TUM

INSTITUT FÜR INFORMATIK

Design of Reactive Systems and their Distributed Implementation with Statecharts

Peter Scholz

TUM-I9821
August 98

TECHNISCHE UNIVERSITÄT MÜNCHEN

TUM-INFO-08-I9821-100/1.-FI

Druck: Institut für Informatik der
Technischen Universität München

Design of Reactive Systems and their Distributed Implementation with Statecharts

Peter Scholz

Fakultät für Informatik
der Technischen Universität München

Design of Reactive Systems and their Distributed Implementation with Statecharts

Peter Scholz

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: ....

Prüfer der Dissertation:

1. ....
2. ....
3. ....

Die Dissertation wurde am .... bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am .... angenommen.

Abstract

In this thesis, a design process for reactive systems using μ-Charts, a visual formalism that is similar to the specification language Statecharts, is developed. The design process presented here, comprises abstract description of reactive systems, systematic transformation of abstract specifications into detailed specifications, formal verification through model checking, and centralized as well as distributed implementation. All design steps are formally described.

For the state-based description of reactive systems, we first define the language μ-Charts. μ-Charts are a variant of Harel's Statecharts, which, however, avoid the semantic problems and inconsistencies of the latter and are therefore better suited as a basis for distributed implementation of a specification. A formal semantics for μ-Charts is developed. While the core language of μ-Charts merely consists of three different syntactic constructs, namely sequential automata, a composition operator, and an operator for signal hiding, further syntactic concepts are expressed as syntactic abbreviations. One such example is hierarchical decomposition. The lean core syntax of μ-Charts eases the formal definition of all design steps and is a prerequisite for efficient implementation.

In order to support a systematic design process, we define a refinement calculus for μ-Charts, which builds the basis for transforming abstract behavioral specifications into detailed ones. The soundness of this calculus with respect to the formal semantics is proved.

For formal verification of reactive systems, a translation scheme for μ-Charts to the formalisms of two well-known model checking tools is given. How to check safety critical properties of a specification is demonstrated exemplarily by means of a running example.

A technique that is based on using finite state machines to generate centralized implementations is described. The thesis concludes with an approach for distributed implementation of μ-Chart specifications. Problems that may occur while constructing a distributed implementation are discussed and appropriate restrictions to avoid non-feasible implementations are made. The semantic equivalence between distributed implementation and original specification is verified.

Kurzfassung

In dieser Arbeit wird ein Entwurfsprozeß für reaktive Systeme unter Verwendung eines an der Spezifikationssprache Statecharts angelehnten visuellen Formalismus mit dem Namen μ-Charts entwickelt. Der vorgestellte Entwurfsprozeß umfaßt die abstrakte Beschreibung reaktiver Systeme, den systematischen Übergang von abstrakten zu detaillierten Spezifikationen, die formale Verifikation durch Model Checking, sowie die monolithische und verteilte Implementierung. Alle Entwurfsschritte werden formal fundiert.

Für die zustandsbasierte Beschreibung reaktiver Systeme wird zunächst die Sprache μ-Charts definiert. μ-Charts sind eine Variante von Harels Statecharts, die allerdings deren semantische Probleme und Inkonsistenzen umgeht und damit als Ausgangsbasis für die verteilte Implementierung einer Spezifikation geeignet ist. Für μ-Charts wird eine formale Semantik entwickelt. Während die Kernsprache von μ-Charts lediglich mit drei unterschiedlichen syntaktischen Konstrukten, nämlich sequentiellen Automaten, einem Kompositionsoperator und einem Operator zur Definition lokaler Signale auskommt, werden weitere syntaktische Konzepte, wie etwa die für Statecharts typische hierarchische Dekomposition, durch Abkürzungsmechanismen ausgedrückt. Die schlanke Kernsytax von μ-Charts erleichtert die formale Fundierung aller Entwurfsschritte und ist Voraussetzung für eine effiziente Implementierung.

Zur Unterstützung eines systematischen Entwurfsprozesses wird für μ-Charts ein Verfeinerungskalkül definiert, der zur Transformation von abstrakten in detaillierte Verhaltensbeschreibungen verwendet werden kann. Die Korrektheit dieses Kalküls für die formale Semantik wird bewiesen.

Um die formale Verifikation von reaktiven Systemen zu ermöglichen, wird ein Anschluß von μ-Charts an zwei bekannte Model Checking Werkzeuge realisiert. Anhand eines durchgängigen Beispiels wird der Nachweis sicherheitskritischer Systemeigenschaften exemplarisch durchgeführt.

Ein auf endlichen Zustandsmaschinen basierendes Verfahren zur Erzeugung monolithischer Implementierungen wird angegeben. Den Abschluß der Arbeit bildet ein Ansatz zur verteilten Implementierungen von Spezifikationen mit μ-Charts. Hierbei auftretende Probleme werden diskutiert und entsprechende Einschränkungen des Implementierungsspielraumes vorgenommen. Die semantische Äquivalenz der verteilten Implementierung mit der ursprünglichen Spezifikation wird nachgewiesen.

Acknowledgment

I wish to take the opportunity to thank all those people who have helped me, either directly or indirectly, to develop this thesis.

First and foremost, thanks go to my advisors Manfred Broy and Carlos Delgado Kloos. Manfred Broy enabled me to join his research group and so opened to me a wide field of interesting research topics and introduced me into the world of the leading international researchers who work on topics related to my thesis. Though being far away from Munich, Carlos Delgado Kloos also agreed to supervise my thesis. I would like to thank him for getting involved with this adventure and for the useful discussions we have had. I am indebted to him and the people of his group at the Universidad Carlos III de Madrid for their hospitality. A special "gracias" goes to Peter T. Breuer, Andrés Marín López, Natividad Martínez Madrid, and Luis Sánchez Fernández for supporting me with personal affairs while being in Madrid.

When I started working as research assistant, Dieter Nazareth and Franz Regensburger helped me to solve both technical and personal problems whenever I needed their help. Without them, I would not have discovered the interesting subject of this thesis.

I am also very grateful to many friends and colleagues for comments and discussions on the subject. In particular, I would like to thank Jan Philipps, Bernhard Rumpe, and Bernhard Schätz for numerous fruitful discussions on this and on related topics. I am proud of being Jan Philipps' co-author of some common papers. Thanks also go to Michael von der Beeck, Max Breitling, Simon Pickin, Oscar Slotosch, and Katharina Spies for carefully reading parts of draft versions of this thesis. I am particularly indebted to Ingolf Krüger for reading a previous version of the complete thesis and for many helpful, constructive comments.

The contributions of many others are less direct, but just as important. A final thank-you to my parents, brothers, and to all friends for enduring me during this time and for reminding me of the better things in life.

Contents

1 Introduction

In this thesis, a design process for reactive systems using μ-Charts, a visual formalism that is similar to the specification language Statecharts, is developed. The design process presented here, comprises abstract description of reactive systems, systematic transformation of abstract specifications into detailed specifications, formal verification through model checking, and centralized as well as distributed implementation. This section is organized as follows: we start with a motivation for the thesis. After that, we present and discuss relevant related work. Then, we summarize the results of the thesis on one page. Finally, to support the reader in navigating through this document, we sketch the latter's outline.

1.1 Motivation for this Thesis

Today's computer systems can be divided into three categories: (purely) transformational, interactive, and reactive systems [HP85, Hal93b]. These systems are mainly distinguished by the way they transform input into output. *Transformational systems* simply transform input that is completely available at the start of a system execution into output. This output is not available before the execution terminates. In such systems, the user or, more generally, the environment is unable to interact with the system and to influence a running execution.

In contrast, *interactive systems*, such as operating systems in personal computers, do not only compute output when terminating, but interact and synchronize continuously with their environment. Here, the interaction is determined by the system itself and not by the environment: whenever the system needs new input to continue its execution, the environment is prompted by the system. If the synchronization is determined by the environment, we speak of *reactive systems*.

Reactive Systems

Systems that do not continuously react to input coming from the environment (including actors and sensors) at their own rate, but at a speed that is entirely determined by the environment itself, are called *reactive* systems (Figure 1.1). Reactive systems have a widespread field of application: they occur, for instance, in avionics, automotive, telecommunication, and chemical industry. Today, more microprocessors are used in reactive systems than in personal computers, and the former's number is continuously growing.

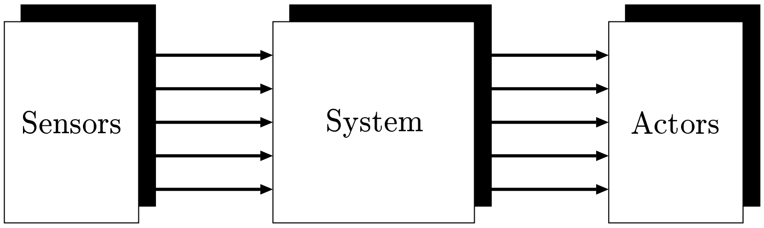


Figure 1.1: Reactive system

Reactive systems can either implement control functions, or data communication and processing, or both. A number of specification techniques to describe control-oriented reactive systems is based on automata-like languages [Har87, LT89, GKRB96, Bro97b]. In this thesis, we concentrate on reactive, control-oriented systems, which we describe by a *Statecharts*-like visual formalism called μ-Charts. Statecharts, originally introduced in [Har87], are a visual specification language proposed for specifying reactive systems. They extend conventional state transition diagrams with structuring and communication mechanisms. These mechanisms enable the description of large and complex systems. Due to this fact Statecharts have become quite successful in industry.

The main features of reactive systems, which also are of importance in this contribution, are (see [Hal93b] for a more detailed overview): they are often concurrent, are required to be reliable and to guarantee timing constraints, can be realized both in hardware and in software, and may be implemented on a computer network.

In this context, the notion "environment" not only refers to the natural surrounding in which the system acts, but also includes the controlled part of a larger system. Here, we say that the complex controlling part, i.e. the system itself, is embedded in a mechanical, chemical, biological, etc. environment.

The reaction of such a system in general consists of three steps. First, the system (the controlling part) samples data from the environment (the controlled part), processes it, and finally gives a response [TBYS96]. For these systems their functional correctness is particularly important for several reasons: first, human life may be at risk and second, from an economical point of view, time-to-market times can be delayed or already delivered products have to be returned to the manufacturer and repaired. For instance, we expect that a central locking system always unlocks the car doors in the case of a crash.

In many applications, these operations have to be performed not only functionally correct, that is, that the reaction yields a correct output, but also within specified times, that is, neither too early nor too late. In this case, we also speak of *real-time* systems. Real-time systems can be classified as *hard* or *soft* real-time systems depending on the time frame that is available for a reaction. While hard real-time restrictions have to be fulfilled accurately under all circumstances, that is, deadlines must not be exceeded, soft real-time restrictions are less rigorous and allow that reactions are performed within tolerance ranges.

Thus, also the consequences a timing constraint violation entails are different for hard and soft real-time systems: when deadlines are not accurately met in hard real-time systems, life or health of human beings can be affected or technical equipment damaged. Violations of soft real-time timing constraints have less fatal implications. Typical examples for systems with such constraints are navigation systems or central locking systems of modern cars. Interactive systems, in contrast, do not have to meet deadlines, but must be capable to give reasonably quick responses.

One particularly interesting application field of reactive systems are aircrafts or automobiles: reactive systems can, in the form of *embedded systems*, be a part of automobile electronics. In a modern upper class car, more than 50 processors or electronic control units can be integrated. These electronic systems increasingly take over tasks of the driver and assist him or her in safety critical or standard situations. Mostly, these processors are dedicated to solve a distinct task, like the opening or closing of doors as part of a central locking system. Other well-known systems of this type are the Automatic Stability Control (ASC) and the Anti Blocking System (ABS).

More and more functions will be implemented by software in the cars of the future. However, these systems do not act standalone, but interact to solve complex problems. So-called "virtual functionalities" that consist of a number of already existing functionalities spring up [Spr96]. Today up to 30 percent of the development costs of an upper class car are caused by electric and electronic units [Spr96]. Hence, every single unit must have a high performance so that the additional costs to implement it in the car are justified. Thus, there is a growing interest in industry to use not only one processor for a single task, but to appoint it to different tasks which may not be related at all. However, the specification of a task still should be implementation independent. Hence, a description technique is needed that still leaves room for various implementation strategies, but nevertheless provides all necessary prerequisites for arbitrary implementations. Further reasons why reactive systems are implemented on distributed architectures are [Hal93b]:

- The distributed implementation is imposed by the target architecture through sensor and actor locations.
- The distributed implementation improves the overall performance of the reactive system as the inherent concurrency in the model can be (better) exploited.
- Through several redundant implementations a higher fault-tolerance can be achieved. This is in particular important for safety critical systems.

Altogether, the design of reactive systems and their distributed implementation is a challenging task that requires elaborated and adapted design techniques. The aim of this thesis is to give a complete design process for the development of reactive systems with the visual formalism Statecharts. "Complete" in this context means that the designer is guided from the behavioral system specification stage to the final, distributed implementation on a processor network.

To achieve this challenging task, we follow the subsequent strategy:

1. We define the compositional Statecharts dialect μ-Charts that is suited for both centralized and distributed implementation. For this language, also a precise semantics is formalized.

2. We present a refinement calculus for this dialect.

3. We show how μ-Chart specifications can be formally verified by model checking.

4. We discuss how single μ-Chart specifications can be partitioned and implemented on both single processors and processor networks.

The remainder of this section contains a more detailed motivation for each of the above topics. It is structured in accordance with the chapters of this thesis.

Statecharts

First of all, we develop the theory of μ-Charts. μ-Charts are a dialect of Statecharts [Har87], a visual specification language for reactive systems. They extend conventional state transition diagrams with structuring and communication mechanisms. A typical application area is rapid prototyping of embedded systems [Spr96]. Among others, their success is explained by two facts. First, it is a relatively easy-to-learn language for design specialists who have more often a degree in electrical or electronic engineering than a solid background in computer science. Due to the state-based specification concept, these engineers have a better intuition of the meaning of automata than, for instance, of algebraic specification techniques. Second, Statecharts are available as a description technique in commercial products like STATEMATE [i-L90, i-L97]. Therefore, these specifications are also applicable for practitioners and have become quite successful in industry.

In recent years, much scientific work has been invested to improve the Statecharts language. However, up to now most approaches focus more on defining new semantics for Statecharts than on providing solid specification techniques. Several informal and formal semantics for Statecharts and related dialects have been proposed (see [vdB94] for a good, but no longer complete overview). Some approaches, like Argos [Mar92, MH96], are closely related to the language ESTEREL [Ber91, BdS91, BG92]. When designing our dialect μ-Charts we were inspired by visual programming languages like Argos.

In this thesis, we define the semantics of μ-Charts in terms of sets of I/O behaviors or, in other words, I/O histories. We illustrate that three syntactic concepts, sequential automata, signal hiding, and a composition operator including multicasting are sufficient to express also more complex Statecharts constructs; hierarchy and pure parallel composition can be defined on top of them as syntactic sugar. This strategy has two main advantages. First, we restrict ourselves to the most essential language concepts and so can motivate that Statecharts are not as complex as assumed in the hitherto existing literature. Second, we get a semantics that leads to simple proofs when reasoning about the semantics itself. Third, treating hierarchical decomposition as parallel composition and communication is an elegant way to establish distributed implementation of charts in different hierarchical levels.

Our dialect features a formal semantics for non-deterministic Statecharts with instantaneous feedback. Specifications with instantaneous feedback fulfill the perfect synchrony hypothesis [Ber89]. As noted in previous works on the semantics of Statecharts [HRdR92, PS91], or Statecharts-like languages like Argos [Mar92, MH96], instantaneous feedback can lead to causality conflicts (causal loops or contradictions) if, for instance, trigger events with negation are allowed.

Nevertheless, we prefer this kind of instantaneous feedback, since delayed communication is not a suitable communication concept for behavioral refinement. When refining a sub-chart to a set of more concrete sub-charts, additional delays are introduced. Thus, the I/O-behavior of the Statechart changes. Refinement rules would have to be more complex to compensate additional delays [Kle97]. As observed in [HdR91], this is not the case for instantaneous feedback. Section 1.2.2 contains further motivation for choosing this communication mechanism.

Refinement

Our μ-Charts formalism is considered to be a specification technique rather than a programming language like Argos, ESTEREL, LUSTRE, or SIGNAL. Therefore, we show in this thesis how to use μ-Charts in an incremental development process. We demonstrate what it means to develop a system step-by-step. We present a refinement calculus with rules that are easy to understand, but at the same time describe formal design steps towards the final system. One of our goals is to show that μ-Charts are more than a simple two-dimensional programming language. A design methodology is needed, supported by a set of purely syntactic refinement rules that tell the user how to transform an abstract, *under-specified* system description into a more concrete one. The essential rules we present are thought to be applicable not only for μ-Charts, but also for any other version of Statecharts.

On the semantic level, under-specification is reflected by non-determinism (see the design process on Page 8). Talking about non-determinism we have to distinguish between intended and unintended non-determinism. In synchronous programming languages like ESTEREL and others, non-determinism is entirely excluded. In particular, *unintended* non-determinism that is obtained by composition with instantaneous multicasting is avoided by static analysis. Besides this unintended non-determinism by composition there is also *intended* non-determinism to express under-specification of single components. Intended non-determinism is volitional by the user and reflects that design decisions for a component are still left open at an intermediate stage of development. The concretization of the latter then is called *refinement*.

Though in both μ-Charts and FOCUS [BDD+93], the respective notion of refinement follows in principle the same idea of making a system more concrete, there exists an important difference: while refinement rules for decomposition are the central concept in FOCUS, in μ-Charts these are refinement rules for composition. In FOCUS, there are refinement rules for decomposing a specification F into components F_1 and F_2 such that F is equivalent to $F_1 \otimes F_2$, where $\otimes$ represents the composition operator in FOCUS. In order to guarantee that $F_1 \otimes F_2$ has the same properties as F, it is necessary to define the semantics of composition by the least fixed point. However, in μ-Charts, there is no such refinement rule for decomposition. In Section 2.5.1, we will see that also hierarchical decomposition is defined through composition. In μ-Charts, we therefore do not provide any refinement rule for decomposition in the sense of FOCUS, but only for composition. Rather, we are interested in finding syntactic requirements to guarantee that $S_1 \lhd L \rhd S_2$ is a correct refinement of S_1. Here, S_1 is not decomposed, but the new chart S_2 is added to S_1. The refinement rule for composition that we will give in Section 3.2.2 secures that the semantics of S_1 is not affected when composing S_1 with S_2 to $S_1 \lhd L \rhd S_2$. Hence, though possibly introducing additional non-determinism for S_2 by composition (see Page 8), S_1 has exactly the same properties when it is composed with S_2 as before.

Verification

In this thesis, we demonstrate how to use the semantic model we introduce as starting point for an efficient formal verification, based on symbolic model checking techniques. We give a scheme, demonstrated by an example, that translates μ-Charts specifications into μ-calculus formulae and the SMV input language. These formulae are checked against temporal specifications using a μ-calculus verifier and the SMV model checker, respectively.

Implementation

As a further result of our work on symbolic verification of μ-Charts, we show how a μ-Chart can be implemented as a finite state machine, using a register and a combinational logic block that represents the transition relation of the system.

However, we do not only provide a strategy for centralized implementation, but in addition discuss problems that occur and restrictions that have to be fulfilled when a μ-Chart specification will be realized with more than one processor. The architecture template we focus on is a realistic architecture that is widespread in automobile industry: it consists of a number of electronic control units that get inputs from sensors, deliver outputs to actors, and are connected via one or more field busses.

Partitioning

Before a single μ-Chart specification can be implemented on a network of processors, however, some deliberation is necessary. We discuss which problems occur when a specification is partitioned in order to implement it on a processor network.

Apart from the aforementioned reasons, we had to develop our own dialect of Statecharts to bridge the gap between an abstract, formal system description and a distributed implementation. The μ-Charts language fulfills several technical requirements that are a prerequisite for distributed implementations.

It only consists of three syntactic constructs: sequential automata, signal hiding, and a composition operator that combines parallel composition and multicasting. Hierarchical decomposition is defined as syntactic abbreviation: it is achieved by parallel composition and message passing. Thus, we can solve the question how to implement charts of different levels of hierarchy on different processors. With Statecharts semantics that treat hierarchy on the semantic level this could not be achieved that elegantly. A syntactically rich language, in which every single construct has its own semantics would neither be suited as starting point for the definition of refinement techniques nor for partitioning nor for distributed implementation.

A further prerequisite for distributed implementation is compositionality of the underlying description technique. The μ-Charts semantics presented in this thesis is compositional and therefore builds a solid basis for partitioning.

Not every specification can be implemented ad hoc on a processor network. In many cases, the specification first has to be transformed in an equivalent one. We therefore give a transformation rule that guides the user to transform a given μ-Chart specification in one that is semantically equivalent, but better suited for the specific target architecture. Our target architecture consists of a finite number of processors that are connected through a set of busses.

Since their syntax is defined incrementally with sequential automata as basic construct, μ-Charts straightforwardly lend themselves to partitioning specifications on the granularity level of automata. A more fine-grained granularity would yield a far too complex partitioning strategy and a more coarse-grained granularity possibly would yield inefficient partitionings.

When partitioning a specification in μ-Charts, care has to be taken that communication between the different parts is deadlock-free or, in other words, does not contain any causality errors. The fixed point semantics we define for the deterministic μ-Charts version exactly reflects the signal causality relationships within one instant and therefore also the communication of distributed μ-Charts components in detail. As a consequence, this semantics serves as mathematical basis to value the partitioning and to verify that no communication deadlocks arise.

As the overall result, this thesis offers a design process for a Statecharts dialect that spans the gap between visual formalization of reactive systems up to their distributed implementation on a processor network in an embedded system. Apart from the very early development phase requirements engineering, all important stages of system development (description, refinement, formal verification, partitioning, and implementation) are covered. In each design step, we use equivalent formal semantics for μ-Charts. As a consequence, we do not loose any information between the different stages. Thus, the user can be sure that the system he or she gets in the final product is semantically equivalent to the one that has been abstractly specified, refined, and formally verified. Hence, we follow the principle of "what you specify and verify is what you implement" and all concepts presented here harmonize with each other. For instance, the fact that parts of a μ-Chart specification of different hierarchy levels possibly are implemented on different processors has already been taken into account in the definition of the language. To demonstrate this with clarity, in the following, we will briefly describe a process for the design of possibly distributed reactive systems with μ-Charts.

Design Process

Our design process for reactive systems starts with the state-based *system description* by the Statecharts-like specification language μ-Charts. As a user might not have a precise imagination how the system under development shall look like in detail in this phase, the system yet might be *under-specified*. Roughly speaking, this means that design alternatives for certain system parts or, more precisely, certain sequential automata yet have been left open by him or her. In the context of this thesis, under-specification is a purely syntactic notion that only refers to sequential automata. An automaton can be under-specified because of the following three reasons.

First, the user may have specified more than one initial state for the automaton. Second, the designer may have specified more than one transition for at least one input event in one or more states. Last but not least, no transition might be defined for at least one input event in one or more states. The last type of under-specification is termed *non-responsiveness*. If for every state and every input event of an automaton at least one transition has been specified, we call the automaton *responsive* because it can fire a transition in every state on every input event. This is not the case for non-responsive automata. In the first two cases of under-specification, we say that the system under development is *(syntactically) non-deterministic*. An automaton where exactly one transition is defined for each possible input event in each state and that only has one single initial state is both *deterministic* and *responsive*. We call such an automaton also *completely specified*. If all automata of a μ-Chart specification are completely specified, we also call the overall μ-Chart itself completely specified and under-specified otherwise. If all automata are responsive, we also term the μ-Chart itself responsive and non-responsive otherwise.

Here, we would like to add a warning: apart from this purely syntactic notion of non-determinism, there also exists a semantic notion of non-determinism. A μ-Chart is *(semantically) non-deterministic* if and only if its semantics contains more than one input/output history. For sequential automata, under-specification is isomorphic to semantic non-determinism. However, for the composition operator of μ-Charts this is in general not true. Later on, we will see that syntactic non-determinism also is semantic non-determinism, but that the opposite is not true. With μ-Charts, semantic non-determinism cannot only be caused by syntactic non-determinism or, more generally, under-specification, but also by composition. This phenomenon is common to most synchronous languages (see Section 1.2.2). While syntactic non-determinism is intended by the user, semantic non-determinism that is caused by composition is not. Detecting semantic non-determinism caused by composition requires subtle analysis techniques which have been discussed in detail for centralized implementations in the literature (cf. ESTEREL). In this thesis, we will present an approach for distributed implementation. Table 1.1 summarizes the aforementioned classification.

Syntax	*Semantics*
A syntactically non-deterministic	$[\![A]\!]$ non-deterministic
A non-responsive	$[\![A]\!]$ not defined

Table 1.1: Under-specification of automaton A

Taking a look at Table 1.1, we recognize that non-responsive automata do not have a defined semantics since the designer did not define any behavior. In order to ease the mathematical formulation of the behavior, that is, the semantics, and the mathematical reasoning with it, we provide a technique to unify both syntactic notions, non-determinism and non-responsiveness, on the semantic level. The technique we use is *chaos completion* [Rum96]. It allows to interpret non-responsiveness as non-determinism, too. Whenever a sequential automaton of a μ-Charts specification is non-responsive in a state, we assume that this automaton can perform every possible reaction in this state, that is, outputs an arbitrary event that is included in its output interface and has an arbitrary successor configuration. The latter means that the subsequent control and data states can have an arbitrary value within their admissible ranges. Once having started to react chaotically, we cannot make any assumption how the automaton will behave in future steps. As a consequence, the overall system reaction (an infinite stream of consecutive steps) is chaotic.

We define chaos completion in a way such that the automaton carries on behaving chaotically forever. Instead of no system reaction we so get all possible system reactions. Therefore, we say that this part of the system behaves *chaotically*. The mathematical result of the discussion above is a stream semantics for μ-Charts with chaos completion. This stream semantics is tailored for both the mathematical definition of the behavior of possibly non-deterministic and non-responsive μ-Chart specifications and the formal reasoning about how to get implementable specifications: since reactive systems must behave reliably and completely predictably in all situations, we want our final systems to behave (semantically) deterministic.

As a consequence, in the course of system development, under-specification has to be resolved in order to get an implementable system. Hence, in the next design phase, the *refinement* phase, we provide the system developer with easy-to-apply refinement techniques. These are based on a set of syntactic refinement rules. Each rule supports the designer with context restrictions that must be fulfilled in order to get correct refinement steps. All rules are combined to a *refinement calculus* that guarantees the correct stepwise concretization of a system under development towards a final deterministic and responsive description that fits for implementation. We would like to mention that through refinement also those specifications that include (unintended) semantic non-determinism that has been introduced through composition may be transformed into deterministic specifications.

As soon as the system engineer comes up with a responsive specification, which yet may be non-deterministic, she or he can begin to formally verify certain properties of the system by *model checking*. Note that we do not support the model checking of non-responsive μ-Charts. This is because of methodological and technical reasons. Once a system has started to react chaotically, any system behavior is possible. Hence, the system does not have any longer a "reasonable" behavior, for which interesting properties could be verified. Moreover, since our semantic view of non-responsiveness is chaos, we would have to explicitly encode chaotic behavior with the input language of a model checker, too. If we skipped these explicit encodings, our model for formal verification, a Kripke structure, possibly could perform deadlocks in those configurations where the μ-Charts is non-responsive. However, the explicit encoding of chaos is impracticable. As a consequence, we decided to support model checking of responsive μ-Charts only, which is not a strong restriction in practice. Notice that in contrast to the explicit encoding for model checking, we can formulate chaos implicitly with the theoretical semantics.

While the development phases description and refinement can be formally treated with the same semantics, we have to define an extra semantics for model checking. The reason for this is as follows: while the stream semantics is excellently suited for formal reasoning of complete input/output histories of μ-Charts, it is not an adequate means to formalize the transition relations for model checking. Here, a step semantics is needed to encode transition relations. We will prove that step and stream semantics are equivalent for responsive specifications.

Model checking must not necessarily be performed in a separate phase of development shortly before the implementation. Rather, system properties of different levels of abstraction can be verified at different phases of development. These properties can be divided into liveness and safety properties. Informally, the first guarantees that eventually something good will happen and the latter that nothing bad will happen. Once having verified a certain safety property, the application of correct refinement rules guarantees that this property still holds after the application of this rule. While safety properties are not affected by refinement steps, liveness properties are. This is explained by the fact that non-determinism is restricted through refinement. As a consequence, behavioral alternatives that have been included on an earlier design stage possibly are excluded by subsequent refinement steps.

When we finally arrive at a complete specification for which all required properties have been formally verified, we can implement it either on a centralized or on a distributed target architecture. While the first is relatively unproblematic, this is not the case for the latter. Locations for every single sequential automaton have to be defined. We call this process, which can be optimized by tool support, *allocation*. However, before a complete specification can be implemented on a distributed target architecture, a number of considerations is needed. First, this specification has to be partitioned. However, not every partition is feasible. We discuss context restrictions and provide the system engineer with guidelines how to partition a μ-Chart specification.

In order to reason about partitioning and distributed implementation, we consult the formal semantics of μ-Charts again. However, the mathematical discussion of multicast communication in the case of distributed implementation requires a slightly different semantic view: in each instant, distributed components send and receive signals according to a chain reaction until the overall system reaches a stable configuration. This stabilization process can be mathematically understood as fixed point computation. Hence, we provide a tailored semantics for deterministic and responsive μ-Chart specification to do so. For deterministic and responsive specifications this semantics is just a concretization of the original stream semantics.

1.2 Related Work

This thesis was influenced by a number of related approaches. Since the thesis spans the bridge between a number of theoretical and practical research topics that are usually considered as self-standing works, we highlight in the following the related work clustered by research topics. The subsequent items are also reflected to some extent in the overall structure of the thesis.

1.2.1 Statecharts and Related Approaches

Statecharts were originally presented in [Har87] and later on implemented in the commercial tool Statemate [i-L90, i-L97]. The full Statemate Statecharts language, however, contains many mechanisms that cause problems concerning both syntax and semantics. A description of these problems together with an overview of some approaches can be found in [vdB94]. The most important weak points are: they suffer from the shared variable model that is used, the *implicit* interaction between so-called orthogonal states, possible infinite chain reactions, and non-compositionality.

These deficiencies have been avoided in the μ-Charts dialect that we present here; though μ-Charts are much simpler than Statemate Statecharts, they are powerful enough to describe large and complex reactive systems. Nevertheless, we can assign a concise, formal semantics to them. All aforementioned problems are avoided. The most important restrictions are that only *explicit multicasting* is featured by μ-Charts and we neither allow shared variables nor *inter-level transitions*. These are transitions between nodes on different levels, which means that they cross the borderline of one or more states.

Without further assumptions, inter-level transitions impede the definition of a compositional semantics. As a consequence, STATEMATE Statecharts cannot be developed in a *modular* way. However, μ-Charts are clearly decomposed into sub-charts and can be constructed by simply sticking them together.

Argos [Mar92] and the approach followed in [HRdR92] provide steps into the same direction. Our work extends their approaches in many respects. First, Argos does not offer data variables. Second, though Argos uses an explicit feedback operator for communication it does not generate causality chains for signals as we do in our step semantics that is the basis for distributed implementation. Instead, it simply computes the set of broadcast signals by solving signal equations. In our step semantics used for implementation, however, each system step consists of a number of micro-steps. In each such micro-step, further signals for multicasting in the current system step are added until a fixed point is reached. Furthermore, for Argos in [MH96] non-determinism is introduced by using external prophecy signals. This is different from the approach we follow. In our stream semantics, these prophecies are not necessary since non-determinism is treated internally. Finally, though in the Statecharts version described in [HRdR92] data variables are available, it is not possible to construct larger charts by simply sticking its components together as in Argos or μ-Charts.

A model that covers specification, partitioning, and implementation of mixed software-hardware systems are *Codesign Finite State Machines (CFSMs)* [CGH$^+$93a, CGH$^+$93b, CGH$^+$94]. They are based on potentially non-deterministic finite state machines and are particularly suited for a specific class of systems with relatively low algorithmic complexity. In contrast to μ-Charts, CFSMs use a non-zero unbounded reaction delay model. Hence, though they can in principle be implemented both in hardware and in software, this time model is more dedicated to software than to hardware design [CGH$^+$93a]. As a general rule, CFSMs require to implement as much as possible in software, leaving hardware only the most performance critical components. The communication primitive for a network of interacting finite state machines is, similar to most Statecharts-like languages, broadcasting: events are emitted by a CFSM or the environment and can be detected by one or more CFSMs. In contrast to μ-Charts and related approaches, an event is present and can be detected not only at the instant of its emission, but until it is either detected or overwritten by another event of the same type. Whenever CFSMs are composed to a network, there is the additional restriction that all output interfaces are pairwise disjoint. The CFSM model is not meant to be used directly as specification technique by designers due to its relatively low level abstraction. Rather, the language provides an intermediate format for Hardware/Software-Codesign [Buc95], where specifications in higher level languages will be directly translated into CFSMs. Their semantics is defined by timed event traces. Since events are not instantaneous, besides their causality in these traces, further conditions (see [CGH$^+$93a]) are required to get meaningful descriptions. The CFSM model does not enforce any fairness restrictions.

In the real-time object modeling language ROOM [SGW94], the high-level behavior of an actor of a specific class is represented by a sort of extended state machines, termed

ROOMcharts, which are based on original Statecharts [Har87]. In contrast to μ-Charts, where both input/output interfaces and state-based behavior are included in one chart, actor behavior is shown separately from actor structure. In ROOMcharts, each non-initial transition must have an attached definition containing one or more port-signal combinations and an optional guard function that must return true in order to fire the transition. Different from μ-Charts, actions can be attached to both transitions and states (as entry and exit actions). However, as we will show in Section 2.5.3, entry as well as exit actions can easily be introduced as syntactic abbreviations. The action code may be empty or contain detailed C^{++} code. On the level of ROOMcharts, there is no notion of parallel composition. Communication between ROOMcharts of different actors is achieved by delayed (that is, strongly pulse driven in the sense of Focus [BDD$^+$93, BS97]) point-to-point communication. While parallel composition of ROOMcharts is not allowed, hierarchical structuring by means of decomposition is. Here, even inter-level transitions are possible. How for these a compositional semantics can be defined is discussed in [GSB98].

In contrast to the automata-based languages discussed so far, *I/O automata* [LT87, LT89, GSSAL93, LV95, LV96, Mül98] provide a model of distributed computation in *asynchronous systems*, where a strong analysis between a system and its environment is effected. This analysis captures the game-theoretic interplay between a system and its environment. Input actions here have the unique property of being enabled from every state, that is, for every input action there is a transition labeled with this action from every state. Hence, a system specified with an I/O automaton must be able to accept any input at any time. In μ-Charts, the situation is slightly different: though a μ-Chart can perform a reaction in every system state on every single input event, sometimes this reaction can be chaotic (see Page 9). This happens if and only if the system reaction for the specific input event has not been defined. In other words, there is no explicit transition labeled with the corresponding event. This chaotic behavior that stems from non-responsiveness can later on be refined by an appropriate calculus as we will demonstrate in Chapter 3. Besides enabledness, a further notion that plays a fundamental role in the model of I/O automata and that at the same time is contradictory to the notion of refinement defined in this thesis is fairness. Of course, it is possible to define a semantic notion of refinement for fair I/O automata based on game theory or temporal logic. However, we are interested in a notion of refinement for which we can deduce easy-to-apply syntactic rules for μ-Charts. It would be rather complicated to define syntactic refinement rules that leave fairness invariant. We are not aware of any existing syntactic calculus for I/O automata. Further note that for I/O automata a special kind of fairness is required [LT89]. Informally, an I/O automaton is fair, if for each of its transitions holds that if the transition is infinitely often enabled then it also is infinitely often taken. This is not true for non-deterministic μ-Charts.

1.2.2 Synchrony

In the following, we give a brief motivation why we use instantaneous feedback or, in other words, *perfect synchrony* [BG92] to model multicast communication. The notion of synchrony is overloaded in computer science. In the literature, it is applied to at least the following three different concepts:

Clock synchrony: Two concurrent components (inter-)act clock synchronously, if their processes advance in lock-step with a common, so to say "global", clock. Both components always make a step simultaneously with respect to this clock. This clock can be defined explicitly or implicitly by certain stimuli from the environment. The clock of a micro-processor, for instance, is a special case of an explicit clock with equidistant clock phases. However, in this thesis we consider the more general case, where timing intervals of the clock can be of arbitrary length. The transition from one clock interval to the next one is called *tick*. Often, there is also an extra signal that indicates the tick.

I/O synchrony: Whenever no time elapses between receiving the input and sending the output, we speak of I/O synchrony. In this case, input and output occur in the same clock interval and therefore are considered as instantaneous.

Message synchrony: In the case of message synchronous communication, the sender of a message is blocked if the receiver is not ready-to-receive.

Table 1.2 outlines a classification for some prominent examples that all are to some extent related to synchrony. Less known is CIRCAL, a process algebraic approach to describe the behavior of hardware devices. It is similar to CCS or CSP, but models concurrency naturally without recourse to the arbitrary interleaving used in these formalisms and so reconciles determinism and concurrency [Mil83, Mil84]. CIRCAL is flexible enough to model both clock synchronous and clock asynchronous behavior. Discrete time can be introduced easily by means of a special tick signal.

Statecharts are considered as a so-called *synchronous* language, where all system components described by Statecharts work together in lock-step, that is, driven by a global clock. As two different semantics are realized in STATEMATE, Table 1.2 contains two entries for clock and I/O synchrony, respectively. We already have discussed that reactive systems often have to satisfy the requirement of bounded response time. A further requirement is that for their implementation in an embedded system only *restricted memory* is at disposal due to both cost and space restrictions. Hence, reactive systems should be implemented as efficiently as possible. One possible design methodology to do this is to use a synchronous specification technique like Statecharts.

If a language is both I/O synchronous and clock synchronous we say that it is *perfectly synchronous*. The notion "perfect" here comes from the fact that the assumption that between input and output does not pass any time is an approximation of the reality.

Synchrony Type	Clock	I/O	Message
ARGOS	yes	yes	no
ESTEREL	yes	yes	no
LUSTRE	yes	yes	no
μ-Charts	yes	yes	no
STATEMATE Statecharts	yes/no	yes/no	no
CSP	no	no	yes
Combinational Hardware	no	yes	no
Sequential Hardware	yes	no	no
SDL	no	no	no
CIRCAL	yes/no	yes/no	yes

Table 1.2: Classification of synchrony

Clock Synchrony

In general, reactive systems consist of many different, cooperating and communicating components. For specification techniques for such systems it is therefore crucial that these interactions can be described in an easy-to-understand, implementation independently, and unambiguous fashion.

Formalisms like Petri nets [Rei86], CSP [Hoa78, HH83, HJH90], Occam [Sch88], Ada [Gon86] or I/O-Automata [LT89, LV95, LV96, GSSAL93], which are based on (clock) asynchronous communication, have in contrast to synchronous languages a number of disadvantages. The following unnatural behaviors can arise if reactive systems are specified with asynchronous languages [BG92].

Reactive systems are concurrent systems, where different reactions compete with each other. New inputs can reach the system before the current system reaction is terminated. Here, actions and communications of the current reaction compete with the actions and communications of the newly starting reaction. Choosing a clock asynchronous semantics, there is possibility to express when a message will reach its receiver. Thus, the communication delay cannot be determined a priori. As a consequence, clock asynchronous communication for reactive systems leads to the following problems.

The termination time of a reaction cannot be determined. The only practical solution would be to require that a reaction is atomic. This is, however, not a very elegant solution since the concept of atomicity is not supported by most clock asynchronous languages, and, even more important, it is not very realistic. In order to meet deadlines of real-time systems, reactions with low priority often have to be interrupted by reactions with higher priority.

If one uses clock synchronous description techniques like μ-Charts rather for specification than for simple low-level programming, it is unrealistic to make assumptions on communication delays already in the early system design phases. Remember that the concept of parallel composition and multicasting for Statecharts was introduced to avoid the state explosion problem. Parallel components are not necessarily implemented on hardware that operates in parallel. Hence, when specifying a reactive system with Statecharts, we are not able to say whether the multicast communication in the specification will later on really be implemented as physical communication among distributed processors. Moreover, even if it was safe to say that a specific communication on the specification phase also would be a physical communication on the implementation phase, the designer is usually not capable to say how much time this communication would take. One possibility to bypass this problem is to use an instantaneous communication mechanism.

If we used clock asynchronous communication, it would be much more difficult to define the concept of interrupt. With clock synchronous mechanisms, however, one is able to not only reason about interrupts and their effects, but in addition can distinguish between different levels of interrupts like weak or strong interrupts, for instance.

Furthermore, a synchronous semantics can be implemented with bounded memory since it provides no possibility to model, potentially unbounded, message buffers of asynchronously communicating system parts. Apart from this reason, there is a second one: asynchronous communication possibly cannot be transacted in one single state but may lead to a large number of intermediate states, which are unnecessary for synchronously communicating systems. However, note that though Statecharts components interact in lock-step, there are versions of semantics for Statecharts discussed in the literature, especially one of the two semantics implemented in the STATEMATE tool, which require intermediate steps within one interaction: as we will see in the remainder of this thesis, one system step often has to be divided into a chain of micro-steps.

In the remainder of this thesis, we will speak of synchronous systems whenever clock synchronous ones are meant unless explicitly noted. However, note that the term synchronous language refers to perfect synchronous languages as discussed in the following.

Perfect Synchrony

All aforementioned problems do not appear if we use *perfect synchrony* as communication principle [BG92]. Perfect synchrony assumes that each reaction is instantaneous, that is, it consumes no time. Of course, "to consume no time" must not be misinterpreted and needs some further explanations. This hypothesis is discussed in the literature in detail (see [HG91, BG92, Ber98], for instance). In the following, we will summarize the most important characteristics.

Perfect synchrony assumes an ideal reactive system, where each reaction is carried out by an infinitely fast machine. Thus, time only is consumed when the system resides in a certain system configuration. Transitions, however, do not take any time at all. Surely, this is an abstraction of reality, but simplifies controller programming. In addition, well-known, sophisticated implementation, optimization, and verification techniques from hardware design can be exploited [Ber96].

For reactive systems, it is important to know how much time elapses between input and the resulting output [HG91]. There are several approaches to solve this problem. First, for each step of the system the concrete amount of time can be specified. This is too troublesome and forces the designer to think about quantifying time from the very beginning. The approach we aim at is more abstract. Hence, we could fix the reaction time for each step to a finite, uniform delay. Though this approach is a bit simpler than the first one, it is still not abstract enough. This technique has further disadvantages. First, in practice, a fixed delay defines an upper bound for the reaction time of the implementation of the reactive system. Thus, single executions possibly have to be artificially delayed. However, real-time systems often have to react as fast as possible. Second, fixed execution times of a reaction impede the refinement of this reaction by a chain reaction. As refinement plays a central role in this work, we have to find another solution. Third, hierarchical decomposition could not be defined by parallel composition and communication if we assumed fixed reaction times. One possibility to remedy this is to define that each reaction takes an arbitrary positive amount of time. Of course, this approach now is abstract enough, unfortunately too abstract, because it introduces a lot of non-determinism and it is therefore impossible to prove any interesting system properties at this design stage.

The solution that remains is to assume zero reaction time. Here, we simply assume that the physical time the implementation needs for the execution of one system step is shorter than the rate of the incoming events. This assumption of perfect synchrony has the following advantages. First, all reaction times are known already on the stage of abstract specification. Second, reaction times do not depend on the concrete implementation. Third, these times are as short as possible and no artificial delays have to be introduced. Fourth, each reaction can be refined to several sub-reactions, as the timing behavior is not changed ($0 + 0 = 0$). These and further advantages of perfect synchrony also can be looked up in [HG91, Ber98].

However, we want to point out once more that clock synchronous languages do not necessarily base on the existence of an explicit clock that generates equidistant clock signals (ticks). Rather, the time intervals between ticks can be arbitrary. The essential point is that at any system configuration all transitions are fired synchronously with respect to a specific signal, maybe the tick.

Synchronous Languages

Synchronous languages have been introduced to make the programming of reactive systems easier. They are based on the principle that all parallel components share the same discrete time scale. This has several advantages. First, time reasoning is made easier, and second, interleaving based non-determinism (like in I/O automata) disappears. This makes program debugging and verification easier [CGP94].

Apart from the already discussed perfect synchronous language Argos, Esterel [BCG86, BG88, Ber91, Ber92, BG92, Ber93, SBT96, Ber96, Ber98] is a further candidate. Esterel is an imperative, parallel programming language, where statements can be divided into commands to manipulate local variables or signals. Similar to μ-Charts, there are no shared variables. Signals are used to express interaction of parallel components that communicate through instantaneous broadcasting. To this respect, μ-Charts and Esterel do not differ. However, there are conceptual differences between them: while μ-Charts are an abstract specification technique, Esterel is an imperative programming language. Non-determinism is not possible in Esterel programs, neither for components nor for composition. Hence, to check the determinism of an Esterel program requires subtle compiler techniques. Esterel provides commands like assignment, signal emission, sequential composition, case analysis, loops, exits and explicit parallelism that do not consume any time at all; the assumption is that they can be operated in zero-time. However, temporal expressions like watchdogs do consume time. The interrupt mechanism of Esterel is more complicated than the one of μ-Charts: instead of two, four different kinds of interrupt exist. Though Esterel is tailored for programming of both hardware and software, its focus is more on hardware.

Another high-level language for programming complex temporal behaviors and their translation into synchronous circuits is *YASL (Yet Another Synchronous Language)* [CHF97]. The precursor of YASL and its circuit translation has been developed without knowledge of the body of works on synchronous languages. Since their developers became aware of it, they have been influenced by Esterel. However, there are some minor differences between YASL and Esterel; a comparison is given in [CHF97]. For instance, the termination behavior of their parallel composition operators differ. In Esterel, the composition of two processes terminates if both terminate. In YASL, the composition already terminates if just one process terminates; the other is reset. Hence, both languages follow a different philosophy. While the composition operator in Esterel includes synchronization, the one of YASL includes preemption. Unlike Esterel, in YASL nested interrupts by the trap-exit mechanism are not possible. On the other hand, YASL contains a temporal projection operator that is not included in Esterel. However, there are possibilities to find workarounds for the description of both mechanisms in the other language, respectively.

In [Bou91], an extension of the C programming language, called *Reactive C (RC)*, is described. The main notions of RC directly come from Esterel. However, there are also some differences between Esterel and RC. A comparison is given in [Bou91]; we just give a brief overview here. Loosely speaking, RC is more general than Esterel in several aspects: similar to μ-Charts, conditions in RC are Boolean expressions and are not restricted to single tests for signal presence as in Esterel. Moreover, RC gives a way to explicitly break one instant into several micro instants and to join it again. Furthermore, the operator for parallel composition in RC is, in contrast to μ-Charts and Esterel, not commutative. Finally, the main difference to Esterel— and also to μ-Charts— is RC's communication mechanism: two different kinds of broadcasting are available. The first one is the instantaneous broadcasting from Esterel. However, the causality analysis here is not performed before runtime by a compiler, but the correct access to signals is dynamically checked. This dynamic analysis is not necessary for the second broadcasting type, where the reception of one signal is not in the same instant as its emission, but in the very next one.

In contrast to the imperative language Esterel, Lustre and Signal are data flow languages. Synchronous data flow languages restrict data flow systems to only those that can be implemented as bounded memory, automata-like programs in the sense of Esterel. Due to graphical data flow diagrams and operator networks Lustre [PC87, HCRP91] provides a comprehensive programming technique for software engineers. On the other hand, it is closely related to temporal logic [PH88] and therefore also is based on a precise and formal semantics. Lustre can even be considered as a subset of temporal logic [PH88, HCRP91]. Causality problems, already encountered in Esterel and μ-Charts, appear in Lustre as cyclic definitions, if a variable depends on itself. Lustre provides data operators that only operate on operands sharing the same clock, like addition, subtraction, and multiplication. Besides these operators, Lustre has four temporal operators: a previous, a followed by, a sample, and an interpolation operator. The first two operate on synchronous flows, where all data elements share the same clock. The last two, however, are used to down-sample to a slower or interpolate to a faster clock. Hence, similar to Esterel, Lustre also provides a *mulitform concept of time*, where more than one clock is relevant for the overall program. Though more than one clock is around, all of them can be down-sampled to the *unique basic clock* of the specific program.

However, Signal does not have one unique minimal basic clock. Thus, in contrast to Signal, Lustre does not allow basic clock time intervals to be split into smaller ones. Apart from the data operators as in Lustre, Signal provides three temporal operators, a delay, an extraction (similar to the sample operator in Lustre), and a deterministic merge operator. The latter specifies the union of two clocks. Though Lustre and Signal are both declarative languages, they differ, apart from the just mentioned points, significantly: Lustre is a *functional language*, where any operator defines a function from its input sequences to its output sequences, whereas Signal is a *relational language*. A Signal program defines a relation between input and output flows. Hence, the way an output flow is used may also constrain the input flow. Thus, any Signal component induces its own constraints. The consistency and completeness, that is, the existence of a unique solution for the conjunction of all constraints must be checked by a compiler. As clocks play a much more important role in Signal than in Lustre, this compiler is based on a rather complicated clock calculus.

1.2.3 State Based Refinement

Josephs [Jos88] presents a state-based approach to CSP including a sound and complete collection of simple rules by means of which one process can be shown to be a refinement of another and various CSP operators are investigated. These rules are based on the idea of downward and upward simulation [Bro93]. Soundness and completeness are achieved by exploring a normal form for processes.

For the state-based, behavioral specification of non-deterministic components in object oriented software systems the theory of *spelling automata* is developed in [Rum96]. For these automata, a refinement calculus is defined. In contrast to μ-Charts, the automata presented in [Rum96] do not provide any composition or decomposition mechanism and do not have a perfect synchronous semantics.

While the approach in [Rum96] introduces the notion of refinement for object oriented systems quite formally, [CHB92] only outlines a pragmatic and informal approach for sub-typing. To this end, so-called *Objectcharts* are introduced that combine object-oriented analysis and design techniques with Harel's Statecharts. Since Objectcharts describe the behavior of a class, they can be used to determine whether a proposed inheritance conforms to sub-typing. A class here can inherit from another in two ways: the descendant class can add new behavior through the addition of new services or by a re-definition of an existing service. Hence, in [CHB92], an inheritance is a sub-type inheritance if and only if the specification of the parent holds also for the descendant, although the descendant may add behavior. This is in contrast to μ-Charts, where the refined chart is not allowed to add behavior, but just can reduce the non-deterministic behavior of the original chart.

In [Kle97], *reactive transition systems* are presented that are similar to I/O automata, but here both input and output can occur on one and the same transition. Also different from I/O automata is the concept of explicit time actions, where the time is incremented. For these transition systems exists a formal semantics, which provides a basis for formal refinement. In [Kle97], upward and downward simulation are defined. The resulting calculus is similar to the one of [Rum96], but provides more general techniques: among others, rules for sequential concatenation of transitions are presented. The presented rules and their soundness proofs are relatively complex, since transitions are not taken in zero-time as in the case of the perfect synchrony hypothesis. Rather, each transition consumes a positive amount of reaction time.

In [EGKP97], a categorial approach to define refinement morphisms for a subset of STATEMATE Statecharts is presented. This contribution was developed independently of and simultaneously to the refinement calculus for μ-Charts. Therefore, both approaches do not have much concepts in common. Transformations are studied in analogy to techniques for graph transformation systems and Petri nets. With the approach in [EGKP97], only two refinement techniques are possible: decomposition of automata and parallel composition, which are there called horizontal and vertical structuring. However, for these kinds of refinement no syntactic and therefore for the software engineer applicable rules are given, and modifications of single automata are not possible at all. This approach suffers from a further weakness: since the semantics of Statecharts is not defined in terms of input/output histories, but as "runs" of subsequently enabled system states, also their notion of refinement is not based on reducing non-deterministic behavior. Rather, in order to define behavior, compatibility for every system state of the refined chart the corresponding state of the original chart has to be constructed. Hence, refinement cannot be defined by restricting the set of possible input/output behaviors, but must be defined by morphisms that relate state spaces of refined and original charts. The idea is that a state of the original chart is always reflected in one or more corresponding states of the refined chart.

A notion of behavior preserving transformations for Statecharts based on morphisms is also very briefly discussed in [Per95]. However, these discussions are merely based on a negation-free variant of Statecharts with an event-structure-based compositional semantics of so-called timed configuration systems [Per93]. Hence, they are based on a very specific semantics, and no syntactic transformation rules are given at all.

In [US93, US94a], a process algebraic semantics for Statecharts via state refinement agreeing with the semantics of [PS91] is presented. In particular, a translation of Statecharts into a process algebra with state refinement is provided. The semantics of a Statechart is given by the labeled transition system [US94b] of its translation. This approach represented one of the first attempts to capture hierarchically structured state space within a purely process algebraic setting.

1.2.4 Formal Verification with Statecharts

In [Day93], an operational semantics for STATEMATE Statecharts has been formalized by embedding them in the logical framework of an interactive proof-assistant system called HOL. HOL is used in combination with a binary decision diagram based verification tool, and a model checker for Statecharts, which tests whether an operative Statecharts specification satisfies a descriptive requirement specification. The target language of HOL is a subset of higher order logic that can be informally regarded as a functional programming language. The model checking procedure has been integrated as a simple higher-order logic function which executes the Statecharts' semantics. Combining a model checker with a theorem prover allows us the use of mathematical reasoning techniques, like induction and abstraction, to prove results beyond the capacity of a model checker (see also [Mül98]). Furthermore, properties of the semantics itself, like correctness of definitions, can be directly verified with the theorem proofer.

Further scientific work on abstraction techniques that are suitable for Statecharts can be found in [Kel96].

Independently from [Day93], further investigations on integrating STATEMATE Statecharts and model checking have been presented in [HSD+93]. Here, requirement specifications are expressed by timing diagrams, a graphical formalism for a subset of propositional linear temporal logic [BW98].

In [PS97b], we have demonstrated how a Statecharts semantics with instantaneous multicasting can be model checked using a precursor version of μ-Charts. This precursor differs in both syntax and semantics from the version that is presented in this thesis. In particular, parallel composition and multicasting have not been integrated into a single operator, and hierarchical decomposition has not been defined as syntactic abbreviation in [PS97b]. In the semantics, we have explicitly modeled chain reactions caused by instantaneous feedback as the least fixpoint of a transition relation. This chain of transitions has been embedded into the outer transition relation that described the observable behavior. The well-known causality conflicts that arise under instantaneous feedback from negative trigger conditions have been resolved semantically through oracle signals in [PS97b]. However, the approach presented in this thesis is more general, mathematically easier, and therefore also more efficient for formal verification. The semantics of instantaneous multicasting now is not defined as a least fixed point of a chain reaction (this view is merely necessary for distributed implementation of deterministic charts; see Section 5.1). Instead, the semantics here includes not only the least one, but all fixed points. Hence, the least fixed point is not explicitly computed for yet under-specified charts in this semantics, and therefore we do not need oracle signals, which predict the presence or absence of the corresponding signal, any longer. Since for formal verification one step now does not have to be partitioned into micro-steps, we get a more efficient model checking procedure.

1.2.5 Code Distribution

The code distribution proposed for SIGNAL is based on the structure of the source program [Hal93b, BG97]. Ideally, from a SIGNAL program that consists of a number of data flow statements composed in parallel, one would like to compile all statements independently and run them on distributed processors. However, not every partition leads to distributed programs that have the same semantics as the original one. Often code restructuring measures have to be taken. We will discuss this further in Section 6.2.

Another approach to generate distributed code, which initially was developed for LUSTRE, is presented in [Hal93b, Gir94, CGP94, CFG95, CCGJ97]. As it works on the so-called object code OC, a common intermediate format for Argos, ESTEREL, and LUSTRE, it also is applicable to Argos and ESTEREL. An OC program is a finite deterministic automaton with a finite memory for performing operations over infinite types [CGP94]. The principle assumption of the code distribution algorithm for OC is that each site (processor) is responsible for the computation of some variables of the automaton. The basic algorithmic idea is as follows: first, the code of the entire automaton, i.e. the overall OC program, is replicated on each site. Second, in each replication, the instructions (on the transitions) that do not concern the considered site are deleted. Then, sending and receiving statements are inserted in order to communicate values of the now only locally available variables. Finally, auxiliary communication has to be introduced to synchronize the distributed code replications. This approach differs significantly from the one we propose in this thesis. While the goal of the distribution method for OC is to localize computations of single variables, we aim at the distributed implementation of entire sequential automata. In order to keep the memory consumption of the overall reactive system as small as possible, we do not replicate any parts of the code. In the OC approach, however, the control structure as a whole is replicated several times.

The problem of partitioning a program into several parts and to implement it on a possibly heterogeneous target architecture, consisting of both hardware and software is generally known as Hardware/Software-Codesign. A good, but no longer complete overview of relevant literature can be found in [Buc95], and a general introduction in this topic is given in [RB95, DS96].

1.3 Results in a Nutshell

In this section, we summarize the most important aims and results of this thesis:

1. A lean, non-deterministic Statecharts dialect, μ-Charts, is presented. In contrast to related approaches, for this dialect suffice three syntactic constructs: Mealy machines, signal hiding, and a composition operator including multicast communication. Concepts like hierarchy, inter-level transitions, entering, exiting, and instate signals are eliminated in the core language. These constructs are defined by means of syntactic abbreviation. Additional semantic concepts are not necessary.

2. For μ-Charts, a precise, formal semantics is developed, where under-specification is mathematically treated as non-determinism.

3. Pathological specifications that stem from signal causality conflicts are discussed. Due to non-determinism, in μ-Charts some of these examples yet have a well-defined semantics whereas related languages have to reject these specifications as not being well-defined from the very beginning.

4. A framework for behavioral refinement for Statecharts-like languages is provided. Apart from the related literature discussed in Section 1.2.3, we are not aware of other approaches in this direction. A formal refinement calculus is given. The soundness of all refinement rules is proven. In particular, it is demonstrated under which circumstances hierarchical decomposition indeed can be considered as behavioral refinement.

5. A translation scheme from the formal semantics for μ-Charts to the input languages of model checkers is given. As an example, we provide translations to the input languages of two wide-spread model checking tools SMV and μcke. This enables efficient formal verification of safety critical system properties for specifications in μ-Charts. In contrast to more complex dialects, model checking is very efficient in our case.

6. Problems that arise when a μ-Chart specification is partitioned with the aim of distributed implementation on a processor network, where each part is realized as finite state machine, are discussed and solutions are proposed. Restrictions for practical application are pointed out.

7. The allocation problem for μ-Charts on a processor network is formalized by integer linear programming.
8. The distributed implementation of multicasting is described mathematically. Guidelines to achieve efficient communication schedules are given.
9. An approach to realizing a μ-Chart specification as a single finite state machine is proposed. Problems that arise when one wants to implement the same specification on a processor network are outlined. In addition, restrictions that are required in practice to obtain feasible distributed implementations are pointed out, and analysis techniques to detect these restrictions are presented.

1.4 Outline of the Thesis

This thesis is structured as follows. First of all, in Chapter 2 we introduce the visual formalism μ-Charts. We start with the description of our running example, a central locking system for cars. This example will support the reader throughout the entire thesis. Then, core syntax and semantics of μ-Charts are presented. After that, we discuss some pathological cases that potentially occur when dealing with instantaneous feedback. Finally, we show how more complex syntactic constructs like hierarchy and inter-level transitions can be defined as syntactic sugar of the core syntax.

Chapter 3 deals with refinement techniques for μ-Charts. We develop a calculus for behavioral refinement that is based on the core syntax. We give a set of syntactic refinement rules and prove that these rules are sound.

Chapter 4 contains a translation scheme from μ-Charts to the input languages of two symbolic model checkers, μcke and SMV. By the aid of our running example, we show that model checking is efficiently carried out by these tools.

In Chapter 5, we first discuss problems that can occur when a μ-Chart specification is partitioned in order to be implemented on a network of communicating processors. Then, we formalize the allocation problem by an integer linear program. We discuss two different allocation strategies. In the remainder of this chapter, we analyze the scheduling of messages in a processor network, which implements a μ-Chart specification.

In Chapter 6, we present an implementation scheme for finite state machines with μ-Charts. We first introduce a strategy how to implement a μ-Chart specification by a single state machine. These concepts are extended to distributed synthesis with μ-Charts in the rest of this chapter. We here mainly discuss problems that possibly occur when a specification in μ-Charts is distributed on a processor network and point out restrictions that are necessary to obtain realizable implementations.

Section 7 concludes the thesis and gives some outlook for future work.

2 Specification

Statecharts are a graphical description technique for the state-based, behavioral specification of control-oriented systems. In this chapter, we introduce a Statecharts-like language, termed μ-Charts, which we have developed as basis for design and distributed implementation of reactive systems. The chapter is organized as follows. First, we give an informal introduction to Statecharts and related approaches. Then, we present the running example of the thesis, an abstract version of a realistic central locking system. Third, the core syntax of μ-Charts is illustrated. The syntax section is finished with a second example. Fourth, the stream semantics for the core language is formalized. Finally, the chapter concludes with syntactic extensions for μ-Charts.

2.1 Introduction to Statecharts-like languages

The graphical specification language Statecharts was developed in [Har87] for the description of reactive systems. They combine the operational notions of Mealy machines with graphical structuring mechanisms to concisely describe large state spaces. In recent years, extensive reactive systems have been developed using this graphical formalism or related approaches [LHHR94]. In the sequel, we will first informally introduce the basic ideas that are common for all Statecharts-like languages and then concentrate on our Statecharts dialect, termed μ-Charts.

The main concept of Statecharts are sequential automata. These automata consist of states and transitions. An automaton's state denotes a section of a complex state of a reactive system. Transitions connect those states that describe consecutive system states. A transitions can be labeled with a pair, consisting of the condition that must be fulfilled in order to establish the transition and an action that specifies more detailed system behavior when taking the transition. Most graphical notions follow the convention that this pair is separated by a slash, *condition*/*action*, and we adopt this notation for μ-Charts, too.

To enable description of practically relevant systems, these automata can be composed to larger specifications using two principal structuring mechanisms: parallel composition and hierarchical decomposition. Here, the basic assumption is that automata composed in parallel proceed in lock-step with respect to a common clock. Without any further assumptions, these automata do not interact at all. However, if specified by the user, they also can interchange messages in order to influence the behavior of each other. Reactive systems possibly have complex system states. Hence, single automata states may not be appropriate for a detailed description of these systems, and more elaborated techniques are needed. Statecharts enable to further structure single states of sequential automata by hierarchical decomposition. The Statechart that describes the systems behavior in a specific state in more detail is simply (graphically) inserted into this state. The behavior of an hierarchically decomposed Statechart is comparable with the one of procedure calls in an imperative programming language. An automaton of an higher level of hierarchy "calls" sub-routines, that is, Statecharts of a lower level, whenever a decomposed state is entered. After this general introduction to Statecharts, we now give some informal explanations directed to μ-Charts.

A μ-Chart is a specification of a component in a reactive system that displays cyclic behavior. In each cycle, input is read, output is emitted, and the component changes its configuration, consisting of data and control state. In this respect, μ-Charts are similar to ordinary Mealy machines [Bro97a] or, synonymously, sequential automata. They have a finite set of control states, one or more initial control states, an input and output interface (alphabet), and a transition relation defined as a relation over current configuration, input signals, output signals, and next configuration. In contrast to Moore machines, the current output of a Mealy machine depends not only on the current control state, but both on current control state and on current input.

For a reactive system, the input alphabet can be regarded as a set of signals generated either by the system's environment or the system itself. Similarly, the output alphabet usually consists of events (= sets of signals) that influence the future behavior of both the system's environment and other system components.

The time between two cycles is non-zero and finite. Between two cycles more than one signal that is a relevant input for the system can occur in the environment. The idea is then that these signals are all collected in a set, and this set, which we call *event*, is used as input to the component. More than one output signal can be produced in one such cycle. Hence, the output is then a set of signals. To ease the writing of specifications, transition labels on the syntax level are not simply signal pairs, but consist of a Boolean expression over input signals and local variables, the so-called trigger, and an action. The action itself is described by a small imperative language. Of course, the action also consumes time in practice. However, our assumption is that time passes in the control states and that transitions are fired instantaneously. Hence, time passes in states and not on transitions. Theoretically, signals from the environment could be lost if they occurred during the system reaction. If, however, the system is sufficiently fast in comparison to the environment, we can disregard this problem [Ber98], and arrive at the synchronous time mode. Since system reactions are assumed to run infinitely fast (see Section 1.2), they just divide time flow into finite intervals; at each interval border there is a system reaction where input is read and output produced.

Even when input and output are abstracted to sets of signals, Mealy machines are not always adequate as a specification formalism for large reactive systems. The reason is simply that using state machines for specifications often yields diagrams that are too large to be written down and comprehended. For this reason, Statecharts were suggested by Harel. In the remainder of this thesis, we will use the notion “Statecharts” to express this principle language concept for a visual formalism, that is, to combine Mealy machines by parallel composition, broadcasting, and hierarchy.

These principle language concepts are also included in μ-Charts. As already discussed in Section 1, original Statecharts suffer from a number of weaknesses because they offer various possibilities to specify systems that result in non-modular specifications, are difficult to understand from a methodological point of view, and possibly yield inefficient implementations.

Compared with the graphical language of STATEMATE Statecharts, the syntax of our dialect does not include some notations which are included in the original Statecharts. Notable notations are static reactions, hierarchical decomposition, inter-level transitions, references to state names, and global variables. Some of them are relatively easy to include as syntactic sugar, others are not. Global variables, for instance, have been omitted on purpose since they do not allow the definition of a compositional semantics.

Basically, Statecharts-like languages are defined inductively as follows. Notice that our description differs from the one presented in [Har87].

- A Mealy machine where input and output alphabet are powersets of signals is a Statechart.
- The parallel composition of two Statecharts is a Statechart. Parallel composition is the main technique to reduce the number of states needed for the specification. To express communication between charts that are composed in parallel, broadcast communication is used.
- A Mealy machine, where states are further decomposed by Statecharts is itself a Statechart. This construction is called *hierarchical decomposition* and is the main technique to reduce the number of transition arrows needed for the specification.

In the rest of this section, we explain parallel composition and hierarchical decomposition in more detail. Other language concepts that are not discussed here can be looked up, for instance, in [Har87]. In the remainder of this thesis, we will see that hierarchical decomposition can be syntactically defined by means of parallel composition and communication.

Parallel Composition and Communication

The main technique to reduce the state complexity in the specification is the parallel composition of two or more state machines. The state space of the parallel composition is the algebraic product of the state spaces of the machines composed in parallel, and therefore, the size of the specification grows only linearly.

Intuitively, the components composed in parallel operate in lock-step: for each input signal set, each component makes a transition and emits an output signal set. The output of the composition is the union of the component outputs, but charts composed in parallel do not interact.

However, since reactive systems mostly consist of many components that have to interact, only rarely can a component be specified by the *independent* composition of smaller specifications. In practice, the state machines composed in parallel often have to communicate. Typically, the standard communication mechanism that is used by Statecharts-like languages is *broadcasting*. We also adopt this mechanism in a restricted form (multicasting) as it fits well with the final target architecture we aim at: a number of electronic control units that are interconnected via one or more busses. Communication on a bus is reflected by broadcasting. When a state machine emits an output signal, this signal is visible to the other machines of the specification; there it can then cause further outputs, and so on. Thus, communication can lead to chain reactions of transitions. In these cases, a system step is further divided into a series of micro-steps. However, only the result of the chain reaction with the accumulated output is then the visible reaction of the system.

Together with our synchronous time model, communication can lead to causality conflicts: for example, assume that a machine A_1 produces an output b if and only if it receives input a, machine A_2 produces output a only if it receives input b, and chart A is the parallel composition of A_1 and A_2 with internal communication of a and b. When neither a nor b is input from the environment, should the output of the composed chart be the set $\{a, b\}$ or the empty set? In Sections 2.4.4 and 5.1, we will discuss how pathological examples, that is, specifications with causality conflicts including the one just outlined are treated with our μ-Charts semantics.

Intuitive and at the same time mathematically sound semantic definitions of communication are quite intricate, and they are the major difference between the various Statecharts dialects found in the literature [vdB94]. The communication semantics of our μ-Charts language is defined in Section 2.4. We have decided to use one single syntactic construct to express both composition and communication together. Pure parallel composition without any communication is then just a special case of this more complex composition operator (see Section 2.4.3).

In Chapter 1, we have already discussed that the usage of a synchronous description technique for the specification of reactive systems has the advantage that it leads to systems with bounded memory usage. However, if the communication semantics is too complicated, this advantage can quickly disappear. Hence, defining a semantics for broadcasting, care has to be taken that the number of micro-steps that build one system step is restricted. One remedy is to require that each Mealy machine can fire just one single transition in each system step. Though this may, in contrast to STATEMATE, prevent systems from infinitely long chain reactions with alternating transitions, nevertheless non-terminating sequences of micro-steps can appear in the case of pathological specifications (see Sections 2.4.4 and 5.1). Fortunately, such specifications occur only very rarely, and we will show later how system steps can be restricted to a finite number of micro-steps.

Note that if a system is implemented on a single processor, details of these chain reactions are not visible to the system's environment since the transition relations of communicating charts are combined to one single transition relation which is used for the centralized implementation (see Section 6.1). We would like to mention that only if the system is partitioned and implemented on a distributed target architecture, the chain reactions are visible as messages on the communication medium by which the distributed processors are connected. To take this effect into account, we explicitly use a fixed point semantics to model distributed implementations of deterministic μ-Charts. This semantics will be presented in Section 5.

Hierarchical Decomposition

The second technique to reduce the syntactic complexity of a specification is the introduction of hierarchy: groups of states with transitions which labels and destination states are identical can be gathered in a sub-chart (Figure 2.1). In our graphical descriptions, we follow the convention that basic control states are denoted by ellipses, hierarchically decomposed charts by rectangular boxes, and initial states by double frames (warning: do not confuse this with the graphical notation for final states in finite automata). This way, the number of transitions needed for a specification can be reduced. The decomposed state is termed *controller* or, using notation known from flip-flops, *master* and the chart within the box *controllee* or *slave*.

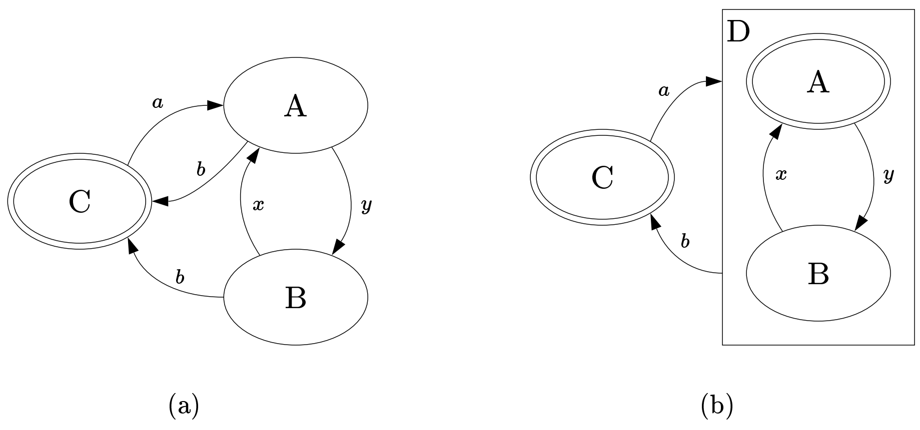


Figure 2.1: Hierarchical decomposition

In μ-Charts, as in most Statecharts dialects, however, hierarchical decomposition is not only employed to cluster states with identical transitions. Instead, hierarchy is also used to model preemption (see Section 2.5.1). In Section 2.5.1, we will also define hierarchy by means of syntactic abbreviation. We will motivate that hierarchical decomposition is, like pure parallel composition, merely a special case of a more complex operator that includes both parallel composition and communication.

Hierarchical decomposition, too, gives rise to interesting semantic questions. One of these questions is, for example, what the proper behavior should be when the signal necessary to leave the controllee, that is, to withdraw the control from it, is produced by the controllee itself? We will discuss this phenomenon, which we call *self-termination*, together with preemption mechanisms also in Section 2.5.1.

2.2 Running Example: Central Locking System

As the running example we use a simplified specification of a central locking system (CLS) for cars. This example was inspired by a case study from car industry. The principal structure of the CLS is sketched in Figure 2.2 and the corresponding μ-Chart is pictured in Figure 2.3; it specifies the locking system of a two-door car.

In our graphical syntax for describing the structure of a system we follow the convention that each sequential automaton is represented by a box, including an additional name for the automaton and its input and output interface, denoted by transparent and filled circles, respectively. In the system structure, we also describe the signal flow between different automata by arrows, which connect output and input interfaces of possibly communicating charts.

Table 2.1 shows the signals used for the specification. We distinguish between signals that are input from the environment, so-called external input signals or stimuli, and signals that are generated by the system itself (internal signals). Recall that ellipses denote basic states of sequential automata while boxes denote states that are hierarchically decomposed by other μ-Charts. Double frames denote initial control states. All signals that potentially can be broadcast between the composed charts are collected between the two dashed lines that express parallel composition. In our syntax, also signal hiding is expressed by an explicit operator. It is graphically indicated by the box on the bottom of Figure 2.3.

Our central locking system consists essentially of three main parts: the Control and the two door motors MotorLeft and MotorRight. These parts are composed in parallel. Note that names for sequential automata like Control, MotorLeft, and MotorRight are merely syntactic annotations in our graphical syntax to enhance reading and do not have any influence on the semantics of the specification. The default configuration of the system is that all doors are unlocked (Unld) or locked (Locked) and both motors are Off. Having more than one initial state in a chart expresses that at the current stage of development the designer is not capable of deciding which one of the possible initial states will be the initial state of the final implementation. This

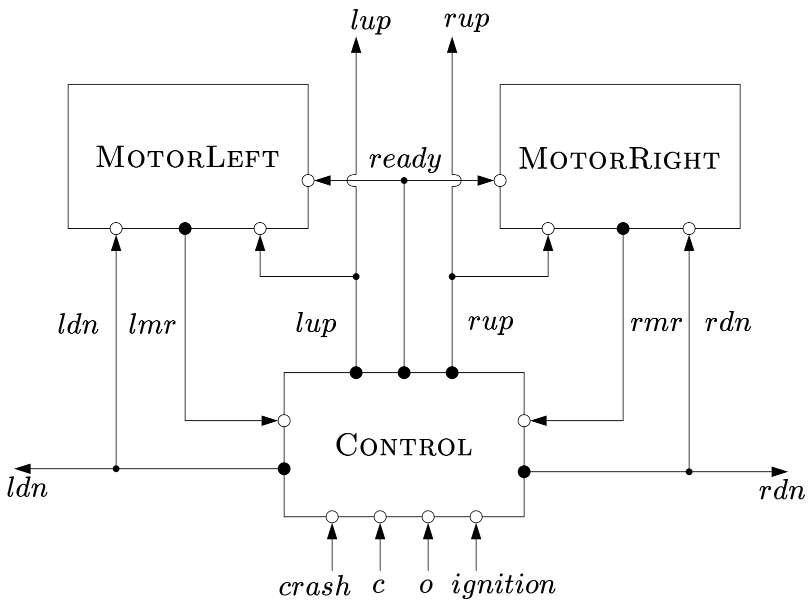


Figure 2.2: Central locking system — structure

is still under-specified, but can be made more concrete in a later design step. The mathematical means to express under-specification is non-determinism. The process of making a specification more concrete is called *refinement*. Section 3 addresses this problem.

The Control is the basic automaton of the central locking system. Initially, the Control is in its Normal state. The system remains in this state until a *crash* signal occurs. The crash sensor is a sensible device that also can be actuated if the car is parked, that is, without having a crash. Thus, we specify that the locking system only then can react on this signal when the *ignition* is on. In such a case, automatically two signals *lup* (for "left motor up") and *rup* (for "right motor up") are generated to cause the motors to unlock the car doors. This feature of unlocking all doors in case of a crash eases the rescue of passengers who have been hurt in an accident.

The driver also can unlock the car if the central locking system is in its Normal state. To specify this, the Normal state has been decomposed by an extra automaton. The driver can unlock or lock the doors either from outside by turning the key or from inside by pressing a button. Locking and unlocking the doors leads to complex signal interactions. Both actions generate the external signals *c* (for "close") or *o* (for "open"). For an overview see Figure 2.2. The Control generates the internal signals *ldn* and *rdn* and enters its locking state Lockg, which is decomposed by the automaton in Figure 2.4.

Instantaneously, influenced by *ldn* and *rdn*, respectively, both motors begin to lock the doors by entering their Down states. These states are decomposed by the sequential automata pictured in Figure 2.5 (in the figure, substitute *xmr* by *lmr* for MotorLeft and by *rmr* for MotorRight). Thus, the motors are additionally in their Start states.

Signal	*Meaning*	*Source*
crash	Crash sensor	External
o	Open/Unlock car doors	
c	Close/Lock car doors	
ignition	Ignition on	
lmr	Left motor ready	Internal
rmr	Right motor ready	
lup	Left motor up	
ldn	Left motor down	
rup	Right motor up	
rdn	Right motor down	
ready	Un-/Locking process ready	

Table 2.1: Signals used in the central locking system

As the speeds of the motors depend on external influences like their temperature, each motor either needs one or two time units to finish the lowering process. Thus, also this part of the specification features a non-deterministic behavior. Only when both motors have sent their ready messages *lmr* (stands for "left motor ready") and *rmr* ("right motor ready"), the CONTROL enters the BOTH state and produces the signal *ready*. The effect of this signal is twofold: on the one hand, the CONTROL terminates itself immediately and enters the LOCKED state. On the other hand, also both motors are triggered by this signal and are switched OFF.

Whenever the crash signal occurs and the ignition is on, the CONTROL changes from the NORMAL state to the CRASH state and generates the signals *lup* and *rup*. In Section 4, we will prove that the crash signal indeed causes the doors to unlock.

2.3 Core Syntax and Informal Semantics

In this section, we formally introduce the essential concepts of our Statecharts dialect. Having given an informal explanation at the beginning of this chapter, we assume the reader to be familiar with the basic ideas of Statecharts now and refer to [Har87, HN96] if a more detailed introduction is necessary.

In this thesis, all elements in the set $\mathcal{S}$ of μ-Charts can be built from only three syntactic constructs: non-deterministic sequential automata, signal hiding, and parallel composition including multicast communication between charts composed in parallel. Other syntactic concepts, which are of interest for Statecharts, like pure parallel composition without communication and hierarchical decomposition, can be derived from these three constructs (see Section 2.5).

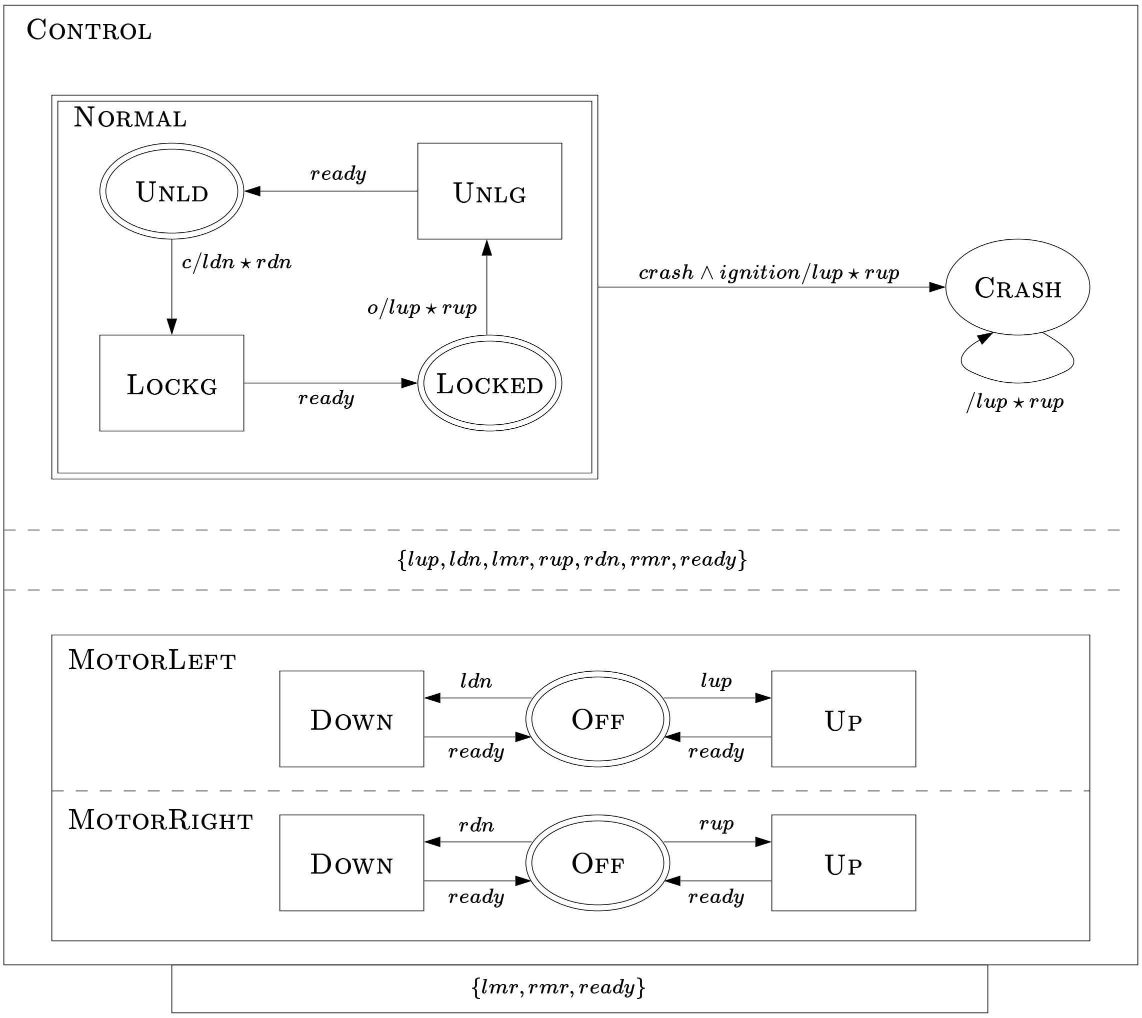


Figure 2.3: Central locking system — behavior

All syntactic constructs provide syntactic input and output interfaces. For component $S \in \mathcal{S}$, elements in $In(S)$ and $Out(S)$ are termed *input* and *output signals*, respectively. If we do not care whether it is input or output, we only say *signal*. Each element x

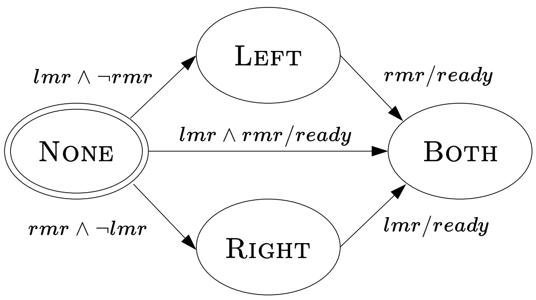


Figure 2.4: Decomposition of LOCKG and UNLG

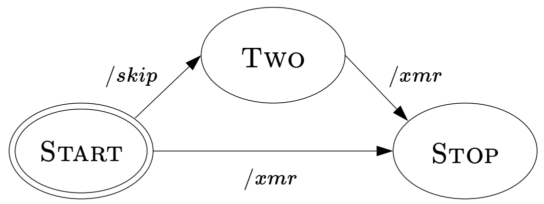


Figure 2.5: Decomposition of Down and Up, $xmr \in \{lmr, rmr\}$

in $\mathcal{I}(S) =_{df} \wp(In(S))$ and $\mathcal{O}(S) =_{df} \wp(Out(S))$ is called an *input* and *output event*, respectively. As a consequence, input and output events are sets of signals in $In(S)$ and $Out(S)$, respectively. If we abstract from input or output, we simply speak of *events*. For each signal s we say s is *present in (the event) x* if and only if $s \in x$. Otherwise, we say that it is *absent in (the event) x*.

2.3.1 Sequential Automata

In the definition of sequential automata, we use the following syntactic, pairwise disjoint sets: M a set of signal names, *States* a set of control states, and V a set of variable names. The 7-tuple

$$(I, O, \Sigma, \Sigma_0, V_l, \varphi_0, \delta)$$

in the sequel abbreviated to A, is an element of $\mathcal{S}$ if and only if the following constraints hold:

1. $I \subseteq M$ is the input interface.
2. $O \subseteq M$ is the output interface. We assume I and O to be disjoint.
3. $\Sigma \subseteq$ *States* is a non-empty finite set of all control states of the automaton.
4. $\Sigma_0 \subseteq \Sigma$ represents the set of initial states.
5. $V_l \subseteq V$ is the set of local (integer) variables of the automaton.
6. For each initial state $\sigma_0 \in \Sigma_0$ the function $\varphi_0(\sigma_0) \in V_l \to \mathbb{Z}$ initializes the local variables. We abbreviate the possible valuations [Win93] $V_l \to \mathbb{Z}$ of automaton A to $\mathcal{E}(A)$.
7. $\delta : \Sigma \to \wp(Bexp(I+V_l) \times Com \times \Sigma)$ is the finite state transition relation that maps a state to a finite set of triples, where each triple consists of a Boolean expressions $Bexp(I + V_l)$ over the algebraic sum of I and V_l as transition predicate (guard, pre-condition, trigger) paired with a command $com \in Com$ and the successor (control-)state. This relation will be explained in more detail in the following. Note that δ is neither total nor deterministic.

In our graphical syntax, the interfaces I and O are omitted because they can easily be derived from the transition labels. We assume that I and O are the set of those signals that occur on transition triggers and commands, respectively.

In our semantics, a transition takes place in exactly one instant, that is, between two consecutive signals of the common system clock. In a specification with several automata working in parallel, all automata fire their transitions simultaneously; for parallel automata, all transitions taken are assumed to occur in the same instant. Recall, however, that every single sequential automaton only is allowed to make one transition in one instant.

Transition Labels: Trigger Conditions and Actions

In the sequel, we characterize the transition relation δ for sequential automata in more detail. We introduce the transition syntax of μ-Charts with local variables and integer-valued signals. In this thesis, we only consider integer numbers as data values. Statecharts in STATEMATE support both integer and floating point arithmetic. Our semantic model also is easily extendible to a formalization of floating point arithmetic, based on a suitable underlying data type model. However, notice that model checking can only be carried out for finite data types. Thus, we found it both reasonable and justifiable to limit ourselves to integers. In practice, these can be easily restricted to finite intervals, which is a prerequisite for formal verification with model checkers. Further finite data types such as enumerations etc. can be introduced straightforwardly.

In this context, arithmetic expressions $a \in Aexp$, Boolean expressions $b \in Bexp$, and commands $c \in Com$ have the form:

$$
\begin{array}{rcl}
a & ::= & n \mid Y \mid a_1 \,\mathsf{binop}\, a_2 \\
b & ::= & \mathsf{true} \mid \mathsf{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid s_i \mid \neg b \mid b_1 \wedge b_2 \\
c & ::= & \mathsf{skip} \mid Y := a \mid s_o \mid c_1 \star c_2
\end{array}
$$

In the syntax of transitions, we follow the convention that $n \in Int$ is an integer, $Y \in V_l$ is a local integer variable, $s_i \in I$ is an input signal, and $s_o \in O$ an output signal. In our graphical notation, we separate condition and action (command) on transitions by a slash: b/c. Though we merely have defined Boolean expressions $b \in Bexp$ using negation and conjunction, we also can introduce all other binary operators in our syntax as abbreviations. Here binop stands for any of the usual binary operations on integers, such as addition, subtraction or multiplication.

In the sequel, we informally introduce the semantics of transitions. The meaning of Boolean and arithmetic expressions is straightforward whereas the meaning of commands needs some explanation. $Y := a$ assigns the value of the arithmetic expression a to the variable Y. In this context, Y is local with respect to the automaton that contains the transition. The command s_o sends the output signal $s_o \in O$ in the instant in which the transition is taken to the automaton's environment. Single skip commands can be omitted in the graphical notation. For example, in Figure 2.3, we simply write ldn and

lup instead of ldn/skip and lup/skip, respectively. Triggers that are equivalent to the Boolean value "true" can be omitted (see Figure 2.5) as well.

The command $c_1 \star c_2$ stands for the parallel composition of the two commands c_1 and c_2. This construct has to be treated with some care. In traditional Statecharts, when two or more commands want to change the same variable in the same step, so-called race conditions [HN96] can occur, which have to be detected by STATEMATE's simulation and dynamic test tools because the values of the variables are unknown before runtime. To us, this seems complicated and superfluous: message passing and shared memory are orthogonal concepts, and in principle, each of them can be used to simulate each other. As a consequence, we have decided to follow another approach in order to get a non-ambiguous meaning and to avoid dynamic analysis needed to detect race conditions.

Hence, in earlier attempts at providing a semantic foundation for Statecharts [Sch96a, Sch96b], we have chosen sequential execution of commands $c_1; c_2$ instead of parallel composition. However, the program counter needed for sequential execution makes both hardware implementation and formal verification more complex since this way each transition reaction would run down in a series of subsequent steps. Apart from these technical reasons contra sequential composition, there are also methodological draw-backs: in our opinion, the way of composing charts in Statecharts is not easy to understand for software engineers because of the subtle communication concept. The subsequent sending and receiving of messages that is introduced by composition yields a — in the case of μ-Charts always finite — chain of reactions. Hence, each step of a reactive system described by Statecharts consists of several, causally dependent micro-steps. To use sequential composition of commands on transitions would mean to further split one micro-step into different, so to speak pico-steps. We doubt that this is reasonable from a methodological point of view.

As a consequence, we have decided to realize a compromise between the two aforementioned approaches in μ-Charts. Instead of the sequential composition $c_1; c_2$ proposed in [Sch96a, Sch96b] we use the parallel composition $c_1 \star c_2$ as in [HN96], but avoid race conditions by requiring that c_1 and c_2 fulfill the so-called *weak Bernstein condition*. This condition requires that no common variable of c_1 and c_2 is allowed to occur on the left-hand-side of an assignment in c_1 and c_2. Take the following composition as example: "$X := 1 \star X := 2$", abbreviated by $c_1 \star c_2$. The unique common location of c_1 and c_2 is X. However, in both c_1 and c_2 we have a write access to X and therefore $c_1 \star c_2$ does not fulfill Bernstein's condition. Such commands are not valid with respect to this condition in our setting. Notice that a procedure that decides whether a command is valid or not, is based on the syntax only and can easily be implemented and therefore is part of the static analysis. Note further that Bernstein's condition implies commutativity, that is, $c_1 \star c_2$ is equivalent to $c_2 \star c_1$. Thus, commutativity is a weaker condition than Bernstein's. Hence, the meaning of commands composed in parallel does not depend on their sequential ordering.

2.3.2 Composition

Parallel composition is used to construct independent, concurrent components. To allow interaction of such components, our language provides a restricted broadcast communication mechanism (multicasting). The broadcasting communication primitive is low-level enough to be implemented efficiently and yet general enough to allow higher-level mechanisms, such as channels, to be defined by the designer [CGH+93a]. In [Har87], for example, this mechanism already is integrated in the parallel composition of Statecharts. Broadcasting is achieved by feeding back all generated signals to all components. In STATEMATE Statecharts, this means that there exists an *implicit feedback* mechanism at the outermost level of a Statechart. Unfortunately, this implicit signal broadcasting leads to a non-compositional semantics and specifications that are hard to grasp. We avoid this problem by introducing communication by *explicit feedback* in the composition operator.

In the literature, different semantic interpretations of the feedback mechanism can be found [vdB94]. For the deterministic version of our language [NRS96, Sch96a, SNR96], we examined several communication operators with different timings. We believe that for non-deterministic, abstract specifications *instantaneous feedback* is the accurate concept (see the advantages discussed in Chapter 1).

Suppose that $S_1, S_2 \in \mathcal{S}$ are arbitrary μ-Charts and L is the set of signals that can be possibly multicast between S_1 and S_2, then their composition, syntactically defined by

$$S_1 \lhd L \rhd S_2$$

is also in $\mathcal{S}$. Graphically, this construction is denoted as signal set between the dashed lines that separate S_1 and S_2 (see Figure 2.6). This set L has to be a subset of $(In(S_1) \cup In(S_2)) \cap (Out(S_1) \cup Out(S_2))$ to get readable specifications. Here, $In(S_n)$ and $Out(S_n)$ denote the input and output interfaces of S_n for $n = 1, 2$, respectively. The interfaces of the composed chart S are defined as follows:

$$\begin{aligned} In(S) &=_{df} (In(S_1)\backslash L) \cup (In(S_2)\backslash L) \\ Out(S) &=_{df} Out(S_1) \cup Out(S_2) \end{aligned}$$

We call input signals in $In(S_n)\backslash L$ *external input signals* or *stimuli* as they are input from the external environment of S_n, and input signals in $In(S_n) \cap L$ are termed *internal input signals* (also see Page 120). The latter are signals that are generated by the system itself and made available by multicasting. Like for signals, a similar distinction for *external* and *internal* input events can be made.

Note that to compose two charts S_1 and S_2 does not mean to mutually feedback the output of S_1 only to the input of S_2 and vice versa, but to feedback both outputs to both inputs. However, this composition operator does not reflect full broadcasting in the sense that the output of a component is fed back to the input of all other components. Rather, to combine the charts S_1 and S_2 by $S_1 \lhd L \rhd S_2$ means that merely all sub-components of

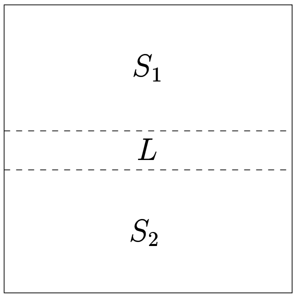


Figure 2.6: Composition — graphical representation

S_1 and S_2 can participate to the broadcasting of all signals that are included in the set L. We automatically get this restricted form of broadcasting, the so-called *multicasting*, using the composition operator. This operator can be used in a nested form. The unique solution to admit full broadcasting between all system components would be to not allow nested usage of the composition operator but rather to use one single feedback concept on the outermost level. However, in this case, we would have to deal with implicit feedback which would destroy compositionality and would yield the well-known problems.

2.3.3 Hiding

Specifying large reactive systems possibly leads to large charts with many signal names. This may promote name clashes which could be avoided by the utilization of local signal hiding, which can be compared with the declaration of local procedure variables in a high-level programming language. Output signals that are multicast using the ternary operator $. \lhd . \rhd .$ are still visible by the environment of S. If signals in K shall be hidden for the part of the specification not belonging to S, we use the hiding operator $[S]_K$, where

$$\begin{aligned} In([S]_K) &=_{df} In(S) \\ Out([S]_K) &=_{df} Out(S) \backslash K \end{aligned}$$

The construct $[S_1 \lhd L \rhd S_2]_L$, for instance, hides all output signals that are fed back. Like for multicasting, there is also a graphical counterpart for hiding; it is a box, attached to the bottom of S, which contains the signals K that are hidden (see Figure 2.7). To see the interplay of all just introduced syntactic concepts, we finally conclude this chapter with another example, a digital stopwatch.

2.3.4 Stopwatch Example

As an example for a μ-Chart with local variables we consider a three-digit stopwatch. Each digit is implemented as a seven-segment display (cf. Figure 2.8). The stopwatch

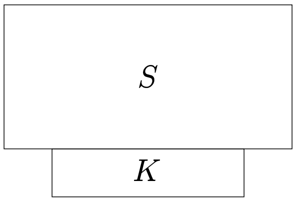


Figure 2.7: Hiding — graphical representation

is part of a benchmark collection for hardware design and verification [Kro94]. The corresponding μ-Chart specification is shown in Figure 2.9.

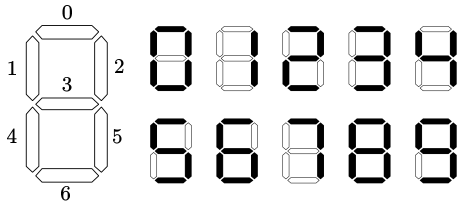


Figure 2.8: Seven segment display

The initial state of the stopwatch is Off. The stopwatch is controlled by two buttons: "start/stop" (s) and "reset" (r). The watch is started by pressing the start/stop button. Here, the specification requires that the reset button is not pressed simultaneously. In its On state, the display then shows the stopped time, where the rightmost display (Lo) shows tenths of seconds and the remaining two (Med and Hi) show seconds and tens of seconds, respectively. Thus, the watch can display times from 00.0 up to 99.9 seconds. Table 2.2 provides an overview over the signals in use. Local variables are denoted by X. To emphasize that all variables are local with respect to one specific sequential automaton, we have not chosen different variable names, but X for all of them, Lo, Med, and Hi. In our graphical syntax, we follow the convention that initial data values are defined by assignments in square brackets like $[X = 0]$, for instance.

Signal	*Meaning*	*Source*
s	Start/Stop	External
r	Reset	
time	Tenths of seconds	Internal
med	Seconds	
low	Tens of seconds	

Table 2.2: Signals used in the stopwatch

The stopwatch is driven by a 1 MHz external clock, hence every 100,000 clock pulses the lowest digit on the display increases by one, if it is less than nine; otherwise, it is

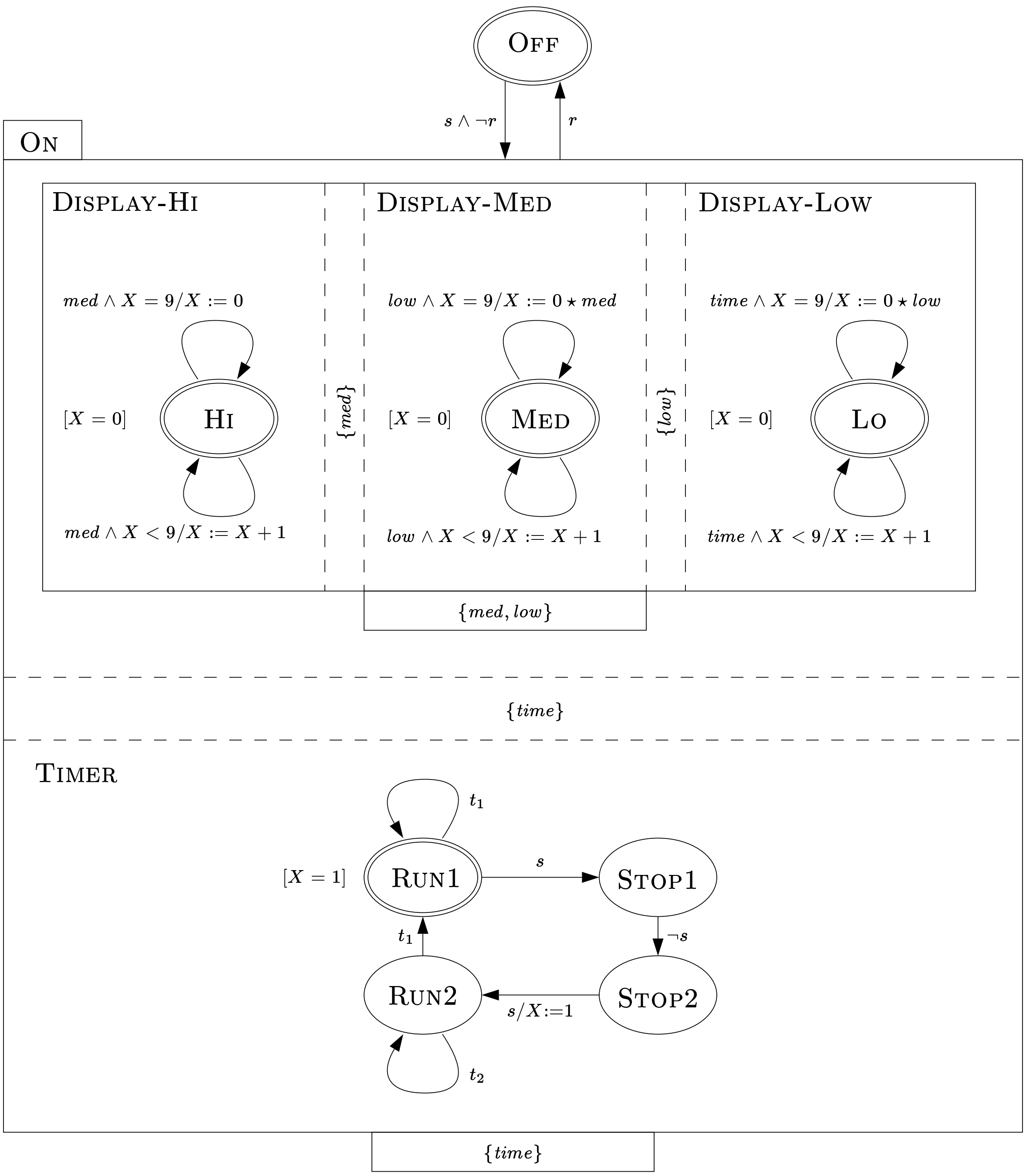


Figure 2.9: Stopwatch

reset to zero, and the medium digit increases by one. When the reset button is pressed, the watch returns to its OFF state, and the display shows 00.0. If, however, instead of the reset button, the start/stop button is pressed again, the watch stops counting until this button is pressed again.

The stopwatch in Figure 2.9 is modeled with five sequential automata: $S_{Stopwatch}$, S_{Timer}, $S_{Display-Hi}$, $S_{Display-Med}$, and $S_{Display-Low}$. The watch contains two basic control states: ON and OFF. The former is decomposed into two parallel components, namely DISPLAY and TIMER: $[[S_{Display}]_{\{med,low\}} \lhd \{time\} \rhd S_{Timer}]_{\{time\}}$. There are two boxes to express hiding: one around $S_{Display}$, and one around the decomposed ON state. The transitions with label t_1 are in each case representative for two transitions with labels $\neg s \wedge X < 10^5/X := X+1$ and $\neg s \wedge X \geq 10^5/X := 0 \star time$, respectively. Similarly, the transition with label t_2 actually stands for two transitions with labels $s \wedge X < 10^5/X := X+1$ and $s \wedge X \geq 10^5/X := 0 \star time$, respectively.

Analyzing Figure 2.9 in more detail, it becomes clear that a kind of parameterization for sequential automata would be helpful to obtain more compact specifications. It is straightforward to extend μ-Charts by a technique that provides this feature. However, this is beyond the scope of this thesis and therefore has been neglected here.

The external clock signal that is needed to generate the 1 MHz clock does not explicitly appear in the specification. Nevertheless, we assume that an appropriate clock signal is sent 10^5 times per second.

2.4 Core Semantics

In this section, we present a stream semantics for μ-Charts to denote their complete input/output history, that is, their externally visible behavior. Hence, changes of configurations (= control and data states) are not visible for an external observer. Before we start presenting this semantics, we first illustrate some preliminaries that are necessary to formulate the semantics mathematically.

2.4.1 Preliminaries

Like other Statecharts dialects [Mar92], μ-Charts are a synchronous language based on a discrete, clock synchronous time model. They follow the principles of the perfect synchrony hypothesis [BG92]. We assume a basic clock, which divides the time into consecutive *instants*. The transition between two consecutive instants is termed a *step*.

A *system reaction* of a μ-Chart consists of a sequence of *steps*. At each step, the system receives a set of signals from the environment (see also Section 1.1). Upon reception of this input set, the system produces a set of output signals, modifies local variables, and changes its control state. Since we deal with instantaneous feedback, the output signals are assumed to be generated in the same instant as the input signals are received. A signal is said to be *present* in a given instant, if it is either input from the environment or generated by the system. Otherwise, it is said to be *absent*. The entire set of signals that are present in one instant is termed *event* (see also Page 32).

To collect signals to sets that are generated by the system itself means that we abstract from the concrete time points within the instant when the signals become present. This is a feasible solution as we assume that these, possibly complex, internal reactions cannot be perceived by the environment. Though these relations are not visible in the semantics they play a crucial role when mathematically describing the interaction of components. As a consequence, we are interested in the causality relations between the signals within one instant. These relations describe which signal presence depends on which other signal presence. Causality cycles therefore have to be avoided by an explicit analysis of the specification (see Section 5.2.3). In the remainder of this thesis, we will discuss this topic more thoroughly.

Reactive systems interact continuously with the environment. Hence, their complete input/output behavior can be described using communication histories. We model the communication history of μ-Charts by pairs of input/output streams, where each stream element consists of a set of signals. One such pair of two infinitely long streams is then called a system reaction. Mathematically, we describe the behavior of μ-Charts by relations over streams. Thus, we briefly have to discuss the notion of streams in the following. For a detailed description we refer, for example, to [Bro93] and [Ste97].

2.4.2 Streams

In this section, we briefly discuss the notion of streams which we will use throughout this thesis. First of all, we have to mention that there is no unique notion of streams in the literature. For a good overview see [Ste97].

Logicians often use the notion of "streams over set M" as a synonym for "ω-words over M", which are just infinite countable ascending enumerations of elements in M. Since such an enumeration is always order isomorphic to the natural numbers $\mathbb{N}$, the notion "ω-words" is used, according to the ordinal ω. For some authors [BDD$^+$93] "streams over set M" are the union of all ω-words over M with the collection of all "finite sequences over M". In this thesis, we will follow this convention and use M^∞ to characterize all "infinite sequences over M". We take the notion "reactive" literally and assume that reactive systems never halt, but continue to react forever. Therefore, we are merely interested in infinite sequences and disregard finite ones. In practice, this is no restriction at all since we can model systems that do not provide any information after a finite number of, say, n steps by streams that consist of the empty signal set beginning with the $(n+1)$st step.

Our notation for the concatenation operator is &. Given an element m of type M and a stream s over M, the term $m\&s$ denotes the stream which starts with the element m

followed by the stream s.

In order to formulate operations over streams we need discriminators and destructors for the type of streams. The destructor *ft* selects the first element of a stream. When applied to $m\&s$ it yields m. The destructor *rt* selects the rest of a stream. When applied to $m\&s$, it yields s.

The constructor & and the destructors *ft* and *rt* together with a fixed point operator are basically all the operations we need to formulate other operations on streams including stream processing functions. However, throughout this thesis we sometimes use equational systems together with local definitions ("let" terms) to formulate recursive operations on streams (in Section 5.1, for instance). Clearly, every such equational system can be re-cast into a pure definition using just λ-abstraction and the fixed point operator [Win93]. As an example for an auxiliary operation on streams consider the operation $\downarrow$ where $s \downarrow n$ yields the n-th element of stream s.

By the aid of this brief technical background the reader should be able to understand the stream semantics for μ-Charts presented in the following section. The semantics presented there is based on relations over streams. However, this overview does by no means give a precise or even complete introduction to streams and related notions. To provide more mathematically profound explanations, a huge technical machinery would be necessary.

For instance, to be more precise, we would have to discuss what implications we get when introducing streams over the cpo (complete partial order) M as the initial solution of the domain equation $X \simeq M \otimes (X_\perp)$ where $\otimes$ denotes the strict product of complete partial orders with bottom $\perp$. This approach stems from domain theory and exploits the techniques from [Pau87, Fre90, Gun92]. The technical details as well as the formalization of the theory in the theorem prover Isabelle [Pau94] are extensively discussed in [Reg94]. A short survey can be found in [Reg95]. A certain school in the computer science community namely builds so-called "domain equations" over certain basic domains including M and refers to distinguished solutions of these equations as "the type of streams over M". There are several variants of this domain theory around, heavily depending on the topologies used to solve (up to isomorphisms) the aforementioned equations. Just to note a few, there are approaches which use ideal completion [Möl85], information systems [Sco82, Win93] or different kinds of complete partial orders [Sco76, Sch86, Pau87, SG90, Gun92]. Also, there is a large amount of work in category theory about domain equations [LS86, Fre90, AL91].

Material that is more directed towards practical application of streams together with a brief introduction to the specification of distributed systems using the FOCUS methodology can be found in [SS95]. For a detailed description of FOCUS we refer to [BDD$^+$93, BS97]. A further, easy to read introduction to FOCUS provide [Fuc94, Spi98].

2.4.3 Semantic Definitions

In this section, we formalize the input/output behavior for each μ-Chart construct. This semantics is defined inductively following the syntactic structure of the language as presented in Section 2.3. In the semantics, μ-Charts are synchronized by a global, discrete clock. However, do not forget that distances between consecutive clock ticks need not necessarily be equidistant (see Section 1.2). The parts of the system reaction that are visible for an external observer, that is, the input and output of signals, build the stream elements. The relationship between two consecutive elements is defined by transition relations. Moreover, each transition relation formally denotes the relationship between two system configurations, i.e. the set of all currently valid control states of all sequential automata between two consecutive instants.

For a chart $S \in \mathcal{S}$ we denote its non-deterministic I/O behavior by the relation $[\![S]\!]_{io} \in \wp(\mathcal{I}(S)^\infty \times \mathcal{O}(S)^\infty)$. Remember that elements in $In(S)$, $Out(S)$, $\mathcal{I}(S) =_{df} \wp(In(S))$, and $\mathcal{O}(S) =_{df} \wp(Out(S))$ are called *input signals*, *output signals*, *input events*, and *output events*, respectively. If we do not care about their direction, we only say *signal* or *event*. The interfaces of each syntactic μ-Chart construct are denoted as follows:

$$
\begin{aligned}
In((I, O, \Sigma, \Sigma_0, V_l, \varphi_0, \delta)) &=_{df} I \\
Out((I, O, \Sigma, \Sigma_0, V_l, \varphi_0, \delta)) &=_{df} O \\
In(S_1 \lhd L \rhd S_2) &=_{df} (In(S_1) \cup In(S_2)) \backslash L \\
Out(S_1 \lhd L \rhd S_2) &=_{df} Out(S_1) \cup Out(S_2) \\
In([S]_K) &=_{df} In(S) \\
Out([S]_K) &=_{df} Out(S) \backslash K
\end{aligned}
$$

Recall that for each signal s we say s is *present in event* x if and only if $s \in x$ and *absent* otherwise. The pure I/O semantics is defined using an auxiliary relational semantics $[\![S]\!] \in \wp(\mathcal{C}(S) \times \mathcal{I}(S)^\infty \times \mathcal{O}(S)^\infty)$ that reflects in addition to the input and output streams also the current configuration $\mathcal{C}(S)$ of chart S. The I/O semantics then is defined as follows:

$$
[\![S]\!]_{io} =_{df} \{(i, o) \mid \exists c.c \in Init(S) \wedge (c, i, o) \in [\![S]\!]\}
$$

where

$$
\begin{aligned}
Init((I, O, \Sigma, \Sigma_0, V_l, \varphi_0, \delta)) &=_{df} \{(\sigma_0, \varphi_0(\sigma_0)) \mid \sigma_0 \in \Sigma_0\} \\
Init(S_1 \lhd L \rhd S_2) &=_{df} Init(S_1) \times Init(S_2) \\
Init([S]_K) &=_{df} Init(S)
\end{aligned}
$$

denote the initial configurations. The notion of configuration plays a central role in the auxiliary relation. The configuration of a sequential automaton with respect to the current instant is a tuple that consists of the automaton's current control state and the current valuation of all its local variables. The application of the composition operator combines the configurations of its components by the algebraic product. Signal hiding does not affect the configuration at all. More formally, a *configuration* of chart S is an element in $\mathcal{C}(S)$, which is inductively defined by:

$$\begin{aligned}\mathcal{C}((I,O,\Sigma,\Sigma_0,V_l,\varphi_0,\delta)) &=_{df} \Sigma \times (V_l \to \mathbb{Z})\\ \mathcal{C}(S_1 \lhd L \rhd S_2) &=_{df} \mathcal{C}(S_1) \times \mathcal{C}(S_2)\\ \mathcal{C}([S]_K) &=_{df} \mathcal{C}(S)\end{aligned}$$

To ease reading, instead of the explicit tuple we often simply write c to denote an arbitrary configuration whenever it is clear from the context what we mean.

Expressions and Commands

With this background we are able to define the semantic functions for the transitions of an automaton $A =_{df} (I,O,\Sigma,\Sigma_0,V_l,\varphi_0,\delta)$. To define the denotational semantics of automaton transitions we first have to introduce a valuation ε as a total function $\varepsilon : V_l \to \mathbb{Z}$. The set of all valuations of A is denoted by $\mathcal{E}(A)$. Note that $\mathcal{E}(A)$ contains total functions: variables have a defined and unique value at every single instant. We define the semantic functions for arithmetic and Boolean expressions, and commands that perform a reaction:

$$\begin{aligned}\mathcal{A}[\![.]\!] &: \mathit{Aexp} \to (\mathcal{E}(A) \to \mathbb{Z})\\ \mathcal{B}[\![.]\!] &: \mathit{Bexp} \to ((\mathcal{E}(A) \times \mathcal{I}(A)) \to \mathbb{B})\\ \mathcal{R}[\![.]\!] &: \mathit{Com} \to (\mathcal{E}(A) \to (\mathcal{E}(A) \times \mathcal{O}(A)))\end{aligned}$$

where $\mathbb{B} =_{df} \{\mathrm{tt}, \mathrm{ff}\}$. We define the denotation of an arithmetic expression by structural induction following the notation of [Win93]:

$$\begin{aligned}\mathcal{A}[\![n]\!]\varepsilon &=_{df} n^{\mathbb{Z}}\\ \mathcal{A}[\![Y]\!]\varepsilon &=_{df} \varepsilon(Y)\\ \mathcal{A}[\![a_1 \,\mathsf{binop}\, a_2]\!]\varepsilon &=_{df} \mathcal{A}[\![a_1]\!]\varepsilon \;\mathit{binop}\; \mathcal{A}[\![a_2]\!]\varepsilon\end{aligned}$$

where $n^{\mathbb{Z}}$ denotes the value of the (syntactic) number n in the carrier set $\mathbb{Z}$, and *binop* denotes the corresponding operation on $\mathbb{Z}$ to the syntactic sign binop. The denotation of a Boolean expression is also defined by structural induction:

$$\begin{aligned}\mathcal{B}[\![\mathsf{true}]\!](\varepsilon,x) &=_{df} \mathrm{tt}\\ \mathcal{B}[\![\mathsf{false}]\!](\varepsilon,x) &=_{df} \mathrm{ff}\\ \mathcal{B}[\![a_1 = a_2]\!](\varepsilon,x) &=_{df} \mathcal{A}[\![a_1]\!]\varepsilon = \mathcal{A}[\![a_2]\!]\varepsilon\\ \mathcal{B}[\![a_1 \le a_2]\!](\varepsilon,x) &=_{df} \mathcal{A}[\![a_1]\!]\varepsilon \le \mathcal{A}[\![a_2]\!]\varepsilon\\ \mathcal{B}[\![s_i]\!](\varepsilon,x) &=_{df} s_i \in x\\ \mathcal{B}[\![\neg b]\!](\varepsilon,x) &=_{df} \neg(\mathcal{B}[\![b]\!](\varepsilon,x))\\ \mathcal{B}[\![b_1 \wedge b_2]\!](\varepsilon,x) &=_{df} \mathcal{B}[\![b_1]\!](\varepsilon,x) \wedge \mathcal{B}[\![b_2]\!](\varepsilon,x)\end{aligned}$$

The definition of the reaction $\mathcal{R}[\![c]\!]$ of a command c is a bit more subtle than the definitions of $\mathcal{A}[\![.]\!]$ and $\mathcal{B}[\![.]\!]$:

$$\begin{aligned}
\mathcal{R}[\![\mathsf{skip}]\!]\varepsilon &=_{\mathit{df}} (\varepsilon, \emptyset)\\
\mathcal{R}[\![Y := a]\!]\varepsilon &=_{\mathit{df}} \text{let } n = \mathcal{A}[\![a]\!]\varepsilon \text{ in } (\varepsilon[n/Y], \emptyset)\\
\mathcal{R}[\![s_o]\!]\varepsilon &=_{\mathit{df}} (\varepsilon, \{s_o\})\\
\mathcal{R}[\![c_1 \star c_2]\!]\varepsilon &=_{\mathit{df}} \text{let } (\varepsilon', y) = \mathcal{R}[\![c_1]\!]\varepsilon \text{ in } (\text{let } (\varepsilon'', z) = \mathcal{R}[\![c_2]\!]\varepsilon' \text{ in } (\varepsilon'', y \cup z))
\end{aligned}$$

We here write $\varepsilon[n/Y]$ for the function obtained from ε by substituting its value in Y by n. The assignment $Y := a$ to a local variable Y induces an update of the valuation ε, and the statement s_o outputs a pure signal and adds it to the set of signals generated so far in the current instant. In the following, we will see that while the first projection, the valuation of $\mathcal{R}[\![c]\!]$ will be valid in the very next instant, the second projection, the set of current output signals will not. The presence or absence of signals is determined in every single instant anew. Since the semantics of parallel composition of commands does not depend on their ordering, $\mathcal{R}[\![c_1 \star c_2]\!]\varepsilon$ equals $\mathcal{R}[\![c_2 \star c_1]\!]\varepsilon$ for arbitrary valuation ε.

The imperative language fragment we use does not provide any means to express non-determinism. Once having "decided" to take a specific automaton transition there is no possibility to introduce further non-deterministic behavior. Non-deterministic reactions of sequential automata are solely included beforehand, that is, when a specific transition is chosen.

Sequential Automata

Before we now will define the auxiliary semantics $[\![A]\!]$ of the sequential automaton $A =_{\mathit{df}} (I, O, \Sigma, \Sigma_0, V_l, \varphi_0, \delta)$, we first have to introduce some further definitions. For each $\sigma \in \Sigma$, $\delta(\sigma)$ denotes the — possibly empty — set of all outgoing transitions from control state σ. Each such transition is defined by a triple, consisting of trigger condition, command, and successor state. Since both trigger condition and command are syntactic terms, we first have to define the semantics

$$[\![\delta]\!] \in \mathcal{C}(A) \times \mathit{In}(A) \to \wp(\mathcal{C}(A) \times \mathit{Out}(A))$$

of δ in order to determine $[\![A]\!]$. The set $[\![\delta]\!]((\sigma, \varepsilon), x)$ denotes A's reaction upon receiving the input event x in configuration (σ, ε). This reaction yields, due to non-determinism, all possible subsequent configurations (σ', ε') together with the output event y. Those are collected in one set. Here, $[\![\delta]\!]$ is defined from the transition relation δ as follows, where the brackets $[\![.]\!]$ are overloaded to ease reading. For all $(\sigma, \varepsilon) \in \mathcal{C}(A)$ and $x \in \mathcal{I}(A)$:

$$\begin{aligned}
[\![\delta]\!]((\sigma, \varepsilon), x) =_{\mathit{df}} \; & \{((\sigma', \varepsilon'), y) \in \mathcal{C}(A) \times \mathcal{O}(A) \mid \\
& \exists t, \mathit{com}.(t, \mathit{com}, \sigma') \in \delta(\sigma) \wedge (\varepsilon, x) \models t \wedge (\varepsilon', y) = \mathcal{R}[\![\mathit{com}]\!]\varepsilon\}
\end{aligned}$$

Here, $(\varepsilon, x) \models t$ is true if and only if the trigger t can be evaluated to true with respect to the valuation ε and the current event x. For instance, the following holds: $(Y \mapsto$

$8, \{a, b\}) \models (5 \leq Y) \wedge a$. More precisely, for every sequential automaton A, input event $x \in \mathcal{I}(A)$, valuation $\varepsilon \in V_l \to \mathbb{Z}$, and trigger condition $t \in Bexp(I + V_l)$ we define the short cut:

$$(\varepsilon, x) \models t \Longleftrightarrow_{df} \mathcal{B}[\![t]\!](\varepsilon, x) = \mathtt{tt}$$

To join all such pairs in one set, we write:

$$[\![t]\!]_A =_{df} \{(\varepsilon, x) \in \mathcal{E}(A) \times \mathcal{I}(A) \mid (\varepsilon, x) \models t\}$$

Finally, the tuple $\mathcal{R}[\![com]\!]\varepsilon$ consists of the next valuation ε' and the output event y that are obtained when the command com is executed with respect to the current valuation ε.

The auxiliary semantics $[\![A]\!] \in \wp(\mathcal{C}(A) \times \mathcal{I}(A)^\infty \times \mathcal{O}(A)^\infty)$ of a sequential automaton A is now defined as the greatest solution with respect to the subset ordering of the following recursive equation:

$$[\![A]\!] = \{(c, x\&i, y\&o) \mid \exists c'.((c', y) \in [\![\delta]\!](c, x) \wedge (c', i, o) \in [\![A]\!]) \vee [\![\delta]\!](c, x) = \emptyset\} \tag{2.1}$$

where $c \in \mathcal{C}(A)$, $x \in \mathcal{I}(A)$, $i \in \mathcal{I}(A)^\infty$, $y \in \mathcal{O}(A)$, and $o \in \mathcal{O}(A)^\infty$. Informally, $[\![A]\!]$ is the set of all those tuples $(c, x\&i, y\&o)$ such that one of the following two cases is true: either A generates the output event y and changes its current configuration from c to c' while reacting on input event x and then behaves similarly in the new configuration c' processing the rest i of the input event stream or the reaction is (yet) unspecified. In this case, the predicate $[\![\delta]\!](c, x) = \emptyset$ is a characterization for the chaotic behavior of A, as the choice of i, y, and o is not restricted at all. In Chapter 3, we will present a set of syntactic rules that allow to resolve under-specifications and to get more defined, that is, more concrete specifications. "Chaotic" here means that whenever for the automaton A in the current configuration c a transition relation for the current input event x is not defined, it can produce an arbitrary output sequence $y\&o$ and can change to an arbitrary successor configuration c'. Later on, in the design process this under-specification can be reduced; mathematically, this means to transform chaotic behavior into well-defined behavior. Finding the greatest solution for Equation 2.1 is equivalent to finding the greatest solution for the alternative equation

$$F(X_0) = X_0$$

i.e. the greatest fixed point $\text{gfp}(F)$ of F with respect to the subset ordering, where F is defined by the lambda abstraction

$$F =_{df} \lambda X.\{(c, x\&i, y\&o) \mid \exists c'.((c', y) \in [\![\delta]\!](c, x) \wedge (c', i, o) \in X) \vee [\![\delta]\!](c, x) = \emptyset\}$$

Notice that $\emptyset$ is always the least fixed point, if $[\![\delta]\!](c, x) \neq \emptyset$ for all configurations c and input events x. Least fixed points yield finite objects, whereas greatest fixed points are related to infinite solutions. As we deal with infinite I/O histories, we therefore look for the greatest fixed point $\text{gfp}(F)$ which is characterized by

$$\bigcup\{X \in \wp(\mathcal{C}(A) \times \mathcal{I}(A)^\infty \times \mathcal{O}(A)^\infty) \mid X \subseteq F(X)\}$$

Because of the dualism between sets and predicates, the set-theoretic greatest fixed point corresponds with the least fixed point with regard to the implication of predicates. Apart from the greatest fixed point further fixed points of F exist. As already mentioned, the empty set is in most cases the least fixed point. In our approach, non-determinism of sequential automata is characterized by an arbitrary choice of all enabled transitions. We do not require any fairness properties. However, if we included fairness in our automata model, we would have to choose a fixed point between the least and greatest one in order to describe the automaton's behavior with the formal semantics. Fairness, however, leads to problems when developing a syntactic refinement calculus. Hence, we have decided to take the greatest fixed point for the construction of the semantics. This coincides with the idea of defining a loose semantics for automata or μ-Charts in general. As a consequence, we exclude exactly those input/output streams in our semantics that are not part of an automaton's behavior, but allow all remaining.

Due to the theorem of Knaster and Tarski [Win93], the monotonicity of F is a sufficient condition for the existence of the greatest fixed point. Indeed, as F is a monotonic function on a complete lattice, namely the power domain, a greatest fixed point always exists. The monotonicity follows straightforwardly from the fact that X only occurs positively in the definition of F. Now, the interesting question arises whether F is in addition continuous so that Kleene's iteration scheme can be used to compute $\text{gfp}(F)$. If we restrict ourselves to finite sets of control states as well as finite data types, the continuity of F is ensured. With the limitation to finite sets of control- and data-states, unbounded internal (that is, not visible for an external observer in the communication history) non-determinism [Fra86] cannot occur. The following theorem summarizes our informal explanation.

Theorem 2.4.1 (Continuity of F)
Let A be the sequential automaton $(I, O, \Sigma, \Sigma_0, V_l, \varphi_0, \delta)$ and F the lambda abstraction of its auxiliary semantics $[\![A]\!]$ as defined above. Furthermore, let $(X_n)_{n\in\mathbb{N}}$ be a descending chain of sets, i.e. $X_n \subseteq X_{n-1}$ for all $n \in \mathbb{N}$. If the number of configurations of A is finite, i.e. $|\mathcal{C}(A)| < \infty$, then F is a continuous function, i.e. $\bigcap_{n\in\mathbb{N}} F(X_n) = F\left(\bigcap_{n\in\mathbb{N}} X_n\right)$. $\Box$

Proof 1 We first prove the $\supseteq$ direction. As for all $n \in \mathbb{N}$ the set inclusion $\bigcap_{m\in\mathbb{N}} X_m \subseteq X_n$ holds and F is a monotonic function, the proposition follows immediately. It remains to prove the opposite $\subseteq$ direction, i.e. $(c, x\&i, y\&o) \in F(X_n) \Rightarrow (c, x\&i, y\&o) \in F(\bigcap_{m\in\mathbb{N}} X_m)$ for all natural numbers n. This is equivalent to showing the following implication:

$$\begin{aligned}&\forall n.(n \in \mathbb{N} \Rightarrow \exists c'.((c', y) \in [\![\delta]\!](c, x) \wedge (c', i, o) \in X_n) \vee [\![\delta]\!](c, x) = \emptyset) \Rightarrow\\ &\exists c'.((c', y) \in [\![\delta]\!](c, x) \wedge \forall n.(n \in \mathbb{N} \Rightarrow (c', i, o) \in X_n)) \vee [\![\delta]\!](c, x) = \emptyset\end{aligned}$$

To prove this implication we have to swap universal and existential quantifiers. First, we notice that neither the variable c' nor n occurs free in $[\![\delta]\!](c, x) = \emptyset$. Therefore, to prove the above implication is equivalent to showing:

$$\begin{aligned}&\forall n.(n \in \mathbb{N} \Rightarrow \exists c'.((c', y) \in [\![\delta]\!](c, x) \wedge (c', i, o) \in X_n \vee [\![\delta]\!](c, x) = \emptyset)) \Rightarrow\\ &\exists c'.((c', y) \in [\![\delta]\!](c, x) \wedge \forall n.(n \in \mathbb{N} \Rightarrow (c', i, o) \in X_n \vee [\![\delta]\!](c, x) = \emptyset))\end{aligned}$$

Notice that in general existential and universal quantifiers cannot be swapped [EFT92]. It remains to be verified whether this is possible in the case under consideration. Assume that the theorem's premise is true. Then we have infinitely many natural numbers, but only finitely many configurations in $\mathcal{C}(A)$. Therefore, there exists at least one $c' \in \mathcal{C}(A)$ such that for infinitely many $n \in \mathbb{N}$ the following is true:

$$(c', y) \in [\![\delta]\!](c, x) \wedge (c', i, o) \in X_n$$

Since $(X_n)_{n\in\mathbb{N}}$ is a descending chain, this is not only true for infinitely many, but for all $n \in \mathbb{N}$ and the proof is completed. □

Since we restrict ourselves to finite configurations in order to get specifications that can be formally verified using model checking techniques (see Chapter 4), we conclude that F is continuous in all practically relevant cases.

Composition

When two charts S_1 and S_2 are composed to $S_1 \lhd L \rhd S_2$, the outputs of both components, say o_1 and o_2, respectively, are collected to, say o, and instantaneously fed back. Hence, the input of S_n for $n = 1, 2$ now does not only consist of the external input stream i that comes from the environment with respect to $S_1 \lhd L \rhd S_2$, but also includes the output stream o. Formally, the semantics of the composition with instantaneous feedback is defined as follows. Recall that $In(S) = (In(S_1) \cup In(S_2))\backslash L$. This input interface and the following formula are reflected in Figure 2.10: apart from the external input signals in $In(S_n)\backslash L$ also internal input signals in $In(S_n) \cap L$ are available for chart S_n in each instant. Following the perfect synchrony hypothesis, we assume that the internal input is produced in the same instant by either S_1 or S_2. Hence, we can conclude that a signal is present in an instant if it is either input from the environment or generated by the system itself. However, for internal input merely signals in $(Out(S_1) \cap L) \cup (Out(S_2) \cap L)$ come into question. Informally, these signals are available on a virtual bus (the box between S_1 and S_2 in Figure 2.10) for exactly one instant. This strategy reflects multicast communication between S_1 and S_2. More precisely, the semantics of composition is defined as follows:

$$\begin{aligned}[\![S_1 \lhd L \rhd S_2]\!] =_{df} \{((c_1, c_2), i, o) \in \mathcal{C}(S) \times \mathcal{I}(S)^\infty \times \mathcal{O}(S)^\infty \mid \\ \exists o_1, o_2 . o_1 \in \mathcal{O}(S_1)^\infty \wedge o_2 \in \mathcal{O}(S_2)^\infty \wedge o = o_1 \uplus o_2 \wedge \\ (c_1, (i \uplus o|_L)|_{In(S_1)}, o_1) \in [\![S_1]\!] \wedge \\ (c_2, (i \uplus o|_L)|_{In(S_2)}, o_2) \in [\![S_2]\!]\}\end{aligned}$$

where $\uplus$ here is the pointwise extension of the set-theoretic union on streams of sets and $s|_X$ the pointwise restriction of stream elements (events) in s to signals in X.

Our notion of instantaneous feedback, that is, feedback in the same instant, was inspired by Argos. It resembles Argos' technique for solving equations [Mar92], but adds non-determinism and chaotic behavior in an elegant way. The pure parallel composition

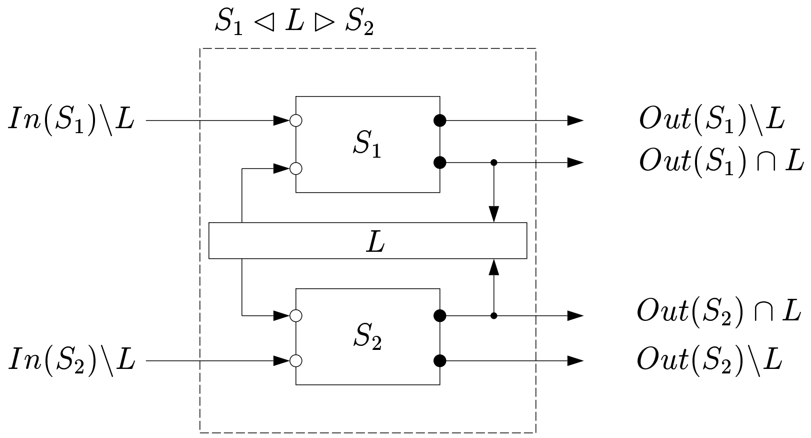


Figure 2.10: Composition — feedback

$S_1 \| S_2$ of two components S_1 and S_2 is defined as special case; $S_1 \| S_2$ is regarded to be a syntactic abbreviation for $S_1 \lhd \emptyset \rhd S_2$:

$$S_1 \| S_2 =_{df} S_1 \lhd \emptyset \rhd S_2$$

The semantics of $[\![S_1 \| S_2]\!]$ of the pure parallel composition can be simplified to:

$$\{((c_1, c_2), i, o) \mid \exists o_1, o_2 . o = o_1 \uplus o_2 \wedge (c_1, i|_{In(S_1)}, o_1) \in [\![S_1]\!] \wedge (c_2, i|_{In(S_2)}, o_2) \in [\![S_2]\!]\}$$

Whenever the input and output interfaces of S_1 and S_2 do not harmonize, that is, the signal sets $In(S_n)$, $Out(S_1) \cup Out(S_2)$, and L are disjoint for $n = 1, 2$, then $[\![S_1 \lhd L \rhd S_2]\!]$ equals $[\![S_1 \| S_2]\!]$. The proof follows directly from the definition of composition.

In synchronous languages, composition is very complex. Therefore, it is worthwhile discussing the composition operator of μ-Charts in detail. First of all, we would like to mention that, according to the results of [HG91], there exists no semantics for reactive systems that is I/O synchronous, modular, and causal at the same time. However, it is possible to union arbitrary two of these three properties in one semantics. This is a very strong result which we had to keep in mind when defining the formal semantics of composition. Thus, as it is impossible to define a I/O synchronous, modular, and causal semantics, we have decided to define a semantics that is synchronous and modular. For instance, also Esterel follows this approach; its semantics is I/O synchronous and modular, but not causal [HG91]. However, in general, a non-causal reactive system cannot be implemented, neither centralized nor distributed. Therefore, in Esterel, causality has to be achieved by an additional, subtle causality analysis or, in other words, a semantic analysis. In order to get implementable specifications, we also provide a semantic analysis for μ-Charts. In Section 5.2.3, we will present this analysis, which is based on so-called signal graphs, and will recognize that it is equivalent to the causality analysis of Esterel.

Since the semantics presented here is not causal, it does not only include causal fixed points, but all existing fixed points. Note that even if we had restricted ourselves to a semantics that merely deals with least fixed points, this semantics still would not be causal as it cannot be guaranteed that the restriction to least fixed points leads to causal and therefore realizable implementations (for instance, see the pathological example with labels a/b and b/a in Sections 2.4.4 and 5.1). As a consequence, we have decided to define a semantics that captures all fixed points. This approach has several advantages. First, a formal semantics that includes all fixed points is mathematically easier than one that is restricted to least fixed points. Reasoning about the semantics itself is therefore easier, too. In particular, this alleviates the soundness proofs of the refinement rules in Chapter 3. Second, this decision has been determined by methodological reasons. There are specifications that may have more than one causal fixed point due to under-specification. Since all of these fixed points are causal, it is not advisable to restrict oneself to the least one; otherwise, we would exclude certain refinement alternatives a priori. Third, there may exist both causal and non-causal fixed points, but nevertheless the specification can be refined to a specification that merely has one single, causal fixed point. In Chapter 3, we will give an example for this.

Signal Hiding

To hide all output signals from K in chart S, we use the local hiding mechanism $[S]_K$. The formal definition is straightforward:

$$[\![[S]_K]\!] =_{df} \{(c, i, o|_{Out(S)\setminus K}) \mid (c, i, o) \in [\![S]\!]\}$$

2.4.4 Semantic Properties

Once we have defined the formal semantics for μ-Charts we can now examine some interesting semantic properties of our language. Properties that come into question are, for instance, commutativity and associativity. A software engineer who uses μ-Charts to specify reactive systems has to be aware of these properties. Otherwise, it might be the case that the specified system is not the intended system. For instance, there exist binary operators, like addition, which are associative and others, like subtraction, which are not associative. It is important to know for a designer that composition of μ-Charts is in general not associative.

The commutativity of the composition operator is also an important feature, since we do not only give a textual, but also a graphical syntax for μ-Charts. In a two-dimensional visual formalism, it should not matter whether the charts S_1 and S_2 are sticked together this or that way in order to graphically express the composition $S_1 \lhd L \rhd S_2$. If an operator is commutative it should be represented graphically by means of symmetric notation, and vice versa. The next theorem guarantees the commutativity of the composition operator.

Theorem 2.4.2 (Commutativity)
Let S_1 and S_2 be two arbitrary μ-Charts and L a set of signals. Then the composition of S_1 and S_2 is commutative, i.e. $[\![S_1 \lhd L \rhd S_2]\!] = [\![S_2 \lhd L \rhd S_1]\!]$. □

This theorem follows immediately from the fact that the binary operators $\Cup$ and $\wedge$ are commutative. One also is tempted to assume that composition is in addition associative. However, in general this is not the case. In particular, the following is not true in general:

$$[\![S_1 \lhd L \rhd (S_2 \lhd L' \rhd S_3)]\!]_{io} = [\![(S_1 \lhd L \rhd S_2) \lhd L' \rhd S_3]\!]_{io}$$

Counterexamples can easily be constructed if we assume, for example, that a part of the input interface of S_1 can be connected either to a part of the output interface of S_2 or to a part of the output interface of S_3. This problem occurs, for instance, if $In(S_1) \cap L \cap Out(S_2) \cap Out(S_3)$ is a non-empty set of signals. In such cases, S_1 potentially can receive input of both S_2 and S_3. However, the construct $S_1 \lhd L \rhd (S_2 \lhd L' \rhd S_3)$ connects both the output of S_2 and the output of S_3 to the input of S_1, whereas $(S_1 \lhd L \rhd S_2) \lhd L' \rhd S_3$ only establishes a connection between S_1 and S_2.

With a number of further context conditions we can, however, establish a property that somewhat resembles associativity. These conditions restrict the input interfaces of the sub-charts S_1, S_2, and S_3. In particular, we require that S_1 is not able to receive any signals that are included in L' or in $L \cap Out(S_3)$. The former is needed to ensure that S_1 cannot react to any input messages in L' if it is embedded in the rest of the specification by $(S_1 \lhd L \rhd S_2) \lhd L' \rhd S_3$. This is because it also cannot react to these signals if it is embedded by $S_1 \lhd L \rhd (S_2 \lhd L' \rhd S_3)$. Furthermore, in $(S_1 \lhd L \rhd S_2) \lhd L' \rhd S_3$, the chart S_1 cannot react to any input signals in $L \cap Out(S_3)$ whereas in $S_1 \lhd L \rhd (S_2 \lhd L' \rhd S_3)$ it potentially can. Hence, this possible difference also has to be excluded by context conditions. We require similar conditions for S_2 and S_3.

Theorem 2.4.3 (Pseudo Associativity)
Let S_1, S_2, and S_3 be three arbitrary μ-Charts and L and L' two sets of signals. Then the equation $[\![(S_1 \lhd L \rhd S_2) \lhd L' \rhd S_3]\!] = [\![S_1 \lhd L \rhd (S_2 \lhd L' \rhd S_3)]\!]$ holds, if the input interfaces are restricted as follows:

1. Input interface of S_1:
   (a) $In(S_1) \cap L' = \emptyset$
   (b) $In(S_1) \cap L \cap Out(S_3) = \emptyset$
2. Input interface of S_2:
   (a) $In(S_2) \cap L \cap L' = \emptyset$
   (b) $In(S_2) \cap L' \cap Out(S_1) = \emptyset$
   (c) $In(S_2) \cap L \cap Out(S_3) = \emptyset$
3. Input interface of S_3:
   (a) $In(S_3) \cap L = \emptyset$
   (b) $In(S_3) \cap L' \cap Out(S_1) = \emptyset$ □

Proof 2 On the one hand, the semantics $[\![(S_1 \lhd L \rhd S_2) \lhd L' \rhd S_3]\!]$ is denoted as follows:

$$\begin{aligned}\{((c_1,c_2,c_3),i,o) \quad | \quad & \exists o_1,o_2,o_3.o_n \in \mathcal{O}(S_n)^\infty, n=1,2,3 \wedge o = o_1 \uplus o_2 \uplus o_3 \\ \wedge \quad & (c_1,((i \uplus o|_{L'})|_{(In(S_1)\cup In(S_2))\setminus L} \uplus o_{12}|_L)|_{In(S_1)}, o_1) \in [\![S_1]\!] \\ \wedge \quad & (c_2,((i \uplus o|_{L'})|_{(In(S_1)\cup In(S_2))\setminus L} \uplus o_{12}|_L)|_{In(S_2)}, o_2) \in [\![S_2]\!] \\ \wedge \quad & (c_3,(i \uplus o|_{L'})|_{In(S_3)}, o_3) \in [\![S_3]\!]\}\end{aligned}$$

where $o_{12} =_{df} o_1 \uplus o_2$. On the other hand, the semantics $[\![S_1 \lhd L \rhd (S_2 \lhd L' \rhd S_3)]\!]$ is denoted as follows:

$$\begin{aligned}\{((c_1,c_2,c_3),i,o) \quad | \quad & \exists o_1,o_2,o_3.o_n \in \mathcal{O}(S_n)^\infty, n=1,2,3 \wedge o = o_1 \uplus o_2 \uplus o_3 \\ \wedge \quad & (c_1,(i \uplus o|_L)|_{In(S_1)}, o_1) \in [\![S_1]\!] \\ \wedge \quad & (c_2,((i \uplus o|_L)|_{(In(S_2)\cup In(S_3))\setminus L'} \uplus o_{23}|_{L'})|_{In(S_2)}, o_2) \in [\![S_2]\!] \\ \wedge \quad & (c_3,((i \uplus o|_L)|_{(In(S_2)\cup In(S_3))\setminus L'} \uplus o_{23}|_{L'})|_{In(S_3)}, o_3) \in [\![S_3]\!]\}\end{aligned}$$

where $o_{23} =_{df} o_2 \uplus o_3$. Both sets are obviously equal if the following three conditions hold:

1. $o|_{L\cap In(S_1)} = o|_{L'\cap In(S_1)\setminus L} \uplus o_{12}|_{L\cap In(S_1)}$
2. $o|_{L\cap In(S_2)\setminus L'} \uplus o_{23}|_{L'\cap In(S_2)} = o|_{L'\cap In(S_2)\setminus L} \uplus o_{12}|_{L\cap In(S_2)}$
3. $o|_{L\cap In(S_3)\setminus L'} \uplus o_{23}|_{L'\cap In(S_3)} = o|_{L'\cap In(S_3)}$

In the remainder of this proof, we shall verify that the three above requirements are indeed fulfilled by the theorem's restrictions. Condition (1) holds since $o|_{L\cap In(S_1)}$ equals $o_{12}|_{L\cap In(S_1)}$ due to Requirement (1.b) of the theorem and $o|_{L'\cap In(S_1)\setminus L} = \emptyset$ due to Requirement (1.a).

To prove Condition (2) is slightly more costly. Due to Requirement (2.a), it is sufficient to show:

$$o|_{L\cap In(S_2)} \uplus o_{23}|_{L'\cap In(S_2)} = o|_{L'\cap In(S_2)} \uplus o_{12}|_{L\cap In(S_2)}$$

Because of Requirements (2.b) and (2.c), $o|_{L'\cap In(S_2)}$ and $o|_{L\cap In(S_2)}$ are equivalent to $o_{23}|_{L'\cap In(S_2)}$ and $o_{12}|_{L\cap In(S_2)}$, respectively and left-hand-side and right-hand-side of the above equation coincide.

Condition (3) holds since $o|_{L\cap In(S_3)\setminus L'} = \emptyset$ due to Requirement (3.a) and $o|_{L'\cap In(S_3)}$ equals $o_{23}|_{L'\cap In(S_3)}$ due to (3.b). This concludes the proof. □

As a direct consequence of this theorem, in particular the equation $[\![(S_1 \lhd L \rhd S_2) \lhd L \rhd S_3]\!] = [\![S_1 \lhd L \rhd (S_2 \lhd L \rhd S_3)]\!]$ holds if for all input interfaces the following holds: $In(S_n) \cap L = \emptyset$ for $n = 1,2,3$. However, this is a fairly restrictive condition as it requires that S_1, S_2, and S_3 cannot receive any signal at all from each other. Thus, in this special case, associativity is restricted to charts that are composed in parallel, but do not interact at all; in particular, $[\![(S_1\|S_2)\|S_3]\!] = [\![S_1\|(S_2\|S_3)]\!]$. Though Theorem 2.4.3 gives hints when associativity for the more general composition can be established we recommend that the designer should not apply this theorem because its application might be error-prone as long as he or she does not have any computer assistance.

Altogether, we realize that a bracket-like concept is also needed for our graphical syntax. We achieve this by drawing a box for each pair of parentheses. Figure 2.3 gives an example. Here, the corresponding textual syntax would read $[S_{Control} \lhd L \rhd (S_{MotorLeft}\|S_{MotorRight})]_K$, where L and K are as indicated in the picture. Would the reader now be capable to judge whether $[(S_{Control} \lhd L \rhd S_{MotorLeft})\|S_{MotorRight}]_K$ has an equivalent behavior? Unfortunately, we have to give a negative answer. Both motors send their *ready* signals to the control. While the first specification can react on the *ready* signal of both, the second cannot if sent by the right motor.

Idempotence and distributivity are other properties that, due to non-determinism, do not hold for μ-Charts; instead we have in general:

- $[\![S\|S]\!]_{io} \neq [\![S]\!]_{io}$
- $[\![(S_1\|S_2) \lhd L \rhd S_3]\!]_{io} \neq [\![(S_1 \lhd L \rhd S_3)\|(S_2 \lhd L \rhd S_3)]\!]_{io}$
- $[\![(S_1 \lhd L \rhd S_2)\|S_3]\!]_{io} \neq [\![(S_1\|S_3) \lhd L \rhd (S_2\|S_3)]\!]_{io}$

Redundant specifications in general increase the non-deterministic behavior of the system or, strictly speaking, $[\![S]\!]_{io} \subseteq [\![S\|S]\!]_{io}$. The opposite direction $[\![S\|S]\!]_{io} \subseteq [\![S]\!]_{io}$ generally does not hold for non-deterministic specifications, but $[\![S]\!]_{io} = [\![S\|S]\!]_{io}$ is true if S is deterministic. However, notice that in general even for a deterministic chart S in general the equality $[\![S]\!]_{io} = [\![S \lhd L \rhd S]\!]_{io}$ is not true. Furthermore, it is straightforward to show that in general, $[\![[S_1 \lhd L \rhd S_2]_K]\!] \neq [\![[S_1]_K \lhd L \rhd [S_2]_K]\!]$; only if K and L are disjoint follows $[\![[S_1 \lhd L \rhd S_2]_K]\!] = [\![[S_1]_K \lhd L \rhd [S_2]_K]\!]$. However, the following equivalences are true: $[\![[[S]_K]_L]\!] = [\![[S]_{K\cup L}]\!] = [\![[[S]_L]_K]\!]$.

The following theorem is essential for distributed implementation (see Sections 5.3 and 5.4). Let us assume that the specification that shall be distributed is $(S_1 \lhd L \rhd S_2) \lhd L' \rhd S_3$. Recalling the definition of composition (see also Figure 2.10) we remember that our notion of communication is multicasting instead of broadcasting. Output signals in L of S_1 and S_2, respectively are merely locally fed back. Intuitively, this feedback is performed by a virtual bus that connects S_1 and S_2. This bus does not transport any other signals than those in L. The composition $S_1 \lhd L \rhd S_2$ is connected with S_3 by a different virtual bus, which is capable of performing communications of signals in L'.

The notion of a virtual bus is introduced by the composition operator that enables explicit multicasting between composed charts. The advantage of this concept is first, that we get a compositional semantics and second, that we can localize communications. The latter, however, also has a practical draw-back. If we implement a μ-Charts specification, not all internal signals can be transmitted by one and the same physical bus. Rather, if we did not establish any further algebraic rules, for each virtual bus also a real, physical bus would have to be implemented.

Recall that our composition operator is not associative in general. Hence, without any further transformation, all feasible implementations would require that S_1 and S_2 are either implemented on one single processor or on different ones but communicate via a specific bus that is not used by component S_3. The reader may agree that this procedure is too restrictive. What we rather need is an equivalent version of $(S_1 \lhd L \rhd S_2) \lhd L' \rhd S_3$ so that S_1, S_2, and S_3 can communicate via a single bus without modification of the original behavior.

We can remedy this restriction and achieve the desired communication scheme by syntactic transformations. Under certain assumptions, the above specification is equivalent to $(S_1 \| S_2) \lhd L \cup L' \rhd S_3$. This syntax now enables a further distributed implementation alternative, where S_1, S_2, and S_3 can communicate using one and the same bus. Which restrictions have to be fulfilled in order to guarantee sharing of one bus is discussed in the next theorem. In Section 5.4, we will discuss yet another reason why this theorem plays an essential role for distributed execution of μ-Chart specifications. For the moment, we have to postpone this discussion because it requires further technical framework which we have not presented so far.

Theorem 2.4.4 (Signal Extrusion)
Let S_1, S_2, and S_3 be three arbitrary μ-Charts and L and L' two signal sets. Then for $l \notin L \cup L'$ and $l \notin In(S_3) \cup Out(S_3)$ the following equation holds: $[\![(S_1 \lhd L \cup \{l\} \rhd S_2) \lhd L' \rhd S_3]\!] = [\![(S_1 \lhd L \rhd S_2) \lhd L' \cup \{l\} \rhd S_3]\!]$. □

Proof 3 On the one hand, the semantics $[\![(S_1 \lhd L \cup \{l\} \rhd S_2) \lhd L' \rhd S_3]\!]$ is denoted as follows:

$$\begin{array}{lll} \{((c_1, c_2, c_3), i, o) & | & \exists o_1, o_2, o_3.o_n \in \mathcal{O}(S_n)^\infty, n = 1,2,3 \wedge o = o_1 \uplus o_2 \uplus o_3 \\ & \wedge & (c_1, ((i \uplus o|_{L'})|_{(In(S_1) \cup In(S_2)) \setminus (L \cup \{l\})} \uplus o_{12}|_{L \cup \{l\}})|_{In(S_1)}, o_1) \in [\![S_1]\!] \\ & \wedge & (c_2, ((i \uplus o|_{L'})|_{(In(S_1) \cup In(S_2)) \setminus (L \cup \{l\})} \uplus o_{12}|_{L \cup \{l\}})|_{In(S_2)}, o_2) \in [\![S_2]\!] \\ & \wedge & (c_3, (i \uplus o|_{L'})|_{In(S_3)}, o_3) \in [\![S_3]\!]\} \end{array}$$

where $o_{12} =_{df} o_1 \uplus o_2$. On the other hand, the semantics $[\![(S_1 \lhd L \rhd S_2) \lhd L' \cup \{l\} \rhd S_3]\!]$ is denoted as follows:

$$\begin{array}{lll} \{((c_1, c_2, c_3), i, o) & | & \exists o_1, o_2, o_3.o_n \in \mathcal{O}(S_n)^\infty, n = 1,2,3 \wedge o = o_1 \uplus o_2 \uplus o_3 \\ & \wedge & (c_1, ((i \uplus o|_{L' \cup \{l\}})|_{(In(S_1) \cup In(S_2)) \setminus L} \uplus o_{12}|_{L})|_{In(S_1)}, o_1) \in [\![S_1]\!] \\ & \wedge & (c_2, ((i \uplus o|_{L' \cup \{l\}})|_{(In(S_1) \cup In(S_2)) \setminus L} \uplus o_{12}|_{L})|_{In(S_2)}, o_2) \in [\![S_2]\!] \\ & \wedge & (c_3, (i \uplus o|_{L' \cup \{l\}})|_{In(S_3)}, o_3) \in [\![S_3]\!]\} \end{array}$$

Both sets are equal if the following three conditions hold:

1. $o|_{L' \cap (In(S_1) \setminus (L \cup \{l\}))} \uplus o_{12}|_{(L \cup \{l\}) \cap In(S_1)} = o|_{(L' \cup \{l\}) \cap (In(S_1) \setminus L)} \uplus o_{12}|_{L \cap In(S_1)}$

2. $o|_{L'\cap(In(S_2)\setminus(L\cup\{l\}))} \uplus o_{12}|_{(L\cup\{l\})\cap In(S_2)} = o|_{(L'\cup\{l\})\cap(In(S_2)\setminus L)} \uplus o_{12}|_{L\cap In(S_2)}$

3. $o|_{L'\cap In(S_3)} = o|_{(L'\cup\{l\})\cap In(S_3)}$

First, we prove Condition (1):

$$\begin{aligned}
o|_{L'\cap(In(S_1)\setminus(L\cup\{l\}))} \uplus o_{12}|_{(L\cup\{l\})\cap In(S_1)} &= o|_{L'\cap(In(S_1)\setminus(L\cup\{l\}))} \uplus o_{12}|_{\{l\}\cap In(S_1)} \uplus o_{12}|_{L\cap In(S_1)} \\
&= o|_{L'\cap(In(S_1)\setminus(L\cup\{l\}))} \uplus o|_{\{l\}\cap In(S_1)} \uplus o_{12}|_{L\cap In(S_1)} \\
&= o|_{L'\cap(In(S_1)\setminus(L\cup\{l\}))} \uplus o|_{\{l\}\cap In(S_1)\setminus L} \uplus o_{12}|_{L\cap In(S_1)} \\
&= o|_{L'\cap In(S_1)\setminus L} \uplus o|_{\{l\}\cap In(S_1)\setminus L} \uplus o_{12}|_{L\cap In(S_1)} \\
&= o|_{(L'\cup\{l\})\cap(In(S_1)\setminus L)} \uplus o_{12}|_{L\cap In(S_1)}
\end{aligned}$$

since $l \notin Out(S_3)$ and $l \notin L$ and $l \notin L'$, respectively. The proof of Condition (2) is up to renaming (substitute $In(S_1)$ by $In(S_2)$) equivalent to the proof of (1). Finally, Condition (3) holds since $l \notin In(S_3)$ by assumption. □

This theorem immediately implies that for $L \cap In(S_3) = \emptyset = L \cap Out(S_3)$ the following holds:

$$[\![(S_1 \lhd L \rhd S_2) \lhd L' \rhd S_3]\!] = [\![(S_1 \| S_2) \lhd L \cup L' \rhd S_3]\!]$$

which finally yields the desired result. This form opens a wide spectrum of possible implementations or, in other words, distribution alternatives. With Theorem 2.4.4 available, a systems engineer now might be tempted to develop all designs by simply combining all charts in parallel without any communication and then use one single composition operator at the outermost level to establish multicasting. However, this possibly leads to inscrutable specifications. Compared with a sequential programming language, this specification style would be similar to merely use global variables and to avoid any encapsulation principles. Thus, the designer should continue applying the principle of locality and leave it to the compiler to transform local signals to global ones by renaming. Let $\oslash$ denote an arbitrary automaton $(\emptyset, \emptyset, \Sigma, \Sigma_0, V_l, \varphi_0, \delta)$ with empty input and output interfaces. Then it is always possible to transform a μ-Chart specification consisting of n sequential automata $S_1, \ldots, S_n$ to an isomorphic one that has the syntactic form

$$(S_1 \| \cdots \| S_n) \lhd L \rhd \oslash \tag{2.2}$$

by renaming conflicting signals. This specification then has up to renaming an equivalent behavior as the original one. We call this form *(μ-Charts) normal form*. Since $\oslash$ does not influence the rest of the specification, we abbreviate the normal form by

$$\triangle_L(S_1 \| \cdots \| S_n)$$

However, note that, should the occasion arise, an hiding operator has to be added. In order to implement a μ-Chart specification S on a distributed hardware, we assume that S is in normal form (see Sections 5.3 and 5.4).

Having discussed the essential properties of the μ-Charts semantics that mainly are related to the specific composition operator we use, we will outline some semantic problems that are caused by this operator. Fortunately, these problems arise only very rarely in the case of so-called pathological specifications, which are the subject of the following section.

Pathological Cases

Instantaneous feedback can lead to well-known causality conflicts [Ber96, Ber98]. In the sequel, we will discuss some of the pathological examples from the literature [Mar92] in the context of μ-Charts. As basis for these examples serves the (generic) chart S_n pictured in Figure 2.11. We distinguish the following three cases for the labels t_1 and t_2:

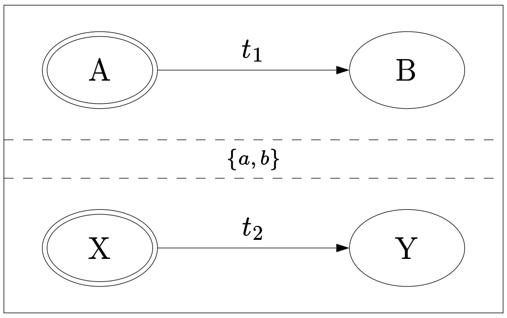


Figure 2.11: Problems with fixed points

1. $t_1 =_{df} a/b$ and $t_2 =_{df} b/a$
2. $t_1 =_{df} \neg a/b$ and $t_2 =_{df} \neg b/a$
3. $t_1 =_{df} a/b$ and $t_2 =_{df} \neg b/a$

Since all other possible input events do not lead to causality conflicts, we merely concentrate on the input event $\emptyset$ and assume that the current configuration of chart S_n for $n = 1, 2, 3$ is the initial configuration (A,X).

In the first case, the semantics $[\![S_1]\!]_{io}$ contains pairs of input/output event streams that start with $\emptyset/\emptyset$ or $\emptyset/\{a, b\}$. This means informally that either S_1 remains in its current configuration (A,X) and the output event is the empty signal set or both transitions fire and the next configuration of S_1 is (B,Y).

We realize that our semantics still leaves room for non-deterministic choices and later refinement steps (see Section 3). At this design stage, we do not bother whether the system should be capable to act without causal reason. When it comes to implementation in Section 5.1, we define a fixed point semantics for deterministic charts that only allows system reactions with a proper signal causality flow.

Notice, however, that this kind of non-determinism does not stem from non-deterministic transitions within one single automaton, but from composition. Though each automaton of S_1 taken for its own is deterministic, the composition can lead to non-determinism. This kind of non-determinism has to be distinguished from that one that stems from intended under-specification. Non-determinism that is additionally introduced by composition may be unintended by the designer.

Synchronous programming languages like ESTEREL or Argos avoid this kind of non-determinism right from the beginning. Since they are pure programming languages that directly serve as basis for later implementation, this is the only way to avoid in-determinable system behavior. The μ-Charts language, however, was developed as a specification language, where systems also can be described on an abstract level. As a consequence, abstract specifications still can contain non-deterministic behavior which is refined later on. In Chapter 3, we will present syntactic rules that allow to refine this example with two fixed points to a specification with only one fixed point, namely the empty signal set.

Also in the second case, the semantics $[\![S_2]\!]_{io}$ is non-deterministic through composition. Both event pairs $\emptyset/\{a\}$ and $\emptyset/\{b\}$ are valid prefixes of pairs of input/output streams in $[\![S_2]\!]_{io}$. Including both behaviors in the semantics informally means that either the upper or the lower automaton can change its current control state. The other remains in its current state. This chart can be used to describe reading data from a scarce resource like a bus, where only one component can access to the resource at one instant of time. In this example, this means that only one of the two automata can access to the resource, but at this design stage we are still not capable of saying which one.

For the last case, we get $[\![S_3]\!]_{io} = \emptyset$, i.e. this chart does not have a well-defined stream semantics as we cannot find any pair (i, o) of input/output streams such that the premise

$$\begin{aligned}\exists o_1, o_2 . o_1 \in \mathcal{O}(S_{31})^\infty \wedge o_2 \in \mathcal{O}(S_{32})^\infty \wedge o = o_1 \uplus o_2 \wedge \\ ((i \uplus o|_L)|_{In(S_{31})}, o_1) \in [\![S_{31}]\!]_{io} \wedge \\ ((i \uplus o|_L)|_{In(S_{32})}, o_2) \in [\![S_{32}]\!]_{io}\end{aligned}$$

is fulfilled, where S_3 is defined as $S_{31} \lhd \{a, b\} \rhd S_{32}$. As a consequence, this chart does not have a defined behavior. We will continue the discussion of pathological examples in Section 5.1, where also a classification will be given.

2.5 Syntactic Extensions

In the last sections, we have defined the core syntax of μ-Charts together with its precise mathematical semantics. So far, many syntactic constructs that can, for instance, be expressed in STATEMATE, are not valid in μ-Charts because they are not included in the latter's core syntax.

While the developers of STATEMATE started with a graphical syntax that included many features and later on tried to find an appropriate semantics, we decided to do it just the other way around. We start with a lean core language, define a formal semantics for it, and then enrich this core language by further syntactic constructs. All constructs not contained in the core language and presented in this thesis are abbreviations of already existing constructs or, in other words, syntactic sugar. In Section 2.4.3, we already have seen that pure parallel composition is considered a special case of composition including communication. This construct and a lot of others do not require new semantic definitions. Furthermore, we are able to extend our μ-Charts language by the following constructs by means of syntactic abbreviations:

- Hierarchical decomposition
- Signals that are generated when entering or exiting a state
- Signals that are generated if a state is part of the current configuration
- Inter-level transitions
- Timeouts

Partly, the definition of these syntactic concepts as abbreviations might yield large specifications. Though these transformed specifications do not have to be visible to the user, they form the basis for code generation, simulation, and verification. For these tasks, it may be more efficient to introduce these constructs not as syntactic abbreviations of the core language, but in a semantic way. For hierarchical decomposition, this has been done for instance in [PS97a]. However, notice that this strategy is more error-prone than ours. Hence, it must be guaranteed that the "efficient" semantics and the derived semantics that results from our transformations are identical. To define an extra semantics for each syntactic construct is not within the scope of this thesis.

2.5.1 Hierarchical Decomposition

To express hierarchical decomposition, which plays a key role in the concept of Statecharts, we do not need a principally new syntactic construct, but derive hierarchy from the composition operator. This facilitates the definition of both formal semantics and refinement calculus. Furthermore, to express hierarchy by parallel composition and multicasting is a convenient way to provide a basis to implement charts of different levels of hierarchy on different processors. With Statecharts semantics that treat hierarchy on the semantic level this could not be achieved that elegantly. In the following, we show how to define hierarchical decomposition on top of composition.

The syntactic notation for *hierarchical decomposition* is $A \triangleright (\Sigma_d, \varrho)$, where A is defined by $(I, O, \Sigma, \Sigma_0, V_l, \varphi_0, \delta)$. Here, the total function $\varrho : \Sigma_d \to \mathcal{S}$ defines for each state in $\Sigma_d \subseteq \Sigma$ the sub-chart by which it is decomposed. Hierarchy can be considered as an abbreviation mechanism. We can translate hierarchical specifications into flat ones applying the following algorithm (hereby we call the sequential automaton A the *controller* or *master* and all sub-charts $\varrho(\sigma)$ with $\sigma \in \Sigma_d$ the *controllees* or *slaves*). The algorithm works differently for weak and strong preemption [Ber93] (we will give an introduction to preemption in the subsequent section). It has to be recursively applied for nested hierarchy.

1. *Composition:* controller A and all controlled charts in $\bigcup_{\sigma \in \Sigma_d} \varrho(\sigma)$ are composed in parallel. Since composition is commutative, controller and controllees can be composed in arbitrary order.

2. *Modification of the master:* the new signals $go(\sigma)$ for all decomposed states $\sigma \in \Sigma_d$ are added to the output interface of A. In the case of weak preemption, for every $\sigma \in \Sigma_d$, the command *com* of every outgoing edge is replaced by $com \star go(\sigma)$. Here, $go(\sigma)$ is a signal which indicates that the slave $\varrho(\sigma)$ attached to σ is currently active and is allowed to fire its transitions. This modification is omitted for other edges than self-loops when strong preemption is desired. Let t_σ be the disjunction of all trigger conditions on all outgoing edges (inclusive self-loops, if they exist) of σ. Then, for every kind of preemption, every state σ is additionally enriched by a self-loop with trigger condition $\neg t_\sigma$ and command $go(\sigma)$.

3. *Modification of the slave(s):* for every $\sigma \in \Sigma_d$ and every sequential automaton in $\varrho(\sigma)$ to every input interface the signal $go(\sigma)$ has to be added. Furthermore, every trigger condition t on every edge has to be substituted by $t \wedge go(\sigma)$ in order to guarantee that the slaves now composed in parallel only react if and only if they are allowed to. If σ is a non-history decomposed state, additional transitions from every state but the default state of every sequential automaton in $\varrho(\sigma)$ with label $\neg go(\sigma)/\varphi(\sigma)$ have to be introduced. This is necessary to initialize the slave whenever the master is left. Otherwise, all slaves would remain in their current states when the master changes its current state; this, however, is only wanted for history decomposed states.

4. *Communication:* in order to enable the communication between master and slave, all above introduced *go* signals have to be fed back and hidden with respect to the parallel composition of the master and all slaves. Feedback applies to all signals in $In(A) \cap Out(\varrho(\sigma))$, too, to enable message passing from slave to master. In particular, this is useful to model self-termination.

Figure 2.12 shows an example. It is the CONTROL part of the locking system. Once having entered the CRASH mode, the system resides in this state forever. Thus, it makes no difference whether NORMAL is history or non-history decomposed in the example. Figure 2.12 shows weak preemption; if we omitted the statement go(NORMAL) on the transition between NORMAL and CRASH, we would model strong preemption.

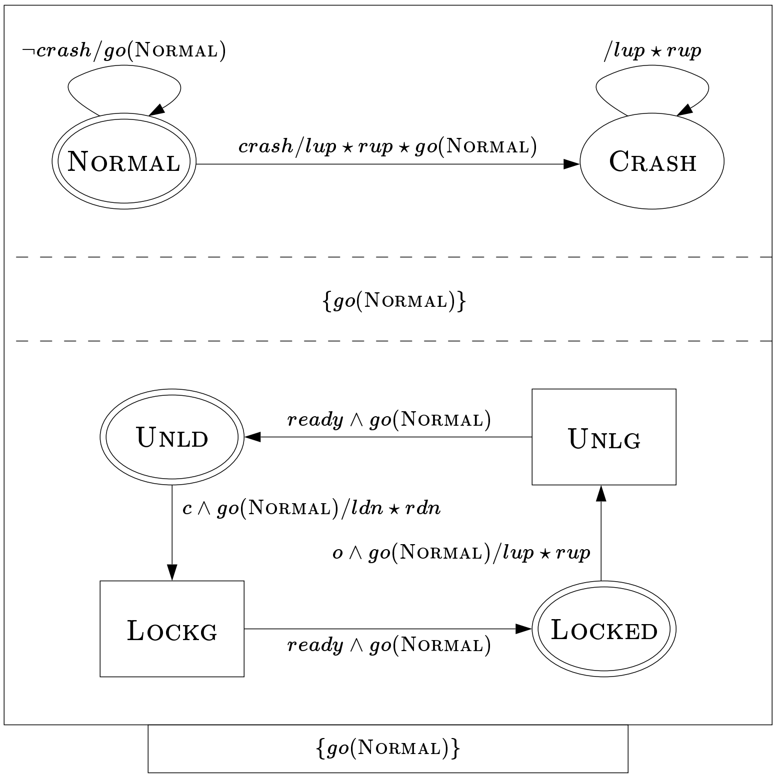


Figure 2.12: Unfolded hierarchy

Interestingly, this algorithm directly yields hierarchical decomposition including the *history* functionality since whenever a slave is left, the current configuration is kept. As indicated above, in order to get decomposition without history, additional transitions from all states of the slaves — but the initial states — to the initial states are needed to initialize both control and data states. In μ-Charts, *deep* as well as *normal history* [Har87] can be modeled easily. Since the slaves are triggered explicitly by *go* signals, it is, in contrast to most related approaches, even possible to define all possible intermediate forms of history.

2.5.2 Interrupts and Priorities

In the last section, we have seen that the control flow of a slave chart depends on the control flow of its master chart. Whenever the master leaves its current control state σ, it also withdraws the control flow from its sub-chart $\varrho(\sigma)$. We say that $\varrho(\sigma)$ has been interrupted. We already broached the subject of interrupts in the last section and will discuss this notion in more detail in the following. We will outline the difference between preemptive and non-preemptive interrupts in our language.

Interrupts

In Section 2.5.1, we have introduced hierarchical decomposition by means of parallel composition and communication. This enables us to model both *preemptive* and *non-preemptive interrupts* (synonymously: *strong* and *weak preemption*) in an elegant way. In the following, we will see that both concepts can be defined similarly.

Weak preemption means that the sub-chart still has the chance to fire a transition before becoming inactive whenever the corresponding master withdraws the control. Though non-preemptive interrupt is suited for modeling preemptive interrupt the inversion is not true. Thus, we choose weak preemption as default mechanism because it is the more general concept [Ber93]. As we will see in the subsequent sections, this kind of interrupt also provides other interesting advantages, like the emulation of outgoing inter-level transitions (see Section 2.5.4).

Before discussing further advantages of non-preemptive interrupts, we give some explanations on semantic differences of both kinds of interrupt. We do this by means of the chart pictured in Figure 2.13. Suppose that the system is in its initial configuration (AA,XX) and that the current input event is $\{a\}$. The two interrupt concepts yield a different behavior. First of all, we realize that both transitions in Figure 2.13 are enabled.

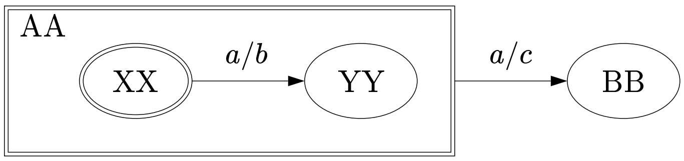


Figure 2.13: Different interrupt types

In the case of non-preemptive interrupt, in Figure 2.13 both transitions are taken. Thus, also both output signals b and c are generated and the next system configuration is (BB,YY). If AA was a history decomposed state, the sub-chart would be re-entered in state YY next time. In contrast to its non-preemptive counterpart, the only signal that is generated by a preemptive interrupt is the signal c as the controller immediately withdraws control from the controllee.

Figure 2.14 shows the μ-Chart of Figure 2.13 after unfolding the hierarchy. Here, the unfolded chart directly reflects non-preemptive interrupt since the $go(AA)$ signal is even then output by the master when state AA is left. Modifying the transition label from $a/c \star go(AA)$ to a/c yields preemptive interrupt, because in this case, the signal $go(AA)$ is not sent from master to slave when the master fires the transition from AA to BB.

It becomes clear that instantaneous feedback and non-preemptive interrupts harmonize. With a delayed feedback we neither would be able to define hierarchy by composition in a proper way nor could we integrate both interrupt concepts smoothly. Moreover, self-termination can easily be expressed by non-preemptive interrupt. Figure 2.15 elucidates this. We here speak of "self-termination" since the controllee itself is responsible for

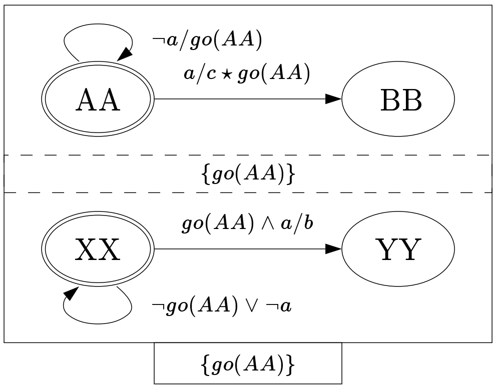


Figure 2.14: Unfolding of Figure 2.13

sending the signal (b in the example) that influences the controller to withdraw the control from the first.

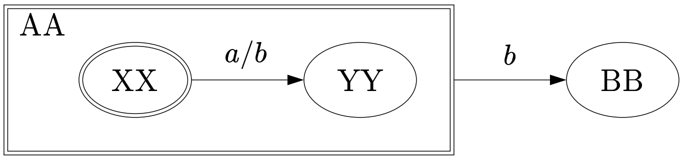


Figure 2.15: Self-termination

The advantage of μ-Charts is that they enable the specification of both preemptive and non-preemptive interrupts. Related approaches differ in this point. On the one hand, STATEMATE Statecharts give priority to the transition of the controller, i.e. preemptive interrupt is implemented. On the other hand, Rhapsody [i-L98], a tool for object-oriented software design, realizes weak-interrupt instead. Hence, in this case the inner transition has a higher priority. Conflicting transitions do not only occur when charts are hierarchically decomposed. Rather, also different outgoing transitions from one control state can be in conflict because the system is yet under-specified. Chapter 3 shows how this under-specification can be solved in principle.

Priorities

We now introduce a priority ordering for all signals in M that are used in a system specification with μ-Charts. Formally, this means to define the injective function

$$prio : M \to \mathbb{N}$$

which assigns to every signal its priority. The function $prio$ induces a total ordering on M. We assume that the priority of a signal a is the higher, the larger the value $prio(a)$ is. Signal priorities can be used to model preemptive interrupts. However, they also serve as solution for a more general problem: they can be used to refine non-deterministic specifications to more deterministic ones (see Section 3). The chart in Figure 2.16 has a non-deterministic behavior in state XX if one of the events $\{a, b\}$ or $\{b, c\}$ occurs, and the subsequent control state either could be YY or ZZ.

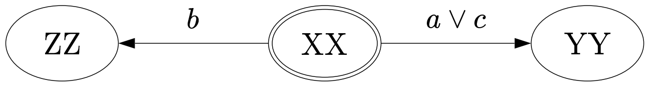


Figure 2.16: Non-determinism through conflicting triggers

To solve this conflict, we can determine the signal priority $prio(a) > prio(b) > prio(c)$. Neither the event $\{a, b\}$ nor $\{b, c\}$ would now lead to conflicts since the signal priorities also imply a priority scheme for transitions. A transition's priority can be defined by the maximum of all signal priorities that occur on the transition label. In the example of Figure 2.16, the transition from XX to YY has therefore the priority $prio(a)$ and the transition from XX to ZZ $prio(b)$.

Signal priorities do not require to introduce any new semantic concepts. They can be completely defined by signal negation. To this end, the transition priorities of a conflicting control state have to be determined first. Then, all triggers t of transitions having this state as source are modified to $t \wedge (\bigwedge \neg t')$, where the t' denote all other triggers of conflicting transitions with lower priority. In the example, this results in the labels as given in Figure 2.17. To give the reader a second example, assume that the transitions t_1, t_2, and t_3 have the conditions a, b, and c, respectively. It is easy to see that all of them are mutually in conflict. Again, we arbitrarily assume the priority scheme $prio(a) > prio(b) > prio(c)$. As a consequence, the new transition labels for t_1, t_2, and t_3 are now a, $b \wedge \neg a$, and $c \wedge \neg b \wedge \neg c$, respectively.

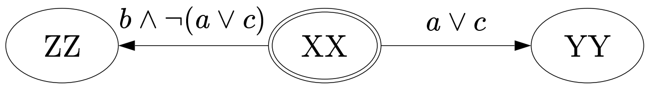


Figure 2.17: Signal priorities resolve conflicts

Introducing a priority ordering for conflicting transitions has an inherent influence on the system behavior. It is an interesting question which particular implications on the overall behavior this modification has. A software engineer might be interested in finding out whether a specification only becomes more concrete by adding priorities. The system should not completely change its behavior. What it formally means to reduce under-specification and when modifications of specifications are feasible will be discussed in Section 3. There, we will find out that to introduce priorities is a correct refinement step (see the rule for modification of trigger conditions).

2.5.3 Entering, Exiting, and Instate Signals

In this section, we present signals that are generated whenever a state of a sequential automaton is entered or exited. The signal that is generated when *entering* or *exiting* the state S is denoted by S↑ and S↓, respectively. Since in so-called "self-loops", i.e. transitions that start and end in state S, a state is both entered and exited, both signals S↑ and S↓ have to be generated. As we will see in the remainder of this section, these self-loop signals play an important role when dealing with hierarchical decomposition. The generation of signals that indicate state switches for simple states is straightforward (see Figure 2.18(a)): each transition that ends in state S, gets S↑ as additional command; each transition that starts in S, is enriched by the command S↓.

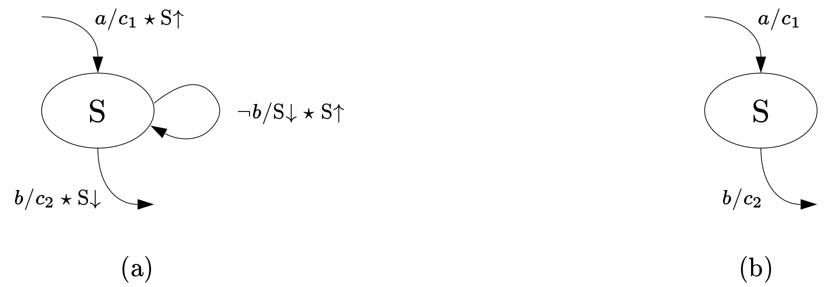


Figure 2.18: State switches

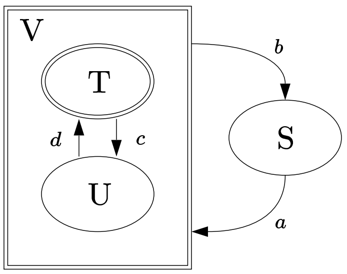
To make specifications more compact, it is allowed to omit both signals in the graphical syntax as shown in Figure 2.18(b). However, if composed charts use these signals as input, explicit multicasting of them by means of the composition operator is still necessary. For hierarchically decomposed states is the situation slightly more complex. Figure 2.19 sketches an example, which is analyzed in the sequel. In a first step, the transition labels of the two sequential automata in Figure 2.19(a) are modified as described above:

a	is modified to	a/S↓ $\star$ V↑
b	is modified to	b/V↓ $\star$ S↑
c	is modified to	c/T↓ $\star$ U↑
d	is modified to	d/U↓ $\star$ T↑

Additionally, we have to introduce new self-loops with the labels $\neg a$/S↓ $\star$ S↑, $\neg b$/V↓ $\star$ V↑, $\neg c$/T↓ $\star$ T↑, and $\neg d$/U↓ $\star$ U↑ for states S, V, T, and U, respectively. The result of these modifications is pictured in Figure 2.19(b). However, so far we have not discussed all modifications that are necessary and that are included in Figure 2.19(b).

Up to now, we only have an incomplete signal scenario. Since V is further decomposed, additional signals have to be generated when entering or exiting V. Whenever V is exited, depending on the current configuration either T↓ or U↓ has to be generated, too. However, these signals cannot be added on the transition labeled with b as we have no static knowledge which state is left in the concrete configuration whenever the signal b appears. Thus, a different solution has to be found. Remember that our default interrupt mechanism is weak preemption. Therefore, the slave automaton in V always has the possibility to perform a last (micro-)step when the control is withdrawn and thus can produce the appropriate exit signal, T↓ or U↓. Since we have added self-loops with entering and exiting signals, the appropriate signal can be generated in any case. However, these signals cannot be output if using strong preemption.

If V is a non-history decomposed chart then, if it is entered, automatically also its default state T is entered. Hence, the reader might assume that we have to add T↑ on the transition labeled with trigger a. However, this is not necessary. To better comprehend how decomposition works, we unfold the hierarchical decomposition of Figure 2.19 as described in the last section and get the chart pictured in Figure 2.20 as result. Whenever the control is withdrawn from the controllee, visible through the absence of signal $go(V)$, it changes its configuration from U to the default state T. As long as the control remains withdrawn, the controllee performs the self-loops of T while generating the output signals T↑ and T↓. Altogether, we see that T↑ need not to be added to the controllers transition as it is generated by the controllee itself.



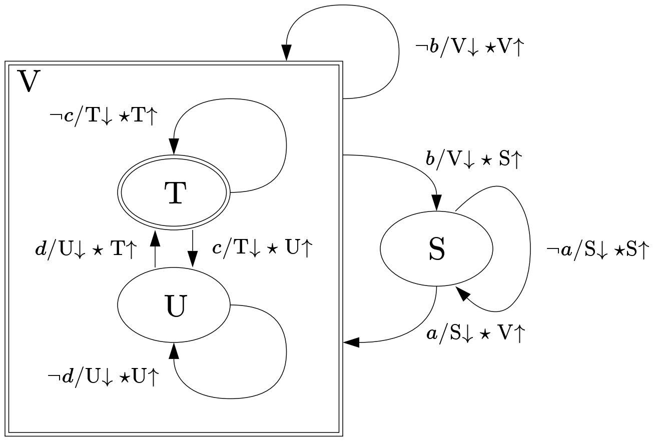


Figure 2.19: State switch signals for hierarchical decomposition

As easy as in the non-history case is the solution if V was history decomposed. To express decomposition with history, the label of the transition from U to T has to be modified to $go(V) \wedge d/\mathrm{U}{\downarrow} \star \mathrm{T}{\uparrow}$ and the one of the transition from U to U itself to $\neg go(V) \vee \neg d/\mathrm{U}{\downarrow} \star \mathrm{U}{\uparrow}$. Informally, this means that the absence of the control signal $go(V)$ now triggers the self-loop of state U instead of the transition from U to the default state T. As a consequence, the automaton remains in its current state instead of making a transition to its default state. This behavior reflects history decomposition and we see that for both kinds of decomposition always the proper entering and exiting signals are generated. We realize that for both history and non-history decomposition entering and exiting signals are easy to introduce.

Additionally to the signals S↑ and S↓ that indicate state switches, we introduce signals

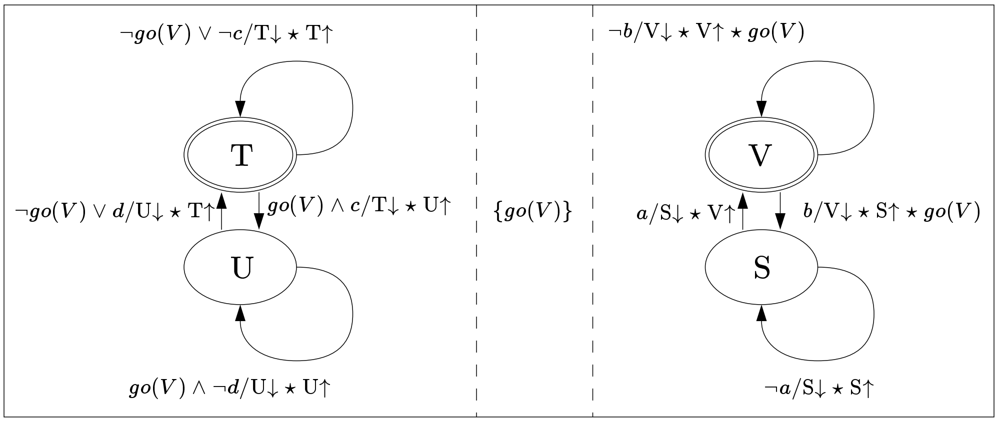


Figure 2.20: Unfolded hierarchical decomposition without history

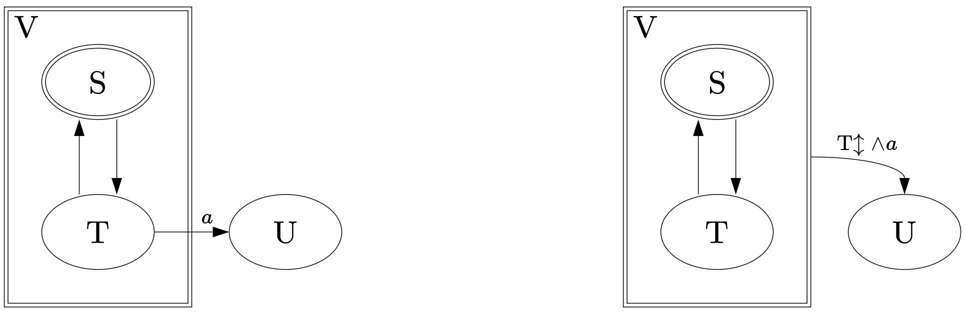
which are generated in every single step for each control state that is included in the current configuration. Whenever the state S is included in the current configuration, this is indicated by the *instate signal* S$\updownarrow$.

Taking Figure 2.18(a) as basis, it is obvious that S$\updownarrow$ has to be added as additional output to self-loops. The overall output then would be "S$\downarrow$ $\star$ S$\uparrow$ $\star$ S$\updownarrow$". The more interesting question now is whether S$\updownarrow$ also has to be added to incoming or outgoing transitions, too. In principle, both solutions are thinkable. Either S$\updownarrow$ can be added when S is entered or when S is exited. However, it is not possible to include S$\updownarrow$ simultaneously in both cases. This would yield an inconsistent signal constellation since in the "border" instants, i.e. whenever the control states of current and next instant are not identical, always two such signals are produced. One for the control state that is currently exited and one for the state that is entered. However, a sequential automaton can only reside in one single control state at a time. Therefore, S$\updownarrow$ can be added either when exiting or entering state S. We decide to add S$\updownarrow$ only when exiting S. This coincides with the idea of weak preemption, and we can always substitute the signal $go(S)$ by S$\updownarrow$.

2.5.4 Inter-level Transitions

The modular structure of μ-Chart specifications requires that transition arrows are always within the scope of one single sequential automaton. Transitions that connect control states of two different, composed automata are not admissible.

Sometimes, a hierarchical decomposition only then shall be exited if the slave automaton is in a certain control state. In Statecharts, this can be specified by non-local or so-called *inter-level transitions* as pictured in Figure 2.21(a). This would not be a proper μ-Chart construct if we merely used our core syntax. However, in μ-Charts there exists a workaround that enables these specifications, too. Figure 2.21(b) demonstrates this; hereby the inter-level transition of Figure 2.21(a) is substituted by the additional condition that when exiting state V the corresponding slave automaton's current state has to be T. This is indicated by the instate signal T$\updownarrow$.



(a) Inter-level (b) Intra-level

Figure 2.21: Inter-level transitions

As we have seen, outgoing inter-level transitions, that is, transitions where the control from the controllee is withdrawn are easy to introduce as syntactic abbreviations on top of the core language. Ingoing inter-level transitions, where the control is handed over from controller to controllee are, however, not possible with our language. Ingoing inter-level transitions would require a typing for interfaces as demonstrated in [GSB98]. This is, however, beyond the scope of this thesis and therefore has been omitted.

2.5.5 Timeouts

As we have discussed in Section 1.1, reactive systems often have to fulfill real-time requirements. In Statecharts, these requirements can be specified by *timeouts*. A timeout is a special signal that occurs a certain, specified amount of time after the initialization of the corresponding "timeout clock". Timeouts are not part of the μ-Charts core syntax.

Figure 2.22(a) shows the extended μ-Charts syntax for the specification of a timeout signal and Figure 2.22(b) the corresponding definition with constructs of the core language including hierarchical decomposition and self-termination. Here, the signal *timeout* that is necessary to withdraw the control from the controllee can be hidden by the hiding operator. The timeout control state is decomposed by a counter. The counter is initialized when the control is transferred from controller to controllee. However, note that if *timeouts* are used, both the specification and implementation of timeouts requires that all steps exactly have the same reaction time.

As we already have discussed, often software engineers want to specify reactions with interrupts. Both concepts interrupts and timeouts can be combined in the following way as outlined in Figure 2.23: the "normal" way to withdraw the control of the controllee

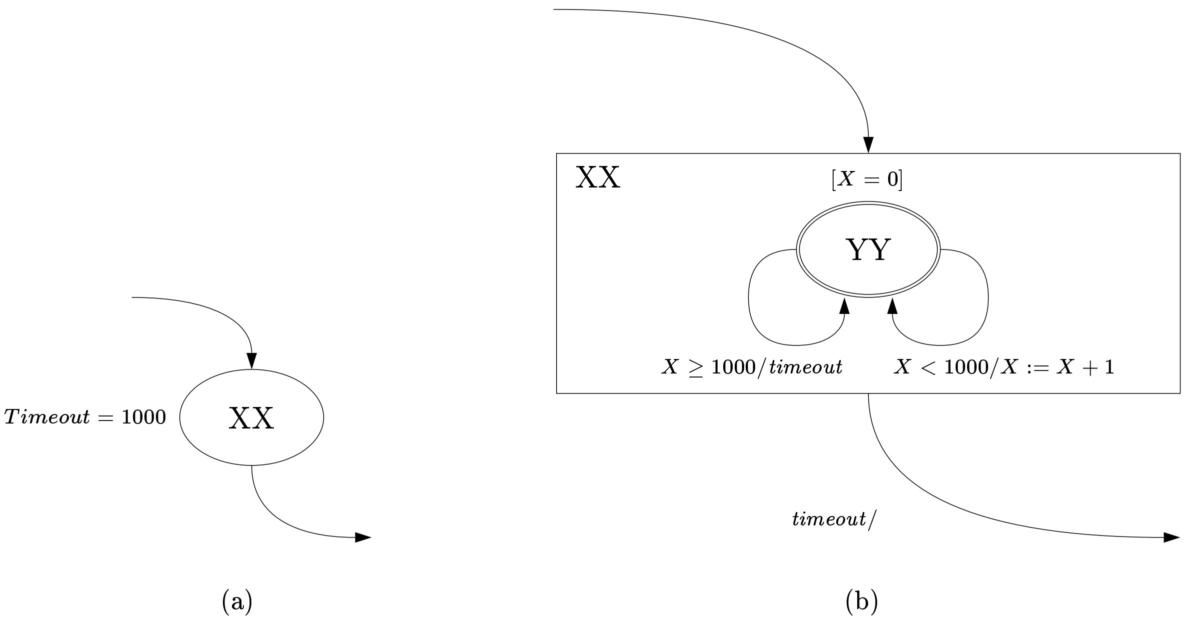


Figure 2.22: Timeouts in μ-Charts

is to leave it when the timeout occurs, i.e. when the signal *timout* is present. However, it is also possible to withdraw the control in an earlier instant if the signal *interrupt* is present.

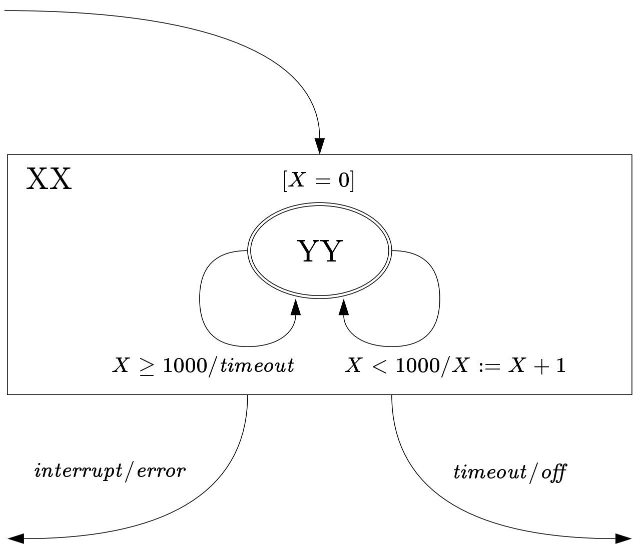


Figure 2.23: Timeout with interrupt

Though the concept for timeouts is suited for the definition of discrete time it suffers from one weakness: the essential assumption is that all steps are separated by constant time intervals. As discussed in Section 2.1, it is in general not a prerequisite for the definition of the system behavior that time is divided into equidistant intervals. However, if the designer wants to specify timeouts, this is no longer the case and the information how long one instant must exactly last has to be annotated to the overall specification as additional requirement. This specification style is similar to an assumption/commitment style [BS97]. In Section 5.2.2, we will see that there may be yet further situations that require this specific specification style in order to get implementable descriptions.

At this point we want to add a warning: whenever a μ-Charts specification is implemented on a distributed architecture the system response times can be slowed down due to communication delays and a global system clock is probably no longer available. As a consequence, those parts of the specification that read and write timeouts better should be implemented on one and the same processor in order to guarantee correct response times.

2.6 Conclusion

In this chapter, we have defined the visual formalism μ-Charts, which is closely related to Harel's Statecharts. In contrast to them, however, the core language of μ-Charts just consist of three syntactic constructs: sequential automata, composition, which includes both parallel execution of charts and multicast communication between them, and hiding of signals. For this description technique, a formal stream semantics has been developed. Based on the core language, syntactic extensions like hierarchical decomposition have been defined that make specification with μ-Charts more comfortable and Statecharts-like. Our language concept has several advantages with respect to both methodological and mathematical aspects in contrast to related approaches. Since we have a very lean core syntax, it is relatively easy to reason about the semantics itself and to prove semantic properties. Moreover, as we will see in the next section, this also eases the definition of a sound refinement calculus for our description technique. Finally, some of the well-known concepts of Statecharts like hierarchical decomposition, for instance, necessarily have to be defined by parallel composition plus communication in order to enable distributed implementations of charts of different levels of hierarchy.

3 Refinement

In the previous section, we have introduced our state-based specification language. We have defined its semantics, i.e. its input/output behavior in terms of streams. However, a specification formalism is of limited value without an appropriate system development process. What we need is to give guidelines how to develop a concrete implementation or realization from an abstract system specification. This transition from an abstract specification to a concrete implementation covers a number of different design steps such as specification, verification, efficient compilation, and refinement.

This section addresses the deductive design of reactive systems using μ-Charts. We show how a specification of a system under development can be transformed into one that is closer to the final system. We present a collection of refinement rules, applicable as syntactic transformations and equipped with syntax-based context conditions. The advantage of a collection of this type is that the designer only has to meet the specific syntactic requirements for a rule when applying it and does not have to verify the rule's soundness. He or she therefore does not need to be aware of the formal semantics of μ-Charts but just has to apply the intuitive syntactic rules in a correct way. Hence, the stepwise refinement within a calculus for μ-Charts is not only a mathematically appealing idea but also a realistic procedure to be applied in an industrial environment. In particular, we will present the following syntactic refinement rules:

- rules for refining sequential automata: removing and adding states as well as manipulating (removing, adding, and modifying) transitions,
- rules for composition, and
- rules for hierarchical decomposition.

Before we can discuss these rules in detail, we have to define our notion of refinement first. Then, we will prove two essential properties for refinement: transitivity and compositionality. The first enables the step-wise application of refinement rules and therefore the incremental system development. The latter secures that a refinement of a part of a system also is a refinement of the overall system. While these properties are discussed in the following section, the calculus itself is presented in Section 3.2.

3.1 Preliminaries

It is usually impossible to carry out the transformation from an abstract specification to one that can be used as basis for implementation in only one step. In practice, the situation is even worse. For complex systems, even a design specialist may not be capable of writing down an abstract specification ad hoc. Rather, such a system will be developed by applying consecutive refinement steps, whereby after every single step the overall system behavior is a bit more concrete. The final implementation then is only the most precise specification that is suitable to run on a certain machine or target architecture. In our setting, we distinguish between the following two basic concepts of refinement:

Behavioral Refinement: A specification S_2 is said to be a behavioral refinement of a specification S_1 if and only if any I/O behavior of S_2 is also an I/O behavior of S_1. As a consequence, S_2 is at most more concrete than S_1, i.e. restricts its I/O behavior, but not a more abstract system view and therefore is not allowed to add additional behavior to the system. In this context, S_1 is termed an *under-specification* of the system as it does not exactly describe one single system behavior but instead a set of behaviors that yet leave design alternatives open. Formally: a specification S_2 is a behavioral refinement of another specification S_1 if and only if the following is true:

$$[\![S_2]\!]_{io} \subseteq [\![S_1]\!]_{io} \wedge In(S_1) = In(S_2) \wedge Out(S_1) = Out(S_2)$$

In this thesis, under-specification is always expressed by non-determinism on the semantic level and behavioral refinement is nothing more than its step-wise reduction.

Interface Refinement: Behavioral refinement only allows the elimination of I/O behaviors but does not support to add I/O behaviors. What we want is not to enlarge the degree of under-specification by such a refinement step but rather to change the interfaces of specifications. In general, interface refinements could allow an arbitrary modification of both input and output interfaces. In our setting, however, only such refinements are necessary that increase the number of input and/or output signals in the interfaces of a μ-Chart. This is because when combining two charts either by composition or through hierarchy the so refined specification at most has more but never less signals in the input and output interfaces. Furthermore, to allow both types of modification, that is, increasing and decreasing the interfaces at the same time, our notion of refinement would not be transitive. Thus, we could not refine μ-Chart specifications incrementally. Moreover, we do not aim at modifications of single interface types like, for example, the refinement of bytes to bits.

Here, a specification S_2 is called an interface refinement of a specification S_1 if and only if S_2 at most adds new input or output interfaces with respect to S_1 and any I/O behavior of S_2 with respect to the I/O interface of S_1 is also an I/O behavior of S_1; a specification S_2 is an interface refinement of another specification S_1 if and only if the following is true:

$$\{(i|_{In(S_1)}, o|_{Out(S_1)}) \mid (i,o) \in \llbracket S_2 \rrbracket_{io}\} = \llbracket S_1 \rrbracket_{io} \wedge In(S_1) \subseteq In(S_2) \wedge Out(S_1) \subseteq Out(S_2)$$

We can combine both notions to one single notion of refinement. A specification S_2 is a *refinement* of another specification S_1 ($S_1 \rightsquigarrow S_2$) if and only if $In(S_1) \subseteq In(S_2)$, $Out(S_1) \subseteq Out(S_2)$ and the following is true:

$$\{(i|_{In(S_1)}, o|_{Out(S_1)}) \mid (i,o) \in \llbracket S_2 \rrbracket_{io}\} \subseteq \llbracket S_1 \rrbracket_{io} \tag{3.1}$$

The commutative diagram in Figure 3.1 summarizes the situation. Whenever the Condition 3.1 is true in both directions, i.e. $S_1 \rightsquigarrow S_2$ and $S_2 \rightsquigarrow S_1$ we write $S_1 \leftrightsquigarrow S_2$. The Definition 3.1 of $\rightsquigarrow$ is equivalent to

$$\forall i \in \mathcal{I}(S_2)^\infty, o \in \mathcal{O}(S_2)^\infty : (i,o) \in \llbracket S_2 \rrbracket_{io} \Rightarrow (i|_{In(S_1)}, o|_{Out(S_1)}) \in \llbracket S_1 \rrbracket_{io} \tag{3.2}$$

$$\begin{tikzcd}
S_1 \arrow[r, "\rightsquigarrow"] \arrow[d, "\llbracket \cdot \rrbracket_{io}"'] & S_2 \arrow[d, "\llbracket \cdot \rrbracket_{io}"] \\
\llbracket S_1 \rrbracket_{io} \arrow[r, "\supseteq"] & \llbracket S_2 \rrbracket_{io}
\end{tikzcd}$$

Figure 3.1: Refinement

Care has to be taken that Condition 3.2 is not mixed up with

$$\forall i \in \mathcal{I}(S_1)^\infty, o \in \mathcal{O}(S_2)^\infty : (i,o) \in \llbracket S_2 \rrbracket_{io} \Rightarrow (i, o|_{Out(S_1)}) \in \llbracket S_1 \rrbracket_{io} \tag{3.3}$$

Both formulae differ in the quantification of the input event i: while in (3.2) i is universally quantified over the input interface of S_1, it is only universally quantified over the input interface of S_2 in (3.3). The latter would consider charts as refinements that are intuitively no refinements; it is too a weak predicate ($S_1 \rightsquigarrow S_2$ implies this condition but not the other way around). In the following, we argue why this difference indeed is significant.

To discuss the difference of the above predicates, we consider the following example: we define two automata A and A' as pictured in Figure 3.2. Their output interfaces are identical: $Out(A) = Out(A') = \{c\}$. Their input interfaces are defined as $In(A) = \{a\}$ and $In(A') = \{a, b\}$, respectively. For A, the transition label t in Figure 3.2 is substituted by a/c and for A', by $a \wedge \neg b/c$. We further assume that both A and A' are in their initial configurations. Let the input event, delivered by the environment, be $\{a, b\}$. If the input event is $\{a, b\}$, then the deterministic output of A is $\{c\}$, since the signal a is included in the input event. With the same input event A' yields either $\{c\}$ or $\emptyset$ as possible output events, because for A' we merely have specified a reaction if a is present and b is absent. The case that both a and b are present has been left unspecified. As A' introduces additional non-determinism, it should not be a refinement of A. However, applying Condition 3.3, A' would be considered as a refinement of A. This is explained by the fact that this predicate just examines input events in $\mathcal{I}(A) = \{\emptyset, \{a\}\}$ and not in $\mathcal{I}(A') = \{\emptyset, \{a\}, \{b\}, \{a, b\}\}$. For all input events in $\mathcal{I}(A)$ both automata yield equivalent behaviors. This is not the case for those in $\mathcal{I}(A')$. In order to distinguish A and A', Condition 3.2 has to be applied.

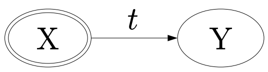


Figure 3.2: Interface refinement

The reason why Condition 3.3 yields a wrong result and Condition 3.2 does not, is explained as follows: dealing with input events consisting of sets of signals, each chart exactly reads that part of the event that is defined in its input interface. Unknown signals are simply ignored. Thus, the presence of unknown signals in input events does not necessarily lead to chaotic behavior. This is in contrast to approaches where merely single signals can be input to components. In [Rum96], for instance, if already one such signal is unknown the reaction yields chaotic behavior in any case.

We want our refinement calculus to be stepwise applicable. Therefore, we have to guarantee that also a sequence of refinement steps is a refinement of the original specification again. As $\rightsquigarrow$ is transitive, we can guarantee this in our case.

Theorem 3.1.1 (Refinement is Transitive)
The refinement relation $\rightsquigarrow$ is transitive. $\square$

Proof 4 Let S_1, S_2, and S_3 be arbitrary charts. We assume that $S_1 \rightsquigarrow S_2$ and $S_2 \rightsquigarrow S_3$ or, equivalently:

$$\forall i \in \mathcal{I}(S_2)^\infty, o \in \mathcal{O}(S_2)^\infty : (i, o) \in [\![S_2]\!]_{io} \Rightarrow (i|_{In(S_1)}, o|_{Out(S_1)}) \in [\![S_1]\!]_{io} \text{ and}$$
$$\forall i \in \mathcal{I}(S_3)^\infty, o \in \mathcal{O}(S_3)^\infty : (i, o) \in [\![S_3]\!]_{io} \Rightarrow (i|_{In(S_2)}, o|_{Out(S_2)}) \in [\![S_2]\!]_{io}$$

We have to show that

$$\forall i \in \mathcal{I}(S_3)^\infty, o \in \mathcal{O}(S_3)^\infty : (i, o) \in [\![S_3]\!]_{io} \Rightarrow (i|_{In(S_1)}, o|_{Out(S_1)}) \in [\![S_1]\!]_{io}$$

Let $i \in \mathcal{I}(S_3)^\infty, o \in \mathcal{O}(S_3)^\infty$ such that $(i, o) \in [\![S_3]\!]_{io}$. Then $(i|_{In(S_2)}, o|_{Out(S_2)}) \in [\![S_2]\!]_{io}$ and $(i|_{In(S_2) \cap In(S_1)}, o|_{Out(S_2) \cap Out(S_1)}) \in [\![S_1]\!]_{io}$. Since $In(S_1) \subseteq In(S_2) \subseteq In(S_3)$ and $Out(S_1) \subseteq Out(S_2) \subseteq Out(S_3)$ this is equivalent to $(i|_{In(S_1)}, o|_{Out(S_1)}) \in [\![S_1]\!]_{io}$, which concludes the proof. $\square$

Besides transitivity, compositionality is a further important property of refinement. It secures that whenever a small part of a large specification is refined, also the entire specification is refined. In the following theorem, it is essential that $L \subseteq Out(S_1) \cup Out(S_3)$ and not $L \subseteq Out(S_2) \cup Out(S_3)$. Otherwise, as $Out(S_1) \subseteq Out(S_2)$, S_2 could possibly display additional behavior due to extra communication between S_2 and S_3 that is not possible between S_1 and S_3.

Theorem 3.1.2 (Refinement is Compositional)
Let S_1, S_2, and S_3 be arbitrary μ-Charts. If $S_1 \rightsquigarrow S_2$ then also $S_1 \lhd L \rhd S_3 \rightsquigarrow S_2 \lhd L \rhd S_3$ for an arbitrary signal set L with $L \subseteq Out(S_1) \cup Out(S_3)$. □

Proof 5 We define $S_{13} =_{df} S_1 \lhd L \rhd S_3$ and $S_{23} =_{df} S_2 \lhd L \rhd S_3$. We have to show that $\{(i|_{In(S_1)}, o|_{Out(S_1)}) \mid (i, o) \in [\![S_2]\!]_{io}\} \subseteq [\![S_1]\!]_{io}$ implies $\{(i|_{In(S_{13})}, o|_{Out(S_{13})} \mid (i, o) \in [\![S_2 \lhd L \rhd S_3]\!]_{io}\} \subseteq [\![S_1 \lhd L \rhd S_3]\!]_{io}$. Let $(i, o) \in [\![S_2 \lhd L \rhd S_3]\!]_{io}$ then there exists $(c_2, c_3) \in Init(S_2 \lhd L \rhd S_3)$ with $((c_2, c_3), i, o) \in [\![S_2 \lhd L \rhd S_3]\!]$. As a consequence, there also exist $o_2 \in Out(S_2), o_3 \in Out(S_3)$ with $o = o_2 \uplus o_3$ and

$$\begin{aligned}&(c_2, (i \uplus (o_2 \uplus o_3)|_L)|_{In(S_2)}, o_2) \in [\![S_2]\!] \land \\ &(c_3, (i \uplus (o_2 \uplus o_3)|_L)|_{In(S_3)}, o_3) \in [\![S_3]\!]\end{aligned}$$

As $S_1 \rightsquigarrow S_2$ there also exists $c_1 \in Init(S_1)$ with

$$(c_1, (i \uplus (o_2 \uplus o_3)|_L)|_{In(S_1)}, o_2|_{Out(S_1)}) \in [\![S_1]\!]$$

Since $L \subseteq Out(S_1) \cup Out(S_3)$ this is equivalent to

$$(c_1, (i \uplus (o_2|_{Out(S_1)} \uplus o_3)|_L)|_{In(S_1)}, o_2|_{Out(S_1)}) \in [\![S_1]\!]$$

Accordingly, the above formula for S_3 can be re-written to the equivalent form

$$(c_3, (i \uplus (o_2|_{Out(S_1)} \uplus o_3)|_L)|_{In(S_3)}, o_3) \in [\![S_3]\!]$$

Hence, with $o_1 =_{df} o_2|_{Out(S_1)}$ there also exists $o_1 \in Out(S_1)$ with $o = o_1 \uplus o_3$ and

$$\begin{aligned}&(c_1, (i \uplus (o_1 \uplus o_3)|_L)|_{In(S_1)}, o_1) \in [\![S_1]\!] \land \\ &(c_3, (i \uplus (o_1 \uplus o_3)|_L)|_{In(S_3)}, o_3) \in [\![S_3]\!]\end{aligned}$$

This concludes the proof. □

Also refinements of specifications including the hiding operator are compositional, that is, whenever $S_1 \rightsquigarrow S_2$ then also $[S_1]_K \rightsquigarrow [S_2]_K$ for arbitrary signal set K. The proof follows directly from the definition of the semantics of $[S]_K$.

3.2 The Calculus

Up to now, we have formally defined our notion of refinement for μ-Charts and have verified that our refinement notion is transitive and compositional. This part of the thesis is intended more as background information for the theoretically inclined computer scientist than for a software engineer. Software engineers, however, should not feel obliged to understand all mathematical details. Rather, for them it is essential to have a set of syntactic rules at hand which they can use to develop reactive systems step-by-step.

In this section, we present syntactic refinement rules and prove their soundness. The only task that remains for the software engineer is then to verify that she or he meets the syntactic guidelines when applying a specific rule. The refinement calculus we present is not complete. Hence, there are pairs of μ-Chart specifications (S_1, S_2), where S_2 is a refinement of S_1, but S_1 cannot be refined to S_2 by merely applying rules of our calculus — possibly even an infinite sequence of refinement rule applications does not yield the desired result.

In particular, we omitted rules for transformations of composed automata in the corresponding flat product automaton and vice versa. A product automaton of two automata composed in parallel can be constructed automatically and therefore needs no user interaction. Moreover, this construction leads to the well-known state explosion problem. Thus, it is not reasonable from a methodological point of view to support it by a transformation rule. Though the opposite transformation leads to specifications with more flexibility of implementation and possibly opens a wider spectrum of distributed implementation alternatives, we do not give a refinement rule for it. An appropriate syntactic rule would require quite a number of context restrictions that would not be useful for practical applications. Furthermore, our calculus does not contain any rule to refine decomposed charts in flat automata and the other way around.

At first sight, the reader also would expect a separate rule for signal hiding. However, since hiding decreases the output interface, we are not able to give an isolated rule for this operator. Remember that allowing both increasing and decreasing of interfaces impedes the transitivity of refinement. We decided to merely permit increasing of interfaces in order to guarantee that composition and hierarchical decomposition are correct refinements. Hence, though we cannot define a separated refinement rule for signal hiding, we will give rules for it in combination with composition and decomposition.

Altogether, the focus of our calculus was not to give a complete set of theoretically possible rules, but to support the user with powerful and at the same time easy-to-apply rules. Hence, we attached more importance to developing a set of methodologically beneficial rules than to defining a calculus that is as complete as possible.

3.2.1 Rules for Sequential Automata

To show that a sequential automaton A_2 is a refinement of another sequential automaton A_1, where $A_k =_{df} (I_k, O_k, \Sigma_k, \Sigma_{0k}, V_{lk}, \varphi_{0k}, \delta_k)$ with $k = 1, 2$ we have to show that

$$\forall i \in \mathcal{I}(A_2)^\infty, o \in \mathcal{O}(A_2)^\infty : (i, o) \in [\![A_2]\!]_{io} \Rightarrow (i|_{In(A_1)}, o|_{Out(A_1)}) \in [\![A_1]\!]_{io}$$

Proving this property is equivalent to showing that the greatest fixed point of the characterizing function of A_2 is a subset of the greatest fixed point of the characterizing function of A_1 (see Page 47). To use this inclusion for our soundness proofs later on is mathematically relatively complicated. Hence, we try to transform this formula to an equivalent one, which is easier to prove. The following lemma provides a first step in this direction. As the following material contains theoretical background information, the practical oriented software engineer can skip the following propositions and re-start reading on Page 78.

Lemma 3.2.1 (Refinement of Sequential Automata)
The automaton A_2 is a refinement of another sequential automaton A_1 if and only if the following holds:

$$\begin{array}{l}\forall i, o.((i \in \mathcal{I}(A_2)^\infty \wedge o \in \mathcal{O}(A_2)^\infty \Rightarrow \\ \quad \exists c_2.(c_2 \in \mathit{Init}(A_2) \wedge \exists X.(X \subseteq \mathcal{C}(A_2) \times \mathcal{I}(A_2)^\infty \times \mathcal{O}(A_2)^\infty \wedge \\ \quad X \subseteq F_{A_2}(X) \wedge (c_2, i, o) \in X))) \Rightarrow \\ \quad \exists c_1.(c_1 \in \mathit{Init}(A_1) \wedge \exists X.(X \subseteq \mathcal{C}(A_1) \times \mathcal{I}(A_1)^\infty \times \mathcal{O}(A_1)^\infty \wedge \\ \quad X \subseteq F_{A_1}(X) \wedge (c_1, i|_{In(A_1)}, o|_{Out(A_1)}) \in X)))\end{array}$$

□

Proof 6 Let $i \in \mathcal{I}(A_2)^\infty$ and $o \in \mathcal{O}(A_2)^\infty$ then (we abbreviate $\tau_i =_{df} \mathcal{C}(A_i) \times \mathcal{I}(A_i)^\infty \times \mathcal{O}(A_i)^\infty$):

$$\begin{array}{l}((i,o) \in [\![A_2]\!]_{io} \Rightarrow (i|_{In(A_1)}, o|_{Out(A_1)}) \in [\![A_1]\!]_{io}) \iff \\ (\exists c_2.(c_2 \in \mathit{Init}(A_2) \wedge (c_2, i, o) \in \mathrm{gfp}(F_{A_2})) \Rightarrow \\ \quad \exists c_1.(c_1 \in \mathit{Init}(A_1) \wedge (c_1, i|_{In(A_1)}, o|_{Out(A_1)}) \in \mathrm{gfp}(F_{A_1}))) \iff \\ (\exists c_2.(c_2 \in \mathit{Init}(A_2) \wedge (c_2, i, o) \in \bigcup\{X \subseteq \tau_2 \mid X \subseteq F_{A_2}(X)\}) \Rightarrow \\ \quad \exists c_1.(c_1 \in \mathit{Init}(A_1) \wedge (c_1, i|_{In(A_1)}, o|_{Out(A_1)}) \in \bigcup\{X \subseteq \tau_1 \mid X \subseteq F_{A_1}(X)\})) \iff \\ (\exists c_2.(c_2 \in \mathit{Init}(A_2) \wedge \exists X.(X \subseteq \tau_2 \wedge X \subseteq F_{A_2}(X) \wedge (c_2, i, o) \in X)) \Rightarrow \\ \quad \exists c_1.(c_1 \in \mathit{Init}(A_1) \wedge \exists X.(X \subseteq \tau_1 \wedge X \subseteq F_{A_1}(X) \wedge (c_1, i|_{In(A_1)}, o|_{Out(A_1)}) \in X)))\end{array}$$

□

The predicate that Lemma 3.2.1 offers as proof obligation for refinement of sequential automata still seems to be complicated. Fortunately, with further restrictions we can find yet an easier equivalent proof obligation. Whenever both A_1 and A_2 have identical interfaces and control and data states, the predicate of Lemma 3.2.1 can be weakened as follows.

Corollary 3.2.2 (Refinement of Sequential Automata)
The automaton $A_2 =_{df} (I, O, \Sigma, \Sigma_0, V_l, \varphi_0, \delta_2)$ is a refinement of the sequential automaton $A_1 =_{df} (I, O, \Sigma, \Sigma_0, V_l, \varphi_0, \delta_1)$ if the following formula holds:
$\forall X \subseteq \mathcal{C}(A_1) \times \mathcal{I}(A_1)^\infty \times \mathcal{O}(A_1)^\infty : X \subseteq F_{A_2}(X) \Rightarrow F_{A_2}(X) \subseteq F_{A_1}(X)$ □

Proof 7 Let $\forall X \subseteq \mathcal{C}(A_1) \times \mathcal{I}(A_1)^\infty \times \mathcal{O}(A_1)^\infty : X \subseteq F_{A_2}(X) \Rightarrow F_{A_2}(X) \subseteq F_{A_1}(X)$ be true. Since the input/output interfaces, the initial/control states, the local variables, and their initialization functions are identical, A_1 and A_2 only differ in their transition relations δ_1 and δ_2. In particular, we have $\mathcal{I}(A_1) = \mathcal{I}(A_2)$, $\mathcal{O}(A_1) = \mathcal{O}(A_2)$, $\mathcal{C}(A_1) = \mathcal{C}(A_2)$, and $Init(A_1) = Init(A_2)$. Thus, the assumption implies

$$\forall i, o.(\exists c.(\exists X.(X \subseteq F_{A_2}(X) \wedge (c, i, o) \in X \Rightarrow F_{A_2}(X) \subseteq F_{A_1}(X))))$$

This is a simplification of the predicate in Lemma 3.2.1. □

With these preliminaries we have set the stage for our presentation of the refinement rules for sequential automata. We provide rules to remove initial states, to add states and transitions, and to remove and modify transitions. Though these rules have been independently developed they are similar to those introduced in [Rum96], but are specialized to μ-Charts and their synchronous semantics. Furthermore, our soundness proofs are mathematically simpler than those in [Rum96], which are not based on a relational semantic model as μ-Charts, but on one that is based on sets of stream processing functions.

Removing States

A sequential automaton may have more than one initial state and therefore may reflect non-deterministic behavior as it can start its reaction in either of the initial states. The reduction of the initial states Σ_0 to $\Sigma_0' \subseteq \Sigma_0$ is a correct refinement step (see Figure 3.3), because the auxiliary semantics (of Section 2.4.3) is not affected at all by this modification, that is, $[\![(I, O, \Sigma, \Sigma_0, V_l, \varphi_0, \delta)]\!] = [\![(I, O, \Sigma, \Sigma_0', V_l, \varphi_0, \delta)]\!]$. Since the number of initial configurations is reduced, it is easy to see that the overall input/output semantics becomes more defined: $[\![(I, O, \Sigma, \Sigma_0', V, \varphi_0, \delta)]\!]_{io} \subseteq [\![(I, O, \Sigma, \Sigma_0, V_l, \varphi_0, \delta)]\!]_{io}$.

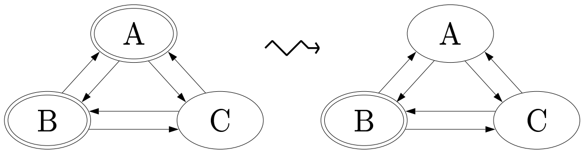


Figure 3.3: Removing initial states

To motivate this rule by an example, we apply it to a part of our central locking system in Figure 2.3 and reduce the initial states of the automaton NORMAL from {UNLD, LOCKED} to {UNLD}. The automaton we obtain is a correct refinement of the original automaton with two initial states.

Moreover, states that are not reachable via a series of transitions in the graph-theoretical sense and in addition are not included in the set of initial states Σ_0 can be removed from the specification. Such states cannot contribute to the system reaction as they never can be part of a system configuration. If we consider A as a directed graph, then these states represent unreachable nodes. Together with an unreachable state σ, all transitions that have σ as source state can be removed (see Figure 3.4). Since σ is not reachable, they never can be fired. The formal justification for this rule directly follows by application of Lemma 3.2.1.

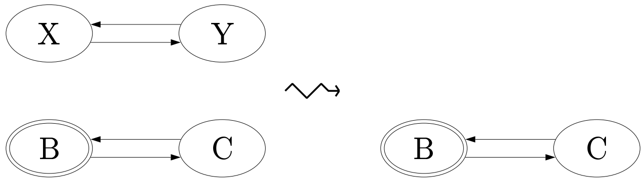


Figure 3.4: Removing states

Theorem 3.2.3 (Removing States)
Let $A_1 =_{df} (I, O, \Sigma_1, \Sigma_0, V_l, \varphi_0, \delta_1)$ and $A_2 =_{df} (I, O, \Sigma_2, \Sigma_0, V_l, \varphi_0, \delta_2)$, where $\Sigma_2 = \{\sigma \in \Sigma_1 \,|\, \sigma \notin \Omega\}$. Here, $\Omega \subseteq \Sigma_1$ denotes the set of graph-theoretically unreachable states with respect to the transition relation δ_1 and $\Omega \cap \Sigma_0 = \emptyset$. If $\forall \sigma \in \Sigma_2 : \delta_2(\sigma) = \delta_1(\sigma)$ then $[\![A_1]\!]_{io} = [\![A_2]\!]_{io}$. □

Proof 8 Let $A_1 = (I, O, \Sigma_1, \Sigma_0, V_l, \varphi_0, \delta_1)$ and $A_2 = (I, O, \Sigma_2, \Sigma_0, V_l, \varphi_0, \delta_2)$, where $\Sigma_2 = \{\sigma \in \Sigma_1 \,|\, \sigma \notin \Omega\}$ and $\Omega \subseteq \Sigma_1$ denotes the set of graph-theoretically unreachable states with respect to the transition relation δ_1. We additionally require that $\Omega \cap \Sigma_0 = \emptyset$. Hence, $\Sigma_0 \cap \Sigma_2 = \Sigma_0$. Furthermore, δ_2 is defined by δ_1 as follows: $\forall \sigma \in \Sigma_2 : \delta_2(\sigma) = \delta_1(\sigma)$ and therefore $\forall \sigma \in \Sigma_2, \varepsilon \in V_l \to \mathbb{Z}, x \in \wp(I)^\infty : [\![\delta_2]\!]((\sigma, \varepsilon), x) = [\![\delta_1]\!]((\sigma, \varepsilon), x)$. As a consequence, for all $X \subseteq \Sigma_0 \times \wp(I)^\infty \times \wp(O)^\infty$ holds $F_{A_2}(X) = F_{A_1}(X)$, where F_{A_2} and F_{A_1} are the characteristic functions of A_2 and A_1, respectively. In order to carry out the remainder of this proof, we follow the strategy proposed in Lemma 3.2.1. Thus, let $i \in \wp(I)^\infty$, $o \in \wp(O)^\infty$ and let the premise of Lemma 3.2.1, that is

$$\exists c_2.(c_2 \in \Sigma_0 \wedge \exists X_2.(X_2 \subseteq \Sigma_0 \times \wp(I)^\infty \times \wp(O)^\infty \wedge X_2 \subseteq F_{A_2}(X_2) \wedge (c_2, i, o) \in X_2))$$

be true. Then, we have to show that

$$\exists c_1.(c_1 \in \Sigma_0 \wedge \exists X_1.(X_1 \subseteq \Sigma_0 \times \wp(I)^\infty \times \wp(O)^\infty \wedge X_1 \subseteq F_{A_1}(X_1) \wedge (c_1, i, o) \in X_1))$$

holds. To perform this proof, we choose $c_1 = c_2$ and $X_1 = X_2 = X$. Since $F_{A_2}(X) = F_{A_1}(X)$ for $X \subseteq \Sigma_0 \times \wp(I)^\infty \times \wp(O)^\infty$, $X \subseteq F_{A_2}(X)$ implies $X \subseteq F_{A_1}(X)$, which implies, since $\Omega \cap X = \emptyset$, $[\![A_1]\!]_{io} = [\![A_2]\!]_{io}$ and thus concludes the proof. □

Adding States

The set of states Σ of a sequential automaton can be enlarged by Σ' and the semantics of the resulting automaton stays exactly the same as long as the initial states are not modified and the new states are not "connected" to the rest of the automaton with already existing transitions (see Figure 3.5). In subsequent refinement steps, however, theses states can be connected with new transitions according to the refinement rules for adding transitions. More formally, the sequential automaton $(I, O, \Sigma, \Sigma_0, V_l, \varphi_0, \delta)$ is equivalent to the automaton $(I, O, \Sigma \cup \Sigma', \Sigma_0, V_l, \varphi_0, \delta)$ if Σ' is a set of new control states not contained in Σ.

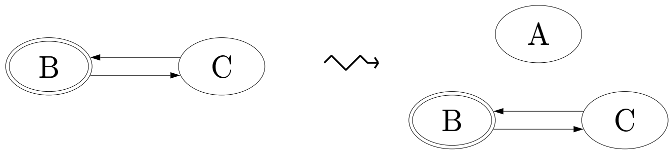


Figure 3.5: Adding states

Theorem 3.2.4 (Adding States)
Let $A =_{df} (I, O, \Sigma, \Sigma_0, V_l, \varphi_0, \delta)$ and Σ' is a set of new control states with $\Sigma' \cap \Sigma = \emptyset$. Then $[\![A]\!]_{io} = [\![(I, O, \Sigma \cup \Sigma', \Sigma_0, V_l, \varphi_0, \delta)]\!]_{io}$. □

Proof 9 Since all $\sigma \in \Sigma$ are isolated, the semantics $[\![(I, O, \Sigma \cup \Sigma', \Sigma_0, V_l, \varphi_0, \delta)]\!]$ is denoted by

$$[\![A]\!] \cup \bigcup_{\sigma \in \Sigma'} \{(\sigma, i, o) \mid i \in \wp(I)^\infty \wedge o \in \wp(O)^\infty\}$$

From this $[\![A]\!]_{io} = [\![(I, O, \Sigma \cup \Sigma', \Sigma_0, V_l, \varphi_0, \delta)]\!]_{io}$ directly follows as $\Sigma' \cap \Sigma_0 = \emptyset$ and no state in Σ' is connected to the rest of A. □

Manipulating Transitions

Now, we will discuss under which circumstances transitions can be removed, added, and modified, respectively. In the sequel, let A_n be the automaton $(I, O, \Sigma, \Sigma_0, V_l, \varphi, \delta_n)$, for $n = 1, 2$, i.e. A_1 and A_2 only differ in their transition relations, but besides this are identical. We start with a rule to remove transitions.

(a) Removing Transitions

If we obtain A_2 from A_1 by deleting the transition (t, com, σ') from $\delta_1(\sigma)$, i.e. $\delta_1(\sigma) = \delta_2(\sigma) \cup \{(t, com, \sigma')\}$, this is a correct refinement step if $t \Rightarrow \bigvee_{t' \in T_{\delta_2}(\sigma)} t'$ is a tautology, where $T_\delta(\sigma)$ yields the first projection, that is, the trigger condition of $\delta(\sigma)$. The premise here means that the trigger condition t of the removed transition is already subsumed in the trigger conditions of the remaining, i.e. not removed transitions that have σ as their source state. The premise guarantees that additional non-determinism is not introduced by removing the transition with trigger t (see Figure 3.6).

Theorem 3.2.5 (Removing Transitions)
Let $A_1 =_{df} (I, O, \Sigma, \Sigma_0, V_l, \varphi_0, \delta_1)$ and $A_2 =_{df} (I, O, \Sigma, \Sigma_0, V_l, \varphi_0, \delta_2)$, where $\delta_1(\sigma) = \delta_2(\sigma) \cup \{(t, com, \sigma')\}$. If $t \Rightarrow \bigvee_{t' \in T_{\delta_2}(\sigma)} t'$ is a tautology, where $T_{\delta_2}(\sigma)$ yields the first projection of $\delta_2(\sigma)$, then $[\![A_2]\!] \subseteq [\![A_1]\!]$. □

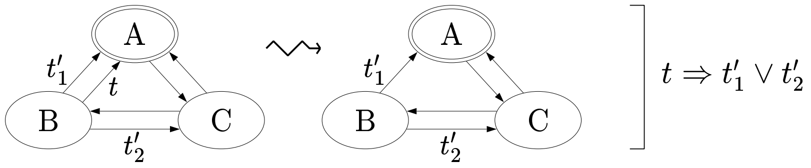


Figure 3.6: Removing transitions

Proof 10 According to Corollary 3.2.2 we take an arbitrary $X \subseteq \mathcal{C}(A_2) \times \mathcal{I}(A_2)^\infty \times \mathcal{O}(A_2)^\infty$. Now let the following be true:

$$\exists c'.(c', y) \in [\![\delta_2]\!](c, x) \wedge (c', i, o) \in X \vee [\![\delta_2]\!](c, x) = \emptyset$$

Then, according to Corollary 3.2.2, we have to show that

$$\exists c'.(c', y) \in [\![\delta_1]\!](c, x) \wedge (c', i, o) \in X \vee [\![\delta_1]\!](c, x) = \emptyset$$

This is done by case analysis. First, we assume that $\exists c'.(c', y) \in [\![\delta_2]\!](c, x) \wedge (c', i, o) \in X$. As $\delta_2(\sigma) \subseteq \delta_1(\sigma)$ implies immediately for all ε and x that $[\![\delta_2]\!]((\sigma, \varepsilon), x) \subseteq [\![\delta_1]\!]((\sigma, \varepsilon), x)$ — note that $\delta(\sigma)$ only occurs positively in the definition of $[\![\delta]\!]((\sigma, \varepsilon), x)$ and therefore this semantic function is monotonic with respect to $\delta(\sigma)$ — this yields the desired result. Second, we assume that $[\![\delta_2]\!](c, x) = \emptyset$. According to the semantic definitions of Section 2.4.3, this means that there does not exist σ'_2, ε'_2, y_2, com_2, and t_2 with

$$(t_2, com_2, \sigma'_2) \in \delta_2(\sigma) \wedge (\varepsilon, x) \in [\![t_2]\!]_{A_2} \wedge (\varepsilon'_2, y_2) = \mathcal{R}[\![com_2]\!]\varepsilon$$

It remains to be shown that there also does not exist σ'_1, ε'_1, y_1, com_1, and t_1 with

$$(t_1, com_1, \sigma'_1) \in \delta_1(\sigma) \wedge (\varepsilon, x) \in [\![t_1]\!]_{A_1} \wedge (\varepsilon'_1, y_1) = \mathcal{R}[\![com_1]\!]\varepsilon$$

We prove this by contradiction and assume that there in fact are σ'_1, ε'_1, y_1, com_1, and t_1 such that the above formula holds. Since $\delta_1(\sigma) = \delta_2(\sigma) \cup \{(t, com, \sigma')\}$ the only possibility to choose t_1, com_1, and σ'_1 is t, com, and σ', respectively. The syntactic restriction for the application of this refinement rule $t \Rightarrow \bigvee_{t' \in T_{\delta_2}(\sigma)} t'$ implies $[\![t]\!]_{A_1} \subseteq [\![\bigvee_{t' \in T_{\delta_2}(\sigma)} t']\!]_{A_2}$. Moreover, $[\![\bigvee_{t' \in T_{\delta_2}(\sigma)} t']\!]_{A_2}$ is equivalent to $\bigcup_{t' \in T_{\delta_2}(\sigma)} [\![t']\!]_{A_2}$. Hence, this set inclusion is equivalent to

$$\forall (\varepsilon, x) \in [\![t]\!]_{A_1}\, \exists t' \in T_{\delta_2}(\sigma) : (\varepsilon, x) \in [\![t']\!]_{A_2}$$

which yields a contradiction to the original proof assumption and also the second case is proven. $\square$

Applying this rule we can simply remove either of the two transitions with START as source state in the chart pictured in Figure 2.5. For instance, we could remove the transition from START to STOP. A software engineer can easily verify this by examination of the triggers of the outgoing transitions from state START. In Figure 2.5, we omitted the trigger conditions. Recall that "empty" pre-conditions on transitions mean that this transition is enabled whenever its source state is included in the current configuration; further restrictions are not made. Hence, the actual trigger condition is simply `true`. The syntactic rule for removing transitions requires to verify for the example at hand that `true` $\Rightarrow$ `true` is a tautology. On the other hand, if we remove the transition from START to TWO, which correctness also is guaranteed by `true` $\Rightarrow$ `true` being a tautology, the state TWO becomes unreachable and therefore can be removed together with the transition from TWO to STOP according to the rule for removal of unreachable states.

(b) Adding Transitions

If we obtain A_2 from A_1 by adding the transition (t, com, σ') to $\delta_1(\sigma)$, i.e. $\delta_2(\sigma) = \delta_1(\sigma) \cup \{(t, com, \sigma')\}$, this is a correct refinement step if $\forall t' \in T_{\delta_1}(\sigma) : t \wedge t'$ is a contradiction. Informally, this premise requires that the trigger condition t of the added outgoing transition of control state σ does not interfere with any trigger of the other, already available transitions that have σ as source state. Here, "not to interfere" means that no possible input event of A_2 could yield non-deterministic behavior that was not included in A_1. Additional non-determinism could occur, for instance, if there existed an input event that could stimulate the new transition and those in $\delta_1(\sigma)$. This clash is avoided by requiring that each conjunction $t \wedge t'$ is a contradiction (see Figure 3.7).

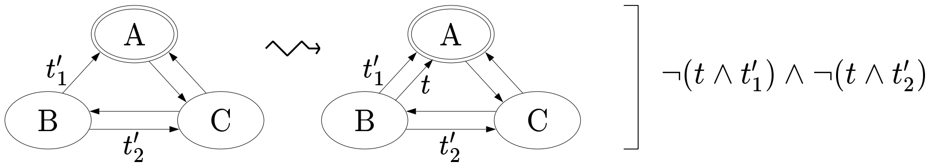


Figure 3.7: Adding transitions

Theorem 3.2.6 (Adding Transitions)
Let $A_1 =_{df} (I, O, \Sigma, \Sigma_0, V_l, \varphi_0, \delta_1)$ and $A_2 =_{df} (I, O, \Sigma, \Sigma_0, V_l, \varphi_0, \delta_2)$, where $\delta_2(\sigma) = \delta_1(\sigma) \cup \{(t, com, \sigma')\}$. If $\forall t' \in T_{\delta_1}(\sigma) : t \wedge t'$ is a contradiction, where $T_{\delta_1}(\sigma)$ yields the first projection of $\delta_1(\sigma)$, then $[\![A_2]\!] \subseteq [\![A_1]\!]$. $\square$

Proof 11 According to Corollary 3.2.2, we take an arbitrary $X \subseteq \mathcal{C}(A_1) \times \mathcal{I}(A_1)^\infty \times \mathcal{O}(A_1)^\infty$ with $X \subseteq F_{A_2}(X)$. Now let the following be true: $\exists c'.((c', y) \in [\![\delta_2]\!](c, x) \wedge (c', i, o) \in X) \vee [\![\delta_2]\!](c, x) = \emptyset$. As we must prove $F_{A_2}(X) \subseteq F_{A_1}(X)$ we have to show that $\exists c'.((c', y) \in [\![\delta_1]\!](c, x) \wedge (c', i, o) \in X) \vee [\![\delta_1]\!](c, x) = \emptyset$. This is done by case analysis. First, we assume that $\exists c'.(c', y) \in [\![\delta_2]\!](c, x) \wedge (c', i, o) \in X$. Let $c = (\sigma, \varepsilon)$. We define $\delta_3(\sigma) =_{df} \{(t, com, \sigma')\}$. Since $[\![\delta_2]\!](c, x) = [\![\delta_1]\!](c, x) \cup [\![\delta_3]\!](c, x)$ — note that $\delta_2(\sigma) = \delta_1(\sigma) \cup \delta_3(\sigma)$ implies $[\![\delta_2]\!]((\sigma, \varepsilon), x) = [\![\delta_1]\!]((\sigma, \varepsilon), x) \cup [\![\delta_3]\!]((\sigma, \varepsilon), x)$; this follows directly from the definition of $[\![\delta]\!]$ and the distributivity of $\cup$ — the tuple (c', y) must either be in $[\![\delta_1]\!](c, x)$ or in $[\![\delta_3]\!](c, x)$. If $(c', y) \in [\![\delta_1]\!](c, x)$ the proof is already completed. Otherwise, $(c', y) \in [\![\delta_3]\!](c, x)$ and thus $(\varepsilon, x) \in [\![t]\!]_{A_2}$ must hold for the added transition with trigger t. From this we can deduce that $\forall t' \in T_{\delta_1}(\sigma) : (\varepsilon, x) \notin [\![t']\!]_{A_1}$ because $\forall t' \in T_{\delta_1}(\sigma) : t \wedge t' \Leftrightarrow \mathrm{ff}$ and therefore

$$\forall t' \in T_{\delta_1}(\sigma) : [\![t]\!]_{A_1} \cap [\![t']\!]_{A_1} = [\![t \wedge t']\!]_{A_1} = [\![\mathrm{ff}]\!]_{A_1} = \emptyset$$

As a consequence, we get $[\![\delta_1]\!](c,x) = \emptyset$ what yields the desired result. Second, we assume that $[\![\delta_2]\!](c,x) = \emptyset$. As $\delta_1(\sigma) \subseteq \delta_2(\sigma)$ implies $[\![\delta_1]\!](c,x) \subseteq [\![\delta_2]\!](c,x)$ also $[\![\delta_1]\!](c,x) = \emptyset$ holds. □

The first case of the proof says that a new transition can only make the specification more precise, but not more chaotic. The second case guarantees that chaotic behavior of A_2 must already have been chaotic in A_1.

Of course, the application of the rule for adding transitions is not restricted to add merely single transitions. Rather, also more than one transition, say $n \in \mathbb{N}$, with equivalent trigger conditions can be added, that is, $\delta_2(\sigma) = \delta_1(\sigma) \cup \{(t, com_1, \sigma'_1), \ldots, (t, com_n, \sigma'_n)\}$. The correctness of this rule follows straightforwardly from the above theorem by substituting the singleton $\delta_3(\sigma)$ by $\{(t, com_1, \sigma'_1), \ldots, (t, com_n, \sigma'_n)\}$ in the proof.

The rule for adding transitions is a very powerful means to transform an abstract specification in a more concrete one. Take, for instance, all sequential automata in the Figures 2.3, 2.4, and 2.5. In many of their control states, these automata react chaotically since the system behavior is completely under-specified when relevant signals do not occur in a step. In both states Down and Up, for example, nothing is said how the motors should behave when the signal *ready* is absent. Under-specifications of this type simply can be made more concrete by adding additional self-loops with label $\neg ready/skip$. As $ready \wedge \neg ready$ equals false, this is a correct refinement step. This way, the automaton does not behave chaotically any more and the specification performs what the designer would expect: whenever the signal *ready* is not present simply nothing happens at all. We introduce these self-loops in all automata that build the central locking system where they are necessary to get a responsive specification.

As we come closer to implementation, for the remainder of this thesis, we assume that these self-loops with skip actions are implicitly given, even if they are not explicitly drawn.

(c) Modifying Transitions

Let $A =_{df} (I, O, \Sigma, \Sigma_0, V_l, \varphi_0, \delta)$, $\sigma \in \Sigma$ a state in A, and $e \in \delta(\sigma)$ the transition to be modified. The two symbols δ and δ' denote the transition relations before and after one modification, respectively. The sequential automaton $(I, O, \Sigma, \Sigma_0, \varphi_0, \delta')$ is a correct refinement of the automaton A whenever δ' yields more concrete, but not more abstract behavior. We then can identify the following rules:

1. The trigger condition t, where $e = (t, com, \sigma')$ and σ' not necessarily differs from σ, can be refined to:

    (a) $t \vee t'$ if the first order formula $\forall t'' \in T_\delta(\sigma) : t \neq t'' \Rightarrow (t' \wedge t'')$ is a contradiction (see Figure 3.8)

    (b) $t \wedge t'$ if the first order formula $(t \wedge \neg t') \Rightarrow \left(\bigvee_{t'' \in T_{\delta'}(\sigma) : t \wedge t' \neq t''} t''\right)$ is a tautology for an arbitrary Boolean term t' over the algebraic sum of input interface and local variables $Bexp(I + V_l)$ of automaton A.

2. The trigger condition $a \wedge a'$, where $e = (a \wedge a', com, \sigma)$, can be refined to a for an arbitrary Boolean term a' in $Bexp(I + V_l)$ if $\forall t'' \in T_\delta(\sigma) : (a \wedge a' \neq t'' \Rightarrow ((a \wedge \neg a') \wedge t'')$ is a contradiction.

3. The trigger condition $b \vee b'$, where $e = (b \vee b', com, \sigma)$, can be refined to b for an arbitrary Boolean term b' over $Bexp(I + V_l)$ if $b' \wedge \neg b \Rightarrow \left(\bigvee_{t'' \in T_{\delta'}(\sigma) : b \neq t''} t''\right)$ is a tautology.

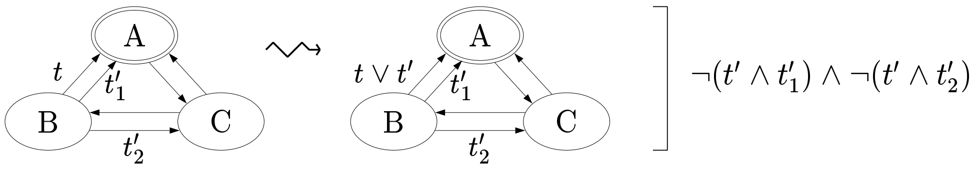


Figure 3.8: Modifying transitions

Rule number (1) can be proven from the rules for adding (1.a) and deleting (1.b) transitions. Rules (2) and (3) are deduced from (1) when t is substituted by $a \wedge a'$ and $b \vee b'$, respectively, t' by $a \wedge \neg a'$ and $b \vee \neg b'$, respectively, and the following equivalences are used:

$$a = a \wedge (\neg a' \vee a') = (a \wedge \neg a') \vee (a \wedge a')$$
$$b = b \vee (b' \wedge \neg b') = (b \vee b') \wedge (b \vee \neg b')$$

In the central locking system, we can, for instance, modify the transition trigger ldn in MotorLeft to $ldn \wedge \neg lup$. As $lup \wedge ldn \Rightarrow ldn \vee (lup \wedge \neg ldn)$ is a tautology, this is a correct refinement step. If we wanted to refine the remaining transition with label ldn to $ldn \wedge \neg lup$, too, we would not get a correct refinement step, because $ldn \wedge lup \Rightarrow (ldn \wedge \neg lup) \vee (lup \wedge \neg ldn)$ is no longer a tautology. The chart for the motor in the right car door MotorRight can be similarly transformed to a deterministic chart.

Finally, we would like to underline once more that all of the above rules only depend on transition triggers. Apart from this, other parts of the transition labels are irrelevant for the notion of refinement.

3.2.2 Rules for Composition

In the last section, we have presented a number of refinement rules to modify sequential automata that are already part of a specification under development. In the sequel, we go a step further and discuss a rule that provides the software engineer with a technique to add new charts to an existing specification by composition. Here, like for automata, also syntactic restrictions have to be required. As we regard hierarchy as special case of composition, this rule also serves as basis for the rule for hierarchical decomposition. Single components can be composed to more complex specifications using the following rule (see also Figure 3.9).

Theorem 3.2.7 (Refinement Through Composition)
Let $S_1, S_2 \in \mathcal{S}$ and L an arbitrary set of signals. If $Out(S_1) \cap Out(S_2) = \emptyset$ and $L \cap In(S_1) = \emptyset$ then $S_1 \rightsquigarrow S_1 \lhd L \rhd S_2$ □

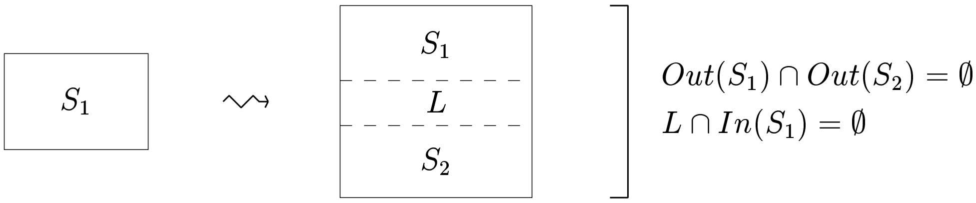


Figure 3.9: Rule for composition

Proof 12 We define $S =_{df} S_1 \lhd L \rhd S_2$. Let $i \in \mathcal{I}(S)^\infty$ and $o \in \mathcal{O}(S)^\infty$ such that $(i, o) \in [\![S_1 \lhd L \rhd S_2]\!]_{io}$. Then there exists $(c_1, c_2) \in Init(S_1) \times Init(S_2)$ such that $((c_1, c_2), i, o) \in [\![S_1 \lhd L \rhd S_2]\!]$. By definition, this is equivalent to

$$
\begin{aligned}
\exists o_1, o_2 . o_1 \in \mathcal{O}(S_1)^\infty \wedge o_2 \in \mathcal{O}(S_2)^\infty \wedge o = o_1 \Cup o_2 \wedge \\
(c_1, (i \Cup o|_L)|_{In(S_1)}, o_1) \in [\![S_1]\!] \wedge \\
(c_2, (i \Cup o|_L)|_{In(S_2)}, o_2) \in [\![S_2]\!]
\end{aligned}
$$

This is again equivalent to

$$
\begin{aligned}
\exists o_1, o_2 . o_1 \in \mathcal{O}(S_1)^\infty \wedge o_2 \in \mathcal{O}(S_2)^\infty \wedge o = o_1 \Cup o_2 \wedge \\
(c_1, i|_{In(S_1)} \Cup o|_{L \cap In(S_1)}, o_1) \in [\![S_1]\!] \wedge \\
(c_2, i|_{In(S_2)} \Cup o|_{L \cap In(S_2)}, o_2) \in [\![S_2]\!]
\end{aligned}
$$

which is, due to $In(S_1) \cap L = \emptyset$, equivalent to

$$
\begin{aligned}
\exists o_1, o_2 . o_1 \in \wp(Out(S_1))^\infty \wedge o_2 \in \wp(Out(S_2))^\infty \wedge o = o_1 \Cup o_2 \wedge \\
(c_1, i|_{In(S_1)}, o_1) \in [\![S_1]\!] \wedge \\
(c_2, i|_{In(S_2)} \Cup o|_{L \cap In(S_2)}, o_2) \in [\![S_2]\!]
\end{aligned}
$$

Since the output interfaces of S_1 and S_2 are disjoint, we finally get $(c_1, i|_{In(S_1)}, o|_{Out(S_1)} \in [\![S_1]\!]$ because $o_1 = o|_{Out(S_1)}$ and the proof is completed. □

Informally, this rule expresses that S_1 can be composed with an arbitrary specification S_2 whenever it can be guaranteed that S_1 cannot introduce additional behavior if composed with S_2. This condition can be ensured by two syntactic restrictions. The first requirement guarantees that S_1 cannot behave more non-deterministically through composition with S_2 than before. We exclude this by the restriction $In(S_1) \cap L = \emptyset$. At first glance, it might seem that this predicate is too strong and yet $In(S_1) \cap L \cap Out(S_2) = \emptyset$ would be a sufficient condition. However, recall that internal events sent via multicasting in one instant can stem from both S_2 and S_1. As a consequence, the latter would be too a weak requirement. Furthermore, the output interfaces of S_1 and S_2 have to be disjoint. If this condition was hurt, S_2 could chatter into the output stream of S_1 and one could not distinguish anymore whether events are generated by S_1 or S_2. Again, additional non-determinism would be introduced.

Since composition is commutative, we also get the similar rule: if $Out(S_1) \cap Out(S_2) = \emptyset$ and $L \cap In(S_2) = \emptyset$ then

$$S_2 \rightsquigarrow S_1 \lhd L \rhd S_2$$

The following proposition gives a refinement rule for pure parallel composition without any communication: if $Out(S_1) \cap Out(S_2) = \emptyset$ then

$$S_1 \rightsquigarrow S_1 \| S_2 \quad \text{and} \quad S_2 \rightsquigarrow S_1 \| S_2$$

The proof of this corollary is straightforward as $L = \emptyset$ is a special case for $In(S_2) \cap L = \emptyset$ and $In(S_1) \cap L = \emptyset$, respectively.

As a consequence, in our running example, we can start to specify the central locking system with the CONTROL component because the following are correct refinement steps:

$$\begin{array}{rcl} S_{LM} & \rightsquigarrow & S_{LM} \| S_{RM} \text{ because } Out(S_{LM}) \cap Out(S_{RM}) = \emptyset \\ S_{RM} & \rightsquigarrow & S_{LM} \| S_{RM} \text{ because } Out(S_{LM}) \cap Out(S_{RM}) = \emptyset \\ S_C & \rightsquigarrow & S_C \lhd L \rhd (S_{LM} \| S_{RM}) \text{ because } Out(S_C) \cap (Out(S_{LM}) \cup Out(S_{RM})) = \emptyset \\ & & \text{ and } In(S_C) \cap L = \{crash, ignition\} \cap L = \emptyset \end{array}$$

where $L = \{lup, rup, ldn, rdn, lmr, rmr, ready\}$ and S_{LM}, S_{RM}, and S_C denote the left and right motor, and the control part of the central locking system, respectively. However, note that we cannot omit the brackets around $Out(S_{LM}) \cup Out(S_{RM})$ as $rdn, rup, ready \in Out(S_C) \cap L \cap In(S_{RM})$ and $rmr \in In(S_C) \cap L \cap Out(S_{RM})$; recall that composition is not associative in general (see Section 2.4.4). Moreover, to start specifying the central locking systems with the motors and then refining it by composing them with the CONTROL is not possible as $S_C \lhd L \rhd (S_{LM} \| S_{RM})$ is no correct refinement of $S_{LM} \| S_{RM}$. The reason for this is that $(In(S_{LM}) \cup In(S_{RM})) \cap L = \{ldn, rdn, lup, rup, ready\} \cap L \neq \emptyset$.

If used in combination with composition, we can also give a rule for signal hiding: if $S_1 \rightsquigarrow S_1 \lhd L \rhd S_2$ and $K \cap Out(S_1) = \emptyset$ then

$$S_1 \rightsquigarrow [S_1 \lhd L \rhd S_2]_K$$

The proof for this rule follows straightforwardly from the definition of signal hiding.

For the central locking system, we thus get $S_C \rightsquigarrow [S_C \lhd L \rhd (S_{LM} \| S_{RM})]_{\{lmr, rmr\}}$. Note, however, that $[S_C \rightsquigarrow S_C \lhd L \rhd (S_{LM} \| S_{RM})]_{\{lmr, rmr, ready\}}$ is no correct refinement of S_C as $ready \in Out(S_C)$.

3.2.3 Rules for Hierarchy

One of the structuring mechanisms in Statecharts in a word is hierarchical decomposition. This technique is used to express the system behavior in more detail. The intuition of hierarchical decomposition is that it performs a similar behavior as the calling of subroutines or procedure calls, where we abstract from any parameter handling strategies. Thus, someone who specifies reactive systems with a Statecharts-like language would expect that the controller hands the control over to the controllee, which starts reacting beginning with the subsequent step. When the controllee terminates its activities, the control is withdrawn from it and the controller starts reacting again.

It is now challenging to ask whether hierarchical decomposition yields correct refinement. For more than a decade software engineers have trusted that this is true, but there did not exist any formal treatment of this question. Often, the notions "decomposition" and "refinement" have been used synonymously. In this section, we will show which restrictions are necessary in order to ensure that decomposition indeed is correct refinement.

In the last chapter, we have seen that hierarchical composition can be interpreted as parallel composition of controller and controllee plus some extra message passing from controller to controllee. It is therefore obvious that the syntactic refinement conditions for the rule for hierarchical decomposition are related to the rules for composition. From the propositions for composition we therefore can deduce the following rule. We define $A =_{df} (I, O, \Sigma, \Sigma_0, V_l, \varphi_0, \delta)$. Furthermore, let $A \rhd (\Sigma_d, \varrho)$ be defined as in Section 2.5.1. The interesting question now is under what circumstances we can guarantee that hierarchical decomposition is indeed a sound refinement step. We have to find easy to comprehend syntactic rules that a software engineer can apply. Formally, we are interested in finding restrictions like the ones for composition such that the following holds:

$$A \rightsquigarrow A \rhd (\Sigma_d, \varrho) \tag{3.4}$$

The syntactic restrictions that guarantee the soundness of this refinement rule depend on the interrupt mechanism. Recall that we distinguish between strong and weak preemption. In the case of strong preemption, it follows straightforwardly that (3.4) is a correct refinement step without any further restrictions. Strong preemption means that whenever the controller can react it immediately reacts within in the same step and also immediately withdraws the control from the controllee. As a consequence, the controllee has no possibility to fire its transitions whenever the controller does. Thus, the controllee does not interfere with the controller. Whenever the environment provides input that can trigger the controller, the controllee is inactive and the overall system behavior is equivalent to the one of a system that does not contain the controllee at all.

However, in the case of weak preemption the situation is somewhat more complex. Here, in instants in which the controller withdraws the control from the controllee, apart from the controller also the controllee potentially can fire one of its transitions. Thus, it still has the possibility to merge its output with the output of the controller in the instant when the control is handed over. Therefore, under weak preemptions the refinement characterized in (3.4) is only a correct refinement step if the following syntactic side conditions are fulfilled:

$$\forall \sigma \in \Sigma_d : Out(A) \cap Out(\varrho(\sigma)) = \emptyset \wedge In(A) \cap Out(\varrho(\sigma)) = \emptyset$$

Informally, this rule requires that the output interfaces of the controller and all its controllees are disjoint and that there is no message passing from controllee to controller (see Figure 3.10). The proof for this rule follows straightforwardly from the refinement rule for composition, the definition of hierarchy, and the fact that $\{go(\sigma)\} \cap In(A) = \emptyset$ as $go(\sigma)$ is a new signal that has not been used so far in the specification.

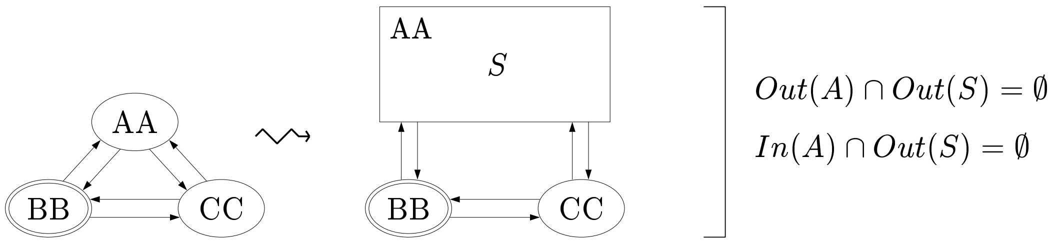


Figure 3.10: Rule for decomposition of automaton A

If we take a look at the central locking system, we ascertain that the automaton in Figure 2.5 is a correct refinement of Down and Up in LeftMotor and RightMotor, respectively. Figure 2.4, however, is no correct refinement of S_U and S_L because $ready \in Out(S_U) \cap In(S_C)$ and $ready \in Out(S_L) \cap In(S_C)$, where S_U, S_L, and S_C denote the charts for unlocking, locking, and the control, respectively. We see that the restriction $In(A) \cap Out(\varrho(\sigma)) = \emptyset$ is really needed to avoid additional non-determinism. In the example, the signal *ready* is fed back on the outermost level of hierarchy, which could initiate self-termination. Therefore, we can conclude that to introduce self-termination (see Section 2.5.2) never is a correct refinement step.

If used in combination with hierarchical decomposition, we can also give a rule for signal hiding: if $A \rightsquigarrow A \triangleright (\Sigma_d, \varrho)$ and $K \cap Out(A) = \emptyset$ then

$$A \rightsquigarrow [A \triangleright (\Sigma_d, \varrho)]_K$$

The proof for this rule follows straightforwardly from the definition of signal hiding.

3.3 Conclusion

In this chapter, we have presented a set of syntactic refinement rules for μ-Charts. It is guaranteed that each application of a rule only makes the overall specification more concrete, but never more abstract. Our notion of refinement is based on restriction of non-deterministic behavior. Thus, one μ-Chart specification is "more refined" than another, if and only if its set of input/output histories is a subset of the set of input/output histories of the abstract one. This notion of refinement is both compositional and transitive. We have proven the soundness of each refinement rule with respect to the stream semantics of Chapter 2. Though our calculus is not complete in a mathematical sense, it comprises all syntactic modifications that are important from a methodological point of view. Our calculus has several advantages. First, as all of our rules are purely syntactic, a system engineer does not have to carry out proofs over the semantics of μ-Charts, but just has to be aware of the rules themselves in order to get more concrete specifications. Second, testing whether a refinement step is feasible can be automated easily. Finally, a refinement calculus as the one we provide enables a system design process that is directed towards correctness by construction.

4 Formal Verification

There is a number of techniques for ensuring the correctness of reactive systems, such as simulation, testing, and formal verification. Simulation is a commonly used technique to analyze properties of reactive systems. STATEMATE, for instance, provides simulation techniques for Statecharts specifications. Simulation can by used to execute specifications, to analyze them, and to visualize system behaviors. However, to detect a high percentage (more than 95 percent) of system errors, exhaustive simulation or testing is necessary. Though this is, for example, in principle possible with the dynamic tools of STATEMATE, the complexity is very high without any methodological support or testing machinery and does not achieve a detection rate of 100 percent. What is rather needed is an automated formal verification technique like model checking.

Hence, a formal description language for reactive systems that does not provide any support for formal verification is of limited practical value. In this chapter, we motivate that μ-Charts are not only a theoretical toy language, but a realistic concept to which also model checking techniques can be applied to. For these purposes, we first define a relational semantics for our language, which is just the step-wise projection of the stream semantics introduced in Section 2.4.3. This semantics has been the basis for the definition of the refinement calculus. For model checking tools, however, which work with state variables and their values in the current and next instant, we have to formulate the μ-Charts behavior with respect to transition relations. These transition relations provide a systematic scheme for translating μ-Charts into input languages of various model checking tools. We prove that both semantics coincide.

We then show how to encode systematically this relational semantics in the symbolic μ-calculus verifier μcke and the symbolic model checker SMV. To demonstrate that our semantics is reasonable and that our translation to the two tools works properly, we have encoded the central locking system and proven some interesting safety critical system properties.

4.1 Transition Relations

In the sequel, we will formulate the transition relations for every single syntactic construct of the μ-Charts language by means of predicates. We have two different categories of predicates: one for initialization and one for the transition from one configuration to the next. These predicates have the type:

$$\begin{array}{rcl} Init_S & : & \mathcal{C}(S) \times \mathcal{I}(S) \times \mathcal{O}(S) \to Bool \\ Trans_S & : & \mathcal{C}(S) \times \mathcal{I}(S) \times \mathcal{C}(S) \times \mathcal{O}(S) \to Bool \end{array}$$

for every μ-Chart S. The predicate $Trans_S(c, x, c', y)$ is true whenever the configuration of the current instant of S is c and S can, stimulated by the input signal set x, reach the successor configuration c' in exactly one instant while producing the output signal set y.

Note that the initialization predicate $Init_S$ has three arguments. Besides the initial configuration we also have to fix which input and output signals can occur within the very first system instant. This is motivated by the following considerations: as we deal with instantaneous feedback, also in the very first step not only input but also output signals can be present. Thus, we have to denote which output events can be generated in combination with which input events. Otherwise, also those input/output signal combinations could occur in the initial instant for which no transition relation is defined. For these configurations no successor configurations are defined. As a consequence, we would get states in the resulting Kripke structure which do not have any outgoing transitions. These Kripke states represent states of the modeled reactive system where the system deadlocks. In order to avoid these unwanted deadlocks, we have to define which input/output signal combinations are possible in the first instant when the system starts running.

4.1.1 Sequential Automata

Initially, a sequential automaton $A =_{df} (I, O, \Sigma, \Sigma_0, V_l, \varphi_0, \delta)$ is in one of its default states $\sigma \in \Sigma_0$ and the local variables in V_l are initialized according to φ_0. For a set of input signals x coming from the environment, S generates a set of output signals y and changes its configuration, i.e. its current state from c to c':

$$\begin{array}{rcl} Init_A((\sigma, \varepsilon), x, y) & \equiv & \sigma \in \Sigma_0 \wedge \forall X \in V_l : \varepsilon(X) = \varphi_0(X) \wedge \\ & & \exists c'.(c', y) \in [\![\delta]\!](c, x) \\ Trans_A(c, x, c', y) & \equiv & (c', y) \in [\![\delta]\!](c, x) \vee [\![\delta]\!](c, x) = \emptyset \end{array}$$

Note that $Trans_A$ possibly contains chaotic behavior. Chaos occurs whenever the equation $[\![\delta]\!](c, x) = \emptyset$ is true as neither the successor configuration c' nor the output event y are fixed in this case.

4.1.2 Composition

The tuple (c_1, c_2) is the initial configuration of chart $S =_{df} S_1 \lhd L \rhd S_2$ whenever c_1 and c_2 are the initial configurations of charts S_1 and S_2, respectively. Let $c_1 \in \mathcal{C}(S_1)$,

$c_2 \in \mathcal{C}(S_2)$, $x \in \mathcal{I}(S)$, and $y \in \mathcal{O}(S)$ then

$$Init_S((c_1, c_2), x, y) \equiv Init_{S_1}(c_1, x|_{In(S_1)}, y|_{Out(S_1)}) \wedge Init_{S_2}(c_2, x|_{In(S_2)}, y|_{Out(S_2)})$$

The case distinction of three disjoint cases as in an earlier published paper [PS97a] is no longer needed. In [PS97a], we did not use a chaos semantics and therefore had to distinguish between three different cases in the definition of the transition relation for composition. Here, chaotic behavior is already included in the transition relation for sequential automata. Let $c_1 \in \mathcal{C}(S_1)$, $c_2 \in \mathcal{C}(S_2)$, $x \in \mathcal{I}(S)$, and $y \in \mathcal{O}(S)$ then

$$\begin{aligned} Trans_S((c_1, c_2), x, (c_1', c_2'), y) \quad \equiv \quad & \exists y_1, y_2. y = y_1 \cup y_2 \wedge \\ & Trans_{S_1}(c_1, (x \cup y|_L)|_{In(S_1)}, c_1', y_1) \wedge \\ & Trans_{S_2}(c_2, (x \cup y|_L)|_{In(S_2)}, c_2', y_2) \end{aligned}$$

4.1.3 Hiding

The definition of the initialization and transition predicates for hiding is straightforward. Let $S =_{df} [S']_K$ then we define:

$$\begin{aligned} Init_S(c, x, y) \quad &\equiv \quad \exists y'. Init_{S'}(c, x, y') \wedge y = y' \backslash K \\ Trans_S(c, x, c', y) \quad &\equiv \quad \exists y'. Trans_{S'}(c, x, c', y') \wedge y = y' \backslash K \end{aligned}$$

4.1.4 System Reactions

The above transition relations can be lifted to a stream semantics for all $S \in \mathcal{S}$. We will prove that this semantics coincides with the one presented in Section 2.4.3 if all sequential automata in S are responsive. The stream semantics $sr[\![S]\!] \in \wp(\mathcal{C}(S) \times \mathcal{I}(S)^\infty \times \mathcal{O}(S)^\infty)$ is characterized by the greatest fixed point of the following recursive equation ("sr" here stands for "system reaction"):

$$sr[\![S]\!] = \{(c, x\&i, y\&o) \mid \exists c'. Trans_S(c, x, c', y) \wedge (c', i, o) \in sr[\![S]\!]\}$$

Note that $\emptyset$ is in any case the least fixed point of this equation. Further note that this stream semantics is "more defined" than the semantics in Section 2.4.3. Here, S behaves chaotically within one step only. In the very next step, S can have a non-chaotic behavior again. This is in contrast to the semantics $[\![S]\!]$, where S also behaves chaotically in each subsequent step if it once started reacting chaotically. Due to this difference we will not be able to prove in general, that is, for arbitrary specifications S, that $[\![S]\!] = sr[\![S]\!]$ holds. This equivalence is merely true if all automata in S are responsive, i.e. the semantics of S does not include chaotic behavior. For arbitrary specifications that do not fulfill the property of responsiveness, it is possible to prove a weaker condition as demonstrated in the following propositions.

Proposition 4.1.1
Let A denote the sequential automaton $(I, O, \Sigma, \Sigma_0, V_l, \varphi_0, \delta)$. Then $sr[\![A]\!] \subseteq [\![A]\!]$ holds. If A is responsive, i.e. there are no $c \in \mathcal{C}(A)$ and $x \in \mathcal{I}(A)$ such that $[\![\delta]\!](c, x) = \emptyset$, then $sr[\![A]\!] = [\![A]\!]$.

Proof 13 As both semantics are defined by greatest fixed points, we first recall their definitions. Remember that $[\![A]\!]$ is defined as the greatest fixed point $\mathrm{gfp}(F)$ of the function F:

$$F =_{df} \lambda X.\{(c, x\&i, y\&o) \mid \exists c'.((c', y) \in [\![\delta]\!](c, x) \wedge (c', i, o) \in X) \vee [\![\delta]\!](c, x) = \emptyset\}$$

On the other hand, $sr[\![A]\!]$ is defined as the greatest fixed point $\mathrm{gfp}(G)$ of the function G:

$$G =_{df} \lambda X.\{(c, x\&i, y\&o) \mid \exists c'.Trans_A(c, x, c', y) \wedge (c', i, o) \in X\}$$

Remember that both functions differ in their treatment of chaos. We now have to prove $\mathrm{gfp}(G) \subseteq \mathrm{gfp}(F)$. In the sequel, we show the property $\forall X.G(X) \subseteq F(X)$ which immediately implies $\mathrm{gfp}(G) \subseteq \mathrm{gfp}(F)$. The property $\forall X.G(X) \subseteq F(X)$ is proven by showing that

$$\exists c'.Trans_A(c, x, c', y) \wedge (c', i, o) \in X$$

— which is equivalent to $\exists c'.((c', y) \in [\![\delta]\!](c, x) \vee [\![\delta]\!](c, x) = \emptyset) \wedge (c', i, o) \in X$ — implies

$$\exists c'.((c', y) \in [\![\delta]\!](c, x) \wedge (c', i, o) \in X) \vee [\![\delta]\!](c, x) = \emptyset$$

To carry out the rest of the proof, we abbreviate $(c', y) \in [\![\delta]\!](c, x)$ to $\alpha(c')$, $[\![\delta]\!](c, x) = \emptyset$ to β, and $(c', i, o) \in X$ to $\gamma(c')$. We transform the original predicate to:

$$\begin{aligned}
& \exists c'.(\alpha(c') \vee \beta) \wedge \gamma(c') \\
\Leftrightarrow\ & \exists c'.(\alpha(c') \wedge \gamma(c')) \vee (\beta \wedge \gamma(c')) \\
\Leftrightarrow\ & (\exists c'.\alpha(c') \wedge \gamma(c')) \vee (\exists c'.\beta \wedge \gamma(c')) \\
\Leftrightarrow\ & (\exists c'.\alpha(c') \wedge \gamma(c')) \vee (\beta \wedge (\exists c'.\gamma(c'))) \\
\Rightarrow\ & \exists c'(\alpha(c') \wedge \gamma(c')) \vee \beta
\end{aligned}$$

This concludes the first part of the proof. In the case that A is responsive, i.e. $[\![\delta]\!](c, x) = \emptyset$ is equal to false, we get the accordance $[\![A]\!] = sr[\![A]\!]$ of the two semantics as β equals false in this particular case. □

Proposition 4.1.2
Let S_1 and S_2 be two arbitrary charts. If $sr[\![S_i]\!] = [\![S_i]\!]$ with $i = 1, 2$ then for $S =_{df} S_1 \lhd L \rhd S_2$ holds $sr[\![S]\!] = [\![S]\!]$.

Proof 14 We prove this proposition by showing that both $sr[\![S]\!] \subseteq [\![S]\!]$ and $[\![S]\!] \subseteq sr[\![S]\!]$ hold. First, we show $sr[\![S]\!] \subseteq [\![S]\!]$. The semantics $sr[\![S]\!]$ is defined as the greatest fixed point $\mathrm{gfp}(H)$ of the following function H:

$$H =_{df} \lambda X.\{(c, x\&i, y\&o) \mid \exists c'.Trans_S(c, x, c', y) \wedge (c', i, o) \in X\}$$

Thus, to show $sr[\![S]\!] \subseteq [\![S]\!]$ is equivalent to showing $\text{gfp}(H) \subseteq [\![S]\!]$ or, as $\text{gfp}(H)$ is characterized by $\bigcup\{X \mid X \subseteq H(X)\}$, equivalently, $\forall X \subseteq H(X) : X \subseteq [\![S]\!]$. We discharge this proof obligation by showing that $\forall X \subseteq H(X) : H(X) \subseteq [\![S]\!]$. The latter is proven by induction over the length of input and output streams in X. To that end, we have to secure that this predicate is admissible. This is true because H is continuous with respect to the power domain, as we limit ourselves to a finite number of configurations, that is, $|\mathcal{C}(S)| < \infty$ for all charts $S \in \mathcal{S}$. The proof of the continuity of H then is similar to the proof of Theorem 2.4.1. Thus, we take arbitrary X with $X \subseteq H(X)$ and $((c_1, c_2), x\&i, y\&o) \in H(X)$, which is equivalent to

$$\exists c_1', c_2'.Trans_S((c_1, c_2), x, (c_1', c_2'), y) \wedge ((c_1', c_2'), i, o) \in X$$

This is per definition of $Trans_S$ equivalent to

$$\begin{aligned}\exists c_1', c_2'.(\exists y_1, y_2 \;.\; & Trans_{S_1}(c_1, (x \cup y|_L)|_{In(S_1)}, c_1', y_1) \wedge \\ & Trans_{S_2}(c_2, (x \cup y|_L)|_{In(S_2)}, c_2', y_2) \wedge \\ & y = y_1 \cup y_2) \wedge ((c_1', c_2'), i, o) \in X\end{aligned}$$

By induction assumption this implies

$$\begin{aligned}\exists c_1', c_2'.(\exists y_1, y_2 \;.\; & Trans_{S_1}(c_1, (x \cup y|_L)|_{In(S_1)}, c_1', y_1) \wedge \\ & Trans_{S_2}(c_2, (x \cup y|_L)|_{In(S_2)}, c_2', y_2) \wedge \\ & y = y_1 \cup y_2) \wedge ((c_1', c_2'), i, o) \in [\![S]\!]\end{aligned}$$

We unfold $((c_1', c_2'), i, o) \in [\![S]\!]$ applying the definition of $[\![S]\!]$ and get the equivalent characterization:

$$\begin{aligned}\exists c_1', c_2'.(\exists y_1, y_2 \;.\; & Trans_{S_1}(c_1, (x \cup y|_L)|_{In(S_1)}, c_1', y_1) \wedge \\ & Trans_{S_2}(c_2, (x \cup y|_L)|_{In(S_2)}, c_2', y_2) \wedge \\ & y = y_1 \cup y_2) \wedge \\ & \exists o_1, o_2.(o = o_1 \uplus o_2 \wedge \\ & (c_1', (i \uplus o|_L)|_{In(S_1)}, o_1) \in [\![S_1]\!] \wedge \\ & (c_2', (i \uplus o|_L)|_{In(S_2)}, o_2) \in [\![S_2]\!])\end{aligned}$$

The premise of this proposition tells us that $sr[\![S_i]\!] = [\![S_i]\!]$. If we additionally apply the recursive definition of the $sr[\![S_i]\!]$, the above predicate is equivalent to

$$\begin{aligned}\exists y_1, y_2, o_1, o_2 \;.\; & y = y_1 \cup y_2 \wedge \\ & o = o_1 \uplus o_2 \wedge \\ & (c_1, ((x \cup y|_L)\&(i \uplus o|_L))|_{In(S_1)}, y_1\&o_1) \in sr[\![S_1]\!] \wedge \\ & (c_2, ((x \cup y|_L)\&(i \uplus o|_L))|_{In(S_2)}, y_2\&o_2) \in sr[\![S_2]\!]\end{aligned}$$

Note that care has to be taken when moving the existential quantifier that binds c_1' and c_2' through the conjunction. However, in this case it is no problem at all, as the c_i'

occur only in the part of the formula that is affected by the application of the recursive formula of $sr[\![S_i]\!]$. The above formula is per definition of the step-wise union operator $\Cup$ on streams equivalent to

$$\begin{aligned}\exists y_1, y_2, o_1, o_2 \,.\, & y = y_1 \cup y_2 \wedge \\ & o = o_1 \Cup o_2 \wedge \\ & (c_1, ((x\&i) \Cup (y\&o)|_L)|_{In(S_1)}, y_1\&o_1) \in sr[\![S_1]\!] \wedge \\ & (c_2, ((x\&i) \Cup (y\&o)|_L)|_{In(S_2)}, y_2\&o_2) \in sr[\![S_2]\!]\end{aligned}$$

Furthermore, we use $sr[\![S_i]\!] = [\![S_i]\!]$ and transform the above predicate equivalently, using the definition of $[\![S]\!]$, to $((c_1, c_2), x\&i, y\&i) \in [\![S]\!]$, which concludes the "$\subseteq$" direction of the proof.

Now, we show $[\![S]\!] \subseteq sr[\![S]\!]$. It is sufficient to prove $[\![S]\!] \subseteq H([\![S]\!])$. Let $((c_1, c_2), x\&i, y\&o) \in [\![S]\!]$. Unfolding the definition of $[\![S]\!]$ and using the equalities $[\![S_i]\!] = sr[\![S_i]\!]$, we have:

$$\begin{aligned}\exists y_1, y_2, o_1, o_2 \,.\, & y = y_1 \cup y_2 \wedge \\ & o = o_1 \Cup o_2 \wedge \\ & (c_1, ((x\&i) \Cup (y\&o)|_L)|_{In(S_1)}, y_1\&o_1) \in sr[\![S_1]\!] \wedge \\ & (c_2, ((x\&i) \Cup (y\&o)|_L)|_{In(S_2)}, y_2\&o_2) \in sr[\![S_2]\!]\end{aligned}$$

which is due to the recursive definition of $sr[\![S_i]\!]$ equivalent to

$$\begin{aligned}\exists y_1, y_2, o_1, o_2, c_1', c_2' \,.\, & y = y_1 \cup y_2 \wedge \\ & o = o_1 \Cup o_2 \wedge \\ & Trans_{S_1}(c_1, (x \cup y|_L)|_{In(S_1)}, c_1', y_1) \wedge (c_1', (i \Cup o|_L)|_{In(S_1)}, o_1) \in sr[\![S_1]\!] \wedge \\ & Trans_{S_2}(c_2, (x \cup y|_L)|_{In(S_1)}, c_2', y_2) \wedge (c_2', (i \Cup o|_L)|_{In(S_2)}, o_2) \in sr[\![S_2]\!]\end{aligned}$$

Using the equality $sr[\![S_i]\!] = [\![S_i]\!]$ once more and applying the definition of $[\![S]\!]$ we get:

$$\begin{aligned}\exists y_1, y_2, c_1', c_2' \,.\, & y = y_1 \cup y_2 \wedge \\ & Trans_{S_1}(c_1, (x \cup y|_L)|_{In(S_1)}, c_1', y_1) \wedge \\ & Trans_{S_2}(c_2, (x \cup y|_L)|_{In(S_2)}, c_2', y_2) \wedge ((c_1', c_2'), i, o) \in [\![S]\!]\end{aligned}$$

Finally, we apply the definition of $Trans_S$ and have:

$$\exists c_1', c_2'.Trans_S((c_1, c_2), x, (c_1', c_2'), y) \wedge ((c_1', c_2'), i, o) \in [\![S]\!]$$

which is just the definition of $H([\![S]\!])$. □

To prove that the two semantics for hiding coincide, that is, $sr[\![S]\!] \subseteq [\![S]\!]$ implies $sr[\![[S]_K]\!] \subseteq [\![[S]_K]\!]$ and $sr[\![S]\!] = [\![S]\!]$ implies $sr[\![[S]_K]\!] = [\![[S]_K]\!]$ for an arbitrary signal set K follows directly from the definitions of $[\![S]\!]$ and $sr[\![S]\!]$.

As we have seen in the propositions above, the two semantics $[\![.]\!]$ and $sr[\![.]\!]$ are equivalent for responsive specifications. Furthermore, as our semantic interpretation of non-responsiveness is chaos, we would have to encode this chaos explicitly in the input language of the model checker if we admitted non-responsive specifications for model checking (see also the discussion in Section 1.1). Therefore, the remainder of this thesis is restricted to responsive μ-Chart specifications.

Now, we have the formal justification that the two semantics we use to express the behavior of reactive systems coincide. This is a prerequisite to guarantee that a system that has been incrementally developed using the refinement calculus is equivalent to the system that is formally verified. The following section applies the systematic translation scheme to two specific model checking tools and shows how to establish model checking for μ-Charts.

4.2 Model Checking

In the previous section, we have defined the semantics of μ-Charts in terms of the relational μ-calculus to provide a mathematical basis for formal verification by model checking. Before explaining model checking for μ-Charts in more detail, we give a brief introduction to the basic ideas of this technology.

Having defined the behavior of a reactive system with μ-Charts, we are interested in finding out whether our specification fulfills important system properties. For the central locking system, such properties are, for example, that the doors will always eventually be locked when the driver wants to lock his or her car from outside by a key. An even more vital property to the driver is that the locking system always unlocks all car doors automatically in the case of an accident. To formally verify these or other conditions by model checking, we have to use an appropriate language. Temporal logics are convenient formalisms for specifying such requirements of systems with a complex temporal behavior [Sch97]. There are a lot of different versions temporal logics which differ in both their semantics and their expressiveness (see [Eme90] for a good overview). A further difference is the modeling of time: for discrete systems discrete points of time are considered, but for hybrid systems, for instance, temporal logics with a continuous model of time are needed [Sta97]. The most frequently used discrete temporal logics are the branching time logic CTL [CE81], the linear time logic LTL [Pnu77], and their superset CTL^* [EH86]. To model check a system property φ of a specification S, we have to have a *model* of our specification S. A model $M(S)$ of the just mentioned temporal logics is often given as a certain class of finite state transitions diagrams, called Kripke structures [Eme90]. Model checking now is nothing more than proving that φ holds in $M(S)$ or, more formally, $M(S) \models \varphi$. If we use BDD [Bry86] based techniques for model checking we speak of symbolic model checking [McM93].

For model checking, a growing number of tools, sometimes commercial, but often public domain, exists that either work with symbolic [McM93] or explicit encodings [GW94] for both models and properties. In [BCM+90], symbolic model checking with BDDs based on the μ-calculus has been proposed as a general framework for various verification problems like verification of LTL and CTL formulae and proofs for bisimulation equivalence and language containment. By the aid of a μ-calculus model checker like μcke, for instance, all these verification tasks can be included within one single tool [Bie97b]. In addition, there exist some applications, like the semantics we presented in [PS97a], that need the expressiveness of the μ-calculus [PS97b].

To demonstrate that μ-Charts are not only an elegant theoretical framework, but also a realistic software engineering language for the design of reactive systems, we have implemented our semantics within two different model checking tools: μcke [Bie97a] and SMV [McM93]. We have chosen μcke since it is based on the relational μ-calculus and we therefore get, up to some fine tuning, a nearly one-to-one encoding of the initial and transition relations of Section 4.1. Also in [Lev97], it has been demonstrated that the propositional μ-calculus is appropriate for the verification of temporal and real-time properties of Statecharts. There, a *compositional proof system* has been developed for deciding whether a Statechart, whose semantics is defined by a process language that is very similar to the one of [PS91], satisfies a μ-calculus formula.

However, μcke was developed within the scope of a single PhD thesis and therefore continual support is not guaranteed. Another disadvantage of μcke is that the generated counter examples and witnesses are hard to grasp. Moreover, its verification runtime is, for our μ-Charts examples, a bit slower than, for instance, the one of the SMV model checker.

This slow-down is not surprising since many successful applications and case studies used model checkers with less expressive languages than the μ-calculus [Bie97b]; for special purpose languages optimized verification tools can be easily built [BCM+90]. The model checker for Unity [Kal96], a programming theory that combines a simple, but yet expressive temporal logic with a programming notation that is suited for the formal specification, design, and analysis of concurrent programs is just one of many examples.

In the first place, SMV was designed to model check hardware designs. Hardware, consisting of combinational logic and registers, has, similar to μ-Charts, also a synchronous semantics. Therefore, it is near at hand to use a tool that already makes use of the synchrony concept. In particular, SMV's module concept coincides with the instantaneous, synchronous composition operator of μ-Charts. On the other hand, for languages with interleaving semantics like I/O automata [Mül98], for instance, it is more appropriate to use tools that also implement an interleaving semantics, like Spin [GW94].

Furthermore, the μ-calculus is not very comprehensible and therefore not suited for direct use by software engineers. As a consequence, we decided to provide a translation scheme for a second model checker and have chosen SMV. In contrast to μcke, SMV uses fixed BDD variable orderings for current and next state variables and specialized algorithms for the computation of the states that are reachable within one transition step [Bie97b]. In the sequel, we discuss both encodings.

4.2.1 Short Introduction to Mucke

Recently, several verification tools for the μ-calculus have been developed [CR94, Bie97b]. In this thesis, we have decided to use μcke [Bie97b]. In the following, we show how a specification in μ-Charts can be encoded in μcke, and how the encoding can be used for property verification. We start with a brief introduction to μcke that is a prerequisite for a better understanding of the encoding.

As we already have noted, the μ-calculus is not very comprehensible. Therefore, it should not be used directly by software engineers. Rather, a front end that translates a specific formal specification into the μ-calculus should be aspired [Bie97b]. In the next section, we will give such a translation scheme for μ-Charts. To ease automatic code generation of μcke code, its input language is similar to the programming language C. It allows the definition of μ-calculus formulae, compiler hints like fixing of variable orderings, and the declaration of variables.

The supported datatypes include Booleans, enumerations, bounded intervals of the natural numbers, vectors, and algebraic products that are defined with a construct "`class`" that is similar to "`struct`" in C. Currently, μcke does not support non-Boolean functions or types corresponding to the algebraic sum ("`union`" in C). Notice that though allowing in principle arbitrary values in $\mathbb{Z}$ for μ-Chart specifications, we have to restrict ourselves to finite intervals of $\mathbb{Z}$ in order to get descriptions that can be model checked.

Relations are defined as Boolean-valued functions in μcke. In the μ-calculus, the least and greatest fixpoint operators are used to recursively define relations. As in C, μcke permits recursion by writing the relation's name at the right-hand side of the relation's definition. In addition, the keywords `mu` or `nu` have to precede the definition to indicate whether the least or greatest fixpoint is intended. In μcke (and also in SMV) syntax, `&`, `|`, `!`, `->`, and `<->` denote Boolean conjunction, disjunction, negation, implication, and equivalence, respectively.

Besides these language constructs there are also other syntactic elements that do not belong to the μ-calculus, but can be considered as additional annotations. One of those are hints for variable orderings. Internally, μcke uses BDD [Bry86] algorithms to evaluate formulas. The variable ordering for the BDDs is computed automatically. However, in many cases it is impossible to generate good orderings automatically, but μcke here offers a good remedy. It is possible to give "hints" in the input files about the ordering and interleaving of variables. There exist, for example, commands to arrange variables in sequential (`x < y`) or interleaved (`x ~+ y`) ordering. A detailed treatment of variable orderings can be found in [Bie97a].

Further annotations that are supported are commands to dump the number of variable assignments that fulfill a certain Boolean function or the size of the BDD that represents a function. Moreover, counter examples and witnesses can be generated and there are still further possibilities, which are not listed here.

4.2.2 Encoding in Mucke

In this section, we illustrate a translation scheme for μ-Charts to μcke. To this end, we directly use the semantics as discussed above. The translation is performed top-down following the structure of μ-Charts. For each construct a configuration datatype, an initialization predicate, and a transition relation is defined.

Signals

For both the external and internal signals that occur in the specification, a single type is defined. While two separate types might seem more natural, the single type helps with the variable ordering used by μcke, and makes the internal representation more efficient. Whenever a signal s is present, the corresponding Boolean variable v_s equals true, and false otherwise. Each signal that occurs in a specification must be declared in μcke as shown by the central locking system example:

```
class Signals {
/* Internal and external signals: */
   bool crash;     bool open;    bool close;   bool ignition;
   bool lmr;       bool rmr;     bool ldn;     bool rdn;
   bool lup;       bool rup;     bool ready;

/* Master/slave interaction: */
   bool go_normal;
   bool go_lockg;     bool go_unlg;
   bool go_left_down; bool go_left_up; bool go_right_down;
                      bool go_right_up;
};
```

Sequential Automata

For each sequential automaton, two state types are introduced. One, the vertex, is an enumeration type that represents the control state. The second type is a product type for the chart's complete configuration. It contains, if necessary, the data state in addition to the control state. Since the locking system is specified without data states, this type consists of the vertex only. The encoding of the CONTROL's states looks like this:

```
enum ControlVertex { Normal, Crash };
class ControlState { ControlVertex c; };
```

In addition, for every automaton two predicates are defined: the first predicate characterizes the initial configuration of the automaton. Initial configurations restrict the control state to the default state of the automaton and also restrict the data components, if used.

```
bool ControlInit(ControlState s, Signals i, Signals o)
(
  s.c = Normal & (( i.crash & o.go_normal &  o.lup &  o.rup) |
                  (!i.crash & o.go_normal & !o.lup & !o.rup))
);
```

The Boolean function `ControlInit` determines that NORMAL is the initial control state of CONTROL. Particular attention is directed to the second part of its conjunction: it is needed to reduce the number of initial variable assignments. Since we deal with instantaneous feedback, also in the very first instant, i.e. when the system starts to react, input and output signals can be present. However, not all possible signal combinations are feasible. Hence, we have to restrict the initial state to input/output signal relationships that are intended by the specification. Taking a look at our running example, we notice that a feasible initial signal combination implies that either *crash*, *lup*, and *rup* are present, or none of them. To the action parts of all outgoing transitions of state NORMAL signals for master/slave interaction have to be added. Therefore, also the Boolean variable `o.go_normal`, which is sent from master to slave in order to model hierarchical decomposition (see Section 2.5.1) has to be considered.

The second predicate defines the transition relation over the current configuration s, the successor configuration t, and input and output signal sets i and o. The relation is a disjunction of the individual transition conditions, together with a condition that restricts the automaton's outputs. The latter is necessary as automaton A should not be able to produce any output signals other than those specified in its output interface $\mathcal{O}(A)$.

```
bool ControlTrans(ControlState s, Signals i, Signals o, ControlState t)
s ~+ t, i ~+ o, i < s
(
  (  s.c=Normal &  i.crash &  o.go_normal &  o.lup &  o.rup & t.c=Crash
  |  s.c=Normal & !i.crash &  o.go_normal & !o.lup & !o.rup & t.c=Normal

  |  s.c=Crash  &            !o.go_normal &  o.lup &  o.rup & t.c=Crash
  )
  & !o.crash & !o.close & !o.open
  & !o.lmr & !o.rmr & !o.ldn & !o.rdn & !o.ready
  & !o.go_lockg & !o.go_unlg & !o.go_left_down
  & !o.go_left_up & !o.go_right_down & !o.go_right_up
);
```

Note the three expressions after the transition relation's header: these are variable ordering hints for μcke. They indicate that the variables for current and successor configuration, as well as the input and output signal variables should be interleaved. Moreover, the input (and thus also the output) variables are strictly ordered before the configurations. Whenever nothing is said about certain variables the checker is free to use its heuristics to compute their position in the BDD for the corresponding transition relation.

Composition and Hiding

The configuration of a μ-Chart S that consists of $n \in \mathbb{N}$ sequential automata is defined as algebraic product of the configurations of the sub-charts. For CONTROL this is demonstrated in the sequel:

```
class ControlAllState {
   ControlState conf_control;
      NormalState conf_normal;
         LockgState conf_lockg;
         UnlgState conf_unlg;
};
```

As we have seen in Section 4.1, the initial predicate $Init_S$ for the chart S is simply the logical conjunction of the initial predicates of all its sub-charts. Hence, the initialization of CONTROL reads as follows:

```
bool ControlAllInit(ControlAllState s, Signals i, Signals o)
(
    ControlInit(s.conf_control,i,o)
  & NormalInit(s.conf_normal,i,o)
  & LockgInit(s.conf_lockg,i,o)
  & UnlgInit(s.conf_unlg,i,o)
);
```

The encoding of $Trans_S$ in μcke is similar. As top level transition relation encoded in μcke we get:

```
bool ControlAllTrans(ControlAllState s, Signals i, Signals o,
                     ControlAllState t)
s ~+ t, i ~+ o, i < s
(
  exists Signals i1, Signals i2, Signals i3, Signals i4,
         Signals o1, Signals o2, Signals o3, Signals o4 .

    (  ControlTrans(s.conf_control,i1,o1,s.conf_control)
    &  NormalTrans(s.conf_normal,i2,o2,s.conf_normal)
    &  LockgTrans(s.conf_lockg,i3,o3,s.conf_lockg)
    &  UnlgTrans(s.conf_unlg,i4,o4,s.conf_unlg)
```

```
      & (i1.crash <-> i.crash)          /* I(Control) */

      & (i2.ready <-> o.ready)          /* I(Normal) */
      & (i2.close <-> i.close)
      & (i2.open <-> i.open)
      & (i2.go_normal <-> o.go_normal)

      & (i3.lmr <-> i.lmr)              /* I(Lockg) */
      & (i3.rmr <-> i.rmr)
      & (i3.go_lockg <-> o.go_lockg)

      & (i4.lmr <-> i.lmr)              /* I(Unlg) */
      & (i4.rmr <-> i.rmr)
      & (i4.go_unlg <-> o.go_unlg)
      & (o.crash <-> (o1.crash | o2.crash | o3.crash | o4.crash))
      & (o.close <-> (o1.close | o2.close | o3.close | o4.close))
        ...
      & (o.go_right_up <-> (o1.go_right_up | o2.go_right_up |
                            o3.go_right_up | o4.go_right_up))
    ) {alloccs i ~+ i1, i ~+ i2, i ~+ i3, i ~+ i4,
               o ~+ o1, o ~+ o2, o ~+ o3, o ~+ o4}
  );
```

Doing Proofs

The μ-calculus has the nice feature that system and requirements specification can be written in the same language. Though this is on the one hand very comfortable and convenient to compare or combine orthogonal system views, it has some drawbacks on the other hand.

Simple properties such as reachability can be straightforwardly expressed in the μ-calculus; others are usually easier formulated in the more specialized and compact and therefore also more readable temporal logics, such as CTL, CTL^*, or LTL. Formulae in these logics can be schematically translated into the μ-calculus. A translation of the branching-time logic CTL into the μ-calculus is described in [McM93]. Translations for CTL^* and LTL can be found in [Dam94].

In the following, we present some vital properties of the central locking system. We will justify the fact that formal verification is absolutely necessary to guarantee that critical properties of a reactive system are really fulfilled by the specification. We will see that some properties which the reader might think are valid for our running example are in fact not true. All system requirements that are formulated in the sequel have been checked with μcke and SMV. For both verifiers this took only a couple of seconds, where SMV found the proofs even faster than μcke. The experimental results are summarized in Table 4.1: here, the number of BDD nodes representing the top level transition relation and the runtimes are given. SMV used 0.73 seconds user time and merely 0.06

seconds system time for both building the BDDs for the top level transition relation and checking all system properties discussed in this section. The tool μcke allows a more detailed calculation of the used resources: it required 6.04 seconds to build the BDDs and 3.24 seconds to perform the subsequent model checking. All results have been computed on a Sparc Ultra 2 Model 2200 from Sun with two 200 MHz processors and 512 MB main memory, 530 MB virtual memory, and Sun OS 5.5.1.

Criterion	μcke	SMV
BDD nodes [integer]	1405	874
Runtime [seconds]	6.04 + 3.24	0.73 (0.06)

Table 4.1: Experimental results (μcke and SMV)

We start with a simple predicate to check whether a given configuration is reachable. Reachability can be defined as a least fixpoint with the following formula, where `TopState`, `TopInit`, and `TopTrans` denote the configuration (data and control state) of the entire system, its initial, and its transition relations:

```
mu bool Reach(TopState s)
        (exists Signals i, Signals o . TopInit(s,i,o)) |
        (exists TopState r, Signals ii, Signals oo.
                Reach(r) & TopTrans(r,ii,oo,s));
```

The μcke encoding can also be used to check a specification for determinism. In μcke, this condition is formulated as follows:

```
forall Signals i, Signals o1, Signals o2,
       TopState s, TopState t1, TopState t2 .
       (Reach(s) & TopTrans(s,i,o1,t1) & TopTrans(s,i,o2,t2)) ->
       (o1 = o2 & t1 = t2);
```

As the reader may already have observed, the original version of the locking system is not deterministic. The reason for this is found quickly: for instance, each of the automata MotorLeft, MotorRight, Down, or Up contains non-deterministic transitions.

In the formula above for checking determinism, we only considered reachable configurations. Thus, our condition is stronger and more precise than syntactic determinism checks. On the other hand, the computation of reachable states can be computationally expensive for larger specifications, especially when local data states are used.

Next, we will prove the most critical property of the locking system, that is, whenever the signal *crash* is present in one instant, eventually both doors will be unlocked. This has to be true for all possible system behaviors. This property can be formalized in CTL (AG and AF stand for "always globally" and "always eventually", respectively):

$$AG(crash \Rightarrow AF(\mathit{LeftDoorUnlocked} \wedge \mathit{RightDoorUnlocked}))$$

where *LeftDoorUnlocked* and *RightDoorUnlocked* are Boolean predicates that encode system configurations where the left and right door are unlocked, respectively. In μcke, this CTL formula reads as follows:

```
mu bool AF_Unlocked(TopState s)
  (s.conf_controlall.conf_unlg.c = Both) |
  (forall TopState t, Signals i, Signals o . TopTrans(s,i,o,t) ->
                                             AF_Unlocked(t));

nu bool AGAF_Crash(TopState s)
  (forall Signals i . i.crash ->
          (s.conf_motorall.conf_motorleft.conf_motorupleft.c = Stop |
          (forall Signals o, TopState t . TopTrans(s,i,o,t) ->
                                          AF_Unlocked(t)))) &
  (forall Signals ii, Signals oo, TopState tt .
          TopTrans(s,ii,oo,tt) -> AGAF_Crash(tt));

Forall TopState s, Signals i, Signals o . (TopInit(s,i,o) ->
                                           AGAF_Crash(s));
```

For the central locking system μcke indeed could prove that this requirement is fulfilled. Another interesting question is whether the doors eventually are unlocked in a less critical situation, i.e. when the driver simply turns the key to enter the car. This requirement reads similarly:

```
mu bool AF_Unlocked'(TopState s)
 (s.conf_controlall.conf_normal.c = Unlocked) |
 (forall TopState t, Signals i, Signals o . TopTrans(s,i,o,t) ->
                                            AF_Unlocked'(t));


nu bool AGAF_Unlocked(TopState s)
   (forall Signals i . i.open ->
           (s.conf_controlall.conf_normal.c = Unlocked |
           (forall Signals o, TopState t . TopTrans(s,i,o,t) ->
                                           AF_Unlocked'(t)))) &
   (forall Signals ii, Signals oo, TopState tt .
           TopTrans(s,ii,oo,tt) -> AGAF_Unlocked(tt));

Forall TopState s, Signals i, Signals o . (TopInit(s,i,o) ->
                                           AGAF_Unlocked(s));
```

However, this property is not true. Taking a look at Figure 2.3 again, we see the reason for that: the doors can be unlocked only when they have been locked previously. Strengthening the above implication premise `i.open` in `AGAF_Unlocked` to

```
i.open & s.conf_controlall.conf_normal.c = Locked
```

we get a true predicate. This shows us that though the specification of our locking system almost fits on one single page, already for simple properties we cannot be sure whether they are valid or not as long as we do not verify them formally. Hence, we see again how important it is to have a precise formal semantics for a description technique.

Even more interesting is the following case: we might expect that a similar statement as for unlocking the car can be made for closing it. Thus, whenever the signal *close* is present, initiated, say, by pressing a button inside the car, we would suppose that in any case eventually the doors will be locked. This is, for instance, an important feature to prevent car napping when waiting in front of traffic lights. Substituting in the above two formulae for unlocking each occurrence of `Unlocked` by `Locked` and vice versa, we get a μ-calculus formalization of this attribute.

However, this property is false, i.e. there may be situations where pressing the locking button does not in any case yield the desired action. However, the car passengers might be quite grateful that this is the case. The mistake we made was to forget the CRASH functionality: while the motors are still closing the doors, there might be an accident that initiates the crash sensor. As a consequence, the closing process has to be interrupted and the doors have to be unlocked. Again, we notice how important formal verification techniques are to understand a specification and all its implications for safety critical situations.

4.2.3 Short Introduction to SMV

As SMV is in widespread use and better known as μcke, we only give a very scarce introduction. The SMV system is a tool for symbolic model checking of finite state systems against specifications in CTL [McM93]. Hence, in contrast to μcke, SMV makes two different languages available for the system and requirement specification. On the one hand, requirements are defined in CTL. On the other hand, systems are specified in a specialized language, which fits for describing concurrent finite state systems and protocols.

This language resembles the synchronous data flow language LUSTRE [HCRP91, HFB93, Hal93a, Hal93b, BCLH93] to some extent. SMV provides two different styles to specify state transition systems. In the parallel assignment style, a program can be viewed as a system of equations whose solutions determine the system's next state. SMV still offers a second description style that allows to specify transition relations directly as propositional formulae in terms of the current and the next values of the state variables.

This is, however, not considered to be a good means to specify systems by hand because it can seduce the user to formulate inconsistent programs through logical absurdities. Rather, this more flexible mechanism may be useful to write front-end translators from other languages to SMV. As a consequence, we also use this technique.

Apart from the similarity with LUSTRE, SMV also provides further concepts, like non-determinism, that are not included in LUSTRE. It allows modular descriptions and therefore the definition of re-usable components. In μ-Charts, we can make extensive use of this feature since many sub-charts, like the charts for LOCKG, UNLG, DOWN, and UP of the locking system, have a generic nature. The main module has a similar meaning as in the programming language C. Finite scalar types are the only basic data types. Structured data types are also at our disposal. In contrast to μcke, SMV provides no user interface for adjusting BDD variable orderings. Likewise for μcke, we have to restrict ourselves to finite intervals of $\mathbb{Z}$ in order to get descriptions that can be model checked.

4.2.4 Encoding in SMV

To carry out the encoding of our semantics in SMV, we follow in principle the same steps as for μcke. Signals are encoded as Boolean variables. In contrast to μcke, we do not have to define a globally accessible class for signals. Rather, we include each input and output signal of a specific automaton as extra formal parameter in the module that implements this automaton.

Also different from μcke, the predicates $Init_S$ and $Trans_S$ of an automaton S are collected in one single module and not as separate functions. However, up to variable renamings that are enforced by the tool, these predicates are identical in SMV and μcke.

```
MODULE control(c,l,r,go)
                            -- c: input signal
VAR
    state: {Normal,Crash};  -- local variable
    l:     boolean;         -- output signal
    r:     boolean;         -- output signal
    go:    boolean;         -- output signal

INIT
    state=Normal & (( c & go &  l &  r) |
                     (!c & go & !l & !r))

TRANS
    state=Normal &  c &  go &  l &  r & next(state)=Crash
  | state=Normal & !c &  go & !l & !r & next(state)=Normal

  | state=Crash  &      !go &  l &  r & next(state)=Crash
```

Notice that, different from the μcke encoding, the explicit negation of all those signal variables which are not in the output interface of the corresponding automaton is not necessary. This is justified by the fact that not the entire signal class is used here as input parameter of an automaton's module, but only those from the automaton's input and output interfaces.

For every single automaton a module similar to the one pictured above is constructed. Finally, these modules are combined in the main module as described in the sequel:

```
MODULE main

VAR
  crash:    boolean;
  close:    boolean;
  open:     boolean;
  ignition: boolean;

  SC:       control(crash,lup_sc,rup_sc,go_normal);
    SCN:    normal(go_normal,close,open,ready,ldn,rdn,
                   lup_scn,rup_scn,go_lockg,go_unlg);
      SCNL: unlg_lockg(go_lockg,lmr,rmr,ready_scnl);
      SCNU: unlg_lockg(go_unlg,lmr,rmr,ready_scnu);

  SL:    motor(lup,ldn,ready,go_left_down,go_left_up);
    SLD: down_up(go_left_down,lmr_sld);
    SLU: down_up(go_left_up,lmr_slu);

  SR:    motor(rup,rdn,ready,go_right_down,go_right_up);
    SRD: down_up(go_right_down,rmr_srd);
    SRU: down_up(go_right_up,rmr_sru);

DEFINE
  ready := ready_scnl | ready_scnu;
  lmr   := lmr_slu | lmr_sld;
  rmr   := rmr_sru | rmr_srd;
  lup   := lup_sc | lup_scn;
  rup   := rup_sc |rup_scn;
```

Here, the three external input signals, *crash*, *close*, *open*, and *ignition* are declared and the automata modules are instanciated; notice that we repeatedly use the modules `unlg_lockg`, `motor`, and `down_up`. Thus, syntactically, the μcke encoding is more than thrice as long as the one of SMV.

The `DEFINE` part is needed for the formalization of multicasting. SMV uses a much more specialized language than the μ-calculus in μcke: the module concept of SMV only provides a channel-oriented communication mechanism. Since messages in μ-Charts are multicast, we have to carry out a workaround. Remember that one of the basic ideas of SMV is to consider a program as a set of equations. This fact implies the following restriction: though a signal, represented by a Boolean variable, can be used as current input parameter of more than one module, only at least one module can have a signal as current output parameter. In the running example, *ready*, *lmr*, *rmr*, *lup*, and *rup* are of this kind.

4.3 Conclusion

In this chapter, we have shown a scheme for the systematic translation of μ-Chart specifications to arbitrary model checking tools. This translation scheme requires the definition of μ-Charts' semantics by transition relations. We have proven that the relational step semantics developed in this chapter and the stream semantics of Chapter 2 coincide for exactly those specifications, where at least one possible reaction for each input signal and each control state of every sequential automaton has been defined by the user. This restriction is important in order to reduce the number of transitions of the final Kripke structure and therefore to guarantee efficient model checking. Apart from the principal translation scheme for arbitrary model checkers by giving the semantics in form of transition relations, we also have shown exemplarily how specifications in μ-Charts can be translated automatically to the input languages of two automatic proof systems, the more general verifier μcke that is based on the relational μ-calculus and the model checker SMV that is primarily used for verification of hardware and clock synchronous languages.

5 Partitioning

Up to now, our method for design with Statecharts covers the system development phases system description, refinement, and formal verification. What is still missing are guidelines how to implement a design on one processor or even on a processor network. Aiming at partitioning a specification and later on implementation on a distributed target architecture, we are faced with many theoretical and practical problems. In principle, two general questions are of interest. The first and central requirement is: is the behavior described by the specification unaffected by this partitioning? In addition, we are interested in knowing whether the real-time behavior of the distributed implementation is different from the centralized implementation. Thus, the second question is: does the distribution slow down reaction times or is the reaction accelerated?

This chapter addresses the problem of partitioning and gives an answer to the first question. It is organized as follows: first, in Section 5.1, we show how the formal semantics of Section 2.4.3 has to be interpreted in order to understand the semantics of composition as, possibly distributed, fixed point construction.

Second, in Section 5.2, we discuss problems that arise when a Statecharts design shall be distributed. These problems can be caused by external stimuli from the system environment, internal signals that are generated by distributed fixed point construction, or scanning of signal absences. In Section 5.3, we present a model to formally express the design's allocation on a processor network. Since this model is based on integer linear programming, it also enables the designer to compute optimal solutions by tool assistance of the allocation problem with respect to an objective function if the solution space is not too large. Finally, the chapter on partitioning concludes with theoretical work on distributed fixed point construction.

It is beyond the scope of this thesis to formally answer the aforementioned second question. Whether system reaction times are influenced by distributed implementation of specification cannot be treated with our behavioral semantics. Rather, this field is an interesting topic for practical research of electrical engineers; it can be investigated by providing good strategies for hardware measurements.

5.1 Implementation-oriented Step Semantics

In this section, we formalize the step-wise behavior of a μ-Charts specification implemented on a distributed target architecture. Though the semantics defined in Section 4.1 immediately can be taken as basis for centralized hardware or software implementation — if the specification is deterministic — as we will demonstrate in Section 6.1, this is not the case for distributed implementations. The reason for that is as follows: the transition relation presented in Section 4.1 completely hides multicasting obtained by composition. The transition relation for composition $Trans_{S_1 \lhd L \rhd S_2}$ can also be interpreted as transition relation of the product automaton of S_1 and S_2. Thus, in the centralized implementation, no multicast communication is needed at all. As we will see in Section 6.1, we can in principle realize the logic formula $Trans_{S_1 \lhd L \rhd S_2}$ as a whole on one single hardware block.

However, if S_1 and S_2 shall be implemented on different processors, we do no longer implement this transition relation on one single processor. Rather, we implement $Trans_{S_1}$ and $Trans_{S_2}$ on different processors, which interchange messages via multicasting. Hence, in contrast to the centralized realization of $Trans_{S_1 \lhd L \rhd S_2}$, the logical multicasting now becomes a physical multicasting. As we are interested in reasoning about the interaction of physically separated μ-Chart programs we also have to interpret this interaction on the semantic level. Questions that are of interest, are, for instance: is the communication deadlock-free; is the behavior of the distributed system the same as of the centralized one? This is done by considering multicast communication as a least fixed point construction in the semantics.

The semantics of μ-Charts as defined in Section 4.1 is interpreted here in a fully functional way. It can be immediately executed by a suitable interpreter. Thus, we do not only define a theoretical semantics, but in addition provide a simple program for simulating and prototyping μ-Charts. This is in contrast to existing tools like STATEMATE, where the semantic behavior of the prototyping tool sometimes differs from published Statecharts semantics. Even the authors of STATEMATE admit that STATEMATE's simulation and dynamic tests tools, and its various code generators "execute a step under somewhat different circumstances" [HN96]. Furthermore, for STATEMATE Statecharts two different semantics are implemented [HN96, PU97]. One is called the "synchronous semantics" that is mostly used when the system shall be implemented in hardware. Apart from this, there also exists an "asynchronous semantics" (do not confuse these notions with our terminology of Chapter 1), which is used to describe software.

In our approach, there exists exactly one semantics that can be used to prototype and simulate reactive systems. The step semantics for μ-Charts presented in this section provides just another perspective, but is better suited for explaining distributed implementation. We will prove that it coincides with the stream semantics of Section 2.4.3 for responsive and deterministic systems that do not introduce additional non-determinism through composition.

For the completely specified, that is, semantically deterministic sequential automata A

we define its step semantics, that is, its reaction between two consecutive instants by:

$$st[\![A]\!](c,x) =_{df} \text{let } \{(c',y)\} = [\![\delta]\!](c,x) \text{ in } (c',y)$$

where "let $w = g$ in f" is a rephrasing of the lambda term $(\lambda w.f)g$.

For the definition of the step semantics for composition let $S =_{df} S_1 \lhd L \rhd S_2$. Recall that $In(S) = (In(S_1) \cup In(S_2)) \backslash L$. The elements in L are exactly those signals that potentially can be sent to and received from S_1 and S_2. Mathematically, communication in our case can be understood as iterative (Kleene) construction of a least fixed point. In this section, we show that this perception coincides with the original semantics defined in Section 2.4.3. In Section 5.4, we will analyze whether this fixed point also can be computed in the case of a distributed implementation, when parts of this computation are executed on different processors.

Semantic feedback arises from this specific signal propagation mechanism. A transition triggered by a signal set x_1 may generate the event x_2 as an action, i.e. if x_1 occurs, the transition is taken and x_2 occurs as next output set. A subset x_2' of this set may now trigger other transitions. Again, signals x_3 may be generated, which could trigger further transitions and so forth. Obviously, we get a chain reaction within one single instant. In this reaction, several transitions of interacting automata may be fired. However, recall that each Mealy machine can only fire one of its transitions in one step. In the sequel, we formalize this explanation and present the step semantics for composition including communication.

The synchrony hypothesis [Ber89] demands that an action and the event causing this action occur at the same instant. As a consequence, the aforementioned chain reaction instantaneously takes place. The signals in z generated by μ-Chart S are intersected with the signals L to be fed back and then to be unified with the external signals in x. This signal set is passed to S at the same instant of time. Mathematically, this is expressed as follows.

$$\begin{aligned}
&st[\![S_1 \lhd L \rhd S_2]\!]((c_1,c_2),x) =_{df}\\
&\qquad \text{let } (c_1',y_1) = st[\![S_1]\!](c_1, x|_{In(S_1)\backslash L} \cup \text{lfp}(f_x)|_{In(S_1)\cap L});\\
&\qquad\qquad (c_2',y_2) = st[\![S_2]\!](c_2, x|_{In(S_2)\backslash L} \cup \text{lfp}(f_x)|_{In(S_2)\cap L})\\
&\qquad \text{in } ((c_1',c_2'), y_1 \cup y_2)
\end{aligned}$$

where $\text{lfp}(f_x)$ is characterized as the least fixed point of the function

$$f_x =_{df} \lambda z.\pi_1(st[\![S_1]\!](c_1, x|_{In(S_1)\backslash L} \cup z|_{In(S_1)\cap L})) \cup \pi_1(st[\![S_2]\!](c_2, x|_{In(S_2)\backslash L} \cup z|_{In(S_2)\cap L}))$$

with respect to the subset ordering on signal sets. Here, π_1 denotes the projection of a tuple on its first component. To ease reading, we omit the configuration (c_1, c_2) in the abbreviation f_x. Then $z_0 =_{df} \text{lfp}(f_x)$ is characterized as the least solution of the following equation:

$$z \;=\; \pi_1(st[\![S_1]\!](c_1, x|_{In(S_1)\backslash L} \cup z|_{In(S_1)\cap L})) \cup \pi_1(st[\![S_2]\!](c_2, x|_{In(S_2)\backslash L} \cup z|_{In(S_2)\cap L}))$$

Here, the reader should not get confused: the least fixed point is adequate to denote the stabilization of the interaction between distributed components within one instant. However, to formalize the stream semantics, that is, the complete I/O behavior of sequential automata as in Section 2.4.3, the greatest fixed point is needed. In the remainder of this section, we yet will discuss if and when the step semantics of composition is well-defined.

Of course, we have to verify whether the interpretation of multicasting as fixed point calculation coincides with the semantics used in the preceding sections. The important difference is that the semantics presented in Section 2.4.3 and Chapter 4 collect all possible fixed points that exist (see Section 2.4.4), whereas the semantics presented in this section merely chooses the least fixed point, if one exists. The following proposition formalizes this description.

Proposition 5.1.1
The transition relation $Trans_S$ and the step semantics $st[\![.]\!]$ can be related as follows for all completely specified (deterministic and responsive) charts $S \in \mathcal{S}$.

- If S is the completely specified sequential automaton $(I, O, \Sigma, \Sigma_0, V_l, \varphi_0, \delta)$ with $\Sigma_0 = \{\sigma_0\}$ then for all configurations c, c' and all input/output events x/y:

$$(c', y) = st[\![S]\!](c, x) \Leftrightarrow Trans_S(c, x, c', y)$$

- For the composition $S =_{df} S_1 \lhd L \rhd S_2$ the following holds: If for all $c_n, c'_n \in \mathcal{C}(S_n)$, $x \in \mathcal{I}(S)$, and $y_n \in \mathcal{O}(S_n)$ with $n = 1, 2$

$$(c'_n, y_n) = st[\![S_n]\!](c_n, x|_{In(S_n)}) \Leftrightarrow Trans_{S_n}(c_n, x|_{In(S_n)}, c'_n, y_n)$$

then also

$$((c'_1, c'_2), y_1 \cup y_2) = st[\![S]\!]((c_1, c_2), x) \Rightarrow Trans_S((c_1, c_2), x, (c'_1, c'_2), y_1 \cup y_2)$$

□

Note that in the above proposition of the last item only the implication, but not the equivalence can be guaranteed. The reason is as follows: remember the pathological examples of Section 2.4.4. In these examples, even if all automata involved were deterministic, the composed chart need not necessarily to be, too, because unintended non-determinism caused by instantaneous feedback could occur. Thus, $Trans_S$ includes all existing fixed points, whereas $st[\![S]\!]$ only chooses the least one. This explains why we are only capable to deduce implication instead of equivalence. We now will carry out the proof.

Proof 15 This proof is done by structural induction over $\mathcal{S}$.

- For the first case we define $S = (I, O, \Sigma, \Sigma_0, V_l, \varphi_0, \delta)$ with $\Sigma_0 = \{\sigma_0\}$. As S is responsive and deterministic, we can conclude that $[\![\delta]\!](c,x) \neq \emptyset$ holds and that there exist exactly one c' and y such that $(c', y) \in [\![\delta]\!](c,x)$ for all c and x. As a consequence, we get:

$$\begin{aligned} & (c', y) = st[\![(I, O, \Sigma, \Sigma_0, V_l, \varphi_0, \delta)]\!](c,x) \\ \Leftrightarrow\ & \{(c', y)\} = [\![\delta]\!](c,x) \\ \Leftrightarrow\ & Trans_S(c, x, c', y) \end{aligned}$$

- We now prove the second case of the structural induction. Let $S =_{df} S_1 \lhd L \rhd S_2$ and the premise be true. Furthermore, let $(c_1, c_2), (c_1', c_2') \in \mathcal{C}(S)$, $x \in \mathcal{I}(S)$, and $y \in \mathcal{O}(S)$. Then $((c_1', c_2'), y) = st[\![S_1 \lhd L \rhd S_2]\!]((c_1, c_2), x)$ implies $y = \mathrm{lfp}(f_x)$ and therefore is, using the definitions for $st[\![S_1 \lhd L \rhd S_2]\!]$ and $Trans_{S_1 \lhd L \rhd S_2}$, equivalent to:

$$\begin{aligned} \Leftrightarrow\ & \exists y_1, y_2 . \mathrm{lfp}(f_x) = y_1 \cup y_2 \wedge \\ & (c_1', y_1) = st[\![S_1]\!](c_1, x|_{In(S_1)\setminus L} \cup \mathrm{lfp}(f_x)|_{In(S_1)\cap L}) \\ & (c_2', y_2) = st[\![S_2]\!](c_2, x|_{In(S_2)\setminus L} \cup \mathrm{lfp}(f_x)|_{In(S_2)\cap L}) \\ \Leftrightarrow\ & \exists y_1, y_2 . \mathrm{lfp}(f_x) = y_1 \cup y_2 \wedge \\ & Trans_{S_1}(c_1, x|_{In(S_1)\setminus L} \cup \mathrm{lfp}(f_x)|_{In(S_1)\cap L}, c_1', y_1) \wedge \\ & Trans_{S_2}(c_2, x|_{In(S_2)\setminus L} \cup \mathrm{lfp}(f_x)|_{In(S_2)\cap L}, c_2', y_2) \\ \Leftrightarrow\ & Trans_{S_1 \lhd L \rhd S_2}((c_1, c_2), x, (c_1', c_2'), \mathrm{lfp}(f_x)) \end{aligned}$$

□

As we already pointed out in Section 2.4.4, instantaneous feedback can imply semantic inconsistencies due to causality conflicts. In the following, we will discuss by means of the pathological examples known from Section 2.4.4, what these causality conflicts semantically mean for the deterministic semantics we use for implementation. We will see that several or no fixed points may exist. However, we will recognize that monotonic specifications always have a well-defined step semantics, where a unique least fixed point exists.

Existence of Several Fixed Points

Let us consider the example of Section 2.4.4 in Figure 5.1 again, where we first define $t_1 = a/b$ and $t_2 = b/a$. Table 5.1 shows the input/output behavior of f_x. Obviously, for $f_\emptyset$ two fixed points, namely $\emptyset$ and $\{a, b\}$ exist. For this reason, Argos [Mar92, MH96] would reject this chart as not being well-formed. Our approach differs in this point: as we have seen in Section 2.4.4, we can define a stream semantics for this example by including all fixed points. Hence, the designer still has the possibility do modify this specification to one that merely contains one fixed point by refinement.

Of course, this non-deterministic behavior is not implementable. Therefore, we have to choose one of the fixed points. In case of the centralized implementation we could choose the least one of the two fixed points, with respect to the subset ordering $\subseteq$ on signal sets. This choice seems to be reasonable, because it coincides with the result we get when operationally computing the fixed point starting with the empty stimuli set. However, if we aim at implementing the automata with labels t_1 and t_2 on different processors, we would get a deadlock situation since the (data driven) implementation of each automaton "waits" for a signal of its neighbor, which, however, never is sent. Without any additional stimulus, the system would therefore deadlock. Since deadlocks can cause critical situations in reactive systems, we have to avoid this adverse behavior. In Section 5.2.3, we will discuss a technique, based on so-called signal graphs, that detects specifications of this type.

We will now demonstrate that it is indeed possible to refine the above example to a specification that merely contains one fixed point. According to the rule for interface refinement on Page 72, we extend the input interfaces of both automata in Figure 5.1 to $\{a, c\}$ and $\{b, c\}$, respectively. Moreover, we modify the transition labels t_1 and t_2 to $c \wedge a/b$ and $\neg c \wedge b/a$, respectively. It is not difficult to see that the corresponding functions $f_\emptyset$ and $f_{\{c\}}$ now only have one single fixed point, namely $\emptyset$. Thus, we recognize that it is indeed possible to refine non-causal to causal specifications (see definitions in Section 5.2.3).

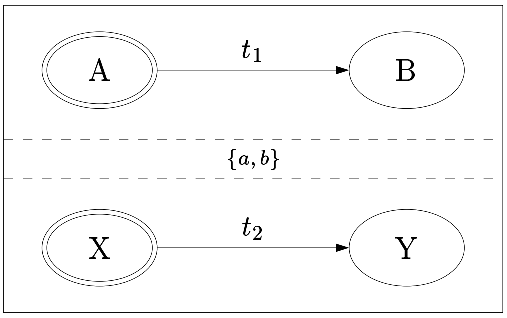


Figure 5.1: Problems with fixed points

It is also possible to have several fixed points without having a least one. To demonstrate this, we modify t_1 and t_2 in Figure 2.11 to $t_1 = \neg a/b$ and $t_2 = \neg b/a$, respectively. Table 5.2 shows the input/output behavior of f_x. This chart has both $\{a\}$ and $\{b\}$ as fixed points for the empty input set. These fixed points are not comparable with respect to the subset ordering $\subseteq$. Furthermore, $\emptyset$ is not a fixed point. Hence, a least fixed point does not exist.

The reason for this result is that $\emptyset \subseteq \{b\}$, but $f_\emptyset(\emptyset) \not\subseteq f_\emptyset(\{b\})$, i.e. $f_\emptyset$ is not monotonic with respect to the subset ordering $\subseteq$ on signal sets. An equivalent observation can be made for $f_{\{a\}}$ and $f_{\{b\}}$. Generally, f_x may be non-monotonic if a signal occurs "negatively" in a trigger condition. However, as we have already pointed out, negations of signals in trigger conditions are necessary to get deterministic automata. Whenever

z	$f_\emptyset(z)$	$f_{\{a\}}(z)$	$f_{\{b\}}(z)$	$f_{\{a,b\}}(z)$
$\emptyset$	$\underline{\emptyset}$	$\{b\}$	$\{a\}$	$\{a,b\}$
$\{a\}$	$\{b\}$	$\{b\}$	$\{a,b\}$	$\{a,b\}$
$\{b\}$	$\{a\}$	$\{a,b\}$	$\{a\}$	$\{a,b\}$
$\{a,b\}$	$\underline{\{a,b\}}$	$\underline{\{a,b\}}$	$\underline{\{a,b\}}$	$\underline{\{a,b\}}$

Table 5.1: Several fixed points for $t_1 = a/b$ and $t_2 = b/a$

z	$f_\emptyset(z)$	$f_{\{a\}}(z)$	$f_{\{b\}}(z)$	$f_{\{a,b\}}(z)$
$\emptyset$	$\{a,b\}$	$\{a\}$	$\{b\}$	$\underline{\emptyset}$
$\{a\}$	$\underline{\{a\}}$	$\underline{\{a\}}$	$\emptyset$	$\emptyset$
$\{b\}$	$\underline{\{b\}}$	$\emptyset$	$\underline{\{b\}}$	$\emptyset$
$\{a,b\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Table 5.2: No least fixed point for $t_1 = \neg a/b$ and $t_2 = \neg b/a$

f_x is non-monotonic, a least fixed point may not exist. This may lead to an undefined semantics. Therefore, this time we have to *reject* this kind of μ-Charts. To "reject" here means that the compiler outputs an error message while performing semantic analysis. Also Argos would reject similar specifications. In general, using μ-Charts, we can give a semantics for monotonic functions only. Otherwise, the fixed point computation (by Kleene iteration) and thus the communication between distributed charts would possibly not terminate.

For the original semantics in Section 2.4.3, we have demonstrated a possibility how for these specifications nevertheless a well-defined semantics can be formalized and how a semantics can be given even to non-monotonic functions. Remember that this has been achieved through non-determinism. With a non-deterministic semantics we can express that either $\{a\}$ or $\{b\}$ is the observation in the current step/instant. For instance, choosing $\{a\}$ to be the semantics of this chart in the current instant would mean that the lower automaton in Figure 5.1 reacts faster than the upper.

Non-existence of Fixed Points

A further problem is the possible non-existence of fixed points. We continue the example pictured in Figure 5.1 by slightly modifying it to $t_1 = a/b$, $t_2 = \neg b/a$. Again, the input/output behavior of f_x is shown in Table 5.3.

z	$f_\emptyset(z)$	$f_{\{a\}}(z)$	$f_{\{b\}}(z)$	$f_{\{a,b\}}(z)$
$\emptyset$	$\{a\}$	$\{a,b\}$	$\underline{\emptyset}$	$\{b\}$
$\{a\}$	$\{a,b\}$	$\{a,b\}$	$\{b\}$	$\{b\}$
$\{b\}$	$\emptyset$	$\underline{\{b\}}$	$\emptyset$	$\underline{\{b\}}$
$\{a,b\}$	$\{b\}$	$\{b\}$	$\{b\}$	$\{b\}$

Table 5.3: No fixed point for for $t_1 = a/b$ and $t_2 = \neg b/a$

Note that $f_\emptyset$ does not have any fixed point. Again, the reason is the non-monotonicity of f_x. Hence, we require that the compiler rejects, that is, produces an error message while analyzing the corresponding chart. Though we are unable to give a well-defined deterministic step semantics for this chart S, with the non-deterministic version of this semantics of Section 2.4.3 we get $[\![S]\!] = \emptyset$.

Altogether, we can observe that there indeed are cases when no well-defined step semantics can be given to a specification. In theses cases, the function f_x has been non-monotonic. As a consequence, to get guaranteed *implementable specifications* we restrict ourselves to monotonic functions to fulfill the premise for the theorem of Knaster and Tarski (see Section 5.4). Non-monotonic charts are rejected of the compiler by semantic analysis. However, notice that successfully passing this analysis is just one step towards correct implementations and does not yet guarantee deadlock free distributed implementation in any case. Apart from the examples presented here, there may be other specifications, where all compositions are monotonic, but nevertheless cannot be implemented. Examples of this type will be discussed in the following.

5.2 Problems with Signals through Partitioning

In this section, we discuss the problems that potentially may occur when one tries to partition a design with Statecharts and to implement the different parts on a processor network. Problems of this type are the lack of a global clock for a loosely coupled network, communication deadlocks, and the scanning for signal absences in synchronous languages. Note that these aspects are not problematic in case of centralized implementations, but are common to most synchronous languages.

5.2.1 Motivation

Partitioning a specification written in a synchronous language, we are faced with a number of interesting problems that are common to most synchronous languages, such as Esterel, Lustre, Signal, Argos, and μ-Charts. In the following, we will discuss these problems in detail by means of μ-Charts and present strategies to remedy them. These strategies provide partly automated solutions and partly syntactic restrictions to the design process of reactive systems with synchronous languages. We begin with a classifications of the envisaged problems and then sketch appropriate solutions.

First, synchronous languages are based on a perfect synchronous time model; in particular, the assumption is that all components of a specification interact in lock-step. While this assumption can be fulfilled when the specification is implemented on centralized and tightly coupled target architectures, respectively, this is not the case for loosely coupled systems. Distributed processors in embedded systems, as they occur in automobile industry, are mostly loosely coupled: they communicate via one or several field busses, such as the CAN bus [Ets94], for instance. For these systems, we cannot longer assume that all distributed components are triggered by one single global system clock. Hence, it is difficult to guarantee that also distributed implementations of synchronous specifications interact in lock-step. In particular, care has to be taken that all distributed components start to react in the same instant. Since external input signals are responsible to start reaction in each step, we have to examine these signals and their mutual dependence in more detail. We will see that it is difficult to argue about the relative presence or absence of external signals. As a consequence, we require for distributed implementation that each component can only scan single external input signals in each instant, but not groups of them.

Second, while external signals are produced by the system environment, internal signals are sent from other system components. As we will see later on, the classification into external and internal signals plays an important role in our analysis. If components written in a synchronous language are partitioned and implemented on a distributed architecture, the logical communication of the specification becomes a physical communication. Now, it is important to guarantee that this communication, which possibly consists of a number of sending and receiving actions and in which a number of components can be involved, does not have deadlocks.

The third problem is common to external and internal signals. As a global clock is not available in loosely coupled systems, it is not possible for a distributed reactive system to react on the absolute absence of a signal. The problem here is to decide when to react on the absence of a signal.

There are in principal two solutions for the first problem. First, we give syntactic restrictions that help the designer to specify sequential automata in a way that they always react on just one single external input signal. However, for composed charts we cannot give easy-to-understand syntactic guidelines. Here, tool support is necessary. We discuss how this in principle has to look like. Since for some reactive systems it may be too restrictive to require that its distributed implementation merely reacts on the presence of single signals, but also on the relative presence or absence of more than one, we sketch how this restriction can be weakened using additional information about the external signals of the environment.

For the second problem we provide an algorithm to detect deadlocks that is similar to the approach for causality analysis suggested by ESTEREL.

Last but not least, we give syntactic guidelines that support the specifier in writing specifications that do not cause problems in case of distributed implemented due to scanning of absolute absence of external or internal input signals. In the following, we will discuss these three problems and their solutions in more detail. To that end, we first give a classification for input signals.

Input Signals

Whether a signal in the design is an internal signal or an external stimulus can be decided with respect to the overall design of the chart S. Without loss of generality (see normal form), we assume that S has the form $S_1 \lhd L \rhd S_2$, that is, $S_1 \lhd L \rhd S_2$ is the μ-Chart specification of the entire system under development. External stimuli are signals that are read from the sensors and internal signals are generated by any part of the specification. We say that a signal is *external* with respect to S, whenever it is contained in $(In(S_1) \cup In(S_2)) \backslash L$. Signals in $(In(S_1) \cup In(S_2)) \cap L$ are termed *internal* instead.

5.2.2 Problems with External Input Signals

This section addresses problems of distributed implementation of reactive systems specified with μ-Charts that are caused by external signals. However, before we are able to concentrate on distributed implementation we have to understand how a centralized implementation of these programs works. Later on, we will try to apply concepts of centralized implementation to distributed implementation.

As we have already discussed, μ-Charts assume, typically for synchronous languages, a *synchronous* implementation model. This model implies that for every single incoming and outgoing signal we must be able to determine in which instant it occurs, i.e. there must be a total mapping from signals to instants. This mapping links the physical environment time to the logical system time. In the implementation, this mapping will be provided by the system interfaces.

Figure 5.2 (inspired by [Gir94, CGP94, CFG95, CCGJ97]) shows an electronic control unit (ECU) with four input interfaces A to D. Each of them receives external input signals from the sensors. Different signal instances are distinguished by different indices in Figure 5.2. All signals arrive at the interfaces at specific, real-valued time points and collected in one input queue according to the mapping that determines the instants in which the signals occur. Therefore, the task of this mapping is to aggregate these signals to sets of input signals, that is, to events as shown in Figure 5.2.

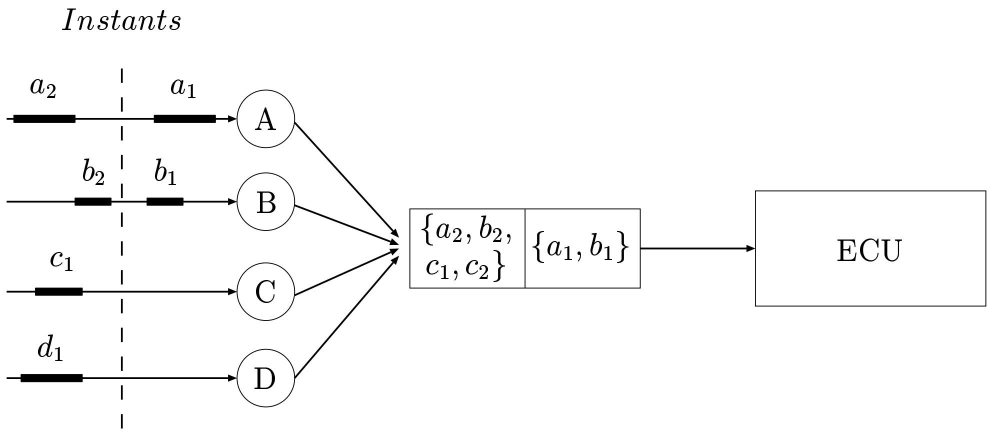


Figure 5.2: Centralized implementation

In the definition of the semantics, we have simply assumed that this mapping is known. The reality, however, is slightly more complicated. In the sequel, we will see that it is by no means trivial to determine this mapping. For better understanding, we first analyze how reactive systems are executed by a centralized implementation. They interact with their environment in a cycle-oriented way:

1. They read the current input
2. They react on this input, that is, perform an internal, for the environment invisible action
3. They write the current output

In principle, these three steps are executed sequentially as long as the system is running. The interesting question now is when a cycle is started. This question corresponds one-to-one to the problem of finding a mapping from signals to instants because all input signals that are collected in the same current input set are considered to occur in the same instant.

One could imagine three possibilities when a new cycle can be executed:

1. The preceding cycle terminates (demand driven implementation)
2. An external clock triggers the system (demand driven implementation)
3. New input on the sensor(s) is available (data driven implementation)

The reader may agree that the behavior of a reactive system should never depend on the performance of processors, interfaces, sensors, actors, or other devices. Instead, the behavior should be the same as formulated in the mathematical semantics and identical for all possible implementations, independent of speed or number of processors used. As a consequence, it is not surprising that the implementation alternative (1) does not come into question at all as it heavily depends on the reaction time. Thus, only the alternatives (2) and (3) remain.

Solution of Lustre

First of all, we will analyze the implementation strategy of the synchronous data flow language LUSTRE and discuss to what extent it is applicable for μ-Charts. Figure 5.2 outlines the technique applied in [Gir94]. In LUSTRE, it is possible to define relations between inputs: exclusions, implications, clock constraints, and others. These relations supplement the original LUSTRE program and must be fulfilled by the implementation, that is, more precisely, the interface. These relations prescribe to the interface which input signals are collected in the same instants. Moreover, if there are no relations at all, it is also possible to aggregate signals to events; in this case, the largest prefix of the incoming signals including each input signal at most once is constructed and collected in one instant. In Figure 5.2, this is $\{a_1, b_1\}$. The very next input, b_2, cannot be added to this set because the data flow language LUSTRE requires that subsequent occurrences of a single signal must be distinguishable.

The concept of simply collecting largest prefixes that fulfill the property that subsequent signal instances must not be collected in on instant, is not suitable in non-data-flow languages like ESTEREL or Argos. One of the attributes of them is just that parallel statements and automata, respectively are allowed to produce the same signals instantaneously using broadcasting. Thus, different signal instances possibly must be collected in one instant. In such cases, for non-Boolean signals, conflicts occur that are solved by resolution functions [Ber96, Sch96b]. Also μ-Charts in principle do not require that the output interfaces of all sequential automata are disjoint. However, as we will see in Section 5.4, it is advisable to limit ourselves to automata with disjoint output interfaces for distributed implementation in order to guarantee distributed fixed point computation without loss of generality. Nevertheless, it is worthwhile investigating other techniques to find an appropriate mapping from physical to logical time for μ-Charts.

Demand Driven Implementations

Let us consider the implementation strategy (2). Here, we assume that we have an external clock signal as sketched in Figure 5.3. This clock signal has the property that it occurs more frequently, that is, at a higher rate than any other signal. The idea to identify one single signal which is responsible to cause a reaction also has been adopted in the data flow language SIGNAL.

For SIGNAL, Benveniste et al. [BG97] call a state transition system where the control only depends on the previous state, but not on the relative presence or absence of signals,

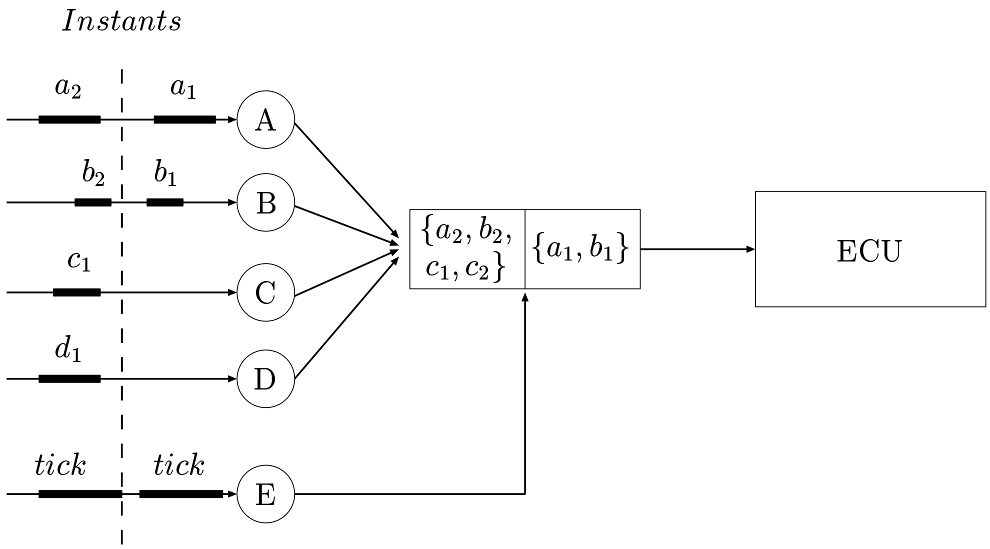


Figure 5.3: Centralized implementation with global clock

an *endochronous* state transition system. These systems have the advantage that tests for relative absence or presence is not needed because only one external signal enables a systems reaction at each instant.

Systems that are not endochronous are called *exochronous* by Benveniste. However, it is always possible to transform an exochronous system to an endochronous one by simply adding a new environment signal s to the system specification, which is assumed to be more frequently present than any other environment signal considered so far. Hence, no tests for mutual absence/presence of the other signals are needed anymore as we can test for the *absolute* absence/presence of s now.

However, choosing a clock-dependent implementation model, we have to remember that every transition can only be fired when a clock signal occurs. Hence, every transition trigger t actually means that t has to be true and in addition a clock tick (in [BG97], the most frequent signal s is equal to this tick) must occur; the overall transition condition then is $t \wedge tick$ instead of simply t as pictured in Figure 5.4 for the left motor of the central locking system (compare with Figure 2.3).

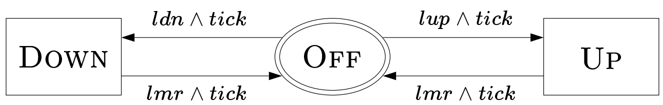


Figure 5.4: Left motor with ticks

It is not hard to imagine that the overall behavior of a specification including such a clock varies when moving to a faster or slower clock. Clearly, in such a specification it is assumed implicitly that the implementation is fast enough so that the reaction can be terminated within two clock cycles. There are specifications where a clock-oriented behavior is wanted, may be even needed like in a digital stop-watch (see Figure 2.9) or examples with timeouts, for instance. However, there also may be specifications where it is not necessary to fix a clock-oriented behavior from the very beginning. Note, however, that we nevertheless have to deal with the problem of aggregating signals to events in such cases, too.

A further problem is the distribution of specifications including a global clock. As we have shown above, every single transition can only be fired when a clock tick occurs. For a distributed implementation this means that each control unit that executes a part of the implementation must be aware of this clock signal. This can be solved by two different strategies.

First, we can connect each electronic control unit by a wire to the global clock. However, for embedded systems this solution is hardly applicable, as physical wiring is complicated and therefore expensive. As an example, let us assume that the global clock is located in the engine compartment of an automobile and that the control units for the power windows are located in the car's doors. In this case, clock wires from engine compartment to doors would have to be installed, which is far away from being a cheap solution.

Second, one could think of sending the clock signals via the bus. Due to the protocols used in the marketable field bus systems, messages can be delayed. As receiving and sending of clock signals has to be carried out instantaneously, this is not a feasible solution. A more applicable solution would be to use local clocks for each control unit that run at the same speed. Though using clocks that synchronize each other from time to time is technically possible, we aim at a solution of this problem that works — under certain circumstances — without clocks at all. How to achieve this goal will be the contents of the following.

Data Driven Implementations

We will motivate that it is indeed possible to implement reactive systems specified with μ-Charts without the usage of a global system clock. Whether a specification fulfills this property depends on two facts. First, the specifier has to meet certain methodological requirements with respect to external input signals and second, a causality analysis for internal input signals is needed.

For external signals we require that the semantics of $S =_{\mathit{df}} S_1 \lhd L \rhd S_2$ be independent of the reaction cycle's velocity. This can be guaranteed whenever a specification has equivalent behaviors for all non-empty external input events that can be ordered by the subset ordering on sets. This implies that S only is allowed to react on single input signals, but not on sets of input signals. This is a rather strong condition, but remember that it has to hold merely if S will be implemented on a distributed architecture. This condition is formally expressed by:

$$\forall c \in \mathcal{C}(S), x, x' \in \mathcal{I}(S) : \emptyset \neq x \subseteq x' \Rightarrow st[\![S]\!](c, x) = st[\![S]\!](c, x') \tag{5.1}$$

This semantic predicate, which in particular requires deterministic specifications, ensures that the μ-Chart specification S is independent of the scanning rate for external inputs. In the following, we will see that this requirement can be weakened if additional assumptions on the environment are made. It is now an interesting question to derive syntactic requirements from this semantic property and to embed these requirements in a design method for engineers. However, it will turn out that for reasons of complexity we cannot fully guarantee this property by methodological hints only. We will show to which extent this is possible and when we must use additional machine support to filter out specifications that do not satisfy this requirement.

The above semantic requirement reduces the number of Boolean predicates over $In(S)$ that can be used as valid trigger conditions. We show this in the sequel. The stimuli $\emptyset$, $\{a\}$, $\{b\}$, and $\{a, b\}$ are all events that are possible when the input interface of the Mealy machine S consists of signals a and b; these events are ordered according to the subset relation over the power domain.

Due to the above requirement, events $\{a, b\}$, $\{a\}$, and $\{b\}$ must yield identical reactions in each configuration. The only Boolean predicate over $\{a, b\}$ that performs this is the disjunction $a \vee b$. This heavily restricts the number of trigger conditions that are valid. In particular, the transition $crash \wedge ignition$ of our central locking system in Figure 2.3 would not be valid without any further assumptions on the relative presence between the external stimuli $crash$ and $ignition$.

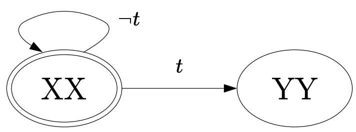


Figure 5.5: Non-valid triggers for external stimuli

To give some examples of non-valid trigger conditions we take a look at the automaton A in Figure 5.5 with input interface $\{a, b\}$. To define t as $a \wedge b$, $a \wedge \neg b$, $a \vee \neg b$, or even a yields non-valid triggers (see Table 5.4). Let us now analyze these triggers in more detail. The trigger $t = a \wedge b$ causes troubles because it separates the events $\{a, b\}$ from $\{a\}$ and $\{b\}$, respectively. To "separate" in this context means that the event $\{a, b\}$ triggers the transition labeled with t, whereas $\{a\}$ and $\{b\}$ do not. More formally: $st[\![A]\!](XX, \{a, b\}) \neq st[\![A]\!](XX, \{a\}) = st[\![A]\!](XX, \{b\})$. In practice, this implies that a "fast" implementation of the automaton pictured in Figure 5.5 with a fast scanning rate possibly would cause a state transition from state XX to YY and a slow scanning rate would force the automaton to remain in its XX state. A similar observation can be made for the other triggers listed in Table 5.4.

For sequential automata, we can give the following methodological guidelines for software engineers: if no additional relationships for external input signals are given, for each sequential automaton $A =_{df} (I, O, \Sigma, \Sigma_0, V_l, \varphi_0, \delta)$ and each of its states $\sigma \in \Sigma$ the following is required. The trigger conditions of all transitions with σ as source state has the pattern $t \wedge t'$, where t has to be equivalent to $\bigvee_{s \in In(A) \setminus L} s$ and t' is an arbitrary term over local variables and/or internal signals. The shape of t' is the subject of the following section. If, in contrast, additional signal assumptions are available, it suffices to verify

t	$\{x \in \mathcal{I}(S) \mid x \models t\}$	$\{x \in \mathcal{I}(S) \mid x \models \neg t\}$
$a \wedge b$	$\{\{a, b\}\}$	$\{\emptyset, \{a\}, \{b\}\}$
$a \wedge \neg b$	$\{\{a\}\}$	$\{\emptyset, \{b\}, \{a, b\}\}$
$a \vee \neg b$	$\{\emptyset, \{a\}, \{a, b\}\}$	$\{\{b\}\}$
a	$\{\{a\}, \{a, b\}\}$	$\{\emptyset, \{b\}\}$

Table 5.4: Non-valid triggers for external stimuli

the weaker Condition 5.2 as shown above. However, it does not only depend on single automata whether the implementation is independent of the interface sampling speed. Rather, composed automata have to be verified, too, since configurations do not only consist of single machine states, but tuples of states of all composed automata. Figure 5.6 shows an example.

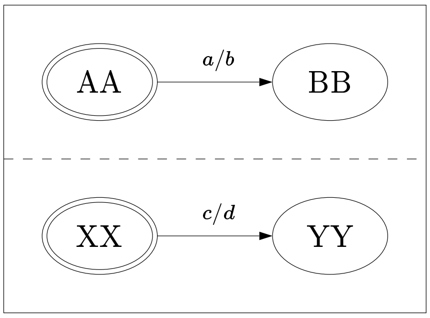


Figure 5.6: Composition with non-valid triggers

To find out whether the composition of two or more automata fulfills Requirement 5.1, first all reachable configurations have to be computed. To do this is a much too complex and error-prone task to be done by paper and pencil. Therefore, we cannot give any methodological hints in the case of composition. Rather, we recommend to use automatic verification techniques (see Section 4). As we will demonstrate in the very next section, machine assistance is anyway necessary to compute causality relations of internal signals.

It would be convenient for software engineers, if they could also use other predicates over external input events than merely disjunctions as required above. The sole possibility to achieve this for distributed implementations without using a global clock is to introduce additional restrictions on the presence or absence of external inputs that exclude combinations of certain signals to events.

We therefore define $\Xi(S) \subseteq \mathcal{I}(S)$ as the set of *possible external input events for* S. Each $\xi \in \Xi(S)$ specifies a set of external input signals that can occur simultaneously in one instant in the input interface of specification S. This is a requirement to the environment of S and means that it cannot deliver any external input events $\xi \in \mathcal{I}(S) \backslash \Xi(S)$.

With this definition, the Requirement 5.1 can now for external input be weakened to:

$$\forall c \in \mathcal{C}(S), x, x' \in \Xi(S) : \emptyset \neq x \subseteq x' \Rightarrow st[\![S]\!](c, x) = st[\![S]\!](c, x') \tag{5.2}$$

For example, if we defined $\Xi(S) =_{df} \{\emptyset, \{b\}, \{a, b\}\}$ in the example pictured in Figure 5.5 and Table 5.4, the trigger $t = a \wedge \neg b$ would be valid, if we defined $\Xi(S) =_{df} \{\emptyset, \{a\}, \{a, b\}\}$ both triggers $t = a \vee \neg b$ and $t = a$ would be valid, and if we defined $\Xi(S) =_{df} \{\emptyset, \{a\}, \{b\}\}$ all triggers of Table 5.4 and Figure would be valid.

As already mentioned, LUSTRE and ESTEREL provide calculi to express relationships between environment input signals. Rules that are possible in the calculus are, for instance, mutual exclusion of certain signals and signal implication: $a \,\textsf{then}\, b$ means that whenever signal a is present, then also signal b has to be; $a \,\textsf{xor}\, b$ requires that there cannot be any event x such that both a and b are included in x. These relationships can be expressed by means of possible external input events, too:

$$\begin{aligned} a \,\textsf{then}\, b &\implies \forall x \in \mathcal{I}(S) : (a \in x \Rightarrow b \in x) \Rightarrow x \in \Xi(S) \\ a \,\textsf{xor}\, b &\implies \forall x \in \mathcal{I}(S) : ((a \in x \wedge b \notin x) \vee (a \notin x \wedge b \in x)) \Rightarrow x \in \Xi(S) \end{aligned}$$

Signal relations together with the actual μ-Chart specification represent an assumption/-commitment or rely/guarantee like specification style [Stø95, SDW95, Stø95, BS97]: if the signal assumptions are fulfilled by the implementation, a time independent behavior is guaranteed.

5.2.3 Problems with Internal Input Signals

Up to now, we have analyzed relationships between external signals. We have found out that only under certain circumstances trigger conditions over these signals are valid. In this section, we will discuss relationships of internal signals, i.e. signals that are not produced by the environment, but by any system component and are made available using multicasting. The internal signals are possibly sent between distributed components through a communication medium according to their causality relationship. Therefore, for distributed implementation a causality analysis is needed.

In the sequel, we define a relation $\rightarrow \subseteq M \times \wp(M) \times \wp(M) \times M$ for signals for each reachable configuration c of the overall system. This relation denotes the dependence of the presence of a signal on the presence or absence of other signals. Since this relation varies from system configuration to system configuration, it has to be computed for every single instant anew. It is defined as follows.

Definition 5.2.1 (Causality Relation for Internal Signals)
If the presence of signal s in one instant implies the presence of signal s' in the same instant under the condition that also all signals in the set p^+ are present and all signals in the set a^- are absent, we say that s' is *causally dependent on s with respect to the presence set p^+ and the absence set a^-*. This relation is denoted by $s \rightarrow^{p^+}_{a^-} s'$. □

Since this causality relation only deals with the presence of specific signals, it implicitly requires the monotonicity of the specification in order to compute all existing causality relations of the specification. However, as we have to limit ourselves to monotonic charts anyway (see Section 5.4.2), this is in practice no restriction at all. Now, we determine this causality relation for μ-Charts by structural induction over $\mathcal{S}$, that is, for sequential automata and composition; the operator for signal hiding does not have to be examined separately because this relation is identical for $[S]_K$ and S for all $S \in \mathcal{S}$. Since we consider relations within single instants, each relation only is valid for a specific configuration. We start with defining the causality relation for sequential automata.

Let $A =_{df} (I, O, \Sigma, \Sigma_0, V_l, \varphi_0, \delta)$ be a deterministic, sequential automaton and $\sigma \in \Sigma$ its current control state. In state σ, s' is causally dependent on s with respect to the presence set x, where $s \notin x$, and the absence set $\mathcal{I}(A)\backslash x$, if and only if there exist $\sigma' \in \Sigma$ and $y \in \mathcal{O}(A)$ such that $s' \in y$ and $(\sigma', y) = st[\![A]\!](\sigma, x \cup \{s\})$. This is abbreviated to $s \to^{x}_{(\mathcal{I}(A)\backslash x)} s'$).

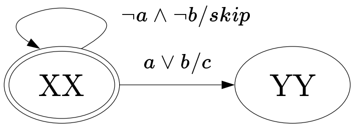


Figure 5.7: Causality relation

To give an example, we compute this relation for the automaton in Figure 5.7 in control state XX:

$$a \to^{\{b\}^+}_{\emptyset^-} c, \quad a \to^{\emptyset^+}_{\{b\}^-} c, \quad b \to^{\{a\}^+}_{\emptyset^-} c, \quad b \to^{\emptyset^+}_{\{a\}^-} c$$

Obviously, the presence of c immediately follows from the presence of a or b without taking the presence or absence of b or a into account, respectively. As a consequence, we "minimize" the causality relation as follows in order to avoid redundant dependencies:

$$a \to^{\emptyset^+}_{\emptyset^-} c, \quad b \to^{\emptyset^+}_{\emptyset^-} c$$

We require that this minimization of causal signal dependencies always has to be carried out for sequential automata. This is possible whenever there are causality relations $s_1 \to^{p_1^+}_{a_1^-} s_1'$ and $s_2 \to^{p_2^+}_{a_2^-} s_2'$ for automaton A and a signal $s \in In(A)$ (in the example b and a, respectively) such that $s \in p_1^+ \cap a_2^-$ or $s \in p_2^+ \cap a_1^-$. Then, both dependencies are simplified to $s_1 \to^{p_1^+\backslash s}_{a_1^-} s_1'$, $s_2 \to^{p_2^+}_{a_2^-\backslash s} s_2'$ and $s_1 \to^{p_1^+}_{a_1^-\backslash s} s_1'$, $s_2 \to^{p_2^+\backslash s}_{a_2^-} s_2'$, respectively. Moreover, we use the following abbreviations:

$$\begin{aligned} s \to s' &\iff_{df} s \to^{\emptyset^+}_{\emptyset^-} s' \\ s \to^{x^+} s' &\iff_{df} s \to^{x^+}_{\emptyset^-} s' \\ s \to_{x^-} s' &\iff_{df} s \to^{\emptyset^+}_{x^-} s' \end{aligned}$$

Note that with this relation also relationships between external signals can be expressed: s then s', for instance, is equivalent to $s \to s'$.

The single relations between signal pairs can be combined to a directed, labeled graph, termed *signal graph*. The signal graph for a μ-Chart that consists of a number of sequential automata simply is constructed by composing the signal graphs of the different automata. Here, nodes that denote identical signals are simply joined to one single node.

Signal graphs can contain cycles, which potentially can lead to causality conflicts. Since causality conflicts correspond to communication deadlocks in the case of distributed implementations, it is important to avoid such specifications because they cannot be implemented free of deadlocks.

However, as we already know from ESTEREL, not every cycle necessarily leads to causality conflicts [SBT96, Ber96, Tom97, Ber98] (discussed by means of cyclic, combinational hardware in this literature). In ESTEREL, cycles can be distinguished into *harmless* and *harmful* cycles. In the sequel, we follow this classification and define a class of cycles in signal graphs that are considered to be *causality error free*, i.e. harmless. If a signal graph does not contain any cycle with causality errors, the entire graph is termed causality error free. For a harmless or, in other words, a causality error free cycle for all signals (= nodes) we can determine non-ambiguously whether it is absent or present in the current instant. For cycles with causality errors this is not the case.

Before we give a formal definition of causality error free cycles, we first motivate, by the aid of some well-known examples, which cycles contain causal loops and which not. The first example is the cyclic ESTEREL program taken from [SBT96]:

```
present i then
   present s then emit t end
else
   present t then emit s end
end
```

The example contains a path from `s` to `t` and one in the opposite direction. Hence, this ESTEREL code has a cycle in its signal graph (see Figure 5.8). However, taking a closer look at the program, we notice that this cycle does not cause any causality conflict since the presence or absence of signal `i` only enables one path simultaneously. In Figure 5.8, this fact becomes clear through the contradictory cycle labels $\{i\}^+$ and $\{i\}^-$.

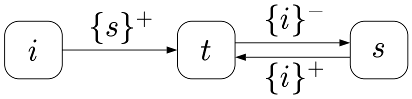


Figure 5.8: Signal graph for ESTEREL example

Note the small, but significant difference between this ESTEREL program and the pathological μ-Charts example with labels a/b and b/a in Sections 2.4.4 and 5.1. There, we do not have any signal like i that causes a case distinction and therefore breaks the causality cycle. The cycle in this example does contain a causality error as the signal graph in

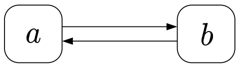


Figure 5.9: Signal graph for the pathological a/b, b/a example

Figure 5.9 shows. Here, the signals a and b are involved in a mutual dependency: the presence of a depends on the presence of b and vice versa.

The ESTEREL example does not only demonstrate that not all cycles necessarily must lead to causality conflicts, but also shows that signal graphs are, similar to *constructive cycles* in hardware [SBT96, Ber96, Tom97, Ber98], also a convenient means to characterize cycles in ESTEREL programs and combinational hardware. In the remainder of this section, we will outline the analogy between constructive Boolean logic — which is isomorphic to electrically stable circuits [Shi96] — and causality error free signal graphs.

The next example is the instantaneous dialog [Mar92] (Figure 5.10). The corresponding signal graph is pictured in Figure 5.11. Though we do not have any contradictory labels on the cycle arcs as in the ESTEREL example, this is a harmless cycle. The reason is that the presence of both signals in the cycle a and y can be determined non-ambiguously.

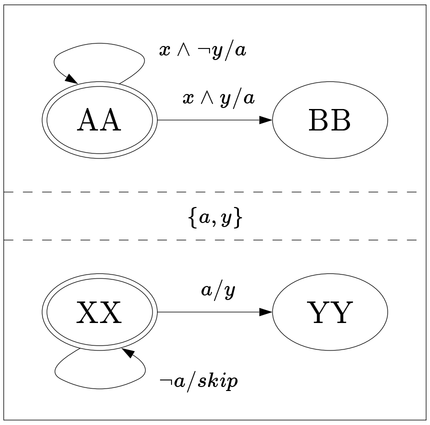


Figure 5.10: Instantaneous dialog

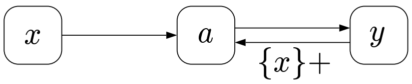


Figure 5.11: Signal graph for instantaneous dialog

We now have made the acquaintance of both harmful and harmless cycles. Stimulated by the examples, we define the notion of *causality error free cycles*. The cycle $Z \subseteq G$ of causality relations in the signal graph G is free of causality errors if the status (presence or absence) of each signal in the cycle can be non-ambiguously determined. Formally, one of the following cases has to be fulfilled:

- The labels of the cycle arcs are contradictory, that is

$$\bigcup_{p \in Z^+} p \cap \bigcup_{q \in Z^-} q \neq \emptyset$$

where Z^+ and Z^- are the set of all presence and absence sets, respectively, of arcs that are included in the cycle, that is, $Z^+ =_{df} \{p^+ \mid (s \rightarrow^{p^+}_{a^-} s') \in Z\}$ and $Z^- =_{df} \{a^- \mid (s \rightarrow^{p^+}_{a^-} s') \in Z\}$.

- Though the labels of the cycle arcs are not contradictory, that is

$$\bigcup_{p \in Z^+} p \cap \bigcup_{q \in Z^-} q = \emptyset$$

holds, at least one of the signals $\widetilde{s} \in \{s \mid (s \rightarrow^{p^+}_{a^-} s') \in Z\}$ in the cycle is an external input signal or is reachable from an external input signal $s^\star$ by a path $P \subseteq G$ that does not contain any label $p^+ \in P$ whose signals also occur in the cycle, that is, $P \cap Z^+ = \emptyset$.

This definition of harmless cycles is equivalent to the one given for Esterel [SBT96, Ber96, Tom97, Ber98]. Here, constructive Boolean logic is used for reasoning. A cycle in a combinational hardware circuit is harmless or, in other words, *constructive* whenever for any input constructive Boolean values for all outputs can be determined. It has been proven in [Shi96] that constructive circuits are equivalent to *electrically stable* circuits. A circuit is electrically stable if and only if its outputs are stable, i.e does not oscillate for any input and for arbitrary inertial delays [Tom97]. This correspondence is comparable with the Curry-Howard isomorphism between computations and proofs [GLT89].

It also has been proven that a circuit is constructive if and only if the value of any output in the least fixed point in Scott's Boolean domain of the circuit has no bottom element [Ber98]. In Scott's Boolean domain (presence, absence, and bottom), a bottom value of an output signal reflects the fact that it cannot be determined whether this signal is present or absent. Thus, signals with bottom values are equivalent to signals that cause causality errors in signal graphs. As a consequence, both notations, constructive, cyclic circuits in Esterel and causality error free cycles in signal graphs are equivalent. Hence, signal graphs do not only provide a means to visualize and analyze cycles by hand, but in addition a possibility for automated cycle verification. For these purposes, a μ-Charts specification has to be translated into an isomorphic Esterel program. This translation is not discussed in this thesis, but more background how this compilation in principle could look like can be found in publications of Modecharts [MPS95].

Further note that the signal graph that is constructed by computing the relations of type $s \rightarrow^{p^+}_{a^-} s'$ can be regarded as a Petri net with inhibitor edges (the transitions with circles in Figures 5.12 and 5.13) [Rei86]. The principle translation scheme is pictured in Figure 5.12, where $p^+ =_{df} \{s_1^+, \ldots, s_k^+\}$ and $a^- =_{df} \{s_1^-, \ldots, s_m^-\}$. Figure 5.13 shows an example; it is the Petri net interpretation for the signal graph of the ESTEREL program on Page 129. Interpreting signal graphs as Petri nets has the advantage that causality relations easily can be visualized, simulated, and verified by a number of already existing and accepted, often industrial, tools. The Petri net view may ease the access for software engineers through visual verification.

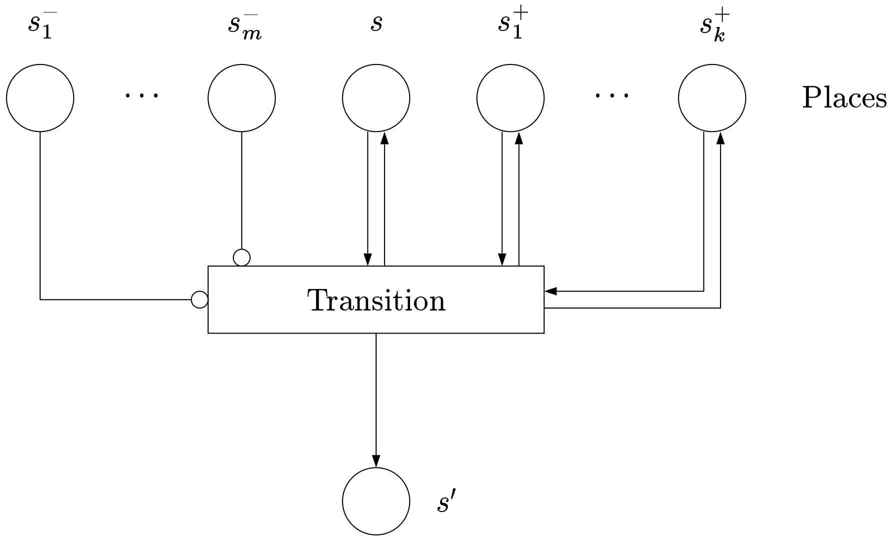


Figure 5.12: Petri net interpretation of the signal relation

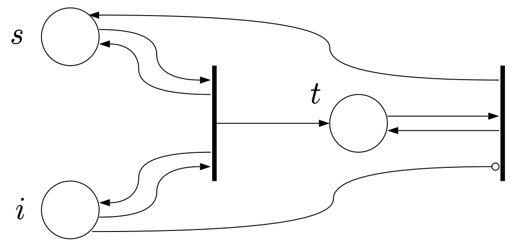


Figure 5.13: Signal graph viewed as Petri net (ESTEREL example)

The relation $\rightarrow$ presented in this section only deals with "positive" signal relationships, i.e. where the presence of a certain signal implies the presence of another signal. This relation does not tell us anything about the implications of the absence of a single signal. Though there is the possibility to express the absence of a set of signals in the label a^- in combination with the presence of at least one other signal, relationships like $\neg s \rightarrow s'$ are not feasible. Signal graphs can be constructed for monotonic functions only. In the subsequent section, we will discuss why it possibly leads to non-implementable specifications if the presence of a signal s' depends on the absence of a single signal s without any further positive dependency.

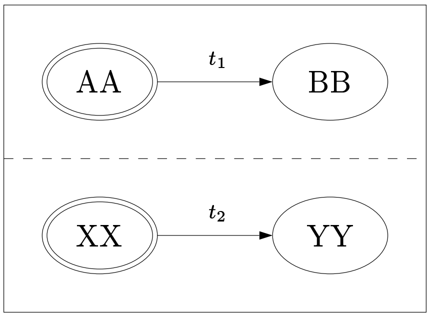


Figure 5.14: Absence of signals

5.2.4 Absence of Signals

One of the most important differences between synchronous and asynchronous languages is the concept of scanning the absence of signals in trigger conditions [BG97]. In asynchronous programs, it is only possible to express which reaction has to be performed when one or more certain signals are present. Scanning for the absence of (both external or internal) signals is not reasonable here, as asynchronous languages do not provide any notion of time that tells the system when to check for signal absence. In synchronous programs, however, this is easily possible using the concept of instants. Therefore, presence or absence only get a meaning with respect to one instant.

The possibility to trigger reactions by the absence of certain signals implies in particular that system components can even then react when no signal at all is present in the current instant. However, though such reactions do not cause any semantic problems in case of centralized implementation with a global clock they may heavily affect the behavior of distributed reactive systems. Figure 5.14 shows an example (set t_1 and t_2 to a/b and b/c, respectively). We assume that both automata are allocated to different electronic control units. The reader will agree that this example can easily be implemented using more than one control unit as we have an error-free causality flow from the environment signal a via the internal signal b to the output signal c.

The crucial point why this works is because the implementation of the lower automaton in Figure 5.14, which must receive the signal b from the implementation of the upper automaton in Figure 5.14, remains in its current state (and does not generate any output signal) until it indeed receives b.

Now, we slightly modify the label t_2 to $\neg b/c$ in Figure 5.14. The important difference to $t_2 = b/c$ is the fact that the corresponding automaton now does not change its current configuration if b is present and otherwise produces c while changing its configuration.

However, in this example, both automata cannot be allocated to different control units. The communication of signal b might be delayed for certain target architectures. In particular, messages can be delayed when using a common bus as a communication medium. Since each automaton only has a local view of the world, the lower automaton cannot predict that the upper is sending b in the current instant and that the message passing is just delayed. With this local view, it is undecidable whether b is delayed or has never been sent. Therefore, the lower automaton either might be deadlocked or it assumes that b is not present in the current instant. Both cases are critical and may yield a system malfunction and backtracking is not possible for reactive systems.

As a consequence, we require that a specification must not depend on the delay of certain signals in the case of distributed implementation. This requirement is formally denoted by:

$$\forall c, c' \in \mathcal{C}(S)\, \forall y \in \mathcal{O}(S) : (c', y) = st[\![S]\!](c, \emptyset) \Rightarrow (c = c' \wedge y = \emptyset)$$

Roughly speaking, this formula requires the brief condition "no signals, no actions". Notice, however, that this requirement does not exclude trigger conditions that contain negated signals entirely. It is straightforward to derive syntactic requirements from the above semantic condition.

From a theoretical point of view, we also could overcome the problem illustrated in this section by simply introducing signals that are sent from a component to indicate the absence of signals. Though it is possible to define a formal semantics for this, these signals increase the amount of data that has to be transmitted by the communication medium. Since the communication load is a critical point in realistic distributed reactive systems, this is, however, not an applicable solution.

5.2.5 Conclusion

Here, we summarize the results of Section 5.2. Before a μ-Chart specification can be partitioned on a processor network, a number of requirements have to be fulfilled. These concern:

- *External input signals:* the reaction must not depend on the scanning velocity of the input interfaces. As a consequence, only a very restricted number of Boolean predicates over external input signals are feasible as trigger conditions if no additional assumptions are made. Additional assumptions may increase the number of feasible conditions.

- *Internal input signals:* a causality analysis is needed. Causality graphs must not contain any cycles with causality errors. Recall that cycles in signal graphs with causality errors correspond to so-called harmful cycles in ESTEREL. As a consequence, apart from the graph-based analysis technique presented in this thesis, also the techniques implemented in the ESTEREL compiler v5 can be applied.

  Do not forget that we require the perfect synchrony hypothesis to be fulfilled not only for centralized implementations, but also for distributed ones. In practice, this means that also in case of distributed realization of a specification it must be guaranteed that the system reaction is faster than the environment. In particular, it must be guaranteed by the distributed implementation that all internal signals are multicast before new stimuli from the environment are available.

- *Both kinds of input signals:* a transition may not be triggered merely by the absence of one single signal. Whenever a transition shall be triggered by a negative signal, simultaneously the presence of at least one further signal must be sampled.

In Section 5.1, we have discussed pathological specifications and have made the experience that methodological design rules cannot be given for composition. We have seen that determinism is not closed under composition and that these examples can be divided into different categories.

Classification

In the remainder of this section, we classify μ-Chart specifications whose signal graphs contain cycles. The classification scheme was inspired by [Tom97] for combinational hardware. First of all, we can distinguish between specifications whose step semantics has a unique fixed point (1) and those with no fixed point (2.a) or more than one fixed point (2.b) in an instant (see the pathological examples in Sections 2.4.4 and 5.1). For all of them, we can define a formal stream semantics as presented in Section 2.4.3. The term "Boolean correct" here reflects the fact that there exists exactly one solution for the Boolean equations that describe the causality relation for communication of internal signals of composed charts (see also [Mar92]). If there is more than one or no solution at all, we speak of a "Boolean incorrect" causality relation. However, a Boolean correct specification possibly cannot be implemented correctly without any communication deadlocks. This is equivalent for combinational hardware with cycles as shown in [Tom97].

Specifications with more than one fixed point (2.b) can be further divided into those whose fixed points are ordered (2.b.ii) and those whose fixed points are not (2.b.i). Since monotonicity is a sufficient, but not necessary condition for the existence of a unique least fixed point the first class captures both monotonic (2.b.ii.A) and non-monotonic specifications (2.b.ii.B). The following list summarizes the situation:

1. Boolean correct = one fixed point

   (a) Contains no causality errors: these specifications remain for implementation.

   (b) Contains causality errors: these specifications cannot be implemented and therefore are rejected by causality analysis (of signal graphs).

2. Boolean incorrect

   (a) No fixed point: these specifications are rejected since $[\![S]\!] = \emptyset$. As an example, take the pathological chart with transition labels $t_1 = a/b$ and $t_2 = \neg b/a$ in Section 5.1.

(b) More than one fixed point

i. Non-ordered fixed points: the pathological chart in Section 5.1 with labels $t_1 = \neg a/b$ and $t_2 = \neg b/a$ gives an example. Specifications of this type are also rejected for lack of monotonicity.

ii. Ordered fixed points

A. Monotonic specification: these specifications are rejected because of harmful cycles. As an example, take the pathological chart with transition labels $t_1 = a/b$ and $t_2 = b/a$ in Section 5.1 (see Figure 5.9).

B. Non-monotonic specification: these specifications are rejected for lack of monotonicity.

Note that both classification and rejection can be done automatically by a compiler for synchronous languages. For instance, the ESTEREL v5 compiler rejects all programs that are not contained in class (1.a). While monotonicity is a comparatively easy to check property, causality analysis requires very complex and subtle algorithms, as demonstrated by ESTEREL.

So far, we have discussed problems that can occur when a μ-Charts specification is partitioned. The next phase after partitioning in our design process is allocation. Allocation means to determine physical locations for sequential automata and internal signals, that is, electronic control units and busses, respectively. The subsequent section addresses this task.

5.3 Allocation

In this section, we describe a model for the mathematical formalization of the allocation problem. For every single sequential automaton of the specification we have to fix its location on the target architecture, that is, the processor on which it will be implemented. In addition, we have to determine the inter-processor communication structure of the bus system.

Most reactive systems are embedded in complex environments, such as aircrafts and automobiles. In modern cars, several dozen electronic control units can be integrated. Mostly, these processors are dedicated to specific tasks as, for instance, locking or unlocking car doors. As some tasks only have to be performed from time to time, many of these processors have free capacities. This situation is not satisfactory, since each additional electronic control unit in the car increases both the risk of system malfunction and the product costs.

Therefore, it is challenging, due to technical and economical reasons, to use resources of an embedded system as efficiently as possible. However, specifications of reactive systems should be independent of a concrete target architecture. It is desirable that a specialist can specify the reactive system without knowing the target architecture of the embedded system at all. Another specialist then can determine this architecture and implement the original specification on it. While one engineer is responsible for abstract specification, refinement, and verification, for instance, another engineer performs partitioning, allocation, and implementation.

Our design process enables an independent procedure of this kind. In particular, it provides strategies for allocation that can be easily supported by a tool. Once again in this thesis, we recognize that a certain degree of formality pays: having a formal specification language at disposal eases the allocation process. Since μ-Charts are a precisely and formally defined language, we can give a formal model for allocation, too.

The method we apply for allocation is based on *integer linear programming (ILP)*. To use integer linear programs (ILPs) has two major advantages. First, ILPs are a convenient technique to formalize optimization problems with discrete solution space, and most software engineers are familiar with them. Second, there is tool support, for instance [Sav92, Cpl94], to calculate the optimal solution for a problem formalized as an ILP.

The problem of assigning a set of tasks spatially (and often also temporally), usually with the aim of optimizing an objective function, is well-known in the science community for more than 25 years. The problem complexity depends on several aspects; [GJ79] and [KN84] provide an overview. In this thesis, we assume a predefined and fixed number of more than one processor and more than one bus. Therefore, the allocation problem discussed here falls in the class of *NP-hard* [Weg93] problems. Nevertheless, for small and medium-sized instances of optimization problems optimal solutions can be computed by tool support [Sch94, Har95]. For problem instances with a large number of processors and tasks, however, the global optimum of the objective function often cannot be computed even by fast machines within hours. In these cases, there is another way out: heuristics can be used standalone or in combination with linear programming techniques to compute (almost) optimal solutions [Sch94, Har95, SH97].

5.3.1 Preliminaries

In this section, we outline the target hardware architecture with which we realize our specifications with μ-Charts. The architecture we have chosen is an abstraction of the realistic architecture that, for instance, car manufactures typically use for modern upper class cars. By means of this architecture, we describe parameters and variables of the ILP to formalize the allocation problem. The reader should note that our model for allocation can be modified in future work. It is thinkable that further model restrictions are added, already existing ones are removed, the objective function is modified, or even completely different target architectures are used.

The abstract target architecture used in this thesis is outlined in Figure 5.15 [Fuc97]. Here $ECU_1, \ldots, ECU_N$ denote the N electronic control units that are part of the em-

K ECUs:	$ECU_1, \ldots, ECU_k, \ldots, ECU_K$
M Sensors:	$Sen_1, \ldots, Sen_m, \ldots, Sen_M$
N Actors:	$Act_1, \ldots, Act_n, \ldots, Act_N$
L Busses:	$Bus_1, \ldots, Bus_l, \ldots, Bus_L$

Table 5.5: Target architecture parameters

bedded system and are available for the allocation of sequential automata. These control units can be realized by microprocessors, field programmable logic gate arrays (FPGAs), application specific integrated circuits (ASICs), and so on. In the remainder of this section, we will use the notions "processor" and "(electronic) control unit" synonymously. It is part of our model assumptions that one or more sequential automata of the overall specification are allocated to these processors. Furthermore, the architecture contains a system of L independent busses $Bus_1, \ldots, Bus_L$. Control units may be connected through one or more different busses. To answer the question which ECUs are finally connected with which busses is part of the optimization problem. All internal messages are broadcast by this bus system.

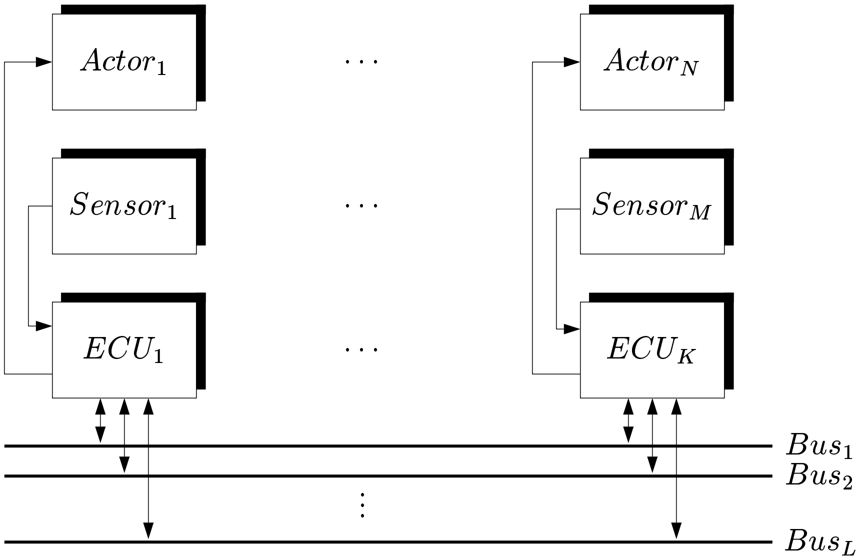


Figure 5.15: Target architecture

Busses are treated in our approach — like processors — as scarce resources which probably have to be shared by many communications. Therefore, in addition to the processor allocation, we have to fix the bus allocation.

Moreover, also M sensors $Sensor_1, \ldots, Sensor_M$ and N actors $Actor_1, \ldots, Actor_N$ are part of the architecture. External signals are received from and sent to sensors and actors, respectively. Table 5.5 summarizes the situation.

We assume that the specification that shall be implemented already has been transformed in the μ-Charts normal form (see Equation 2.2). The result of this process is pictured in Figure 5.16, where in principle the structure of Figure 2.10 is reflected. In this normal form, each specification has the shape $\triangle_L(S_1\|\cdots\|S_J)$, where $S_1,\ldots,S_J$ denote all automata that are part of the specification. According to the Theorem 2.4.4 for signal extrusion, the process of establishing the normal form for a specification can be automated. To this end, the theorem has to be applied from the inside to the outside. This means that the original specification must be transformed such that communication is only possible on the outermost composition level. Though it would be theoretically possible to leave the specification in its original form, this prospectively would reduce the number of allocation alternatives as the composition operator assumes a “private” message exchange between the composed charts. Only specifications that are available in the normal form provide the full spectrum of all design alternatives that are offered by the target architecture. All internal signals that occur in the specification are summarized in the set

$$\{C_1,\ldots,C_H\} =_{df} \bigcup_{j=1}^{J} In(S_j) \cap L \cap \bigcup_{j=1}^{J} Out(S_j) \subseteq L$$

Here, each C_h, where $1 \leq h \leq H$, denotes one internal signal. The set $\{C_1,\ldots,C_H\}$ is abbreviated to Com_{Int}. Table 5.6 summarizes the software model, that is, those units that shall be implemented on the target architecture.

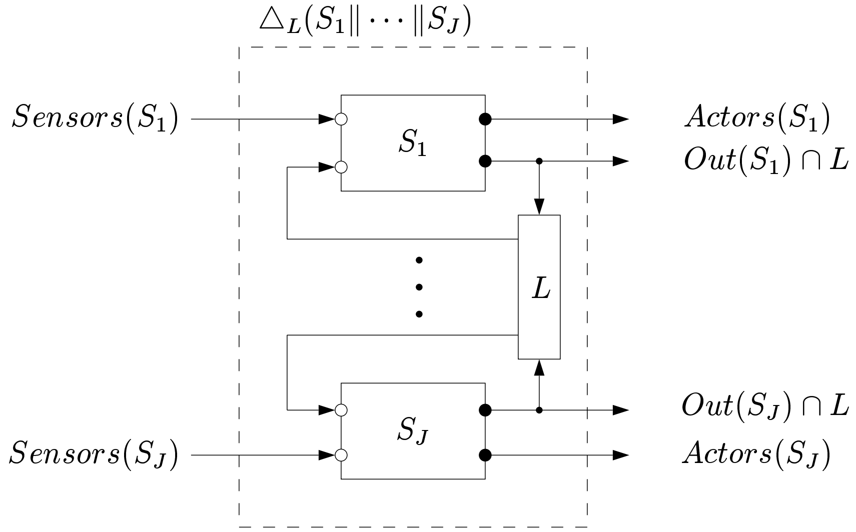


Figure 5.16: Normal form

Besides internal communications between processors, also communications with actors

J Sequential Automata:	$S_1, \ldots, S_j, \ldots, S_J$
H Internal Signals:	$C_1, \ldots, C_h, \ldots, C_H$

Table 5.6: Software model

and sensors are possible:

$$Com_{Act} =_{df} \bigcup_{1 \leq j \leq J} \bigcup_{A \in Actors(S_j)} Com(S_j, A)$$
$$Com_{Sen} =_{df} \bigcup_{1 \leq j \leq J} \bigcup_{S \in Sensors(S_j)} Com(S_j, S)$$

where $Com(S_j, A)$ and $Com(S_j, S)$ denote those messages that are sent from S_j to actor A and sensor S to S_j, respectively. All signals that are possibly broadcast or sent to and received from the actors and sensors, respectively, are summarized in the signal set Com:

$$Com = Com_{Int} \cup Com_{Act} \cup Com_{Sen}$$

Allocation Variables

In this section, we describe the 0/1 variables we use to formalize the optimization problem as an integer linear program. Formally, an *allocation* is a tuple

$$((x_{kj})_{1 \leq k \leq K, 1 \leq j \leq J}, (y_{lh})_{1 \leq l \leq L, 1 \leq h \leq H})$$

Here, the 0/1 variable x_{kj} equals one if and only if the automaton S_j is allocated to the control unit ECU_k and zero otherwise. The task of the 0/1 variable y_{lh} is similar; it equals one if and only if the communication of signal C_h is allocated to bus Bus_l.

To calculate the hardware costs, that is, costs for processors and busses that accrue when constructing the target architecture for a particular specification, we use the following abbreviations:

$$\widetilde{x}_k = 1 \iff_{df} \exists 1 \leq j \leq J : x_{kj} = 1$$
$$\widetilde{y}_l = 1 \iff_{df} \exists 1 \leq h \leq H : x_{lh} = 1$$

We will discuss how to bring this definition in a form that is appropriate for integer linear programming later in this section. Table 5.7 summarizes all ILP variables used for the formalization of the allocation problem.

Allocation Constants

In order to formalize the allocation problem as an integer linear program we do not only need to define the 0/1 variables that span the solution space or, in other words, the

x_{kj}:	Automaton S_j is allocated to control unit ECU_k
$\widetilde{x}_k$:	At least one automaton is allocated to control unit ECU_k
y_{lh}:	Internal signal C_h is allocated to bus Bus_l
$\widetilde{y}_l$:	At least one communication is allocated to bus Bus_l

Table 5.7: Variables

$dist(ECU_k, Sen_m)$:	Distance from ECU_k to Sen_m
$dist(ECU_k, Act_n)$:	Distance from ECU_k to Act_n
$dist(ECU_k, Bus_l)$:	Distance from ECU_k to Bus_l
$cost(ECU_k)$:	Costs if ECU_k is used in the design
$cost(Bus_l)$:	Costs if Bus_l is used in the design
$speed(Bus_l)$:	Transport speed of Bus_l
$weight(C_h)$:	Weight of internal signal C_h
$memory(S_j)$:	Memory consumption of automaton S_j
$load(S_j)$:	Work load consumption of automaton S_j
$Storage(ECU_k)$:	Available memory of ECU_k
$Capacity(ECU_k)$:	Available computation capacity of ECU_k
$Busload(Bus_l)$:	Available transportation power of Bus_l

Table 5.8: Constants

polytope that contains all feasible solutions, but in addition have to denote the design parameters (see Table 5.8). They are included in the model as constants and must be defined by the user.

In the target architecture we use, we assume that control units and sensors and actors, respectively, are spatially separated and not implemented on a single chip. As a consequence, they have to be connected by an extra and potentially complex wiring. Actors or sensors that are attached at badly accessible locations like inside the doors, for instance, possibly cause an expensive wiring. These costs have to be considered in the allocation model.

The distances from ECU_k to Sen_m, Act_n, and Bus_l are denoted by the constants $dist(ECU_k, Sen_m)$, $dist(ECU_k, Act_n)$, and $dist(ECU_k, Bus_l)$, respectively. Notice, however, that in this context “distance” not necessarily has to be the Euklidian distance. Since also for components that are mounted fairly close with respect to this metric, the wiring in between also can be extremely costly as some locations may be difficult to be accessed. Hence, the notion “distances” here rather expresses connection costs than Euklidian distances.

Many embedded systems are produced in mass production, where even small cost factors can heavily influence the product price. Here, only one single additional hardware unit possibly makes the overall product too expensive. Thus, costs for hardware units also have been included in our model for allocation.

Remember that the 0/1 variables $\widetilde{x}_k$ and $\widetilde{y}_l$ equal to one whenever the electronic control unit ECU_k and the bus Bus_l, respectively are used for the entire design and therefore must not be added to the objective function more than once. Hence, $\widetilde{x}_k \cdot cost(ECU_k)$ and $\widetilde{y}_l \cdot cost(Bus_l)$ have to be added to the overall allocation cost function.

In Table 5.8, $cost(ECU_k)$ and $cost(Bus_l)$ denote the price of one unit of processor ECU_k and bus Bus_l, respectively. These cost factors only accrue once in the entire design. They express the amount of money that the embedded system becomes more expensive when realizing the system including ECU_k and Bus_l, respectively.

The constant $speed(Bus_l)$ denotes the transmission rate of Bus_l. For instance, the transmission rate of the CAN bus is up to 1 MBaud per second (for a distance of maximal 40 meters) [Ets94]. However, the transmission time for a signal does not only depend on the transmission rate of the medium, but also on the amount of data that has to be transported. The latter is taken into account by the constant $weight(C_h)$. In our special case, there is just one signal type: pure signals. However, as we have discussed in Section 2.4.3, it would be no problem to introduce further signal types, which possibly require different capacities to be stored and transferred. Whether these constants are of consequence also depends on the maximal message length that can be used on the bus. The CAN bus protocol [Ets94], for instance, allows maximally 111 bits for effective use. The constant $Busload(Bus_l)$ reflects the total bus capacity that is available on Bus_l.

Whenever an automaton S_j is realized on an electronic control unit, it consumes a certain amount of memory and computation power. Both factors are expressed by the constants $memory(S_j)$ and $load(S_j)$, respectively. Since both memory and computation power of an electronic control unit is restricted, also the upper memory limit $Storage(ECU_k)$ and the upper computation power capacity $Capacity(ECU_k)$ have to be taken into account when formalizing the allocation problem.

5.3.2 Objective Function

In this section, we discuss the objective function, which has to be minimized. Mathematically, this means to find a valuation for the problem variables such that the value of the object function is minimal. The objective function expresses the overall costs that have to be expended for the distributed design. This cost function is composed by a number of individual factors. In our model, we distinguish the following design costs.

First of all, the design costs depend on the number and individual costs of all electronic control units that are used for the specific development. A control unit ECU_k is used in a design whenever at least one automaton is allocated to ECU_k. Once one automaton is allocated to ECU_k, further allocations of other automata do not influence this cost component any more. Thus, these costs contain the price of the control unit (which have to be payed as soon as the unit is built into the embedded system), the installing costs, and so on.

However, these costs are an optional part of the objective function. It might be the case that ECU_k is not only used to run the specific system under development, but already is known to be used of other components of the embedded system that are not part of this system, too. For instance, if the regarded control unit to which a part of, say, the central locking system of a car shall be allocated to is already used by the injection control, we must not count these costs twice. In this case, this part of the objective function has to be omitted.

If, however, included in the model, this first part of the cost function is denoted as the sum of the individual costs of all electronic control units:

$$\sum_{k=1}^{K} cost(ECU_k)\widetilde{x}_k =_{df} G_1$$

A similar observation can be made for the individual costs of a bus that is used in the design. Like the aforementioned costs, these costs are also optional:

$$\sum_{l=1}^{L} cost(Bus_l)\widetilde{y}_l =_{df} G_2$$

In embedded systems, connections between control units and actors and sensors, respectively, cause considerable costs. If a control unit is connected to a sensor, for example, mains have to placed between sensors and controllers, which are possibly far distant from each other. How expensive it is to connect them mainly depends on two facts: first, how large the distance between the two is and second, how "complicated" it is to connect them. For instance, it is not complicated to connect electronic units in a car's cockpit, but it is cumbersome to do this if mains have to be layed to badly accessible locations like the window motors in the car doors.

These parameters are subsumed in the constants $dist(ECU_k, Sen_n)$ and $dist(ECU_k, Act_m)$ between control unit ECU_k and sensor Sen_n and actor Act_m, respectively. Whether a connection between ECU_k and sensor Sen_n and actor Act_m depends on the input and output interfaces of the automata that are allocated to ECU_k. If we assume that the automaton S_j is allocated to this control unit, mathematically expressed by $x_{kj} = 1$, the specific connection costs are denoted by

$$\sum_{Act \in Actors(S_j)} dist(ECU_k, Act) \cdot weight(Com(S_j, Act)) =_{df} Z_1(k, j)$$

plus

$$\sum_{Sen \in Sensors(S_j)} dist(ECU_k, Sen) \cdot weight(Com(S_j, Sen)) =_{df} Z_2(k, j)$$

where $weight(Com(S_j, Act)) =_{df} \sum_{C \in Com(S_j,Act)} weight(C)$ and $weight(Com(S_j, Sen))$ $=_{df} \sum_{C \in Com(S_j,Sen)} weight(C)$, respectively. The overall connection costs now are calculated as follows:

$$\sum_{k=1}^{K} \sum_{j=1}^{J} x_{kj}(Z_1(k,j) + Z_2(k,j)) =_{df} G_3$$

The costs considered so far only include costs for hardware, like control units, sensors, actors, and connections between them. However, in a distributed implementation of a control system, the control units do not only communicate with actors or sensors, but also among one another. As these communications are carried out by busses, which are a restricted resource, also costs for communications have to be borne in mind during allocation. Different busses with different transport speeds may be available. Thus, the overall communication costs are denoted by:

$$\sum_{l=1}^{L} \sum_{h=1}^{H} y_{lh} \frac{weight(C_h)}{speed(Bus_l)} =_{df} G_4$$

In addition, control units and busses have to be connected:

$$\sum_{l=1}^{L} \sum_{h=1}^{H} y_{lh}(dist(ECU_{Sender(C_h)}, Bus_l) + dist(ECU_{Receiver(C_h)}, Bus_l))weight(C_h) =_{df} G_5$$

where $Sender(C_h)$ and $Receiver(C_h)$ denote sender and receiver of message C_h, respectively. The overall design costs are now defined by the sum of all above listed costs:

$$\sum_{i=1}^{5} \alpha_i G_i$$

where $\alpha_1, \ldots, \alpha_5$ are positive real constants less than one with $\sum_{i=1}^{5} \alpha_i = 1$. These parameters are used to express the mutual relationship of the single G_is. A value close to 1 of α_i means that more emphasis is put on the minimization of G_i than on the other sub-cost functions and vice versa.

5.3.3 Restrictions for the Model without Multiple Computation

Minimizing the above cost function, a number of side conditions that restrict the free play of design alternatives, i.e. possibilities for allocation, have to be met. In this section, we will discuss relevant side conditions.

For each single electronic control unit ECU_k we can give the memory and computation capacity still available, which are denoted by $Storage(ECU_k)$ and $Capacity(ECU_k)$, respectively. Developing a distributed system using these architecture components, care has to be taken that both upper limits are not exceeded. This means formally:

$$\forall 1 \leq k \leq K : \sum_{j=1}^{J} x_{kj} \cdot memory(S_j) \leq Storage(ECU_k)$$

and

$$\forall 1 \leq k \leq K : \sum_{j=1}^{J} x_{kj} \cdot load(S_j) \leq Capacity(ECU_k)$$

This kind of restriction does not only have to be met for control units, but also for communication media; busses have a certain capacity up to which they can transport data. This capacity must not be exceeded as otherwise the transport delays are so long that the system communication breaks down. For some communication nets even a bus load of 20 percent is considered as overload. Hence, we require that:

$$\forall 1 \leq l \leq L : \sum_{h=1}^{H} y_{lh} \cdot weight(C_h) \leq Busload(Bus_l)$$

Furthermore, in our model, we require that every single automaton is allocated to exactly one electronic control unit. The mathematical formulation for this reads as follows:

$$\forall 1 \leq j \leq J : \sum_{k=1}^{K} x_{kj} = 1 \tag{5.3}$$

In the next section, we will come back to the question how the restrictions have to be re-formulated if an automaton can be allocated to more than one control unit, i.e. can be multiply "computed" [NT93]:

$$\forall 1 \leq j \leq J : \sum_{k=1}^{K} x_{kj} \geq 1 \tag{5.4}$$

In this context, we speak of the notion of *multiple computation*. Multiple computation of one automaton means that this automaton is implemented and executed on several control units to minimize communication overhead and to exploit more parallelism. For a detailed introduction see [NT93].

Up to now, all restrictions could be formulated straightforwardly. In the sequel, however, we will focus upon slightly more subtle side conditions. We have expressed what it means that an automaton is allocated to exactly one (5.3) or at least one (5.4) processor. Now, we will discuss the allocation of communication messages to busses. Remember that all possible inter-processor communication of the system under development is subsumed in the set Com_{Int}. Then the transmission of signal $C_h \in \{C_1, \ldots, C_H\} = Com_{Int}$ is allocated to Bus_l whenever the corresponding variable y_{lh} is set to one.

However, communications are cost intensive as we can see in the objective function. Therefore, a communication only should be carried out when necessary and useless communications should be avoided. A communication is needed if and only if both sender and receiver of the message are allocated to different control units. Otherwise, an inter-processor communication via a bus is not necessary. Though this requirement sounds evident, we have to formulate a mathematical condition in the integer linear program. Therefore, if both sender and receiver of message C_h are allocated to different units, exactly one bus has to be allocated for its transmission:

$$\forall 1 \leq h \leq H : \sum_{l=1}^{L} y_{lh} = 1$$

Otherwise, if sender and receiver of C_h are allocated to a single processor, no inter-processor communication is needed since intra-processor communication is sufficient. In this case, we require that no bus at all has to be allocated for the signal's transmission:

$$\forall 1 \leq h \leq H : \sum_{l=1}^{L} y_{lh} = 0$$

However, there are still two problems left. First, we have to formulate mathematically what it means that "two automata are located on different control units". Second, we have to include the above case analysis into one equation as this is the only way to fix it in an integer linear program.

Let us concentrate on the first problem. Two automata S_i and S_j with $i \neq j$ are allocated to the same control unit whenever there exists a unit ECU_k such that $x_{ki} = 1$ and $x_{kj} = 1$. Dealing with a finite number of control units, this existential quantification can be expressed by the following equation:

$$\sum_{k=1}^{K} x_{ki} x_{kj} = 1$$

Otherwise, we get

$$\sum_{k=1}^{K} x_{ki} x_{kj} = 0$$

It now remains to be demonstrated how this case analysis can be expressed by one equation:

$$\forall 1 \leq h \leq H : \sum_{l=1}^{L} y_{lh} = 1 - \sum_{k=1}^{K} x_{k,Sender(C_h)} x_{k,Receiver(C_h)}$$

So far, nothing has been said how the variables $\widetilde{x}_k$ and $\widetilde{y}_l$ have to be restricted. It has to be defined formally what it means that $\widetilde{x}_k = 1$ if and only if at least one automaton is allocated to ECU_k, expressed by:

$$\forall 1 \leq k \leq K : \sum_{j=1}^{J} x_{kj} \geq 1$$

Do not confuse this with Restriction 5.4. The case analysis can be combined to one single equation:

$$\forall 1 \leq k \leq K : \widetilde{x}_k = \left\lceil \frac{\sum_{j=1}^{J} x_{kj}}{J} \right\rceil$$

Note, however, that the ceil operator $\lceil . \rceil$ is no valid operation in a linear program. Thus, we have to re-formulate this equation by the aid of the following proposition.

Proposition 5.3.1
Let $\widetilde{x}_k$ and x_{kj} be 0/1 variables for $1 \leq k \leq K$ and $1 \leq j \leq J$. Then the following is true:

$$\widetilde{x}_k = \left\lceil \frac{\sum_{j=1}^{J} x_{kj}}{J} \right\rceil \quad \Longleftrightarrow \quad \frac{\sum_{j=1}^{J} x_{kj}}{J} \leq \widetilde{x}_k \leq \sum_{j=1}^{J} x_{kj}$$

Proof 16 The proof is done by case analysis:

1. First, if we assume that $\sum_{j=1}^{J} x_{kj}$ equals zero then we get on the left-hand-side of the equivalence $\widetilde{x}_k = 0$ and on the right-hand-side $\frac{0}{J} \leq \widetilde{x}_k \leq 0$, i.e. both sides coincide.
2. If, however, $\sum_{j=1}^{J} x_{kj} \geq 1$ then we have on the one side $\left\lceil \frac{\sum_{j=1}^{J} x_{kj}}{J} \right\rceil = 1$ and on the other side $0 < \frac{1}{J} \leq \widetilde{x}_k \leq J$. As zero and one are the unique values that are feasible for a 0/1 variable, $\widetilde{x}_k$ is restricted to one by the right-hand-side of the above equivalence. $\square$

Applying this theorem to the $\widetilde{y}_l$, the restrictions read similar:

$$\forall 1 \leq l \leq L : \frac{\sum_{h=1}^{H} y_{lh}}{H} \leq \widetilde{y}_l \leq \sum_{h=1}^{H} y_{lh}$$

The proof is similar to the above proof for $\widetilde{x}_k$. Last not least there is the more technical requirement that all variables are of type 0/1 only:

$$\begin{aligned} \forall 1 \leq k \leq K \, \forall 1 \leq j \leq J : &\quad x_{kj} \in \{0,1\} \\ \forall 1 \leq l \leq L \, \forall 1 \leq h \leq H : &\quad y_{lh} \in \{0,1\} \\ \forall 1 \leq l \leq L : &\quad \widetilde{y}_l \in \{0,1\} \\ \forall 1 \leq k \leq K : &\quad \widetilde{x}_k \in \{0,1\} \end{aligned}$$

$$
\begin{array}{rl}
\multicolumn{2}{c}{\text{Minimize } \sum_{i=1}^{5} \alpha_i G_i \text{ under the conditions:}} \\
\hline
\forall 1 \le k \le K: & \sum\limits_{j=1}^{J} x_{kj} \cdot memory(S_j) \le Storage(ECU_k) \\
\forall 1 \le k \le K: & \sum\limits_{j=1}^{J} x_{kj} \cdot load(S_j) \le Capacity(ECU_k) \\
\forall 1 \le l \le L: & \sum\limits_{h=1}^{H} y_{lh} \cdot weight(C_h) \le Busload(Bus_l) \\
\forall 1 \le j \le J: & \sum\limits_{k=1}^{K} x_{kj} = 1 \\
\forall 1 \le h \le H: & \sum\limits_{l=1}^{L} y_{lh} = 1 - \sum\limits_{k=1}^{K} x_{k,Sender(C_h)} x_{k,Receiver(C_h)} \\
\forall 1 \le k \le K: & \frac{\sum_{j=1}^{J} x_{kj}}{J} \le \widetilde{x}_k \le \sum\limits_{j=1}^{J} x_{kj} \\
\forall 1 \le l \le L: & \frac{\sum_{h=1}^{H} y_{lh}}{H} \le \widetilde{y}_l \le \sum\limits_{h=1}^{H} y_{lh} \\
\forall 1 \le k \le K \, \forall 1 \le j \le J: & x_{kj} \in \{0,1\} \\
\forall 1 \le l \le L \, \forall 1 \le h \le H: & y_{lh} \in \{0,1\} \\
\forall 1 \le l \le L: & \widetilde{y}_l \in \{0,1\} \\
\forall 1 \le k \le K: & \widetilde{x}_k \in \{0,1\} \\
\hline
\end{array}
$$

Table 5.9: Integer linear program

Table 5.9 summarizes the integer linear program for the allocation model without using multiple computations of single automata on different processors. The next section is dedicated to the model including multiple computations. As this model enables more design alternatives it also offers a larger spectrum of allocation alternatives and therefore to find an optimal solution of the objective function is more “complex”, although we do not leave the class of NP-hard problems.

5.3.4 Restrictions for the Model with Multiple Computation

The transmission speed of a bus may be low, its load high, and the connection costs to the corresponding control units also high. Thus, communications can be delayed and costs for extensive use of bus resources may be high. One possible workaround could simply be to use another, faster and less overloaded bus with possibly cheaper connection costs. However, for this bus the situation could be similar and possibly no other, more appropriate bus can be found.

Fortunately, there is a further way out: a lower overall cost value could be achieved by repeating the computation of the sending or receiving automaton. However, this multiple computation of single automata has one draw-back: the number of possible solutions of the design problem increases and the search for the optimum therefore gets more time consuming.

Using the concept of multiple computation, some of the above restrictions have to be re-formulated. First of all, this principle does not require any longer the validity of the equation

$$\forall 1 \leq j \leq J : \sum_{k=1}^{K} x_{kj} = 1$$

Instead, this condition has to be substituted by

$$\forall 1 \leq j \leq J : \sum_{k=1}^{K} x_{kj} \geq 1$$

In contrast to Section 5.3.3 the following inequations are no longer true

$$\forall 1 \leq i, j \leq J : i \neq j \Rightarrow 0 \leq \sum_{k=1}^{K} x_{ki} x_{kj} \leq 1$$

as there can be more than one k such that $x_{ki}x_{kj}$ equals one. Thus, the number of possible solutions of the allocation problem increases. Each message $C_h \in Com_{Int}$ now possibly has to be transferred on more than one bus as there may be more than one control unit on which senders and/or receivers are allocated. The above formula is modified to:

$$\forall 1 \leq i, j \leq J : i \neq j \Rightarrow 0 \leq \sum_{k=1}^{K} x_{ki} x_{kj} \leq K$$

Since the above sum is naturally restricted by K, we also could omit the upper limit without any consequences for the optimal solution. Moreover, we also replace

$$\forall 1 \leq h \leq H : \sum_{l=1}^{L} y_{lh} = 1 - \sum_{k=1}^{K} x_{k,Sender(C_h)} \cdot x_{k,Receiver(C_h)}$$

by

$$\forall 1 \leq h \leq H : \sum_{l=1}^{L} y_{lh} \geq 1 - \left\lceil \frac{\sum_{k=1}^{K} x_{k,Sender(C_h)} \cdot x_{k,Receiver(C_h)}}{K} \right\rceil$$

Recall that the ceil operator cannot be used in linear programs. Hence, we rewrite this restriction to:

$$\forall 1 \leq h \leq H : \sum_{l=1}^{L} y_{lh} \geq 1 - \xi_h$$

Minimize $\sum_{i=1}^{5} \alpha_i G_i$ under the conditions:	
$\forall 1 \leq k \leq K:$	$\sum_{j=1}^{J} x_{kj} \cdot memory(S_j) \leq Storage(ECU_k)$
$\forall 1 \leq k \leq K:$	$\sum_{j=1}^{J} x_{kj} \cdot load(S_j) \leq Capacity(ECU_k)$
$\forall 1 \leq l \leq L:$	$\sum_{h=1}^{H} y_{lh} \cdot weight(C_h) \leq Busload(Bus_l)$
$\forall 1 \leq j \leq J:$	$\sum_{k=1}^{K} x_{kj} \geq 1$
$\forall 1 \leq h \leq H:$	$\sum_{l=1}^{L} y_{lh} \geq 1 - \xi_h$
$\forall 1 \leq h \leq H:$	$\frac{\sum_{k=1}^{K} x_{k,Sender(C_h)} \cdot x_{k,Receiver(C_h)}}{K} \leq \xi_h$
$\forall 1 \leq h \leq H:$	$\xi_h \leq \sum_{k=1}^{K} x_{k,Sender(C_h)} \cdot x_{k,Receiver(C_h)}$
$\forall 1 \leq k \leq K:$	$\frac{\sum_{j=1}^{J} x_{kj}}{J} \leq \widetilde{x}_k \leq \sum_{j=1}^{J} x_{kj}$
$\forall 1 \leq l \leq L:$	$\frac{\sum_{h=1}^{H} y_{lh}}{H} \leq \widetilde{y}_l \leq \sum_{h=1}^{H} y_{lh}$
$\forall 1 \leq k \leq K \, \forall 1 \leq j \leq J:$	$x_{kj} \in \{0,1\}$
$\forall 1 \leq l \leq L \, \forall 1 \leq h \leq H:$	$y_{lh} \in \{0,1\}$
$\forall 1 \leq l \leq L:$	$\widetilde{y}_l \in \{0,1\}$
$\forall 1 \leq k \leq K:$	$\widetilde{x}_k \in \{0,1\}$

Table 5.10: Integer linear program (multiple computations allowed)

where the ξ_h are auxiliary variables, which are computed as follows (the proof is equivalent to the proof of Proposition 5.3.1):

$$\forall 1 \leq h \leq H: \frac{\sum_{k=1}^{K} x_{k,Sender(C_h)} \cdot x_{k,Receiver(C_h)}}{K} \leq \xi_h \leq \sum_{k=1}^{K} x_{k,Sender(C_h)} \cdot x_{k,Receiver(C_h)}$$

It is worthwhile to ask why it is sufficient to denote a lower, but not an upper bound for $\sum_{l=1}^{L} y_{lh}$. One could think that omitting an upper bound would yield to a number of useless communications. However, as the variables of type y_{lh} are part of the objective function they are minimized anyway and no further restriction is needed. Table 5.10 summarizes the integer linear program for the allocation problem including multiple computations.

With the results of the last two sections we are now able to fix the physical locations of sequential automata. If two or more automata that interact through message passing are allocated to different control units, in addition a bus that establishes the transfer of this message has to be fixed. Both allocation problems can be mathematically formalized by the integer linear program presented in this section. By tool assistance, even an optimal solution can be found if the number of design alternatives is not too large.

In Section 5.1, we have seen that multicasting amounts to solving a fixed point construction. Thus, whenever components that broadcast are allocated to distributed processors, a distributed fixed point computation is needed. However, neither that this distributed computation stabilizes nor that the computed fixed point is identical to the one computed in the case of a centralized implementation is obvious. The next section addresses this problem.

5.4 Scheduling of Communications

Executing multicasting in the case of distributed implementation of the sending and receiving components amounts to solving a distributed fixed point computation in one instant. Since these components may be implemented on control units with different computation power and therefore also with different reaction times, it is by no means obvious whether this distributed computation yields the same fixed point as the original, centralized fixed point generation. In particular, Kleene's iteration is no longer an appropriate model to express fixed point computation on a distributed target architecture. Rather, we need a chaotic, that is, non-incremental fixed point generation. "Chaotic" here means that we cannot make any assumption on the relative speeds of the control units and therefore cannot predict the relative execution orders of parts of the function for which the fixed point has to be computed. Note that the notion of "chaos" here is adopted from [Edw97] and is not related at all to "chaos completion" as discussed in the first part of this thesis. In this section, we will see that indeed centralized and distributed fixed point computations coincide. Since we give a formal proof, we first have to provide the mathematical machinery for this.

5.4.1 Preliminaries

Executing a μ-Chart specification in one instant possibly induces the execution of a chain of communications. As we have already analyzed in Section 5.1, this mathematically means to compute the fixed point iteration of the following function:

$$\lambda z.\pi_1(st[\![S_1]\!](c_1, x|_{In(S_1)\setminus L} \cup z|_{In(S_1)\cap L})) \cup \pi_1(st[\![S_2]\!](c_2, x|_{In(S_2)\setminus L} \cup z|_{In(S_2)\cap L})) \qquad (5.5)$$

which we abbreviate to f_x. As we want to implement our distributed reactive systems as efficiently as possible, we clearly aim at minimizing the number of iterations needed to compute this fixed point. In the sequel, we will discuss to what extent this is possible. First, however, we have to define some mathematical notations that are a prerequisite for the understanding of this section. The following definitions and propositions are taken from [Win93]. The corresponding proofs also can be looked up there. Since [Win93] provides an excellent overview over the related theory we repeat only very briefly the essential mathematical concepts to understand the central statement of this section.

Definition 5.4.1 (Partial Order)
A *partial order* is a set P on which there is a relation $\sqsubseteq$ that is:

1. Reflexive: $\forall p \in P : p \sqsubseteq p$
2. Transitive: $\forall p, q, r \in P : p \sqsubseteq p \wedge q \sqsubseteq r \Rightarrow p \sqsubseteq r$
3. Antisymmetric: $\forall p, q : p \sqsubseteq q \wedge q \sqsubseteq p \Rightarrow p = q$ □

Definition 5.4.2 ((Least) Upper Bound)
For a partial order $(P, \sqsubseteq)$ and a subset $Q \subseteq P$ an *upper bound* of Q is an element $p \in P$ such that

$$\forall q \in Q : q \sqsubseteq p$$

A *least upper bound* of Q is an element p such that p is an upper bound of Q and for all upper bounds q of Q we have $p \sqsubseteq q$. □

Definition 5.4.3 (Chain)
A *chain* is a totally-ordered set $C \subseteq P$ of a partial order $(P, \sqsubseteq)$, i.e. for all $c_1, c_2 \in C$ either $c_1 \sqsubseteq c_2$ or $c_2 \sqsubseteq c_1$ holds. □

Definition 5.4.4 (Complete Partial Order (CPO))
The partial order $(P, \sqsubseteq)$ is a *complete partial order (CPO)* if every chain in P has a least upper bound. □

Note that a finite chain always has a unique least upper bound. Furthermore, a partial order which contains only finite chains is a complete partial order.

Definition 5.4.5 (CPO with Bottom)
We say that $(P, \sqsubseteq)$ is a *CPO with bottom* if it is a CPO which has a least element $\perp$, termed "bottom". □

Note that the power domain [Win93] $(\wp(X), \subseteq)$ with X finite is a complete partial order with bottom $\emptyset$. Further note that a bottom element, if it exists, is unique.

Definition 5.4.6 (Monotonic/Continuous Function)
Let D and E be complete partial orders. Then a function $f : D \to E$ is a *monotonic function* if and only if:

$$\forall d_1, d_2 \in D : d_1 \sqsubseteq d_2 \Rightarrow f(d_1) \sqsubseteq f(d_2)$$

The function f is in addition *continuous* if and only if it is monotonic and for all chains $d_0 \sqsubseteq d_1 \sqsubseteq \cdots \sqsubseteq d_n \sqsubseteq \ldots$ in D we have

$$\bigsqcup_{n \in \omega} f(d_n) = f\left(\bigsqcup_{n \in \omega} d_n\right)$$

□

Notice that a continuous function is monotonic and that a monotonic function on a complete partial order which only contains finite chains is always continuous. Furthermore, the sequential composition $f \circ g$ of two continuous functions f and g is again continuous.

Definition 5.4.7 ((Pre-)Fixed Point)
Let $f : D \to D$ be a continuous function on a CPO D. A *fixed point* of f is an element $d \in D$ with $f(d) = d$. A *prefixed point* of f is an element $d \in D$ such that $f(d) \sqsubseteq d$. A prefixed point d for which we have $d \sqsubseteq d'$ for all prefixed points d' is called a *least prefixed point.* □

Kleene's theorem provides a possibility to compute the least fixed point of a continuous function on a CPO with bottom iteratively.

Theorem 5.4.1 (Kleene Iteration)
Let $f : D \to D$ be a continuous function on a CPO D with bottom $\perp$. Then

$$\mathrm{lfp}(f) =_{df} \bigsqcup_{n \in \omega} f^n(\perp)$$

exists and is both the unique least fixed point and the unique least prefixed point of the function f. □

In practice, we do not deal with arbitrary complete partial orders with bottom, but restrict ourselves to CPOs with finite heights. This further facilitates the usage of Kleene's theorem.

Definition 5.4.8 (Height)
The *height* $h(D)$ of a finite CPO D with bottom is defined to be the length of its longest chain minus one. The *height* $h(f)$ of a function $f : D \to D$ is $h(D)$. □

Note that the height of the power domain $\wp(X)$ for finite, non-empty sets X is $|X| - 1$. Moreover, finite chains have finite heights. The following theorem provides a general technique for computing a fixed point by iterating.

Theorem 5.4.2 (Iterative Fixed Point Computation)
Let $f : D \to D$ be a continuous function on a CPO with bottom $\perp$ and finite height. Then $\mathrm{lfp}(f) = f^{h(f)}(\perp)$. □

Let $S =_{df} S_1 \lhd L \rhd S_2$. As the power domain is a CPO with bottom $\emptyset$ and f_x with

$$f_x : Out(S_1) \cup Out(S_2) \to Out(S_1) \cup Out(S_2)$$

is a monotonic function over a finite power domain $\wp(Out(S_1) \cup Out(S_2))$ — recall that we restricted ourselves to charts with monotonic semantics — it is in addition continuous and we can compute the least fixed point of f_x iteratively:

$$\mathrm{lfp}(f_x) = f^{|Out(S_1) \cup Out(S_2)|-1}(\emptyset)$$

Though the above formula provides a basis to compute the fixed point of f_x if it is implemented on a single processor, it does not provide a solution how to compute this fixed point in the case that f_x is implemented on a distributed architecture, where each control unit does not compute f_x entirely, but computes only a part of f_x instead. One severe problem now arises in the case of physically distributed implementation of these parts: since we cannot make any assumptions concerning the relative speeds of the processors that realize the parts, all possible sequential and parallel execution orders can occur. This procedure results in a chaotic instead of an iterative Kleene computation of the least fixed point. In the following section, we will show that this chaotic iteration stabilizes and yields the least fixed point.

5.4.2 Distributed Fixed Point Computation

In the sequel, we will show what we mean by "a part of f_x" and will furthermore show that the distributed execution of these parts is possible. Therefore, we first need some definitions.

Let X be a finite set and $f : \wp(X) \to \wp(X)$ be a function on the CPO $(\wp(X), \subseteq)$ with bottom $\emptyset$. Then the function $f^{[Q]}$ for $Q \subseteq X$ is defined as follows:

$$f^{[Q]}(x) =_{df} f|_Q(x) \cup x$$

where $f|_Q(x)$ is defined by $f(x) \cap Q$. If f is monotonic, also $f^{[Q]}$ is monotonic since for all $x \in \wp(X)$ we have the following chain of implications:

$$x \subseteq y \Rightarrow f(x) \subseteq f(y) \Rightarrow f(x) \cap Q \subseteq f(y) \cap Q \Rightarrow (f(x) \cap Q) \cup x \subseteq (f(y) \cap Q) \cup y$$

Furthermore, if f is continuous, also $f^{[Q]}$ is. The proof is straightforward if we recall that monotonic functions on finite CPOs are always continuous. Moreover, the following is true:

$$f^{[Q]}(\mathrm{lfp}(f)) = \mathrm{lfp}(f) \tag{5.6}$$

as we easily show by: $f^{[Q]}(\mathrm{lfp}(f)) = (f(\mathrm{lfp}(f) \cap Q) \cup \mathrm{lfp}(f) = \mathrm{lfp}(f)$. Theorems 5.4.1 and 5.4.2 provide a possibility to compute the least fixed point of a recursive equation by an iterative procedure. However, in the case of a distributed implementation we cannot guarantee that the fixed point generation is computed incrementally. Rather, it can be computed chaotically, that is, not incrementally. The following theorem, which is in its original version due to [Rob86] and was inspired by [Edw97], provides the mathematical basis for a chaotic fixed point construction. Since the semantic model used in [Rob86] and [Edw97] differs from the one for μ-Charts, it was necessary to adapt and re-formulate the original theorem. In particular, also the definition of a "part" of a function $f^{[Q]}$ is different. As a consequence, though we could use the proof sketch of the original literature it was also necessary to prove the modified theorem.

Theorem 5.4.3 (Chaotic Fixed Point Computation)
Let X be a finite set and $f : \wp(X) \to \wp(X)$ be a continuous function on the power domain $\wp(X)$. Furthermore, let the function $g : \wp(X) \to \wp(X)$ be defined as:

$$g =_{df} f^{[Q_k]} \circ \cdots \circ f^{[Q_1]}$$

where $Q_i \subseteq X$ for all $1 \leq i \leq k$ and $\bigcup_{i=1}^{k} Q_i = X$. Then f and g have the same unique least fixed point. $\square$

Proof 17 As f is a continuous function on a CPO with bottom $\emptyset$, f has a unique least fixed point $\mathrm{lfp}(f)$. Furthermore, all functions $f^{[Q_i]}$ are continuous, too. Therefore, g is, defined as the composition $f^{[Q_k]} \circ \cdots \circ f^{[Q_1]}$, also continuous. As a consequence, g has a unique least fixed point $\mathrm{lfp}(g)$, which can be computed by Kleene iteration. It remains to be shown that $\mathrm{lfp}(g) = \mathrm{lfp}(f)$. To this end, we first prove that $\mathrm{lfp}(g) \subseteq \mathrm{lfp}(f)$.

From Proposition 5.6 we know that $f^{[Q_i]}(\mathrm{lfp}(f)) = \mathrm{lfp}(f)$ for all $1 \leq i \leq k$. Due to the monotonicity of the $f^{[Q_i]}$ and this proposition:

$$\begin{array}{rcl}
\emptyset & \subseteq & \mathrm{lfp}(f) \Rightarrow \\
f^{[Q_1]}(\emptyset) & \subseteq & f^{[Q_1]}(\mathrm{lfp}(f)) = \mathrm{lfp}(f) \Rightarrow \\
(f^{[Q_2]} \circ f^{[Q_1]})(\emptyset) & \subseteq & f^{[Q_2]}(\mathrm{lfp}(f)) = \mathrm{lfp}(f) \Rightarrow \\
 & \vdots & \\
g(\emptyset) & \subseteq & \mathrm{lfp}(f)
\end{array}$$

As $f^{[Q_i]}(\mathrm{lfp}(f)) = \mathrm{lfp}(f)$ also $g(\mathrm{lfp}(f)) = \mathrm{lfp}(f)$ for all $1 \leq i \leq k$ and therefore

$$\begin{array}{rcl}
(g \circ g)(\emptyset) & \subseteq & g(\mathrm{lfp}(f)) = \mathrm{lfp}(f) \Rightarrow \\
 & \vdots & \\
g^{h(g)}(\emptyset) & \subseteq & \mathrm{lfp}(f)
\end{array}$$

As the height $h(g)$ equals $|X| - 1$ and therefore is finite, we have $\mathrm{lfp}(g) = g^{h(g)}(\emptyset) \subseteq \mathrm{lfp}(f)$.

In the sequel, we show $\mathrm{lfp}(f) \subseteq \mathrm{lfp}(g)$. First, we have:

$$\mathrm{lfp}(g) \subseteq f^{[Q_1]}(\mathrm{lfp}(g)) \subseteq (f^{[Q_2]} \circ f^{[Q_1]})(\mathrm{lfp}(g)) \subseteq \cdots \subseteq g(\mathrm{lfp}(g)) = \mathrm{lfp}(g)$$

As a consequence, $f^{[Q_i]}(\text{lfp}(g)) = \text{lfp}(g)$ for all $1 \leq i \leq k$. Moreover, since $\bigcup_{i=1}^{k} Q_i = X$ we have

$$\begin{aligned} f(\text{lfp}(g)) &= \bigcup_{i=1}^{k} f|_{Q_i}(\text{lfp}(g)) \\ &= \bigcup_{i=1}^{k} (f(\text{lfp}(g)) \cap Q_i) \\ &\subseteq \left(\bigcup_{i=1}^{k} (f(\text{lfp}(g)) \cap Q_i)\right) \cup \text{lfp}(g) \\ &= \bigcup_{i=1}^{k} ((f(\text{lfp}(g)) \cap Q_i) \cup \text{lfp}(g)) \\ &= \bigcup_{i=1}^{k} f^{[Q_i]}(\text{lfp}(g)) = \bigcup_{i=1}^{k} \text{lfp}(g) = \text{lfp}(g) \end{aligned}$$

Thus, $\text{lfp}(g)$ is a prefixed point of f. As $\text{lfp}(f)$ is the least prefixed point of f, also $\text{lfp}(f) \subseteq \text{lfp}(g)$ holds. Altogether, we have $\text{lfp}(g) = \text{lfp}(f)$ which finally concludes the proof. $\square$

This theorem bridges the gap between centralized and distributed execution of a μ-Chart program. Let us assume that $\triangle_L(S_1 \| \cdots \| S_J)$ is the normal form of the composition of J sequential automata and that its current configuration is $c =_{df} (c_1, \ldots, c_J)$. The step semantics of $\triangle_L(S_1 \| \cdots \| S_J)$ in configuration c with stimulus x is computed by computing $st[\![S_1 \| \ldots \| S_J]\!](c, x)$, which again is computed compositionally by independently computing the semantics $st[\![S_j]\!](c_j, x)$ of all automata S_j. If we abbreviate the semantics $st[\![S_1 \| \cdots \| S_J]\!](c, x)$ to f_x then each of the "restrictions" $f_x^{[Q_j]}$ of f_x, where $Q_j =_{df} Out(S_j)$, is exactly represented by $st[\![S_j]\!](c_j, x)$ if the output interfaces of $S_1, \ldots, S_J$ are disjoint. Now, it becomes clear what we meant with a "part" of the behavior. Computing the least fixed point of f_x means to compute $\text{lfp}(f_x)$ iteratively by:

$$\text{lfp}(f_x) = f_x^{|Out(S_1) \cup \cdots \cup Out(S_J)|-1}(\emptyset)$$

However, this way of computing $\text{lfp}(f_x)$ is very restrictive and not applicable for distributed implementations of f_x because all $f_x^{[Q_j]}$ have to be executed in parallel. Though this is a realistic assumption for centralized, or distributed, but tightly coupled implementations, it is not for distributed executions, where we cannot make any assumptions on the relative execution speeds of distributed control units. What we rather need for a loosely coupled distributed execution is that the $f_x^{[Q_j]}$ can be executed in arbitrary order. Exactly this is guaranteed by Theorem 5.4.3. Here, the central property f_x has to fulfill is continuity. Notice, however, that a continuous f_x does not have to have necessarily a causality error free signal graph, too. Thus, for specifications that shall be implemented on a distributed target architecture we require both, their continuity and the absence of causality errors. Further note that the normal form is a prerequisite for the applicability of this theorem to μ-Charts. If a specification was not in normal form, according to Formula 5.5 a nested fixed point computation would be necessary to compute the overall fixed point of f_x.

Only one draw-back remains so far: this theorem only tells us that distributed, chaotic execution of composed automata is possible, but does not give any hints how to compute the semantic fixed point efficiently. The main disadvantage is that in any case $|Out(S_1) \cup \cdots \cup Out(S_J)| - 1$ iterations are needed. The next section will concentrate on this question and will show how to optimize the distributed fixed point computation.

5.4.3 Efficient Communication

This section addresses the problem of computing a distributed fixed point efficiently. In particular, we aim at decreasing the number of iterations that are necessary to generate the fixed point. For these purposes, we need an efficient way to schedule the messages that occur in an instant. While the central theorem of the last section has shown that in principle each possible communication schedule is feasible, the goal now is to find a schedule that is as efficient as possible.

This schedule is formally derived from the so-called *dependence graph*, which reflects the causal relationships of the multicasting signals of one instant. Dependence graphs are directed graphs that are, in contrast to those in [Edw97], acyclic. A dependence graph is directly computed from the signal graph: two signals s and s' are connected by an arc in the dependence graph, whenever $s < s'$ holds, which is defined by:

$$s < s' \Longleftrightarrow_{df} \exists x^+, x^-.(s \rightarrow^{x^+}_{x^-} s' \wedge x^+ \subseteq \psi(s) \wedge x^- \cap \psi(s) = \emptyset \wedge s' \notin \psi(s))$$

where the set $\psi(s)$ denotes those signals that are, apart from s itself, needed to guarantee the presence of s in the current instant. Hence, whenever $s \in \mathrm{lfp}(f_x)$ then also $\psi(s) \cap \mathrm{lfp}(f_x) = \psi(s)$. More formally, $\psi(s)$ is the smallest set $\psi'(s)$ with respect to the subset ordering that fulfills the following predicate:

$$(s \rightarrow^{x^+}_{x^-} s' \wedge x^+ \subseteq \psi'(s) \wedge x^- \cap \psi'(s) = \emptyset \wedge s' \notin \psi'(s)) \Rightarrow \{s'\} \cup \psi'(s) \subseteq \psi'(s')$$

We have to compute the set $\psi(s)$ for each node s in the signal graph. For these nodes $\{s_1, \ldots, s_N\}$ in the signal graph that are not successor nodes of any other node (i.e. the external signals) we initialize $\psi(s_n)$ for all $1 \leq n \leq N$ to the set $\{s_1, \ldots, s_N\}$.

The set $\psi(s) \backslash s$ denotes the set of signals which must be present in the current instant in order to guarantee that also s is present in the current instant. Figure 5.18 shows the signal graph G for the chart pictured in Figure 5.17. The sets $\psi(s)$ for all signals s that occur as nodes in G are defined as follows: first of all, we initialize $\psi(a) = \{a\}$. The remaining presence statuses are computed as follows:

$$\psi(b) = \{a, b\} \quad \psi(c) = \{a, b, c\} \quad \psi(d) = \{a, b, c, d\} \quad \psi(e) = \{a, b, c, d, e\}$$

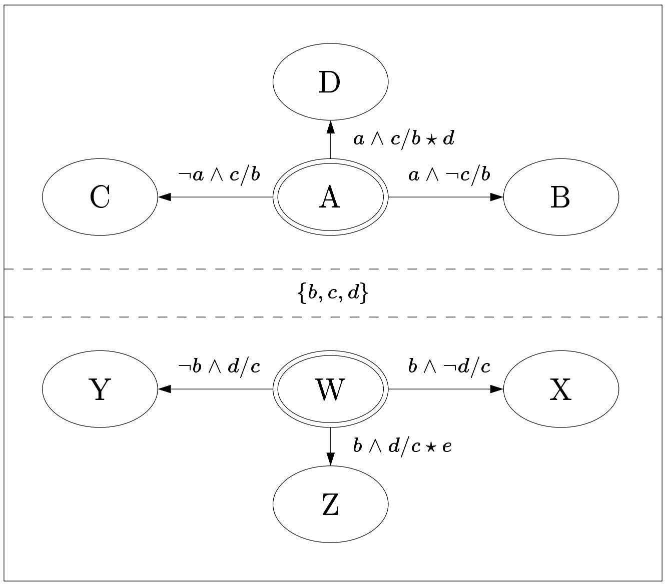


Figure 5.17: Instantaneous dialog (2)

Recall that we exclude signal graphs that contain cycles with causality conflicts. For these signal graphs acyclic dependence graphs cannot be computed. Though it is in principle possible to compute communication schedules for cyclic dependence graphs, this is a computational complex operation as demonstrated in [Edw97], where a polynomial divide-and-conquer algorithm has been presented. Unfortunately, this algorithm has been developed for static configurations only. However, in the context of μ-Charts we mostly have to deal with continuously changing configurations. In each configuration, there may be a different causality relation between the internal signals. As a consequence, the optimal schedule has to be computed for every single configuration anew. Private discussions with the author of [Edw97] have shown that in all probability there is no polynomial time algorithm to calculate the schedule of two successor configurations from a cyclic dependence graph incrementally, that is, by using the schedule of the predecessor. We guess that this is true, even if both consecutive dependence graphs are "very similar", i.e. merely differ by more or less a single node.

An even more important argument is that the optimal schedules for cyclic dependence graphs require more iterations than those for dependence graphs without cycles. Since we have restricted ourselves to causality error free signal graphs, we automatically get acyclic dependence graphs and therefore can give an easier and more efficient computation scheme for distributed fixed points than the one in [Edw97].

The dependence graph for the signal graph pictured in Figure 5.18 is pictured in Figure 5.19. Though we here have a linear causality relation, in general, the dependence graph has a tree-like structure, possibly with more than one root.

As we want to find optimal schedules for functions $f^{[Q_k]}, \ldots, f^{[Q_1]}$ we have to extend the

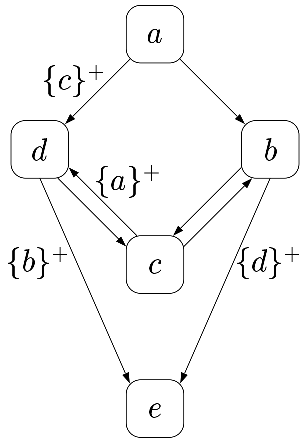


Figure 5.18: Signal graph

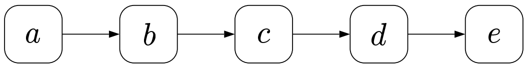


Figure 5.19: Dependence graph for Figure 5.18

relation $<$ to signal sets as follows:

$$\forall i \neq j : Q_i < Q_j \Longleftrightarrow_{df} \exists s_i \in Q_i \backslash Q_j, s_j \in Q_j \backslash Q_i : s_i < s_j$$

The dependence graphs pictured in Figures 5.20, 5.21, and 5.22 show examples for possible selections such that $Q_1 < Q_2$, $Q'_1 < Q'_2$, and $Q''_1 < Q''_2 < Q''_3 < Q''_4$ hold, respectively. Note that in general $Q_i < Q_j \Rightarrow \neg(Q_j < Q_i)$ is not true. This is the case whenever Q_i or Q_j are incoherent sets. As an example, see Figure 5.23, where $Q'''_1 < Q'''_2 \wedge Q'''_2 < Q'''_1$.

These signal sets also can be interpreted as hyper nodes. The resulting graph then is a *hyper graph*, constructed from the dependence graph. This interpretation will play an important role in Section 6.2. For instance, we obtain a hyper graph with two nodes (Q'''_1 and Q'''_2) and two edges (one from Q'''_1 to Q'''_2 and one from Q'''_2 to Q'''_1). We realize that, since we have limited ourselves to acyclic dependence graphs, incoherent signal sets correspond with cyclic hyper graphs and coherent sets with acyclic hyper graphs. Therefore, requiring coherent signal sets is equivalent to requiring acyclic hyper graphs.

However, for efficient schedules this condition is a prerequisite as we will see in the following theorem. This theorem gives a sufficient condition for the choice of “parts” of a continuous function f in order to compute its least fixed point as efficient, that is, with as few iterated function applications as possible. The theorem even guarantees that with a “good” choice for the Q_i each part $f^{[Q_k]}$ of f just has to be applied only once.

Theorem 5.4.4 (Efficient Schedules)
Let X be a finite set and $f : \wp(X) \to \wp(X)$ be a continuous function on the power domain $\wp(X)$. Furthermore, let $Q_1, \ldots, Q_k \subseteq \text{lfp}(f)$ such that:

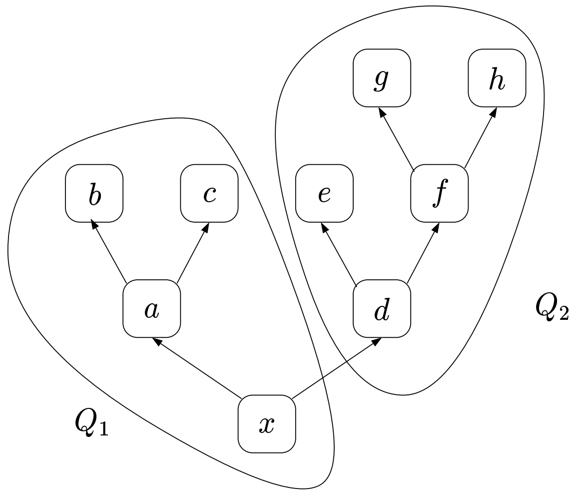


Figure 5.20: Dependence graph with optimal schedule (1)

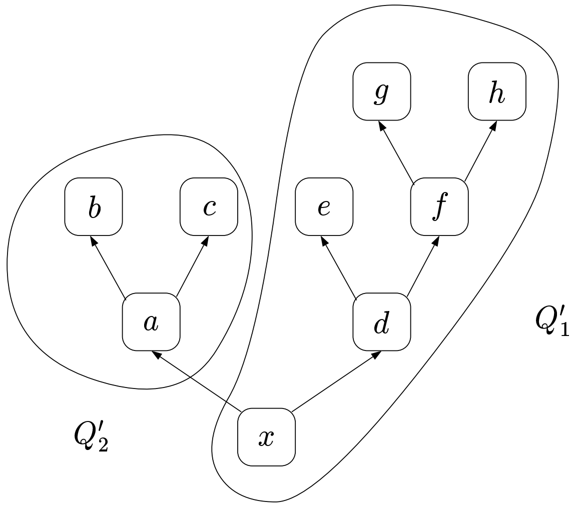


Figure 5.21: Dependence graph with optimal schedule (2)

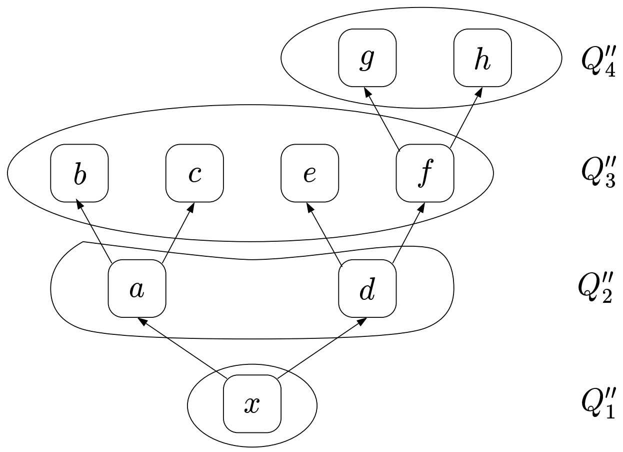


Figure 5.22: Dependence graph with optimal schedule (3)

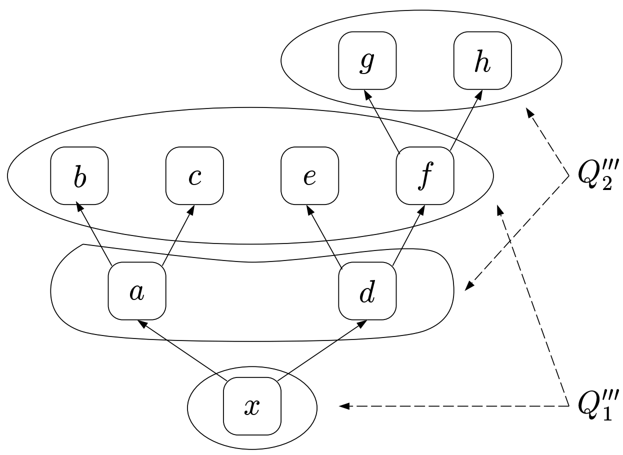


Figure 5.23: Dependence graph with sub-optimal schedule

1. $\bigcup_{i=1}^{k} Q_i = \text{lfp}(f)$,
2. $Q_1 < \cdots < Q_k$, and
3. $\forall i \neq j : \neg(Q_i < Q_j \wedge Q_j < Q_i)$.

Then $\text{lfp}(f) = f^{[Q_k]} \circ \cdots \circ f^{[Q_1]}(\emptyset)$. □

Proof 18 First, we show that $f|_{Q_i}(Q_{i-1}) = Q_i$ for all $1 \leq i \leq k$. We prove this intermediate proposition by showing that $f|_{Q_i}(Q_{i-1}) \subseteq Q_i$ and $Q_i \subseteq f|_{Q_i}(Q_{i-1})$. The subset relation $f|_{Q_i}(Q_{i-1}) \subseteq Q_i$ trivially holds by definition of $f|_{Q_i}$. More subtle is the proof of $Q_i \subseteq f|_{Q_i}(Q_{i-1})$ for all $1 \leq i \leq k$, which is carried out by contradiction. Hence, we assume that there exists i with $1 \leq i \leq k$ such that $f|_{Q_i}(Q_{i-1}) \subset Q_i$. As a consequence, there must be a $y \in Q_i$ that is not contained in $f|_{Q_i}(Q_{i-1})$. Since $Q_1 < \cdots < Q_k$ there must be an $x \in Q_n$ for $n \geq i+1$ such that $x < y$. Thus, also $Q_n < Q_i$ which is a contradiction to $\neg(Q_i < Q_j \wedge Q_j < Q_i)$ because already $Q_i < Q_n$ holds, and the assumption $f|_{Q_i}(Q_{i-1}) \subset Q_i$ must have been false.

We now prove the actual proposition of the theorem. By definition we have $f^{[Q]}(x) = f|_Q(x) \cup x$. Therefore

$$f^{[Q_k]} \circ \cdots \circ f^{[Q_1]}(\emptyset) = f|_{Q_1}(\emptyset) \cup f|_{Q_2}(f|_{Q_1}(\emptyset)) \cup \cdots \cup f|_{Q_k}(\cdots(f|_{Q_1}(\emptyset))\cdots)$$

Defining Q_0 to be the empty signal set, this is due to the above intermediate proposition equivalent to $Q_1 \cup Q_2 \cup \cdots \cup Q_k$, which equals by Assumption (1) of the theorem the fixed point $\text{lfp}(f)$. □

Notice that conditions (2) and (3) in the above theorem require that the dependence graph is partitioned into coherent signal sets. Recall that coherent signal sets are equivalent to acyclic hyper graphs.

To get an example, we take a closer look at the dependence graphs with their schedules as pictured in Figures 5.20, 5.21, 5.22, and 5.23. We realize that the first three are optimal schedules whereas the last one is merely sub-optimal because we have both $Q_1''' < Q_2'''$ and $Q_2''' < Q_1'''$ and therefore neither $f_x^{[Q_1''']} \circ f_x^{[Q_2''']}(\emptyset)$ nor $f_x^{[Q_2''']} \circ f_x^{[Q_1''']}(\emptyset)$ equals to $\text{lfp}(f_x)$.

With Theorem 5.4.4 we now have a method at hand to partition a μ-Chart specification in an efficient way. We have seen that efficient schedules require a partitioning in coherent subsets of the overall set of signals that is multicast in one instant. Recall that this is equivalent to requiring that the hyper graph that results from this partitioning is acyclic. In addition, as sequential automata are considered as atomic units for partitioning, we require that the output interface of one automaton never can be separated into different signal sets. Hence, each of the signal sets can only consist of one or more complete output interfaces of one or more sequential automata. In Section 6.2, we will see that this property is not only a prerequisite for efficient schedules, but in addition is generally necessary to get partitions that can be implemented on a distributed architecture, where each distributed part is based on finite state machines.

5.5 Conclusion

In this chapter, we have presented an approach for partitioning a specification in μ-Charts to a set of individual parts such that each part potentially can be implemented on a different processor. The chapter has dealt with several topics that though all being related to partitioning are relatively standalone. First, a implementation-oriented step semantics for deterministic μ-Charts has been presented. In contrast to the stream semantics in Chapter 2, the step semantics of this chapter defines instantaneous feedback obtained by multicasting as explicit fixed point construction and is therefore more closely related to implementation and is better suitable to reason about distributed implementation. We have proven that both semantics coincide for implementable, that is, (semantically) deterministic specifications. In addition, we have shown that distributed implementation and centralized program have the same behavior with respect to signal propagation between sequential automata.

If a specification shall be partitioned and later on implemented on a loosely coupled distributed processor network, where processors only communicate via one or more busses, but do not necessarily interact in a synchronous fashion, a number of restrictions has to be made in order to guarantee that the distributed implementation features the behavior described by the original synchronous specification. We have discussed these restrictions and clustered them into classes of problems that are caused by absences of input signals and so-called internal and external input signals. For internal input signal an approach for causality analysis has been proposed.

In order to describe the distributed implementation of a μ-Chart specification mathematically, we have discussed an approach to model the allocation problem, that is, to determine the mapping of sequential automata to processors and communications between them to busses. Since our approach is based on integer linear programming, it can be directly used to compute optimal allocations with respect to an objective function. However, communication between distributed components cannot be optimized while solving the allocation problem because it is not possible to treat both problems with the same mathematical concepts. Therefore, it has been demonstrated separately how to obtain efficient communications by referring to the causality analysis again.

6 Code Generation

In the previous chapter, we have discussed how to partition a μ-Chart specification and how to achieve efficient allocations and schedules. We have seen that a number of conditions must be fulfilled in order to guarantee a partitioning strategy that preserves the original semantics of the specification. In this section, we will recognize that further problems can occur, and restrictions are necessary when finally implementing the specification on a net of electronic control units. In the code generation scheme we aim at, control units are implemented as finite state machines. This approach guarantees maximal flexibility for both hardware and software generation. In particular, mixed hardware/software designs are possible. Therefore, our implementation scheme is applicable for Hardware/Software-Codesign. The problems we will discuss in this section are due to our specific, distributed implementation scheme that is based on finite state machines. Using other implementation techniques, the problems discussed in the following do not necessarily occur. In particular, they can be avoided if one does not target at distributed implementation, but rather at distributed simulation. The latter is interesting if large specifications are to be tested in real-time. However, before we discuss distributed implementation we first outline a scheme for centralized code generation.

6.1 Centralized Code

In this section, we present a hardware implementation scheme for μ-Charts. In contrast to the tools Statemate [i-L90] and Statemate Magnum [i-L97], we aim at a direct implementation, and not at compilation to VHDL code. This direct implementation has one major advantage: whenever a visual formalism is translated to a high-level programming language like VHDL or C, the compiler is mostly written ad-hoc, that is, without formally verifying that translation semantics and original semantics coincide. This proof is not necessary in our case because we immediately use the semantics of Chapter 4, which we also took for model checking.

We are aware of two previous approaches for direct hardware implementations of a Statecharts-like language, presented by Drusinsky in [DY89, DY91]. The first one [DY89] described the (centralized) implementation of a Statechart as a tightly coupled network of communicating finite state machines, even if the implementation is on a single chip. Especially for small and medium-sized systems this scheme pretendedly introduces considerable communication overhead; in addition, Drusinsky himself admits that it is difficult to implement this scheme correctly [DY91].

In the second approach [DY91], he therefore suggested to realize a Statechart as a single logic block. However, it is not clear how this approach scales to larger specifications. Neither of these two approaches is based on a formal semantics. Thus, it is not possible to formally verify designs based on these approaches. Moreover, neither approach allows to specify systems with data states.

Our implementation scheme is based on the formal semantics introduced in Chapter 4; in particular, implementations are generated from the same transition relations that are used for formal verification of μ-Charts with symbolic model checkers. While our centralized scheme also avoids the communication overhead of communicating components, the implementations can naturally be divided into smaller, independent logic blocks, each of them responsible for the value of a single output signal bit.

Re-Use of Symbolic Encoding

In Chapter 4, we presented the formal step semantics of μ-Charts. It has been shown that the step semantics of a given μ-Chart S can be described as a finite transition relation

$$Trans_S(x, c, y, c')$$

where x and y are finite encodings of the input and output signals, respectively. The current and next system configurations are encoded in c and c', respectively. A similar predicate $Init_S(c)$ for initialization has been formulated.

These predicates are used for the hardware implementation according to Figure 6.1. The input to the logic block consists of the external inputs x, the finite encoding of the current system configuration c, and a reset wire. The logic block has the system output y and the new system state encoding c' as output. In addition to this combinational logic block, we need a state register (edge triggered) for storing the control and data states and an external clock that triggers the μ-Chart steps.

The logic block is derived in the following way. The transition relation $Trans_S$ is converted to a family of Boolean functions, one for each output signal y_i, and one for each bit in the encodings of the control and data states. The Boolean functions for a bit are derived from the transition relation $Trans_S$ by existentially quantifying the other output signals. This operation can, for instance, be efficiently carried out with BDD techniques. The conversion to Boolean functions is possible, since for hardware generation we restrict ourselves to deterministic μ-Charts. The check whether a μ-Chart is deterministic can also be performed on the transition relation $Trans_S$ (see also Section 4.2.1). As shown in Figure 6.2, the individual Boolean functions can be implemented as separate logic blocks. Note that we have removed the connections for the *reset* wire to keep the figure easy to survey. Indeed, reset wires are needed to all logic blocks $Logic_1, \ldots, Logic_n$ and $SLogic_1, \ldots, SLogic_n$.

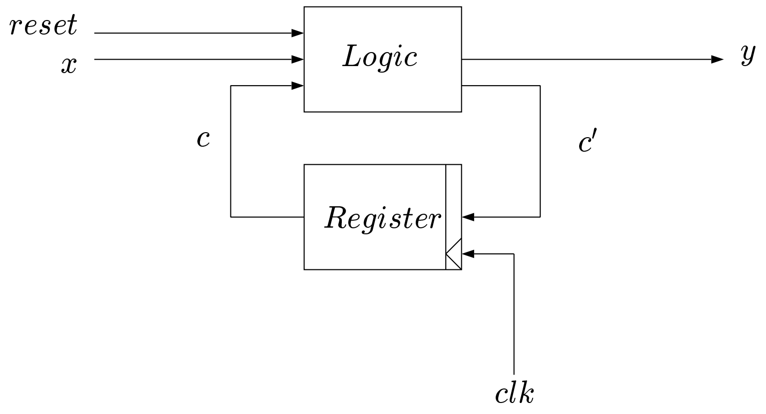


Figure 6.1: Hardware implementation scheme (1)

Each Boolean function for an output signal contains an abstraction of the complete specification. This is the reason that our centralized implementation scheme does not require explicit communication between the logic blocks. The abstraction contains exactly those aspects of the complete system specification that are needed to calculate the signal's value — neither more nor less. As a consequence, the individual Boolean functions can be represented with comparatively small logic blocks, much smaller than the complete transition relation $Trans_S$.

Stopwatch Example

In this section, we apply our implementation technique to the stopwatch example of Section 2.3.4. Since only a finite subset of integers is used for the data states in the example, we can encode them, for instance, with BDDs. For the clock counter in the sub-chart TIMER, we employ the standard binary encoding for integers between 0 and 100,000; this requires 17 bits. We could represent the ten values for the digit displays using 4-bit integers, but instead we have chosen a direct encoding of the seven segments for each digit. This allows us to use a slight, but efficient variation of the implementation scheme: instead of dedicated output signals for the segments, we use the state encoding itself to drive the display.

Since the watch consists of five sequential automata, two parallel compositions, and two feedback constructions, there are nine initialization and nine transition relations. The top level relations, which are those of $S_{Stopwatch} \rhd (\{\mathrm{ON}\}, \varrho_{Stopwatch})$, are the ones used for hardware generation. The transition relation is then converted to a family of Boolean functions. It is not possible, in general, to predict the logic block sizes for a given μ-Chart. We can, however, calculate the number of bits needed to encode the data and

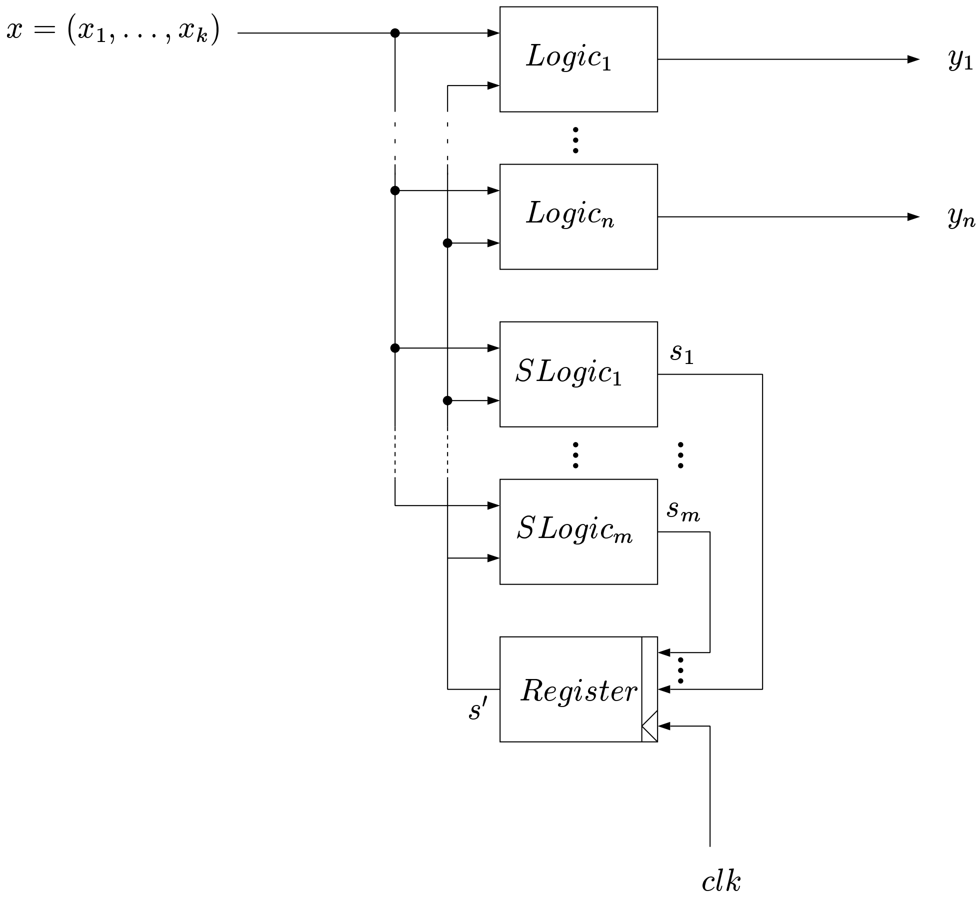


Figure 6.2: Hardware implementation scheme (2)

control states, and thus the width of the state register:

$$\begin{aligned}
bits((I, O, \Sigma, \Sigma_0, V_l, \varphi_0, \delta)) &= \lceil ld(|\Sigma|) \rceil + \lceil ld(\sum_{X \in V_l} range(X)) \rceil \\
bits(S_1 \lhd L \rhd S_2) &= bits(S_1) + bits(S_2) \\
bits([S]_K) &= bits(S)
\end{aligned}$$

where $range(X)$ denotes the range of the variable X. Recall that for model checking and implementation X has to have a finite type. Using the above formula for calculating the width of the state register that is necessary to store the configuration of hierarchically decomposed charts would yield needlessly large register width. Since at each instant at most one controllee can be active, we derive to the following formula:

$$bits(A \rhd (\Sigma_d, \varrho)) = bits(A) + \max\{bits(\varrho(\sigma)) \mid \sigma \in \Sigma_d\}$$

For STATEMATE Statecharts, efficient state assignments are given in [DY91]. In the stopwatch example, we get $ld(2) = 1$ flipflop for the automaton $S_{Stopwatch}$, $ld(4) + \lceil ld(10^5) \rceil = 2 + 17 = 19$ flipflops for S_{Timer}, and $3 \cdot (7 + 1) = 24$ flipflops for $S_{Display}$.

As aforementioned, we directly use the encoding of the local states to control the seven segment displays and so avoid additional logic to decode the logic for each segment out of a four bit state. Altogether we need a register 44 bit wide.

Referring to Figure 6.2 again, the register width is the constant m, in our example 44. The constant k is the number of external input signals. In the stopwatch example, we get $k = 2$ since the only inputs are for the two buttons on the watch. Finally, n is the number of output signals. In the example, n should be $3 \cdot 7 = 21$ for the three seven segment displays. However, since we directly use the state encoding as outputs for the display, our example gives $n = 0$, since there are no other outputs of the system. Thus, for the example 44 logic blocks have to be implemented. Note that *low*, *med*, and *time* are internal signals only. Their use is implicitly modeled in the Boolean functions.

6.2 Distributed Code

In the preceding section, an implementation scheme for μ-Charts specifications on a single finite state machine has been presented. It is also a challenging question to implement the same model on a distributed architecture. As target architecture we take the one described in Section 5.3. The distributed implementation of a perfect synchronous specification based on finite state machines is difficult and possibly leads to several problems, which will be discussed in the sequel. The approaches that are followed by Lustre and Signal have already been outlined in Section 1.2.

Though clock asynchronous languages in contrast permit a much easier programming of distributed systems than synchronous ones, they are — due to the draw-backs outlined in Section 1.2.2 — inappropriate description techniques for reactive systems. The code distribution we propose for μ-Charts is based on the syntactic structure of the source specification. Ideally, to get a distributed implementation of a model, say the central locking system, we follow the strategy given below:

1. The original specification is transformed step-by-step by applying the refinement rules of Section 3 to a deterministic specification. Specifications that lead to causality conflicts (see Section 5.1) have to be rejected, since these conflicts correspond to deadlocks in networks.
2. When the specification is stable and deterministic and all safety critical properties have been verified through model checking (Section 4), the allocation for all sequential automata and signal transfers has to be found. This procedure is based on the μ-Charts normal form and can either be done by hand or by using the optimization techniques presented in Section 5.3. As next step towards a distributed implementation, this allocation now has to be put into practice.

Hence, as next and last step one might think that it should be sufficient to simply map the transformed specification on the target architecture. After all, we have shown in

Section 5.2.3 that a specification without harmful cycles in any signal graphs of any reachable configuration can be implemented deadlock-free on a distributed architecture. Furthermore, we have formally proven in Section 5.4 that a distributed fixed point computation is possible if the specification has a continuous semantics. However, as we will discuss in the remainder of this section, some further difficulties have to be kept in mind. They occur because our method for code generation is based on finite state machines. Thus, in addition to the restrictions made in the previous chapter and which have been necessary to obtain deadlock-free communications, yet other requirements have to be considered. While the problems discussed in Chapter 5 have been related to the communication between different sequential automata, in the following, we will concentrate on problems caused by single automata.

Problems caused by Product Automata

Each step consists of a number of subsequent micro-steps, in which one or more parallel transitions are fired. All single micro-steps considered together build a chain reaction of transitions. Here, the first transition is triggered by an external event, and all subsequent ones by the same external event or internal events. One of the basic assumptions of μ-Charts is that in every instant each sequential automaton can fire only one single transition. Apart from the reason to avoid infinite transition chains, this restriction has been made to get implementable specifications since in each clock cycle a finite state machine can execute its combinational logics only once. In the following, we will see that though we have restricted chain reactions within one instant requiring that each sequential automaton can fire on single transition only, further implementation problems can occur.

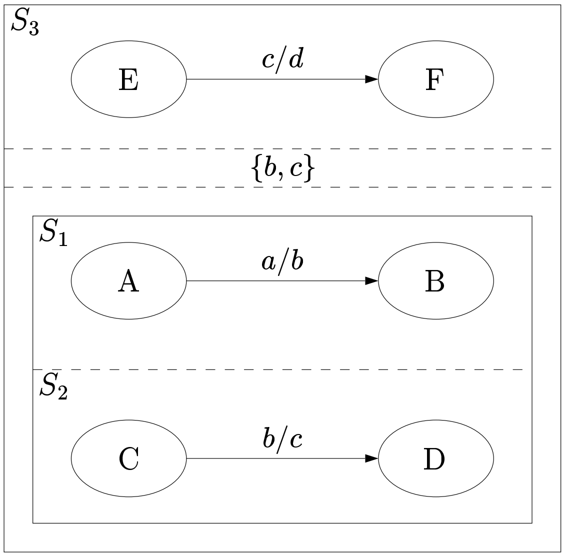


Figure 6.3: Problem with distributed implementation

We explain the problems that can arise when compiling a μ-Chart specification on a processor network by the aid of the example pictured in Figure 6.3. We assume that two processors, say P_1 and P_2, are available to implement this specification. Therefore, two possible allocation are to (1) either run $S_1 \| S_2$ on P_1 and S_3 on P_2 or (2) to run $S_1 \| S_3$ on P_1 and S_2 on P_2. Of course, the parallel compositions $S_1 \| S_2$ and $S_1 \| S_3$, respectively, are implemented on P_1 by realizing the corresponding transition relation. This is equivalent to implementing the corresponding product automata.

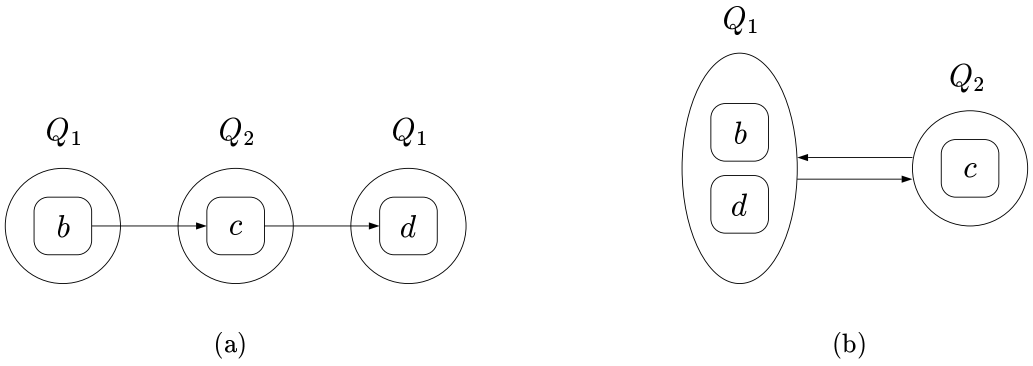


Figure 6.4: (a) Incoherent partitioning and (b) corresponding cyclic hyper graph

Obviously, the first alternative only requires the transmission of the signal c over the net whereas for the latter two transmissions, namely the ones of b and c, are necessary (see Figure 6.4). However, this is not the unique and, above all, not the worst problem. Figure 6.5 shows the product automaton for $S_1 \| S_3$ in Figure 6.3. For the input event $\{a\}$ one would expect, according to the formal semantics, the output event $\{b, c, d\}$. However, since each finite state machine can change its configuration only once (!) between two subsequent clocks, the de facto output is $\{b, c\}$ and the successor configuration (BE,D) and not, as specified, (BF,D). This simple example illustrates the problem we are facing. In LUSTRE, this problem is solved by so-called "restructuring" of the affected program [Hal93b]. In this example, this would mean to forbid the allocation alternative number (2). Through analysis of the corresponding dependence graph, we also are able to restrict ourselves to feasible allocations.

This can be done by analysis of the dependence graph. Recall from Section 5.4.3 that we always obtain acyclic dependence graphs from causality error free signal graphs. As we have noticed, this is a prerequisite for distributed implementation. Furthermore, remember that nodes of dependence graphs are represented by signal names. We can now build hyper graphs from dependence graphs by collecting nodes (= signal names) to hyper nodes (= sets of signal names). Here, we collect exactly these signals names to one hyper node that are part of the output interfaces of those sequential automata that are implemented on the same processor. Each hyper node then is just the "output interface" of the processor. In contrast to nodes, vertices are not affected at all from this transformation and we simply retain all dependencies. In the following, this hyper graph will be termed *processor graph*. If we collect all signals of the output interface of

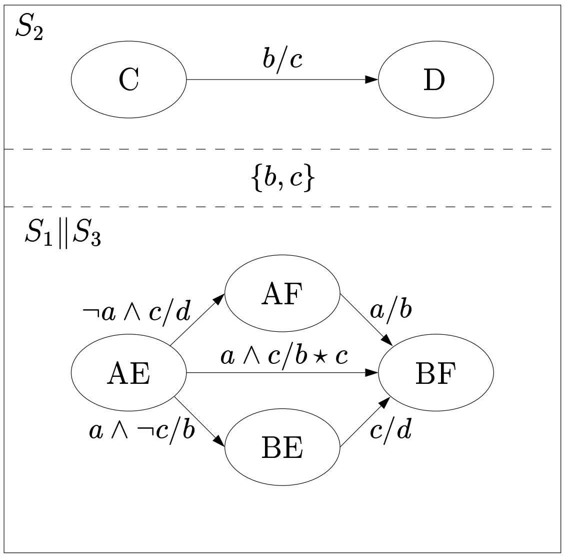


Figure 6.5: Product automaton

one single sequential automaton to one hyper node and use separate nodes even when two automata are implemented on the same processor, we speak of an *automaton graph*.

In the example (see Figure 6.4), we get the hyper nodes $\{b, c\}$ and $\{d\}$ for allocation (1) and $\{b, d\}$ and $\{c\}$ for allocation (2). Note that while the processor graph for (1) is acyclic the one for (2) contains a cycle (see Figure 6.4(b)). However, also notice that the corresponding automaton graph is not cyclic (see Figure 6.4(a)).

As a consequence, we can conclude that processor graphs also have to be free of cycles in order to enable distributed implementation. Whenever a processor graph is cyclic, one processor and therefore the finite state machine, that is, the product automaton that is implemented on this processor would have to realize two or more state transitions within one instant. Since this behavior is in general impossible for finite state machines, we have to avoid cyclic processor graphs by static analysis. Recall our results in Section 5.4.3 and notice that, due to Theorem 5.4.4, acyclic processor graphs imply efficient schedules. As a consequence, the analysis of the dependence graph only has to be carried out once in order to solve both problems. This is a further advantage of our approach.

However, to find implementable solutions by analysis of the processor graph is not always a practical solution for control-oriented specifications with possibly many configurations, since an allocation is static and therefore configuration independent. To forbid certain allocations due to the above phenomenon might reduce the number of feasible allocations to a minimum. Different configurations could require contradictory allocations and finally potentially no allocation at all would be feasible. Often the unique solution is not to implement composed automata as a product automaton, but to look for implementations, where the above problem does not arise. Then, the only possible solution is to implement each automaton by a separate finite state machine — even if more than one state machine are implemented on one processor. However, this implementation strategy has the disadvantages discussed in [DY91].

While the problem discussed above potentially occurs when one tries to group certain automata together, build the corresponding product automaton, and then implement this product automaton on one processor, in the following, we will get to know a further problem that may occur just in those cases if two automata are not implemented on the same processor. Therefore, the phenomenon we analyze in the sequel, is orthogonal to the one discussed above. However, though being orthogonal, we will see that both problems can be detected and solved with the same technique that is based on analysis of the dependence graph. Here, we do not construct the processor graph of the dependence graph, but the automaton graph instead. As a processor graph V_P collects more signals of a dependence graph to one hyper node than the corresponding automaton graph V_A, it is obvious that V_P potentially can have more cycles than V_A.

Problems caused by Instantaneous Dialog

Figure 5.10 illustrates a μ-Chart whose distributed execution causes problems that are even worse and cannot be remedied by implementing each automaton as single finite state machine as described above. The example is known from [Mar92] as instantaneous dialog. This example is a monotonic specification, whose signal graph does not contain any cycles with causality errors (see Figure 5.18). Thus, the corresponding dependence graph is acyclic. However, this example nevertheless demonstrates a severe implementation problem.

We assume that the chart in Figure 5.10 is in its initial configuration (AA,XX) and that the external stimulus is $\{x\}$. In order to determine the reaction of this μ-Chart, one has to decide whether the internal signals a and y are present in the current instant. Indeed, the only possible step is that both automata change their states from AA and XX to BB and YY, respectively and simultaneously emit a and y. Thus, both internal signals are present.

Since we want to implement both automata on different processors, we now want to understand this step as chain reaction. In the first micro-step of the step, the upper automaton in Figure 5.10 outputs $\{a\}$ while remaining in state AA.

In the second micro-step, the lower automaton, being triggered by the internal signal a, changes its current configuration from XX to YY while emitting y. In the subsequent micro-step, both x and y are present and therefore also the upper automaton changes its configuration from AA to BB. This is the critical part of the reaction because the upper automaton withdraws its first “decision” to fire the self-loop transition.

However, it is neither possible that a finite state machine carries out two state transitions in one instant nor that it withdraws the first state transition in order to perform another. Though this specification has a well-defined semantics and a least fixed-point, namely $\{a, y\}$, it is not possible to implement both automata of this example on different processors. The only possibility is either to reject specifications of this kind or to implement them on one single processor. In the instantaneous dialog example, this would mean to realize the entire specification on one single processor, that is, by one finite state machine.

Like the problem that was caused by construction of product automata, also this one can be detected by analysis of the dependence graph. However, in this case, it is sufficient to construct the automaton graph because we immediately detect a cycle in this graph. In the case of instantaneous dialog, it is not necessary to deal with a particular processor mapping in order to get a cyclic hyper graph. Here, the problem occurs because two sequential automata are separated and not implemented on the same processor. In contrast to the automata graph, the processor graph does not take into account a specific processor mapping. By the aid of the processor graph, we can determine whether a specific mapping, where certain automata are implemented on the same processor, is feasible or not.

We can conclude that apart from the techniques presented in Chapter 5 the analysis of automaton graph and processor graph, respectively is sufficient to determine whether specific distributed implementations are feasible.

6.3 Conclusion

In this chapter, we have presented a technique for centralized and later for distributed code generation. The implementation scheme we aim at is based on finite state machines. This approach is general enough to build a basis for implementations in both hardware and software and is therefore also suited for mixed hardware/software designs. While centralized code generation is relatively easy and does not cause any problems, a number of adverse effects can occur in the case of distributed code generation. Recall that the problems discussed in this chapter are orthogonal to the ones analyzed in Chapter 5. All these problems are caused by the fact that finite state machines can only perform one single state transition per clock cycle. We have discussed possibilities to detect these problems and to decide whether a specific distributed implementation is feasible. Like the analysis techniques of Chapter 5, the techniques presented in this Chapter are also based on dependence graphs that can be computed from signal graphs. Therefore, we can provide a homogeneous framework of analysis techniques for distributed implementation of specifications in μ-Charts.

It is now an interesting question to ask whether the designer should first find an (almost) optimal allocation (as indicated in Section 5.3) and then verify whether this allocation is feasible (Section 6.2) when developing a distributed implementation, or just the other way around. Though, in principle, both alternatives are possible, we propose the following strategy. Since the automaton graph can be built independently of a specific allocation, we suggest to calculate automata graphs first in order to evaluate the feasible allocations. It is then straightforward to restrict the number of feasible allocations in the ILP by introducing additional requirements. If two charts S_i and S_j with $i \neq j$

cannot be allocated to one processor but have to be implemented separately, we simply require:

$$\sum_{k=1}^{K} x_{ki} \cdot x_{kj} = 0$$

Otherwise, if both cannot be separated, we add the following equation to the model without multiple computations:

$$\sum_{k=1}^{K} x_{ki} \cdot x_{kj} = 1$$

In the model with multiple computations we write:

$$\sum_{k=1}^{K} x_{ki} \cdot x_{kj} \geq 1$$

Then, an optimal allocation can be calculated. Finally, examining the appropriate processor graph, it has to be verified whether this particular allocation is feasible (see Section 6.2).

Comparison: Centralized versus Distributed Implementation

After having presented both centralized and distributed implementation we can now compare them with each other. In particular, we will give a classification for reactive systems specified in μ-Charts or any other synchronous language dependent on the target architecture. To this end, we briefly highlight the restrictions having discussed in Chapters 5 and 6 again.

To overcome the problems with *external input signals* of distributed implementations we have outlined in Section 5.2.2, we have proposed two different strategies. We can either restrict the feasible trigger conditions to conditions that only can react on single input signals or we can make assumptions to the system environment by using an rely/guarantee-like specification style. Of course, in principle, a centralized implementation has to overcome the same problem because its input interface also has to collect signals to events, that is, sets of signals. However, this can be done much easier by a clock signal in the case of centralized implementation. Hence, the problem with external input signals can be avoided for centralized implementations. As we have seen, using the same clock for all processors of a loosely coupled distributed target architecture is not possible due to technical reasons. The same argument with the global clock also applies to the problem of dealing with *signal absences:* using only one single clock in a centralized implementation enables the scanning of signal absences.

However, the situation is somewhat different in case of the problems *internal input signals* cause in distributed implementations. While for centralized implementations it is sufficient to know that a specification or, more precisely, its semantics, is continuous (recall that monotonicity immediately implies continuity here since we deal with finite sets of input and output signals), we had to require additionally for distributed implementations that their signal graphs do not contain any harmful cycle in any reachable configuration. This is necessary to avoid communication deadlocks. For centralized implementations, this kind of causality analysis is not needed. Here, already monotonicity alone is adequate, because monotonic charts always have a least fixed point. In order to generate a centralized implementation this is a sufficient property. One specification of this type is the example with parallel labels a/b and b/a we have discussed in Section 5.1. It can be implemented on a centralized, but not on a distributed architecture. Since signal graphs can only be constructed for monotonic charts, the existence of a signal graph without harmful cycles also guarantees the chart's monotonicity.

In addition, in Section 5.4.2, we have realized that it is necessary to require disjoint output interfaces when partitioning a specification. Here, the interfaces of those parts of the specification that are provided for distributed implementation have to be disjoint in order to apply Theorem 5.4.3. Informally, this is necessary in order to determine at any point of the chain reaction which part already has sent a message and which has not. In practice, however, this is no restriction at all as signals can be renamed easily. Of course, this renaming is not needed in case of centralized implementations.

Finally, all problems caused by finite state machines and discussed in Section 6.2, of course do not emerge in centralized implementations.

7 Concluding Remarks

In this chapter, we review the results obtained in this thesis. In addition, we give an outline of future work in the area of design of reactive systems, as well as their distributed implementation, with Statecharts-like languages.

7.1 Summary

This work was driven by the idea of supporting a design method for Statecharts that covers the design phases description, refinement, verification, and generation of centralized and distributed code, independently of whether the code is implemented in hardware or software. As a first step towards this goal we developed the visual formalism μ-Charts, a dialect of Harel's Statecharts. In contrast to Statecharts and many related approaches, μ-Charts overcome many semantic problems inherent in the former notation. We defined a formal, stream-based semantics for μ-Charts, discussed different semantic properties, and showed how to extend the μ-Charts core language with syntactic abbreviations. The most important example of these is hierarchical decomposition. The core language of μ-Charts consists of only three constructs: sequential automata, signal hiding, and a composition operator. Here, the last is the most subtle concept of the three. It combines parallel composition and multicast communication in one single language construct. Starting with a lean core language and then extending it by means of abbreviation mechanisms has several advantages. First, it eases the task of obtaining specifications that can be partitioned and implemented on a distributed processor network. Second, it eases reasoning about the semantics itself.

When defining the refinement calculus for μ-Charts we also took advantage of this property. To support the designer in specifying reactive systems, we made a set of syntactic refinement rules available to him or her. Each of the rules guarantees that by its application the overall specification only becomes more concrete, never more abstract. Whether a specification in μ-Charts is more concrete than another one is defined by its set of input/output histories. We say that the μ-Chart S_2 refines chart S_1 if its semantics, that is, its set of input/output histories is a subset of the semantics of S_1. This notion of refinement is compositional and transitive. Therefore, first, refinements of parts of the specification are also refinements of the overall specification, and second, systems can be developed incrementally. Since all rules are syntactic, the user does not have to carry out complicated proofs on the semantic level, but can just verify whether all syntactic conditions for the application of a rule are fulfilled. This task can be easily automated and is an important step towards a system design process that guarantees correctness by construction.

However, not only do we assist the software engineer with a refinement calculus, but also by providing a framework for formal verification of μ-Chart specifications with model checking techniques. We presented a general translation scheme for μ-Charts to input languages of model checkers by defining μ-Charts' behavior by means of finite transition relations. We then went a step further and showed how to translate these relations to the input languages of two specific verifiers, μcke [Bie97b] and SMV [McM93], and so provided direct tool support for verification.

This work also contributes to the generation of centralized and distributed code from Statecharts specifications. Both compilation techniques are based on finite state machines and are therefore suited for hardware, software, or even mixed realizations. While generating centralized code is not complicated, a number of conditions have to be borne in mind if one wants to produce distributed implementations. First of all, we pointed out restrictions that have to be fulfilled in order to obtain distributed implementations that have the same behavior as a centralized. Then, we provided a model to describe and optimize the allocation process. In the allocation process, for each sequential automaton and each communication an appropriate target processor and bus, respectively, is determined. Finally, apart from optimizing the allocation, we also presented a technique to compute efficient communication schedules.

In summary, the main contribution of this thesis is the development of a lean dialect of Statecharts that solves the semantic problems of many other related languages, but is still powerful enough to describe practically relevant systems and is at the same time appropriate for refinement, formal verification, and, potentially distributed, implementation while, at the same time, combining all these phases to a seamless design process.

7.2 Future Work

In the design process presented here, we factored out the requirements engineering phase. Hence, there is still a gap between this phase and the state-based description of a reactive system with μ-Charts. This gap can be filled, for instance, by applying techniques that are based on message sequence charts (MSCs). In contrast to μ-Charts, these provide a possibility for event- or scenario-based development of a system in its early design phase. In order to get a seamless design process that also covers requirements engineering, it would be interesting to develop an appropriate MSC-like description technique that is based on the same semantic model as μ-Charts and therefore can be automatically translated to an equivalent μ-Chart specification. Furthermore, these MSC-like techniques could serve to represent counter examples and witnesses from model checking results, or simply as a front end for the input language of an appropriate model checker.

Though our model checking results are quite promising, for large applications it may also be necessary to further improve the translation to model checkers to obtain even more efficient automatic verification. Another way to achieve more efficiency here is of course to investigate compositional verification techniques. Here, it would seem that a promising avenue of exploration would be to apply the techniques we presented for partitioning to compositional proving.

Our refinement calculus only allows the correct refinement of specifications without considering the preservation of liveness properties. Though the property that "nothing bad will happen", that is, safety properties are preserved after applying one of our rules, the feature that "something good will happen" (liveness property) may be affected by refinement. The reason for this is because the refined component in general provides less input/output histories than the corresponding abstract one. Therefore, an interesting investigation is to extend your calculus by rules that include invariants representing liveness conditions which are true before and after applying it to a specific specification.

A further interesting direction for future work is the investigation of other compilation techniques for μ-Charts that are not based on the generation of finite state machines. As for STATEMATE [Har90, i-L90, HN96, PU97, i-L97], hardware generation techniques that are based on VHDL could be envisaged. Since VHDL code generation of STATEMATE is not very efficient, our approach could thus gain more importance for practical applications.

All the aforementioned extensions would further support the user in developing error-free reactive systems. It would also be of interest to give more methodological support. An interesting extension of this work would be to develop a cookbook-like methodology for the development of reactive systems with Statecharts-like languages. We hope that this is easily possible with the methodological framework we have given in this thesis. Apart from more methodological assistance, the approach presented in this thesis could be applied more easily by engineers working in industry if more tool support was available. Promising future work therefore seems to be the development of prototype tools.

Last but not least, we would like to mention once more that a number of those problems we discussed in Sections 5 and 6.2 and that occur when a distributed implementation is envisaged, can be avoided when using delayed feedback instead of instantaneous feedback as the semantic model for communication. Hence, it would be interesting to introduce an additional composition operator that is based on delayed feedback. This idea is realized in the synchronous language "Reactive C" [Bou91], for example. One thus could use instantaneous feedback to model local, intra-processor communication and delayed feedback for inter-processor multicasting. However, it has to be borne in mind that this means restricting the number of design alternatives in the early design stage. Moreover, recall that hierarchical decomposition cannot be defined by means of a composition operator that is based on delayed feedback. In providing different communication delays, CFSMs [CGH+93a, CGH+93b, CGH+94] even go one step further: they allow arbitrary event durations and therefore arbitrary communication delays to be defined.

We hope that interested researchers in related topics feel encouraged to work out some of the here proposed directions for future work.

Bibliography

[AL91] A. Asperti and G. Longo. *Categories, Types, and Structures*. Foundations of Computing. The MIT Press, 1991.

[BCG86] G. Berry, P. Couronné, and G. Gonthier. Synchronous Programming of Reactive Systems: An Introduction to Esterel. Technical Report 647, INRIA, 1986.

[BCLH93] A. Benveniste, P. Caspi, P. LeGuernic, and N. Halbwachs. Data-flow Synchronous Languages. In *A Decade of Concurrency*, volume 803 of *Lecture Notes in Computer Science*, Noordwijkerhout, The Netherlands, June 1993. Springer.

[BCM+90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the Conference on Logic in Computer Science*, pages 428–439, Philadelphia, June 1990. IEEE Computer Society Press.

[BDD+93] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. Gritzner, and R. Weber. The Design of Distributed Systems An Introduction to FOCUS - Revised Version. Technical Report TUM-I9202-2, Technische Universität München, Fakultät für Informatik, TUM, 80290 München, Germany, January 1993.

[BdS91] F. Boussinot and R. de Simone. The Esterel Language. *Proceedings of the IEEE*, 79(9):1293–1304, September 1991.

[Ber89] G. Berry. Real Time Programming: Special Purpose or General Purpose Languages. *Information Processing 89*, 1989.

[Ber91] G. Berry. A Hardware Implementation of Pure Esterel. Rapport de Recherche 06/91, Ecole des Mines, CMA, Sophia-Antipolis, France, 1991.

[Ber92] G. Berry. A Hardware Implementation of Pure Esterel. *Sadhana, Academy Proceedings in Engineering Sciences, Indian Academy of Sciences*, 17(1):95–130, 1992.

[Ber93] G. Berry. Preemption in Concurrent Systems. In *Foundations of Software Technology and Theoretical Computer Science : 13th Conference Bombay,* *India, December 15-17*, volume 761 of *Lecture Notes in Computer Science*, pages 72 – 93. Springer, 1993.

[Ber96] G. Berry. *A Quick Guide to Esterel.* Unpublished Esterel Primer, 1996.

[Ber98] G. Berry. The Foundations of Esterel. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner.* MIT Press, 1998.

[BG88] G Berry and G. Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. Technical Report 842, INRIA, 1988.

[BG92] G. Berry and G. Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming*, 19:87–152, 1992.

[BG97] A. Benveniste and P. Le Guernic. Compositionality in Dataflow Synchronous Languages: Specification and Code Generation. In *Proceedings of the International Symposium "Compositionality — The Significant Difference" (COMPOS'97).* Christian Albrecht University at Kiel, Germany, August 1997.

[Bie97a] A. Biere. *Effiziente Modellprüfung des μ-Kalküls mit binären Entscheidungsdiagrammen* (in German). PhD thesis, Universität Karlsruhe, 1997.

[Bie97b] A. Biere. μcke — Efficient μ-Calculus Model Checking. Number 1254 in Lecture Notes in Computer Science, pages 468–471, 1997.

[Bou91] F. Boussinot. Reactive C: An Extension of C to Program Reactive Systems. *Software-Practice and Experience*, 21(4):401–428, April 1991.

[Bro93] M. Broy. Interaction Refinement - The Easy Way. In *Program Design Calculi*, volume 118 of *NATO ASI Series F: Computer and System Sciences.* Springer, 1993.

[Bro97a] M. Broy. *Informatik (Eine grundlegende Einführung)* (in German), volume 1. Springer, Berlin, 2nd edition, 1997.

[Bro97b] M. Broy. The Specification of System Components by State Transition Diagrams. Technical Report TUM-I9729, Technische Universität München, 1997.

[Bry86] R. E. Bryant. Graph Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 8(C-35):677–691, 1986.

[BS97] M. Broy and K. Stølen. FOCUS on System Development – A Method for the Development of Interactive Systems, 1997. Manuscript.

[Buc95] K. Buchenrieder. *Hardware/Software Codesign (An Annotated Bibliography)*. IT Press Chicago, Bruchsal, 1995.

[BW98] U. Brockmeyer and G. Wittich. Tamagotchis need not die – Verification of Statemate Designs. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98), March, Lisbon (Portugal)*, volume 1382 of *Lecture Notes in Computer Science*, 1998.

[CCGJ97] B. Caillaud, P. Caspi, A. Girault, and C. Jard. Distributing Automata for Asynchronous Networks of Processors. *European Journal of Automation (RAIRO-APII-JESA)*, 31(3):503–524, 1997.

[CE81] E.M. Clarke and E.A. Emerson. Characterizing Properties of Parallel Programs as Fixpoints. Number 85 in Lecture Notes in Computer Science, pages 169–181. Springer, 1981.

[CFG95] P. Caspi, J.-C. Fernandez, and A. Girault. An Algorithm for Reducing Binary Branchings. In P.S. Thiagarajan, editor, *Fifteenth Conference on the Foundations of Software Technology and Theoretical Computer Science, FST&TCS'95*, volume 1026 of *Lecture Notes in Computer Science*, Bangalore, India, December 1995. Springer Verlag.

[CGH+93a] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, and A. Sangiovanni-Vincentelli. A Formal Specification Model for Hardware/Software Codesign. Technical Report UCB/ERL M93/48, UC Berkeley, 1993.

[CGH+93b] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, and A. Sangiovanni-Vincentelli. Synthesis of Mixed Software-Hardware Implementations from CFSM Specifications. Technical Report UCB/ERL M93/49, UC Berkeley, 1993.

[CGH+94] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, A. Sangiovanni-Vincentelli, E. Sentovich, and K. Suzuki. Synthesis of Software Programs for Embedded Control Applications. Technical Report UCB/ERL M94/87, UC Berkeley, 1994.

[CGP94] P. Caspi, A. Girault, and D. Pilaud. Distributing Reactive Systems. In *7th International Conference on Parallel and Distributed Computing Systems, PDCS'94, Las Vegas, USA*. ISCA, 1994.

[CHB92] D. Coleman, F. Hayes, and S. Bear. Introducing Objectcharts or How to Use Statecharts in Object-Oriented Design. *IEEE Transactions on Software Engineering*, 18(1):9–18, 1992.

[CHF97] C.-T. Chou, J.-L. Huang, and M. Fujita. A High-Level Language for Programming Complex Temporal Behaviors and its Translation into Synchronous Circuits. 1997. Hardware Description Languages and their Application (CHDL'97), Toledo.

[Cpl94] Cplex Optimization Inc. *Using the Cplex Callable Library.* Cplex Optimization Inc., 1994.

[CR94] M.-M. Corsini and A. Rauzy. Symbolic Model Checking and Constraint Logic Programming: A Cross-Fertilization. ESOP'94, pages 180–194, 1994.

[Dam94] M. Dam. CTL* and ECTL* as Fragments of the Modal μ-Calculus. Number 126 in Theoretical Computer Science, 1994.

[Day93] N. Day. *A Model Checker for Statecharts (Linking CASE tools with Formal Methods).* Master's thesis, University of British Columbia, 1993.

[DS96] G. DeMicheli and M. Sami. *Hardware/Software Co-Design*, volume 310 of *NATO ASI Series E: Applied Sciences.* Kluwer Academic Publishers, 1996.

[DY89] D. Drusinsky-Yoresh. Using Statecharts for Hardware Description and Synthesis. Number 8 in IEEE Transactions on Computer-Aided Design, pages 798–807, 1989.

[DY91] D. Drusinsky-Yoresh. A State Assignment for Single-Block Implementation of State Charts. Number 10 in IEEE Transactions on Computer-Aided Design, pages 1569–1576, 1991.

[Edw97] S.A. Edwards. *The Specification and Execution of Heterogeneous Synchronous Reactive Systems.* PhD thesis, University of California at Berkeley, 1997.

[EFT92] H.-D. Ebbinghaus, J. Flum, and W. Thomas. *Einführung in die mathematische Logik* (in German). BI-Wissenschaftsverlag, 3rd edition, 1992.

[EGKP97] H. Ehrig, R. Geisler, M. Klar, and J. Padberg. Horizontal and Vertical Structuring Techniques for Statecharts. In A. Mazurkiewicz and J. Winkowski, editors, *CONCUR'97: Concurrency Theory, 8^{th} International Conference, Warsaw, Poland*, volume 1243 of *Lecture Notes in Computer Science*, pages 181–195. Springer, July 1997.

[EH86] E. A. Emerson and J. Y. Halpen. Sometimes and Not Never Revisited: On Branching versus Linear Time Temporal Logic. *Journal of the ACM*, 33(1):151–178, 1986.

[Eme90] E.A. Emerson. *Handbook of Theoretical Computer Science*, volume B, chapter 16. Temporal and Modal Logic, pages 995–1072. The MIT Press, 1990.

[Ets94] K. Etschberger. *CAN Controller-Area-Network: Grundlagen, Protokolle, Bausteine, Anwendungen* (in German). Carl Hanser Verlag, München, 1994.

[Fra86] N. Francez. *Fairness*. Springer, Berlin, 1986.

[Fre90] P. Freyd. Recursive Types Reduce to Inductive Types. In *Proceedings of the fifth annual IEEE symposium on Logic in Computer Science*. 1990.

[Fuc94] M. Fuchs. *Technologieabhängigkeit von Spezifikationen digitaler Hardware* (in German). PhD thesis, Technische Universität München, 1994.

[Fuc97] M. Fuchs. The Architecture of Distributed Systems in Automobile Industry, 1997. Private Communication.

[Gir94] A. Girault. *Sur la Répartition de Programmes Synchrones* (in French). PhD thesis, INPG, Grenoble, France, January 1994.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, San Francisco, 1979.

[GKRB96] R. Grosu, C. Klein, B. Rumpe, and M. Broy. State Transition Diagrams. Technical Report TUM-I9630, Technische Univerität München, 1996.

[GLT89] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge University Press, 1989.

[Gon86] D. W. Gonzalez. *Ada Programmer's Handbook and Language Reference Manual*. The Benjamin/Cummings Publishing Company, 1986.

[GSB98] R. Grosu, Gh. Stefanescu, and M. Broy. Visual formalisms revisited. In *CSD '98, International Conference on Application of Concurrency to System Design, Aizu-Wakamatsu City, Fukushima*. IEEE, March 1998.

[GSSAL93] R. Gawlick, R. Segala, J.F. Sogaard-Andersen, and N.A. Lynch. Liveness in Timed and Untimed Systems. Technical Report MIT/LCS/TR-587, Laboratory for Computer Science, MIT, Cambridge, MA., December 1993. Extended abstract in Proceedings ICALP'94.

[Gun92] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. The MIT Press, 1992.

[GW94] P. Godefroid and P. Wolper. A Partial Approach to Model Checking. *Information and Computation*, (110):305–326, 1994.

[Hal93a] N. Halbwachs. Delay Analysis in Synchronous Programs. In *Fifth Conference on Computer-Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, Elounda (Greece), July 1993. Springer.

[Hal93b] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academics Publishers, Dordrecht, Boston, London, 1993.

[Har87] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231–274, 1987.

[Har90] D. Harel. STATEMATE: A Working Environment for the Development of Complex Reactive Systems. *IEEE Transactions on Software Engineering*, 16:403–413, 1990.

[Har95] E. Harbeck. *Erweiterung und Verbesserung von verschiedenen Zuweisungsverfahren für den Entwurf von Mehrprozessorsystemen mit "harten" Echtzeitanforderungen* (in German). Master's thesis, Universität Passau, 1995.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The Synchronous Dataflow Programming Language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[HdR91] C. Huizing and W.-P. de Roever. Introduction to Design Choices in the Semantics of Statecharts. *Information Processing Letters*, 37, 1991.

[HFB93] N. Halbwachs, J.-C. Fernandez, and A. Bouajjanni. An Executable Temporal Logic to Express Safety Properties and its Connection with the Language Lustre. In *Sixth International Symposium on Lucid and Intensional Programming, ISLIP'93, Quebec*, April 1993.

[HG91] C. Huizing and R. Gerth. Semantics of Reactive Systems in Abstract Time. In J.W. de Bakker, C. Huizing, W.P. de Roever, and G. Rozenberg, editors, *Real-Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, pages 291–314, Mook - The Netherlands, June 1991.

[HH83] E.C.R. Hehner and C.A.R. Hoare. A More Complete Model of Communicating Processes. In *Theoretical Computer Science*, volume 1-2, pages 105–120, North-Holland, Amsterdam, September 1983.

[HJH90] J. He, M.B. Josephs, and C.A.R. Hoare. A Theory of Synchrony and Asynchrony. In M. Broy and C.B. Jones, editors, *Programming Concepts and Methods*, pages 459–478, North-Holland, Amsterdam, 1990.

[HN96] D. Harel and A. Naamad. The Statemate Semantics of Statecharts. *ACM Transactions On Software Engineering and Methodology*, 5(4):293–333, 1996.

[Hoa78] C.A.R. Hoare. Communicating Sequential Processes. In *Communications of the ACM*, volume 8, pages 666–677, 1978.

[HP85] D. Harel and A. Pnueli. On the Development of Reactive Systems. In *Logics and Models of Concurrent Systems*, volume 13 of *NATO ASI Series F: Computer and System Sciences*, pages 477–498. Springer, 1985.

[HRdR92] J.J.M. Hooman, S. Ramesh, and W.P. de Roever. A Compositional Axiomatization of Statecharts. *Theoretical Computer Science*, 101:289–335, 1992.

[HSD+93] J. Helbig, R. Schlör, W. Damm, G. Döhmen, and P. Kelb. VHDL/S — Integrating Statecharts, Timing Diagrams, and VHDL. *Microprocessing and Microcomputing*, (38):571–580, 1993.

[i-L90] i-Logix Inc., Three Riverside Drive, Andover, MA 01810, U.S.A., http://www.i-logix.com. *Languages of Statemate*, 1990.

[i-L97] i-Logix Inc., Three Riverside Drive, Andover, MA 01810, U.S.A., http://www.i-logix.com. *Magnum*, 1997.

[i-L98] i-Logix Inc., Three Riverside Drive, Andover, MA 01810, U.S.A., http://www.i-logix.com. *Rhapsody*, 1998.

[Jos88] M.B. Josephs. A State-based Approach to Communicating Processes. *Distributed Computing*, (3):9–18, 1988.

[Kal96] M. Kaltenbach. *Interactive Verification Exploiting Program Design Knowledge: A Model-Checker for UNITY.* PhD thesis, The University of Texas at Austin, 1996.

[Kel96] P. Kelb. *Abstraktionstechniken für automatische Verifikationsmethoden* (in German). PhD thesis, Carl von Ossietzky Universität Oldenburg, 1996.

[Kle97] C. Klein. *Anforderungsspezifikation durch Transitionssysteme und Szenarien* (in German). PhD thesis, Technische Universität München, 1997.

[KN84] H. Kasahara and S. Navita. Practical Multiprocessor Scheduling Algorithms for Efficient Processing. *IEEE Transactions on Computers*, 33(11), 1984.

[Kro94] T. Kropf. Benchmark Circuits for Hardware-Verification. In R. Kumar and T. Kropf, editors, *Theorem Provers in Circuit Design*, volume 901 of *Lecture Notes in Computer Science*, pages 1–12. Springer-Verlag, 1994.

[Lev97] F. Levi. *Verification of Temporal and Real-Time Properties of Statecharts.* PhD thesis, Universitá Degli Studi di Pisa, 1997.

[LHHR94] N.G. Leveson, M.P.E. Heimdahl, H. Hildreth, and J.D. Reese. Requirements Specification for Process-Control Systems. *IEEE Transactions on Software Egnineering*, 20(9):864–707, 1994.

[LS86] J. Lambek and P.J. Scott. *Introduction to Higher Order Categorical Logic*. Cambridge University Press, 1986.

[LT87] N.A. Lynch and M.R. Tuttle. Hierarchical Correctness Proofs for Distributed Algorithms. Technical Report MIT/LCS/TR-387, Laboratory for Computer Science, MIT, Cambridge, MA., April 1987.

[LT89] Nancy Lynch and Mark Tuttle. An Introduction to Input/Output Automata. *CWI Quarterly*, 2(3):219–246, 1989.

[LV95] N.A. Lynch and F. Vaandrager. Forward and Backward Simulations – Part I: Untimed Systems. *Information and Computation*, 121(2):214–233, 1995.

[LV96] N.A. Lynch and F. Vaandrager. Forward and Backward Simulations – Part II: Timed Systems. *Information and Computation*, 128(1):1–25, 1996.

[Mar92] F. Maraninchi. Operational and Compositional Semantics of Synchronous Automaton Compositions. In W.R. Cleaveland, editor, *Proceedings CONCUR'92*, volume 630 of *Lecture Notes in Computer Science*, pages 550 – 564. Springer-Verlag, 1992.

[McM93] Kenneth L. McMillan. *Symbolic Model Checking*. PhD thesis, Carnegie Mellon University, 1993.

[MH96] F. Maraninchi and N. Halbwachs. Compositional Semantics of Nondeterministic Synchronous Languages. In Riis Nielson, editor, *Programming languanges and systems - ESOP'96, 6th European Symposium on programming*, volume 1058 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[Mil83] G.J. Milne. CIRCAL and the Representation of Communication, Concurrency and Time. Technical Report CSR-151-83, University of Edinburgh, Department of Computer Science, November 1983.

[Mil84] G.J. Milne. A Model for Hardware Description and Verification. In *Design Automation Conference, New Mexico*, 1984.

[Möl85] B. Möller. On the Algebraic Specification of Infinite Objects - Ordered and Continuous Models of Algebraic Types. *Acta Informatica*, 22:537–578, 1985.

[MPS95] A.K. Mok, C. Puchol, and D. Stuart. Compiling Modechart Specifications. In *Real-Time Systems Symposium (RTSS '95), Pisa, Italy*. Proceedings of the IEEE, 1995.

[Mül98] O. Müller. *A Verification Environment for I/O Automata based on Formalized Meta-Theory*. PhD thesis, Technische Universität München, 1998.

[NRS96] D. Nazareth, F. Regensburger, and P. Scholz. Mini-Statecharts: A Lean Version of Statecharts. Technical Report TUM-I9610, Technische Universität München, D-80290 München, 1996.

[NT93] M.G. Norman and P. Thanisch. Models of Machines and Computation for Mapping in Multicomputers. *ACM Computer Surveys (3)*, 25, 1993.

[Pau87] L.C. Paulson. *Logic and Computation, Interactive Proof with Cambridge LCF*, volume 2 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1987.

[Pau94] L.C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.

[PC87] D. Pilaud P. Caspi. Lustre: A Declarative Language for Programming Synchronous Systems. In *14th ACM Conf. on Principles of Programming Languages*, Munich, January 1987.

[Per93] A. Peron. *Synchronous and Asynchronous Models for Statecharts.* PhD thesis, Universita di Pisa-Genova-Udine, 1993.

[Per95] A. Peron. Statecharts, Transition Structures and Transformations. In Peter D. Mosses, Mogens Nielsen, and Michael I. Schwartzbach, editors, *TAPSOFT'95: Theory and Practice of Software Development, 6th International Joint Conference CAAP/FASE, Aarhus, Denmark*, volume 915 of *Lecture Notes in Computer Science*, pages 454–468. Springer, 1995.

[PH88] D. Pilaud and N. Halbwachs. From a Synchronous Declarative Language to a Temporal Logic Dealing with Multiform Time. In *Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 331 of *Lecture Notes in Computer Science*, Warwick, September 1988. Springer.

[Pnu77] A. Pnueli. The Temporal Logic of Programs. In *18th Annual Symposium on Foundations of Computer Science*, pages 46–57, New York, June 1977. IEEE Computer Science Press.

[PS91] A. Pnueli and M. Shalev. What is in a Step: On the Semantics of Statecharts. In T. Ito and A.R. Meyer, editors, *Proccedings of the "Theoretical Aspects in Computer Software 91"*, volume 526 of *Lecture Notes in Computer Science*, pages 244 – 264. Springer-Verlag, 1991.

[PS97a] J. Philipps and P. Scholz. Compositional Specification of Embedded Systems with Statecharts. In *TAPSOFT'97: Theory and Practice of Software Development*, volume 1214 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[PS97b] J. Philipps and P. Scholz. Formal Verification of Statecharts with Instantaneous Chain Reactions. In *TACAS'97: Tools and Algorithms for the Construction and Analysis of Systems*, volume 1217 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[PU97] C. Petersohn and L. Urbina. A Timed Semantics for the Statemate Implementation of Statecharts. In *Formal Methods Europe (FME'97)*, Lecture Notes in Computer Science. Springer, 1997.

[RB95] J. Rozenblit and K. Buchenrieder. *Codesign*. IEEE Press, 1995.

[Reg94] F. Regensburger. *HOLCF: Eine konservative Erweiterung von HOL um LCF*. PhD thesis, Technische Universität München, 1994.

[Reg95] F. Regensburger. HOLCF: Higher Order Logic of Computable Functions. In E. Thomas Schubert, Phillip J. Windley, and James Alves-Foss, editors, *Higher Order Logic Theorem Proving and Its Application (HOL'95)*, volume 971 of *Lecture Notes in Computer Science*, pages 293–307. Springer, 1995.

[Rei86] W. Reisig. *Petrinetze – Eine Einführung*. Studienreihe Informatik. Springer Verlag, Berlin/Heidelberg/New York/Tokyo, 1986.

[Rob86] F. Robert. *Discrete Iterations: A Metric Study*, volume 6 of *Springer Series in Computational Mathematics*. Springer, Stuttgart, 1986.

[Rum96] B. Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme (in German)*. PhD thesis, Technische Universität München, 1996.

[Sav92] M.W.P. Savelsbergh. *Functinal Description of Minto, a Mixed INTeger Optimizer (Version 1.5a)*. Eindhoven University of Technology, The Netherlands, 1992.

[SBT96] T.R. Shiple, G. Berry, and H. Touati. Constructive Analysis of Cyclic Circuits. In *International Design and Testing Conference, Paris*, 1996.

[Sch86] D.A. Schmidt. *Denotational Semantics*. Allan and Bacon, 1986.

[Sch88] A. Schütte. *Occam2-Handbuch*. B. G. Teubner Verlag, Stuttgart, 1988.

[Sch94] P. Scholz. *Statische Aufteilung von Programmoduln auf Mehrprozessorsysteme für Echtzeitanwendungen* (in German). Master's thesis, Universität Passau, 1994.

[Sch96a] P. Scholz. An Extended Version of Mini-Statecharts. Technical Report TUM-I9628, Technische Universität München, D-80290 München, 1996.

[Sch96b] P. Scholz. A Light-Weight Formalism for the Specification of Reactive Systems. In *XXIII-rd Seminar on Current Trends in Theory and Practice of Informatics (SOFSEM'96), Milovy, Slovakia*, volume 1175 of *Lecture Notes in Computer Science*, pages 425 – 432, 1996.

[Sch97] K. Schneider. CTL and Equivalent Sublanguages of CTL*. 1997. Hardware Description Languages and their Application (CHDL'97), Toledo.

[Sco76] D. Scott. Data Types as Lattices. *SIAM Journal of Computing*, 5(3), September 1976.

[Sco82] D. Scott. Domains for Denotational Semantics. Number 140 in Lecture Notes in Computer Science. Springer, 1982.

[SDW95] K. Stølen, F. Dederichs, and R. Weber. Specification and refinement of networks of asynchronously communicating agents using the assumption/commitment paradigm. *Formal Aspects of Computing*, 1995.

[SG90] D. Scott and C. Gunter. Semantic Domains and Denotational Semantics. In *Handbook of Theoretical Computer Science*, chapter 12, pages 633–674. Elsevier Science Publisher, 1990.

[SGW94] B. Selic, G. Gullekson, and P.T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, 1994.

[SH97] P. Scholz and E. Harbeck. Task Assignment for Distributed Computing. In *Advances in Parallel and Distributed Computing (APDC'97), Shanghai*. IEEE Computer Society Press, 1997.

[Shi96] T. Shiple. *Formal Analysis of Cyclic Circuits*. PhD thesis, University of California at Berkeley, 1996.

[SNR96] P. Scholz, D. Nazareth, and F. Regensburger. Mini-Statecharts: A Compositional Way to Model Parallel Systems. 1996. 9th International Conference on Parallel and Distributed Computing Systems (PDCS'96), Dijon, France.

[Spi98] K. Spies. *Eine Methode zur formalen Modellierung von Betriebssystemkonzepten* (in German). PhD thesis, Technische Universität München, 1998.

[Spr96] M. Spreng. *Rapid Prototyping elektronischer Steuerungssysteme in der Automobilentwicklung* (in German). PhD thesis, Universität Fridericiana Karlsruhe, 1996.

[SS95] B. Schätz and K. Spies. Formale Syntax zur logischen Kernsprache der Focus-Entwicklungsmethodik. Technical Report TUM-I9529, Technische Universität München, Fakultät für Informatik, 80290 München, Germany, 1995.

[Sta97] T. Stauner. *Specification and Verification of an Electronic Height Control System using Hybrid Automata.* Master's thesis, Technische Universität München, 1997.

[Ste97] R. Stephens. A Survey of Stream Processing. *Acta Informatica*, 34:491–541, 1997.

[Stø95] K. Stølen. Assumption/Commitment Rules for Data-flow Networks — with an Emphasis on Completeness. Technical Report TUMI-9516, Technische Univerität München, 1995.

[TBYS96] J.J.P. Tsai, Y. Bi, S.J.H. Yang, and R.A.W. Smith. *Distributed Real-Time Systems.* John Wiley and Sons Inc., New York, Chichester, Brisbane, Toronto, Singapore, 1996.

[Tom97] H. A. Toma. *Analyse Constructive et Optimisation Séquentielle des Circuits Générés à partir du Langage Synchrone Réactif Esterel* (in French). PhD thesis, Ecole des Mines de Paris, September 1997.

[US93] A. Uselton and S.A. Smolka. State Refinement in Process Algebra. In *2nd North American Workshop on Process Algebra, Ithaca, NY, U.S.A*, 1993.

[US94a] A. Uselton and S.A. Smolka. A Process-Algebraic Semantics for Statecharts via State Refinement. In *Proceedings of PROCOMET '94, San Miniato, Italy.* Elsevier, North Holland, 1994.

[US94b] A.C. Uselton and S.A. Smolka. A Compositional Semantics for Statecharts using Labeled Transition Systems. In Bengt Jonsson and Joachim Parrow, editors, *CONCUR'94: Concurrency Theory*, volume 836 of *Lecture Notes in Computer Science*, pages 2–17, Uppsala, August 1994. Springer-Verlag.

[vdB94] M. von der Beeck. A Comparison of Statecharts Variants. In H. Langmaack, W.-P. de Roever, and J. Vytopil, editors, *Proc. Formal Techniques in Real–Time and Fault–Tolerant Systems (FTRTFT'94)*, volume 863 of *Lecture Notes in Computer Science*, pages 128–148. Springer, 1994.

[Weg93] I. Wegener. *Theoretische Informatik* (in German). Teubner Verlag, Stuttgart, 1993.

[Win93] G. Winskel. *The Formal Semantics of Programming Languages.* The MIT Press, 1993.

Index