1.2 Universal LSCs

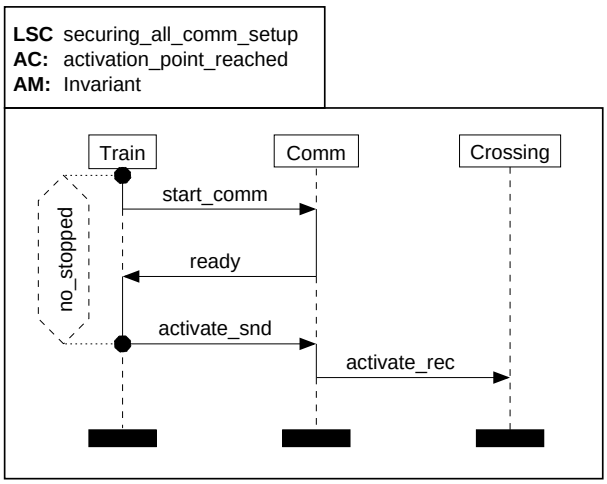


Figure A.10: Universal LSC for establishing the communication channel

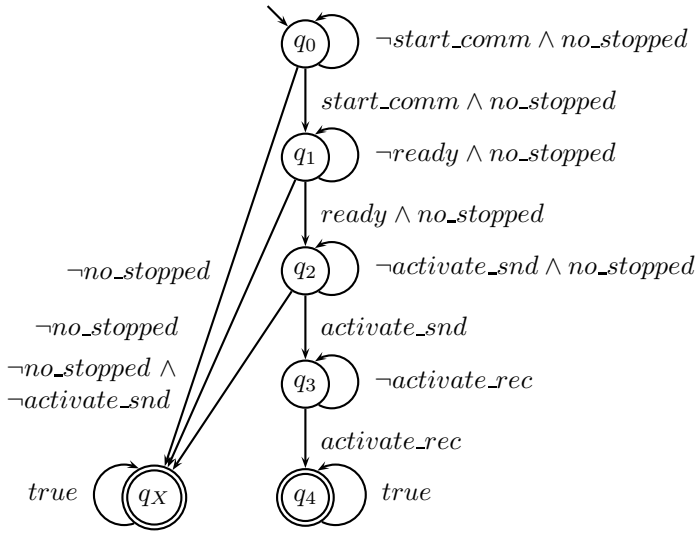


Figure A.11: TSA for LSC body of `securing_all_comm_setup` (weak interpretation)

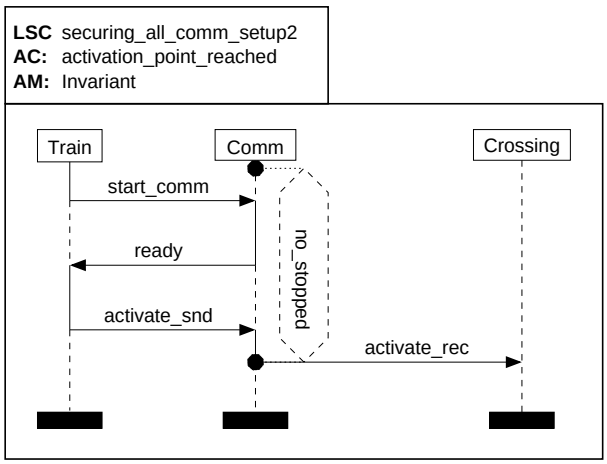


Figure A.12: Alternate universal LSC for establishing the communication channel

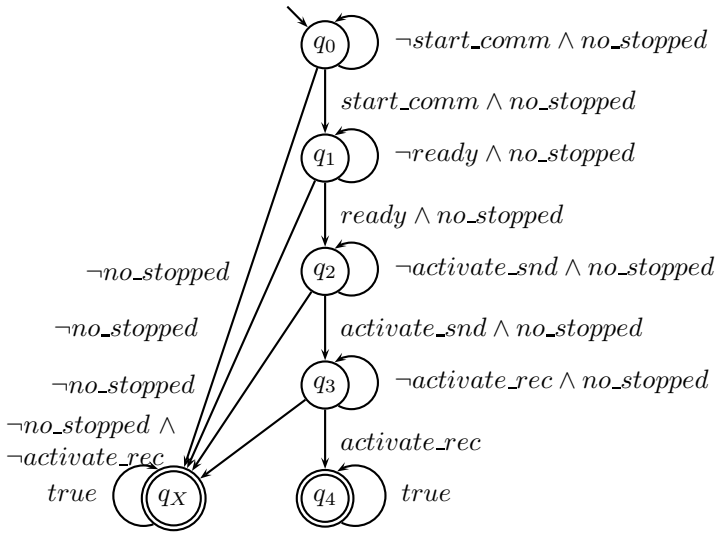


Figure A.13: TSA for LSC body of `securing_all_comm_setup2` (weak interpretation)

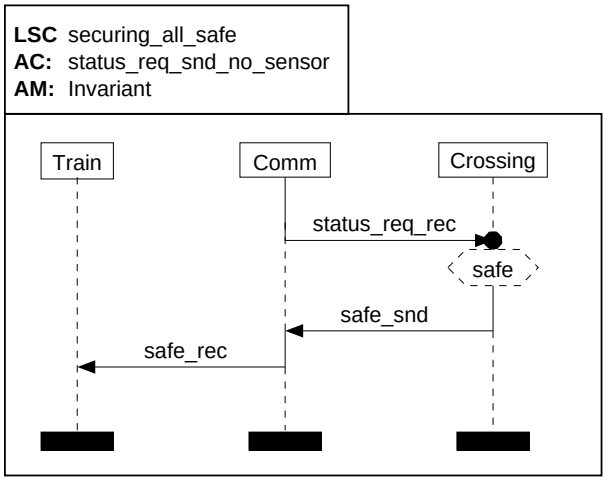


Figure A.14: Universal LSC for a positively answered status request

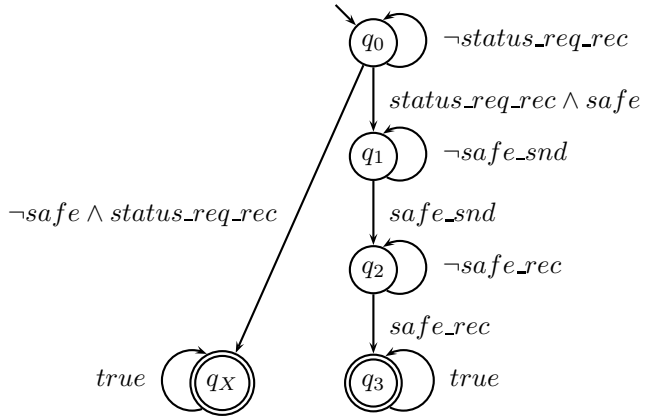


Figure A.15: TSA for LSC body of `securing_all_safe` (weak interpretation)

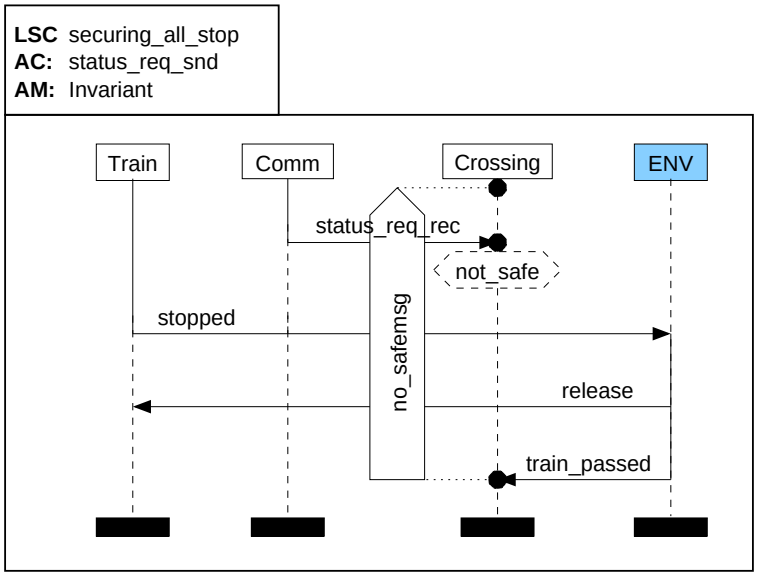


Figure A.16: Universal LSC for a not answered status request

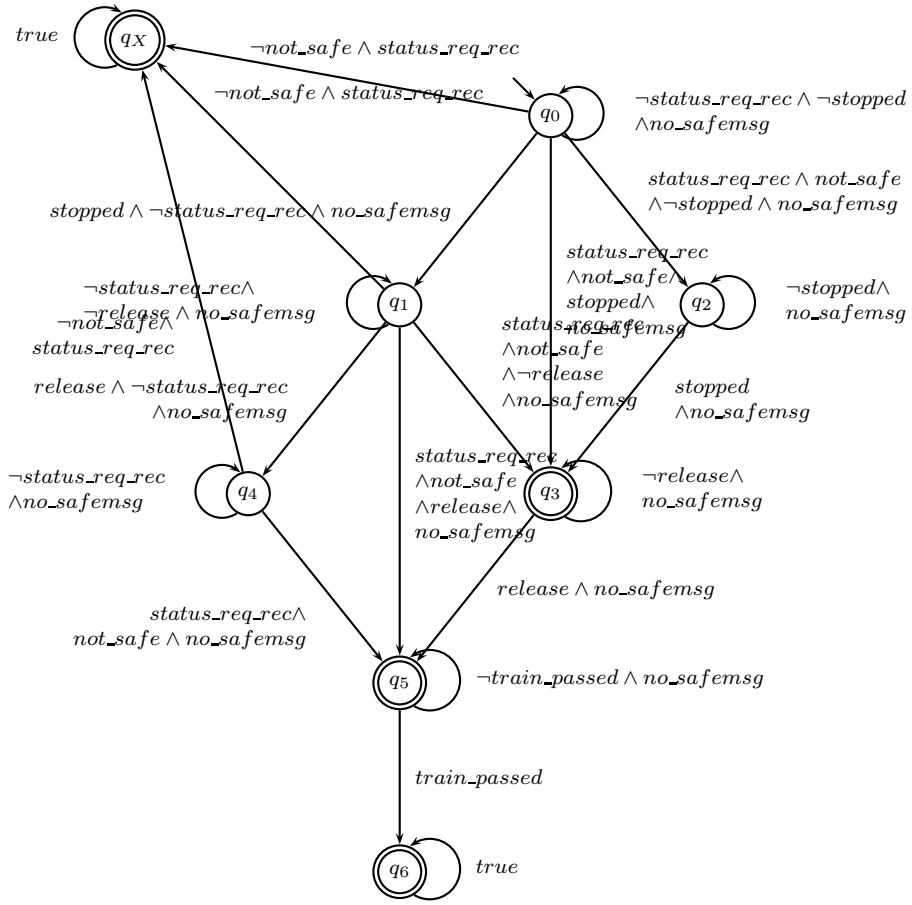


Figure A.17: TSA for LSC body of `securing_all_stop` (weak interpretation)

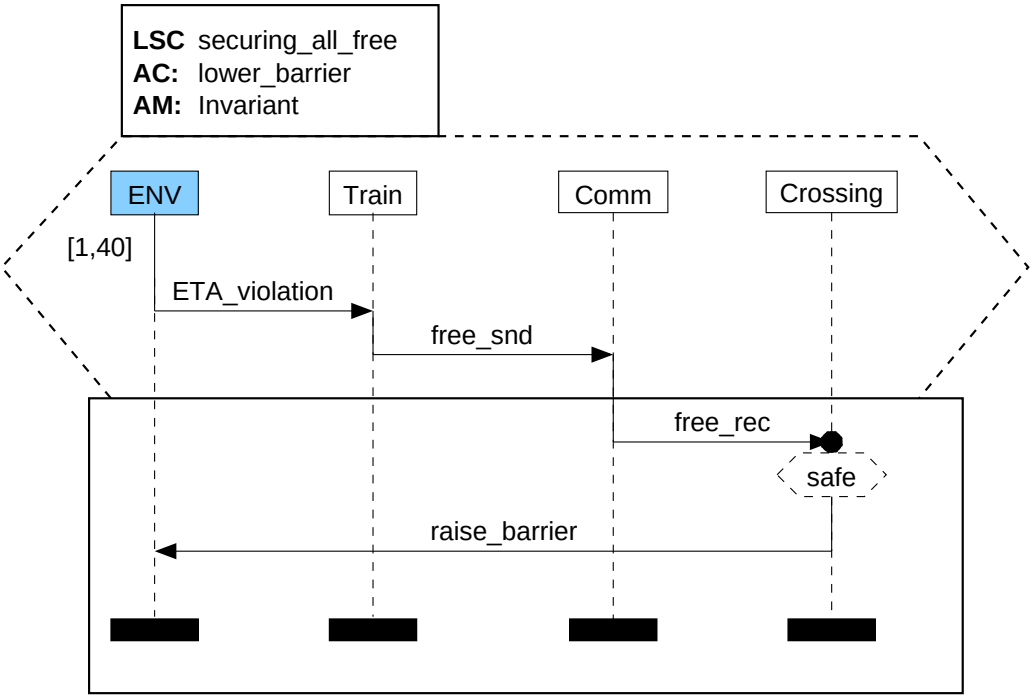
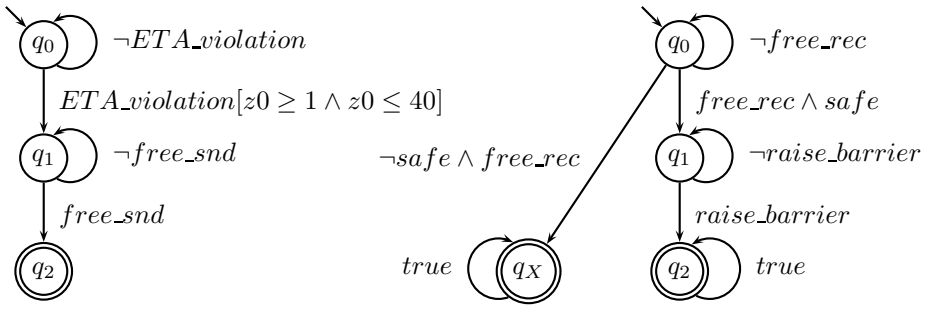


Figure A.18: Universal LSC for the sending of the free message



(a) Finite automaton for pre-chart (b) TSA for LSC body

Figure A.19: Automata for LSC `securing_all_free` (weak interpretation)

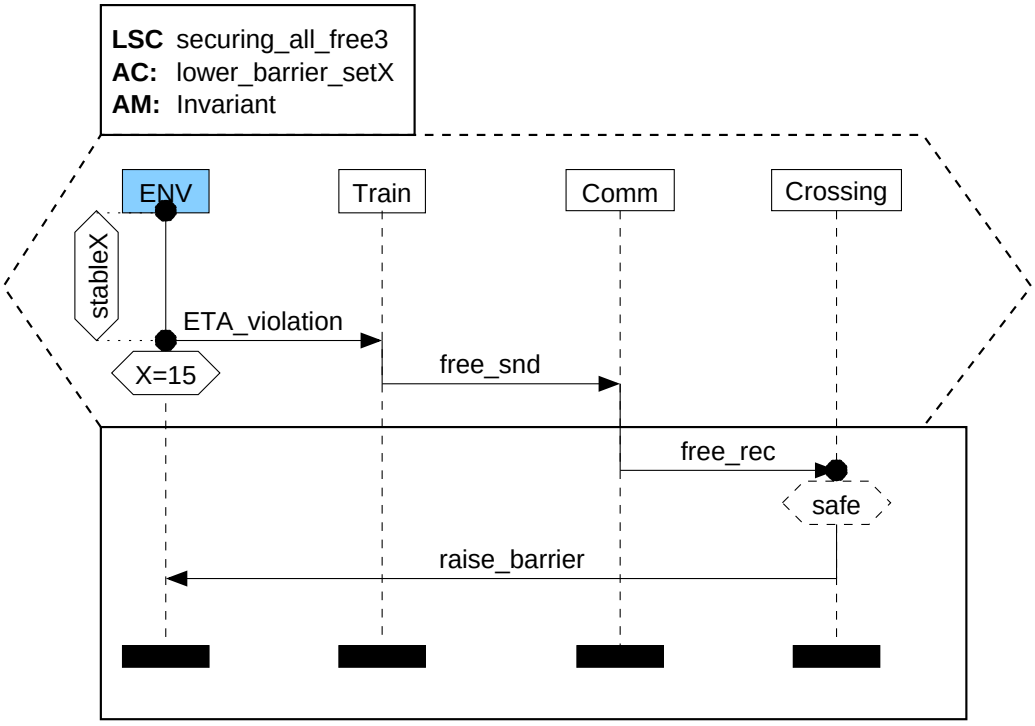
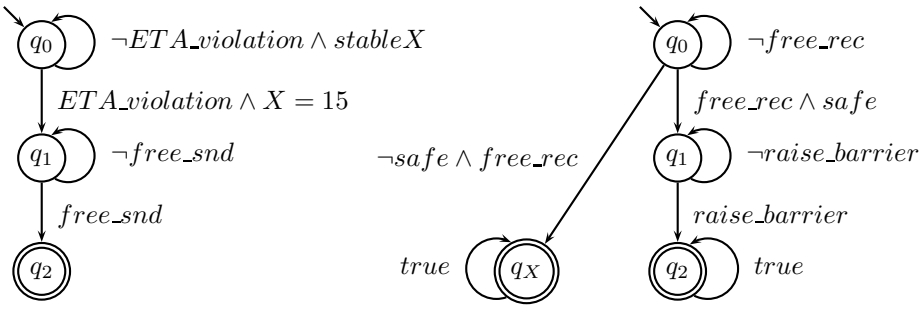


Figure A.20: Universal LSC for the sending of the free message (superstep semantics, counter approach)



(a) Finite automaton for pre-chart (b) TSA for LSC body

Figure A.21: Automata for LSC `securing_all_free3` (weak interpretation)

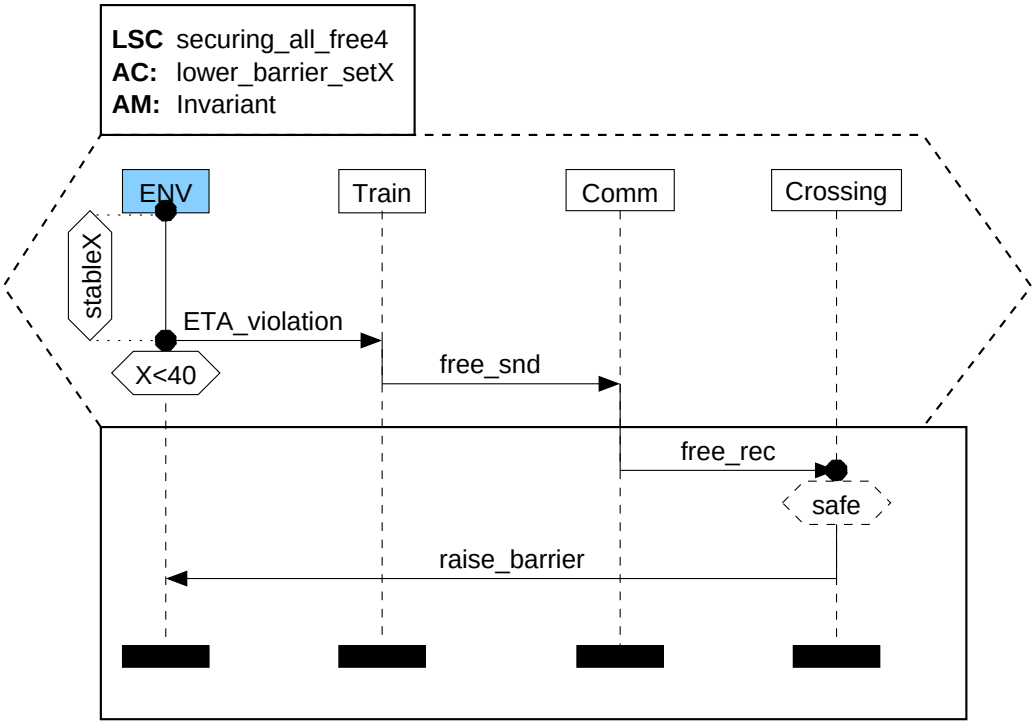
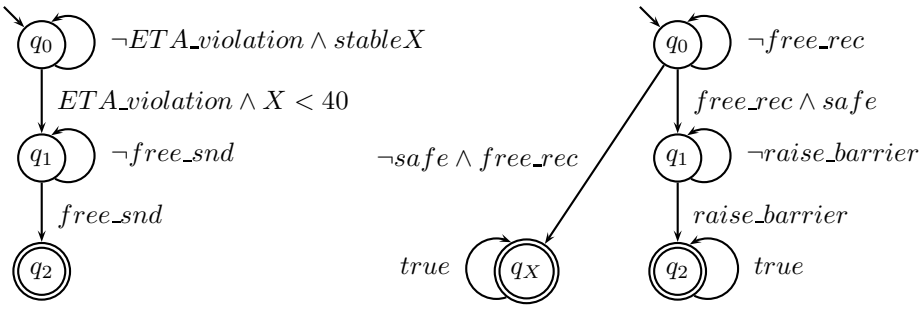


Figure A.22: Universal LSC for the sending of the free message (superstep semantics, counter approach)



(a) Finite automaton for pre-chart (b) TSA for LSC body

Figure A.23: Automata for LSC `securing_all_free4` (weak interpretation)

A.1.3 Assumption LSCs

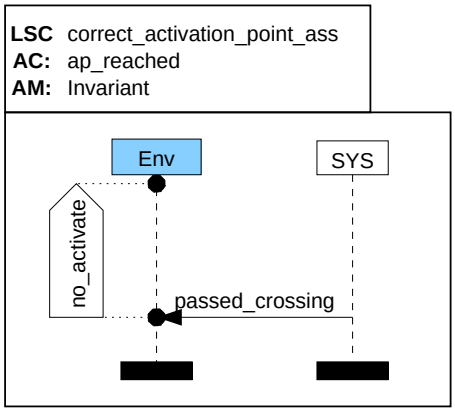


Figure A.24: Assumption LSC for the restriction of reaching the activation point

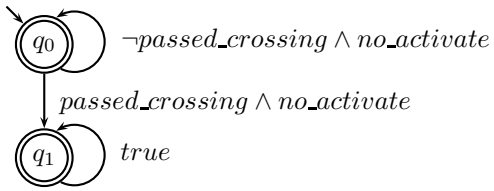


Figure A.25: TSA for LSC body of `correct_activation_point_ass` (weak and strict interpretation)

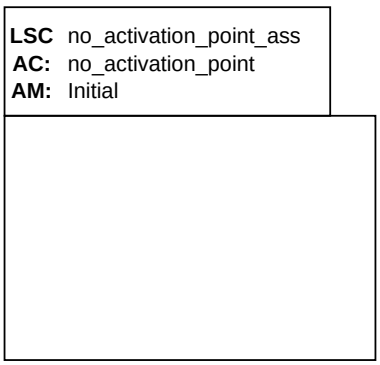


Figure A.26: Assumption LSC forbidding the initial indication of reaching the activation point

A.2 LSCs for Train

A.2.1 Existential LSCs

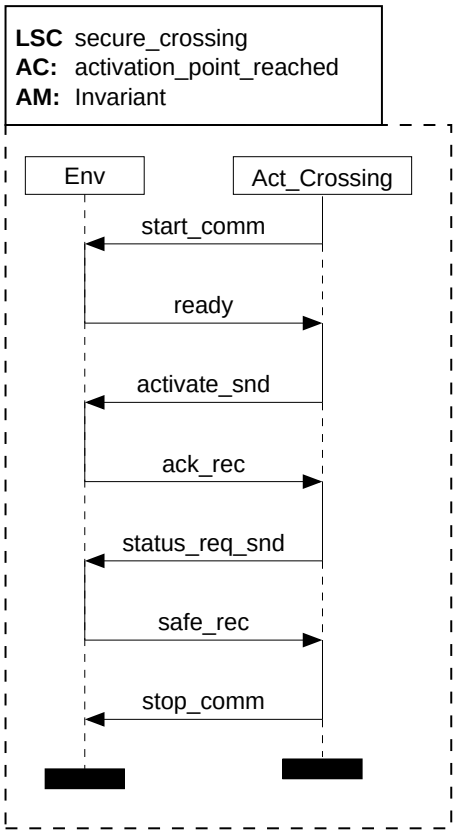


Figure A.27: Existential LSC for the good case (successful securing of the crossing)

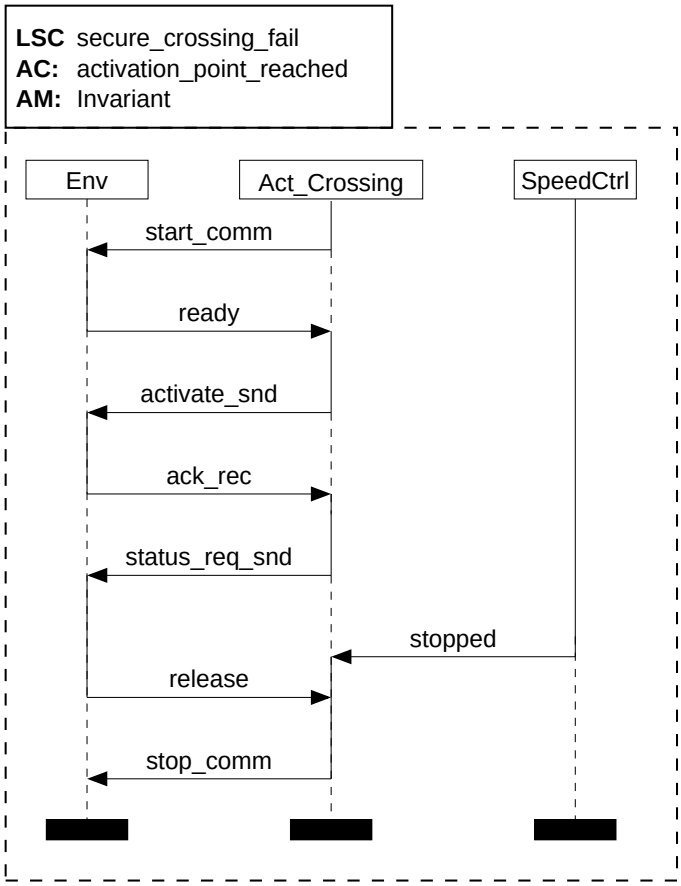


Figure A.28: Existential LSC for a failed securing of the crossing

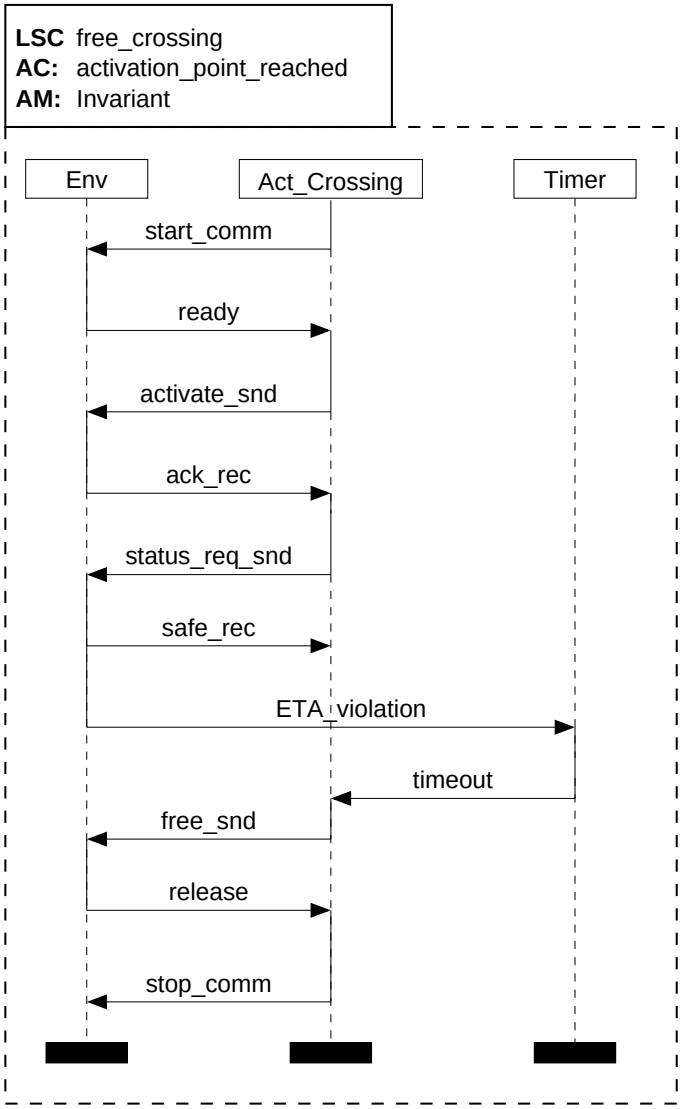


Figure A.29: Existential LSC for the sending of the free message

A.2.2 Universal LSCs

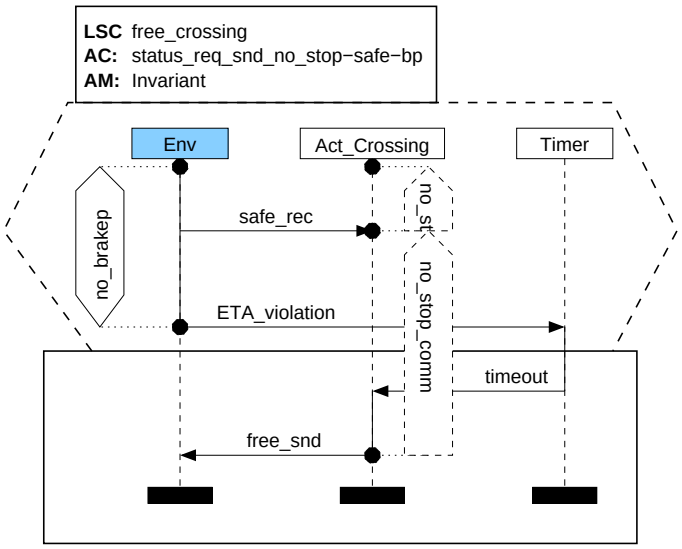
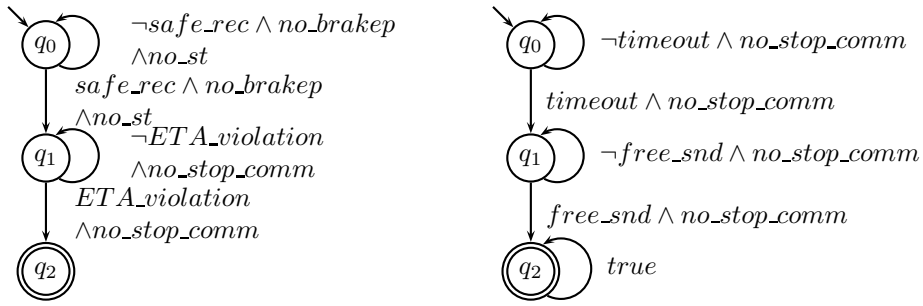


Figure A.30: Universal LSC for the sending of the free message



(a) Finite automaton for pre-chart (b) TSA for LSC body

Figure A.31: Automata for LSC `free_crossing`

A.3 LSCs for Crossing

A.3.1 Existential LSCs

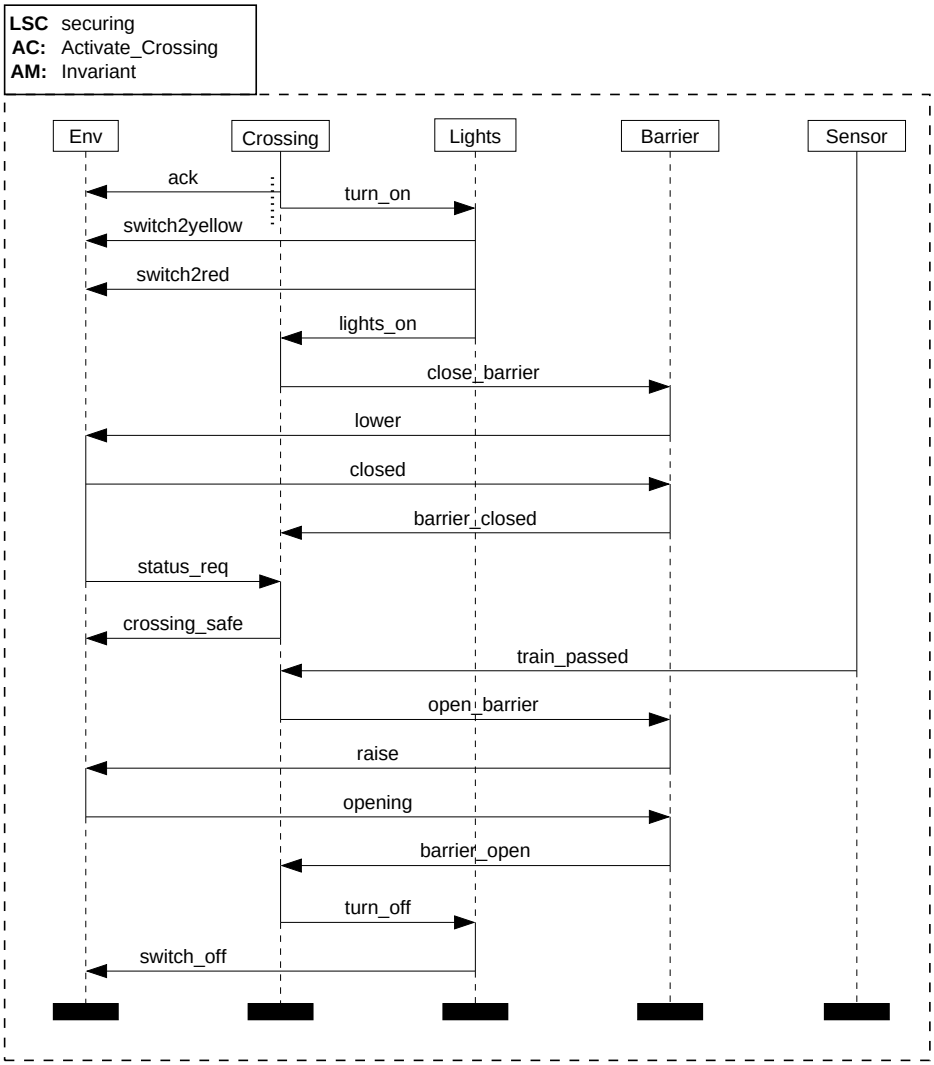


Figure A.32: Existential LSC for the good case (successful securing of the crossing)

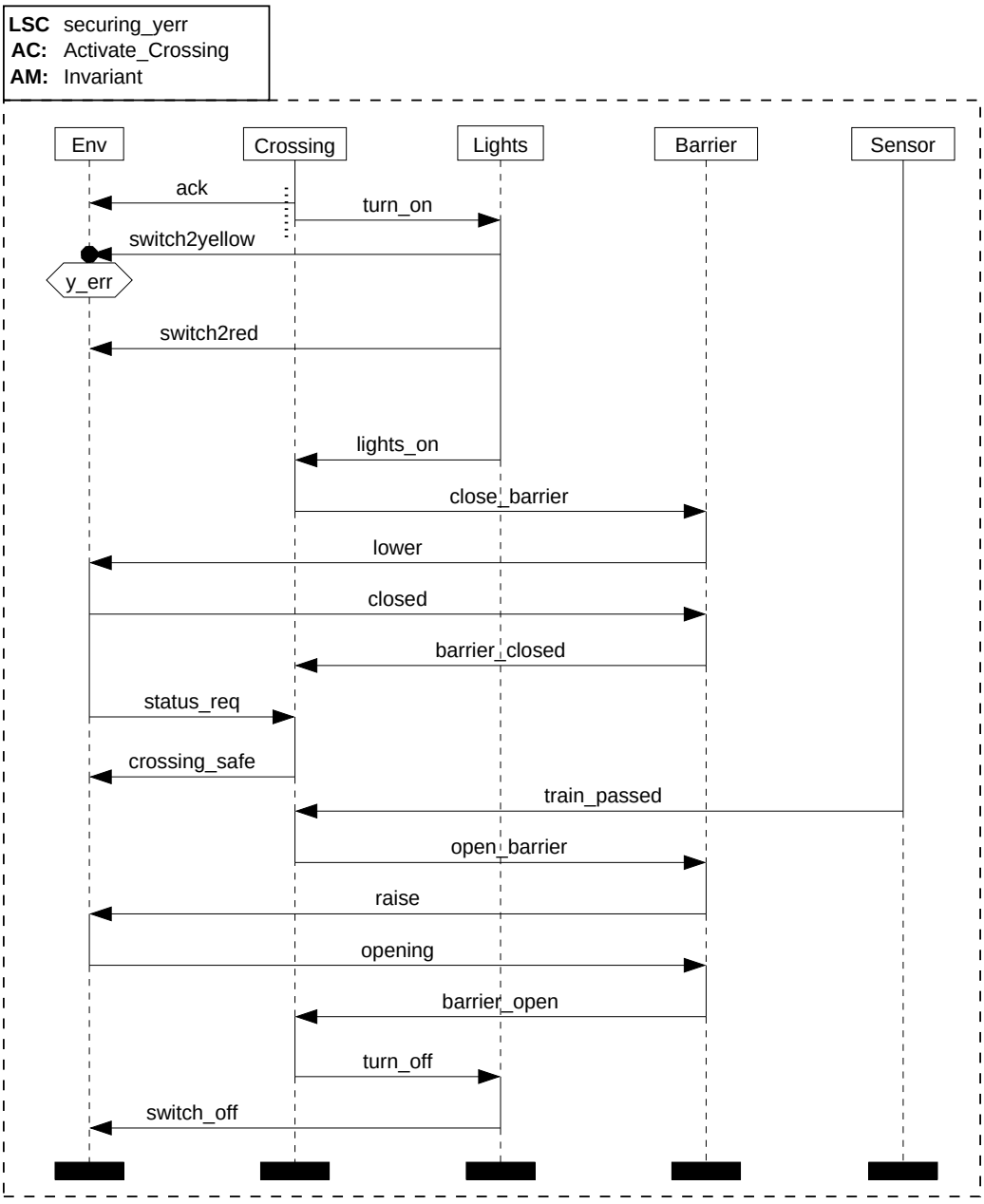


Figure A.33: Existential LSC for a defect yellow light

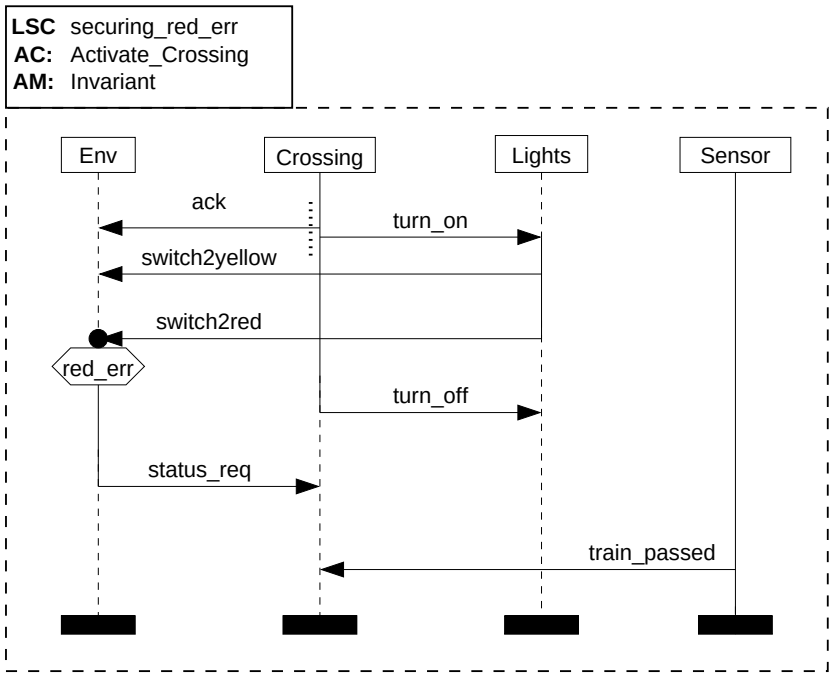


Figure A.34: Existential LSC for a defect red light without explicit prohibition of the safe message

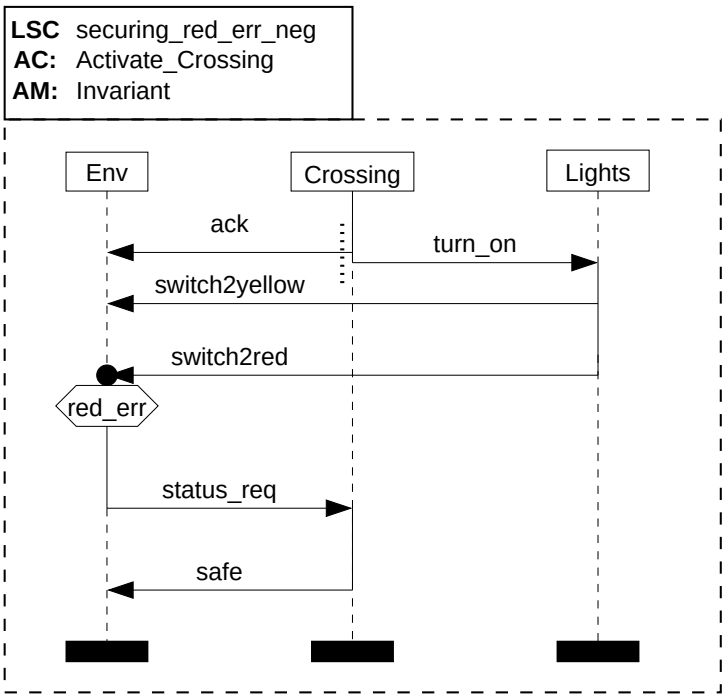


Figure A.35: Negatively interpreted existential LSC showing the erroneous situation of the crossing sending the safe message although a red light failure has occurred

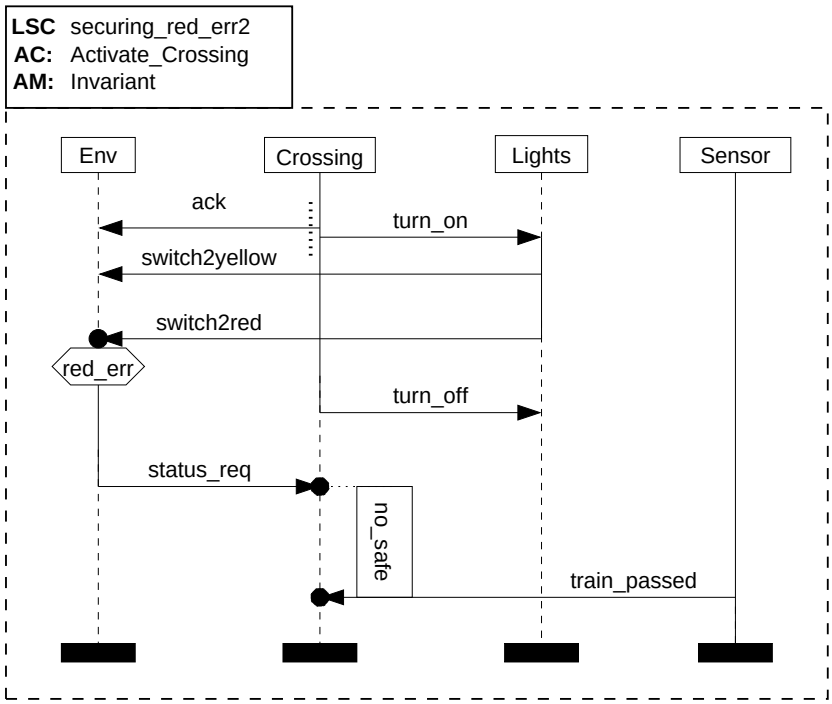


Figure A.36: Existential LSC for a defect red light with explicit prohibition of the safe message

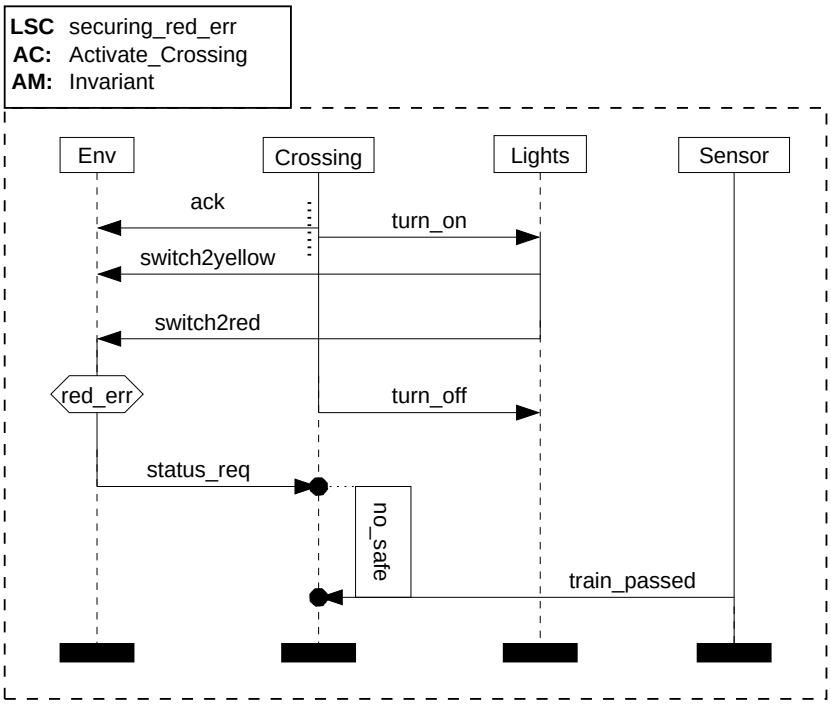


Figure A.37: Existential LSC for a defect red light with explicit prohibition of the safe message

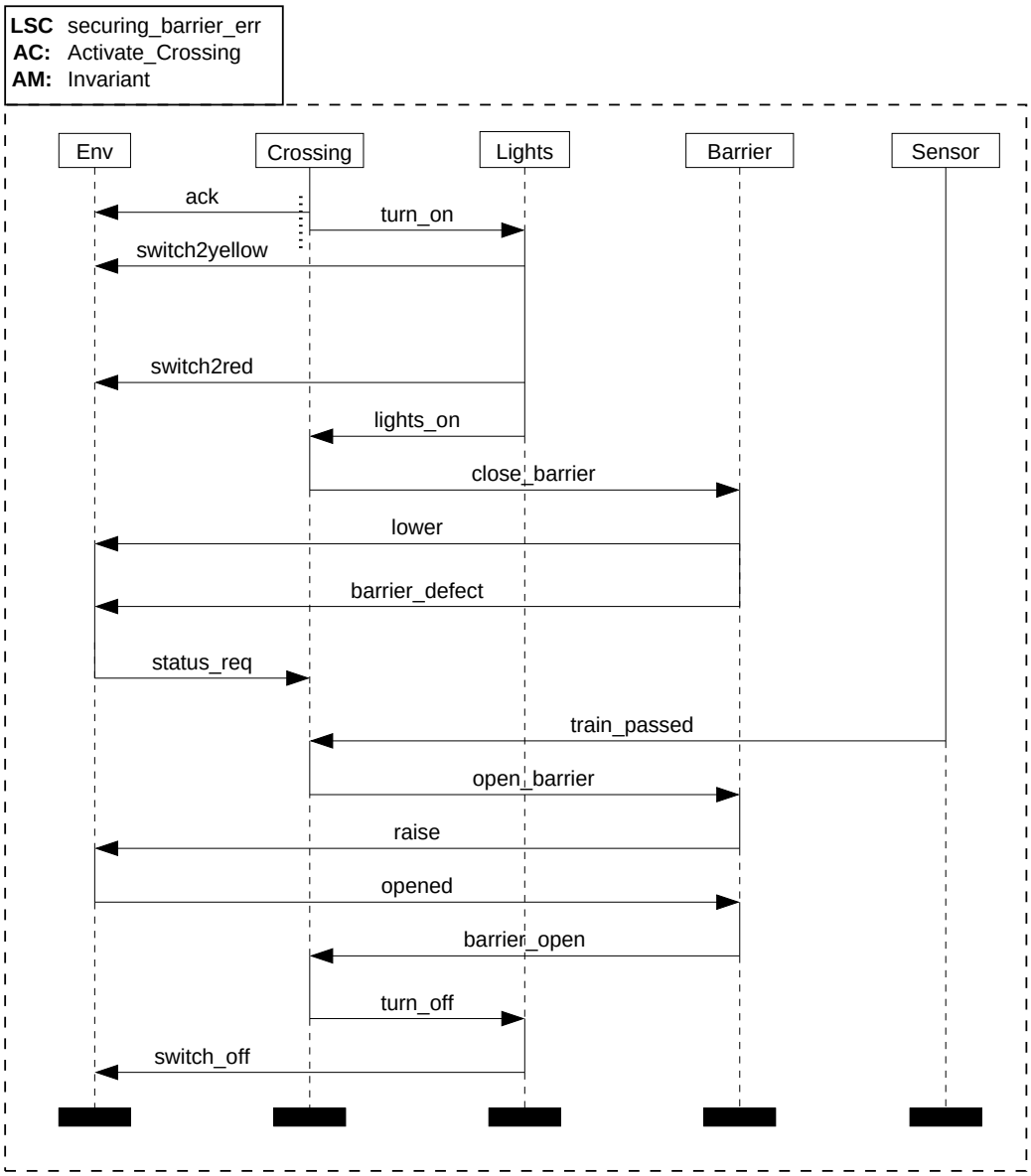


Figure A.38: Existential LSC for a defect barrier without explicit prohibition of the safe message

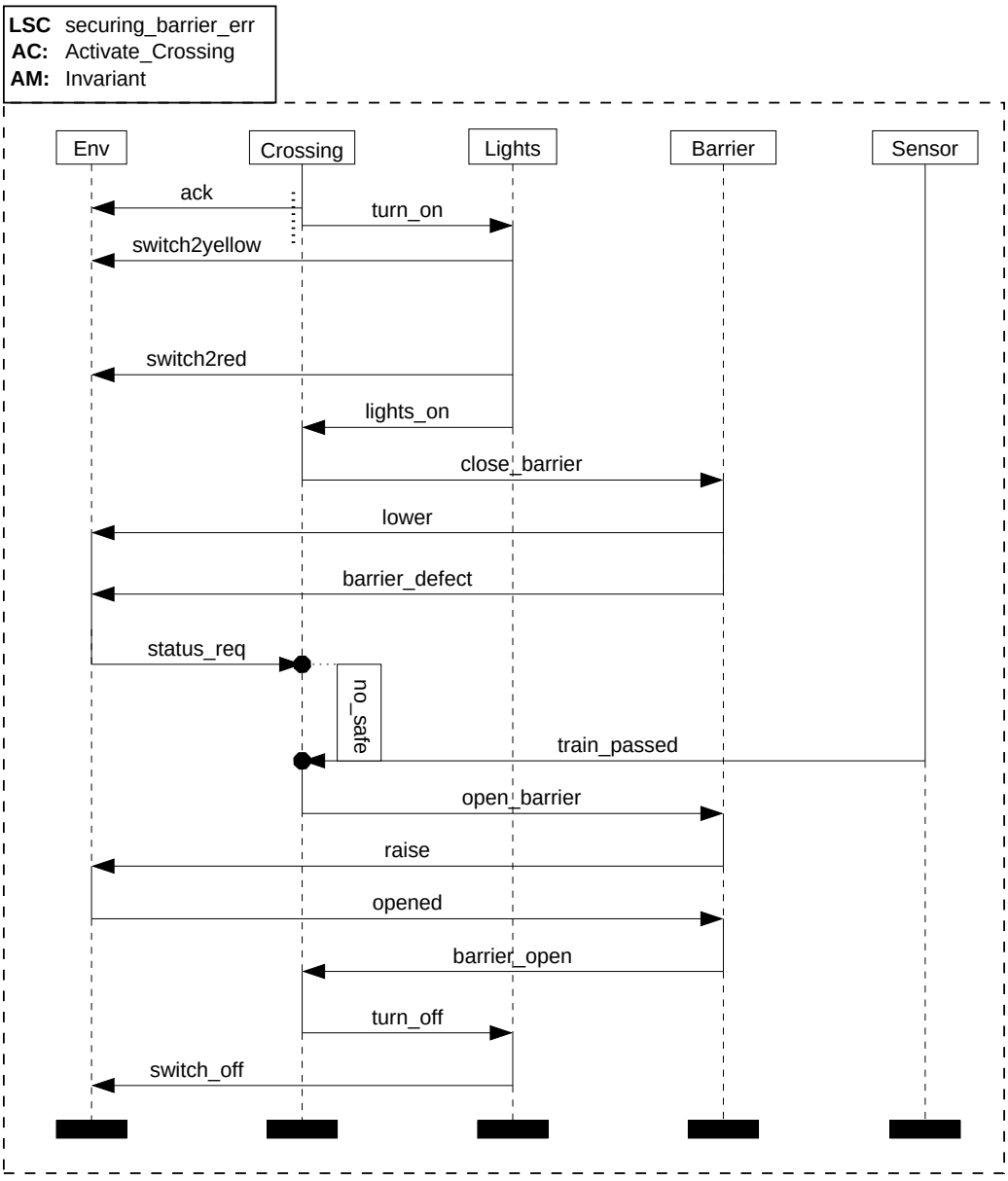


Figure A.39: Existential LSC for a defect barrier with explicit prohibition of the safe message

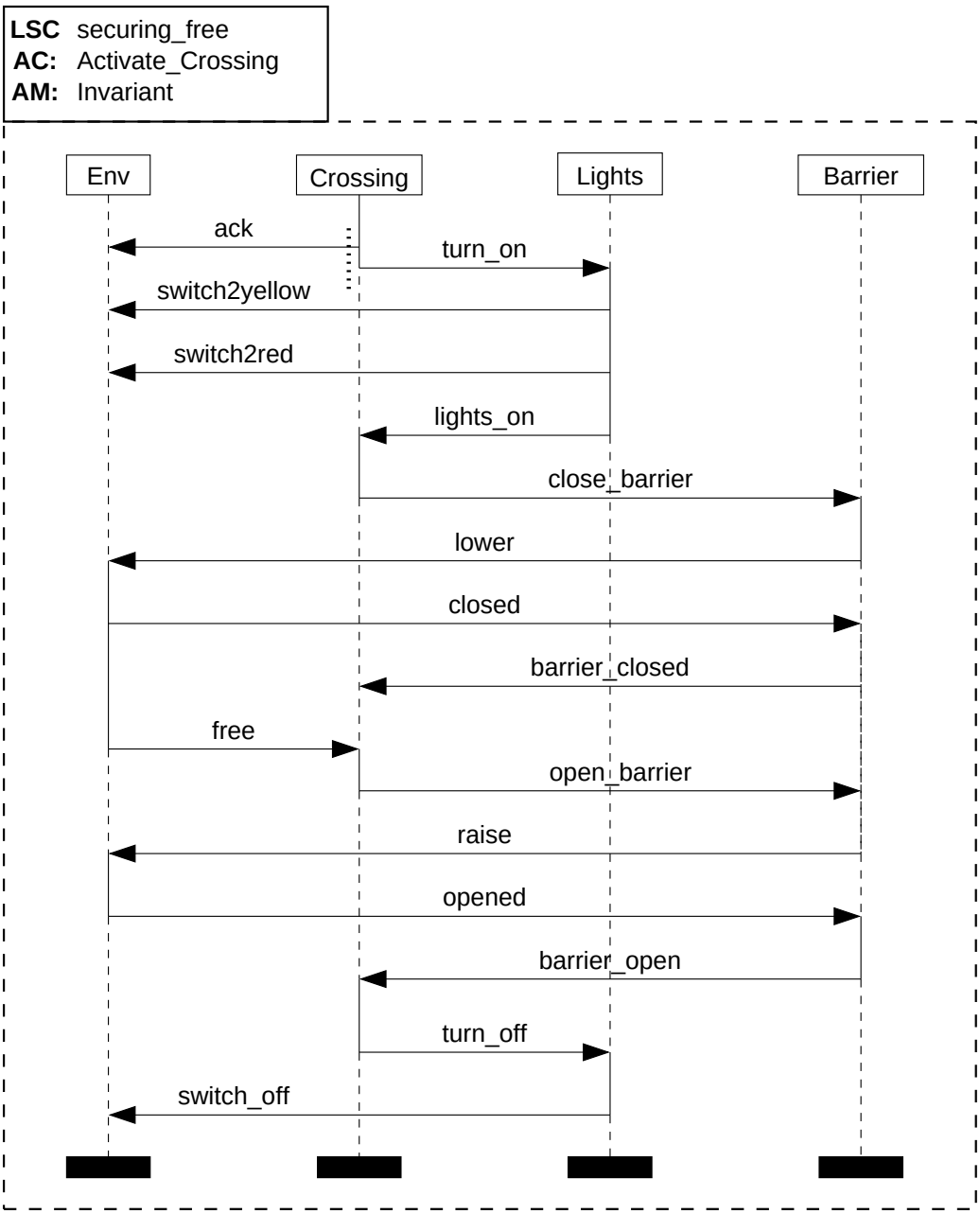


Figure A.40: Existential LSC for receipt of the free message

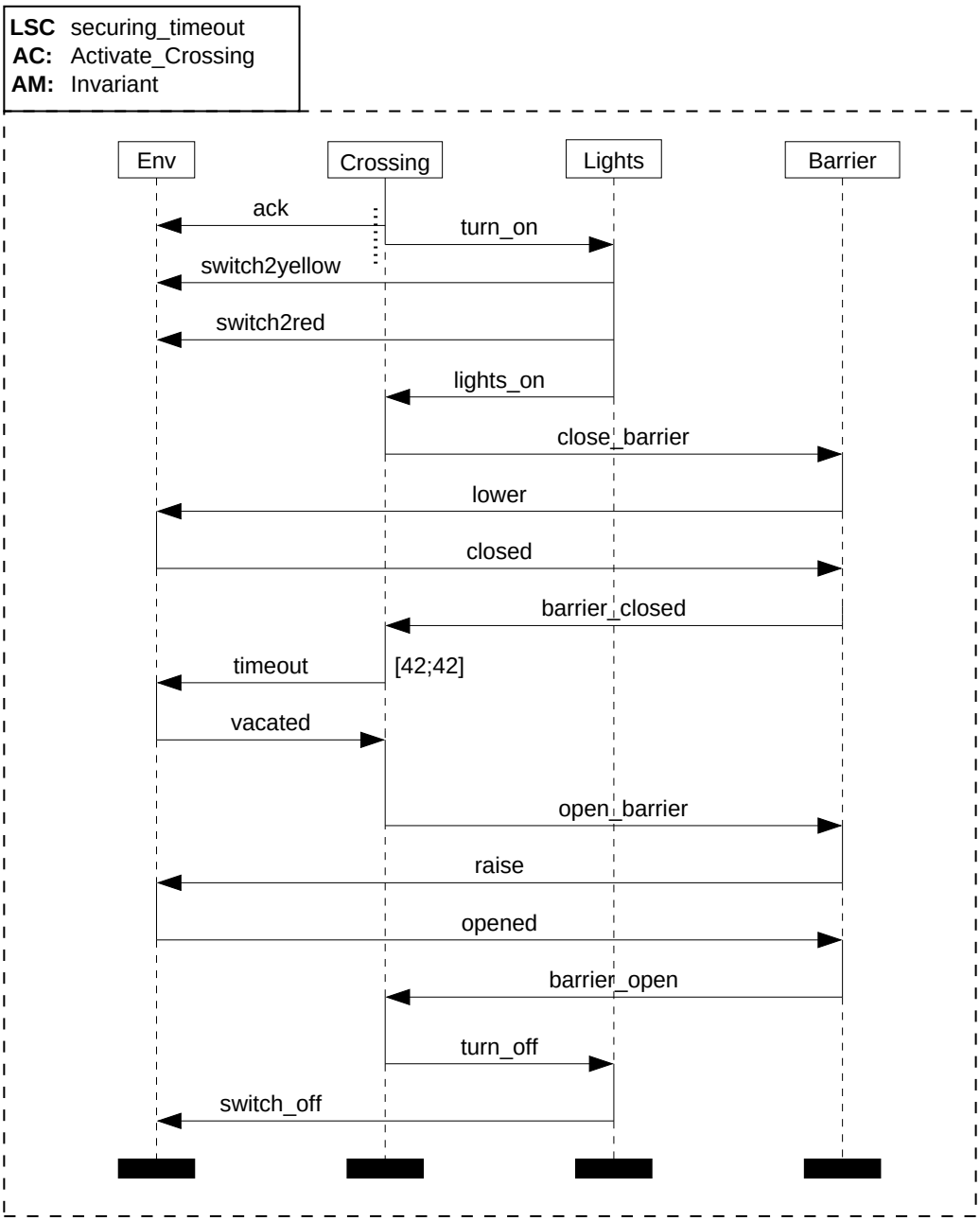


Figure A.41: Existential LSC for exceeding of the maximum barrier closed time with timing constraint

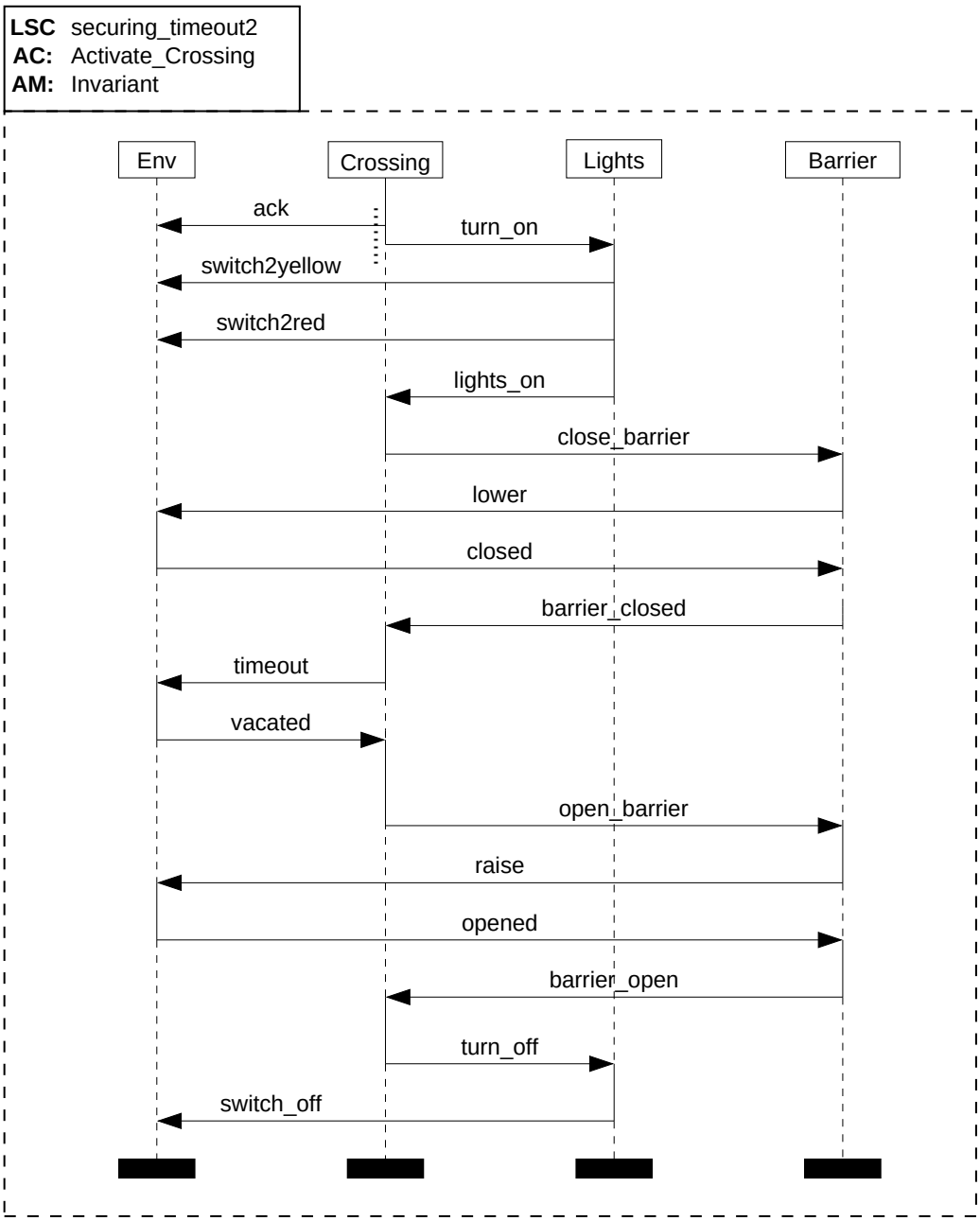


Figure A.42: Existential LSC for exceedance of the maximum barrier closed time without timing constraint

A.3.2 Universal LSCs

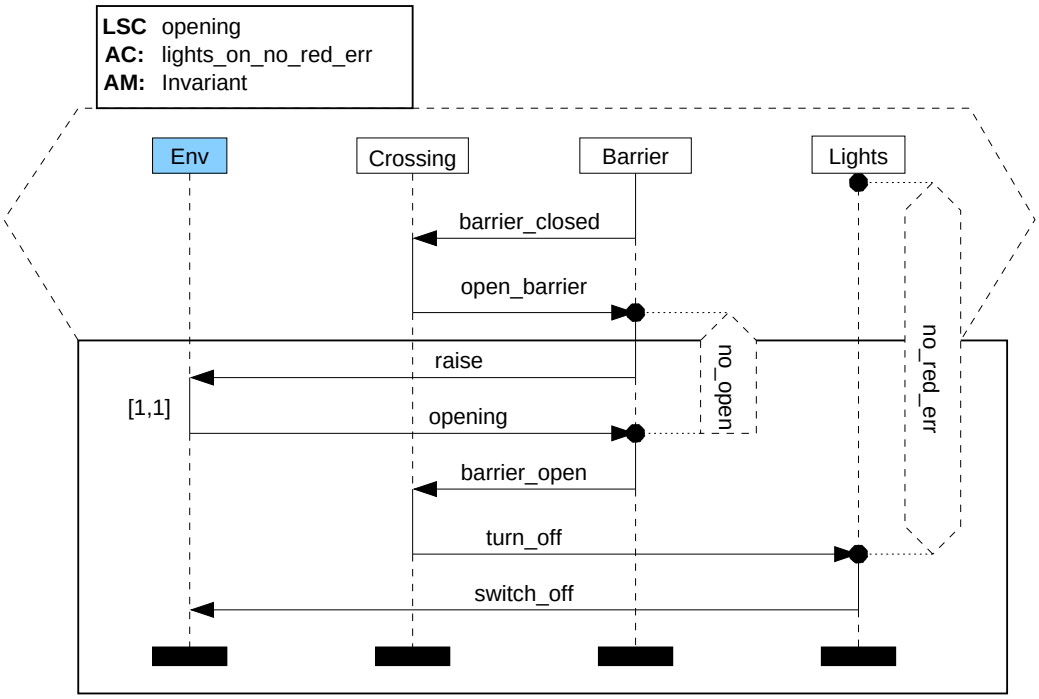
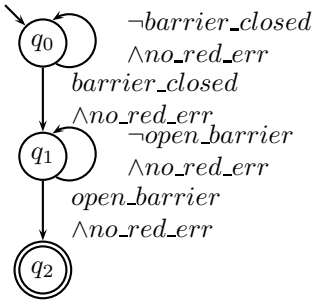
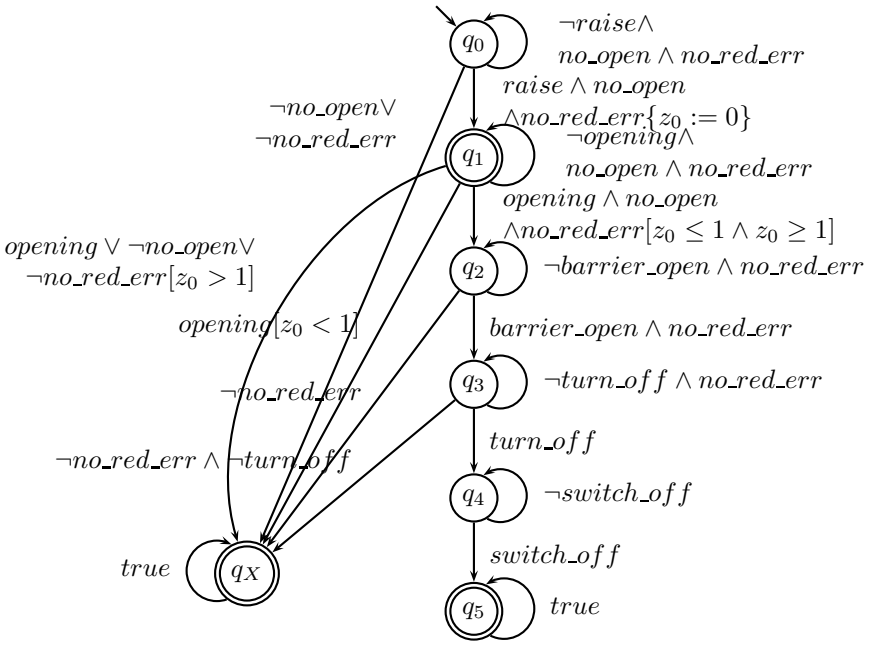


Figure A.43: Universal LSC for the opening of the barrier



(a) Finite automaton
for pre-chart



(b) TSA for LSC body

Figure A.44: Automata for LSC `opening` (weak interpretation)

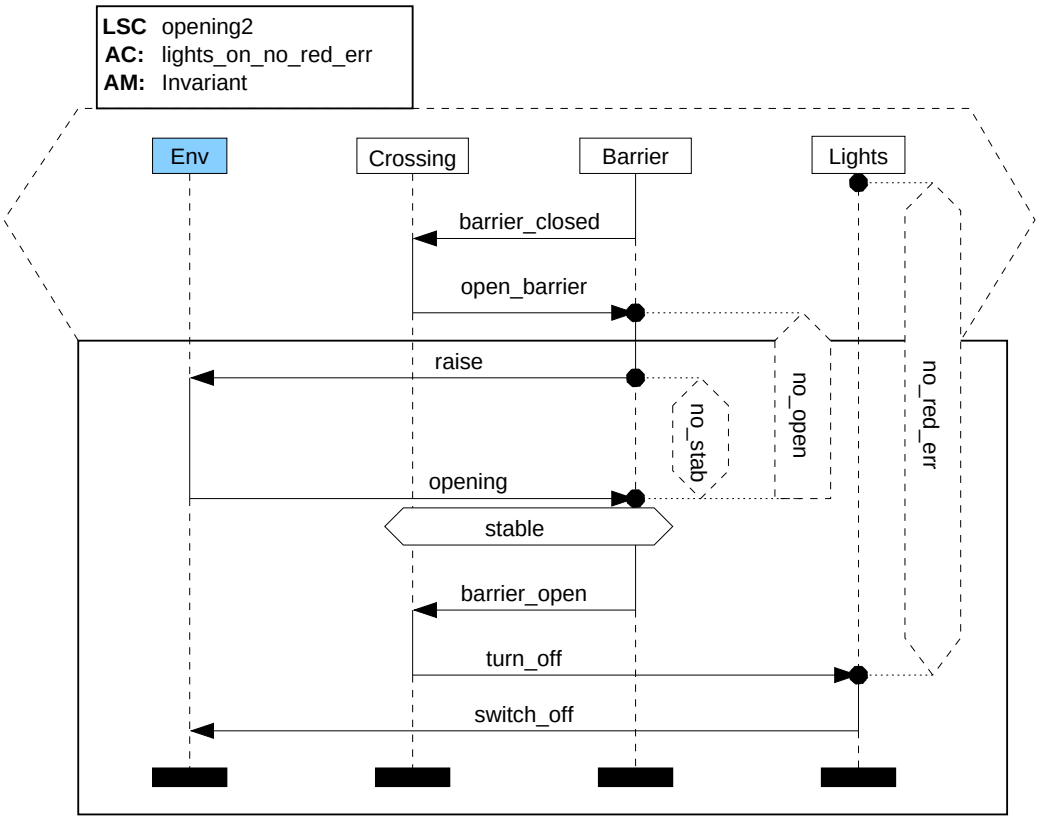
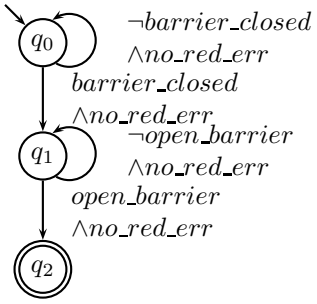
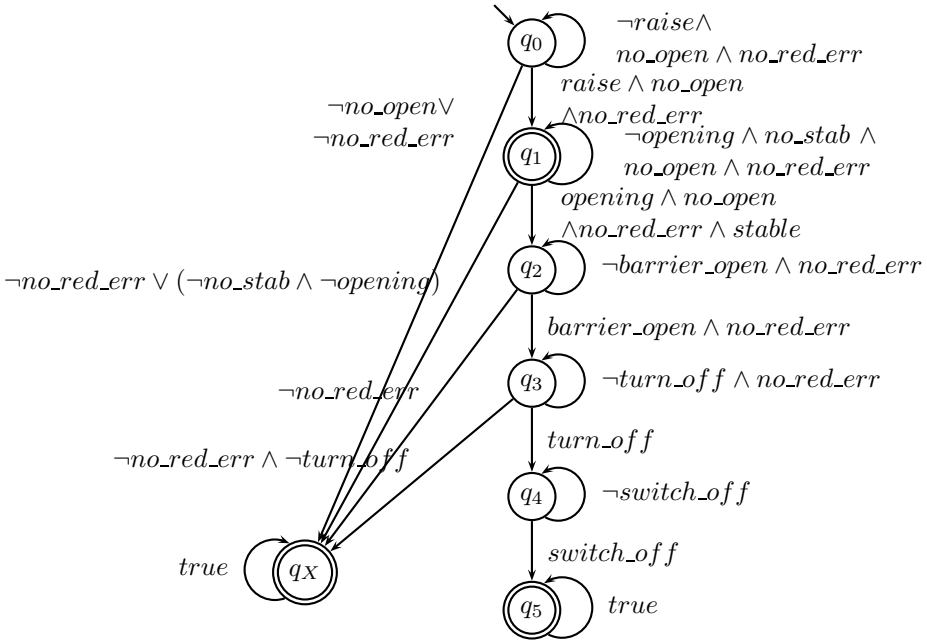


Figure A.45: Universal LSC for the opening of the barrier (superstep semantics, explicit enumeration)



(a) Finite automaton
for pre-chart



(b) TSA for LSC body

Figure A.46: Automata for LSC `opening2` (weak interpretation, superstep semantics, explicit enumeration)

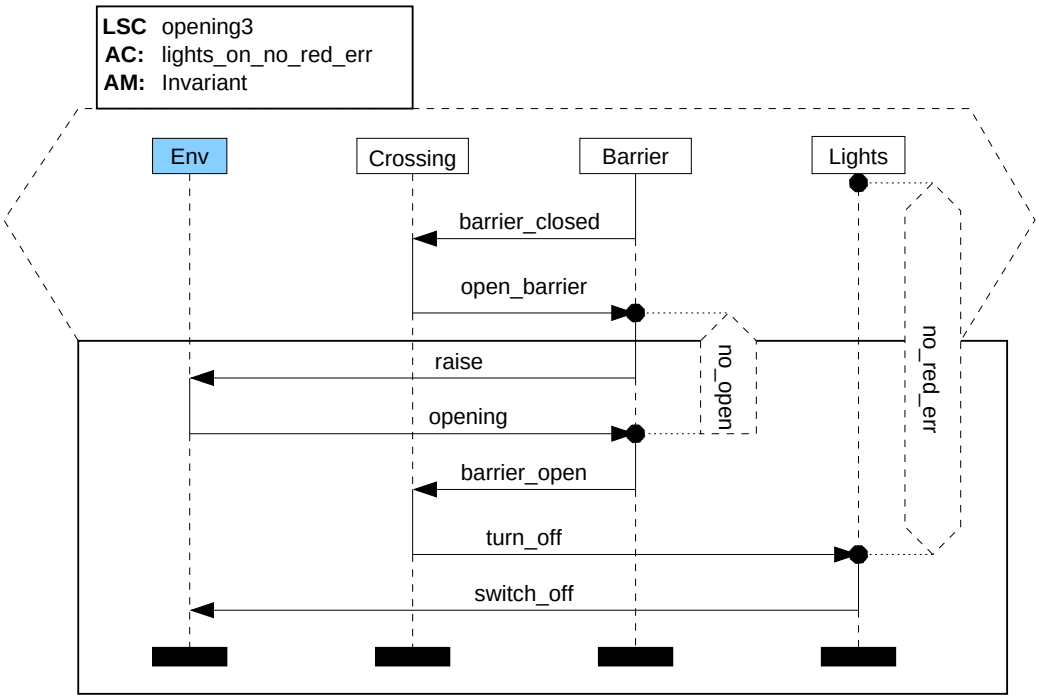
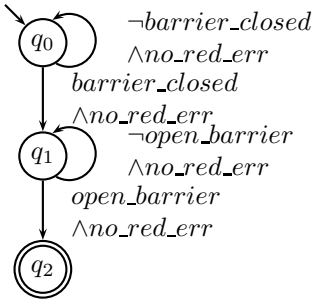
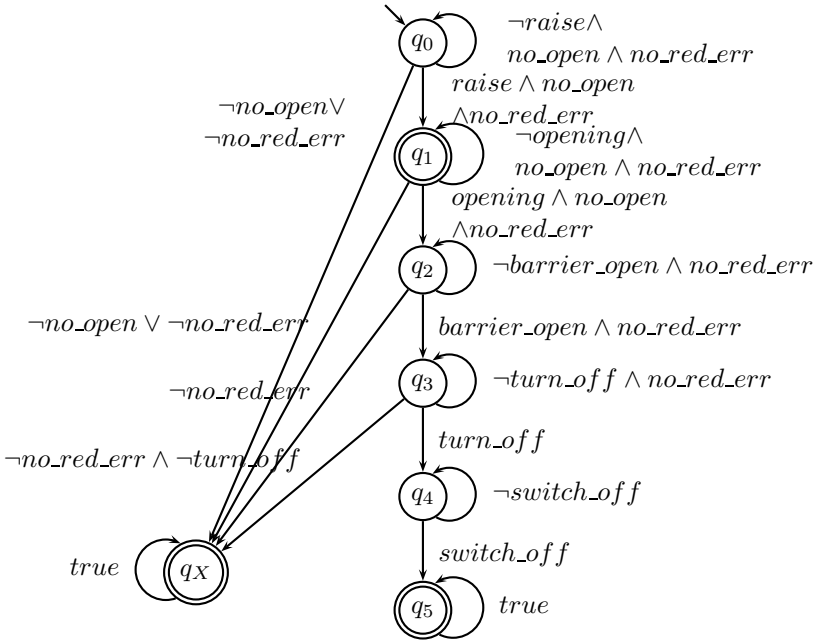


Figure A.47: Universal LSC for the opening of the barrier (superstep semantics, user-specified assumption)



(a) Finite automaton
for pre-chart



(b) TSA for LSC body

Figure A.48: Automata for LSC `opening3` (weak interpretation, superstep semantics)

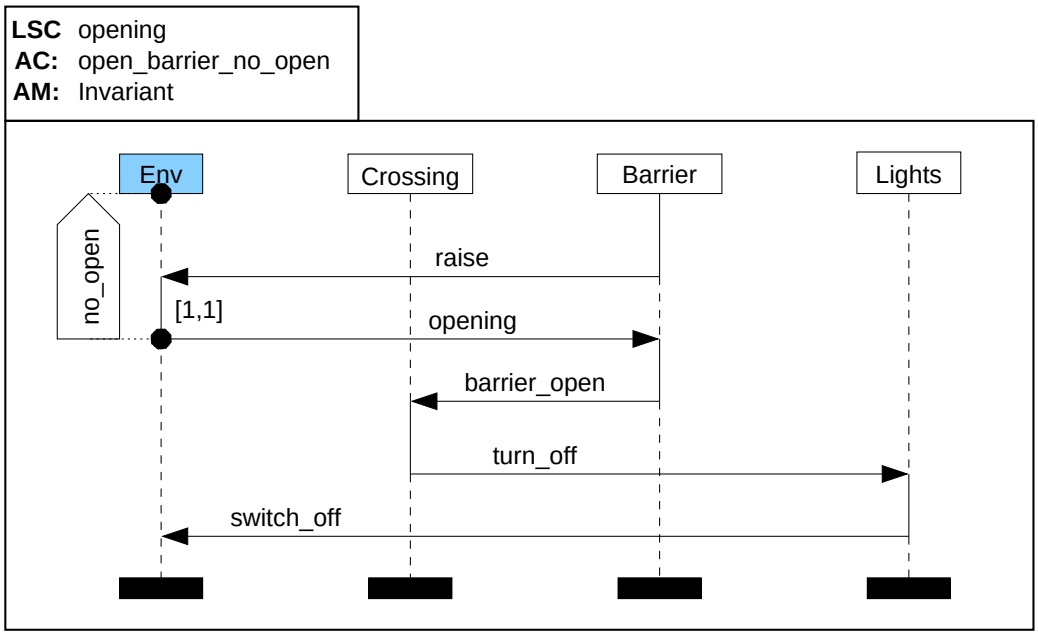


Figure A.49: Alternative universal LSC for the opening of the barrier without pre-chart

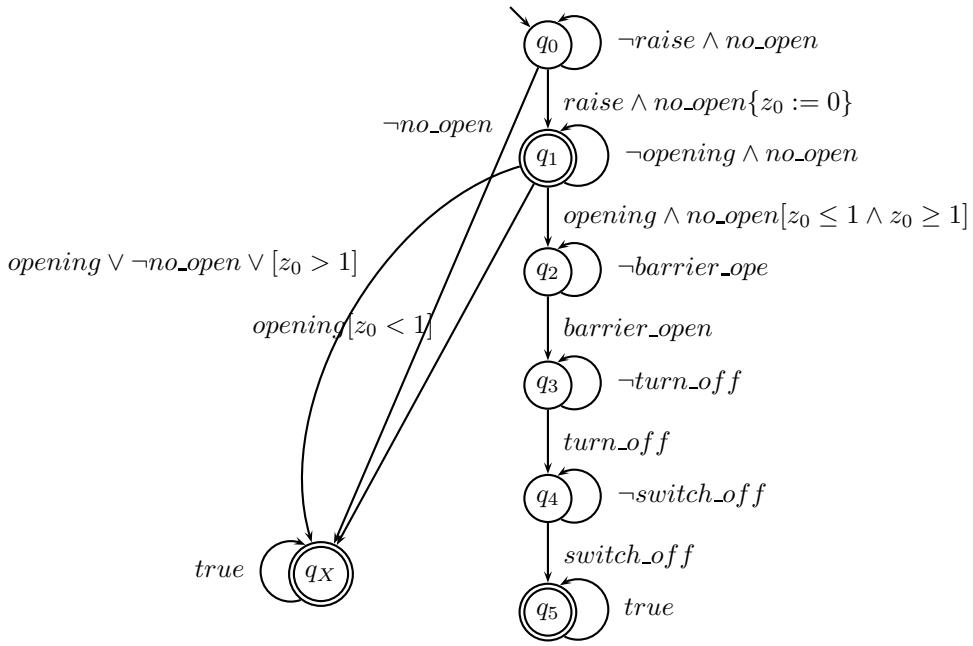


Figure A.50: Automata for LSC `opening` without pre-chart (weak interpretation)

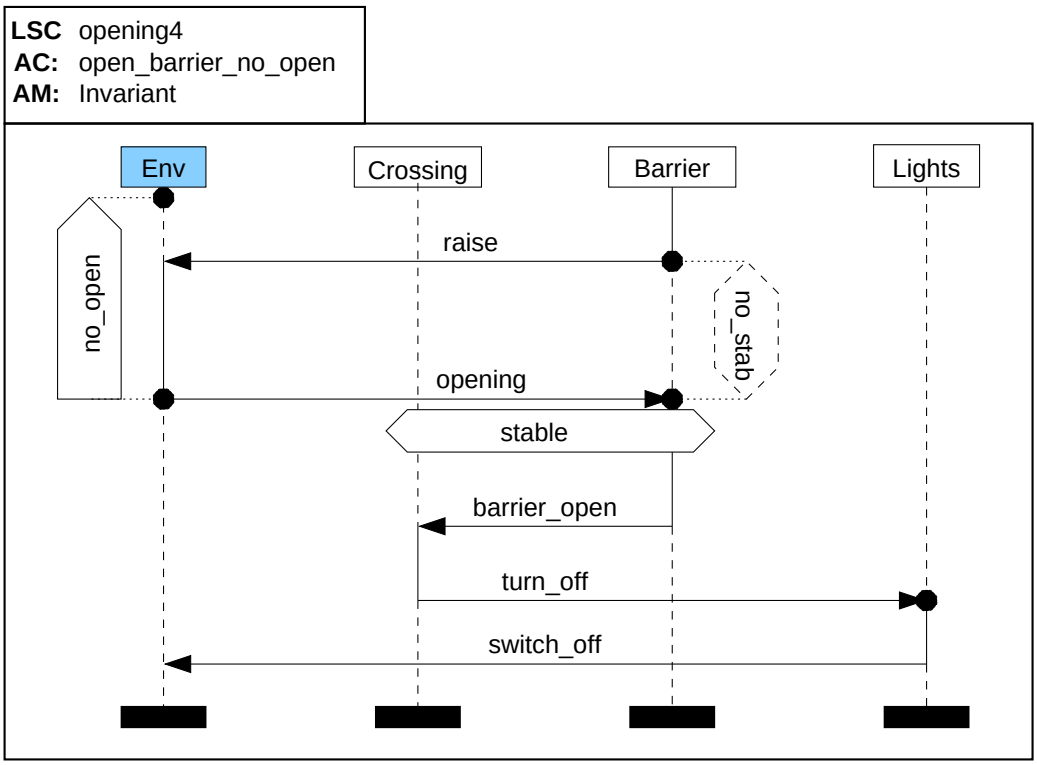


Figure A.51: Alternative universal LSC for the opening of the barrier without pre-chart (superstep semantics, explicit enumeration)

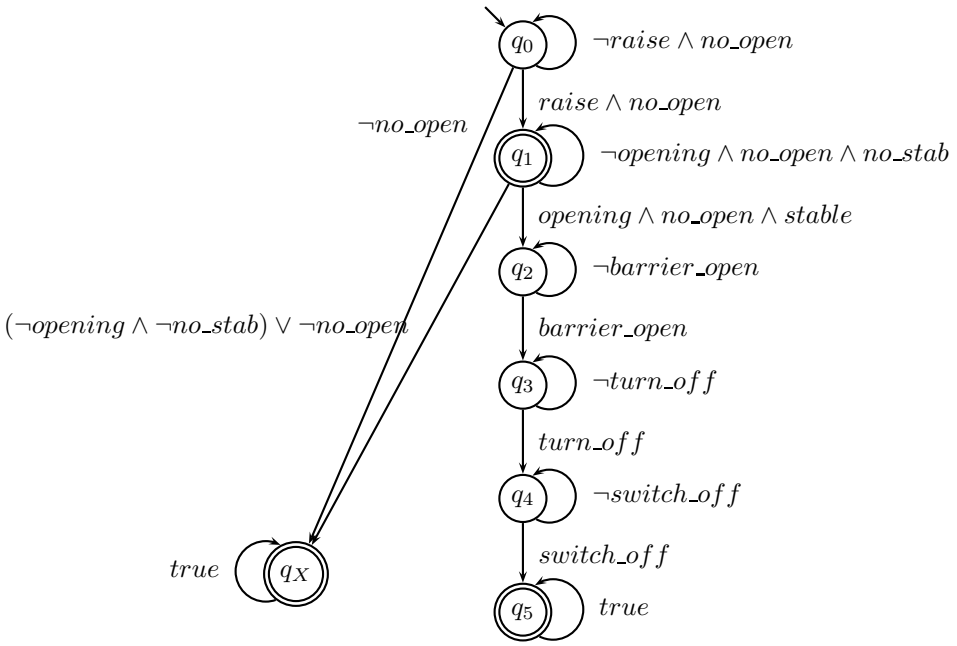


Figure A.52: Automata for LSC `opening4` without pre-chart (weak interpretation, superstep semantics)

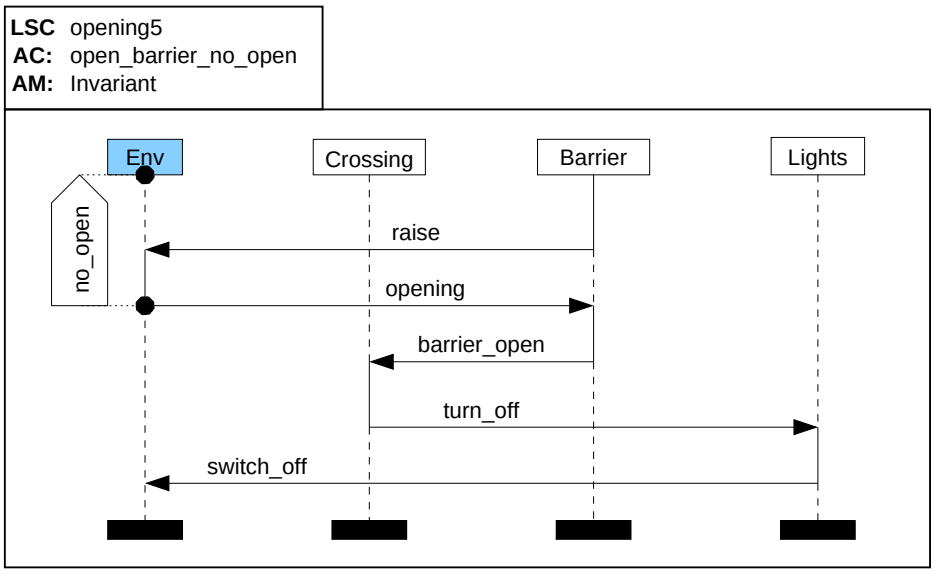


Figure A.53: Alternative universal LSC for the opening of the barrier without pre-chart (superstep semantics, user-specified assumption)

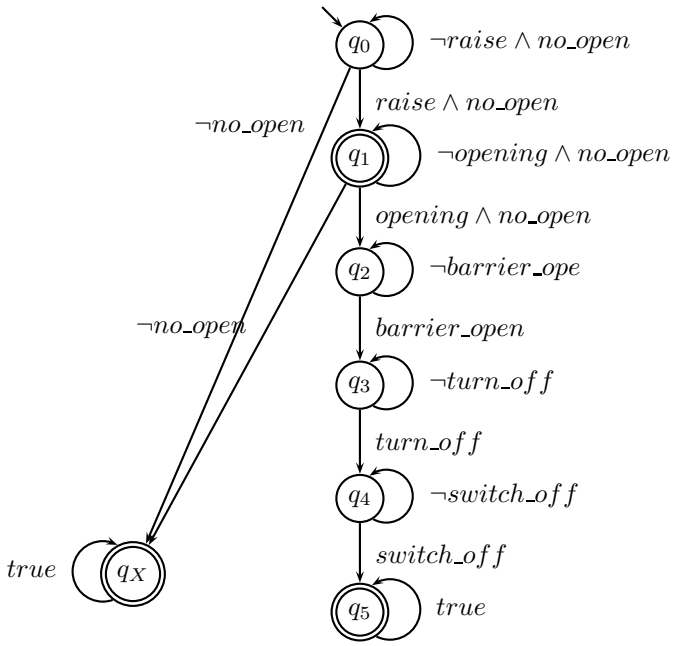


Figure A.54: Automata for LSC `opening5` without pre-chart (weak interpretation, superstep semantics)

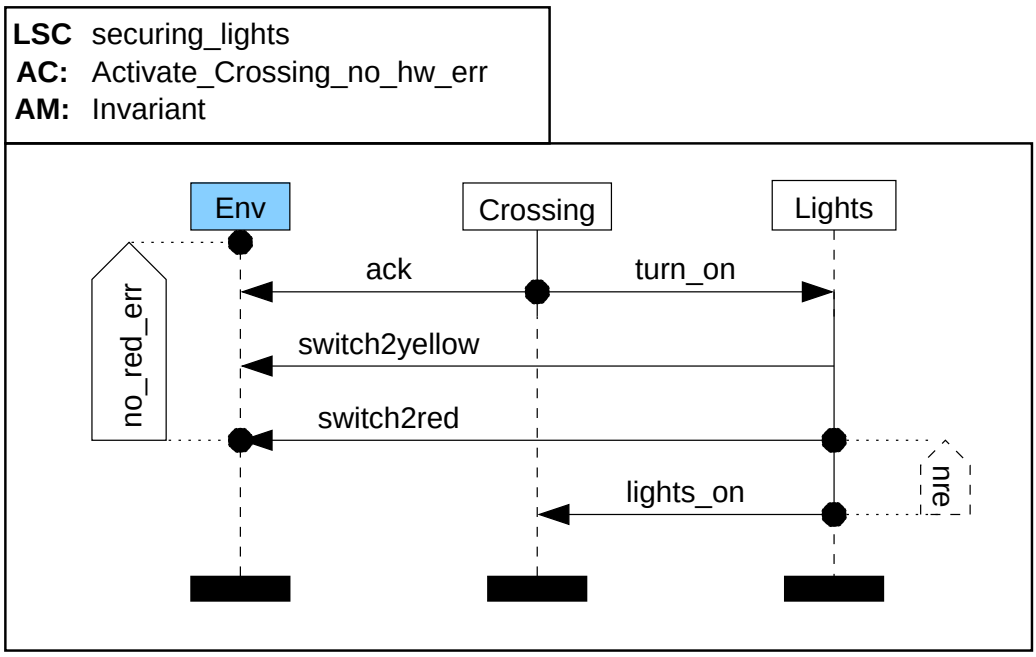


Figure A.55: Universal LSC for switching on the traffic lights

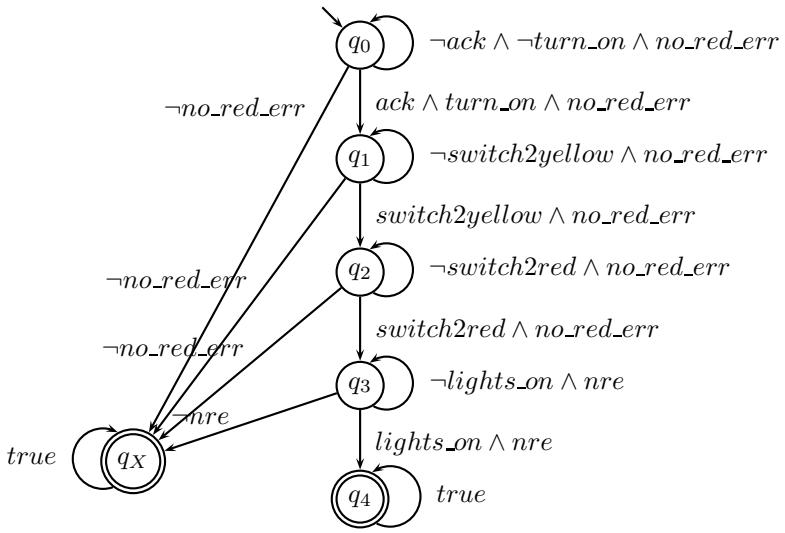


Figure A.56: TSA for LSC body of `securing_lights` (weak interpretation)

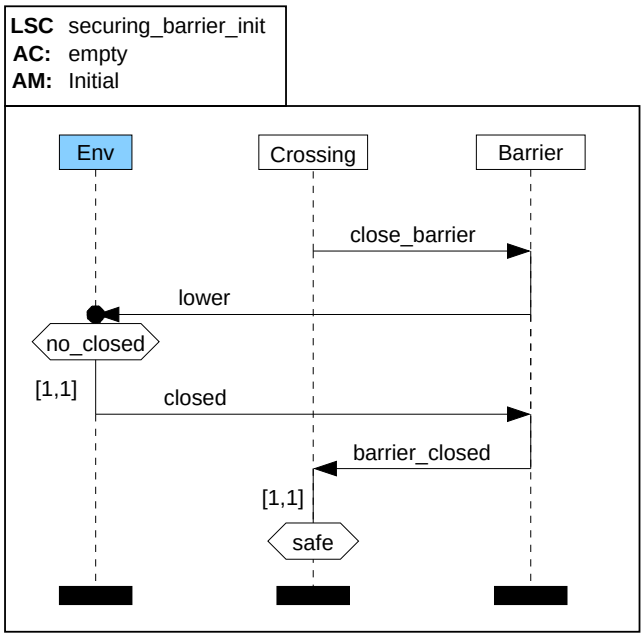


Figure A.57: Initial universal LSC for the closing of the barrier

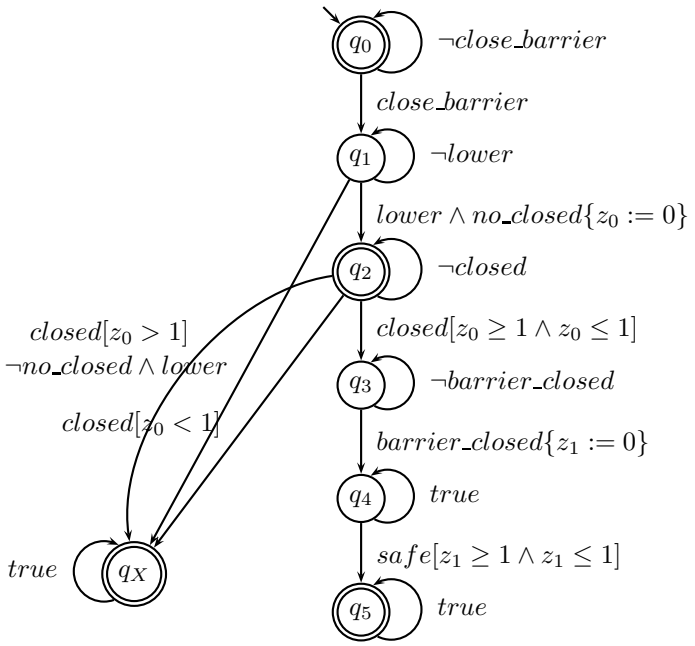


Figure A.58: TSA for LSC body of `securing_barrier_init` (weak interpretation)

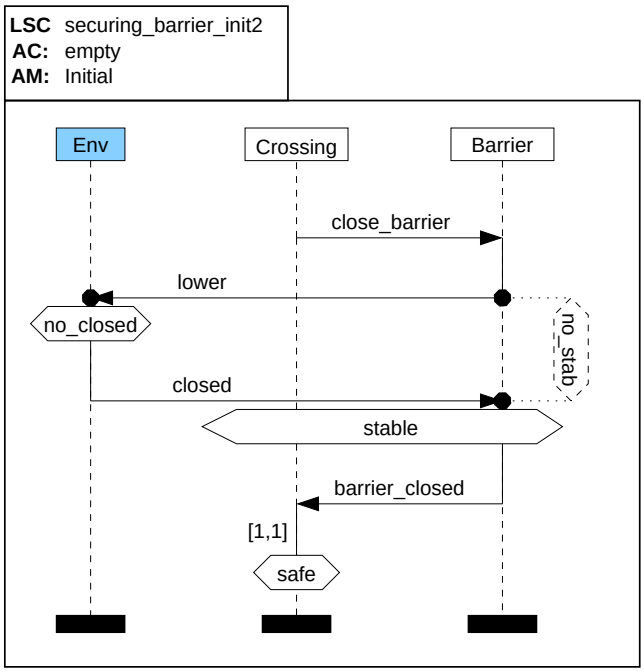


Figure A.59: Initial universal LSC for the closing of the barrier (superstep semantics, explicit enumeration)

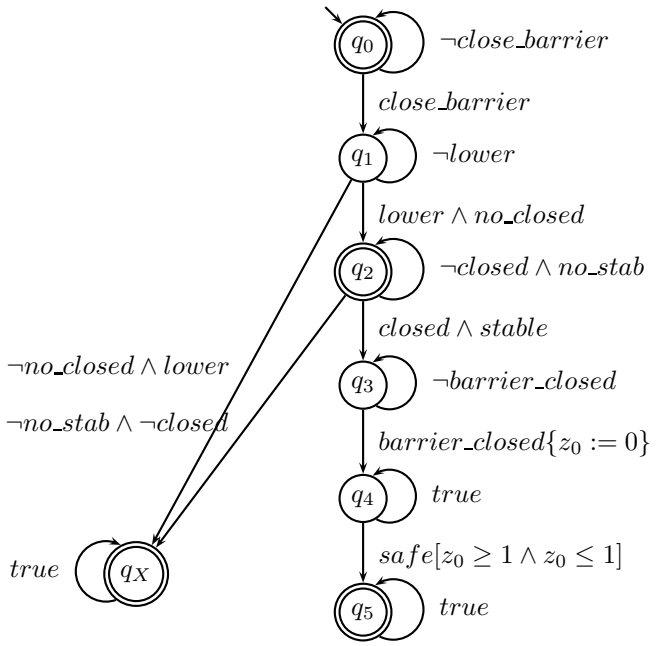


Figure A.60: TSA for LSC body of `securing_barrier_init2` (weak interpretation, superstep semantics)

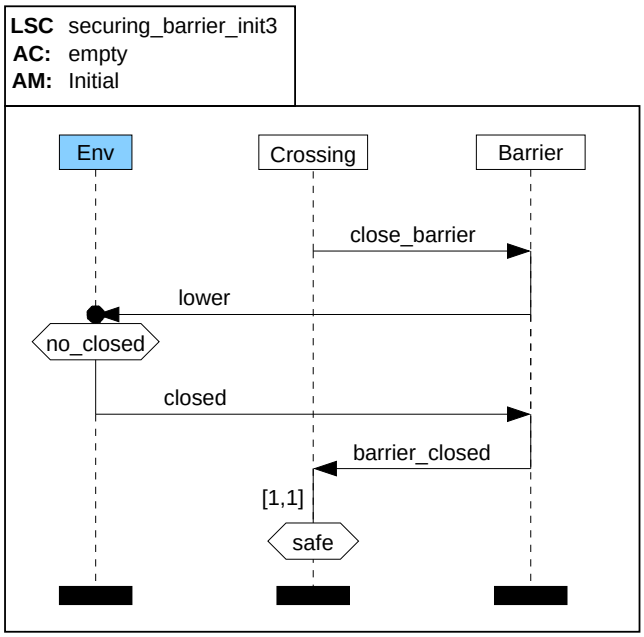


Figure A.61: Initial universal LSC for the closing of the barrier

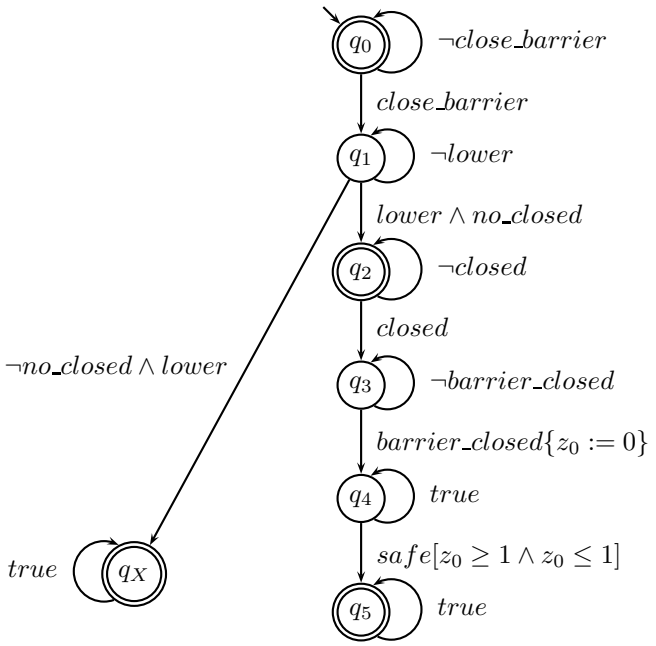


Figure A.62: TSA for LSC body of `securing_barrier_init` (weak interpretation)

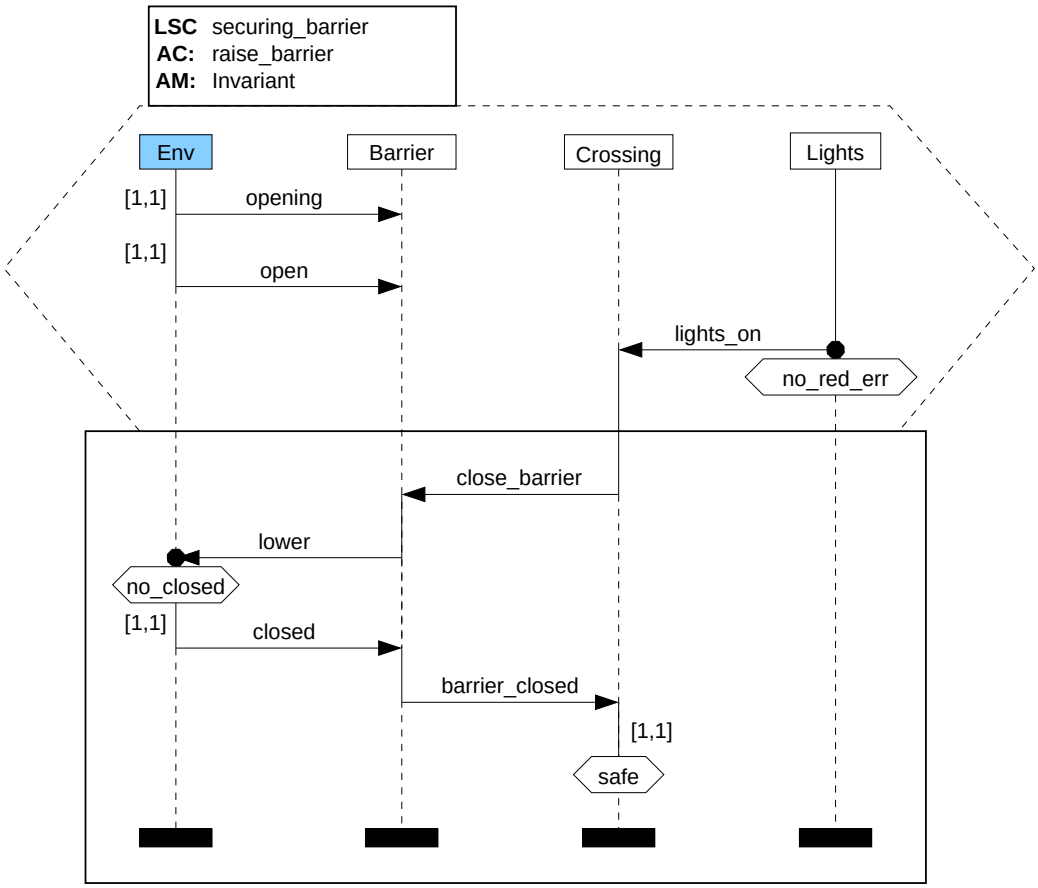


Figure A.63: Universal LSC for the closing of the barrier

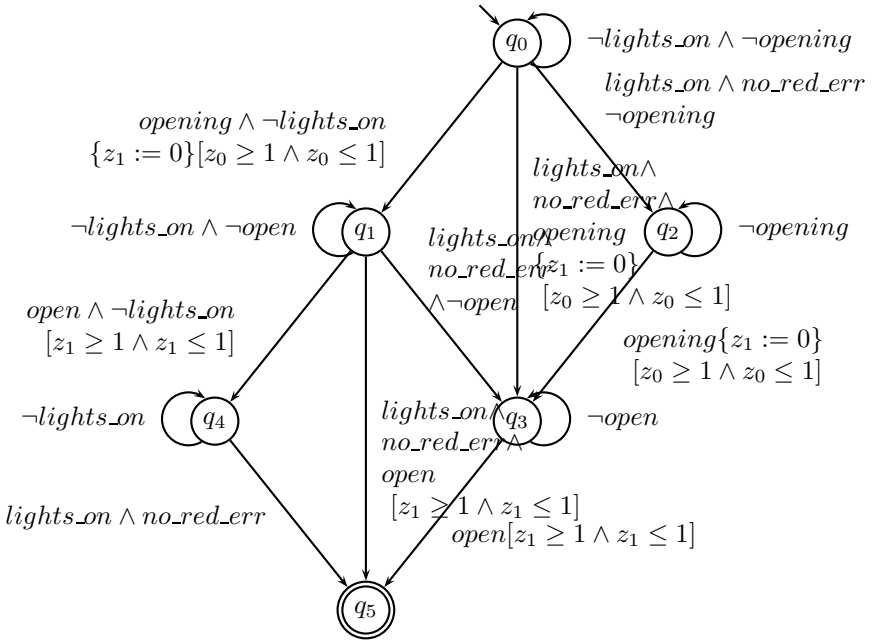


Figure A.64: Finite automaton for pre-chart of LSC `securing_barrier` (weak interpretation)

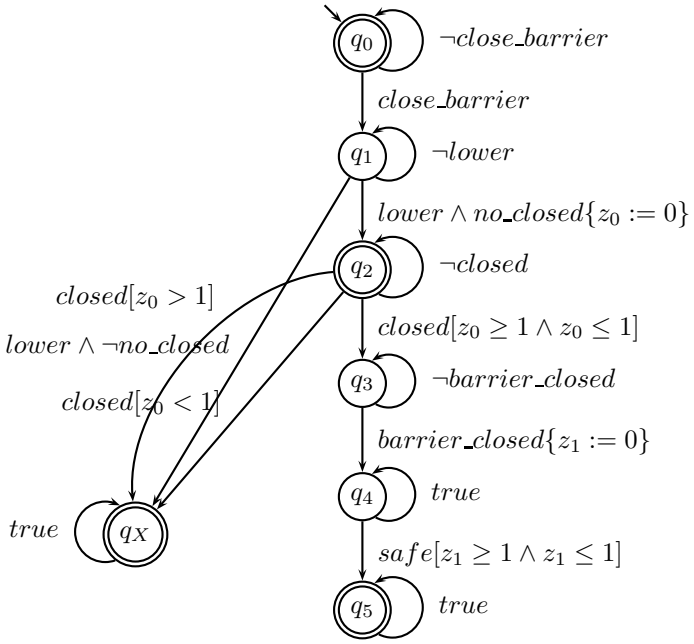


Figure A.65: TSA for LSC body of `securing_barrier` (weak interpretation)

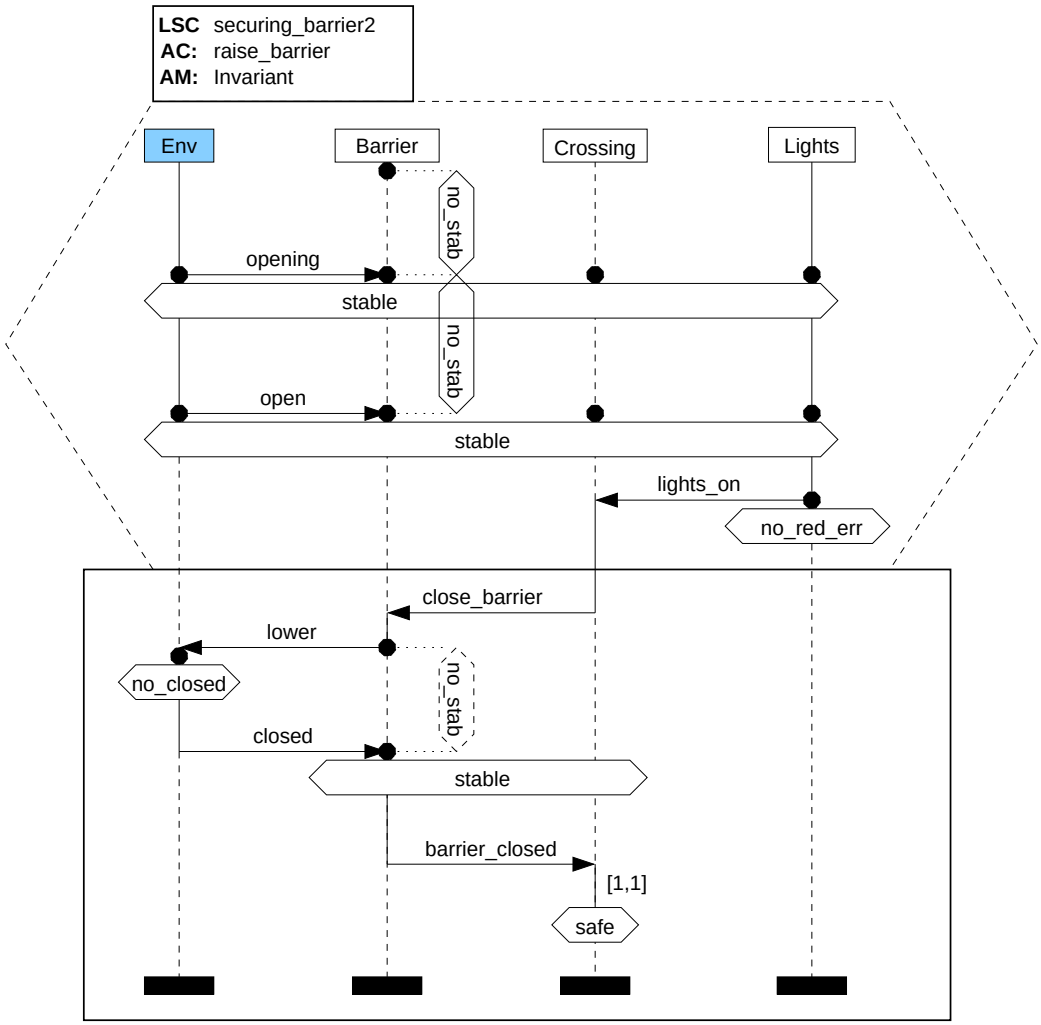


Figure A.66: Universal LSC for the closing of the barrier (superstep semantics, explicit enumeration)

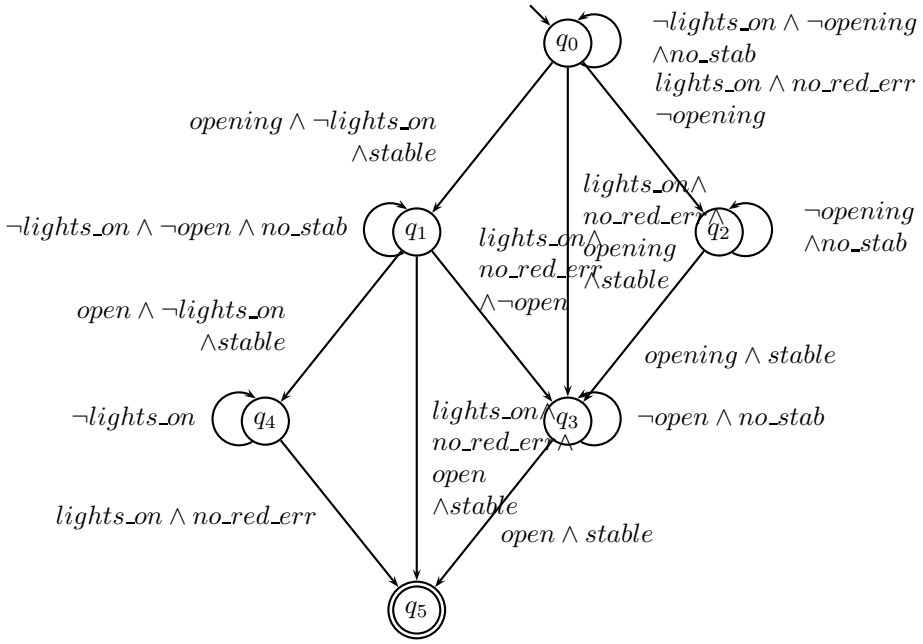


Figure A.67: Finite automaton for pre-chart of LSC `securing_barrier2` (weak interpretation, superstep semantics)

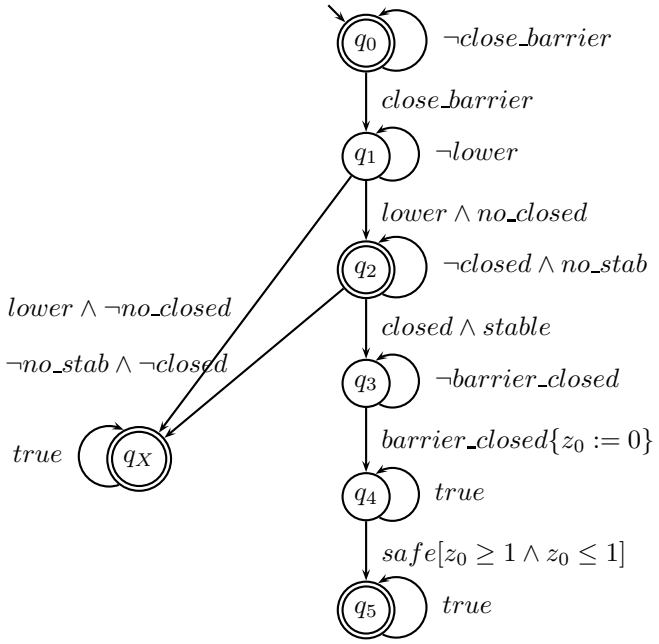


Figure A.68: TSA for LSC body `securing_barrier2` (weak interpretation, superstep semantics)

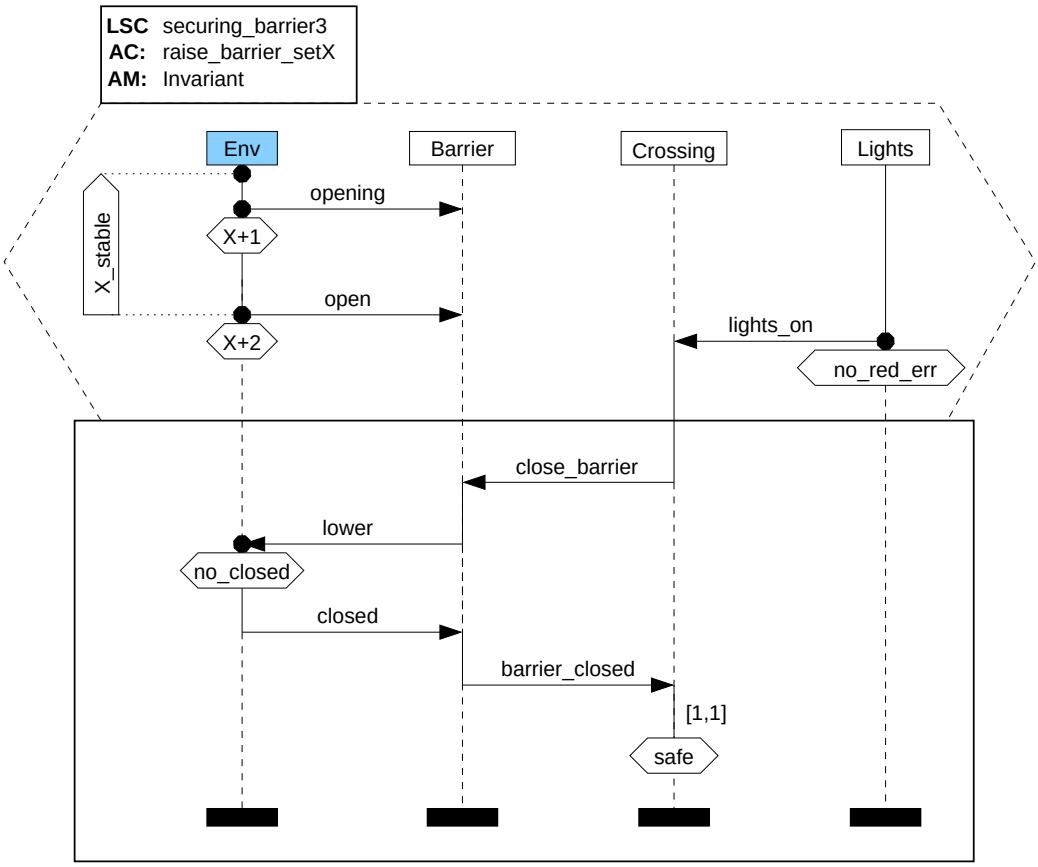


Figure A.69: Universal LSC for the closing of the barrier (superstep semantics, using counter and user-specified assumption)

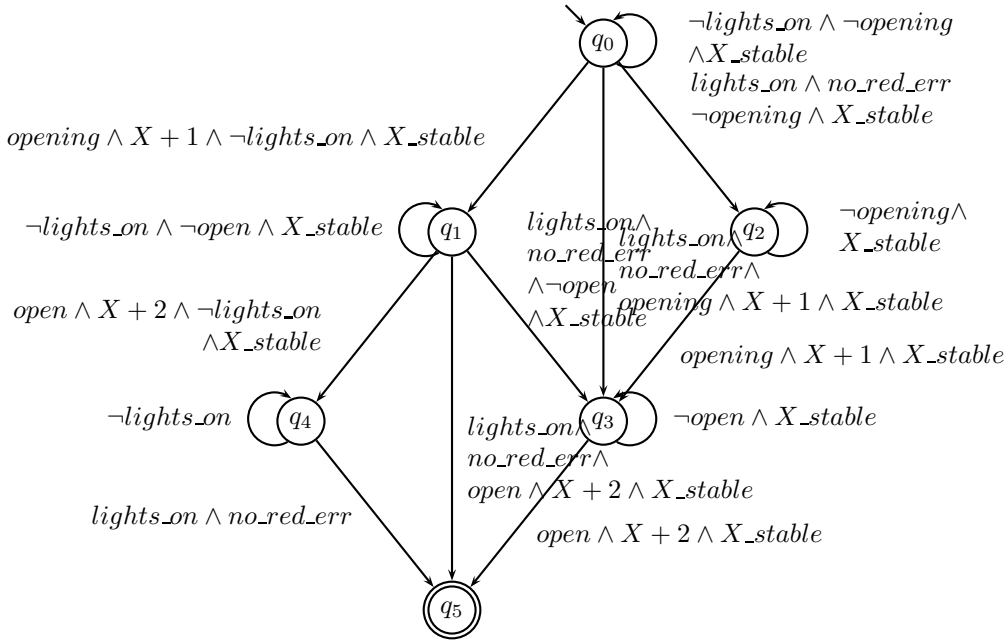


Figure A.70: Finite automaton for pre-chart of LSC `securing_barrier3` (weak interpretation, superstep semantics)

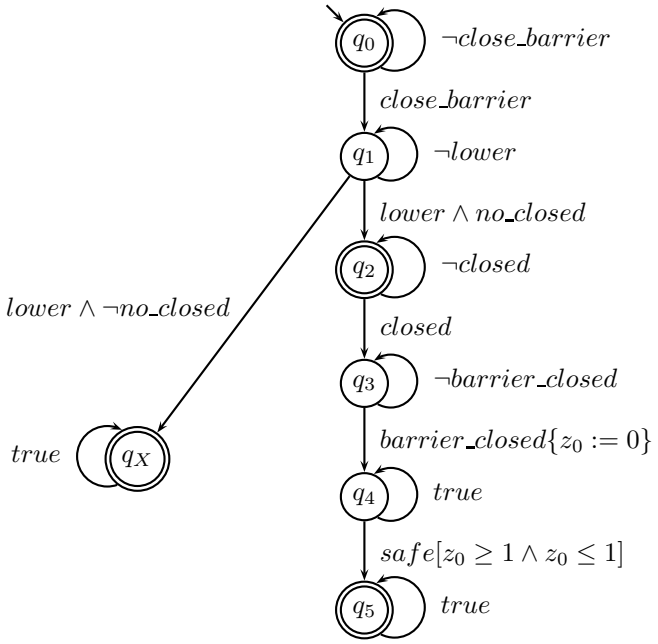


Figure A.71: TSA for LSC body `securing_barrier3` (weak interpretation, superstep semantics)

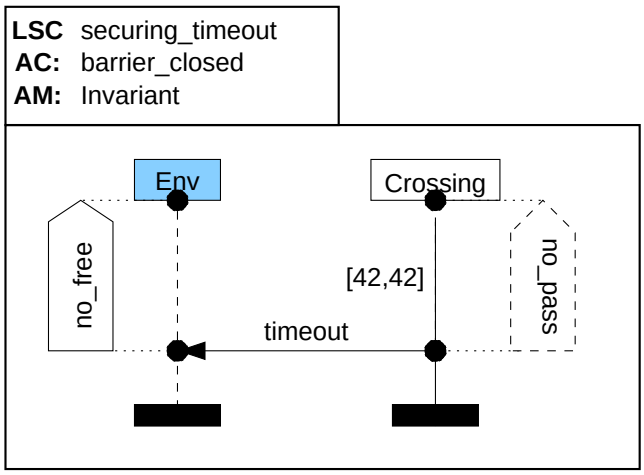


Figure A.72: Universal LSC for exceedance of the maximum barrier closed time

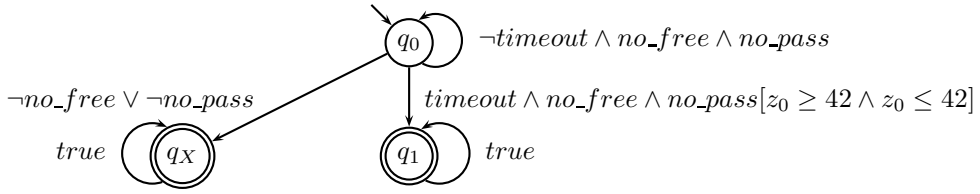


Figure A.73: TSA for LSC body of `securing_timeout` (weak and strict interpretation)

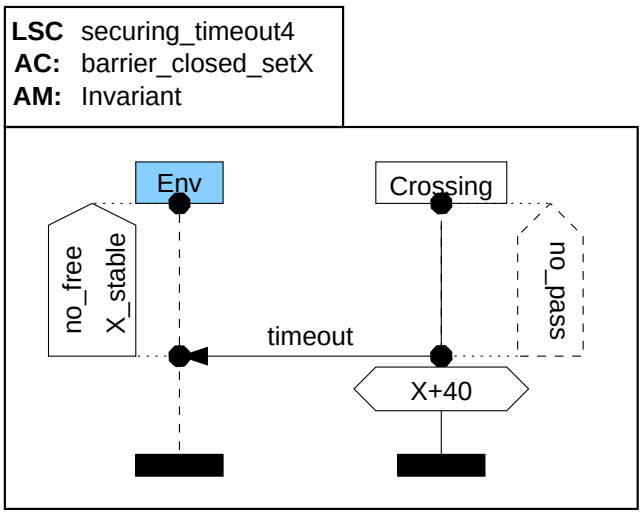


Figure A.74: Universal LSC for exceedance of the maximum barrier closed time (superstep semantics, counter)

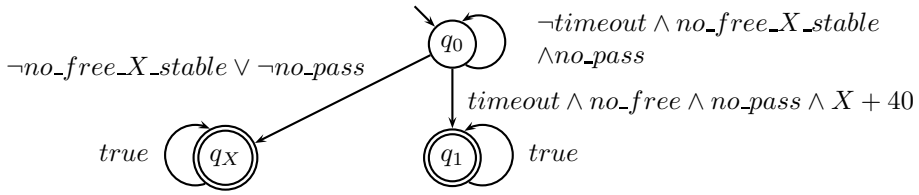


Figure A.75: TSA for LSC body of `securing_timeout4` (weak and strict interpretation, superstep semantics)

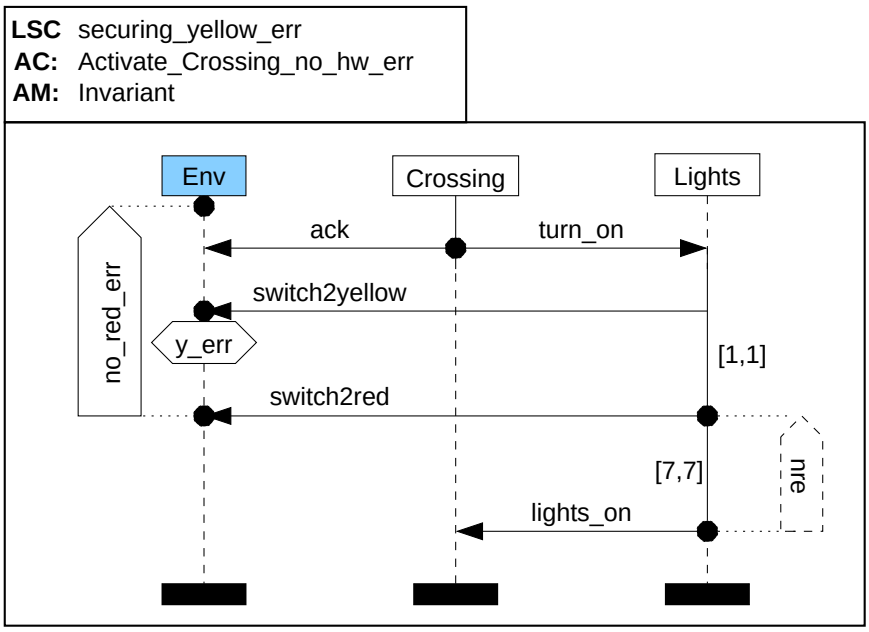


Figure A.76: Universal LSC for a defect yellow light with timing constraints

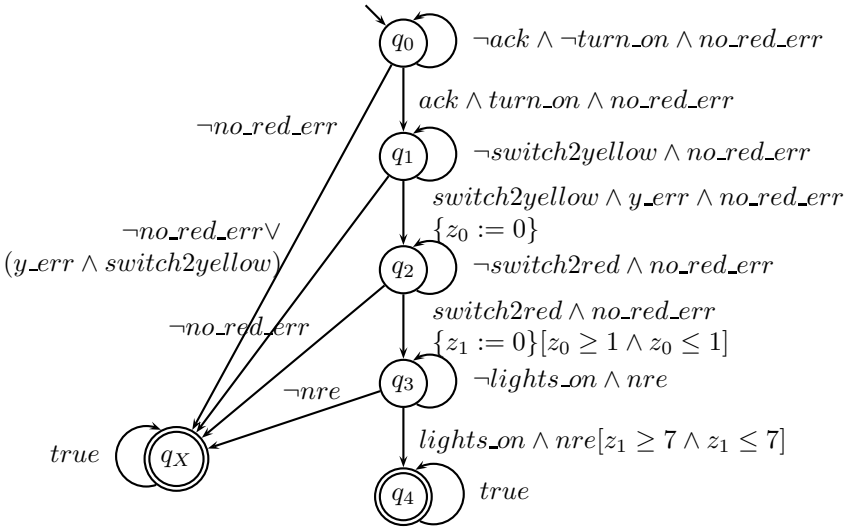


Figure A.77: TSA for LSC body of `securing_yellow_err` (weak interpretation)

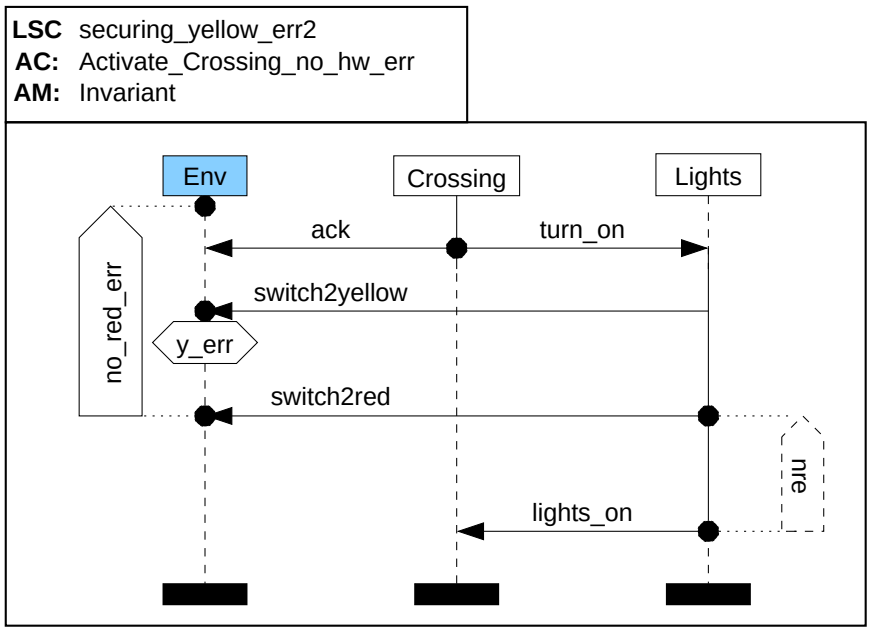


Figure A.78: Universal LSC for a defect yellow light without timing constraints

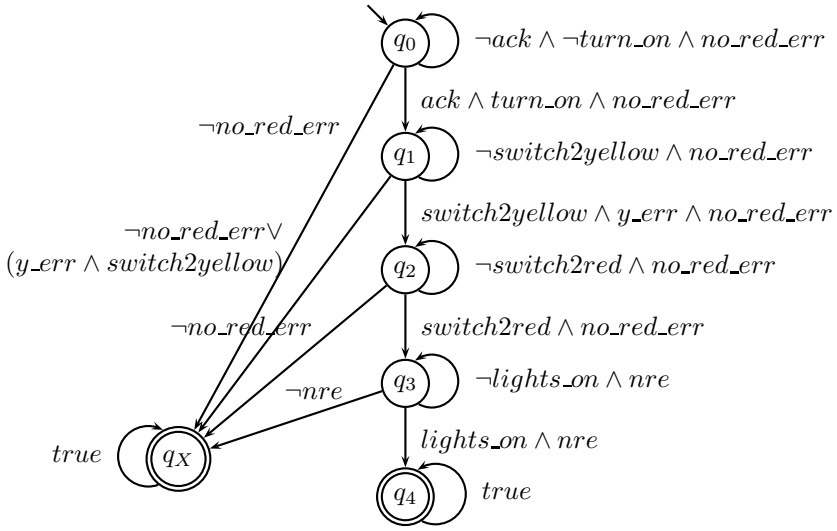


Figure A.79: TSA for LSC body of `securing_yellow_err2` (weak interpretation)

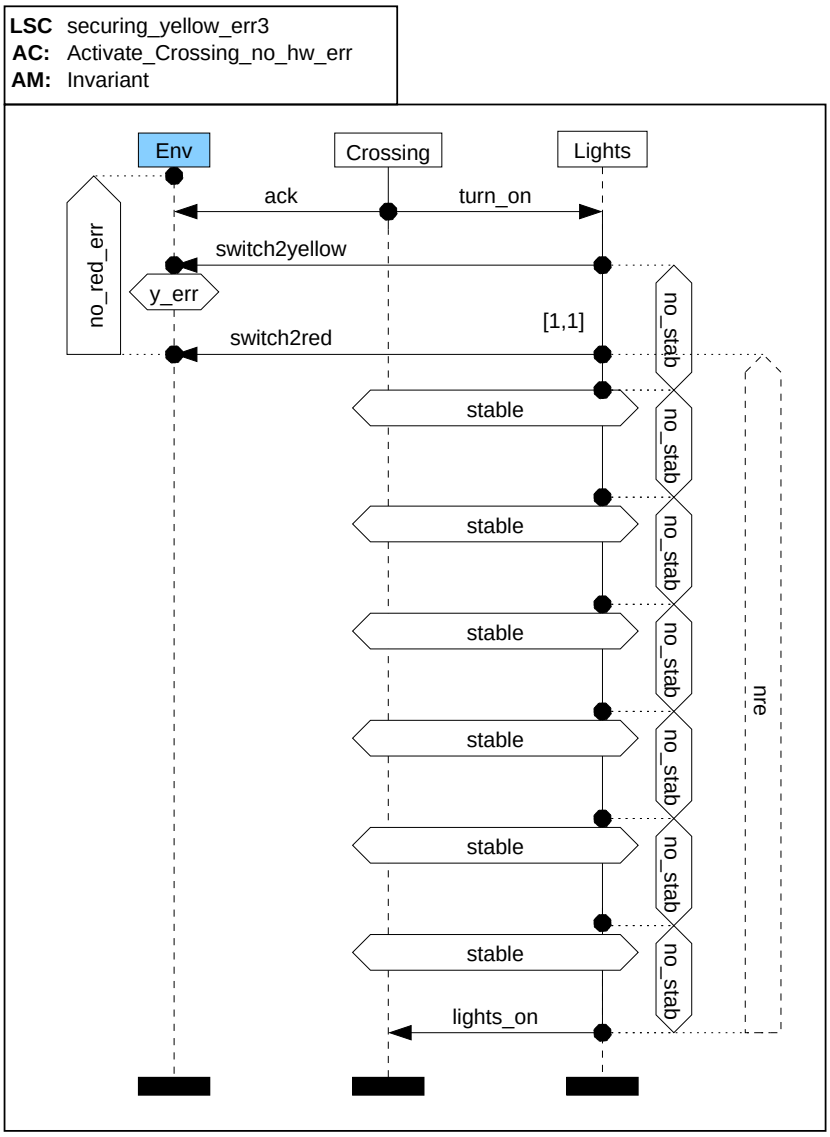


Figure A.80: Universal LSC for a defect yellow light (superstep semantics, explicit enumeration)

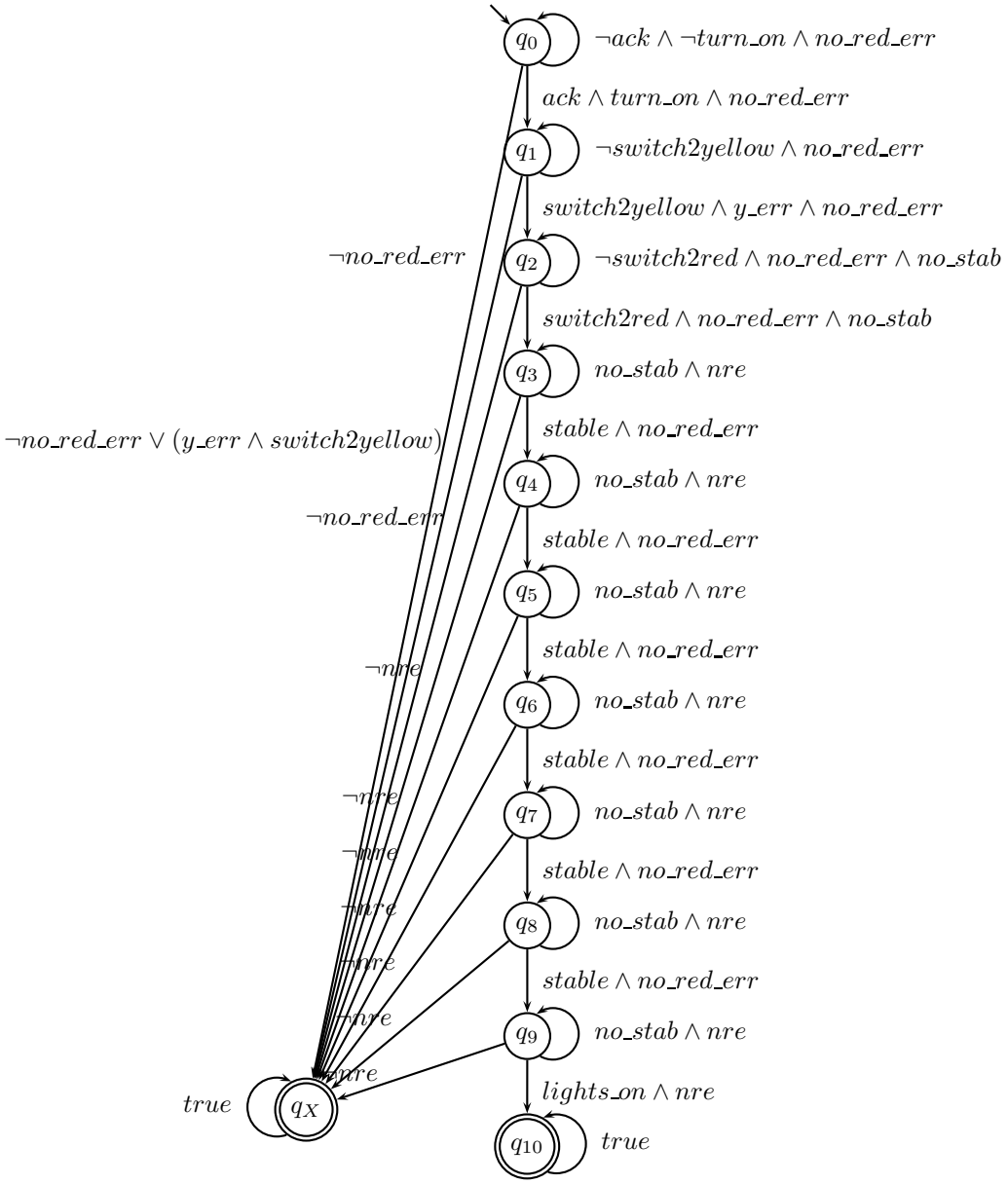


Figure A.81: TSA for LSC body of `securing_yellow_err3` (weak interpretation, superstep semantics)

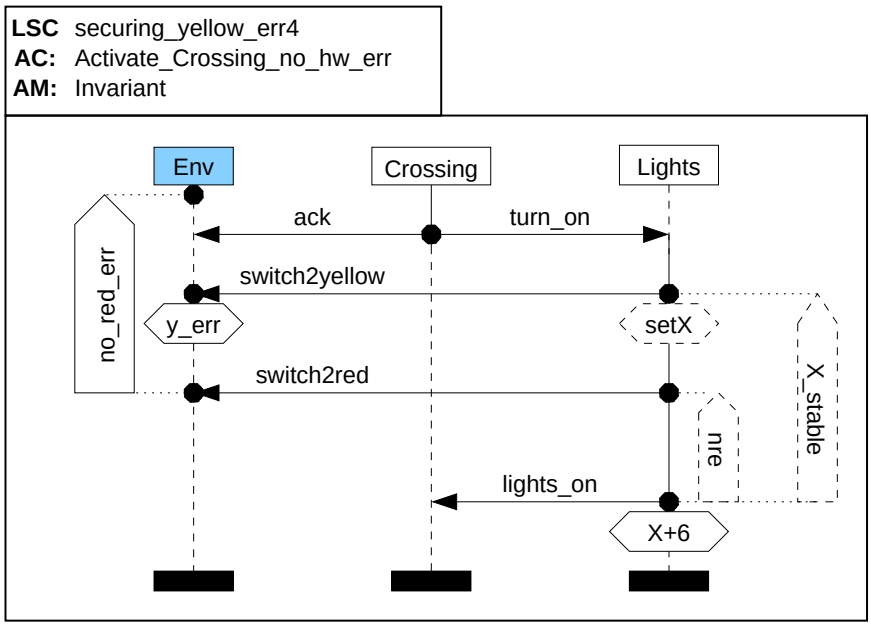


Figure A.82: Universal LSC for a defect yellow light(superstep semantics, explicit enumeration)

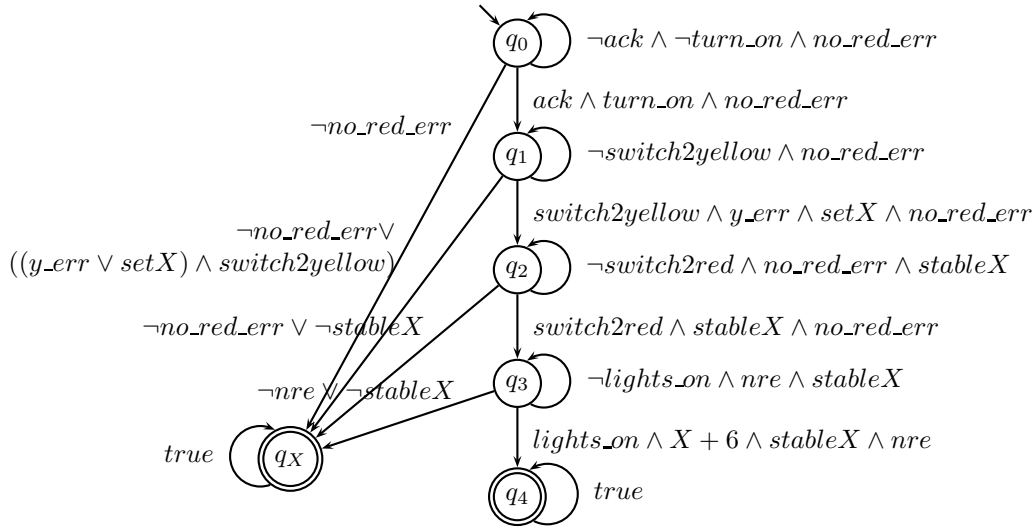


Figure A.83: TSA for LSC body of `securing_yellow_err2` (weak interpretation, superstep semantics)

A.3.3 Assumption LSCs

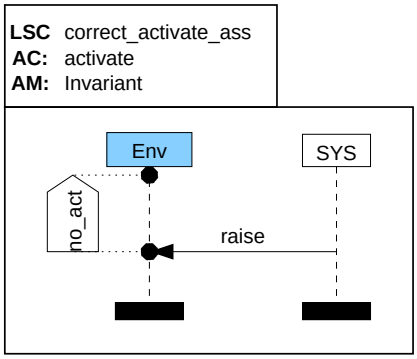


Figure A.84: Assumption LSC for restricting the occurrence of activation messages

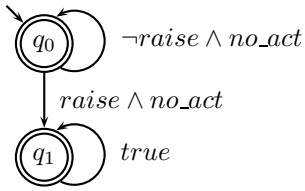


Figure A.85: TSA for LSC body of `correct_activate_ass` (weak and strict interpretation)

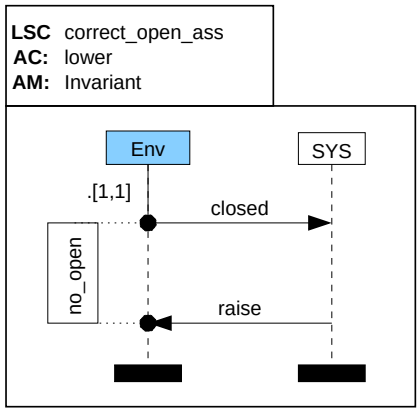


Figure A.86: Assumption LSC for the correct setting of the opened condition

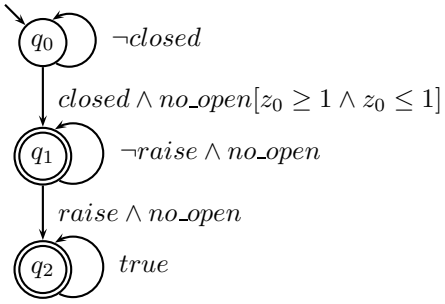


Figure A.87: TSA for LSC body of `correct_open_ass` (weak interpretation, step semantics)

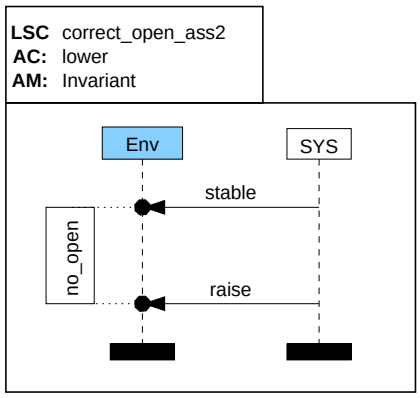


Figure A.88: Assumption LSC for the correct setting of the opened condition

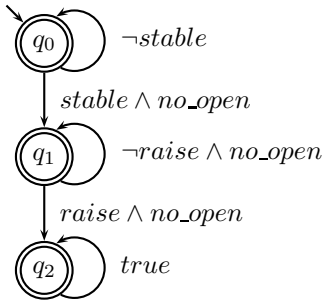


Figure A.89: TSA for LSC body of `correct_open2_ass` (weak interpretation, superstep semantics)

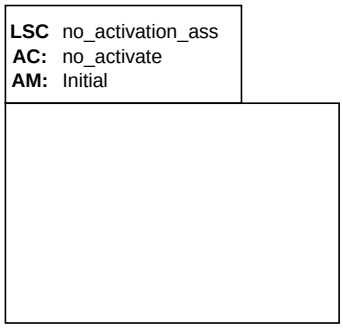


Figure A.90: Assumption LSC forbidding the receipt of an activation request in the initial state

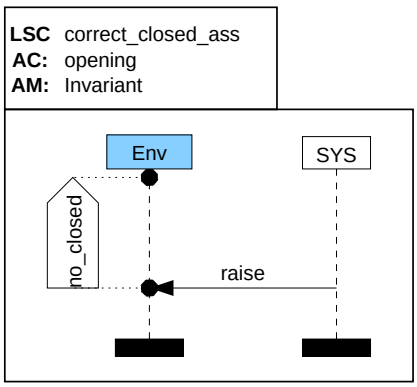


Figure A.91: Assumption LSC for restricting the occurrence of closed

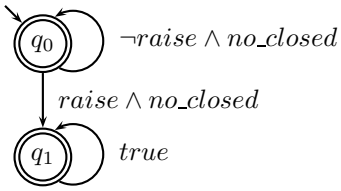


Figure A.92: TSA for LSC body of `correct_closed_ass` (weak and strict interpretation)

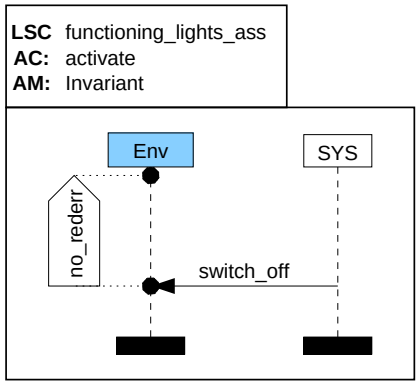


Figure A.93: Assumption LSC for prohibiting a failure of the red light

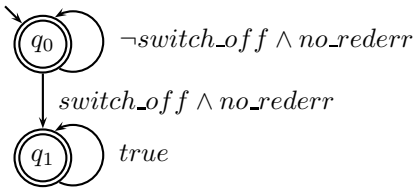


Figure A.94: TSA for LSC body of `functioning_lights_ass` (weak and strict interpretation)

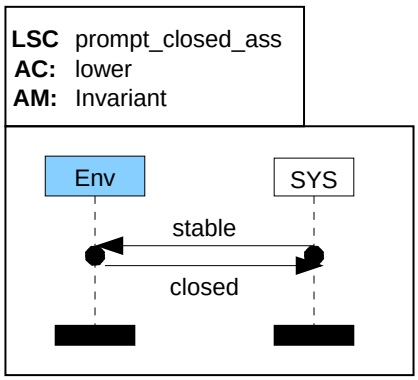


Figure A.95: Assumption LSC enforcing a timely closing of the barriers (superstep semantics)

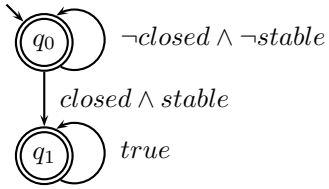


Figure A.96: TSA for LSC body of `prompt_closed_ass` (weak and strict interpretation)

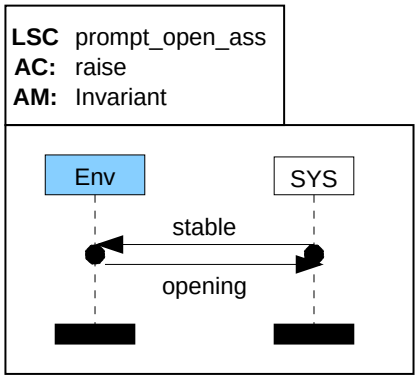


Figure A.97: Assumption LSC enforcing a timely opening of the barriers (superstep semantics)

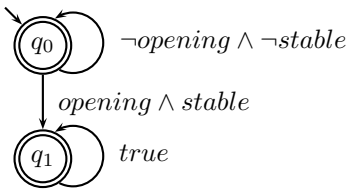


Figure A.98: TSA for LSC body of `prompt_open_ass` (weak and strict interpretation)

Appendix B

Information Flows and Constants of the Statemate Model for the Train Control Application

This chapter provides more detailed information on the STATEMATE model presented in chapter 2. Listed are the values for the constants used in the model and the contents of the information flows grouped according to their respective level of hierarchy.

B.1 Constants

B.2 SYSTEM

T_SEND: TRAIN → COMMUNICATION

Name	Type
ST_COMMUNICATION	Event
SP_COMMUNICATION	Event
ACTIVATE_CROSSING_SND	Event
STATUS_RQ_SND	Event
CROSSING_FREE_SND	Event

Data item	Meaning	Value
CCT	crossing closing time	8
ELT	establish lag time	1
MBCT	maximum barrier closed time	40
MCT	maximum close time	3
MGT	minimum green time	4
MOT	maximum open time	3
MRTC	minimum red time closing	4
MYT	minimum yellow time	2
TRAIN_D.LENGTH	train length	16
TRAIN_D.MAX_DEC	maximum deceleration of train	8
TRAIN_D.MAX_ACC	maximum acceleration of train	8
TRAIN_D.MIN_SPD	minimum speed of train	0
TRAIN_D.MAX_SPD	maximum speed of train	16

Table B.1: Constants used in the model of the train control system

T_RECEIVE: COMMUNICATION → TRAIN

Name	Type
COMMUNICATION_ESTABLISHED	Event
ACK_REC	Event
CROSSING_SAFE_REC	Event

C_RECEIVE: COMMUNICATION → CROSSING

Name	Type
ACTIVATE_CROSSING_REC	Event
STATUS_RQ_REC	Event
CROSSING_FREE_REC	Event

C_SEND: CROSSING → COMMUNICATION

Name	Type
ACK_SND	Event
CROSSING_SAFE_SND	Event

DIAGNOSTIC: TRAIN → DRIVER

Name	Type
EMERGENCY_STOP	Event
PASSED_XING	Event

LIGHT_HW_REPLY: LIGHTS → CROSSING

Name	Type
YELLOW_ERR	Condition
RED_ERR	Condition

LIGHT_HW_COMMANDS: CROSSING → LIGHTS

Name	Type
SWITCH_ON	Event
SWITCH_OVER	Event
SWITCH_OFF	Event

BARRIER_HW_REPLY: BARRIER → CROSSING

Name	Type
CLOSED	Condition
OPENED	Condition

BARRIER_HW_COMMANDS: CROSSING → BARRIER

Name	Type
LOWER	Event
RAISE	Event

SENSOR_HW_REPLY: SENSOR → CROSSING

Name	Type
SENSOR_ON	Condition
SENSOR_ERR	Condition

DEFECTS: CROSSING → OPERATIONS_CENTER

Name	Type
YELLOW_DEFECT	Event
RED_DEFECT	Event
BARRIERS_DEFECT	Event
SENSOR_DEFECT	Event
TIMEOUT_OPCENTER	Event

RESPONSES: OPERATIONS_CENTER → CROSSING

Name	Type
CROSSING_VACATED	Event

B.3 CROSSING

LIGHT_COMMANDS: CROSSING_CONTROL → LIGHTS_CONTROL

Name	Type
TURN_LIGHTS_ON	Event
TURN_LIGHTS_OFF	Event

LIGHTS_REPLY: LIGHTS_CONTROL → CROSSING_CONTROL

Name	Type
LIGHTS_ON	Event

BARRIER_COMMANDS: CROSSING_CONTROL → BARRIER_CONTROL

Name	Type
CLOSE_BARRIER	Event
OPEN_BARRIER	Event

BARRIER_REPLY: BARRIER_CONTROL → CROSSING_CONTROL

Name	Type
BARRIER_CLOSED	Event
BARRIER_OPENING	Event
BARRIER_ERR	Event

SENSOR_REPLY: SENSOR_CONTROL → CROSSING_CONTROL

Name	Type
PASSED	Event

Appendix C

LSC Grammar

```
<lsc document> ::= <document head> <live sequence chart>*

<end> ::= [ <comment> ] ;

<comment> ::= comment  <character string>

<text definition> ::= text  <character string> <end>

<document head> ::= { mscdocument | lscdocument }
                    <lsc document name> <end>

<identifier> ::=  <name>

<live sequence chart> ::=  { [ universal | existential ] }
                           [ lsc_kind ]
                           { lsc | msc } <lsc head> <lsc body>
                           { endlsc | endmsc }  <end>

<lsc_kind> ::= prechart | assumption

<lsc head> ::= <lsc name> <end>
                [ activation condition : condexpr <string>
                  endexpr activation mode <mode name> ]
                [ assumption <assumption list> ]

<assumption list> ::= <lsc name>+

<lsc body> ::= <lsc statement>*
```

```
<lsc statement> ::=   <text definition> [ end ]
                    | <event definition> [ end ]
                    | <old instance head statement>
                          <instance event list> [ end ]

<text definition> ::=  text  <character string> <end>

<event definition> ::=  <instance name>  :  <instance event list>
                      | <instance name list> :
                            <multi instance event list>

<instance event list> ::= { <instance event> } +

<instance event> ::= { <orderable event> |
                       <non-orderable event> }
                     <end> [ <delay> ]

<orderable event> ::= [ { hot | cold } ]
                      {  <message event>
                       | <timer statement>
                       | <action>
                       | <sim region> }
                      [ <layout info> ]

<layout info> ::= <character string>

<non-orderable event> ::= [ { hot | cold } ]
                           { { <coregion>
                              | <shared condition>
                              | <shared lsc reference>
                              | <instance head statement>
                              | <instance end statement> }
                                   [ <layout info> ]
                            | empty }

<instance name list> ::= <instance name> {, <instance name> }*
                       | all

<multi instance event list> ::= { <multi instance event> <end> } +

<multi instance event> ::= <condition> | <lsc reference>
                         | <inline expr>

<old instance head statement> ::= instance [ { hot | cold } ]
                                     <instance name> [ <sim region> ]
```

```
                                    [ <layout info> ] <end> [ <delay> ]

<instance head statement> ::= instance

<instance end statement> ::= endinstance

<message event> ::= { <message output> | <message input> } [ <delay> ]

<message output> ::= out [ { hot | cold } ] [ { sync | async } ]
                       <msg identification>  to  <input address>

<message input> ::= in [ { hot | cold } ] [ { sync | async } ]
                       <msg identification>  from  <output address>


<msg identification> ::= <message name> [ , <message instance name> ]
                           [ ( <parameter list> ) ]

<parameter list> ::= <parameter name> [  ,  <parameter list> ]

<output address> ::= <instance name>

<input address> ::= <instance name>

<shared condition> ::= <condition identification> <shared>

<condition identification> ::= condition  { [ hot | cold ] }
                                 <condition name list> :
                                 condexpr <string> endexpr

<condition name list> ::= <condition name>{ , <condition name> }*

<shared> ::= shared  { [ <shared instance list> ] |  all  }

<shared instance list> ::= <instance name> [, <shared instance list> ]

<condition> ::= <condition identification>

<timer statement> ::= <set> | <reset> | <timeout>

<set> ::= set  <timer name> [  ,  <timer instance name> ]
          [ (<duration name>) ]

<reset> ::= reset  <timer name> [  ,  <timer instance name> ]
```

```
<timeout> ::= timeout  <timer name> [  ,  <timer instance name> ]

<action> ::= action  <action character string>

<coregion> ::= concurrent  <end> <coevent>*  endconcurrent

<coevent> ::= <orderable event> <end>

<sim region> ::= simultaneous <sim region event list> endsim

<sim region event list> ::= <sim reg event> <end>
                              { <sim reg event> <end> }+

<sim reg event> ::=   <message event>
                    | <shared condition>
                    | <timer statement>
                    | <action>
                    | <local invariant event>

<local invariant event> ::=   inv start [ hot | cold ] <inv name>
                            | inv end <inv name>

<loop boundary> ::= { ‘ < ‘ <inf natural> [ , <inf natural> ] ‘ > ‘ }
                  | <delay>

<delay> ::= { ’(’ | ’[’ } <natural name> ’,’
               <natural name> { ’)’ | ’]’ }

<inf natural> ::= inf  | <natural name> +

<shared lsc reference> ::= reference
                             [ <lsc reference identification>  : ]
                             <lsc ref expr> <shared>

<lsc reference> ::= reference  [ <lsc reference identification> :  ]
                      <lsc ref expr>

<lsc reference identification> ::= <lsc reference name>

<lsc ref expr> ::=  <shared condition> then <lsc name>
                              [ else <lsc name> ]

<name> ::=  <word> { <underline> <word>}*

<word> ::=   { @ | <alphanumeric> | <full stop> }*  <alphanumeric>
```

```
            { @ | <alphanumeric> | <full stop> }*

<alphanumeric>  ::= <letter> | <decimal digit>

<letter>  ::=    A | B | C | D | E | F | G | H | I | J | K | L | M
               | N | O | P | Q | R | S | T | U | V | W | X | Y | Z
               | a | b | c | d | e | f | g | h | i | j | k | l | m
               | n | o | p | q | r | s | t | u | v | w | x | y | z

<decimal digit>  ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<char string>  ::= <apostrophe> {   <alphanumeric>
                                  | <other char>
                                  | <special>
                                  | <full stop>
                                  | <underline>
                                  | <space>
                                  | <apostrophe><apostrophe>
                                } <apostrophe>

<other char>  ::= ? | & | % | + | - | ! | / | > | * | " | < | =

<special>  ::= ( | ) | , | ; | :

<full stop> ::= .

<underline> ::= _

<apostrophe>  ::= ’



<text>  ::= {   <alphanumeric>
              | <other char>
              | <special>
              | <full stop>
              | <underline>
              | <space>
              | <apostrophe> }*

<note>  ::= /* <text> */
```

Bibliography

[ACS99] A. Allara, S. Comai, and R. Schlör. System Verification using User-Friendly Interfaces. In *Design, Automation and Test in Europe / User Forum*, pages 167–172. IEEE Computer Society Press, 1999.

[AD94] Rajeev Alur and David Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126(2):183–236, 1994.

[AEKN00] N. Amla, E.A. Emerson, R.P. Kurshan, and K.S. Namjoshi. Model Checking Synchronous Timing Diagrams. In *Proceedings Formal Methods in Computer-Aided Design (FMCAD)*, LNCS, pages 283–298. Springer Verlag, 2000.

[AHP96] R. Alur, G.J. Holzmann, and D. Peled. An analyzer for Message Sequence Charts. In T. Margaria and B. Steffen, editors, *Proceedings 2nd International Worshop on Tools and Algorithms for the Construction and Analysis of Systems, TACAS96*, volume 1055 of *Lecture Notes in Computer Science*, pages 35–48, Passau, Germany, March 1996. Springer Verlag.

[BAL97a] H. Ben-Abdallah and S. Leue. Expressing and analyzing timing constraints in message sequence chart specifications. Technical Report 97-04, Department of Electrical and Computer Engineering, University of Waterloo, April 1997.

[BAL97b] H. Ben-Abdallah and S. Leue. Timing constraints in Message Sequence Chart specifications. In T. Mizuno, N. Shiratori, T. Higashino, and A. Togashi, editors, *Formal Description Techniques and Protocol Specification, Testing and Verification. Proceedings*

of FORTE X and PSTV XVII '97, pages 91–106, Osaka, 1997. Chapman & Hall.

[BBB+99] Tom Bienmüller, Jürgen Bohn, Henning Brinkmann, Udo Brockmeyer, Werner Damm, Hardi Hungar, and Peter Jansen. Verification of Automotive Control Units. In Ernst-Rüdiger Olderog and Bernd Steffen, editors, *Correct System Design*, number 1710 in LNCS, pages 319–341. Springer Verlag, 1999.

[BBD+99] Tom Bienmüller, Udo Brockmeyer, Werner Damm, Gert Döhmen, Claus Eßmann, Hans-Jürgen Holberg, Hardi Hungar, Bernhard Josko, Rainer Schlör, Gunnar Wittich, Hartmut Wittke, Geoffrey Clements, John Rowlands, and Eric Sefton. Formal Verification of an Avionics Application using Abstraction and Symbolic Model Checking. In Felix Redmill and Tom Anderson, editors, *Towards System Safety – Proceedings of the Seventh Safety-critical Systems Symposium, Huntingdon, UK*, pages 150–173. Safety-Critical Systems Club, Springer Verlag, 1999.

[BBHW00] Tom Bienmüller, Udo Brockmeyer, Hans Jürgen Holberg, and Hartmut Wittke. Automatic Debugging for STATEMATE Designs, 2000. 8. Deutsches Anwenderforum für STATEMATE Magnum.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In W. R. Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems. Part of European Conferences on Theory and Practice of Software, ETAPS'99, Amsterdam*, volume 1579 of *LNCS*, pages 193–207. Springer-Verlag, 1999.

[BCM+92] J.R. Burch, E.M. Clarke, K. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, 98(2):142–170, June 1992.

[BDKW01] Tom Bienmüller, Werner Damm, Jochen Klose, and Hartmut Wittke. Formale Analyse und Verifikation von Statemate Entwürfen. *it+ti*, 43(1), 2001.

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

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

[BG02] Annette Bunker and Ganesh Gopalakrishnan. Verifying a VCI Bus Interface Model Using an LSC-based Specification. In H. Ehrig, B. J. Krämer, and A. Ertas, editors, *Proceedings of the Sixth Biennial World Conference on Integrated Design and Process Technology*, page 48. Society of Design and Process Science, June 2002.

[Bie03] Tom Bienmüller. *Reducing Complexity for the Verification of Statemate Designs*. PhD thesis, Carl von Ossietzky Universität Oldenburg, 2003.

[Bit00] Friedemann Bitsch. Classification of Safety Requirements for Formal Verification of Software Models of Industrial Automation Systems. In *Proceedings of the 13th Conference on Software and Systems Engineering and their Applications - ICSSEA 2000*, 2000.

[Bit01] Friedemann Bitsch. Safety patterns — the key to formal specification of safety requirements. *Lecture Notes in Computer Science*, 2187, 2001.

[BKL01] Udo Brockmeyer, Jochen Klose, and Marc Lettrari. UML Validation Suite. In J. Tretmans and E. Brinksma, editors, *Proceedings of FATES'01 - Formal Approaches to Testing of Software*, 2001.

[Boe88] Barry W. Boehm. A Spiral Model of Software Development and Enhancement. *IEEE Computer*, 21(5):61–72, 1988.

[Bon01] Yves Bontemps. Automated verification of state-based specifications against scenarios. Master's thesis, University of Namur, June 2001.

[Bro99] Udo Brockmeyer. *Verifikation von STATEMATE Designs*. PhD thesis, Carl-von-Ossietzky Universität Oldenburg, Oldenburg, Dezember 1999.

[BW98] U. Brockmeyer and G. Wittich. Tamagotchis Need Not Die – Verification of STATEMATE Designs. In Bernhard Steffen, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1384 of *Lecture Notes in Computer Science*, pages 217–231, 1998.

[CE81] E.M. Clarke and E.A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In D. Kozen, editor, *Proceedings of the Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71, Yorktown Heights, New York, May 1981. Springer-Verlag.

[CEN01] CENELEC. *EN 50128, Railway Applications, Communications, Signaling and Processing Systems - Software for Railway Control and Protection Systems*. European Committee for Electrotechnical Standardization, 2001.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[DAC98] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Property specification patterns for finite-state verification. In Mark Ardis, editor, *Proceedings of the 2nd Workshop on Formal Methods in Software Practice (FMSP-98)*, pages 7–15, New York, 1998. ACM Press.

[DAC99] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 1999 International Conference on Software Engineering (ICSE'99)*, pages 411–421, New York, 1999. Association for Computing Machinery.

[DDK99] W. Damm, G. Döhmen, and J. Klose. Secure Decentralized Control of Railway Crossings. In S. Gnesi and D. Latella, editors, *Fourth International ERCIM Workshop on Formal Methods in Industrial Critical Systems*, 1999.

[DH98] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. Technical Report CS98-09, The Weizmann Institute of Science, Rehovot, Israel, July 1998.

[DH99] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. In *FMOODS'99 IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems*, 1999.

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

[Die96] C. Dietz. Graphical formalization of real-time requirements. In B. Jonsson and J. Parrow, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, number 1135 in Lecture Notes in Computer Science, pages 366–385. Springer Verlag, 1996.

[DJHP98] Werner Damm, Bernhard Josko, Hardi Hungar, and Amir Pnueli. A compositional real-time semantics of STATEMATE designs. In W.-P. de Roever, H. Langmaack, and A. Pnueli, editors, *Proceedings COMPOS'97*, Lecture Notes in Computer Science 1536. Springer-Verlag, 1998.

[DJPV03] W. Damm, B. Josko, A. Pnueli, and A. Votintseva. Understanding UML: A Formal Semantics of Concurrency and Communication in Real-Time UML. In *Proceedings of the First International Symposium on Formal Methods for Components and Objects (FMCO'02)*, 2003. to appear.

[DK01] Werner Damm and Jochen Klose. Verification of a Radio-based Signaling System Using the Statemate Verification Environment. *Formal Methods in System Design*, 19(2), 2001.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, July 1960.

[DW03] W. Damm and B. Westphal. Live and Let Die: LSC-based Verification of UML-Models. In *Proceedings of the First Interna-*

tional Symposium on Formal Methods for Components and Objects (FMCO'02), 2003. to appear.

[EBF+98] A. Evans, J-M. Bruel, R. France, K. Lano, and B. Rumpe. Making uml precise. In *OOPSLA'98 Workshop on "Formalizing UML. Why and How?" 10 Seiten, Vancouver, Canada. October 1998.* OOPSLA'98, 1998.

[EFLR99] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. The uml as a formal modeling notation. In Jean Bezivin and Pierre-Alain Muller, editors, *The Unified Modeling Language - Workshop UML'98: Beyond the Notation*. Springer Verlag Berlin, LNCS, 1999.

[Ek98] Anders Ek. Automatic Debugging of Communicating Systems Using the Tau SDL Validator. Telelogic White Paper, 1998. http://www.telelogic.com.

[EK99] Andy Evans and Stuart Kent. Core meta-modelling semantics of uml: the puml approach. In R. France and B. Rumpe, editors, *UML'99 - The Unified Modeling Language: Beyond the Standard*, number 1723 in Lecture Notes in Computer Science. Springer-Verlag, 1999.

[Eme90] E. Allen Emerson. Temporal and modal logic. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 995–1072. Elsevier Science Publishers, 1990.

[ESt97] Bundesministerium des Inneren EStdIT. V-Model, Development Standard for IT-Systems of the federal Republic of Germany, 1997.

[Fey96] Konrad Feyerabend. Realtime Symbolic Timing Diagrams. Technical report, Carl von Ossietzky Universität Oldenburg, 1996.

[FHD+99] Thomas Firley, Michaela Huhn, Karsten Diethers, Thomas Gehrke, and Ursula Goltz. Timed sequence diagrams and tool-based analysis - a case study. In R. France and B. Rumpe, editors, *UML'99 - The Unified Modeling Language: Beyond the Standard*, number 1723 in Lecture Notes in Computer Science. Springer-Verlag, 1999.

[Fis99] K. Fisler. Timing Diagrams: Formalization and Formal Verification. *Journal of Logic, Language and Information*, 8(3), 1999.

[FJ97] Konrad Feyerabend and Bernhard Josko. A visual formalism for real time requirement specifications. In *Proceedings of the 4th International AMAST Workshop on Real-Time Systems and Concurrentand Distributed Software, ARTS'97, Lecture Notes in Computer Science 1231*, pages 156–168, 1997.

[FMR00] S. Flake, W. Müller, and J. Ruf. Structured English for Model Checking Specification. In *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, GI/ITG/GMM Workshop*, 2000.

[FNMD03] Martin Fränzle, Jürgen Niehaus, Alexander Metzner, and Werner Damm. A semantics for distributed execution of STATEMATE. Submitted to International Journal on Formal Aspects of Computing (FAC), Special Issue on Semantic Foundations of Engineering Design Languages, to appear., 2003.

[GDO98] V. Grabowski, C. Dietz, and E.R. Olderog. Semantics for timed Message Sequence Charts via constraints diagrams. In Y. Lahav, A. Wolisz, J. Fischer, and E. Holz, editors, *Proceedings of the 1st Workshop of the SDL Forum Society on SDL and MSC*, number 104 in Informatik-Berichte, pages 251–260, Berlin, Germany, June 1998. Humboldt-Universität zu Berlin.

[GHN+98] T. Gehrke, M. Huhn, P. Niebert, A. Rensink, and H. Wehrheim. A process algebra semantics for Message Sequence Charts including conditions. In H. König and Langendörfer, editors, *Proceedings of the 8th GI/ITG-Workshop Formale Beschreibungstechniken für verteilte Systeme (FBT'98)*, pages 185–196, Cottbus, June 1998.

[GHRW98] T. Gehrke, M. Huhn, A. Rensink, and H. Wehrheim. An algebraic semantics for Message Sequence Chart documents. In S. Budkowski, A. Cavalli, and E. Najm, editors, *Formal Description Techniques and Protocol Specification, Testing and Verification (FORTE/PSTV'98)*, pages 3–18. Kluwer Academic Publishers, 1998.

[GR99] Martin Große-Rhode. On a reference model for the formalization and integration of software specification languages. *Bulletin of the EATCS No 68, The Formal Specification Column, Part 8*, pages 81–89, June 1999.

[GRG93a] P. Graubmann, E. Rudolph, and J. Grabowski. The Standardization of Message Sequence Charts. In *Proceedings of the IEEE Software Engineering Standards Symposium (SESS'93)*, pages 48–63, Brighton, UK, September 1993. IEEE Computer Society.

[GRG93b] P. Graubmann, E. Rudolph, and J. Grabowski. Towards a Petri Net based semantics definition for Message Sequence Charts. In O. Færgemand and A. Sarma, editors, *SDL'93 – Using Objects, Proceedings of the Sixth SDL Forum*, pages 179–190, Darmstadt, October 1993. Elsevier Science Publishers/North Holland.

[Gro96a] The VIS Group. VIS : A System for Verification and Synthesis. In *8th international Conference on Computer Aided Verification*, number 1102 in LNCS, 1996. VIS 1.3 is available from the VIS home-page: `http://www-cad.eecs.Berkeley.EDU/~vis`.

[Gro96b] The VIS Group. VIS: A System for Verification and Synthesis. In *FMCAD*, 1996.

[Har00] David Harel. From Play-In Scenarios to Code: An Achievable Dream. In *International Conference on Fundamental Approaches to Software Engineering (FASE)*, volume 1783 of *Lecture Notes in Computer Science*. Springer Verlag, 2000.

[Hey00] S. Heymer. A semantics for MSC based on Petri-Net components. In S. Graf, C. Jard, and Y. Lahav, editors, *SAM2000. 2nd Workshop on SDL and MSC*, pages 262–275, Col de Porte, Grenoble, June 2000.

[HK99] Alexander Holt and Ewan Klein. A semantically-derived subset of English for hardware verification. In *Proc. 37th Annual Meeting of the Association for Computational Linguistics: Maryland, USA*, pages 451–456. Association for Computational Linguistics, 1999.

[HK01] David Harel and Hillel Kugler. Synthesizing State-based Object Systems from LSC Specifications. In *Proceedings of the 5th International Conference on Implementation and Application of Automata (CIAA 2000)*, volume 2088 of *Lecture Notes in Computer Science*, pages 1 – 33. Springer Verlag, 2001.

[HK02] David Harel and Hillel Kugler. Synthesizing State-based Object Systems from LSC Specifications. *International Journal of Foundations of Computer Science*, 13(1):5 – 51, 2002.

[HKMP02] David Harel, Hillel Kugler, Rami Marelly, and Amir Pnueli. Smart Play-Out of Behavioral Requirements. In *Proceedings of the* 4^{th} *International Conference on Formal Methods in Computer-Aided Design (FMCAD'02)*, pages 378–398, 2002.

[HLN+90] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. STATEMATE: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16:403 – 414, 1990.

[HM02] David Harel and Rami Marelly. Playing with Time: On the Specification and Execution of Time-Enriched LSCs. In *Proceedings 10th IEEE/ACM Int. Symp. on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2002)*, 2002.

[HN96] D. Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4), 1996.

[Hol99] Alexander Holt. Formal verification with natural language specifications: guidelines, experiments and lessons so far. *South African Computer Journal*, 24:253–257, 1999.

[HP96] David Harel and Michal Politi. *Modeling Reactive Systems with Statecharts: The STATEMATE Approach*. Part No. D–1100–43. i-Logix Inc., Three Riverside Drive, Andover, MA 01810, June 1996.

[IT88] ITU-T. *ITU-T Recommendation Z.100: Specification and Description Language (SDL)*. ITU-T, Geneva, 1988.

[IT93] ITU-T. *ITU-T Recommendation Z.120: Message Sequence Chart (MSC)*. ITU-T, Geneva, 1993.

[IT95] ITU-T. *ITU-T Annex B to Recommendation Z.120: Algebraic Semantics of Message Sequence Charts*. ITU-T, Geneva, 1995.

[IT96a] ITU-T. *ITU-T Annex C to Recommendation Z.120: Static Semantics of Message Sequence Charts*. ITU-T, Geneva, 1996.

[IT96b] ITU-T. *ITU-T Recommendation Z.120: Message Sequence Chart (MSC)*. ITU-T, Geneva, October 1996.

[IT98] ITU-T. *ITU-T Annex B to Recommendation Z.120: Formal Semantics of Message Sequence Charts*. ITU-T, Geneva, 1998.

[IT99] ITU-T. *ITU-T Recommendation Z.120: Message Sequence Chart (MSC)*. ITU-T, Geneva, 1999.

[IT00] ITU-T. *ITU-T Recommendation Z.100: Specification and Description Language (SDL)*. ITU-T, Geneva, 2000.

[JBR99] I. Jacobsen, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.

[Jos93] Bernhard Josko. Modular specification and verification of reactive systems. Carl von Ossietzky Universität Oldenburg, 1993. Habiltationsschrift.

[JP01] B. Jonsson and G. Padilla. An execution semantics for MSC-2000. In *SDL'01: Meeting UML*, Lecture Notes in Computer Science, Copenhagen, June 2001. Springer Verlag.

[KGSB99] I. Krüger, R. Grosu, P. Scholz, and M. Broy. From MSCs to statecharts. In Franz J. Rammig, editor, *Distributed and Parallel Embedded Systems, proceedings DIPES'98*, pages 61–71. Kluwer Academic Publishers, 1999.

[KL01] Jochen Klose and Marc Lettrari. Scenario-based Monitoring and Testing of Real-time UML models. In M. Gogolla and C. Kobryn, editors, *UML 2001 - The Unified Modeling Language: Modeling Languages, Concepts, and Tools*, volume 2185 of *LNCS*. Springer Verlag, 2001.

[KM00] Jochen Klose and Adam Moik. Modellierung der FORMS-Fallstudien mit Statemate. In Eckehard Schnieder, editor, *FORMS2000 - Formale Techniken für die Eisenbahnsicherung*, number 441 in Fortschritt-Berichte VDI Reihe 12. VDI Verlag, 2000.

[Kos97] P. Kosiuczenko. Time in message sequence charts: A formal approach. In C. Lengauer, M. Griebl, and S. Gorlatch, editors, *EuroPar'97: Parallel Processing*, number 1300 in Lecture Notes in Computer Science. Springer Verlag, 1997.

[KPE00] Olaf Kluge, Julia Padberg, and Hartmut Ehrig. Modeling Train Control Systems: From Message Sequence Charts to Petri Nets. In Eckehard Schnieder, editor, *FORMS2000 - Formale Techniken für die Eisenbahnsicherung*, number 441 in Fortschritt-Berichte VDI Reihe 12. VDI Verlag, 2000.

[KRK02] Jochen Klose, Juergen Ruf, and Thomas Kropf. A Visual Approach to Validating System Level Designs. In *15.th International Symposioum on System Synthesis (ISSS)*, pages 186 – 191, Kyoto, Japan, 2002. IEEE Computer Society Press.

[Krü00] Ingolf Krüger. *Distributed System Design with Message Sequence Charts*. PhD thesis, Technical University of Munich, 2000.

[KT00] Jochen Klose and Andreas Thums. The Statemate Reference Model of the Reference Case Study 'Verkehrsleittechnik'. Technical report, University of Augsburg, 2000. http://www.Informatik.Uni-Augsburg.DE/swt/formosa/RefVL/bericht.ps.gz.

[Kut94] George Kutty. *A Graphical Environment for Temporal Reasoning*. PhD thesis, Deptartment of Electrical and Computer Engineering, University of California, Santa Barbara, 1994.

[KW01] Jochen Klose and Hartmut Wittke. An Automata Based Representation of Live Sequence Charts. In Tiziana Margaria and Wang Yi, editors, *Proceedings of TACAS 2001*, number 2031 in LNCS. Springer Verlag, 2001.

[KW02] Jochen Klose and Bernd Westphal. Relating LSC Specifications to UML Models. In *Proceedings INT2002- International Workshop on Integration of Specification Techniques for Applications in Engineering*, 2002.

[Lam77] Leslie Lamport. Proving the Correctness of Multiprocess Programs. *IEEE Transactions on Software Engineering*, 3(2), 1977.

[LL92a] P.B. Ladkin and S. Leue. An analysis of Message Sequence Charts. Technical Report IAM-92-013, University of Berne, Institute for Informatics and Applied Mathematics, 1992.

[LL92b] P.B. Ladkin and S. Leue. An automaton interpretation of Message Sequence Charts. Technical Report IAM 92-012, University of Berne, Institute for Informatics and Applied Mathematics, 1992.

[LL95] P.B. Ladkin and S. Leue. Interpreting Message Flow Graphs. *Formal Aspects of Computing*, 7(5):473–509, 1995.

[LL99a] Xuandong Li and Johan Lilius. Timing analysis of message sequence charts. Technical Report 255, Turku Center for Computer Science, March 1999.

[LL99b] Xuandong Li and Johan Lilius. Timing analysis of uml sequence diagrams. In R. France and B. Rumpe, editors, *UML'99 - The Unified Modeling Language: Beyond the Standard*, number 1723 in Lecture Notes in Computer Science. Springer-Verlag, 1999.

[LMM99a] Diego Latella, Istvan Maijzik, and Mieke Massink. Towards a formal operational semantics of uml statechart diagrams. In *3rd International Conference on Formal Methods for Open Object-Oriented Distributed Systems*, Lecture Notes in Computer Science. Kluwer Academic Publishers, 1999.

[LMM99b] Diego Latella, Istvan Majzik, and Mieke Massink. Automatic Verification of a Behavioural Subset of UML Statechart Diagrams

Using the SPIN Model-checker. *Formal Aspects of Computing*, 11(6):637–664, 1999.

[LMR98] S. Leue, L. Mehrmann, and M. Rezai. Synthesizing ROOM models from Message Sequence Chart specifications. Technical Report 98-06, Department of Electrical and Computer Engineering, University of Waterloo, April 1998.

[LP99] Johan Lilius and Ivan Porres Paltor. Formalising uml state machines for model checking. In R. France and B. Rumpe, editors, *UML'99 - The Unified Modeling Language: Beyond the Standard*, number 1723 in Lecture Notes in Computer Science. Springer-Verlag, 1999.

[LPY97] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

[Man01] Nikolai Mansurov. Automatic synthesis of SDL from MSC and its applications in forward and reverse engineering. *Computer Languages*, 27(1–3):115–136, 2001.

[McM93] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[MHK02] Rami Marelly, David Harel, and Hillel Kugler. Multiple Instances and Symbolic Variables in Executable Sequence Charts. In *Proceedings of the 17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'02)*, pages 83–100, 2002.

[MP92] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems, Specification*. Springer Verlag, 1992.

[MR94] S. Mauw and M.A. Reniers. An algebraic semantics of Basic Message Sequence Charts. *The Computer Journal*, 37(4):269–277, 1994.

[MRW00] S. Mauw, M.A. Reniers, and T.A.C. Willemse. Message Sequence Charts in the software engineering process. In S.K. Chang, editor,

Handbook of Software Engineering and Knowledge Engineering. World Scientific, 2000.

[OMG01] OMG. *Unified Modeling Language Specification, Version 1.4*. Object Management Group, 2001. http://www.omg.org/docs/formal/01-09-67.pdf.

[OS97] Sam Owre and Natarajan Shankar. The formal semantics of PVS. Technical Report SRI-CSL-97-2, Computer Science Laboratory, SRI International, Menlo Park, CA, August 1997.

[OSC02a] OSC Group and I-Logix. *Statemate MAGNUM Model Certifier Pattern Library User Guide*, 2002.

[OSC02b] OSC Group and I-Logix. *Statemate MAGNUM Model Checker & Model Certifier User Guide 2.0*, 2002.

[QS82] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. 5th Int'l Symp. on Programming*, Lecture Notes in Computer Science, Vol. 137, pages 337–371. SV, 1982.

[RACH00] G. Reggio, E. Astesiano, C. Choppy, and H. Hussmann. Analysing uml active classes and associated state machines - a lightweight formal approach. In T. Maibaum, editor, *FASE, Fundamental Approaches to Software Engineering*, number 1783 in Lecture Notes in Computer Science, pages 127–146. Springer-Verlag, 2000.

[Ren99] M.A. Reniers. *Message Sequence Chart: Syntax and Semantics*. PhD thesis, Eindhoven University of Technology, June 1999.

[RHTR01] J. Ruf, D. W. Hoffmann, T.Kropf, and W. Rosenstiel. Simulation-guided property checking based on multi-valued ar-automata. In *Design Automation and Test in Europe (DATE)*. IEEE Conmputer Society Press, 2001.

[RKG97] G. Robert, F. Khendek, and P. Grogono. Deriving an SDL specification with a given architecture from a set of MSCs. In A. Cavalli and A. Sarma, editors, *SDL'97: Time for Testing – SDL, MSC and Trends, Proceedings of the Eighth SDL Forum*, pages 197–212,

Evry, France, September 1997. Elsevier Science Publishers/North Holland.

[Roy70] Winston W. Royce. Managing the Development of Large Software Systems: Concepts and techniques. In *WESCON Technical Papers, v. 14*, pages A/1–1–A/1–9, Los Angeles, August 1970. WESCON. Reprinted in *Proceedings* of the Ninth International Conference on Software Engineering, 1987, pp. 328–338.

[Sch00] Rainer Schlör. *Symbolic Timing Diagrams : A Visual Formalism for Model Verification*. PhD thesis, Universität Oldenburg, 2000.

[SD93] R. Schlör and W. Damm. Specification and verification of system-level hardware designs using timing diagrams. In *Proceedings of the European Conference on Design Automation with the European Event in ASIC Design*, pages 518–524. IEEE Computer Society Press, 1993.

[SS98] Mary Sheeran and Gunnar Stålmarck. A tutorial on Stålmarck's proof procedure for propositional logic. In Ganesh Gopalakrishnan and Phillip J. Windley, editors, *Proceedings Formal Methods in Computer-Aided Design, FMCAD'98, Palo Alto/CA, USA*, volume 1522 of *Lecture Notes in Computer Science*, pages 82–99, 1998.

[SvG98] J. Seemann and J. Wolff von Gudenberg. Extension of UML sequence diagrams for realtime systems. In *Proceedings International UML Workshop*, Mulhouse, June 1998.

[Tho90] Wolfgang Thomas. Automata on Infinite Objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*. Elsevier, 1990.

[Wit03] Hartmut Wittke. *A Framework for Specification Verification for Complex Embedded Systems*. PhD thesis, Carl von Ossietzky Universität Oldenburg, 2003. to appear.