# On the Combination of Model- and Property-Oriented Description Techniques

Vom Fachbereich Informatik
der Universität Kaiserslautern
zur Verleihung des akademischen Grades
Doktor der Naturwissenschaften (Dr. rer. nat)
genehmigte Dissertation
von

Dipl. Inform. Thomas Deiß

Datum der wissenschaftlichen Aussprache: 11. Juni 1999

Dekan: Prof. Dr. Michael M. Richter

Promotionskommission:
Vorsitzender: Prof. Dr. Reinhard Gotzhein
Berichterstatter: Prof. Dr. Klaus E. Madlener
Prof. Dr. Gerhard Zimmermann

D-386

# Zusammenfassung


Um Eigenschaften eines Systems zu beschreiben, werden verschiedene Beschreibungstechniken benötigt. Dies gilt für Eigenschaften, die zu unterschiedlichen Aspekten eines Systems gehören. Aber selbst um Eigenschaften zu beschreiben, die zu einem Aspekt gehören, genügt meist eine einzelne Beschreibungstechnik nicht.

In dieser Arbeit untersuchen wir, wie sich Beschreibungstechniken zur Beschreibung des Aspektes "Verhalten" reaktiver Systeme kombinieren lassen. Dies geschieht anhand einer Kombination der beiden Beschreibungstechniken statecharts und TLA.

Die Beschreibungstechniken statecharts und TLA unterscheiden sich sowohl in Bezug auf den Beschreibungsstil als auch darin, wann Änderungen von Größen wirksam werden. Die Beschreibungstechnik statecharts steht für modellorientierte Beschreibungstechniken. Mit solchen Beschreibungstechniken werden Eigenschaften eines Systems durch Angabe abstrakter Modelle beschrieben. TLA steht für eigenschaftsorientierte Beschreibungstechniken, mit denen Eigenschaften meist durch logische Formeln explizit ausgedrückt werden. In Bezug auf das Wirksamwerden von Änderungen unterscheiden sich statecharts und TLA darin, daß in statecharts die Änderungen erst nach ihren Auslösern wirksam werden. In TLA ist es hingegen möglich zu beschreiben, daß eine Änderung synchron, d.h. zum gleichen Zeitpunkt, mit ihrem Auslöser stattfindet. Wegen dieser Unterschiede ist es nicht offensichtlich, wie statecharts und TLA sinnvoll kombiniert werden können.

In der vorliegenden Arbeit werden statecharts und TLA auf syntaktischer Ebene dadurch kombiniert, daß TLA-Formeln mit Zuständen des statecharts assoziiert werden. Die intendierte Bedeutung ist, daß diese Formeln gelten sollen, solange der Zustand aktiv ist. Auf semantischer Ebene werden statecharts und TLA kombiniert, indem statecharts in äquivalente TLA-Formeln übersetzt werden. Mit diesen TLA-Formeln können dann die mit den Zuständen assoziierten TLA-Formeln verknüpft werden.

Es wird gezeigt, wie sich diese Beschreibungstechniken zusammen in unterschiedlichen Arten von Beschreibungen einsetzen lassen. Die Beschreibungen unterscheiden sich dabei sowohl in Bezug auf ihren Abstraktheitsgrad — Beschreibung einer ungesteuerten Anlage gegenüber dem Entwurf einer Steuerung, als auch dem Modus ihrer Bedeutung — mögliches Verhalten einer Anlage gegenüber ihrem gewünschten Verhalten.

Ein wesentliches Problem bei einer derartigen Kombination von Beschreibungstechniken ist es sicherzustellen, daß die entsprechenden Beschreibungen konsistent, also erfüllbar im Sinne der Logik, sind. In dieser Arbeit stellen wir Ansätze vor, mit denen nachgewiesen werden kann, daß eine Beschreibung konsistent ist.

# Summary

Different description techniques have to be used to describe properties of a system. This is the case if the properties belong to different aspects of a system. But even to describe properties of a single aspect it is often not sufficient to use a single one description technique.

We investigate in this dissertation how description techniques which are suitable for describing the aspect "behaviour" of reactive systems can be combined. This investigation is based upon a combination of the description techniques statecharts and TLA.

The description techniques statecharts and TLA differ w.r.t. the description style as well as w.r.t. the point of time when changes of entities become effective. The description technique statecharts is a representative of the class of model-oriented description techniques. By using such description techniques properties of a system are expressed by writing abstract models of a system. TLA is a representative of the class of property-oriented description techniques. Using them properties are often expressed by logical formulae. W.r.t. the point of time when changes become effective statecharts and TLA differ because in statecharts changes become effective after their stimuli. In TLA it is possible to describe that a change becomes effective together with its stimulus. Due to these differences it is not obvious how statecharts and TLA can be combined in a useful in way.

In this dissertation we combined statecharts and TLA on a syntactical level by associating TLA-formulae with states of a statechart. The intended meaning is that the TLA-formulae are valid as long as the corresponding state is active. On the semantical level statecharts and TLA are combined by translating statecharts into equivalent TLA-formulae. These TLA-formulae can be combined with the TLA-formulae that are associated with statecharts-states.

We show how these description techniques can be used together in different kinds of descriptions. The descriptions differ w.r.t. their level of abstraction — descriptions of an uncontrolled plant versus a design of a controller — as well as w.r.t. the mode of its meaning — possible behaviour of a plant versus its intended behaviour.

It is an essential problem of such a combination of description techniques to ensure that such descriptions are consistent. That means, that such descriptions are satisfiable. We present approaches to show that a description is consistent.

# Danksagung

Ich danke Professor Madlener dafür, daß er die Erstellung dieser Arbeit ermöglicht hat. Seine Anregungen und Fragen haben sehr zum Gelingen der Arbeit beigetragen. Professor Zimmermann danke ich für die Übernahme der Zweitbegutachtung. Professor Gotzhein danke ich für die Leitung der Promotionskommission.

Den Mitgliedern der Arbeitsgruppen von Professor Avenhaus und Professor Madlener und des SFB 501 danke ich für eine angenehme Arbeitsatmosphäre und viele wertvolle Diskussionen. Besonders Robert Eschbach, Martin Kronenburg, Birgit Reinert und Martin Schütze haben hierdurch zum Entstehen dieser Arbeit beigetragen.

Denen, die das Entstehen der Arbeit durch ihre Gebete begleitet haben, danke ich besonders.

Ich danke meinen Eltern, daß sie mir ermöglicht haben zu studieren. Dadurch haben sie die Grundlage für diese Arbeit gelegt. Ihnen und meiner Frau gilt mein herzlicher Dank. Sie haben mich über viele Jahre in Liebe begleitet.

Für Andrea
und Annegret

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

When developing or maintaining a system a large variety of descriptions of the system has to be made. The descriptions can be quite different w.r.t. the aspect of the system considered in the descriptions as well as w.r.t. the level of abstraction on which properties of the system are described. Therefore, there cannot be a single description technique which is suitable for such different descriptions. This is even true when considering only one aspect of a system on different levels of abstraction as well as when considering different aspects on the same level of abstraction. Hence, different description techniques have to be used together.

In this dissertation we investigate a combination of two description techniques to describe the behaviour of a special class of systems, so-called reactive systems. We combine the description techniques statecharts and TLA. We want to achieve two goals by this combination. At first, we want to provide adequate means to describe a large variety of properties of the behaviour of systems. At second, we want to use the combined description technique on different levels of abstraction and we want to allow a smooth transition between descriptions on different levels.

Up to now we have spoken quite generally of systems, descriptions, and description techniques. Before we go into more detail on the topic of combining descriptions and description techniques and the contribution of this dissertation to it in Chapter 2, we want to give our understanding of these terms. Thereby we concentrate on systems, their behaviour, and a set of descriptions we are interested in.

## 1.1 Systems and Their Behaviour

In a very general sense a *system* can be considered as a set of interacting *components*, cf. [Schwarzenbach and Gill, 1978]. Components can be physical components as well as functional units. Usually, a system is embedded in an *environment*, which itself can be considered as a set of interacting components. Which component is considered as a part of of the system and which one as a part of the environment,

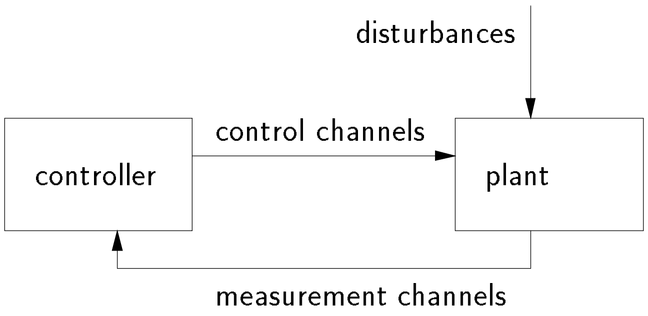



Figure 1.1: General structure of a control system, [Ostroff, 1989, page 4]

i.e. to determine the boundary of the system, is a question of modeling and depends on the aspects of the system which are considered.

A component of a system can be considered as a system itself and it can consist of subcomponents. Other components interacting with such a component become parts of the environment of this component. Hence there is a hierarchy of components, this can be viewed as the *structure* of a system.

In general, there is an exchange of information between a system and its environment. A system which exchanges information with its environment is called an *open system*. If it does not exchange information with its environment, it is called a *closed system*. Because closed systems are not able to influence their environment, nor can be influenced by it, they seem to be of low interest. But an open system together with its environment can be considered a system again, and this can be a closed one to exclude further influences. Hence, it is useful to consider open as well as closed systems.

We are interested in systems which maintain an ongoing interaction with their environment, typically by reacting to changes of or to stimuli from the environment. These systems are called *reactive systems*. Typical examples of such systems are monitoring- and control-systems. Their general structure is depicted in Figure 1.1, [Ostroff, 1989, page 4]. In this context, the system is the component controller and the environment of the system consists usually of the component plant. The controller senses the state of the plant via measurement channels and influences it via control channels. The plant itself is subject to further disturbances by its environment. But these disturbances are not visible directly to the controller, only their influence on the plant becomes visible to the controller. In Figure 1.1 only the highest level of the components controller and plant are shown, these can consist themselves of components as described before.

To describe the behaviour of a reactive system we make the basic assumption, that a set of *variables* is associated with each component. These variables are called *distinguished qualities* in [Chestnut, 1970]. The values of these variables can change over time. A valuation of variables with specific values at a point of time is called a *state*. For a sequence of points of time, a corresponding sequence of states is called a *run*. The set of runs of a component is called its *behaviour*. These terms will be defined more formally later on.

A variable can be associated with more than one component, in this case it is *visible* to or *shared* with the other components. If it is associated with only one component it is an *unshared* or *local* variable.

## 1.2 Descriptions of Behaviour

There can be descriptions of the behaviour of a system on very different levels of abstraction and with different intentions. In the following we present those descriptions we are interested in. These descriptions are characterised according to [Zave and Jackson, 1997]. They are presented in the context of control systems as described before.

The descriptions we are interested in are the more abstract ones. They can be distinguished w.r.t. two aspects: At first the descriptions can either refer to variables of the controller, of the plant, or of both of them. At second, it is possible to describe either intended or possible behaviour of a component. In [Zave and Jackson, 1997] descriptions of possible behaviour are said to be in *indicative mood*, descriptions of intended behaviour are in *optative mood*.

To describe the behaviour of a control system it is necessary to use terminology which is meaningful for its environment, that is the terminology has to be meaningful to an external observer. For example, variables which are visible only to a controller cannot be used. Knowledge about the plant, called *domain knowledge* in [Zave and Jackson, 1997], is captured in a description which we call a *reference world.* More specifically, we call it the *reference world (observer)* or RW-O for short. In such a description the variables visible to the plant and to an external observer are captured and their change over time is described. It is assumed in such a description, that the plant is not influenced by a controller. Such a description is in indicative mood.

In RW-O it can be refered to the variables of the plant visible to an external observer. These variables do not need to be shared with the controller. Dependencies between the variables refered to in RW-O and those shared between the controller and the plant are described in a second description. These dependencies express how control commands effect the state of the plant and vice versa how the state of the plant can be sensed by the controller. Such a description is in indicative mood. Following [Zave and Jackson, 1997] it is called *reference world (machine)*, or RW-M for short.

Requirements on the behaviour of a control system describe the intended behaviour of the plant, when it is actually influenced by an appropriate controller. The descriptions of requirements are called *Requirements* or REQ themselves. These descriptions are in optative mood. As in RW-O, variables visible only to the controller cannot be refered to in REQ. This restriction is necessary to avoid premature design decisions.

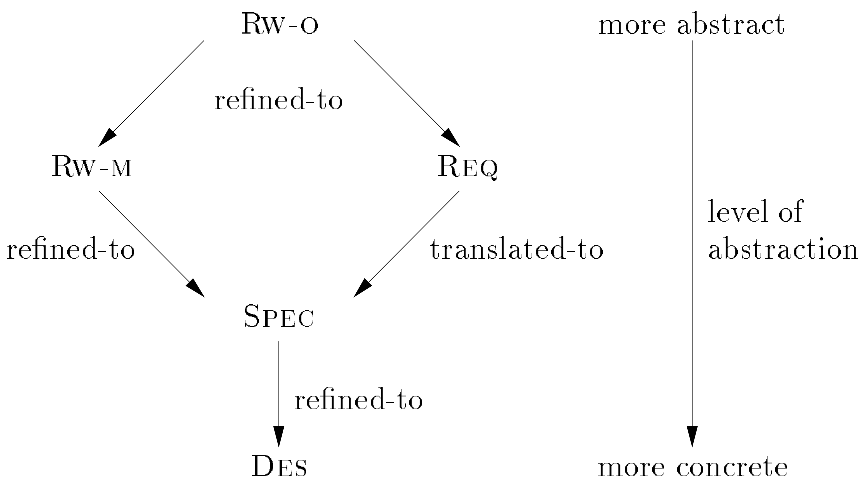



Figure 1.2: Relations between descriptions

Req should be a refinement of Rw-o.

From the requirements, the intended behaviour of a controller has to be determined. This behaviour is described in terms of the variables shared between controller and plant. Such a description is called a *specification* or Spec for short. They are in optative mood. Together with the domain knowledge expressed by Rw-m, Spec should be a refinement of Req.

A description in indicative mood of the controller is called here a *design* or Des. It describes how the controller actually behaves, or will behave if implemented correctly. The only variables of the plant it can refer to are those which are shared between controller and plant.

Let us note, that the definition of a design as a description in indicative mood is not as clear as described before, because a design can consist of a description how a controller is decomposed into components and specifications of each of these components. These specifications are descriptions in optative mood. The description of the structure of the controller and as a consequence the description how the behaviours of the components are composed to that of the controller is in indicative mood. Therefore we consider a design as being in indicative mood.

Some relations between these descriptions are depicted in Figure 1.2. The relation *refined-to* means that each run of a behaviour of a more concrete description is also a run of the more abstract one. In the case of the relation between Rw-o and Rw-m only variables which are used in Rw-o are taken into account when comparing the behaviours, other variables used only in Rw-m are hidden.

In Req, the behaviour of the uncontrolled plant as seen by an observer is restricted. In a similar way, Spec restricts the behaviour of the controller. The relation between Req and Spec is more complicated. The individual requirements have to be translated from the terminology of the plant to that of the controller. As an example, a safety property stated as an invariant has to be translated into a control strategy and the corresponding commands which are sufficient to maintain this invariant.

The behaviour of a controller, as described by DES, has to be a refinement of SPEC.

There is a further relation not shown in Figure 1.2: The composition of DES and RW-M is itself a refinement of REQ. Only the variables refered to in both RW-M and REQ are taken into account in this refinement. When both relations on DES and the other descriptions are actually valid, one can say, that REQ has been translated correctly to SPEC.

The relations between the descriptions depicted in Figure 1.2 suggest to proceed top-down when developing them. It should be clear that there have to be further descriptions from which these can be derived. These can be for example the needs of a customer given in an informal way or a layout or block diagram of the plant. A second point, which should also be clear, is that proceeding top-down does not imply that description on higher levels have to be fixed when starting to develop lower-level descriptions. We consider it essential to be able to change the higher-level descriptions again. An example of a situation in which this is necessary is the following one: When translating requirements to specifications it can become clear that a requirement cannot be realized to its full extend or that several requirements cannot be realized together. In this situation it is possible that the requirements are changed in accordance with the customer.

To support this process we propose to express REQ by taking RW-O as given and by adding the requirements as further properties incrementally to it. In a similar way it can be described in RW-M how the controller is connected to the plant by adding appropriate extensions to RW-O. In both descriptions non-deterministic behaviour expressed by RW-O and the set of input sequences can be restricted.

## 1.3 Description Techniques

There is a large variety of *description techniques* to describe the behaviour of systems. These range from informal over partially formalised or semi-formal to formal description techniques. Examples of such description techniques are natural language as an informal one, most of the description techniques accompanying object-oriented analysis and design methods as semi-formal ones, and the various kinds of temporal logics as formal ones.

A description technique is considered a formal one, if its syntax as well as its semantics are defined in a precise way. Formal description techniques can be defined formally themselves as *institutions*, cf. [Goguen and Burstall, 1992]. Advantages of formal description techniques are that they allow to make precise statements about a system and that they allow to analyse formal descriptions with mathematical rigour.

## 1.4 Description Styles

There are at least two fundamental approaches or *description styles* to describe the behaviour of a system. On the one hand a model of the system can be given, and the set of possible runs of this model is considered as the behaviour. On the other hand a set of behaviours can be described by expressing common properties of it. In the literature the description styles are named differently: *model-* and *property-oriented* in [Astesiano and Reggio, 1997], *construction-oriented* and *axiomatic* in [Astesiano and Reggio, 1997], and *system specification style* and *requirements specification style* in [Pnueli, 1992].

In [Pnueli, 1992] some relative advantages of using these two description styles are given. He examines the following aspects:[1]

**Incremental modification:** How easy is it to change a description, once it is realized that a property of the behaviour is missing or needs to be modified?

**Consistency:** Does a description describe a non-empty set of runs?

**Completeness:** Is it possible to detect that all important properties of the behaviour have been described and none is missing?

**Validation:** How easy is it to ensure that a description models actually the behaviour of the system as intended by a designer?

**Scalability:** Does the approach allow to deal with large systems?

Pnueli draws the conclusions shown in Table 1.1, [Pnueli, 1992]. As one can see, the approaches have quite different advantages and disadvantages. Besides the aspects considered by Pnueli there are further ones.[2] To our opinion an important advantage

[1] In [Pnueli, 1992] the terms specification and requirements are used in a different way as defined in Section 1.2. Therefore we use the more neutral term description here.

[2] A large list of criteria to investigate different aspects of formal methods has been collected in [Deiß *et al.*, 1997]. By some of these criteria details of specific techniques or methods are investigated which are not relevant to distinguish these fundamental approaches, but other ones can be applied here, too.

| | property-oriented | model-oriented |
|---|---|---|
| incremental modification | easy | difficult |
| consistency | viable | ensured |
| completeness | difficult | ensured |
| validation | good chance | difficult |
| scalability | problematic | promising |

Table 1.1: Relative advantages of approaches to describe behaviour, [Pnueli, 1992]

of property-oriented descriptions is that the properties can be stated in an explicit and often more succinct way. On the contrary, in model-oriented descriptions, properties must often be encoded in the models. As an example consider that a state invariant has to be expressed, in model-oriented descriptions appropriate reactions of the system have to be described such that the invariant is not violated, but the invariant itself is not stated.

An advantage of model-oriented description techniques, especially to describe behaviours, is that properties capturing specific sequences of states can be expressed in a more comprehensible way. Often model-oriented specification techniques are executable, therefore it is often more easy to determine the reaction of a system in a given scenario. Such a property is for example the description of operational modes of a system and possible transitions between them.[3]

There is a further distinction between the approaches to describe behaviours which becomes relevant when comparing different descriptions of the behaviour of a system. Throughout the development of a system descriptions of different levels of abstractness and with different intentions have to be related. In [Pnueli, 1992] it is distinguished between *single* and *dual language* approaches, according to whether the same description technique is used for both descriptions or whether different ones are used. There are single language approaches for model-oriented as well as for property-oriented description approaches. Dual language approaches typically use a property-oriented description technique for the abstract description, and a model-oriented one for the concrete description. As an example consider the use of temporal logic and fair transition systems in [Manna and Pnueli, 1992].

[3]Whether it is better to describe this graphically or in a tabular notation as proposed e.g. in [Courtois and Parnas, 1993] depends to some extend on factors as the number of different modes and the number of transitions between them.

# Chapter 2

# Combination of Description Techniques

As has already been stated, one goal of composing descriptions written in different description techniques is to be able to use a well-suited technique for each property to be described. Well-suited means that it is possible to produce succinct descriptions and that powerful and efficient techniques to analyse the descriptions are available. A second goal, stated in [Zave and Jackson, 1993], is that by allowing to compose descriptions in different description techniques it becomes possible to simplify the description techniques themselves. Zave and Jackson consider statecharts [Harel, 1987] as an example. They suggest simplifications leading to deterministic finite-state[1] automata with hierarchical states. Other features of statecharts such as decomposition of states into parallel substates or operators on data should be handled by other formalisms.

To combine descriptions on the same level of abstraction and which are written in different description techniques it is proposed in [Wing, 1990] to use a common semantic domain of the description techniques. The semantics of the description techniques has to be given in or translated to the common semantic domain. An example of such a common domain is first-order logic. This domain is suggested in [Zave and Jackson, 1993]. In this semantic domain, composing several descriptions is easy. The meaning of the composition of the descriptions is the conjunction of the meaning of the composed descriptions, i.e. there is '*conjunction as composition*' [Zave and Jackson, 1993]. An attempt to give a common semantic domain of description techniques to describe the behaviour of systems can be found in [Eschbach *et al.*, 1998].

By using first-order or some other kind of logic it becomes clear that consistency of the composition of several descriptions is not guaranteed, even if the single descriptions are consistent. In [Zave and Jackson, 1993] it is considered as infeasible to do consistency checking on the level of the translated semantics. In such a translation

[1]States of an automaton should not be confused with states describing valuations of variables.

the formulae tend to become large and incomprehensible, furthermore knowledge allowing specialised and efficient consistency checking is lost. Zave and Jackson expect that most consistency checking is done on the level of the partial descriptions. From the results of these consistency checks the consistency of the composed description can be established by following specific (de-) composition styles.

In addition to composing descriptions it is also possible to combine the description techniques themselves. Combining description techniques is to some extend contradictory to one of the goals one wants to achieve by combining descriptions, namely to simplify description techniques. Nevertheless this can be useful for two reasons. At first, when describing the properties belonging to one aspect of a system, it can be the case that none of the available description techniques is fully adequate to describe all of these properties. By combining description techniques it becomes possible to use a description technique when it is adequate and to avoid its weak points. At second, when properties belonging to different aspects are tightly connected, it is useful to describe them together. This implies that the corresponding description techniques have to be used together.

When relating descriptions on different levels of abstraction, there is a further reason to combine description techniques. A combination of two description techniques, where one is more oriented towards expressing problems more abstractly and the other one is oriented towards describing more concrete solutions, can be used for example to refine a specification successively in small steps to a design or a implementation. In the intermediate steps the descriptions will consist to varying degrees of parts written in one of the description techniques.

Description techniques can be combined syntactically in three different ways:

- The description techniques are not combined at all, each description is written using one description technique.
- The description techniques are merged, it is possible to use the syntactical constructs from several description techniques in one description.
- A completely new description technique is defined. Descriptions are again written using one description technique, namely the new one.

To merge two description techniques means that it is still possible to use each of the constituting description techniques without the other one in a description or a large part of it. The combined description technique can be seen as an extension of its constituents. This should be true for its semantics, too. A description written purely in one of the constituents should have the same meaning as when using the original description technique. This facilitates the usage of the combined description technique. Nevertheless it can be necessary to adapt syntax and semantics of the constituting description techniques to allow their combination. In this case the changes to syntax and semantics should be kept as small as possible for the same reason.

To combine two description techniques semantically it is possible on the one hand to define their semantics in the common semantic domain separately for each of them. The common semantic domain can be that of one of the description techniques or a new one. It is also possible to achieve a common semantic domain by translating descriptions in one description technique to the other one and to define the meaning of the original descriptions to be that of the translated ones.

## 2.1 Characterisation of Our Combination

The descriptions which shall be written in a combination of description techniques have already been presented in Section 1. They can be summarised as descriptions of the behaviour of reactive systems on high levels of abstraction. The combination of description techniques presented in this dissertation can be characterised in short as follows. More details of the characterisation are given subsequently.

**Diversity of Descriptions:** Different description techniques to describe properties which belong to one aspect of a system are combined. The aspect we are interested in is the behaviour of systems. The composition of different kinds of descriptions is not considered here.

**Goals:** At first, we aim at providing description techniques to allow adequate descriptions. The relative strengths of the model- and property-oriented description styles shall be made available. At second, we want to use the combined description technique on different levels of abstraction and we want to allow a smooth transition between such descriptions.

**Description Techniques:** The concrete description techniques combined are statecharts [Harel, 1987] and TLA [Lamport, 1994b].

**Semantics:** The semantic domain of TLA is taken as the common one. The semantics of a statechart is given by translating it to TLA-formulae.

**Syntax:** The description techniques are merged by associating TLA-formulae with the states of a statechart.

**Diversity of Descriptions** We concentrate on the combination of description techniques for one aspect of systems, namely their behaviour. Composition of different kinds of descriptions is not considered here. Nevertheless, composition of descriptions can be relevant here for two reasons. Both reasons are related to the choice of one of the concrete description techniques we combine, namely statecharts. At first, statecharts are accompanied with activity-charts, a description technique suitable to describe the functional decomposition of a system. Activity-charts allow to compose activities to more complex ones. This can be seen as a composition of descriptions. At second, statecharts and activity-charts allow to describe hierarchies of states and activities, respectively. In general, the composition of descriptions can lead to inconsistency. In a similar way it has to be shown that the higher levels of such hierarchies are consistent with the lower ones.

**Goals** At first, by combining the description techniques we want to be able to express a large range of properties of behaviours in a succinct way. Especially, the relative advantages of the model- as well as of the property-oriented description style shall be usable in one description. Aiming at expressiveness we want to be able to state properties explicitly and we want to be able to represent sequences of states in a comprehensible way.

At second it should be possible to add further properties to an already existing description. This is necessary to support the process of developing descriptions as presented in Section 1.2.

To which degree other advantages of these description styles are still available in the combination will be investigated in this dissertation. We concentrate on the question whether descriptions are consistent.

**Description Techniques** The description techniques we combine in this dissertation are TLA (Temporal Logic of Actions) [Lamport, 1994b] as a property-oriented and statecharts [Harel, 1987] as a model-oriented technique. Together with statecharts we consider activity-charts.

TLA has been chosen because it emphasises the notion of an action, i.e. the relation between two consecutive states in a run. This corresponds in a suitable way to the notion of steps of statecharts. In addition, there are theoretically solid foundations of refinement in TLA [Abadi and Lamport, 1991] and there are extensions to model real-time [Abadi and Lamport, 1994].

Statecharts are based on finite-state automata and are extended by concepts of hierarchy and orthogonality. These allow to represent large automata compactly. In statecharts it is possible to describe operations on data and at least in a rudimentary form timing aspects. Statecharts have already been used successfully in industrial projects. Hence it can be seen as a mature and suitable description technique for behaviours of reactive systems.

The possibility to construct hierarchies of states is considered an advantage by us over other description techniques such as e.g. SDL [CCITT, 1993] and the tabular SCR-notation, cf. e.g. [Heninger, 1980, Courtois and Parnas, 1993]. In addition, the SCR-notation does not allow to describe hierarchically related activities or processes. In SDL the hierarchy of blocks could be used for this purpose. To our opinion there is a further disadvantage of SDL. Its basic model of communication, i.e. asynchronous message passing as opposed to the broadcasting of events and the use of shared variables in statecharts, seems to be less suited for the descriptions in which only terms of the environment are used.

Statecharts and TLA use the same model of time: it is discrete, linear, and has a starting point. Nevertheless, by combining statecharts and TLA we get two possibilities when the effects of an action become visible. In statecharts, at least in the semantics we use here, the effects of an action or of taking a step become visible at the end of this step, it is possible to react to this effect only in the next step. By using TLA-formulae changing a variable in one step can influence another variable in the same step. This corresponds to the *synchrony hypothesis*. Hence it becomes possible to describe in one description synchronous as well as delayed reactions.

Although reasons have been given for our choice of statecharts and TLA it should be clear, that this does not imply that other description techniques cannot be combined or that this will be more difficult. At the end of this work we will critically review our choice of description techniques and we will give hints how our results can be transfered to combinations of other description techniques.

**Semantics** As a common semantic domain we take that of TLA, i.e. a description has a set of runs as its meaning. Such a set of runs is closed under adding or removing finitely many stuttering steps. This implies that the semantics of statecharts has to be given in this domain. To do so, we determine a set of variables to describe the current status of a statechart and describe the steps of a statechart by TLA-actions. Together with a description of the initial status of a statechart and fairness of taking steps we obtain a TLA-formula for a statechart. The meaning of a statechart is defined to be that of this TLA-formula. This TLA-formula captures the meaning of a statechart as described in [Harel and Naamad, 1996]. By the translation of a statechart to a TLA-formula we get again a set of runs, which is closed under stuttering, as its meaning. Because statecharts taken alone do not have a semantics which is closed under stuttering, we will take care, that stuttering steps will not be introduced on the level of statecharts implicitly.

An advantage of using the semantic domain of TLA as a common one is that it becomes possible to apply theoretical results on TLA on descriptions written in the combined description technique. For example, the definition of refinement of TLA can be used.

**Syntax** Statecharts and TLA are combined syntactically by associating TLA-formulae with states and activities. The intended meaning is that if such a state or activity is currently active, its associated formulae have to be valid. Those runs of the pure statechart or activity-chart which do not satisfy these associated TLA-formulae are discarded from the behaviour of the statechart or activity-chart.

The TLA-formulae associated with a state or activity can be used to express safety properties by initial conditions and next-state relations. An important special case of the latter ones are invariants. Furthermore, liveness properties can be expressed by fairness and by leads-to formulae.

## 2.2 Structure of This Dissertation

This dissertation consists of three parts:

- Combining statecharts and activity-charts with TLA
- Applying the combination of statecharts and TLA to examples
- Examining their combination, especially whether descriptions are consistent

The first part starts with an introduction to TLA in Chapter 3. This presents the essential concepts of TLA as far as they are used in this work. Chapters 4 and 5 contain the translation of statecharts to TLA, where Chapter 5 is devoted solely to data-variables. When formalising statecharts we omit some parts of its syntax which are not relevant for the combination examined here. For example, we omit history and deep-history connectors. On the other hand we introduce a new kind of transitions which we call *static transitions*. These are a combination of the usual transitions and static reactions. They are used to introduce a special case of non-determinism in behaviours. Chapter 5 contains also an extension of statechart-actions to provide non-determinism.

Two approaches of combining statecharts and TLA are presented in Chapter 6. In the first one we use TLA-actions to restrict whether a transition is enabled. This approach leads to descriptions which are difficult to understand and is discarded therefore. In the second one TLA-formulae are associated with states. The semantics of associating formulae with states is defined such that the behaviour of the pure statechart is restricted.

The second part which consists essentially of Chapter 7 shows how descriptions written in the combination of statecharts and TLA look like. The descriptions of a simple lift system from RW-O to several variants of the design of a controller are given. In the lift system synchronisation of actions occurs rarely. This is examined by giving a second example. In the second example we describe a part of the well-known case study "*production cell*" [Lewerentz and Lindner, 1995].

Chapter 8 resumes the first part by translating activity-charts to TLA, associating TLA-formulae with activities, and by structuring the descriptions of the lift-system with activities. Activities are important because in contrast to statecharts they allow to describe interfaces of components. In the third part we will see that this is valuable information when checking whether descriptions are consistent.

The main topic of the third part is to show that a description written in the combination of statecharts and TLA is consistent. This is considered in Chapter 9. By using TLA synchrony between changes of different variables is introduced, hence causality will play an important part here.

In Chapter 10 conclusions are drawn from this concrete combination of description techniques. We give hints how other description techniques than statecharts and

TLA can be combined.

In the remainder of this chapter work related to the combination of description techniques is presented. Work related to giving a formal semantics of statecharts and activity-charts can be found in the corresponding chapters.

## 2.3 Combinations of Programming Languages

The considerations on combining description techniques given before have been oriented towards more abstract descriptions. But there are also approaches to combine programming languages, some of them are presented in the following. Further ones, namely combinations of synchronous languages, can be found in Section 2.5. The common goal of these approaches is to be able to use the language which is most suited for the (sub-) problem at hand.

### 2.3.1 Functional Logic Programming

There have been lots of approaches to combine functional and logic programming languages. Both classes of languages are declarative languages. Besides increased expressive power, stated as a goal in [Dershowitz and Plaisted, 1985], several further aims have been given. In [Bellia and Levi, 1986] and [Hanus, 1994] some of these goals are collected. They can be grouped into three classes:

1. Logic programming languages allow the definition of relations, whereas some problems can be described more naturally by functions. It seems to be advantageous to use functional languages in the latter case. Examples of this kind of problems are those related to higher-order functions.

2. Functional languages have advantages w.r.t. implementation issues. Functional programs contain more control information than logic programs, this can be used to write more efficient programs. A further, initial goal, was to make the more powerful implementation techniques and development environments of functional languages available for logic programming, too. This goal does not seem to play a role nowadays anymore.

3. On the other hand, logic programming is more powerful because it is inherently a problem solving technique. Given a predicate $p(x_1, \dots, x_n)$ and values of some of the $x_i$, it is possible to compute values of the other $x_j$, such that $p(x_1, \dots, x_n)$ is satisfied. There is no distinction in logic programming between input and output variables of $p$. This makes it possible to *invert* computations.

The operational semantics of such a combination is often given by narrowing. For example, in [Prehofer, 1996, Prehofer, 1998] functional logic programming is considered as a special case of narrowing with left-linear, higher-order rewrite rules.

Restrictions on the rules are used to guarantee decidability of e.g. unification and to allow efficient narrowing techniques. In this combination the constructs of the functional language are extended by constructs supporting logic programming, namely extra variables in left-hand sides of rewrite rules are allowed and rewriting is replaced by narrowing as operationalisation.

The combination of functional and logic programming is achieved by a common generalisation of both approaches, one language is not embedded in the other one. This does not preclude that the syntax of a functional logic programming language is dominated by either functional or logic programming constructs.

### 2.3.2 Constraint Logic Programming

Another combination of paradigms is constraint logic programming, a combination of logic programming and constraint solving, see e.g. [Jaffar and Maher, 1994]. Such a combination can be described in short as incorporating constraints and techniques for solving them in a logic-based language. A first example of constraints are equations over the algebra of terms. In this case constraint programming corresponds to pure logic programming, the equations are implicit in the use of unification. Another example are linear arithmetic constraints over the real numbers. In [Brzoska, 1991] it is shown that the temporal logic programming language Templog [Abadi and Manna, 1989] is an instance of constraint logic programming over a suitable algebra $\mathcal{A}$.

The combination of the paradigms is motivated by the observation that some problems seem to be expressible naturally using such a combination. Examples of this kind of problems are combinatorial search problems. The logic programming part is used to enumerate possible solutions, while constraints are used to keep the search space manageable.

Pure logic programming as compared to constraint logic programming can be seen as specialisation of the latter one to a specific constraint domain. It seems to be the case that constraint solving and logic programming are not very different paradigms.

Constraint logic programming is related to the work presented here because we use a similar approach of combining two languages, one of them (statecharts) generating possible runs, the other one (TLA) excluding runs not satisfying temporal constraints expressed as TLA-formulae. This is for example the case when using invariants to decrease non-determinism of a statechart. In our case it is also possible to use TLA directly to describe possible runs by TLA-programs as usual.

### 2.3.3 Multiparadigm Programming

[Zave, 1989] presents an approach to use several programming languages belonging to different programming paradigms together to solve problems. To combine different paradigms it is necessary, that the incompleteness of one language is represented as non-determinism of the other. She identifies several different kinds of non-determinism which can be present in one language and resolved by another.

An example of a simple telephony system is given, in which such diverse languages as Prolog, Awk, CSP [Hoare, 1985], and PAISLey [Zave, 1991] are used. The last one is a process-oriented, applicative, and interpretable specification language. It is used to glue the programs written in the other languages together and it is used for performance modeling. The most interesting part w.r.t. the work presented here is the relation between the CSP and the PAISLey program. CSP is used in this example to describe the protocols to control the telephone features. The CSP and PAISLey programs are synchronised by shared events. The PAISLey program resolves scheduling non-determinism of the CSP-program. Vice versa the CSP-program resolves functional non-determinism in the PAISLey program by providing return values of function calls. To realize the sharing of events a CSP-interpreter was written for this case study. This CSP-interpreter was able to receive scheduling decisions from the outside. The combination of CSP and PAISLey is interesting because the languages and the programs overlap to a large extend. Hence there is potential for inconsistency. This is solved on the one hand by stating clearly which program exhibits which kind of non-determinism to be resolved by the other one. On the other hand it is stated that the message passing behaviours of processes in PAISLey and the corresponding CSP-processes is the same.

This approach shows also the necessity that programs, and more generally descriptions, provide non-determinism which is resolved then by other means. In a similar way we will usually have a non-deterministic description RW-O, which is restricted to the more deterministic descriptions RW-M and REQ.

## 2.4 Diagrams and Logic

There are already some approaches to combine logics and description techniques based on diagrams. In the first two of the approaches presented here the diagrams are used to illustrate or visualise descriptions in logics. The third one is similar to ours in that both the diagrams and the logic formulae contribute to the meaning descriptions. A fourth approach is described in Section 2.5.4.

### 2.4.1 Combination of TLA and Diagrams

In analogy with state-transition diagrams so called predicate-action diagrams are introduced in [Lamport, 1995]. Vertices in these diagrams are labeled with predicates and correspond to sets of states, in the sense of valuations of variables. Edges are labeled with TLA-actions and correspond to transitions between the states. The meaning of a diagram is defined by constructing a TLA-formula from the diagram and by using the meaning of this formulae as the one of the diagram. In predicate-action diagrams only safety-properties are expressed.

These diagrams are intended as explanations of descriptions and to illustrate proofs. They are not intended to describe the behaviour of a system completely. It is possible to use several complementary diagrams to illustrate one description. Consistency between such diagrams is guaranteed, if each of them is an explanation of the same description, this is captured formally by requiring that the description is a refinement of each of the formulae corresponding to the diagrams.

This approach differs from the one presented in this dissertation thereby that the diagrams are used "only" as illustrations of descriptions given completely as TLA-formulae. Also, the diagrams are of a very simple form, there is for example no hierarchical organisation of the vertices in these diagrams. Quite similar to our approach is that the meaning of a diagram is defined by constructing a corresponding TLA-formulae.

### 2.4.2 Statecharts to Visualise cTLA

In [Mester and Mattick, 1996] it is proposed to use statecharts to visualise different views on a description written in cTLA. In cTLA [Herrmann and Krumm, 1995, Mester and Krumm, 1995] distributed systems are described in a process-oriented and compositional way. A view on a system is build by defining a group of relevant variables and the corresponding cTLA-actions and initial conditions.

Starting point to visualise a description is the choice of a so-called major state component. This means to choose variables representing the control information of the system. The states of a statechart which visualises a view are labeled with state predicates. Usually these predicates express properties of the values of the major state component, but it is also possible to express properties of all variables. cTLA-actions are split into a part related to the major state component and a part related to a minor state component. By the pre- and post-conditions on the part corresponding to the major state component source and target states of a corresponding transition of the statechart are fixed. The part corresponding to the minor state component is considered as a part of the triggering condition or action associated with this transition. A refinement of the major state component induces a corresponding hierarchy of further states of the statechart.

### 2.4.3 Diagrams and Linear Time Logic

In [Blom and Jonsson, 1996] a specification of a memory which can be accessed via remote procedure calls is given in a logic which is similar to TLA. The main difference to TLA are the use of flexible predicate variables to represent synchronisation events. State transition diagrams are used to specify those parts of a system which involves control. Vertices of such a diagram represent different values of a control variable.

The label of a node is the value represented by this node. The arcs are labeled with predicate variables — representing synchronisation — and by transition formulae. The semantics of a state-transition diagram is given by a conjunction of formulae corresponding to the arcs.

In [Blom and Jonsson, 1996] the graphical syntax is used together with the textual one and the state-transition diagrams are actually representing additional information. As such this approach is very similar to the one presented here, although only predicate variables and actions as inscriptions of arcs are the only temporal operators which can be used within the diagrams. The synchronisation of actions is similar to the one we use in the example in Section 7.8.

## 2.5 Synchronous Languages

When associating temporal constraints with states, the change of one variable can cause the change of other ones in the same step. In this case the so-called *synchrony hypothesis*, stating that the output as reaction to an input occurs at the same instant of time, is followed. We will give some general remarks on *synchronous languages* here first, some relations of our work to recent developments in the area of synchronous languages are described subsequently.

Four representatives of these kind of languages are *Argos*, *Esterel*, *Lustre*, and *Signal*. As a common reference see [Halbwachs, 1993]. These languages follow different paradigms. Argos is a graphical language similar to statecharts. Whether transitions can be taken depends on the presence or absence of signals. *Argos* avoids some of the problem of statecharts, for example those arising by transitions between orthogonal states. *Esterel* is an imperative, textual language. It contains operators to express parallelism. Following the synchrony hypothesis, most operators do not consume time. *Lustre* and *Signal* are declarative languages. Lustre is a data flow language, whereas in Signal relations between the values of signals and constraints between their occurrence can be stated.

This body of work is similar, resp. differs, from the one presented here in the following points.

- The synchronous languages are intended mainly as programming languages, that is to implement systems. With the combination of statecharts and TLA we aim at more abstract descriptions. Our combination of statecharts and TLA shall be usable for descriptions on various levels of abstraction.
- As a consequence, issues of compiling programs into executable code are of less importance in this dissertation. On the other hand, it becomes more important to check whether one description is a refinement of another.
- Statecharts without temporal constraints cannot be considered a synchronous language. But by the temporal constraints it becomes possible that reactions

take place in the same step as their causes. As a consequence it becomes necessary, as for synchronous languages, to check causality between variables, events, and states, in order to guarantee that steps can be taken at all.

### 2.5.1 Synchronous Transition Systems

In [Benveniste *et al.*, 1997] so-called *synchronous transition systems* (STS) are presented.[2] An STS $\Phi$ is defined as a system $\Phi = \langle V, Q, \rho \rangle$, where $V$ is a finite set of variables, $Q(V)$ is an assertion characterising initial states and $\rho(V^-, V)$ is the transition relation between previous and current states. STS are closely related to TLA, but neither are fairness assumptions stated nor does stuttering occur explicitly in the definition of the transition relation.

STSs are used in [Benveniste *et al.*, 1997] as a basic synchronous formalism to illustrate problems of separate compilation of synchronous programs. As such, STSs are not directly related to the combination of languages. But this separate compilation depends heavily on causality analysis to derive so-called scheduling specifications. When considering consistency of statecharts with temporal constraints causal dependencies will play an important role. We will try to derive the causal dependencies from the way the statechart is written. In contrast, in [Benveniste *et al.*, 1997] the scheduling specifications are used to denote explicitly these dependencies. It is required there that if there is a cyclic dependency the corresponding assignments cannot be executed except if the dependency is a cycle not involving other variables. Furthermore, it is required that if a variable can be assigned the values of two different expressions, this does not happen in the same step.

### 2.5.2 Separate Compilation of Synchronous Languages

In [Benveniste *et al.*, 1994] and [Benveniste *et al.*, 1997] the causality analysis of synchronous languages is considered. In [Benveniste *et al.*, 1994] this is based on a simple synchronous data flow language called $\mu GC$, in [Benveniste *et al.*, 1997] it is based on synchronous transition systems, see also Section 2.5.1.

An aim of this work on causality analysis is to allow for the separate compilation of programs in synchronous languages. It is intended to combine programs written in different languages. To achieve this a common format to which different languages can be compiled is used. In this common format enough information is contained to derive legal schedules when combining separately compiled programs. Similar problems can occur in the work presented here when combining statecharts and activities and relating their behaviours by temporal constraints. The composition of activities can introduce cycles in the causality relation.

The same remark as in Section 2.5.1 can be made here w.r.t. the availability of

[2] In [Benveniste *et al.*, 1997] STS are attributed to Amir Pnueli.

syntactical means to express causality and w.r.t. the requirements on the dependency relation.

### 2.5.3 Combination of Synchronous Languages

To enable the combination of descriptions written in different synchronous languages a semantic framework and *synchronous automata* as a basic formalism are provided in [Poigné and Holenderski, 1998].

The semantic framework is based on *traces* of signals. This means that a linear model of time is used which is modeled by the set $\mathbb{N}$ of natural numbers and the usual ordering on it. A trace of a signal $s$ is specified by a subset $!\delta_s \subseteq \mathbb{N}$ and a valuation $\delta_s$: $!\delta_s \to \mathcal{W}$, where $\mathcal{W}$ denotes a set of values. $!\delta_s$ characterises the points of time at which the signal is *present*, $\delta_s$ describes a value the signal 'carries' if the signal is present.

Synchronous automata are defined as a set of actions *if* $\phi$ *then* $x = \epsilon$, where $x$ is a name, $\epsilon$ a data expression, and $\phi$ a pure (signal) expression. The intended meaning is that $x$ is updated to $\epsilon$, when $\phi$ is present. It is shown how various constructs of different synchronous languages can be translated to these actions.

Although it is mentioned that equations and imperative statements can be mixed, [Poigné and Holenderski, 1998, Section 6.1], this is not described in more detail or used in actual descriptions.

### 2.5.4 Combination of Argos and Lustre

Similar combinations of the data flow language Lustre with the statecharts like language Argos are presented in [Jourdan *et al.*, 1994, Maraninchi and Rémond, 1998a] and [Maraninchi and Rémond, 1998b]. More precisely, only a subset of Lustre, called mini-Lustre, and Argos-like operations on states such as parallel and hierarchical composition are considered. The approach to combine the languages is to associate equations (of Lustre) with states (of Argos), the semantics is provided by a compilation of the mixed language into Lustre. Hence, the general approach is quite similar to the work presented here.

In this work some hints are given, which aspects of behaviours can be described adequately by which language and shall be available in the combined one. In [Jourdan *et al.*, 1994] it is stated, that the imperative style of Argos is adequate for expressing interrupts and exceptions. This is supported by the hierarchical structure of the automata. On the other hand it is stated, that the regular behaviour of a system is described more suitable by the declarative style of the data flow language Lustre. In [Maraninchi and Rémond, 1998b] the need to talk about different *running modes* of a system, described by using a data flow synchronous language, is given as motivation. It is stated that it is preferable to use automata in addition to the data flow language instead of encoding the modes into the data flows. Using automata the mode structure remains readable and usable to improve the generation of code or the decomposition of proofs.

These approaches differ from our combination of statecharts and TLA in the following three points: At first, even within the automata, inputs and outputs are defined. This makes causality analysis much easier. At second, the automata are more restricted: They are required to be deterministic, i.e. each state has at most one distinct successor for each possible input, and to be reactive, i.e. each state has at least one successor for each input. Also, parallel composition is defined only if the outputs of the composed parts are disjoint. At third, whether a state becomes active, does not depend on the data flow equations associated with it. Whether a transition between states is taken, depends only on conditions on previous values of variables. This is in contrast to our combination of statecharts and TLA.

### 2.5.5 Different Forms of Broadcasting

In [Nazareth *et al.*, 1996] and [Philipps and Scholz, 1997] two simplified and slightly different variants of statecharts are introduced. These are called *mini-statecharts* and *$\mu$-charts*, respectively. It is of interest here that three different mechanisms to feedback signals are investigated. Signals in these variants correspond to events of statecharts and feedback corresponds to broadcasting. The three different kinds considered are:

**Instantaneous feedback:** The signals generated in a step are made available to trigger other transitions in the same step. Hence, instantaneous feedback satisfies the synchrony hypothesis. It is possible that there is not a unique fixpoint of the set of signals in one step or that there is not a fixpoint at all.

**Delayed feedback:** The signals generated in one step are made available as inputs of the next step only. Therefore, delayed feedback does not satisfy the synchrony hypothesis. On the other hand, there do not occur any problems with fixpoints of the set of signals existing in a status.

**Macro-Micro step feedback:** It is distinguished here between signals generated by the environment and internal signals of a statechart. After being triggered by external signals the statechart takes steps, triggered by internal signals, until "useful" signals cannot be produced any longer. The problems are to define under which conditions such a sequence of (internal) steps shall be considered as terminated and that there can be infinite sequences of such steps. The signals generated are transmitted after termination of the sequence. This kind of feedback does not satisfy the synchrony hypothesis.

Some advantages are reported in [Nazareth *et al.*, 1996, Philipps and Scholz, 1997] for the different forms of feedback. Instantaneous feedback is stated to be most suitable when refinement of descriptions is considered. A chart can be replaced by more concrete subcharts without introducing additional delays. This argument is stated originally in [Huizing and de Roever, 1991]. Note that this kind of refinement means to refine a chart into a chart with more orthogonal components, but not to refine a transition into a sequence of finer transitions with intermediate states. The latter one corresponds to some extend to macro-micro step feedback. One step as described by a reaction to external stimuli corresponds to a sequence of internal steps. An advantage of Macro-Micro step feedback is, that it distinguishes between internal and external signals, which is, according to [Nazareth *et al.*, 1996], required by industry. On the other hand using a cascade of these feedback operators leads to a hierarchical notion of time, which can be complicated to understand. At last, delayed feedback has the most simple semantics to understand.

In [Nazareth *et al.*, 1996] it is stated that all kinds of feedback can be used in one description, but that this leads to a semantics which is not clear intuitively. This problem will be considered in the work presented here, too. For statecharts we choose a semantics corresponding to delayed feedback, but by temporal constraints instantaneous propagations of events and changes of variables becomes possible. We will try to show how this can be mixed without producing incomprehensible descriptions.

## 2.6 Further Related Work

In this section we present approaches which compose different descriptions written in different description techniques (Sections 2.6.1 and 2.6.2), which use different description techniques in one description when refining it successively (Section 2.6.3), and which add real-time constraints to the behaviour of reactive systems (Section 2.6.4).

### 2.6.1 Statecharts and Temporal Logic

A combination of statecharts and a temporal logic called FNLOG is proposed in [Sowmya and Ramesh, 1998]. It is intended to use the description techniques in a complementary way to define the *behaviour* and the *function* of a reactive system. The corresponding specifications are described as:

- The *behaviour specification* of a system specifies *how* a system behaviour is generated. The information processing required to generate the behaviour is specified, [Sowmya and Ramesh, 1998, page 216]. Statecharts are used as description technique for behaviour specifications.

- The *function specification* of the system specifies *what* the system behaviours are. The causal and temporal relationships between events as well as the transformations between input and output are specified, [Sowmya and Ramesh, 1998, page 216]. The temporal logic FNLOG is used for this kind of descriptions.

The keywords *what* and *how* usually refer to the *specification* and the *implementation* of the behaviour of a system, respectively. Furthermore, the authors propose to start with a behaviour specification in statecharts, translate it to an *equivalent* function specification in FNLOG and verify properties of this specification. The properties are stated in FNLOG, too. This means to our opinion that the specifications are actually *not* complementary, the approach seems to be a common dual language approach, cf. Section 1.4.

### 2.6.2 Combination of Statecharts and Z

In [Weber, 1996] a combination of statecharts and Z [Spivey, 1992] is described. It is intended in this combination to describe the life cycle of objects by (a variant of) statecharts. This is called the *reactive model* of a system. The *functional model*, comprising for example data definitions and data invariants, is described using Z. These models are complemented by a third model, describing an architectural or structural view using an object-oriented notation.

The reactive and functional models of a system describe complementary aspects, nevertheless these are related here. In the functional model the local state of each component and the input-output relation of its operations are contained. For each statecharts-state in the reactive model there is a corresponding schema in the functional model defining a Z-state, i.e. a valuation of variables. A set of conditions is given to establish consistency between the models. As an example, the Z-state corresponding to a basic-state has to be satisfiable. In a similar way, consistency conditions between statechart transitions and Z operations are given.

Due to these consistency conditions a great amount of redundancy between these two models exists. Statecharts can be seen here as illustrating the reactive behaviour which is contained in the functional model. This is similar to the approach of Lamport described in Section 2.4.1, but the relation between the statecharts and Z-schemata is stronger here. This allows to proceed differently when describing a system. The reactive model can be described at first and used as input to derive the more detailed functional model.

### 2.6.3 Mixed Terms

In [Rössig, 1994] two different kinds of combinations of description techniques are used: The behaviour of a system is described by a state part and a trace part and in intermediate stages of development a description can consist of a mixture of specification and programming language constructs.

The description of the behaviour of a system is split into two parts in [Rössig, 1994]: In the *trace part* the sequencing of the messages sent and received on externally visible communication channels is described. This sequencing is described by using regular languages over the channels. Thereby the kind or name of the messages are used as letters of the alphabets over which the languages are defined. In the *state part* the change of the internal state of a process when receiving a message with specific values is described. Similarly, the exact values sent as parameters of messages are described in the state part.

The central topic of [Rössig, 1994] is the development of systems by applying transformations to its descriptions. In [Rössig, 1994] there is a sequence of descriptions leading from a specification, written in a specification language, to an implementation, written in an `occam`-like programming language. In the intermediate steps a mixture of specification and programming language constructs can be used. This is made possible by giving specifications as well as programs a predicative semantics.

Both kinds of combinations are related to the combination we present in this dissertation. The use of the state and the trace part to describe behaviour can also be seen as a combination of a model- and a property-oriented description technique. In some contrast to [Rössig, 1994] we want to use statecharts as well as TLA-formulae already in requirements and specifications and not only in intermediate descriptions. Furthermore, we will concentrate more on defining the combination of statecharts and TLA, but less on developing transformation rules as it is done in [Rössig, 1994].

### 2.6.4 Real-Time Estelle

In [Fischer, 1996] the formal description technique Estelle [ISO, 1989] is extended by a real-time logic to express timing requirements.

In Estelle, transitions between states, so-called global situations, are considered as being instantaneous. The natural numbers are used as model of time together with an interleaving semantics which is standard for Estelle. Timed observation sequences are obtained from untimed sequences of states by adding a function $f\colon S \to \mathbb{N}$, which maps each state to its time of occurrence.

To describe states in formulae it is possible to refer to observable propositions of Estelle modules and their boolean combinations. Time can be refered to by the time function *now* and can be "stored" in time variables. The temporal operators *henceforth* and *eventually* are available and can be used in combination with *now* and the time variables to express properties which are bounded in time. There is also an explicit construct to describe what happens if a real-time requirement cannot be satisfied at run time. The logic is quite similar to RTTL [Ostroff, 1989]. Timing requirements can be expressed on the level of Estelle modules.

The approach to define the semantics of adding real-time requirements is quite similar as in the work presented here. An Estelle specification is considered as generating a set of timed state sequences, those satisfying in addition the real-time requirements are considered as the meaning of the complete description.

In contrast to our combination of statecharts and TLA the description techniques are used for different purposes. In Estelle the untimed (w.r.t. *real*-time) behaviour is described. The timing requirements are given solely in the temporal logic. Neither are timing properties expressed operationally, nor is untimed behaviour given in logic. Whether the timing requirements are consistent with the untimed behaviour is not considered at all.

## 2.7 Summary of Related Work

From the related work presented in the previous sections one can see that the combination of description techniques is investigated in many different areas of research and that there is a great diversity of approaches. A common aim of many of these approaches is to provide adequate means to describe properties of a system.

The approaches differ w.r.t. how close the description techniques are combined. There are only few approaches which merge the description techniques as we propose to do.

# Chapter 3

# Temporal Logic of Actions

In this chapter we present TLA, the Temporal Logic of Actions [Lamport, 1994b]. Section 3.1 provides the definitions of syntax and semantics, Section 3.2 contains the description of a simple traffic light system as an example of how the behaviour of a reactive system can be described in TLA. This example is taken up again in Chapter 4 as a statecharts description. In Section 3.3 some more advanced concepts, e.g. refinement, in the context of TLA are presented. These are relevant only in the Chapters 8 and 9. Therefore Section 3.3 can be skipped on first reading.

TLA is a temporal logic to describe the behaviour of a system. This means that infinite sequences of values of variables are described by a TLA-formula. The central notion of TLA is that of an *action* which describes the relation between the values of variables at two consecutive points of time.

To give a first impression of how the behaviour of a system is described in TLA let us assume that we want to describe that the values of the variables $x$ and $y$ are incremented repeatedly, starting with 0. This behaviour is described by the TLA-formula $\Phi$ depicted in Figure 3.1. $\Phi$ consists of three parts:

- A description of the initial values of the variables. This is the TLA-formula $init_\Phi$ stating that $x$ and $y$ equal 0.
- A description of the relation between two consecutive valuations of the variables. This is the TLA-formula $\Box[\mathcal{M}]_{\langle x,y\rangle}$ stating that each pair of consecutive

$$
\begin{array}{lcl}
init_\Phi & \equiv & (x = 0) \land (y = 0) \\
\mathcal{M}_1 & \equiv & (x' = x + 1) \land (y' = y) \\
\mathcal{M}_2 & \equiv & (y' = y + 1) \land (x' = x) \\
\mathcal{M} & \equiv & \mathcal{M}_1 \lor \mathcal{M}_2 \\
\Phi & \equiv & init_\Phi \land \Box[\mathcal{M}]_{\langle x,y\rangle} \land WF_{\langle x,y\rangle}(\mathcal{M}_1) \land WF_{\langle x,y\rangle}(\mathcal{M}_2)
\end{array}
$$

Figure 3.1: The formula $\Phi$, describing that $x$ and $y$ are incremented repeatedly.

valuations is related by the action $\mathcal{M}$ or that $x$ and $y$ have the same value in consecutive valuations. That $x$ and $y$ do not change is described by the subscript $\langle x, y\rangle$ of $\Box[\mathcal{M}]$. The action $\mathcal{M}$ itself describes that either $x$ is incremented by one and $y$ does not change — $\mathcal{M}_1$ — or $y$ is incremented and $x$ does not change — $\mathcal{M}_2$.

- Two TLA-formulae expressing that, if possible, the action $\mathcal{M}_1$ must be taken eventually and similarly for $\mathcal{M}_2$. This is described by the fairness-formulae $WF_{\langle x,y\rangle}(\mathcal{M}_1)$ and $WF_{\langle x,y\rangle}(\mathcal{M}_2)$.

## 3.1 Syntax and Semantics of TLA

In this section we give a formal definition of syntax and semantics of TLA. The definitions which can be found in literature vary w.r.t. the degree of formality as well as w.r.t. the content of the definitions. The main reference on TLA is [Lamport, 1994b]. It introduces TLA and explains its use, but the definitions presented there are not strictly formal. A formal definition of TLA can be found in [Abadi and Merz, 1996]. The following definitions are based on this reference.

One aspect which varies between the various definitions is whether boolean values can be assigned to variables. In [Lamport, 1994b] this is not the case. This implies that no actions of the form $[\mathcal{N}]_{\langle z_1,\ldots,z_n\rangle}$ can be written in which any of the $z_i$ is a boolean variable. In other definitions this distinction is not made, e.g. in [Abadi and Lamport, 1994] predicates are considered as boolean-valued functions. This allows to write actions as described above.

We want to be able to state that the value of a boolean variable or the truth-value of a predicate does not change between two states. Hence we will follow [Abadi and Lamport, 1994] and consider predicates as boolean-valued functions.

Syntax and semantics will be defined in two parts. At first we will define functions and predicates as in first-order logic. Roughly, these are interpreted over single states or pairs of states. At second we will define syntax and semantics of (temporal) TLA-formulae. These will be interpreted over runs, i.e. infinite sequences of states. At last, we present some notational conventions.

### 3.1.1 First-Order Logic Part of TLA

The definitions in this first part are similar to that of first-order logic, except that it is distinguished between *rigid* and *flexible* variables. Each flexible variable $x$ has a *primed* counterpart $x'$. Unprimed as well as primed variables can be used in expressions denoting functions. An expression containing free primed variables is called a *transition function* and will be interpreted over a pair of states and a valuation of the rigid variables. A transition function containing no free primed variables is called a *state function* and will be interpreted over a single state and a valuation of the rigid variables.

The sets of variables and function symbols are usually not given explicitly in TLA. In the following we present some assumptions on these sets.

We assume as given an infinite set $\mathcal{V}$ of variables. This is partitioned into an infinite set $\mathcal{V}_R$ of *rigid* and a set $\mathcal{V}_F$ of *flexible* variables. The set $\mathcal{V}_{F'} = \{x' \mid x \in \mathcal{V}_F\}$ is the set of *primed flexible* variables. Because there are no primed rigid variables these will be called primed variables for short. We assume $\mathcal{V}_{F'} \cap \mathcal{V}_R = \emptyset$. The set of all variables is $\mathcal{V}_E = \mathcal{V}_R \cup \mathcal{V}_F \cup \mathcal{V}_{F'}$.

To distinguish boolean variables from the other ones we assume that each variable is assigned one of the sorts `any` and `bool`. We assume that there are infinitely many variables of each of the sorts. The sort of a variable $x$ is denoted by $sort(x)$. We assume that $sort(x) = sort(x')$ for all flexible variables $x$. Lamport defined TLA as an untyped logic, therefore we do not introduce any further sorts. Note, to enhance readability of definitions we will later on use declarations in an informal way to describe domain and range of functions and predicates, see the remarks on notational conventions in Section 3.1.3.

Furthermore we assume a sequence $\mathcal{F}$ of *function symbols* which is disjoint to $\mathcal{V}_E$. Each symbol is assigned a natural number, its arity. Each $n$-ary symbol $f$ is assigned a declaration describing the sorts of the domain and the range of $f$. These are denoted by $dom(f)$ and $rng(f)$, respectively.

The function symbols with $rng(f) = \texttt{bool}$ are called *predicate symbols*. The set of all predicate symbols in $\mathcal{F}$ is denoted by $\mathcal{P}$.

We assume that $\mathcal{F}$ contains at least the following symbols:

- The symbols common in set theory, e.g. $\emptyset$ and $\in$
- The symbol $\langle\ \rangle$ to construct tuples[1]
- The equality sign $=$
- The logical operators of first-order logic including nullary predicate symbols *true* and *false*

We write $\bigvee$ and $\bigwedge$ to denote existential and universal quantification, respectively, because $\exists$ is used as a temporal operator in TLA.

**Syntax** As it is usual in first-order logic, expressions can be built from variables and function symbols. These expressions are called *transition functions* in TLA. Transition predicates, state functions, and state predicates are defined as special kinds of transition functions.

[1] Strictly speaking, we need a symbol to denote the empty tuple and one to construct pairs.

The declaration of the function symbols is respected when constructing transition functions. The set of *transition functions* is the smallest set, such that

- If $x \in \mathcal{V}_E$, then $x$ is a transition function.
- If $f$ is an $n$-ary function symbol and $v_1, \ldots, v_n$ are transition functions with $\langle sort(v_1), \ldots, sort(v_n) \rangle = dom(f)$, then $f(v_1, \ldots, v_n)$ is a transition function with $sort(f(v_1, \ldots, v_n)) = rng(f)$.

Note that a transition function can contain primed variables.

In the following we do not adhere strictly to the prefix notation. Especially, the assumed symbols are written as usual. For example, $\in, \vee, \wedge, \ldots$ are written infix. The tuple of the transition functions $v_1, \ldots, v_n$ is written $\langle v_1, \ldots, v_n \rangle$.

If $v$ is a transition function with $sort(v) = \texttt{bool}$ then $v$ is called a *transition predicate*. Transition predicates are called *actions* in TLA.

*Free* variables and *substitution* can be defined as usual for first-order logic. A *state function* is a transition function with no free primed variables. Similarly, a *state predicate* is an action with no free primed variables. Often we call state predicates simply *predicates*.

**Semantics** In the following we define at first the semantics of transition functions over a valuation of all variables. From this we derive the semantics of a transition function over a pair of states and of state functions over a single state.

We assume a set $\mathcal{U}$ of values which contains at least the boolean values `true` and `false`. A *valuation* over a set of variables $\mathcal{W}$ is a mapping from $\mathcal{W}$ to $\mathcal{U}$ which is consistent with the sort of the variables. This means that each boolean variable is mapped to one of the boolean values. The other variables can be mapped to arbitrary values, even the boolean ones. A *state* is a valuation over the set $\mathcal{V}_F$ of flexible variables. The set of states is written $\mathbf{ST}(\mathcal{V}_F, \mathcal{U})$, or $\mathbf{ST}$ if $\mathcal{V}_F$ and $\mathcal{U}$ are clear from the context.

If $\alpha$ is a valuation over $\mathcal{W}$, $x \in \mathcal{W}$ is a variable, and $e \in \mathcal{U}$ is a value, then $\alpha\{x \leftarrow e\}$ is the valuation which equals $\alpha$ except that it maps $x$ to $e$. In the case that $x$ is a boolean variable, $e$ has to be one of the boolean values, otherwise $\alpha\{x \leftarrow e\}$ is not defined. Two valuations $\alpha$ and $\beta$ are called *similar up to* $x$ if and only if $\beta = \alpha\{x \leftarrow e\}$ for some $e$. This is written $\beta \simeq_x \alpha$.

Corresponding to the sequence $\mathcal{F}$ of function symbols we assume a sequence of functions with matching arity, domain, and range. There is a bijection between the function symbols and the functions which is denoted by $C$.

The meaning of a transition function $v$ under a valuation $\alpha$ over $\mathcal{V}_E$ is written $[\![ v ]\!]_\alpha$. It is defined as:

- $[\![ x ]\!]_\alpha$ is $\alpha(x)$, for $x \in \mathcal{V}_E$.

- $[\![ f(v_1, \dots, v_n) ]\!]_\alpha$ is $C(f)([\![ v_1 ]\!]_\alpha, \dots, [\![ v_n ]\!]_\alpha)$, for $f \in \mathcal{F}$ and transition functions $v_1, \dots, v_n$.

We require that the assumed function symbols have their usual meaning. I.e. $\emptyset$ denotes the empty set, $\in$ denotes set membership, etc. For convenience, we present the definitions for equality, some of the logical operators, and existential quantification. The other logical operators and universal quantification can be defined as usual. Let $v_1$ and $v_2$ be transition functions and $\mathcal{N}, \mathcal{N}_1$, and $\mathcal{N}_2$ be actions, then

- $[\![ v_1 = v_2 ]\!]_\alpha$ is `true` iff $[\![ v_1 ]\!]_\alpha$ equals $[\![ v_2 ]\!]_\alpha$.
- $[\![ \neg\mathcal{N} ]\!]_\alpha$ is `true` iff $[\![ \mathcal{N} ]\!]_\alpha$ is `false`.
- $[\![ \mathcal{N}_1 \wedge \mathcal{N}_2 ]\!]_\alpha$ is `true` iff $[\![ \mathcal{N}_1 ]\!]_\alpha$ is `true` and $[\![ \mathcal{N}_2 ]\!]_\alpha$ is `true`.
- $[\![ \bigvee x.\mathcal{N} ]\!]_\alpha$ is `true` iff $[\![ \mathcal{N} ]\!]_\beta$ is `true` for some $\beta$ with $\beta \simeq_x \alpha$.

The semantics $[\![ v ]\!]_{\alpha,\langle s,t\rangle}$ of a transition function $v$ under a valuation $\alpha$ over $\mathcal{V}_R$ and at a pair of states $\langle s, t\rangle$ is given by defining a valuation $E(\alpha, s, t)$ over $\mathcal{V}_E$ and using this in the definitions above:

$$[\![ v ]\!]_{\alpha,\langle s,t\rangle} \text{ is } [\![ v ]\!]_{E(\alpha,s,t)}$$

$E(\alpha, s, t)$ is defined itself as

$$\begin{array}{ll} E(\alpha,s,t)(x) = \alpha(x) & \text{if } x \in \mathcal{V}_R \\ E(\alpha,s,t)(x) = s(x) & \text{if } x \in \mathcal{V}_F \\ E(\alpha,s,t)(x') = t(x) & \text{if } x \in \mathcal{V}_F \end{array}$$

This means that unprimed flexible variables are interpreted over the first state and primed ones are interpreted over the second state. A state function does not contain free primed variables, therefore its meaning does not depend on the second state. Hence, if $f$ is a state function, $[\![ v ]\!]_{\alpha,\langle s,t\rangle}$ can be shortened to $[\![ v ]\!]_{\alpha,s}$.

Note, because actions are transition functions and predicates are state functions, their meaning is captured already by the definitions before.

**Abbreviations** Up to now only flexible variables can be primed. But priming a state function or predicate can be useful, too. This can be introduced as a temporal operator similar to the *next*-operator of temporal logic. To keep the number of temporal operators small we follow [Abadi and Merz, 1996] and define $f'$ as the result of replacing all occurrences of free flexible variables in the state function $f$ with their primed counterparts.

We defined already actions as predicates which are interpreted over a pair $\langle s, t\rangle$ of states and a valuation $\alpha$ over the rigid variables. If $\mathcal{N}$ is an action with $[\![ \mathcal{N} ]\!]_{\alpha,\langle s,t\rangle}$ equals `true` then we call $\langle s, t\rangle$ an $\mathcal{N}$*-step*. We also say that the action $\mathcal{N}$ is taken in the step $\langle s, t \rangle$. In TLA one often considers steps which are either *$\mathcal{N}$-steps* or which leave the value of a state function $f$ unchanged, i.e. such steps are $(\mathcal{N} \vee (f = f'))$-steps. The action $\mathcal{N} \vee (f = f')$ is abbreviated to $[\mathcal{N}]_f$. Although it is usually the case, it is not required that in an $\mathcal{N}$-step the value of $f$ changes actually. The action $\mathcal{N} \wedge (f \neq f')$ describes such steps and is abbreviated $\langle \mathcal{N} \rangle_f$. That a state function $f$ does not change its value is written $Uc(f)$.

$$\begin{aligned} [\mathcal{N}]_f &\equiv \mathcal{N} \vee f = f' \\ \langle \mathcal{N} \rangle_f &\equiv \mathcal{N} \wedge f \neq f' \\ Uc(f) &\equiv f = f' \end{aligned}$$

A step in which the value of a state function $f$ does not change is called an *$f$-stuttering step*. In an $f$-stuttering step the values of other state functions can change, especially the values assigned to flexible variables not occurring in $f$ can change. Most often, $f$ is clear from the context, then we speak of a stuttering step instead of an $f$-stuttering one.

An action $\mathcal{N}$ is called *enabled* in a state $s$ if there is a state $t$ such that $\langle s, t \rangle$ is an $\mathcal{N}$-step. This can be expressed by a state predicate *Enabled* $\mathcal{N}$. Let $x'_1, \ldots, x'_n$ be the free primed variables occurring in $\mathcal{N}$, then *Enabled* $\mathcal{N}$ can be defined as an abbreviation of quantifying existentially over the $x'_i$:

$$\mathit{Enabled}\,\mathcal{N} \equiv \bigvee x'_1 . \ldots \bigvee x'_n . \mathcal{N}$$

The order of quantification is not relevant.

In this dissertation we will quantify often over finite sets. Let $S = \{s_1, \ldots, s_n\}$ be a finite set and $\varphi$ be a predicate with free variable $x$, then we use $\bigvee_{x \in S} \varphi(x)$ as an abbreviation of the finite disjunction $\mathit{false} \vee \varphi(x/s_1) \vee \ldots \vee \varphi(x/s_n)$. $\varphi(x/s)$ denotes substitution of free occurrences of $x$ by $s$ in $\varphi$. Similarly we consider $\bigwedge_{x \in S} \varphi(x)$ as an abbreviation of finite conjunction. *false* and *true*, respectively, are relevant if $S$ is empty.

$$\bigvee_{x \in S} \varphi(x) \equiv \mathit{false} \vee \varphi(x/s_1) \vee \ldots \vee \varphi(x/s_n)$$

$$\bigwedge_{x \in S} \varphi(x) \equiv \mathit{true} \wedge \varphi(x/s_1) \wedge \ldots \wedge \varphi(x/s_n)$$

### 3.1.2 Temporal Logic Part of TLA

**Syntax** To define the syntax of *temporal* or *TLA-formulae* we define at first the atomic ones: Each predicate is an atomic TLA-formula. Furthermore, for an action $\mathcal{N}$ and a state function $f$, $\Box[\mathcal{N}]_f$ is an atomic TLA-formula. These are called *next-state relations*. Note that an action alone is not an atomic TLA-formula. Choosing

$\Box[\mathcal{N}]_f$ instead of $\mathcal{N}$ as atomic TLA-formula is essential and ensures that TLA is invariant under stuttering. This will be made more precise soon.

The set of atomic TLA-formulae is the smallest set, such that

- If $\varphi$ is a predicate, then $\varphi$ is an atomic TLA-formula.
- If $\mathcal{N}$ is an action, $f$ a state function, then $\Box[\mathcal{N}]_f$ is an atomic TLA-formula.

TLA-formulae are build from the atomic ones using the usual boolean connectives, the henceforth or always operator $\Box$ and existential quantification over rigid variables, written $\exists$, and over flexible variables, written $\boldsymbol{\exists}$. Quantification is used to "hide" variables.

The set of TLA-formulae is the smallest set containing the atomic TLA-formulae, such that:

- If $F$ is a TLA-formula, so is $\neg F$.
- If $F$ and $G$ are TLA-formulae, so is $F \wedge G$.
- If $F$ is a TLA-formula, so is $\Box F$.
- If $x \in \mathcal{V}_R$ is a rigid variable, $F$ a TLA-formula, so is $\exists x.F$.
- If $x \in \mathcal{V}_F$ is a flexible variable, $F$ a TLA-formula, so is $\boldsymbol{\exists} x.F$.

**Semantics** The meaning of a TLA-formula is defined over an infinite sequence $s_0, s_1, s_2, \ldots$ of states and a valuation of the rigid variables.

An infinite sequence $s_0, s_1, s_2, \ldots$ of states is called a *run*[2]. $\mathbf{ST}^\infty$ denotes the set of infinite sequences of states. If $\sigma = s_0, s_1, \ldots$ is a run, then $\sigma_{|n}$ is its suffix $s_n, s_{n+1}, \ldots$. If $\rho$ is a finite sequence of states, then $\rho \circ \sigma$ denotes the result of prefixing $\sigma$ with $\rho$. The finite sequence consisting of the states $t_0, \ldots, t_n$ is denoted by $\langle t_0, \ldots, t_n \rangle$.

The meaning of a TLA-formula $F$ over a run $\sigma$ and a valuation $\alpha$ over $\mathcal{V}_R$ is denoted by $[\![ F ]\!]_{\alpha,\sigma}$. We say that $\sigma$ *satisfies* $F$ (under $\alpha$) iff $[\![ F ]\!]_{\alpha,\sigma}$ is `true`. Usually, $\alpha$ is clear from the context and is not mentioned explicitly.

Informally, a run $\sigma$ satisfies a predicate $\varphi$, if $\varphi$ is satisfied by the first state of $\sigma$. A run satisfies a next-state relation $\Box[\mathcal{N}]_f$ if every consecutive pair of states is either an $\mathcal{N}$-step or an $f$-stuttering step. A run satisfies a TLA-formula $\Box F$ if every suffix of the run satisfies $F$. More precisely, let $\sigma = s_0, s_1, \ldots$ be a run, and $\alpha$ be a valuation over $\mathcal{V}_R$, then

- If $\varphi$ is a predicate, then $[\![ \varphi ]\!]_{\alpha,\sigma}$ is $[\![ \varphi ]\!]_{\alpha,s_0}$.

[2] A *run* is called a behaviour in [Lamport, 1994b]. We use the term *behaviour* to denote the set of runs satisfying a TLA-formula.

- If $\mathcal{N}$ is an action, $f$ a state function, then $[\![\, \Box[\mathcal{N}]_f \,]\!]_{\alpha,\sigma}$ is `true` iff $[\![\, [\mathcal{N}]_f \,]\!]_{\alpha,\langle s_n, s_{n+1}\rangle}$ is `true` for all $n \geq 0$.
- If $F$ is a TLA-formula, then $[\![\, \neg F \,]\!]_{\alpha,\sigma}$ is `true` iff $[\![\, F \,]\!]_{\alpha,\sigma}$ is `false`.
- If $F$ and $G$ are TLA-formulae, then $[\![\, F \wedge G \,]\!]_{\alpha,\sigma}$ is `true` iff $[\![\, F \,]\!]_{\alpha,\sigma}$ is `true` and $[\![\, G \,]\!]_{\alpha,\sigma}$ is `true`.
- If $F$ is a TLA-formula, then $[\![\, \Box F \,]\!]_{\alpha,\sigma}$ is `true` iff $[\![\, F \,]\!]_{\alpha,\sigma_{|n}}$ is `true` for all $n \geq 0$.

The symbols $\neg$ and $\wedge$ are used as first-order operators as well as temporal operators, hence the truth value of $\neg\varphi$ and $\varphi \wedge \psi$ for predicates $\varphi$ and $\psi$ can be determined in two different ways. Because these truth values coincide we do not distinguish between $\neg$ and $\wedge$ as first-order and temporal operators.

To define the meaning of $\exists$ and $\exists$ it is necessary to relate two runs w.r.t. stuttering equivalence. *Stuttering equivalence* is the smallest relation on runs, such that any two runs $\rho \circ \langle t, t \rangle \circ \sigma$ and $\rho \circ \langle t \rangle \circ \sigma$ are equivalent. The pair $\langle t, t \rangle$ is called a stuttering step, too. Note that *all* variables keep their values in such a stuttering step.

Two runs $\sigma = s_0, s_1, \ldots$ and $\tau = t_0, t_1, \ldots$ are *equal up to* $x$ iff $s_i \simeq_x t_i$ for all $i \geq 0$. $\sigma$ and $\tau$ are *similar up to* $x$, or $\sigma$ is an *$x$-variant* of $\tau$, iff there exists $\sigma'$ and $\tau'$ such that

- $\sigma'$ and $\tau'$ are equal up to $x$,
- $\sigma$ and $\sigma'$ as well as $\tau$ and $\tau'$ are stuttering equivalent.

We write $\sigma \simeq_x \tau$ if $\sigma$ is an $x$-variant of $\tau$. This means that two runs are equal up to $x$ if each corresponding pair of states is similar up to $x$. Two runs are similar up to $x$ if they are equal up to $x$ after adding or removing sufficiently many stuttering steps.

Informally, $\exists x.F$ is satisfied by a run $\sigma$ if $x$ can be assigned a value such that F is satisfied under $\sigma$. $\exists x.F$ is satisfied by a run $\sigma$ if there is an $x$-variant of $\sigma$ satisfying $F$. Hence $x$ can be assigned different values in different states.

- If $x \in \mathcal{V}_R$, then $[\![\, \exists x.F \,]\!]_{\alpha,\sigma}$ is `true` iff $[\![\, F \,]\!]_{\beta,\sigma}$ is `true` for some valuation $\beta$ with $\beta \simeq_x \alpha$.
- If $x \in \mathcal{V}_F$, then $[\![\, \exists x.F \,]\!]_{\alpha,\sigma}$ is `true` iff $[\![\, F \,]\!]_{\alpha,\tau}$ is `true` for some run $\tau$ with $\sigma \simeq_x \tau$.

If $\sigma$ satisfies $F$ (w.r.t. $\alpha$) then $\sigma$ is called a *model* of $F$ (w.r.t. $\alpha$), we write $\sigma \models_\alpha F$ or $\sigma \models F$. The set of runs satisfying $F$ (under $\alpha$) is called the *behaviour* of $F$ (w.r.t. $\alpha$). A TLA-formula $F$ is *valid* iff $\sigma \models_\alpha F$ for all runs $\sigma$ and valuations $\alpha$ over $\mathcal{V}_R$. We write $\models F$. Usually, $\alpha$ is clear from the context and is not mentioned explicitly.

**Invariance under Stuttering** In Section 3.3.1 we will define refinement for TLA. In connection with refinement it is an important property of a logic that it cannot distinguish between stuttering equivalent runs. This is called *invariance under stuttering*. It can be shown that TLA is invariant under stuttering.

**Proposition 3.1 ([Abadi and Merz, 1996])** *If $F$ is a TLA-formulae, $\alpha$ a valuation over $\mathcal{V}_R$, and $\sigma$ and $\tau$ are two stuttering equivalent runs, then $[\![ F ]\!]_{\alpha,\sigma}$ equals $[\![ F ]\!]_{\alpha,\tau}$.*

Hence, the behaviour of a TLA-formula is closed under stuttering equivalence.

**Abbreviations** We give abbreviations for some commonly used TLA-formulae. The boolean connectives $\vee$, $\Leftrightarrow$, and $\Rightarrow$ are defined as usual. Also, $\Diamond F$, *eventually* $F$, is defined as usual. It means that $F$ will be valid eventually.

$\Diamond F \equiv \neg\Box\neg F$, for all TLA-formulae $F$.

$\Diamond \langle \mathcal{N} \rangle_f \equiv \neg\Box[\neg\mathcal{N}]_f$, for all actions $\mathcal{N}$ and state functions $f$.

That whenever the TLA-formulae $F$ is valid, the TLA-formulae $G$ will be valid eventually, too, is expressed by $\Box(F \Rightarrow \Diamond G)$. This is called $F$ *leadsto* $G$ and is written $F \leadsto G$, i.e.:

$F \leadsto G \equiv \Box(F \Rightarrow \Diamond G)$, for all TLA-formulae $F$ and $G$.

**Fairness** Informally, that something will happen eventually is called a *liveness property*. In contrast, a *safety property* states that something does not happen. As an example, a next-state relation in TLA expresses a safety property, $F \leadsto G$ expresses a liveness property. *Properties* are defined more formally in Section 3.3.

Describing liveness properties by formulae $\Diamond F$ or $F \leadsto G$ can have unwanted consequences. It is possible that safety properties are implied by such liveness formulae. Let us assume for example that the formula $\Box\Diamond(x = 0)$ is conjoined to $init_\Phi \wedge \Box[\mathcal{M}]_{\langle x,y \rangle}$ as defined in Figure 3.1, page 27. $\Box\Diamond(x = 0)$ expresses the liveness property that infinitely often $x$ equals 0. This implies that $x$ never changes its value, which is a safety property.

To avoid this problem, liveness is expressed in TLA usually in terms of *fairness*. Informally, fairness means, that if an action can be taken, then it must be taken eventually. There are two variants of fairness, viz *weak* and *strong* fairness.

For an action $\mathcal{N}$ and a state function $f$ *weak fairness* means, that if the action $\langle \mathcal{N} \rangle_f$ is enabled continuously then eventually there has to be a $\langle \mathcal{N} \rangle_f$-step. This is equivalent to state that there are infinitely many $\langle \mathcal{N} \rangle_f$-steps or there are infinitely many steps in which $\langle \mathcal{N} \rangle_f$ is not enabled. *Strong fairness* means, that if $\langle \mathcal{N} \rangle_f$

is enabled infinitely often, then infinitely many $\langle \mathcal{N} \rangle_f$-steps have to occur. This is equivalent to state that there are infinitely many $\langle \mathcal{N} \rangle_f$-steps or $\langle \mathcal{N} \rangle_f$-is enabled only finitely often. Weak and strong fairness of an action $\langle \mathcal{N} \rangle_f$ are denoted by $WF_f(\mathcal{N})$ and $SF_f(\mathcal{N})$, respectively. These are defined as abbreviations as follows:

$$\begin{array}{lcl} WF_f(\mathcal{N}) & \equiv & \Box\Diamond\langle \mathcal{N} \rangle_f \vee \Box\Diamond\neg\, Enabled\,\langle \mathcal{N} \rangle_f \\ SF_f(\mathcal{N}) & \equiv & \Box\Diamond\langle \mathcal{N} \rangle_f \vee \Diamond\Box\neg\, Enabled\,\langle \mathcal{N} \rangle_f \end{array}$$

**TLA-Programs** Usually a TLA-formula of the form $\exists x_1. \ldots \exists x_n.\ init \wedge \Box[\mathcal{N}]_f \wedge L$ is used to describe the behaviour of a system in TLA, where

- $init$ is a predicate.
- $\mathcal{N}$ is an action.
- $f$ is the tuple of the flexible variables occurring in $init$ and $\mathcal{N}$.
- $L$ is the conjunction of TLA-formulae $WF_f(\mathcal{M})$ and $SF_f(\mathcal{M})$, where $M$ is an action used to define $\mathcal{N}$.

The TLA-formulae $\Phi$ in Figure 3.1, page 27, is of this form. A TLA-formula of this form is called a *TLA-program*.

### 3.1.3 Notational Conventions

The formulae needed to describe the semantics of statecharts tend to become very large. Therefore we use syntactical conventions similar to [Lamport, 1994a] to enhance readability of formulae. By using indentation and lists of formulae bulleted with logical operators we present the structure of formulae more clearly and try to avoid a vast number of parentheses. If it is advantageous, e.g. for writing small formulae in text, we still use conventional notation. The correspondence between this kind of writing formulae and the conventional one should become clear from the examples in Table 3.1. The temporal operators $\Box$ and $\Diamond$ are handled in the

| | | |
|---|---|---|
| $\begin{array}{l} \wedge \vee a \\ \quad \vee b \\ \wedge \vee c \\ \quad \vee d \end{array}$<br>$\equiv$<br>$(a \vee b) \wedge (c \vee d)$ | $\begin{array}{l} \wedge \vee \neg a \\ \quad \vee \neg b \\ \wedge \neg \vee c \\ \quad\quad \vee d \end{array}$<br>$\equiv$<br>$(\neg a \vee \neg b) \wedge \neg(c \vee d)$ | $\begin{array}{l} \vee a \\ \vee b \\ \Rightarrow \vee c \\ \quad \vee d \end{array}$<br>$\equiv$<br>$(a \vee b) \Rightarrow (c \vee d)$ |

Table 3.1: Notational conventions

same way as $\neg$ and $\leadsto$ is handled in the same way as $\Rightarrow$. Similarly, we write lists bulleted with the set-theoretic operators $\cap$ and $\cup$ to denote intersection and union, respectively, of several sets.

TLA is an untyped logic. To enhance readability of definitions we precede the definitions of functions and predicates by a corresponding *declaration*. Let $S$ and $T$ be two sets and $f$ a function from $S$ to $T$, then we write the declaration

$$f: S \rightarrow T$$

For a predicate $\varphi$ we write $\varphi: S$ instead of $\varphi: S \rightarrow Bool$.

In this dissertation we will write names of predicates using small letters only. The names of actions will be capitalised.

## 3.2 Example: Traffic Light System

To give an impression, how the behaviour of a reactive system can be described in TLA we present in the following the requirements of a simple traffic light system. The example emphasises the use of actions to describe a behaviour. To keep the example simple we do not distinguish between the description of the uncontrolled plant, i.e. RW-O, and the intended behaviour of the controlled plant, i.e. REQ.

At first we present an informal description of the system and its requirements. In a formal description we identify at first the necessary variables. Then the intended behaviour is described by a TLA-program. The main part is the definition of the actions constituting the next-state relation. At last we give some remarks on this kind of describing behaviours and on this specific description.

### 3.2.1 Informal Description

The situation is as depicted in Figure 3.2. There is a crosswalk with traffic lights for the cars on the road as well as for the pedestrians. On either side of the road there is a button, which can be pushed by a pedestrian to signal that the pedestrian wants to cross the road. The goal of the system is to allow pedestrians to cross the road safely. The requirements on the system can be described informally by the following scenario:

> The traffic on the road shall be stopped if a pedestrian pushes one of the buttons. To stop the traffic the lightings of the traffic lights are changed as usual. When the pedestrian has crossed the road, the lightings of the traffic lights are changed such that the traffic can flow again.

How it is detected that the pedestrian has crossed the road is left unspecified. We assume that both traffic lights for the pedestrians and the cars, respectively, show

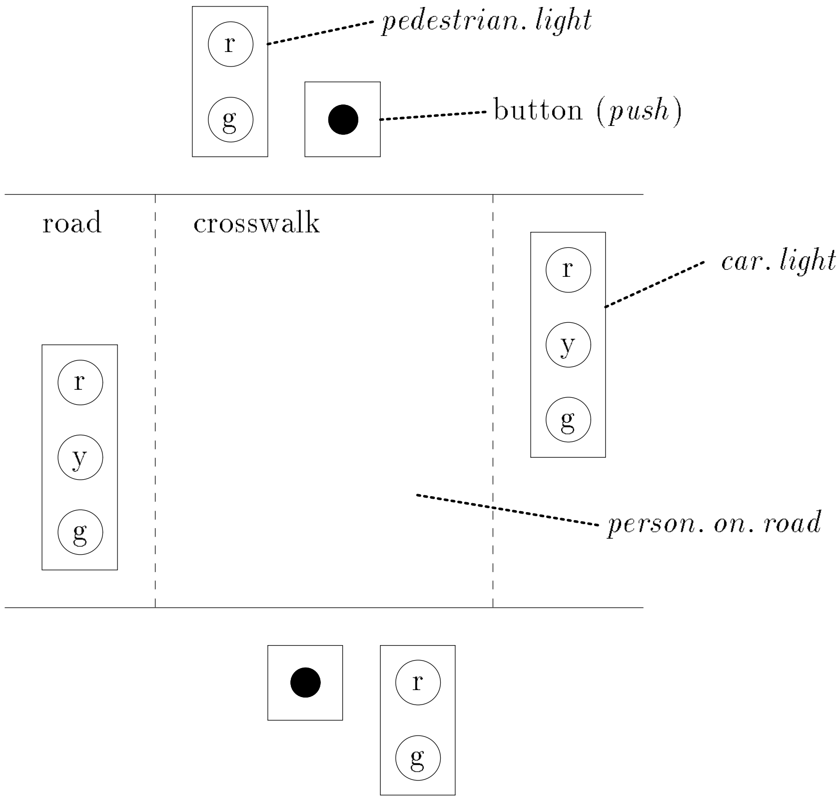



Figure 3.2: Traffic light scene

always the same lighting. Furthermore, for the sake of simplicity we do not distinguish between the buttons on either side of the road. As a further simplification, we do not consider real-time properties, i.e. how long the different lightings of the traffic lights are shown at least or at most.

### 3.2.2 Formal Description

The behaviour of the system is based on the following set of variables. These are summarised in Table 3.2 together with their sorts. For the variables *pedestrian. light* and *car. light* the set of possible values which shall be assigned to them are shown. More precisely, these variables are of sort `any`.

Whether the button is pushed is represented by the variable *push*. The variable *request* is used as an auxiliary variable to store that the button has been pressed, but the traffic has not been stopped yet. The variable *person. on. road* represents whether a person is on the road in the area of the crosswalk. The variables *pedestrian. light* and *car. light* represent the current lighting of the corresponding traffic lights.

All these variables are flexible ones, no rigid variables are used in this example. $w$

denotes the tuple of the variables:

$$w \equiv \langle push, request, person.\,on.\,road, pedestrian.\,light, car.\,light\rangle$$

Initially, the system shall not be unsafe already. We require that both traffic lights show their red light. Furthermore we assume that neither the button is pushed nor that there is a pending request, no assumption is made on whether a person is on the road:

$$\begin{aligned} tls.\,init^{3} \equiv\ & \wedge\ car.\,light = \texttt{red} \\ & \wedge\ pedestrian.\,light = \texttt{red} \\ & \wedge\ \neg\, push \\ & \wedge\ \neg\, request \end{aligned}$$

Each step of the system is described by one of the following actions. These consist of two parts: One part describing which variables are changed under which condition and a second part describing which variables are not changed.

The button can be pushed at any point of time. That it has been pushed is stored in the variable *request* before the button is released. This ordering is achieved by stating that the button can be released only if *request* is true. No assumption is made on whether a person is on the road. The values of the other variables are not changed.

$$\begin{aligned} Push.\,button \equiv\ & \wedge\ \neg\, push \\ & \wedge\ push' \\ & \wedge\ Uc(\langle request, pedestrian.\,light, car.\,light\rangle) \end{aligned}$$

$$\begin{aligned} Store.\,request \equiv\ & \wedge\ push \\ & \wedge\ \neg\, request \\ & \wedge\ request' \\ & \wedge\ Uc(\langle push, pedestrian.light, car.\,light\rangle) \end{aligned}$$

$$\begin{aligned} Release.\,button \equiv\ & \wedge\ request \\ & \wedge\ push \\ & \wedge\ \neg\, push' \\ & \wedge\ Uc(\langle request, pedestrian.\,light, car.\,light\rangle) \end{aligned}$$

[3] Let $W$ denote $Bool \times Bool \times Bool \times \{\texttt{red}, \texttt{green}\} \times \{\texttt{red}, \texttt{red.yellow}, \texttt{yellow}, \texttt{green}\}$, then the declaration of *tls. init* is $W$. The declaration of all actions in this description is $W \times W$.

| | |
|---|---|
| *push*: | `bool` |
| *request*: | `bool` |
| *person. on. road*: | `bool` |
| *pedestrian. light*: | $\{\texttt{red}, \texttt{green}\}$ |
| *car. light*: | $\{\texttt{red}, \texttt{red.yellow}, \texttt{yellow}, \texttt{green}\}$ |

Table 3.2: Traffic light system: Variables

The traffic light of the pedestrians switches from `red` to `green` if the cars are shown a red light and if there is a pending request. Again, no assumption is made on whether a person enters the road. Vice versa, it switches from `green` to `red` if no person is on the road. Simultaneously the request is considered as being served.

$$\begin{aligned} \mathit{Pedestrian.r.g} \equiv\ & \wedge\ \mathit{car.light} = \mathtt{red} \\ & \wedge\ \mathit{request} \\ & \wedge\ \mathit{pedestrian.light} = \mathtt{red} \\ & \wedge\ \mathit{pedestrian.light}' = \mathtt{green} \\ & \wedge\ \mathit{Uc}(\langle \mathit{push}, \mathit{request}, \mathit{car.light} \rangle) \end{aligned}$$

$$\begin{aligned} \mathit{Pedestrian.g.r} \equiv\ & \wedge\ \neg\, \mathit{person.on.road} \\ & \wedge\ \mathit{pedestrian.light} = \mathtt{green} \\ & \wedge\ \mathit{pedestrian.light}' = \mathtt{red} \\ & \wedge\ \neg\, \mathit{request}' \\ & \wedge\ \mathit{Uc}(\langle \mathit{push}, \mathit{car.light} \rangle) \end{aligned}$$

In the other actions it is left open, whether a person is on the road. That one leaves the road is described explicitly.

$$\begin{aligned} \mathit{Person.leaves.road} \equiv\ & \wedge\ \mathit{person.on.road} \\ & \wedge\ \neg\, \mathit{person.on.road}' \\ & \wedge\ \mathit{Uc}(\langle \mathit{push}, \mathit{request}, \mathit{pedestrian.light}, \mathit{car.light} \rangle) \end{aligned}$$

Similarly, the traffic light of the cars switches from `red` to `red.yellow` only if there is no request. Vice versa it switches from `green` to `yellow` if there is a request. Switching from `yellow` to `red` and from `red.yellow` to `green` is done unconditionally.

$$\begin{aligned} \mathit{Car.r.ry} \equiv\ & \wedge\ \neg\, \mathit{request} \\ & \wedge\ \mathit{car.light} = \mathtt{red} \\ & \wedge\ \mathit{car.light}' = \mathtt{red.yellow} \\ & \wedge\ \mathit{Uc}(\langle \mathit{push}, \mathit{request}, \mathit{pedestrian.light} \rangle) \end{aligned}$$

$$\begin{aligned} \mathit{Car.ry.g} \equiv\ & \wedge\ \mathit{car.light} = \mathtt{red.yellow} \\ & \wedge\ \mathit{car.light}' = \mathtt{green} \\ & \wedge\ \mathit{Uc}(\langle \mathit{push}, \mathit{request}, \mathit{pedestrian.light} \rangle) \end{aligned}$$

$$\begin{aligned} \mathit{Car.g.y} \equiv\ & \wedge\ \mathit{request} \\ & \wedge\ \mathit{car.light} = \mathtt{green} \\ & \wedge\ \mathit{car.light}' = \mathtt{yellow} \\ & \wedge\ \mathit{Uc}(\langle \mathit{push}, \mathit{request}, \mathit{pedestrian.light} \rangle) \end{aligned}$$

$$
\begin{aligned}
\mathit{Car.\,y.\,r} \equiv\ & \wedge\ \mathit{car.\,light} = \texttt{yellow} \\
& \wedge\ \mathit{car.\,light}' = \texttt{red} \\
& \wedge\ \mathit{Uc}(\langle \mathit{push}, \mathit{request}, \mathit{pedestrian.\,light} \rangle)
\end{aligned}
$$

The next-state relation of the traffic light system is the disjunction of these actions:

$$
\begin{aligned}
\mathit{Tls.\,next} \equiv\ & \vee\ \mathit{Push.\,button} \vee \mathit{Store.\,request} \vee \mathit{Release.\,button} \\
& \vee\ \mathit{Pedestrian.\,r.\,g} \vee \mathit{Pedestrian.\,g.\,r} \\
& \vee\ \mathit{Person.\,leaves.\,road} \\
& \vee\ \mathit{Car.\,r.\,ry} \vee \mathit{Car.\,ry.\,g} \vee \mathit{Car.\,g.\,y} \vee \mathit{Car.\,y.\,r}
\end{aligned}
$$

The TLA-formula $\mathit{tls.\,init} \wedge \Box[\,\mathit{Tls.\,next}\,]_w$ describes the behaviour of the traffic light system without considering fairness. Four different subsequences of runs satisfying $\mathit{tls.\,init} \wedge \Box[\,\mathit{Tls.\,next}\,]_w$ are depicted in Table 3.3. Each of the sequences starts and ends in an initial state, i.e. one satisfying *tls. init*. No stuttering steps are shown in these sequences. The value of *person. on. road* is given only, if it is of interest. The values which changed in a step are written boldface in the new state. The action being taken in a step is written between the states.

The first subsequence — a) – b) — corresponds to the scenario used to describe the requirements informally. The second one — b) – c) — is a slight variation of the first one. The action *Release. button* is delayed for some time. This shows that

| | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | a) | | | | | | | | | | b) | | | | | | | | | | c) |
| *push* | f | f | f | **t** | t | **f** | f | f | f | f | f | f | f | **t** | t | t | t | t | **f** | f | f |
| *request* | f | f | f | f | **t** | t | t | t | t | t | **f** | f | f | f | **t** | t | t | t | t | t | f |
| *person. on. road* | | | | | | | | | **t** | **f** | | | | | | | | **t** | t | **f** | |
| *pedestrian. light* | r | r | r | r | r | r | r | r | **g** | g | **r** | r | r | r | r | r | r | **g** | g | g | r |
| *car. light* | r | **ry** | **g** | g | g | g | **y** | **r** | r | r | r | **ry** | **g** | g | g | **y** | **r** | r | r | r | r |
| action taken | | 6 | 7 | 0 | 1 | 2 | 8 | 9 | 3 | 5 | 4 | 6 | 7 | 0 | 1 | 8 | 9 | 3 | 2 | 5 | 4 |

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | d) | | | | | | e) | | | | | | | | | f) |
| *push* | f | **t** | t | **f** | f | f | f | f | f | **t** | t | **f** | f | f | f | f |
| *request* | f | f | **t** | t | t | t | **f** | f | f | f | **t** | f | t | t | t | **f** |
| *person. on. road* | | | | | **t** | **f** | | | | | | f | | | **f** | |
| *pedestrian. light* | r | r | r | r | **g** | g | **r** | r | r | r | r | r | r | r | **g** | **r** |
| *car. light* | r | r | r | r | r | r | r | **ry** | **g** | g | g | g | **y** | **r** | r | r |
| action taken | | 0 | 1 | 2 | 3 | 5 | 4 | 6 | 7 | 0 | 1 | 2 | 8 | 9 | 3 | 4 |

| | | |
|---|---|---|
| $0 \equiv \mathit{Push.\,button}$ | $3 \equiv \mathit{Pedestrian.\,r.\,g}$ | $6 \equiv \mathit{Car.\,r.\,ry}$ |
| $1 \equiv \mathit{Store.\,request}$ | $4 \equiv \mathit{Pedestrian.\,g.\,r}$ | $7 \equiv \mathit{Car.\,ry.\,g}$ |
| $2 \equiv \mathit{Release.\,button}$ | $5 \equiv \mathit{Person.\,leaves.\,road}$ | $8 \equiv \mathit{Car.\,g.\,y}$ |
| | | $9 \equiv \mathit{Car.\,y.\,r}$ |

Table 3.3: Subsequences of runs satisfying $\mathit{tls.\,init} \wedge \Box[\,\mathit{Tls.\,next}\,]_w$.

this reactive system behaves non-deterministically. Here, the non-determinism can be contributed to the environment: Persons are free to push or release a button whenever they want.

In the third subsequence — d) – e) — it is shown that between serving two requests it is not necessarily the case, that the traffic on the road is allowed to flow. Clearly this can be repeated ad infinitum, hence there is a run in which the traffic light of the cars remains red forever.

The fourth subsequence — e) – f) — depicts the situation that no person enters the road immediately when the traffic light of the pedestrians becomes green. In this situation it switches back to red and the request is considered as granted.

In this example, fairness cannot be required uniformly for all actions. On some actions, no fairness can be required at all. These are the actions which depend on the environment of the system: *Push. button*, *Release. button*, and *Person. leaves. road*. On most other actions weak fairness can be required. This is sufficient to ensure that something will happen eventually. It becomes clear from the third subsequence — d) - e) — shown in Table 3.3 that the action *Car. r. ry* needs special attention. To exclude that the traffic light for the cars never becomes green we require strong fairness on this action. Hence, it is not possible that the action *Store. request* is taken infinitely often instead of the action *Car. r. ry* if both of them are enabled together infinitely often.

The variable *request* is used only for internal purposes of the description. It should not be made visible to an external observer. Therefore it is hidden by quantifying existentially over it.

The final form of the description is

$$
\begin{aligned}
Tls \equiv \exists\, request\, .\, & \land\ tls.\, init \\
& \land\ \Box[\, Tls.\, next\, ]_w \\
& \land\ WF_w(\, Store.\, request\,) \\
& \land\ WF_w(\, Pedestrian.\, r.\, g\,) \land WF_w(\, Pedestrian.\, g.\, r\,) \\
& \land\ WF_w(\, Car.\, ry.\, g\,) \land WF_w(\, Car.\, y.\, r\,) \land WF_w(\, Car.\, ry.\, g\,) \\
& \land\ SF_w(\, Car.\, r.\, ry\,)
\end{aligned}
$$

### 3.2.3 Remarks on the Formal Description

The actions in the formal description have a typical form. The changes of some variables are described and other variables are required to not change their values. This ensures, that at most one of the actions can be taken in one step. This is not required in TLA. In principle it is possible that more than one action can be taken in a step because two actions can be satisfied by the same pair of states and they are combined by disjunction in the next-state relation. Nevertheless it is usually the case that only one action can be taken in a step which is described by the next-state relation of a TLA-program. Other approaches are shown in [Abadi and Merz, 1996].

In the formal description, the traffic light system has been considered as one monolithic system. Looking more closely at it, four different components of it can be identified:

- the button
- the pedestrians
- the traffic light of the pedestrians
- the traffic light of the cars

It is possible to describe each of the components separately and to compose their descriptions by conjunction. The different parts communicate via common variables. This is considered in detail in [Abadi and Lamport, 1995]. Note that TLA does not provide syntactical means to describe the interface of a component.

By this formal description, an abstract model of the system is given. Hence, at least the safety properties are written in a model-oriented style. The fairness properties are expressed explicitly, hence these can be considered as being in a property-oriented style. When considering the individual components separately, the safety properties of the system can also be given in a more property-oriented manner: The behaviour of the individual components can be described separately, but properties of their composition can be expressed explicitly. In the traffic light system, this is for example the property

$$\begin{array}{l} \Box\neg\wedge\ \mathit{pedestrian.light} = \mathtt{green} \\ \quad\ \wedge\ \mathit{car.light} \neq \mathtt{red} \end{array}$$

This approach will be made more clear by the examples in Chapter 7.

## 3.3 Advanced Concepts of TLA

In Section 3.1 two aspects have been mentioned, but not considered further. These are refinement and the problem, that a liveness property can imply safety properties. Both aspect are relevant in Chapters 8 and 9. They are not necessary to understand Chapters 4 to 7 and therefore this section can be skipped on first reading.

### 3.3.1 Refinement

Let $\Psi$ and $\Phi$ be two TLA-programs, then $\Psi$ *refines* or *implements* $\Phi$ if any run satisfying $\Psi$ can be interpreted as one of $\Phi$. This means that $\Psi$ refines $\Phi$ if the formula

$$\Psi \;\Rightarrow\; \Phi \tag{3.1}$$

is valid. Hence, refinement is logical implication in TLA. Usually, internal variables of TLA-programs are hidden by quantifying existentially over them. In this case Equation 3.1 is written in the following form, cf. [Lamport, 1994b, page 907]

$$(\exists x_1. \ldots \exists x_n. \Psi) \Rightarrow (\exists y_1. \ldots \exists y_m. \Phi) \tag{3.2}$$

To prove this formula, it has to be shown, that each run satisfying $\exists x_1. \ldots \exists x_n. \Psi$ coincides with a run satisfying $\exists y_1. \ldots \exists y_m. \Phi$ on the variables they have in common.

It can be the case, that one step in $\Phi$ is implemented by several steps of $\Psi$. Because TLA is invariant under stuttering, cf. Proposition 3.1, runs satisfying $\Phi$ can be extended by stuttering steps to match the sequence of (sub-)steps of $\Psi$.

In addition, values of the hidden variables $y_1, \ldots, y_m$ of $\Phi$ must be derived. This is done by defining state functions $\overline{y_1}, \ldots, \overline{y_m}$ in terms of the variables occurring in $\Psi$, including its internal variables $x_1, \ldots, x_n$. Then it is sufficient to prove $\Psi \Rightarrow \overline{\Phi}$, where $\overline{\Phi}$ denotes $\Phi$ with occurrences of $y_i$ replaced by the state function $\overline{y_i}$. Such a collection of state functions is called a *refinement mapping*. In [Abadi and Lamport, 1991] it is shown under which conditions on $\Psi$ and $\Phi$ the existence of such refinement mappings follows from validity of Equation 3.2.

### 3.3.2 Properties and Machine Closedness

In Section 3.1 an example of a liveness property has been given which implies a safety property: Conjoining $\Box\Diamond(x = 0)$ to the TLA-formula $init_\Phi \wedge \Box[\mathcal{M}]_{\langle x,y \rangle}$ as defined in Figure 3.1, page 27, implies that $x$ remains 0. In this section it is shown under which conditions the use of fairness instead of liveness avoids this problem.

A *property* is a set of runs which is closed under stuttering equivalence. Remember that stuttering equivalence was the finest relation on runs, such that any two runs $\rho \circ \langle t, t \rangle \circ \sigma$ and $\rho \circ \langle t \rangle \circ \sigma$ are equivalent, where $t$ is a state, $\rho$ is a finite sequence of states, and $\sigma$ is a run. In the stuttering step $\langle t, t \rangle$ all variables keep their values. Because TLA is invariant under stuttering, cf. Proposition 3.1, it follows that the set of runs satisfying a TLA-formula is a property.

A property is defined as a special set of runs, i.e. a set of infinite sequences of states. We say that a *finite* sequence $\rho = \langle r_0, \ldots, r_n \rangle$ of states *satisfies* a property $F$ if the run $r_0, \ldots, r_n, r_n, \ldots$ is in $F$. A property $F$ is a *safety property* if every finite prefix of every run in $F$ satisfies $F$. Each TLA-formula $init \wedge \Box[\mathcal{N}]_f$ describes a safety property.

For an arbitrary property $F$, the *closure* $\mathcal{C}(F)$ of $F$ is the smallest safety property which contains $F$. Let $\Phi$ be a TLA-formula, then the closure of the property described by $\Phi$ can be described itself by a TLA-formula. This is written $\mathcal{C}(\Phi)$, cf. [Abadi and Merz, 1996].

To avoid that liveness properties imply safety properties, one is interested in properties which are described by a formulae $\Pi \wedge L$, such that $\Pi$ describes a safety property, $L$ describes an arbitrary property and $L$ does not imply any further safety property. In this case the pair of formulae $(\Pi, L)$ is called *machine closed*. This can be defined formally as:

$$\mathcal{C}(\Pi \wedge L) = \Pi \tag{3.3}$$

Expressing fairness on an action $\mathcal{N}$ does not imply new safety properties if $\mathcal{N}$-steps are already "contained" in the safety property $\Pi$. An action $\mathcal{N}$ is called a *subaction* of a safety property $\Pi$ if whenever an $\mathcal{N}$-step is possible, then one satisfying $\Pi$ is possible, too. Formally, $\mathcal{N}$ is a subaction of $\Pi$ iff for every finite sequence $\langle r_0, \ldots, r_n \rangle$ which satisfies $\Pi$ and every state $r_n$ which satisfies $\mathit{Enabled}\,\mathcal{N}$, there is a state $r_{n+1}$ such that $\langle r_n, r_{n+1} \rangle$ is an $\mathcal{N}$-step and $\langle r_0, \ldots, r_{n+1} \rangle$ satisfies $\Pi$. In TLA, $\mathcal{N}$ is a subaction of $\mathit{init} \wedge \Box[\mathcal{M}]_f$ iff

$$\begin{aligned}(\mathit{init} \wedge \Box[\mathcal{M}]_f) \ \Rightarrow \ \Box\ & \mathit{Enabled}\,\mathcal{N} \\ & \Rightarrow \mathit{Enabled}(\mathcal{N} \wedge [\mathcal{M}]_f)\end{aligned} \tag{3.4}$$

It follows, that if $\mathcal{N}$ implies $\mathcal{M}$, the action $\mathcal{N}$ is a subaction of $\mathit{init} \wedge \Box[\mathcal{M}]_f$. The following proposition states that conjoining fairness formulae on subactions to a safety property does not introduce new safety properties.

**Proposition 3.2 ([Abadi and Lamport, 1994, page 1550])** *Let $\Pi$ be a safety property and $L$ be the conjunction of a finite or countable infinite number of formulae of the form $WF_w(\mathcal{N})$ and/or $SF_w(\mathcal{N})$ such that each $\langle \mathcal{N} \rangle_w$ is a subaction of $\Pi$, then $(\Pi, L)$ is machine closed.*

# Chapter 4

# Statecharts

To achieve a combination of statecharts and TLA a common semantic domain has to be identified and both description techniques have to be mapped to it. We want to retain the semantic domain of TLA because it is well-suited for considering refinement. Hence we have to map the semantics of statecharts to this semantic domain. This is done by constructing a TLA-program for each statechart which can be considered as equivalent to the statechart.

Before constructing such TLA-programs we introduce the central concepts of statecharts by presenting a description of the traffic light system, cf. Section 3.2, by a statechart.

Then we define the semantics of a large subset of statecharts formally by constructing a TLA-program for each statechart. This is done in two parts. In the remainder of this chapter we concentrate on how steps of a statechart are built and how they contribute to the behaviour of a statechart. We do not consider variables of a statechart and we use only a restricted language to label transitions. These restrictions are removed in Chapter 5.

Several variants of statecharts exist in literature. We will follow the one presented in [Harel and Naamad, 1996]. Some concepts of statecharts which are not relevant in this work will not be defined formally. On the other hand we will extend statecharts by a special kind of transitions, so-called static transitions. At the end of this chapter we review the way we defined the semantics of statecharts and compare it to other approaches.

## 4.1 Introduction to Statecharts

In this section we explain informally the central concepts of statecharts. These are illustrated by a description of the traffic light system which was already used as an example in Section 3.2. We restrict ourselves at first to a subset of the notation. This subset is sufficient to describe the behaviour of the traffic light system. After introducing the central concepts we give a short summary of those concepts which are not explained here. Thereafter we give some remarks on how the behaviour of a system can be described by a statechart.

Statecharts can be used to describe the behaviour of reactive systems. A *behaviour* is a set of runs, where a *run* is a sequence of "snapshots" of the situation of the system. Such a snapshot is called a *status*. The first status in a sequence is called the initial status. Each subsequent status is obtained from its predecessor by taking a *step*.

Because we are using two formal description techniques with overlapping terminology, the following has to be noted:

> *Although a similar terminology is used in statecharts as in TLA, some of the terms have a different meaning. A* run *was defined in TLA as an infinite sequence of* states. *In statecharts, a run is a sequence of* statuses*, the sequence can be finite or infinite. A* state *in TLA is a valuation of the flexible variables. In statecharts, a state is a syntactical element.*
>
> *These terms with their different meanings are used in most references on TLA and statecharts, respectively. To correspond with these references we did not rename the terms to distinguish their meaning. Instead, we write* TLA-*run,* statecharts-*run, etc., if it does not become clear from the context which of the meanings is the intended one.*

In the following we describe at first which information is contained in a status, at second we show how steps are described in statecharts.

### 4.1.1 Status

In statecharts, *states* are related hierarchically in a tree-like fashion. The leaves are called *basic*-states. There are two types of non-leave states: *And*- and *or*-states. To describe the behaviour of the traffic light system we use a statechart with the states as shown in Figure 4.1. A state is drawn as a box with rounded corners. The hierarchical relationship is expressed by inclusion of the boxes. For example, trafficlightsystem.pedestrian.red.wait is a *substate* of trafficlightsystem.pedestrian.red.

The complete name of a state $A$ is the sequence of names of the states on the path from the root-state to $A$. Usually we write only a subsequence which is unique and which allows to locate the state in the diagram easily.

And-states are distinguished from or-states in the statechart by writing the name of and-states in an extra rectangle placed on top of its box and by the dashed line separating its direct descendants. The hierarchy and the type of the states of the statechart in Figure 4.1 are shown in Figure 4.2. The state trafficlightsystem is the root-state of the statechart.

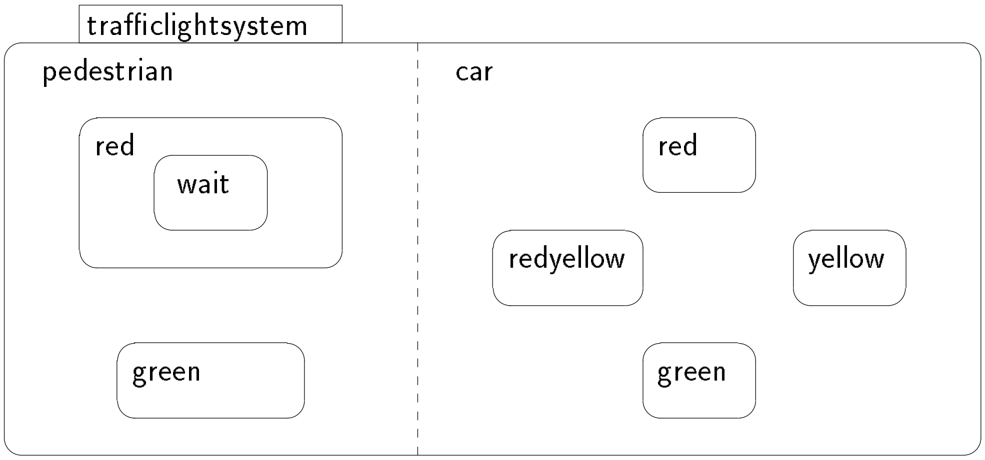



Figure 4.1: Traffic light system: States of the statecharts description

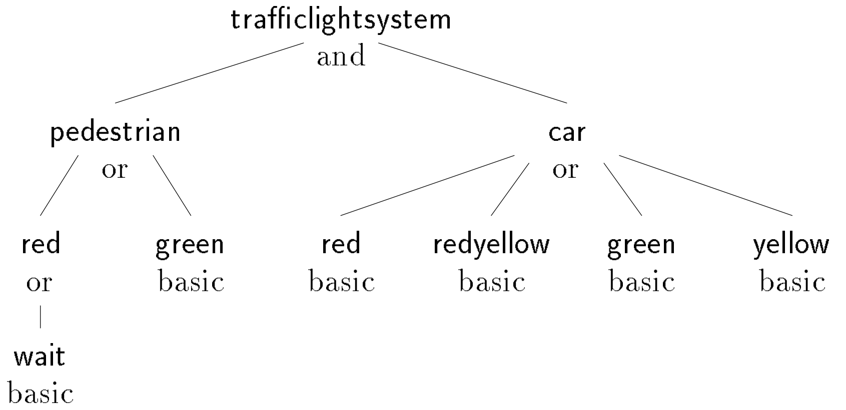



Figure 4.2: Traffic light system: Hierarchy and type of states

---

A *status* describes — besides other information — which of the states are currently active. The hierarchy and the type of the states has to be respected. A *configuration* is a maximal set of states which can be active together. Configurations are characterised by the following conditions:

- A configuration contains the root-state.
- A configuration contains an and-state, if and only if it contains all descendants of the and-state.
- A configuration contains an or-state, if and only if it contains exactly one descendant of the or-state.

For example, in the statechart of the traffic light system, each configuration has to contain the states trafficlightsystem, pedestrian, and car. No configuration contains *both* substates red and green of pedestrian.

To explain, what other information is contained in a status we have to look at the complete description of the traffic light system, cf. Figure 4.3 on page 52. In this figure labeled transitions between states have been added. The labels have the form $\mathsf{e}[\mathsf{c}]/\mathsf{a}$, each of the three parts is optional. The markings $d_1$:, etc., are not part of the labels, they are used only for referencing the transitions.

$\mathsf{a}$ is an *action expression* which is executed when taking a transition, this will be relevant when defining a step.

$\mathsf{e}$ is an *event expression* to trigger the transition. In the traffic light system only one event is used: push. In addition to simple events as used in this example, there are special events to indicate for example, whether a state has been entered. We consider a set of event expressions which contains these simple events and which is closed under the boolean connectives, i.e. if $\mathsf{e}_1$ and $\mathsf{e}_2$ are event expressions, then not $\mathsf{e}_1$, $\mathsf{e}_1$ and $\mathsf{e}_2$, and $\mathsf{e}_1$ or $\mathsf{e}_2$ are event expressions, too. A status describes also which events are existing currently.

$\mathsf{c}$ is a *condition expression* used to guard execution of a transition. In the traffic light system, four condition expressions are used: The boolean variable request, the negations of the boolean variables request and person_on_road, and the compound condition expression in(car.red) and request. in(car.red) is a special condition expression indicating whether the state car.red is active. Similar to event expressions, the set of condition expressions is closed under the boolean connectives. In addition to the boolean variables used in this example, it can be refered to variables of other types, e.g. the integers, in a statechart. In the statechart notation there are predefined predicates over such variables, this allows to write for example $\mathsf{x} > 0$ as a condition expression. A status describes the values of the variables of a statechart. It does not have to describe explicitly the value of in(car.red), this information can be derived already from the set of active states.

In summary, a status of a statechart describes:

- The set of states which are active in this status. This set has to be a configuration.
- The set of events which are existing in this status.
- The values of the variables.

### 4.1.2 Steps

There are two different kinds of steps to relate two statuses. On the one hand there are steps which can be contributed to the environment. They are used to provide stimuli to a statechart. This means that the set of existing events can be enlarged and the values of the variables can be changed. The set of active states does not change in such a step. These steps are not described by a statechart, hence they are not considered further here.

On the other hand, there are the steps which are described by a statechart. The construction of a step is based on the notion of *transitions*, which comes in various variants. The most important ones are that of a *full compound transition*, or *compound transition* for short, and that of *transition segments*.

Transition segments are labeled arrows between states. Compound transitions are specific sets of transition segments which can be taken in a step. Several compound transitions can be taken in one step, thereby it has to be ensured that leaving and entering states in a step results again in a configuration.

How compound transitions are composed from transition segments, which compound transitions contribute to a step and which are the effects of taking a compound transition is described in the following.

**Compound Transitions** A *transition segment* is a labeled arrow relating states. It can have several *source* and *target states*. For the sake of simplicity we assume here at first that a transition segment has one source and one target state. Furthermore, the full notation of statecharts provides so-called *connectors* which can occur as source and target of transition segments. These are omitted here, too. Both restrictions will be removed when defining the semantics formally.

Figure 4.3 shows the statechart of the traffic light system including the transition segments. The source and target states of the transition segments $t_1, \ldots, t_7$ should be clear. $d_1$, $d_2$, and $d_3$ are special transition segments. They are called *default transitions* and are drawn with a dangling end. For each or-state there is a default transition, it is used to mark one of its descendants. The or-state itself is considered as the source state of a default transition.

In Figure 4.3 the ordinary transition segments relate siblings in the hierarchy of states. This is not required in statecharts. It is allowed to draw arrows crossing the boundary of states, these are called *inter-level transitions*. Furthermore, the target of a default transition need not be a *direct* descendant of its source.

The target state of a transition segment indicates which states become active when taking the transition segment in a step. If the target is an or-state, a compound transition is continued by default transitions until basic-states are reached. If an and-state is a target state, then the compound transition must be continued below any of its substates. Note, that there are even more complicated situations due to the inter-level transitions.

The transition segments of a compound transition can be partitioned into sets with different roles. One set, called *initial compound transition*, consists of one non-default transition. This restriction to *one* non-default transition is due to the restriction to one source and target state of transition segments we consider here. The other sets are called *continuation compound transitions* and consist of default transitions only. Again, this restriction to default transitions is due to the restricted subset of statecharts we consider here.

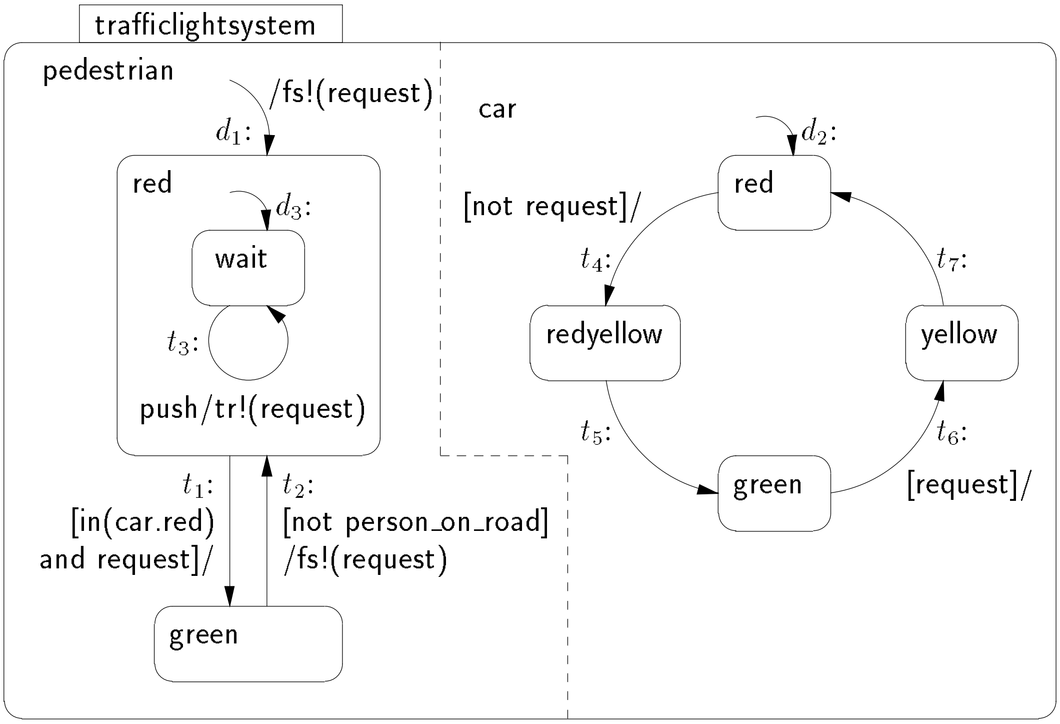



Figure 4.3: Traffic light system: Statecharts description

There is one compound transition which consists of continuation compound transitions only. It describes the initial set of active states, but is not used to relate two statuses.

In the traffic light example there are the following compound transitions:

- $\{d_1, d_2, d_3\}$, describing the initial set of active states
- $\{t_2, d_3\}$, consisting of the initial compound transition $\{t_2\}$ and the continuation one $\{d_3\}$
- the compound transitions $\{t_1\}, \{t_3\}, \{t_4\}, \{t_5\}, \{t_6\}, \{t_7\}$, each consisting of an initial compound transition only

Similar to a transition segment, a compound transition has source and target states. There is still one source state, namely that of the transition segment of the initial compound transition. The set of target states is the set of basic target states of the default transitions in the continuations. Restricted to the part of the statechart which is covered by a compound transition the target states of the compound transition have to form a configuration. The part of a statechart which is covered by a compound transition is determined by the transition segment which constitutes the initial compound transition as defined before. The part covered is that below and including the lowest common ancestor of the source and target state of this transition segment and which is an or-state. It is called the *scope* of the compound transition. In the traffic light system, the scope of the compound transitions $\{t_1\}$ and $\{t_2, d_3\}$ is the state pedestrian, of $\{t_3\}$ it is pedestrian.red, and of $\{t_4\}$, $\{t_5\}$, $\{t_6\}$, and $\{t_7\}$ it is car.

Compound transitions are the basic parts of steps. All transition segments of a compound transition contribute to the decision whether it can be taken. Similarly, all its transition segments contribute to the effect of taking the compound transition.

**Constructing a Step** A step consists of one or more compound transitions. Whether a compound transition can be part of a step depends on the compound transition itself as well as on which other compound transitions can be taken, too.

A compound transition is *enabled* at the status $s$, if its source state is active, the condition expressions of all its transition segments evaluate to true, and the event expressions of all its transition segments occurred. For event expressions $e_1$ and $e_2$ and not $e$ this means that $e_1$ and $e_2$ have to exist in the first case and that $e$ does not have to exist in the second case. The action expressions of the transition segments do not contribute to the decision whether a compound transition is part of a step. Only compound transitions enabled at $s$ can be part of a step from $s$ to another status $t$.

At second, there are compound transitions which cannot be taken together because the union of their basic target states cannot be part of a configuration. See for example the compound transitions $\{t_3\}$ and $\{t_1\}$ in the traffic light system. Both can be enabled at the same status. But their target states pedestrian.red.wait and pedestrian.green cannot be active together, this would imply that both substates of the or-state pedestrian are active. On the other hand, the compound transitions $\{t_3\}$ and $\{t_5\}$ can be taken together, it is possible that their target states pedestrian.red.wait and car.green are active in the same status.

If the scopes of two compound transitions are not related hierarchically, it is guaranteed, that their target states can be active in one configuration. We say, that two compound transitions are in *conflict*, if their scopes are the same or one is above the other in the hierarchy of states.

In statecharts, a *priority* on compound transitions is defined to determine which of two enabled and conflicting compound transitions can be part of a step. Here we consider only *structural priority*. A compound transition has higher priority than another one if its source state is above the source state of the other one in the hierarchy of states. Compound transitions with the same source state or with source states which are not related hierarchically cannot be compared w.r.t. this priority. In our example, $\{t_1\}$ has priority over $\{t_3\}$.

A compound transition is called *executable* at a status $s$ if it is enabled at $s$ and if there is no other enabled compound transition with a higher priority. A step consists of a maximal set, w.r.t. set inclusion, of executable compound transitions which are not in conflict with each other. Because not all conflicts can be resolved by priority, it is possible that there are several steps possible at a status $s$. One of them is chosen non-deterministically to relate the status $s$ with its successor.

That a step is a maximal set implies that executable compound transitions have to be taken, except if another executable compound transition with the same priority is taken. Let us assume that in our example the compound transitions $\{t_3\}$ and $\{t_5\}$ are executable at a status. Then both of them have to be taken in the next step. It is not possible to take only one of them.

**Taking a Step** Taking a step at a status $s$ means to construct the next status, i.e. a new set of active states, a new set of existing events, and to define the new values of the variables.

The new set of active states is constructed from the status $s$ and the compound transitions taken in the step. A state which is not below the scope of any compound transition remains active or inactive. Each active state below the scope of a compound transition becomes inactive and each state between the scope of a compound transition and a target state of it, including the target, becomes active. Instead of becoming active or inactive we say that a state is entered or left, respectively. The scope of a compound transition remains active. It is possible that a state is left and reentered in one step. This is the case for example when taking the compound transition $\{t_3\}$. Its scope is the state pedestrian.red. Taking the compound transition implies that the state pedestrian.red.wait is left because it is active and below the scope, and it is entered because it is a target state of the compound transition.

When a state $A$ is left or entered a corresponding event is generated implicitly. These are the events entered($A$) and exited($A$), written en($A$) and ex($A$) for short.

Other changes to the set of events and changes to the values of the variables are described by the action expressions of the transition segments. There are atomic action expressions to generate an event explicitly. This can be done for events as e.g. push in the traffic light system, but note that this kind of action expressions does not occur in this example. For an event $e$ this is written simply $/e$.

Other atomic action expressions which are of interest here are assignments $x := expr$, where $x$ is one of the variables and $expr$ is an expression over the variables. For a boolean variable $b$ there are the special forms fs!($b$) — make_false($b$) — and tr!($b$) — make_true($b$) — to assign a new value to $b$. Such action expressions are used in the traffic light example.

The execution of action expressions can be guarded by event expressions as well as by condition expressions. Let $a_1$ and $a_2$ be action expressions, $e$ be an event expression, and $c$ be a condition expression, then

if $c$ then $a_1$ else $a_2$ end if
when $e$ then $a_1$ else $a_2$ end when

are action expressions, too. In both cases the else part is optional.

Furthermore, it is possible to use for- as well as while-loops. To do so, a special kind of variables, so-called *context variables*, are needed. Context variables behave somehow differently from ordinary variables w.r.t. assigning several values to them in one step. Let $i$ be a context variable, $k$ and $l$ be integer expressions, $c$ be a condition expression, and $a$ be an action expression, then

for $i$ in $k$ to $l$ loop $a$ end loop
for $i$ in $k$ downto $l$ loop $a$ end loop
while $c$ loop $a$ end loop
break

are also action expressions. break is used to terminate the innermost nested loop. It is possible to write non-terminating action expressions. In the tool STATEMATE this is avoided by an upper bound on the number of iterations of a loop.

At last, it is also possible to compose two action expressions $a_1$ and $a_2$:

$a_1;a_2$

is an action expression, too. It means that both action expressions shall be executed in parallel.

The action expressions to be executed in a step from a status $s$ are executed in an arbitrary order. To keep the results independent of the order, the values of variables are always taken from $s$. Similarly, whether events are existing or states are active is checked w.r.t. $s$. This means for example that executing the action expression $x := x + 1; x := x + 1$ results in increasing x by one, but not by two.

Refering always to the "old" value means that these variables are not useful as a counter of a for-loop and that there is no use of a condition which refers to such variables only. The context variables differ from the ordinary ones in that they are updated immediately and that their most recent value is used. Context variables are not part of the status, they are used only as auxiliary variables within compound action expressions. After executing a step, their value is lost.

As for states becoming active or inactive, there are events which are generated implicitly when a variable $x$ is read, written, or changed. These are the events read($x$), written($x$), and changed($x$), with short forms rd($x$), wr($x$), ch($x$).

If a step from a status $s$ to a status $t$ is taken, all events existing in $s$ are discarded. Only those events will exist in $t$ which are generated explicitly by action expressions or implicitly due to accessing variables or due to states becoming active or inactive.

**Runs and Behaviours** A *run* is a sequence of statuses. The initial status is derived using the compound transition consisting of continuations only. In statecharts, initial values can be assigned to variables, these can be changed by actions of the default transitions. In the traffic light system, the initial value of request is false due to the action expression of the default transition $d_1$. The initial set of events is built as if this was an ordinary step.

Each subsequent status is derived from its predecessor by either a step of the environment as described at the beginning of Section 4.1.2 or by a step of the statechart. It is not required that steps of the statechart are taken if possible, further steps of the environment can be taken instead. In the formal semantics of statecharts defined in Section 4.8 we will exclude such runs by requiring fairness on steps of the statechart.

The *behaviour* of a statechart is the set of all runs of the statechart as defined before.

### 4.1.3 Further Concepts

We want to give here some short explanations on the concepts which have been omitted before.

**Connectors** *Connectors* can be used similar to states as source and target of transition segments. The connectors are used to link several transition segments together such that the connected transition segments correspond again to a — larger — transition segment. Therefore, an initial compound transition can consist of a set of transition segments and can have several source and target states. In this case it has to be ensured that the source and target states can become active together. Similarly, default transitions can be linked to further transition segments.

The use of connectors makes it more difficult to define which transition segments belong to a compound transition. But it does not change the view of a compound transition as a set of transition segments nor how a step is composed from compound transitions.

**History** A special kind of connectors are *history*- and *deep-history*-connectors. A history connectors allows to enter the last active substate of an or-state instead of using the default transition. A deep-history connector allows to enter the last active configuration below an or-state. The information on the last active substate or configuration below an or-state can be cleared by special action expressions.

In the context of this work history is not relevant, hence we will not consider it in the formal definition.

**Priority Numbers** In addition to structural priorities it is planned for future versions of the tool StateMate to assign *priority numbers* to transition segments. These can be used to resolve some of the conflicts between compound transitions with equal structural priority.

**Static Transitions** Because steps are maximal sets of executable, non-conflicting compound transitions, states have to be left whenever this is possible. In some descriptions it is necessary to model a non-deterministic choice between remaining in a state and leaving it. Remaining in a state can be described only to some limited extend by a compound transition from the state to itself, even when history is used to restore the active states below it. Actually, there is a difference between remaining in a state and leaving and reentering it in one step.

For this purpose we *extend* the notation of statecharts by so-called *static transitions*. These can be part of a step similar to a compound transition.

**Static Reactions** When executing a step, not only the action expressions of the transition segments are relevant. It is possible to associate action expressions with states. These are executed when entering, leaving, or remaining in a state, respectively. The last ones are called *static reactions*. Note, that these action expressions do not contribute to the construction of compound transitions from transition segments and of steps from compound transitions.

**Time** There are two constructs to refer to time in a statechart. It is possible to use a timeout mechanism and to schedule action expressions for execution in the future. Time is measured as number of steps taken. We do not consider time in the formal definition.

**Activities** Statecharts are only one of some interrelated notations. Another one is *activity-charts* which can be used to describe the functional decomposition of a system. The behaviour of the functional units, called *activities*, can be described by statecharts. In a statechart it is possible to sense whether activities are active and to influence this by some special action expressions. We concentrate at first on statecharts, activity-charts will be considered in more detail in Chapter 8.

### 4.1.4 Remarks on Describing with Statecharts

In this section we explain how the behaviour of the traffic light system has been described by a statechart and compare it to the TLA-description. At the end of the section we give some remarks on the use of events versus conditions and on whether statecharts are Moore or Mealy automata.

In Figure 4.3, page 52, the traffic light system is described by two parts running concurrently: The traffic lights of the pedestrians and of the cars. This is described by the and-state trafficlightsystem and its substates pedestrian and car.

The traffic light of the cars starts with showing the red light. By taking its compound transitions subsequently it performs a cycle through the usual phases of a traffic light. It switches from showing the red light to showing the red and yellow ones as soon as there is no request. Analogously, it switches from showing the green to showing the yellow light as soon as there is a request. Which lights are shown corresponds in an obvious way to the name of the state which is active.

In a similar way, the traffic light of the pedestrians switches between showing the red and the green light. It switches from red to green, if there is a request and the cars are shown the red light. It switches back to red if there is no person on the road. When taking this compound transition the request is considered as granted. This is described by the action expression fs!(request). In the initial status it is assumed that no request is pending, this is described by the action expression of the default transition $d_1$.

When the state pedestrian.red is active, requests are recognised by the system if the event push exists. In this case, the transition from the state wait to itself is taken with the effect of setting request to true. When the event push exists if request is already true, then request is set again to true.

In the case that request is true, the event push exists, and the state car.red is active, the compound transitions $\{t_3\}$ and $\{t_1\}$ are enabled. Due to structural priority the compound transition $\{t_1\}$ is taken.

If the event push exists in a status in which the state pedestrian.green is active, it is simply ignored. This is true even if at this status the transition segment $t_2$ is taken. Then the default transition $d_3$ is taken, too, but not the transition segment $t_3$.

Table 4.1 shows some sequences of statuses similar to the TLA-ones in Table 3.3, page 41. It is indicated in the table whether the event push exists in a status, which

| | a) | | | | | | | | | b) |
|---|---|---|---|---|---|---|---|---|---|---|
| push | | | | × | | | | | | |
| request | f | f | f | f | **t** | t | t | t | t | **f** |
| person_on_road | | | | | | | | **t** | **f** | |
| pedestrian | r | r | r | r | r | r | r | **g** | g | **r** |
| car | r | **ry** | **g** | g | g | **y** | **r** | r | r | r |
| trans. segments | | $t_4$ | $t_5$ | | $t_3$ | $t_6$ | $t_7$ | $t_1$ | | $t_2, d_3$ |

| | d) | | | | | | | e) | | | | | | | | f) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| push | | × | | | | | | | | | × | | | | | |
| request | f | f | **t** | t | t | t | t | **f** | f | f | f | **t** | t | t | t | **f** |
| person_on_road | | | | | | **t** | **f** | | | | | | | | **f** | |
| pedestrian | r | r | r | r | r | **g** | g | **r** | r | r | r | r | r | r | **g** | **r** |
| car | r | **ry** | **g** | **y** | **r** | r | r | r | **ry** | **g** | g | g | **y** | **r** | r | r |
| trans. segments | | $t_4$ | $t_3, t_5$ | $t_6$ | $t_7$ | $t_1$ | | $t_2, d_3$ | $t_4$ | $t_5$ | | $t_3$ | $t_6$ | $t_7$ | $t_1$ | $t_2, d_3$ |

Table 4.1: Sequences of statuses of the traffic light system.

are the values of the variables request and person_on_road, and which substates of pedestrian and car are active. Between the statuses it is shown which transitions segments are taken in the step.

The sequence a) – b) corresponds to the scenario used to describe the requirements informally. Because in the statechart there is nothing corresponding to the TLA-action *Release. button* no sequence corresponding to b) – c) in Table 3.3 is shown here. Note that the sequence d) – e) shown here differs from the corresponding one of the TLA-description. Between serving two requests the traffic on the road is allowed to flow actually. The sequence e) – f) corresponds to the situation that no person enters the road when the traffic light of the pedestrians switches to green. In this case the traffic light of the pedestrians switches back to red in the next step and the request is considered as served.

**Comparing the TLA- and the Statecharts-Description** The statecharts- and the TLA-description differ in several aspects from each other.

- Instead of the boolean variable *push* which is set and reset, respectively, by two TLA-actions, an event is used in the statecharts-description to describe that the button has been pushed. It is not necessary to describe explicitly that the button is released because this event no longer exists after taking the next step, except if it is generated again.

- Only one TLA-action can be taken in one step of the TLA-description. In the statecharts-description several non-conflicting compound transitions can be taken together. Remember that there are other styles to write TLA-descriptions which allow that several TLA-actions can be taken in one TLA-step.

- Which lights are shown by a traffic light is described in the TLA-description by the values of variables. In the statecharts-description this is described by the states which are active in a status. That only one of the different lightings is shown at one state is ensured in the TLA-description because a variable can have only one value at one state, in the statecharts-description this is ensured because at most one descendant of an or-state can be active.

**Events versus Conditions** An event describes that something has just happened. An event is available for the next-step of a statechart only. Events are volatile, i.e. independent of whether they are used in a step, they are no longer available after taking a step.

On the other hand, a condition is persistent: Its value is defined in every status of a statechart and it changes only when the value of some of the variables over which it is defined changes.

Events and conditions are not unrelated. That a condition changes its value can be seen as something happening. Corresponding events are generated implicitly in statecharts.

**Moore versus Mealy Automata** An automaton with output can be classified as a *Moore* or a *Mealy* automaton. The output of a Moore automaton depends on the active state, the output of a Mealy automaton depends on the active state as well as on the current input. We will argue that statecharts can be used to describe Moore as well as Mealy automata, even mixed in one statechart.

Note that it is not described by a statechart which part of its status has to be considered as output and which one as input.

Let us assume in the traffic light system that it is considered as an output which of the lights are shown. This information is determined in an obvious way by the active substates of the states pedestrian and car, hence this output depends solely on the set of active states. Therefore, the statechart can be considered as a Moore automaton.

Let us assume, too, that the variable request is an output of the statechart. Its value depends on the inputs — e.g. on existence of the event push and on the variable person_on_road — and on which state is active. Hence, this statechart can be seen as a Mealy automaton, too.

## 4.2 Constituents of a Statechart

In this section we define those sets, functions, and predicates which are necessary to describe a statechart. We do not cover here the full notation of statecharts. Only events, which can be generated explicitly and those corresponding to entering and exiting states are considered. We do not consider here variables, these will be considered in Chapter 5. This implies, that labels of transition segments are simplified essentially.

On the other hand we extend statecharts by defining *static transitions* in addition to transition segments.

A statechart is defined by the sets

| | |
|---|---|
| $S$ | (states) |
| $C$ | (connectors) |
| $E$ | (events) |
| $T$ | (transition segments) |
| $SR$ | (static reactions) |

and the following predicates and functions

| | |
|---|---|
| *substate* | (hierarchy of states) |
| *type* | (type of states and connectors) |
| *source*, *target* | (source and target of transition segments) |
| *trigger*, *events* | (trigger and action of transition segments and static reactions) |
| *priority* | (priority number of transition segments) |
| *static. reactions* | (static reactions of states) |
| *entering. events* | (actions taken when entering states) |
| *exiting. events* | (actions taken when leaving states) |

We assume that the sets $S, C, E, T$, and $SR$ are finite and pairwise disjoint.

In the following we present these sets, functions, and predicates in more detail. Furthermore we define additional functions and predicates from the ones stated before.

### 4.2.1 States and Connectors

Each state is of one of the types $\mathsf{and}$, $\mathsf{or}$, and $\mathsf{basic}$. Each connector is of one of the types $\mathsf{and}$ and $\mathsf{or}$. The type of states and connectors is denoted by the function *type*.

$$type\colon S \cup C \to \{\mathsf{and}, \mathsf{or}, \mathsf{basic}\}$$

We require that

$$\bigwedge_{c \in C} type(c) \neq \mathsf{basic}$$

To shorten definitions we introduce the predicates *and*, *or*, and *basic* which are true if and only if a state or connector is of the corresponding type:

$$and\colon S \cup C$$
$$and(s) \equiv type(s) = \mathsf{and}$$

$$or\colon S \cup C$$
$$or(s) \equiv type(s) = \mathsf{or}$$

$$basic\colon S \cup C$$
$$basic(s) \equiv type(s) = \mathsf{basic}$$

In the literature on statecharts the type of the connectors are distinguished even further. There are two subtypes of and-connectors, namely *join-* and *fork*-connectors and there are four subtypes of or-connectors, namely *condition-*, *selection-*, *junction-*, and *diagram*-connectors. The different subtypes are intended to be used for different purposes, but their meaning is the same. Therefore we continue to use only the types **and** and **or**. In a statechart, and-connectors are depicted by a filled circle $\bullet$ and or-connectors by an unfilled one $\circ$. Remember that we do not consider history-connectors.

The following notions involving hierarchy are relevant for states only.

The hierarchical relation of states is denoted by the predicate *substate*. The transitive closure of this relation is denoted by $substate^+$. The transitive and reflexive closure of *substate* is denoted $substate^*$.

$$\begin{array}{ll} substate\colon & S \times S \\ substate^+\colon & S \times S \\ substate^*\colon & S \times S \end{array}$$

where $substate(s_1, s_2)$ is true if and only if $s_1$ is a substate of $s_2$. For a syntactically correct statechart, the relation expressed by *substate* imposes a tree-like structure on the set of states. The root of this tree is denoted by the predicate *root* on the set of states:

$$root\colon S$$

The sets of direct, proper, and all, including the state itself, descendants of a state are denoted by the functions *descendants*, $descendants^+$, and $descendants^*$, respectively.

$$\begin{array}{ll} descendants\colon & S \to 2^S \\ descendants^+\colon & S \to 2^S \\ descendants^*\colon & S \to 2^S \end{array}$$

Conversely, the sets of proper and of all ancestors of a state are denoted by the functions $ancestors^+$ and $ancestors^*$, respectively.

$$\begin{array}{ll} ancestors^+\colon & S \to 2^S \\ ancestors^*\colon & S \to 2^S \end{array}$$

Each of these additional functions and predicates and the property that the states form a tree-like structure can be defined quite easily from the predicate *substate*. Their definitions have been omitted therefore.

The sets of all and of the proper common ancestors of a non-empty set of states are denoted by the functions $ca^*$ and $ca^+$, respectively. Note that $ca^*(S_1)$ contains at least the root-state if $S_1$ is non-empty.

$$ca^*\colon 2^S \setminus \emptyset \to 2^S$$
$$ca^*(S_1) \equiv \bigcap_{s \in S_1} ancestors^*(s)$$

$$ca^+\colon 2^S \setminus \emptyset \to 2^S$$
$$ca^+(S_1) \equiv ca^*(S_1) \setminus S_1$$

The *lowest common ancestor* of a non-empty set of states $S_1 \subseteq S$ is the lowest state in the hierarchy which is above all states in $S_1$. This is denoted by the function *lca*.

$$\begin{array}{l} lca\colon 2^S \setminus \emptyset \to S \\ lca(S_1) = s \text{ iff } \wedge\ s \in ca^*(S_1) \\ \qquad\qquad\qquad \wedge \displaystyle\bigwedge_{s_1 \in ca^*(S_1)} substate^*(s, s_1) \end{array}$$

Two states are called *orthogonal* if their lowest common ancestor is an and-state and if they are not related hierarchically. An and-state is considered to be orthogonal to itself.

$$\begin{array}{l} orthogonal\colon S \times S \\ orthogonal(s_1, s_2) \equiv \wedge\ and(lca(\{s_1, s_2\})) \\ \qquad\qquad\qquad\qquad\quad \wedge\ \neg\, substate^+(s_1, s_2) \\ \qquad\qquad\qquad\qquad\quad \wedge\ \neg\, substate^+(s_2, s_1) \end{array}$$

The type of the states and their hierarchy of a syntactically correct statechart are related by the following properties. The basic-states are the leaves of the tree, i.e. they do not have descendants. The root-state has to be an or-state, and the direct descendants of and-states cannot be and-states again. We will often draw statecharts which do not satisfy the last two conditions. For example the statechart of the traffic light system has an and-state as its root. This does not cause problems because it is always possible to enclose such and-states into a further or-state without changing the behaviour of the statechart.

$$\wedge \bigwedge_{s \in S} \begin{array}{l} basic(s) \\ \Leftrightarrow descendants(s) = \emptyset \end{array}$$

$$\wedge \bigwedge_{s \in S} \begin{array}{l} root(s) \\ \Rightarrow or(s) \end{array}$$

$$\wedge \bigwedge_{s_1, s_2 \in S} \begin{array}{l} \wedge\ and(s_1) \\ \wedge\ s_2 \in descendants(s_1) \\ \Rightarrow \neg\, and(s_2) \end{array}$$

A *configuration* is a set of states which contains the root-state and which is closed under the properties that it contains exactly one descendant of each or-state in it and all descendants of each and-state in it.

$$\begin{array}{l} configuration\colon 2^S \\ configuration(S_1) \equiv \wedge \displaystyle\bigvee_{s \in S_1} root(s) \\ \qquad\qquad\qquad\qquad\quad \wedge \displaystyle\bigwedge_{s_1 \in S_1} \begin{array}{l} or(s_1) \\ \Rightarrow |\ descendants(s_1) \cap S_1 \ | = 1 \end{array} \\ \qquad\qquad\qquad\qquad\quad \wedge \displaystyle\bigwedge_{s_2 \in S} \begin{array}{l} and(s_2) \\ \Rightarrow descendants(s_2) \subseteq S_1 \end{array} \end{array}$$

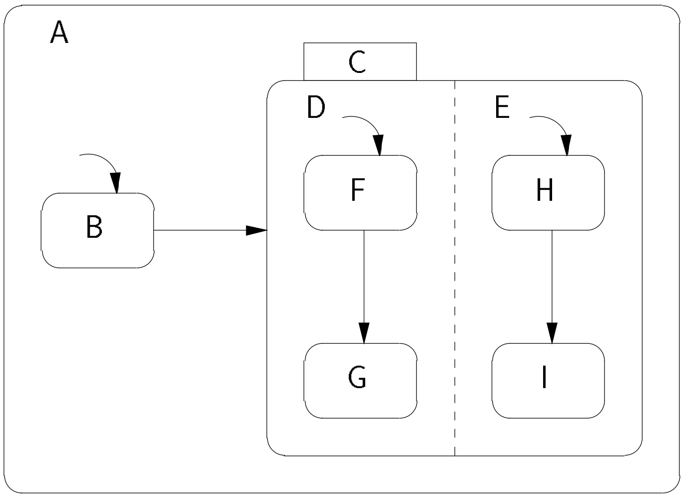



Figure 4.4: Example of a statechart

A configuration is determined uniquely by the basic-states it contains. Note that not any set of basic states can be extended to a configuration.

In the statechart in Figure 4.4 the set of states is $S = \{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}, \mathsf{E}, \mathsf{F}, \mathsf{G}, \mathsf{H}, \mathsf{I}\}$. The relation *substate* corresponds to inclusion of the boxes in the diagram. A is the root-state. B, F, G, H, and I are basic-states. A, D, and E are or-states and C is an and-state. Remember that this is expressed by the dashed line separating it. The lowest common ancestor of the states H and I is E, of H and F it is C, and of H and B it is A. The states H and F are orthogonal, whereas the pairs of states H, I and H, E are not. Two possible configurations of the statechart are $\{\mathsf{A}, \mathsf{B}\}$ and $\{\mathsf{A}, \mathsf{C}, \mathsf{D}, \mathsf{E}, \mathsf{G}, \mathsf{I}\}$. These are determined uniquely by the sets of basic-states $\{\mathsf{B}\}$ and $\{\mathsf{G}, \mathsf{I}\}$, respectively. In contrast, the set of basic states $\{\mathsf{B}, \mathsf{C}\}$ cannot be extended to a configuration.

### 4.2.2 Events

The set of events $E$ is partitioned into a set $E_{impl}$ of *implicit* and $E_{expl}$ of *explicit* events. The explicit events are those which can be generated by executing an action expression. The implicit events have to be generated according to changes of the status in a step. In our restricted context this is the set $E_{conf}$ of *configuration* events:

$$E_{conf} = \{\mathsf{en}(s), \mathsf{ex}(s) \mid s \in S\}$$
$$E_{impl} = E_{conf}$$

Further implicit events will be introduced in the Chapters 5 and 8.

Labels of transition segments can contain more complex expressions over events than a single event. The set $\mathcal{E}(E)$ of event expressions over $E$ is the smallest set, such that

- $E \subseteq \mathcal{E}(E)$
- if $\mathsf{e} \in \mathcal{E}(E)$, then $\mathsf{not}\ \mathsf{e} \in \mathcal{E}(E)$
- if $\mathsf{e}_1, \mathsf{e}_2 \in \mathcal{E}(E)$, then $\mathsf{e}_1$ $\mathsf{and}$ $\mathsf{e}_2$, $\mathsf{e}_1$ $\mathsf{or}$ $\mathsf{e}_2 \in \mathcal{E}(E)$
- if $\mathsf{e} \in \mathcal{E}(E)$, then $(\mathsf{e}) \in \mathcal{E}(E)$

That is, the set of event expressions is closed under the boolean operators and parentheses can be used as usual.

### 4.2.3 Transition Segments

The set $T$ of *transition segments* is partitioned into the set $DT$ of *default transitions*, $ST$ of static transitions, and $TS$ of ordinary transition segments.

Each transition segment can have several source and target states and connectors. This will be restricted for each kind of transition segments. Clearly, at least one source and target state or connector has to exist. These sets are denoted by the functions *source* and *target*:

$$\begin{aligned} &source\colon T \to 2^{S\cup C} \\ &target\colon T \to 2^{S\cup C} \end{aligned}$$

with

$$\bigwedge_{t\in T} \begin{array}{l} \wedge\ source(t) \neq \emptyset \\ \wedge\ target(t) \neq \emptyset \end{array}$$

It is required in statecharts that the source and the target states, respectively, of a transition segment can be part of a configuration. This is guaranteed, if they are pairwise orthogonal:

$$\bigwedge_{t\in T} \begin{array}{l} \wedge \displaystyle\bigwedge_{s_1,s_2\in source(t)\cap S} orthogonal(s_1, s_2) \\ \wedge \displaystyle\bigwedge_{s_3,s_4\in target(t)\cap S} orthogonal(s_3, s_4) \end{array}$$

The condition expressed formally is stronger than the informal one. But it is the one which is actually required in statecharts. Neither of the transition segments in Figure 4.5 satisfies this condition. The source states of the transition segment $t_1$, namely $\mathsf{B}$ and $\mathsf{E}$, can be part of one configuration, but they are not orthogonal as required by the formal definition.

A *default transition* $d$ has to have exactly one source state which has to be an or-state. It does not need to have target *states*, but if it has ones, these must be proper

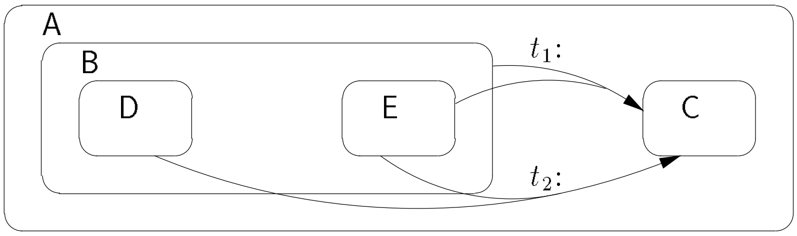



Figure 4.5: Transition segments with non-orthogonal source states

descendants of the source state:

$$\bigwedge_{d \in DT} \bigvee_{s \in S} \begin{array}{l} \wedge\ \{s\} = source(d) \\ \wedge\ or(s) \\ \wedge \displaystyle\bigwedge_{s_1 \in target(d) \cap S} s_1 \in descendants^+(s) \end{array}$$

Furthermore, it is required that each or-state has exactly one default transition:

$$\begin{array}{l} \wedge \displaystyle\bigwedge_{d_1, d_2 \in DT} \begin{array}{l} d_1 \neq d_2 \\ \Rightarrow source(d_1) \neq source(d_2) \end{array} \\ \wedge \displaystyle\bigwedge_{s \in S} \begin{array}{l} or(s) \\ \Rightarrow \displaystyle\bigvee_{d \in DT} source(d) = s \end{array} \end{array}$$

We consider *static transitions* as a special kind of transition segments from a state to itself. Hence we have

$$\bigwedge_{t \in ST} \bigvee_{s \in S} \begin{array}{l} \wedge\ source(t) = \{s\} \\ \wedge\ target(t) = \{s\} \end{array}$$

Each state can have an arbitrary number of static transitions. In a statechart we will indicate them by the keyword **st** inside the box of the state corresponding to its source state, followed by its label.

An ordinary transition segment cannot have the root state as its source or target. This becomes relevant when defining the scope of compound transitions, cf. Sections 4.3.3 and 4.3.4.

$$\bigwedge_{t \in TS} \bigwedge_{s \in S} \begin{array}{l} s \in source(t) \cup target(t) \\ \Rightarrow \neg\, root(s) \end{array}$$

Each transition segment is labeled by an event expression, its *trigger*, and a set of explicit events, its *action*. Hence the atomic action expressions we consider here express that an event is generated. Compound action expressions can be build only

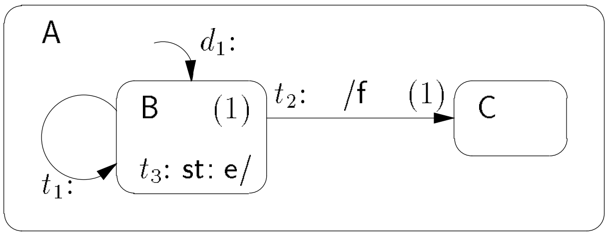



Figure 4.6: Static transition

by using the operator ";". The trigger and the action of a transition segment are denoted by

$$\begin{aligned}&\mathit{trigger}: T \to \mathcal{E}(E) \cup \{\perp\}\\ &\mathit{events}: T \to 2^{E_{expl}}\end{aligned}$$

When a transition segment does not have a trigger, then $\mathit{trigger}(t)$ denotes a bottom element $\perp$, with $\perp \notin E$. A transition segment $t$ with $\mathit{trigger}(t) = \mathsf{e}$ and $\mathit{events}(t) = \{\mathsf{f}_1, \mathsf{f}_2\}$ is labeled in a statechart with $\mathsf{e}/\mathsf{f}_1;\mathsf{f}_2$. If the trigger is empty it is written $/\mathsf{f}_1;\mathsf{f}_2$, if the action is empty it is written $\mathsf{e}/$. If there is neither a trigger nor an action, no label is written at all.

The *priority number* of transition segments is denoted

$$\mathit{priority}: T \to \mathbb{N}$$

The priority number is written in parentheses near the end of the transition segment in the statechart. It is written inside the box of the corresponding state if the transition segment is a static transition. If no priority number is given in a statechart for a transition segment $t$, we assume by default that $\mathit{priority}(t) = 0$ for static transitions and $\mathit{priority}(t) = 1$ for default transitions and ordinary transition segments.

The statechart in Figure 4.6 contains the transition segments $d_1$, $t_1$, $t_2$, and $t_3$, where $d_1$ is a default transition and $t_3$ is a static transition. $t_1$ and $t_2$ are ordinary transition segments. $t_3$ has the event expression consisting of the event $\mathsf{e}$ as trigger and $t_2$ has the action expression $\mathsf{f}$. The priority of each transition segment is 1, for $t_2$ and $t_3$ it is given explicitly, for $d_1$ and $t_1$ by the default.

### 4.2.4 Static Reactions

With each state of a statechart *static reactions* can be associated. $SR$ denotes the set of all static reactions of a statechart. A static reaction is similar to a transition segment because it has a trigger and an action. But it does not have source and target states. The static reactions of a state are denoted by

$$\mathit{static.reactions}: S \to 2^{SR}$$

The domain of the functions *trigger* and *events* is extended to $T \cup SR$:

$$trigger\colon T \cup SR \rightarrow \mathcal{E}(E) \cup \{\perp\}$$
$$events\colon T \cup SR \rightarrow 2^{E_{expl}}$$

The actions taken when entering or exiting a state are actions similar to that of static reactions, but they do not have a trigger. They are denoted by functions from states to the set of events.

$$entering.\ events\colon S \rightarrow 2^{E_{expl}}$$
$$exiting.\ events\colon S \rightarrow 2^{E_{expl}}$$

Neither these actions nor the static reactions are assigned priority numbers.

We write the trigger and action of static reactions similar to static transitions within the box of the corresponding state. They are preceded by the keyword sr. Analogously, we write the actions when entering or exiting a state within its box, preceded by the keywords en and ex, respectively.

## 4.3 Compound Transitions

Due to the presence of connectors and because transition segments can have several sources and targets it becomes more complicated to define which transition segments are part of one compound transition. Nevertheless, a compound transition consists of an initial compound transition and several continuation compound transitions. In the following we define at first initial compound transitions, at second we define continuation ones. To do so we consider only ordinary transition segments and default transitions. At last we relate compound and static transitions.

Compound transitions are the basic parts to build steps. Although in the following definitions we view sets of transition segments as directed graphs, it is sufficient in the subsequent sections to consider a compound transition as a set of transition segments.

For a set $T_1$ of transition segments, the source and target states of $T_1$ are defined simply as the union of all source and target states, respectively, of the transition segments. We overload the function symbols *source* and *target*:

$$source\colon 2^T \rightarrow 2^S$$
$$source(T_1) \equiv \bigcup_{t \in T_1} source(t) \cap S$$

$$target\colon 2^T \rightarrow 2^S$$
$$target(T_1) \equiv \bigcup_{t \in T_1} target(t) \cap S$$

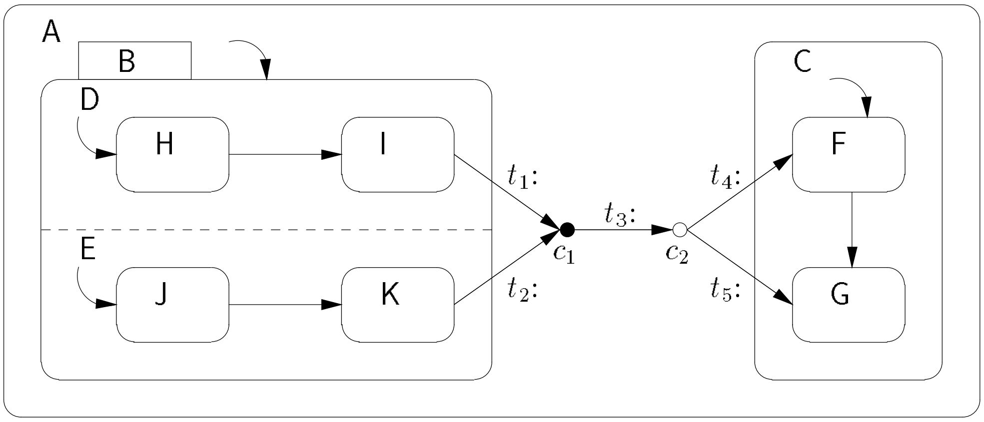



Figure 4.7: Building initial compound transitions

The *scope* of a non-empty set of transition segments is the lowest or-state which is a proper common ancestor of all states of the transition segments. Note that this definition is well-defined only if none of these states is the root-state.

$$
\begin{aligned}
&\mathit{scope}\colon 2^T \setminus \emptyset \to S\\
&\mathit{scope}(T_1) = s \text{ iff } \wedge\ \mathit{or}(s)\\
&\qquad\qquad\qquad\quad \wedge\ s \in \mathit{ca}^+(\mathit{source}(T_1) \cup \mathit{target}(T_1))\\
&\qquad\qquad\qquad\quad \wedge \bigwedge_{s_1 \in \mathit{ca}^+(\mathit{source}(T_1) \cup \mathit{target}(T_1))} \mathit{substate}^+(s_1, s) \Rightarrow \mathit{and}(s_1)
\end{aligned}
$$

## 4.3.1 Initial Compound Transitions

In Section 4.1 an *initial compound transition* consisted of a single transition segment with one source and one target state. The effect on the set of states of taking a step was restricted to the states below the scope of the compound transition. Remember that the scope was already determined by the transition segment of the initial compound transition. This should remain valid if an initial compound transition consists of one transition segment with more than one source and target state. It is possible to use such an initial compound transition to relate configurations because its source and target states, respectively, have to be pairwise orthogonal.

When using a set of transition segments linked together by connectors as an initial compound transition, it should be possible to replace it by a single transition segment. This can be explained by the statechart in Figure 4.7. It contains the and-connector $c_1$ and the or-connector $c_2$. To construct an initial compound transition let us start with the transition segment $t_1$. Its target is the *and*-connector $c_1$. Therefore each transition segment having $c_1$ as a source or target must be part of this initial compound transition. These are the transition segments $t_1$, $t_2$, and $t_3$. $t_3$ itself has the *or*-connector $c_2$ as its target. Therefore the initial compound transition has to contain exactly one transition segment with target $c_2$ and one with source $c_2$. $t_3$ is the only one with target $c_2$ and is already contained in the initial compound transition. $t_4$ and $t_5$ have $c_2$ as source, exactly one of them is contained in the initial compound transition. Hence, from the transition segments $t_1, \ldots, t_5$ two initial compound transitions can be built: $\{t_1, t_2, t_3, t_4\}$ and $\{t_1, t_2, t_3, t_5\}$. Note that each of them can be be replaced by a single transition segment.

A set $T_1 \subseteq T$ is an *initial compound transition* if

- $T_1$ is not empty and consists of ordinary transition segments only.
- The type of the connectors is respected: That is, let $t \in T_1$ be a transition segment and $c \in source(t) \cup target(t)$ be a connector. If $c$ is an and-connector, then $T_1$ contains all transition segments with source or target $c$. If $c$ is an or-connector, then $T_1$ contains exactly one transition segment with source $c$ and exactly one with target $t$.
- $T_1$ defines a graph $(V, E)$, where $V$ is the set of all sources and targets of transition segments in $T_1$ and $(v_1, v_2) \in E$ is an edge if there is a transition segment $t \in T_1$ with $v_1 \in source(t)$ and $v_2 \in target(t)$.

  Each maximal path of this graph has to start and end in a state, as intermediate vertices only connectors are allowed. This means that for each connector $c \in V$ there must be states $s_1, s_2 \in V$ such that there is a path from $s_1$ to $c$ and from $c$ to $s_2$. Furthermore, if there is a path from a state $s_1$ to a state $s_2$ without an intermediate state $s_3$, then this path is either a cycle, i.e. $s_1 = s_2$, or the path cannot be extended, i.e. there is no $t \in T_1$ with $s_2 \in source(t)$.

  This allows to speak of the source and target states of $T_1$.
- As for a single transition segment, the source and target states, respectively, of $T_1$ must be pairwise orthogonal.
- $T_1$ must be a minimal set satisfying these conditions.

Minimality of $T_1$ implies on the one hand that the graph of $T_1$ does not contain cyclic paths consisting of connectors only and on the other hand that $T_1$ does not contain a transition segment which is not required by respecting types of connectors. A relevant case is depicted by the statechart in Figure 4.8. The set of transition segments $\{t_1, t_2\}$ satisfies all conditions except minimality. But in statecharts this is considered as two different initial compound transitions, namely $\{t_1\}$ and $\{t_2\}$.

The conditions stated before can be expressed formally, cf. [Deiß, 1996]. Note that neither the triggering events nor the priority numbers are used in this construction.

### 4.3.2 Continuation Compound Transitions

An initial compound transition has to be extended until a set of basic-states is reached which can be part of a configuration. This extension is restricted to the

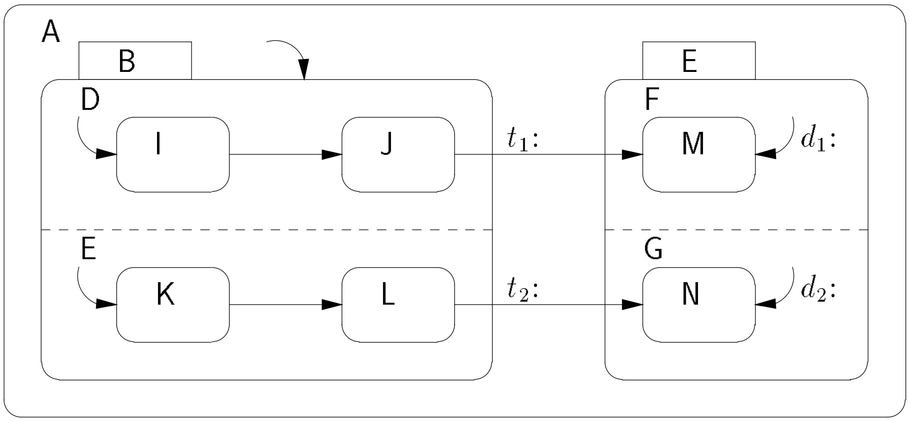



Figure 4.8: Two initial compound transitions

part of the statechart determined by the scope of the initial compound transition. It has to be defined

- which states have to be continued towards a set of basic-states, and
- how states are continued towards a set of basic-states.

A *continuation compound transition* is a set $T_1 \subseteq T$ of transition segments which is defined similar to an initial compound transition. It should be possible to replace it by a single transition segment. In this case this single transition segment would be a default transition. The differences to initial compound transitions are that

- $T_1$ contains exactly one default transition.
- The target states of $T_1$ are strictly below its source state. Note that this is the source state of the default transition.

It is not sufficient to continue only the target states of an initial compound transition. For example, in Figure 4.8 the substate G of the and-state E has to be continued, too, w.r.t. the initial compound transition $\{t_1\}$. In this case this is done by the continuation $\{d_2\}$.

An initial compound transition $T_0$ can be continued by a set $\mathcal{T} = \{T_1, \ldots, T_n\}$ of continuation compound transitions if $T_1, \ldots, T_n$ satisfy the following conditions:

1. Any non-basic target state $s$ of any of the $T_i, 0 \leq i \leq n$, is continued. This means, that if $s$ is an or-state then there is a continuation in $\mathcal{T}$ with source $s$. If $s$ is an and-state, then for any direct descendant $s_j$ of $s$ there is a continuation in $\mathcal{T}$ with source $s_j$.
2. If $s$ is an and-state below the scope of $T_0$ and above a target of any of the $T_i, 0 \leq i \leq n$, then each of the direct descendants of $s$ is a target or is above a target of a $T_j, 0 \leq j \leq n$.

3. $\mathcal{T}$ is minimal.

The first condition expresses that $\mathcal{T}$ contains enough continuations to reach actually basic-states. The second property ensures that basic-states below all or none of the direct descendants of an and-state are reached. The third property excludes superfluous continuations, especially there cannot be two continuations with the same or-state as their source.

That a set of basic-states is reached which can be part of a configuration is guaranteed by the conditions above and because the target states of each of the $T_i, 0 \leq i \leq n$, have been required to be pairwise orthogonal.

### 4.3.3 Full Compound Transitions

Let $T_i, 0 \leq i \leq n$, be sets of transitions segments such that $T_0$ is an initial compound transition which is continued by $\{T_i \mid 1 \leq i \leq n\}$ as defined before. Then $\bigcup_{0 \leq i \leq n} T_i$ is called a *full compound transition*. By $FCT$ we denote the set of full compound transitions of a statechart.

Let $\widetilde{T}$ be a a non-empty full compound transition with initial compound transition $T_0$, then the source states of $\widetilde{T}$ are the source states of $T_0$. The target states of $\widetilde{T}$ are the basic target states of $\widetilde{T}$.

$$source\colon FCT \setminus \emptyset \rightarrow 2^S$$
$$source(\widetilde{T}) \equiv source(T_0)$$

$$target\colon FCT \setminus \emptyset \rightarrow 2^S$$
$$target(\widetilde{T}) \equiv \bigcup_{t \in \widetilde{T}} \{s \mid s \in target(t) \wedge basic(s)\}$$

The scope of $\widetilde{T}$ is the scope of $T_0$. This definition is well-defined because $T_0$ consists of ordinary transition segments only. As a consequence, at least the root-state is a *proper* common ancestor of the source and target states of $T_0$.

$$scope\colon FCT \setminus \emptyset \rightarrow S$$
$$scope(\widetilde{T}) \equiv scope(T_0)$$

### 4.3.4 Further Compound Transitions

**Compound Transition for Initialisation** Initialising a statechart can be considered as a full compound transition, too. Clearly, this one does not have an initial compound transition. It consists of continuation compound transitions only. One of them has the root-state as its source. The set of these compound transitions is denoted by $ICT$.

**Static Transitions as Compound Transitions** We consider each static transition $t \in ST$ as a compound transition, too. Hence, we extend $CT$ as defined before by the singletons $\{t\}$, for $t \in ST$. These are not continued towards basic-states. The *static compound transitions* are characterised by the following predicate.

$$\begin{array}{l} static.compound\colon CT \\ static.compound(\widetilde{T}) \equiv \bigvee_{t \in ST} \widetilde{T} = \{t\} \end{array}$$

The set of static compound transitions is denoted by $SCT$.

The scope of a static compound transition is the lowest or-state above its source state. An exceptional case is that the source state is already the root. In this case we take the root as the scope.

$$\begin{array}{l} scope\colon SCT \to S \\ scope(\{t\}) = s \text{ iff } \wedge\ root(source(t)) \ \Rightarrow\ s = source(t) \\ \qquad\qquad\qquad\qquad\ \wedge\ \neg\, root(source(t)) \ \Rightarrow\ s = scope(t) \end{array}$$

Note that the scope of a static compound transition of a state $s$ is the same as that of a compound transition containing a transition segment from this state to itself, namely the lowest or-state above $s$.

The set of all *compound transitions* is the set $CT$:

$$CT = FCT \cup ICT \cup SCT$$

Note that the sets $FCT$, $ICT$, and $SCT$ are pairwise disjoint.

## 4.4 Enabled Compound Transitions and Static Reactions

The trigger of a transition segment is an event expression. The predicate *exists* denotes whether an event expression is satisfied by a set of events. *exists* is defined recursively over the structure of event expressions:

$$exists\colon \mathcal{E}(E) \cup \{\bot\} \times 2^E$$

$$exists(\mathsf{e}, E_1) \equiv \begin{cases} true & \text{if } \mathsf{e} = \bot \\ \mathsf{e} \in E_1 & \text{if } \mathsf{e} \in E \\ exists(\mathsf{e}_1, E_1) \wedge exists(\mathsf{e}_2, E_1) & \text{if } \mathsf{e} = \mathsf{e}_1 \text{ and } \mathsf{e}_2 \\ exists(\mathsf{e}_1, E_1) \vee exists(\mathsf{e}_2, E_1) & \text{if } \mathsf{e} = \mathsf{e}_1 \text{ or } \mathsf{e}_2 \\ \neg\, exists(\mathsf{e}_1, E_1) & \text{if } \mathsf{e} = \text{not } \mathsf{e}_1 \\ exists(\mathsf{e}_1, E_1) & \text{if } \mathsf{e} = (\mathsf{e}_1) \end{cases}$$

A compound transition $T_1$ is called *enabled* at a set of states $S_1$ and a set of events $E_1$ if all source states of $T_1$ are in $S_1$ and all triggers of $T_1$ are "existing" in $E_1$. More precisely, this is valid only for full and static compound transitions. When initialising a statechart the set of active states is empty. The corresponding compound transitions are used only in this situation.

$$
\begin{array}{l}
\mathit{enabled.transition}\colon CT \times 2^{S} \times 2^{E} \\
\mathit{enabled.transition}(T_1, S_1, E_1) \equiv \wedge \vee \wedge\ \mathit{source}(T_1) \subseteq S_1 \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \wedge\ T_1 \in FCT \cup SCT \\
\qquad\qquad\qquad\qquad\qquad\qquad \vee \wedge\ S_1 = \emptyset \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \wedge\ T_1 \in ICT \\
\qquad\qquad\qquad\qquad\qquad\quad \wedge \bigwedge_{t \in T_1} \mathit{exists}(\mathit{trigger}(t), E_1)
\end{array}
$$

A static reaction $r$ is called *enabled* at a set of states $S_1$ and a set of events $E_1$ and w.r.t. a set of compound transitions $\mathcal{T}$ if the state $s$ to which $r$ belongs is in $S_1$, if the trigger of $r$ is satisfied by $E_1$, and if $s$ is not strictly below the scope of any compound transition in $\mathcal{T}$.

$$
\begin{array}{l}
\mathit{enabled.static.reaction}\colon SR \times 2^{S} \times 2^{E} \times 2^{CT} \\
\mathit{enabled.static.reaction}(r, S_1, E_1, \mathcal{T}) \\
\equiv \bigvee_{s \in S} \wedge\ r \in \mathit{static.reactions}(s) \\
\qquad\quad \wedge\ s \in S_1 \\
\qquad\quad \wedge\ \mathit{exists}(\mathit{trigger}(r), E_1) \\
\qquad\quad \wedge \bigwedge_{T_1 \in \mathcal{T} \cap FCT} s \notin \mathit{descendants}^{+}(\mathit{scope}(T_1))
\end{array}
$$

## 4.5 Conflicts between Compound Transitions

Roughly, two compound transitions can be part of the same step if they cover different parts of the states of the statechart. This part is determined by the scope of the compound transitions. A static compound transition and an arbitrary compound transition can be part of the same step if the static one has the strictly higher scope. Two compound transitions are said to be *conflicting* if they do not satisfy these conditions. Formally:

$$
\begin{array}{l}
\mathit{conflicting.transitions}\colon CT \times CT \\
\mathit{conflicting.transitions}(T_1, T_2) \\
\equiv \vee\ \mathit{scope}(T_1) = \mathit{scope}(T_2) \\
\quad \vee \wedge\ \neg\, \mathit{static.compound}(T_2) \\
\qquad \wedge\ \mathit{substate}^{+}(\mathit{scope}(T_1), \mathit{scope}(T_2)) \\
\quad \vee \wedge\ \neg\, \mathit{static.compound}(T_1) \\
\qquad \wedge\ \mathit{substate}^{+}(\mathit{scope}(T_2), \mathit{scope}(T_1))
\end{array}
$$

Some conflicts between compound transitions can be resolved in statecharts by defining priorities on compound transitions. In the following we define *structural priority*

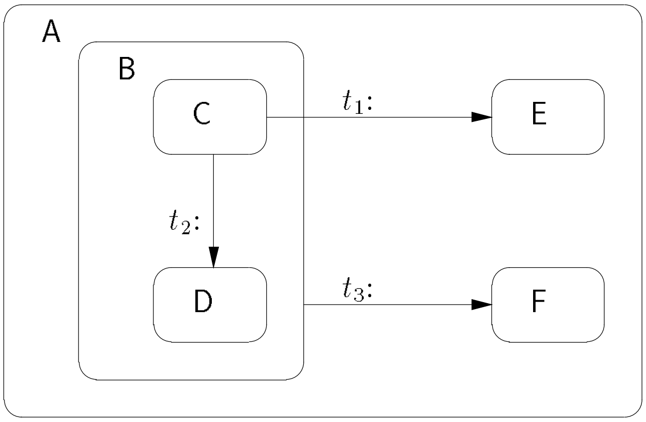



Figure 4.9: Structural priority determinant, [Harel and Naamad, 1996, Figure 28]

based on the hierarchy of states and *explicit priority* based on the priority numbers of the transition segments. Structural priority has higher priority than explicit one. Note that by both kinds of priorities it remains possible that some conflicting compound transitions cannot be compared w.r.t. their priorities.

### 4.5.1 Structural Priority

The definition of the *structural priority* of a compound transition is based on a distinguished state, its so-called *structural priority determinant*. Up to now this was defined in most publications on statecharts and also implemented in the tool STATEMATE [Harel *et al.*, 1990] as the scope of the transition. In [Harel and Naamad, 1996] it is announced, that in the future the source states of a compound transition will be used to define the structural priority determinant. In the work presented here we already use this newer definition.

When defining the structural priority determinant two cases are distinguished. These cases depend on whether a substate of an and-state below the scope is a source state of the compound transition or not. If there is such a state, the highest and-state with this property is the determinant. If this is not the case, then the set of source states of the transition is a singleton because the source states have to be pairwise orthogonal. In this case the determinant is defined as this source state.

As an example consider the statecharts in Figures 4.9 and 4.10. In Figure 4.9 only the second case has to be considered. The structural priority determinants of the compound transitions $\{t_1\}$, $\{t_2\}$, and $\{t_3\}$ are the states $\mathsf{C}$, $\mathsf{C}$, and $\mathsf{B}$, respectively. In Figure 4.10 we have the first case for the compound transition $\{t_1\}$. The and-states $\mathsf{A}$ and $\mathsf{B}$ are higher than the source state of $\{t_1\}$ and are below the scope $\mathsf{S}$ of $\{t_1\}$. The higher of them, i.e. $\mathsf{A}$, is the structural priority determinant of $\{t_1\}$.

Formally, the structural priority determinant of a compound transition $T_1 \in CT$

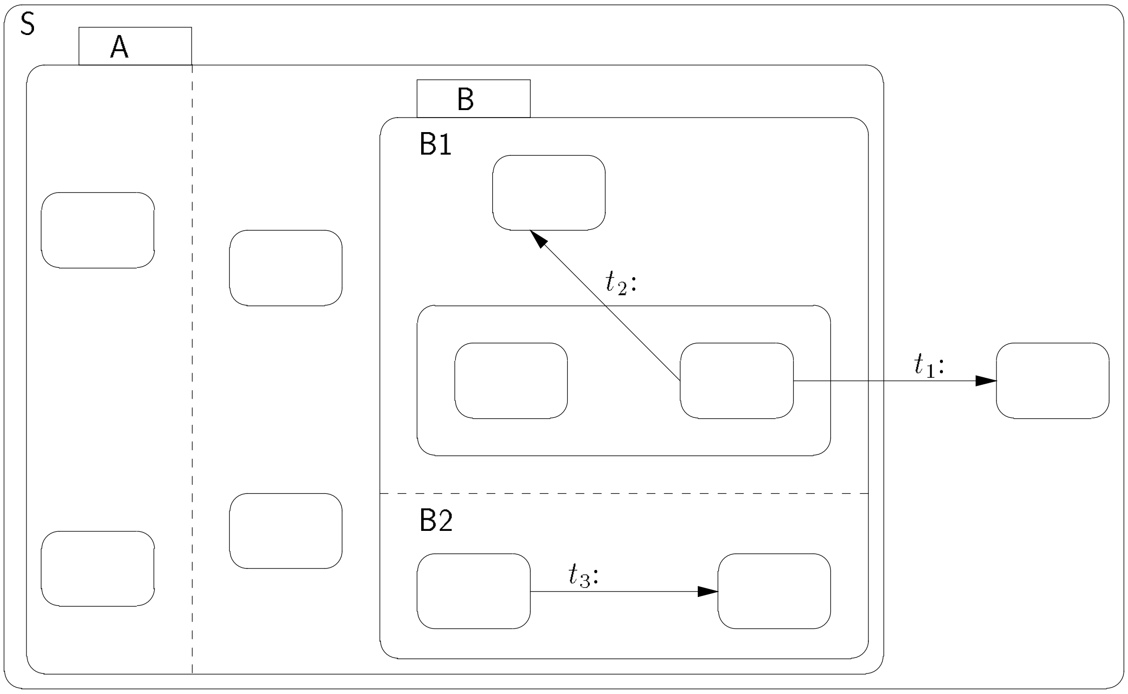



Figure 4.10: Structural priority determinant (cont.), [Harel and Naamad, 1996, Figure 29]

can be defined as:

$$
\begin{aligned}
& sp.\,determinant\colon CT \to S\\
& sp.\,determinant(T_1) = s\\
& \text{iff} \bigvee_{s_1 \in source(T_1)} \wedge \bigvee_{s_2 \in S} \begin{array}{l} \wedge\ substate^+(s_1, s_2)\\ \wedge\ substate^+(s_2, scope(T_1))\\ \wedge\ and(s_2)\\ \wedge \bigwedge_{s_3 \in S} \begin{array}{l} \wedge\ substate^+(s_2, s_3)\\ \wedge\ substate^+(s_3, scope(T_1))\\ \wedge\ or(s_3)\end{array}\end{array}\\
& \qquad\qquad\qquad\quad \Rightarrow s = s_2\\
& \qquad\qquad\quad \wedge \bigwedge_{s_2 \in S} \begin{array}{l} \wedge\ substate^+(s_1, s_2)\\ \wedge\ substate^+(s_2, scope(T_1))\\ \wedge\ or(s_2)\end{array}\\
& \qquad\qquad\qquad\quad \Rightarrow s = s_1
\end{aligned}
$$

The source states of a compound transition $T_1$ are required to be pairwise orthogonal. For two orthogonal source states there must be a common ancestor of them which is an and-state and which is below the scope of $T_1$. Hence, if there is an and-state between at least one source state and the scope of $T_1$, then there is an and-state between each source state and the scope of $T_1$. This implies, that although there is an existential quantifier in the definition of *sp. determinant*, this function is well-defined.

According to this definition the structural priority determinant of a static transition is its source state.

A compound transition $T_1$ has higher structural priority than another compound transition $T_2$ if its structural priority determinant is higher.

$$sp.\,higher\colon CT \times CT$$
$$sp.\,higher(T_1, T_2) \equiv substate^{+}(sp.\,determinant(T_2), sp.\,determinant(T_1))$$

$$sp.\,equal\colon CT \times CT$$
$$sp.\,equal(T_1, T_2) \equiv sp.\,determinant(T_2) = sp.\,determinant(T_1)$$

Note that two compound transitions which are not in conflict and which are not static transitions are not comparable w.r.t. their structural priority. This is due to the fact, that the structural priority determinants are below the scope of the compound transitions. Because the scopes are orthogonal, the structural priority determinants are orthogonal, too.

### 4.5.2 Explicit Priority

In [Harel and Naamad, 1996] *priority numbers* are introduced in addition to structural priority to distinguish between compound transitions with the same structural priority. This kind of priority is called *explicit priority*.

To compare two compound transitions w.r.t. their explicit priority these are considered as sequences of the transition segments of which they consist. If these have and-connectors as source or target, this is not possible directly. In this case the corresponding segments are combined to a so-called *virtual transition segment*. Its priority number is the maximum of the priority numbers of the segments combined. In the statechart in Figure 4.11, there are three virtual transition segments: $\{t_1, t_2, t_3, t_4, t_5\}$, $\{t_6\}$, and $\{t_7, t_8, t_9\}$. These have the priority numbers 3, 2, and 2, respectively.

Two compound transitions are compared w.r.t. their explicit priority by determining the first different transition segments in the linear sequence of (virtual) transition segments. The priority numbers of these first different segments are used to determine the explicit priorities of these compound transitions w.r.t. to each other. If the first different transition segments have the same priority number, then the corresponding transitions have the same explicit priority, too. This is even the case if subsequent segments have different priority numbers. An advantage of this definition is, that one does not have to consider whether such subsequent segments actually exist.

In Figure 4.12 some transition segments together with their priority numbers are given. Table 4.2 shows, how the explicit priorities of the different transitions are

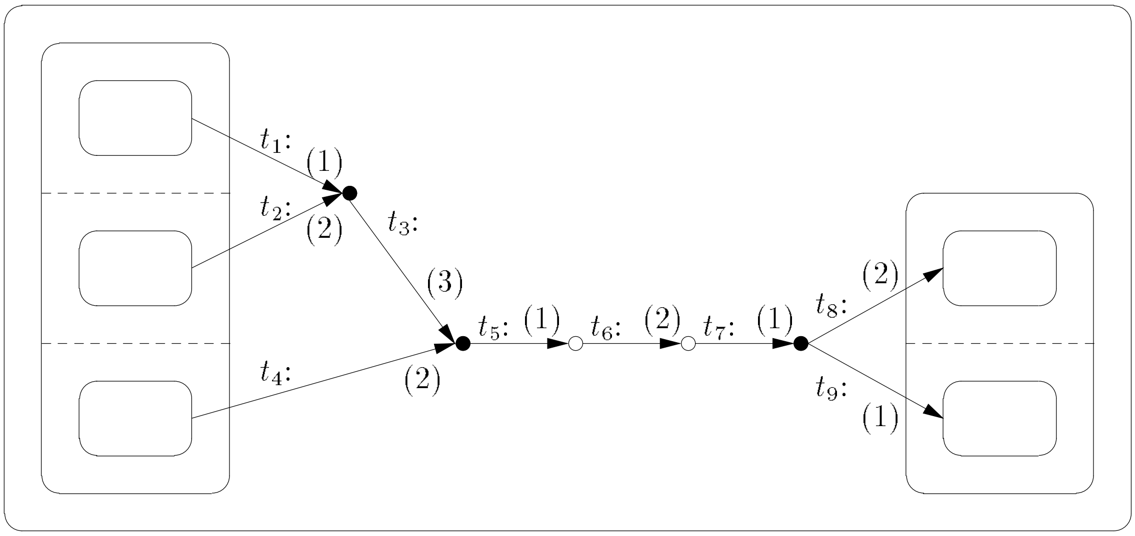



Figure 4.11: Virtual transition segments, [Harel and Naamad, 1996, Figure 32]

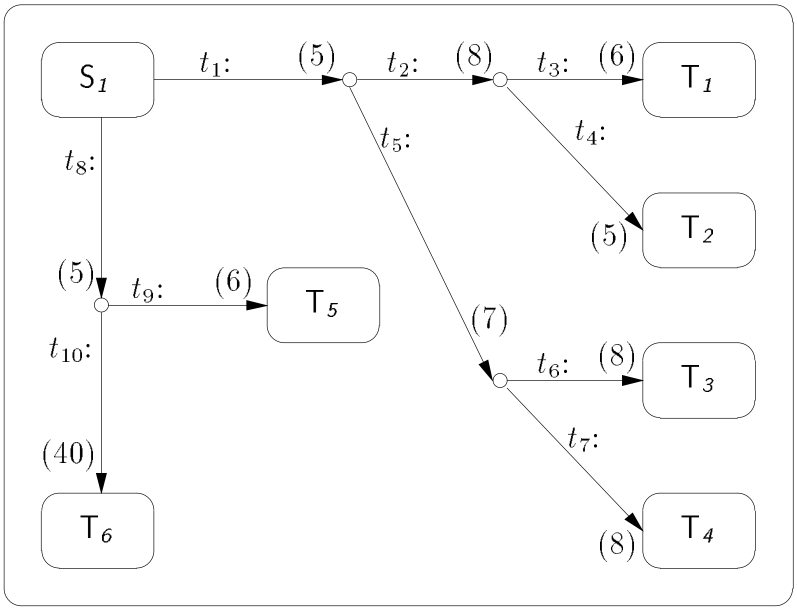



Figure 4.12: Explicit priority, [Harel and Naamad, 1996, Figure 31]

---

related. The entry $> (t_3, t_4)$ in the row denoted by $T_1$ and the column denoted by $T_2$ means that the compound transition from the state S to $T_1$ has higher explicit priority than the compound transition from S to $T_2$. The transition segments deciding the priority are $t_3$ and $t_4$. Note again, that for example the compound transitions leading to the states $T_1$ and $T_5$ have the same explicit priority, although subsequent transition segments have different priority numbers.

To define explicit priority formally, let

$$\mathit{first.\,segment}\colon 2^T \to T$$

denote the first (virtual) transition segment in such a linear sequence of transition segments. Note that although we consider a compound transition as a set of transition segments, this information can be regained from it. The predicate *ep. higher* is defined over sets of transition segments, because when removing the first transition segment from a compound transition, this need not be a compound transition any longer.

$$\begin{aligned}
&\mathit{ep.\,higher}\colon 2^T \times 2^T\\
&\mathit{ep.\,higher}(T_1, T_2)\\
&\equiv \vee \wedge\ \mathit{first.\,segment}(T_1) = \mathit{first.\,segment}(T_2)\\
&\qquad \wedge\ \mathit{ep.\,higher}(T_1 \setminus \mathit{first.\,segment}(T_1),\ T_2 \setminus \mathit{first.\,segment}(T_2))\\
&\quad \vee \wedge\ \mathit{first.\,segment}(T_1) \neq \mathit{first.\,segment}(T_2)\\
&\qquad \wedge\ \mathit{priority}(\mathit{first.\,segment}(T_1)) > \mathit{priority}(\mathit{first.\,segment}(T_2))
\end{aligned}$$

Note that non-conflicting compound transitions can be compared w.r.t. explicit priority.

### 4.5.3 Combining Priorities

Both kinds of priorities are combined in statecharts in a quite simple way: structural priority takes precedence over explicit priority.

$$\begin{aligned}
&\mathit{higher.\,priority}\colon CT \times CT\\
&\mathit{higher.\,priority}(T_1, T_2) \equiv \vee\ \mathit{sp.\,higher}(T_1, T_2)\\
&\qquad\qquad\qquad\qquad\qquad \vee \wedge\ \mathit{sp.\,equal}(T_1, T_2)\\
&\qquad\qquad\qquad\qquad\qquad\quad \wedge\ \mathit{ep.\,higher}(T_1, T_2)
\end{aligned}$$

| | $T_2$ | $T_3$ | $T_4$ | $T_5$ | $T_6$ |
|---|---|---|---|---|---|
| $T_1$ | $> (t_3, t_4)$ | $> (t_2, t_5)$ | $> (t_2, t_5)$ | $= (t_1, t_8)$ | $= (t_1, t_8)$ |
| $T_2$ | | $> (t_2, t_5)$ | $> (t_2, t_5)$ | $= (t_1, t_8)$ | $= (t_1, t_8)$ |
| $T_3$ | | | $= (t_6, t_7)$ | $= (t_1, t_8)$ | $= (t_1, t_8)$ |
| $T_4$ | | | | $= (t_1, t_8)$ | $= (t_1, t_8)$ |
| $T_5$ | | | | | $< (t_9, t_{10})$ |

Table 4.2: Relation of the explicit priorities in Figure 4.12

## 4.6 Building a Step

A compound transition $T_1$ can be executed at a set of states $S_1$ and a set of events $E_1$ if it is enabled and if there is no other conflicting enabled compound transition with higher priority. In this case $T_1$ is called *executable*.

$$
\begin{array}{l}
\mathit{executable.transition}\colon CT \times 2^S \times 2^E \\
\mathit{executable.transition}(T_1, S_1, E_1) \\
\equiv \wedge\ \mathit{enabled.transition}(T_1, S_1, E_1) \\
\quad \wedge \bigwedge_{T_2 \in CT} \begin{array}{l} \wedge\ \mathit{enabled.transition}(T_2, S_1, E_1) \\ \wedge\ \mathit{conflicting.transitions}(T_1, T_2) \\ \Rightarrow \neg\, \mathit{higher.priority}(T_2, T_1) \end{array}
\end{array}
$$

In a step of statechart at a status a maximal set, w.r.t. set inclusion, of executable non-conflicting compound transitions can be taken. We do not require that this set is non-empty. This will be important when considering activities in Chapter 8.

$$
\begin{array}{l}
\mathit{maximal.step}\colon 2^{CT} \times 2^S \times 2^E \\
\mathit{maximal.step}(\mathcal{T}, S_1, E_1) \\
\equiv \wedge \bigwedge_{T_1 \in \mathcal{T}} \mathit{executable.transition}(T_1, S_1, E_1) \\
\quad \wedge \bigwedge_{T_2, T_3 \in \mathcal{T}} \begin{array}{l} \vee\ \neg\, \mathit{conflicting.transitions}(T_2, T_3) \\ \vee\ T_2 = T_3 \end{array} \\
\quad \wedge \bigwedge_{T_4 \in CT \setminus \mathcal{T}} \begin{array}{l} \vee\ \neg\, \mathit{executable.transition}(T_4, S_1, E_1) \\ \vee \bigvee_{T_5 \in \mathcal{T}} \mathit{conflicting.transitions}(T_4, T_5) \end{array}
\end{array}
$$

The last condition prevents that compound transitions which are executable and which are not in conflict with other ones are not part of a step.

## 4.7 Executing a Step

The effects of taking a step are described by several TLA-actions. The flexible variables changed by these TLA-actions have to be identified. Due to the restriction on statecharts we do not use flexible variables corresponding to variables of a statechart. It is sufficient to consider the three variables

| | |
|---|---|
| $S_{active}$ | the set of active states |
| $E_{exists}$ | the set of existing events |
| *step.taken* | a boolean flag |

The variable *step.taken* is used to distinguish steps of the statechart from steps of the environment, cf. Section 4.8. In the following definitions we assume that $S_{active}$

is either the empty set or a configuration. The tuple of these variables is the *status* of the statechart.

$$status \equiv \langle S_{active}, E_{exists}, step.\,taken \rangle$$

We often use the predicate *in* to denote that a state is in the set of active states.

$$in \colon S$$
$$in(s) \equiv s \in S_{active}$$

In the following we describe the effects of taking a step on the set of active states and on the set of existing events.

### 4.7.1 Changing the Configuration

The effects on the set of active states of taking a step consisting of a set of compound transitions are not influenced by static compound transitions. The active states strictly below the scope of a full compound transition are left or become inactive. The states strictly below the scope of a full compound transition and above its target states, including them, are entered or become active. For one full compound transition these sets are defined by:

$$states.\,left \colon CT \to 2^S$$
$$states.\,left(T_1) = \begin{cases} descendants^+(scope(T_1)) \cap S_{active} & \text{if } T_1 \in FCT \\ \emptyset & \text{if } T_1 \in ICT \cup SCT \end{cases}$$

$$states.\,entered \colon CT \to 2^S$$
$$states.\,entered(T_1) = \begin{cases} \cap\ descendants^+(scope(T_1)) & \\ \cap \bigcup\limits_{s \in targets(T_1)} ancestors^*(s) & \text{if } T_1 \in FCT \cup ICT \\ \emptyset & \text{if } T_1 \in SCT \end{cases}$$

The states which are not below the scope of a full compound transition, including the scopes, remain active or inactive. This is described by the following TLA-action, together with the changes caused by the full compound transitions.

$$Change.\,configuration \colon \{\mathcal{T} \subseteq 2^{CT} \mid maximal.\,step(\mathcal{T})\}$$
$$\begin{aligned} Change.\,configuration(\mathcal{T}) \equiv S'_{active} = &\cup\ S_{active} \setminus \bigcup_{T_1 \in \mathcal{T}} states.\,left(T_1) \\ &\cup \bigcup_{T_1 \in \mathcal{T}} states.\,entered(T_1) \end{aligned}$$

Due to the definition of conflicts between compound transitions, the scope of a static compound transition in $\mathcal{T}$ cannot be below the scope of a full compound one. Hence, if $\mathcal{T}$ contains static compound transitions, their states remain active. Below these states it is possible that the configuration is changed.

It remains to show that the TLA-action *Change. configuration* maintains the property that $S_{active}$ is a configuration. Therefore it is sufficient to look at single full compound transitions because the full compound transitions taken in a step are required to be a maximal step, cf. Section 4.6. Therefore they are not in conflict with each other. That the sets of target states of one full compound transition can be part of a configuration has been argued already at the end of Section 4.3.2. After taking a full compound transition the states below the scope and above the target states, including the scope as well as the target states, are active. If the target states can be part of a configuration, then the ancestors of the target states up to the scope of the full compound transition can also be part of a configuration.

### 4.7.2 Generating Events

When taking a step, the action expressions of all transition segments of its compound transitions, of the enabled static reactions, and those on entering or leaving states have to be executed. Remember that this means here to generate explicit events. For a set of compound transitions this is denoted by the function *triggered. events*

$$\begin{aligned}
&\mathit{triggered.\,events}\colon 2^{CT} \to 2^{E_{expl}}\\
&\mathit{triggered.\,events}(\mathcal{T}) = \cup \bigcup_{\{r\in SR \mid enabled.static.reaction(r, S_{active}, E_{exists}, \mathcal{T})\}} \mathit{events}(r)\\
&\qquad \cup \bigcup_{T_1\in\mathcal{T}} \cup \bigcup_{t\in T_1} \mathit{events}(t)\\
&\qquad\qquad \cup \bigcup_{s\in states.left(T_1)} \mathit{exiting.\,events}(s)\\
&\qquad\qquad \cup \bigcup_{s\in states.entered(T_1)} \mathit{entering.\,events}(s)
\end{aligned}$$

Furthermore, the implicit events indicating changes of the status have to be generated. This means to generate configuration events corresponding to entering or leaving a state.

$$\begin{aligned}
&\mathit{configuration.\,events}\colon 2^{CT} \to 2^{E_{conf}}\\
&\mathit{configuration.\,events}(\mathcal{T}) = \bigcup_{T_1\in\mathcal{T}} \cup \{\mathsf{ex}(s) \mid s \in \mathit{states.\,left}(T_1)\}\\
&\qquad\qquad \cup \{\mathsf{en}(s) \mid s \in \mathit{states.\,entered}(T_1)\}
\end{aligned}$$

### 4.7.3 Taking the Step

A step of the statechart is described by a TLA-action which chooses non-deterministically a maximal set of compound transitions as described in Section 4.6. The set of active states is changed according to these compound transitions and the corresponding events are generated. The non-determinism is expressed by existential quantification over the sets of compound transitions.

$$\begin{array}{l} Step\colon \\ Step \equiv \bigvee\limits_{\mathcal{T} \subseteq CT} \begin{array}{l} \wedge\ maximal.\,step(\mathcal{T}, S_{active}, E_{exists}) \\ \wedge\ Change.\,configuration(\mathcal{T}) \\ \wedge\ E'_{exists} = triggered.\,events(\mathcal{T}) \cup configuration.\,events(\mathcal{T}) \end{array} \end{array}$$

Steps with an empty set of executable compound transition are possible. This implies that no compound transition is enabled. Enabled static reactions can still be taken in such a step. In any case, the events existing at the beginning of the step are discarded when taking it.

The configuration events make it visible which states are left or entered in a step of the statechart. This makes it possible to define whether a state is left, entered, or remains active in a step without refering to the compound transitions taken in the step:

$$\begin{array}{l} Entered\colon S \\ Entered(s) \equiv \mathsf{en}(s) \in E'_{exists} \end{array}$$

$$\begin{array}{l} Exited\colon S \\ Exited(s) \equiv \mathsf{ex}(s) \in E'_{exists} \end{array}$$

$$\begin{array}{l} Staying\colon S \\ Staying(s) \equiv \begin{array}{l} \wedge\ in(s) \\ \wedge\ in'(s) \\ \wedge\ \neg\, Entered(s) \end{array} \end{array}$$

## 4.8 Behaviour of a Statechart

Sequences of statuses of a statechart can be defined now as sequences of TLA-states. A TLA-program corresponding to the statechart can be defined. The steps of the statechart are characterised by the TLA-action *Step*. That a statechart-step is actually taken is made visible by changing the boolean variable $step.\,taken$. This allows to distinguish statechart-steps which change neither the set of active states nor the set of existing events from stuttering steps of the TLA-program.

$$\begin{array}{l} Execute.\,step\colon \\ Execute.\,step \equiv \begin{array}{l} \wedge\ Step \\ \wedge\ step.\,taken' \neq step.\,taken \end{array} \end{array}$$

The status of the statechart can be influenced by its environment, too. This can be used to provide one or more stimuli to the statechart. In such a step the set of existing events can be enlarged, the set of active states and the boolean flag are not changed.

$$\begin{aligned}&\mathit{External.\,event}:\\&\mathit{External.\,event} \equiv \wedge\ E_{exists} \subsetneq E'_{exists}\\&\qquad\qquad\qquad\quad \wedge\ S'_{active} = S_{active}\\&\qquad\qquad\qquad\quad \wedge\ \mathit{step.\,taken}' = \mathit{step.\,taken}\end{aligned}$$

To prevent behaviours, in which a step of the statechart is possible forever, but not taken, we require weak fairness for *Execute. step*-steps. This does not imply fairness among different possible steps of a statechart.

The initial status of a statechart is the empty set of active states and an arbitrary set of existing events. This implies that the statechart has to be initialised using the compound transition for initialisation, cf. Section 4.3.4, and the definition of the predicate *enabled. transition*, cf. Section 4.4. The initial value of *step. taken* is chosen, somewhat arbitrarily, as *false*.

$$\begin{aligned}&\mathit{initial.\,status}:\\&\mathit{initial.\,status} \equiv \wedge\ S_{active} = \emptyset\\&\qquad\qquad\qquad\ \ \wedge\ E_{exists} \subseteq E\\&\qquad\qquad\qquad\ \ \wedge\ \neg\,\mathit{step.\,taken}\end{aligned}$$

In summary, the behaviour of a statechart is described by the following TLA-program:

$$\begin{aligned}&\mathit{statechart}:\\&\mathit{statechart} \equiv \wedge\ \mathit{initial.\,status}\\&\qquad\qquad\quad \wedge\ \Box\begin{bmatrix}\vee\ \mathit{Execute.\,step}\\ \vee\ \mathit{External.\,event}\end{bmatrix}_{status}\\&\qquad\qquad\quad \wedge\ \mathit{WF}_{status}(\,\mathit{Execute.\,step}\,)\end{aligned}$$

A statechart can be seen now not only as generating a set of runs, but also as accepting runs. We say, a run $\sigma$ *satisfies* a statechart if and only if $\sigma \models \mathit{statechart}$, where *statechart* is the TLA-program corresponding to this statechart.

## 4.9 Other Semantics of Statecharts

A lot of explanations on statecharts and the TLA-formulae describing their semantics have been given before, but this does not constitute a formal proof that the TLA-program actually capture the semantics of statecharts. To do this it would be necessary to relate our semantics with another one, which itself has to be defined formally, too. It should be clear that this can be done only for the common subset of statecharts captured by both semantics. Another possibility would be to derive properties from the TLA-program *statechart*. The most important properties are that the set of active states is always a configuration and that only non-conflicting compound transitions are taken together.

An overview on different variants of statecharts is given in [van der Beeck, 1994]. Several problems of the semantics of statecharts are identified and explained there. It is shown how these problems are solved in the variants of statecharts.

The semantics described here can be considered as a *single-step* semantics: Each single step counts already as a complete step. Only compound transitions which are executable at the beginning of the step can be part of it and a maximal set of non-conflicting compound transitions has to be taken. It is also possible to define a semantics of statecharts such that the events generated by one compound transition can be used to enable further compound transitions. This is done in the *micro-step* semantics of [Pnueli and Shalev, 1991] and the *super-step* semantics of [Harel and Naamad, 1996]. The three kinds of semantics correspond to the different kinds of feedback of mini-statecharts and $\mu$-charts, cf. Section 2.5.5.

Using the micro-step semantics [Pnueli and Shalev, 1991] a step of a statechart is constructed itself in a stepwise manner. An executable compound transition is picked, the events generated when taking the transition segments are added to the already existing events. This new set of events is used to pick a further executable compound transition which is not in conflict with the first one. This is repeated until no more compound transitions can be added. Because only finitely many non-conflicting compound transitions can be taken together, this construction is guaranteed to stop. All compound transitions collected this way are taken in parallel. When negated events are allowed then it is possible that in one step two compound transitions with contradictory triggers are taken together: One depending on the absence of an event, and the other one depending on the same event being present. It is also possible to construct steps in which not all non-conflicting compound transitions executable at the beginning of a step are taken in it. The situation becomes even more complicated when variables can be assigned new values when taking a compound transition and when compound transitions depend on conditions on these variables. The single-step semantics used here does not have these problems. But note that when associating temporal formulae with states similar problems will be introduced, cf. Section 6.2 and Chapter 9.

Another possibility to construct a step in a stepwise fashion is to take a sequence of steps, where each step is a single step as defined here. Such a sequence of steps is called a super-step in [Harel and Naamad, 1996]. But whereas the construction of steps according to the micro-step semantics terminates this is no longer the case for super-steps. There can be infinite loops of compound transitions enabling each other in turn. Although it can be convenient to view several internal steps as one external one, we do not want to have non-terminating steps as basis to describe the behaviour of statecharts. Note that activities, cf. Chapter 8, can also be used to describe sequences of statuses.

At last we want to present some related work: In [Day, 1993, Day and Joyce, 1993] the semantics of statecharts is defined in HOL. [Glässer and Karges, 1997] define a semantics of SDL in abstract state machines, ASM [Gurevich, 1995], using a similar approach as ours. A semantics of statecharts *and* activity-charts is given in [Damm *et al.*, 1998], comparing their approach to ours is postponed to Section 8.4.3.

In [Day, 1993, Day and Joyce, 1993] the semantics of a part of statecharts is defined by translating statecharts to a variant of higher order logic, more precisely to the input language of the proof assistant HOL [Gordon and Melham, 1993]. The semantics covers a large subset of statecharts including the use of variables. The semantics is a single-step semantics. The general approaches of their work and ours are similar. The conditions and effects of taking one compound transition are defined and this is used to define a next-state relation as a logical formula. This describes which compound transitions can be taken together and the combined effect of the compound transitions taken. But the aims of the work are quite different. The next-state relation defined there is used as an input to a model-checker to prove properties of the system, whereas we will take the TLA-description of the next-state relation as the basis of extending the formalism.

In [Glässer and Karges, 1997] the semantics of SDL [CCITT, 1993] is given by defining corresponding abstract state machines. Abstract state machines provide the possibility to define first order functions, whose values can be changed by so-called *updates*. A set of sequences of first-order structures is obtained by executing repeatedly updates. The sequences obtained this way are not closed under stuttering. There is a close relationship between their and our approach of defining semantics: In both cases one concentrates on describing the changes of a specific set of variables or functions by TLA-actions or ASM-updates, respectively.[1] Note that in ASM fairness-properties cannot be expressed and updates are similar to guarded commands, whereas TLA-actions are predicates on primed and unprimed variables. Hence TLA seems to be more suitable w.r.t. the combination of statecharts with a property-oriented language. We consider ASM to be a model-oriented formalism.

[1] The relationship between TLA and ASM is an interesting problem on its own.

# Chapter 5

# Data Variables

In Section 4.1 we introduced informally variables, condition and action expressions. It is possible in statecharts to restrict enabledness of compound transitions by conditions on the variables. Vice versa, when taking compound transitions in a step it is possible to change these variables by executing the action expressions of the transition segments being part of the compound transitions. In the formal definition of the semantics of statecharts in the rest of Section 4 we concentrated on constructing compound transitions and on their contribution to steps of a statechart. In this section we cover the use of variables in a formal way.

At first we give the statecharts-syntax of action, condition, and event expressions. For each event expression, including the condition expressions, we define a corresponding predicate in TLA. Similarly, for each action expression we define a TLA-action. In the case of action expressions we restrict ourselves such that in each step each variable can be written at most once. This allows us to give a more simple definition of the semantics of executing action expressions. The TLA-predicates and TLA-actions are inserted in the definitions of enabledness of compound transitions and of executing a step in Section 5.2. Thereafter we extend the TLA-actions used towards non-deterministic assignments. This can be seen as a first extension of statecharts by TLA, but this is not an extension w.r.t. the description of temporal properties.

## 5.1 Syntax and Semantics of Expressions

We give here a definition of the syntax of event, condition, and action expressions. These definitions extend those already given in Chapter 4. For each event and action expression of a statechart we define a corresponding TLA-predicate, respectively a TLA-action. These are used to define whether a compound transition is enabled and what are the effects of taking a step.

### 5.1.1 Syntax of Expressions

Each variable of a statechart is of one of the predefined primitive datatypes integer, real, string, bit, and bit_array or of one of the structured datatypes record, one-of, and array. In statecharts so-called *named single conditions* can be used. We consider these simply as boolean variables and therefore allow variables of type boolean, too. The type of a variable is denoted by the function *datatype*, where $W$ denotes the set of variables of the statechart. $W$ is assumed to be finite and disjoint to the sets $S$, $C$, $E$, $T$, and $SR$.

$$W$$

$$datatype\colon W \to \{\mathsf{boolean}, \mathsf{integer}, \mathsf{real}, \mathsf{string}, \mathsf{bit}, \mathsf{bit\_array}, \mathsf{record}, \mathsf{one\text{-}of}, \mathsf{array}\}$$

The datatype record denotes records as usual, whereas the datatype one-of denotes the disjoint union of datatypes. For each of the datatypes typical function symbols are predefined in statecharts. Some examples are given in Table 5.1. Furthermore, there are function symbols denoting conversion between variables of type integer and real and between integer and bit_array, respectively.

**Action Expressions** The *atomic action expressions* we consider here are assignments and those to generate explicit events. Further atomic action expressions related to activities will be introduced in Chapter 8.

The atomic action expression to generate an explicit event $e$ is simply written as $e$.

The variables and predefined function symbols can be used to build expressions. An assignment $x := expr$, where $x$ is a variable and $expr$ an expression of corresponding type, is an atomic action expression. To assign a boolean value to a boolean variable $b$ the atomic action expressions make_true($b$) and make_false($b$) can be used, these are abbreviated to tr!($b$) and fs!($b$), respectively.

Starting from the atomic action expressions, *compound action expressions* can be built. These have been introduced in Section 4.1.2, they are repeated in Table 5.2.

By $\mathcal{A}$ we denote the set of action expressions w.r.t. a statechart.

| datatypes | predefined function symbols |
|---|---|
| integer | +, *, /, ... , min, max, random, ... |
| real | +, *, /, ... , min, max, sine, cosine, ... |
| bit_array | ashl (arithmetic shift left), bits_of (slice of bit array), |
| array | length_of, lindex (left index), rindex |

Table 5.1: Examples of datatypes and predefined function symbols of statecharts

**Condition Expressions** *Condition expressions* are used to define predicates over the status. The *atomic condition expressions* are shown in Table 5.3. The intended semantics is: A condition expression $b$ is valid at a status if the boolean variable $b$ is true. $\mathsf{in}(s)$ is valid at a status if the state $s$ is active. $x \sim y$ denotes the usual comparisons, where $<, =<, >$, and $=>$ can be applied to numerical variables only. $\mathsf{all}(c)$ is valid if each element of the array of boolean variables $c$ is true, $\mathsf{any}(c)$ is defined similarly.

As before, further condition expressions related to activities will be introduced in Chapter 8.

The set of condition expressions is the smallest set containing the atomic ones and which is closed under the boolean operators and, or, and not. Parentheses can be used if necessary.

**Event Expressions** In Section 4.2.2 we introduced already the sets $E_{expl}$ and $E_{impl}$ of explicit and implicit events, respectively. $E_{impl}$ contains the set $E_{conf}$ of configuration events. The implicit events we consider here are shown in Table 5.4 together with their short forms.

The implicit event $\mathsf{changed}(x)$ is generated whenever the value of the variable $x$ changes in a step. $\mathsf{true}(b)$ indicates that the value of the boolean variable $b$ changed to true, $\mathsf{false}(b)$ indicates that it changed to false.

The implicit events $\mathsf{read}(x)$ and $\mathsf{written}(x)$ are generated when the variable $x$ is read or written in a step, respectively. We consider a variable as being read if it occurs in the right hand side of an assignment or in the condition expression $c$ of a compound

| | | | |
|---|---|---|---|
| $a_1;a_2$ | where | $a$, $a_1$, $a_2$ | are action expressions |
| if $c$ then $a_1$ else $a_2$ end if | | $c$ | is a condition expression |
| when $e$ then $a_1$ else $a_2$ end when | | $e$ | is an event expression |
| for $i$ in $k$ to $l$ loop $a$ end loop | | $k$, $l$ | are integer expressions |
| for $i$ in $k$ downto $l$ loop $a$ end loop | | $i$ | is a context variable |
| while $c$ loop $a$ end loop | | | |
| break | | | |

Table 5.2: Compound action expressions

| | | |
|---|---|---|
| $b$ | where | $b$ is a boolean variable |
| $\mathsf{in}(s)$ | | $s$ is a statecharts-state |
| $x \sim y$ | | $\sim \in \{=, /=, <, =<, >, =>\}$ |
| $\mathsf{all}(c)$ | | $c$ is an array of boolean variables |
| $\mathsf{any}(c)$ | | $c$ is an array of boolean variables |

Table 5.3: Atomic condition expressions

action expression when $c$ ... end when and which are executed in a step. A variable is considered as written if it is assigned a value by an action expression. This view does not coincide exactly with that of statecharts where there are special atomic action expressions to denote that one of these events is generated.

Further event expressions related to activities will be introduced in Chapter 8.

The syntax of compound event expressions is more complicated as shown in Section 4.2.2. As before, the set of event expressions is closed under the boolean connectives and, or, and not. But $[c]$ and $e[c]$ are event expressions, too, where $e$ is an event expression and $c$ is a condition expression. $e[c]$ expresses that the event $e$ occurs *when* the condition $c$ is valid.

This syntax allows to write event expressions such as $e[c_1][c_2]$ and $(\text{not } e[c_1])[c_2]$. The first of these expressions means that the event $e[c_1]$ occurs when the condition $c_2$ is valid. This means that $e$ occurs when $c_1$ and $c_2$ are valid. Hence $e[c_1][c_2]$ has the same meaning as $e[c_1 \text{ and } c_2]$. Analogously, $(\text{not } e[c_1])[c_2]$ means, that not $e[c_1]$ occurs when $c_2$ is valid. That not $e[c_1]$ occurs means that either $e$ does not occur or $c_1$ is not valid. Hence $(\text{not } e[c_1])[c_2]$ has the same meaning as $(\text{not } e)[c_2] \text{ or } [(\text{not } c_1) \text{ and } c_2]$. Similarly the meanings of $(e_1 \text{ and } e_2)[c]$ and $(e_1 \text{ or } e_2)[c]$ coincide with the meanings of $e_1[c] \text{ and } e_2[c]$ and $e_1[c] \text{ or } e_2[c]$, respectively.

Without showing this formally, it should be clear, that it is sufficient to consider only event expressions $e[c]$ and their boolean combinations, where $e$ itself does not contain a condition expression.

Let $\mathcal{E}(E)$ denote the set of event expressions over the set $E$ of events and the variables as defined before.

### 5.1.2 Semantics of Expressions

In the previous section we defined the syntax of action, condition, and event expressions. Here we define their semantics. To do so we use a TLA-predicate for each event expression and a TLA-action for each action expression to define what it means that an event expression is valid and an action expression is executed.

We assume that each transition segment has a label consisting of an event expression and an action expression. These are denoted by the functions *trigger* and *action*.

| | | | | | | |
|---|---|---|---|---|---|---|
| entered($s$) | en($s$) | exited($s$) | ex($s$) | where | $s$ | is a statecharts-state |
| changed($x$) | ch($x$) | | | | $x$ | is a variable |
| true($b$) | tr($b$) | false($b$) | fs($b$) | | $b$ | is a boolean variable |
| read($x$) | rd($x$) | written($x$) | wr($x$) | | | |

Table 5.4: Implicit events and their abbreviations

Similarly, when entering or leaving a state an action expression can be executed. We use $\perp$ to denote an "empty" event as well as an "empty" action expression.

$$
\begin{aligned}
&\mathit{trigger}: T \cup SR \to \mathcal{E}(E) \cup \{\perp\}\\
&\mathit{action}: T \cup SR \to \mathcal{A} \cup \{\perp\}\\
&\mathit{entering.\,actions}: S \to \mathcal{A} \cup \{\perp\}\\
&\mathit{leaving.\,actions}: S \to \mathcal{A} \cup \{\perp\}
\end{aligned}
$$

The functions *action*, *entering. actions*, and *leaving. actions* replace the functions *events*, *entering. events*, and *leaving. events* introduced in Section 4.2.

Similar to the sets of active states and existing events the variables occurring in the expressions belong to the status of the statechart. The tuple of these variables is denoted by $w$. The status is extended to

$$\mathit{status} \equiv \langle S_{\mathit{active}}, E_{\mathit{exists}}, w, \mathit{step.\,taken}\rangle$$

**Event and Condition Expressions** In Section 4.4 we defined the predicate *exists* to check whether an event expression not containing a condition expression is valid at a status. To define a similar predicate for condition expressions it would be necessary to leave the first-order logic framework used here. To avoid this we use a predicate $\mathit{trigger}_t$ on the status for each transition segment and static reaction $t$.

$$\mathit{trigger}_t:$$

We do not give a formal definition of how $\mathit{trigger}_t$ can be constructed from the event expression $\mathit{trigger}(t)$. But it should be clear that this can be done for each event expression of a statechart. For a syntactically correct statechart each of the predicates $\mathit{trigger}_t$ is total and computable.

**Action Expressions** In a way similar to event and condition expressions we use a TLA-action $\mathcal{N}_{\mathsf{a}}$ to define the meaning of executing an action expression $\mathsf{a}$. $\mathcal{N}_{\mathsf{a}}$ is a TLA-action over $w, w', E_{\mathit{exists}}, E'_{\mathit{exists}}$, and $S_{\mathit{active}}$. But it does not refer to $S'_{\mathit{active}}$.

This view is too general to describe the semantics of executing a set of action expressions. Some critical aspects of the composition of systems cannot be handled then. At first, in a step of the statechart several action expressions can be executed simultaneously. According to [Harel and Naamad, 1996] the semantics of this composition is an arbitrary interleaving of the execution of the individual action expressions.[1] If a variable is used by several action expressions, then for each action expressions its value at the beginning of this sequence of executions has to be taken. Vice versa, if a variable in such an interleaved execution is changed by several action expressions, the value determined by the last change is the new value of the variable.

[1]Note, that the semantics of the statechart operator ; to compose action expressions $\mathsf{a}_1$ and $\mathsf{a}_2$ to $\mathsf{a}_1;\mathsf{a}_2$ is defined as executing $\mathsf{a}_1$ and $\mathsf{a}_2$ in an arbitrary order, but *not* as executing $\mathsf{a}_1$ before $\mathsf{a}_2$.

Therefore, the semantics of executing action expressions simultaneously cannot be defined simply as conjunction of the corresponding TLA-actions.

At second, it can depend on the current status, whether a variable is changed when executing an action expression or not. As an example, consider the action expression a defined as **if** $c$ **then** $x_1 := y_1$ **else** $x_2 := y_2$ **end if**, where $c$ is a condition expression and $x_1$, $x_2$, $y_1$, and $y_2$ are variables of appropriate types. Depending on the value of $c$ one of the variables $x_1$ and $x_2$ is changed and one of $y_1$ and $y_2$ is read. Nothing is said about the value of the other variable, especially it is not required that its value is not changed. As a consequence, the meaning of this action expression cannot be described by a function $f_a(c, x_1, x_2, y_1, y_2)$ and a TLA-action of the simple form $\langle x_1, x_2\rangle' = f_a(c, x_1, x_2, y_1, y_2)$.

In the following we impose therefore some restrictions on the action expressions which allow, that the composition of action expressions can be considered as conjunction of the corresponding TLA-actions.

The goal of the restrictions on the action expressions is to allow for certain static analysis of the action expressions and to define simultaneous execution of action expressions in a simple way, i.e. by conjunction of the corresponding TLA-actions. By these restrictions it is made clear, which variables are written by an action expression and which ones are read. This means that two subsets $w_{in,a}$ and $w_{out,a}$ of $w$ are determined for each action expression $a$ and that the change of the values can be described by a function $f_a$ with $w'_{out,a} = f_a(S_{active}, E_{exists}, w_{in,a})$. Analogously, the set of events generated explicitly when executing this action expression can be described by a function $generated.\ events_a(S_{active}, E_{exists}, w_{in,a}) \subseteq E_{expl}$. To restrict execution of these functions to specific statuses of the statechart they can be protected by a predicate $guard_a(S_{active}, E_{exists}, w_{in,a})$.

In the following we consider only action expressions $a$ which can be written as TLA-actions of the following form and which are total and computable. This implies that some action expressions containing **while**-loops are excluded.

$$\begin{aligned} \mathcal{N}_a \equiv\ & guard_a(S_{active}, E_{exists}, w_{in,a}) \\ & \Rightarrow \wedge\ w'_{out,a} = f_a(S_{active}, E_{exists}, w_{in,a}) \\ & \quad\ \wedge\ E'_{exists} \supseteq generated.\ events_a(S_{active}, E_{exists}, w_{in,a}) \end{aligned} \tag{5.1}$$

The semantics of the compound action expression $a;b$ and of executing two action expressions $a$ and $b$ simultaneously in one step can then be defined simply as $\mathcal{N}_a \wedge \mathcal{N}_b$ provided that the action expressions do not change the same variables, that is $w_{out,a} \cap w_{out,b} = \emptyset$.

This property is guaranteed by the following condition:

$$\begin{aligned} & w_{out,a} \cap w_{out,b} \neq \emptyset \\ & \Rightarrow \bigwedge_{S_{active} \subseteq S} \bigwedge_{E_{exists} \subseteq E} \bigwedge_{w \in W} \begin{array}{l} \neg\wedge\ guard_a(S_{active}, E_{exists}, w_{in,a}) \\ \ \ \ \wedge\ guard_b(S_{active}, E_{exists}, w_{in,b}) \end{array} \end{aligned} \tag{5.2}$$

Equation 5.2 is actually too strong, because it does not take into account which statuses are reachable and whether the action expressions $\mathsf{a}$ and $\mathsf{b}$ can be executed at all together. This means to take into account whether the transition segments to which the action expressions belong can be executable at one status. To which degree this information is used can be varied further. The following degrees seem to be useful:

1. No information on the static structure of the statechart is used at all.

2. The type of states, e.g. that transition segments having different direct descendants of one *or*-state as source state cannot be taken together in one step, or information about structural priority is used.

3. The values of boolean variables and the existence of events are used.

4. The values of some or all of the data variables $w$ occurring in triggers are used.

It is possible to check the conditions using the different kinds of information in successive order. If one of these checks is successful, i.e. it is shown that two action expressions cannot be taken together in one step, then the other ones need not be applied for this pair of action expressions.

With increasing amount of information used, the effort to perform the checks increases, too. On the other hand the precision of the result of the checks increases, this means that it can be shown for more pairs of action expressions that their guards cannot be valid simultaneously.

It seems to be useful to consider in this kind of analysis especially those variables, out of $S_{active}$, $E_{exists}$, and $w$, which influence the control flow of the statechart. Thereby it should be assumed that no control information has been expressed only implicitly within data variables.

By this restriction on the kind of actions and on the guards it is possible to achieve a simple semantics of executing action expressions without restricting the expressiveness too much.

## 5.2 Steps of Statecharts

Because we consider now a larger set of event expressions than in Chapter 4 the definition of *enabled.transition* has to be adapted appropriately. Similarly, the definition of the TLA-action *Step* has to be adapted to account for executing the additional kinds of action expressions. This is slightly complicated by the need to generate implicitly the events describing access to the variables and those describing changes of variables. This has to be done for steps of the environment, too. These changes to the definitions are presented in the following.

### 5.2.1 Enabling Compound Transitions

Enabledness of compound transitions and static reactions can be adapted to the complete set of triggers considered here by simple changes to the corresponding definitions. For compound transitions we have

$$\begin{aligned}
&enabled.transition : CT\\
&enabled.transition(T_1) \equiv \wedge\ source(T_1) \subseteq S_{active}\\
&\qquad\qquad\qquad\qquad\quad \wedge \bigwedge_{t \in T_1} trigger_t
\end{aligned}$$

This can be used as before to define whether a compound transition is executable at a status. The definition of *executable.transition* has to be adapted slightly because we refered here to the variables of the status directly instead of passing them as parameters.

The definition of *enabled.static.reaction* can be changed in a similar way.

In Section 6.1 we extend the notation by allowing for transitions segments and static reactions that $trigger_t$ is a TLA-action instead of a predicate. The definition above of *enabled.transition* can still be used, but note that it becomes also a TLA-action in this case.

### 5.2.2 Taking a Step

We define now what it means that action expressions are executed when taking a step of a statechart. It does not suffice to consider only the TLA-actions corresponding to the action expressions. In addition to the events generated explicitly those refering to accessing variables have to be generated. These are described by the set $access.events_{\mathsf{a}}$ for an action expression $\mathsf{a}$.

$$\begin{aligned}
access.events_{\mathsf{a}} \equiv &\cup \{\mathsf{rd}(\mathsf{w}_j) \mid \mathsf{w}_j \in w_{in,\mathsf{a}}\}\\
&\cup \{\mathsf{wr}(\mathsf{w}_j) \mid \mathsf{w}_j \in w_{out,\mathsf{a}}\}
\end{aligned}$$

The events $\mathsf{ch}(\mathsf{x})$ will be handled later on.

For each action expression $\mathsf{a}$ of a transition segment or a static reaction[2] and which can be described by a TLA-action of the form shown in Equation 5.1 we define a TLA-action $Take_{\mathsf{a}}$:

$$\begin{aligned}
&Take_{\mathsf{a}} :\\
&Take_{\mathsf{a}} \equiv\ guard_{\mathsf{a}}(S_{active}, E_{exists}, w_{in,\mathsf{a}})\\
&\qquad\qquad \Rightarrow \wedge\ w'_{out,\mathsf{a}} = f_{\mathsf{a}}(S_{active}, E_{exists}, w_{in,\mathsf{a}})\\
&\qquad\qquad\quad\ \ \wedge\ E'_{exists} \supseteq generated.events_{\mathsf{a}}(S_{active}, E_{exists}, w_{in,\mathsf{a}})\\
&\qquad\qquad\quad\ \ \wedge\ E'_{exists} \supseteq access.events_{\mathsf{a}}
\end{aligned}$$

[2]For simplicity, we consider the action expressions to be executed when entering or leaving a state, cf. *entering.actions* and *leaving.actions*, as static reactions, too.

The definition of the TLA-action *Step* which describes a step of the statechart has to be adapted as follows. It can be used as before in the definition of *Execute. step*.

$$
\begin{array}{l}
\mathit{Step}: \\
\mathit{Step} \equiv \bigvee\limits_{\mathcal{T} \subseteq CT} \begin{array}[t]{l}
\wedge\ \mathit{maximal.\,step}(\mathcal{T}, S_{active}, E_{exists}) \\
\wedge\ \mathit{Change.\,configuration}(\mathcal{T}) \\
\wedge\ \mathit{configuration.\,events}(\mathcal{T}) \subseteq E'_{exists} \\
\wedge\ \bigwedge\limits_{T_1 \in \mathcal{T}} \bigwedge\limits_{t \in T_1} \mathit{Take}_{action(t)} \\
\wedge\ \bigwedge\limits_{r \in SR} \begin{array}[t]{l} \mathit{enabled.\,static.\,reaction}(r, S_{active}, E_{exists}, \mathcal{T}) \\ \Rightarrow \mathit{Take}_{action(r)} \end{array}
\end{array}
\end{array}
$$

With respect to changes of variables and to the generation of explicit events it is described here what has to happen at least. All variables which are not output variables of the action expressions of one of the transition segments or static reactions taken in this step should retain their value. Similarly, $E'_{exists}$ should contain only explicit events which are generated by one of the action expressions executed. This will be expressed later on when considering activities, because it has to be distinguished between input, output and local variables of a statechart. These notions will be defined when considering activities, cf. Section 8.4.1.

Here, a statechart may not only generate new events, but it may also change some of the variables. By a step of the environment, the configuration of the statechart cannot be changed, events cannot be thrown away, and there has to be a change of a variable or some events have to be generated:

$$
\begin{array}{l}
\mathit{External.\,change}: \\
\mathit{External.\,change} \equiv \begin{array}[t]{l}
\wedge\ S'_{active} = S_{active} \\
\wedge\ \mathit{step.\,taken}' = \mathit{step.\,taken} \\
\wedge\ E'_{exists} \supseteq E_{exists} \\
\wedge\ \langle E_{exists}, w\rangle' \neq \langle E_{exists}, w\rangle
\end{array}
\end{array}
$$

Consistency on the variables $w$ when taking a step is already ensured by the form of the actions and the condition in Equation 5.2 on their guards. But it is still possible that there are "contradicting" implicit events. An event $\mathsf{ch}(\mathsf{w}_j)$ has to be generated if and only if a step with $w_j \neq w'_j$ is taken. An event $\mathsf{tr}(\mathsf{b}_j)$ is generated if and only if the value of a boolean variable changes from *false* to *true* in a step, vice versa an event $\mathsf{fs}(\mathsf{b}_j)$ is generated. These consistency conditions are captured by the TLA-action *Change. events*:

$$
\begin{array}{l}
\textit{Change. events}: \\
\textit{Change. events} \equiv \land \bigwedge\limits_{w_j \in w} \begin{array}{l} \land\ \mathsf{ch}(w_j) \in E'_{exists} \Leftrightarrow w_j \neq w'_j \\ \land\ \mathsf{ch}(w_j) \in E'_{exists} \Rightarrow \mathsf{wr}(w_j) \in E'_{exists} \end{array} \\
\qquad\qquad\qquad \land \bigwedge\limits_{b_j \in w} \begin{array}{l} datatype(b_j) = \mathsf{boolean} \\ \Rightarrow \land\ \mathsf{tr}(b_j) \in E'_{exists} \Leftrightarrow \begin{array}{l} \land\ b'_j \\ \land\ b_j \neq b'_j \end{array} \\ \quad\ \land\ \mathsf{fs}(b_j) \in E'_{exists} \Leftrightarrow \begin{array}{l} \land\ \neg b'_j \\ \land\ b_j \neq b'_j \end{array} \end{array}
\end{array}
$$

The definition of *statechart* is changed to use the new definition of *External. change* and to ensure consistency of events as described by *Change. events*.

$$
\begin{array}{l}
\textit{statechart}: \\
\textit{statechart} \equiv \land\ \textit{initial. status} \\
\qquad\qquad\quad \land\ \Box \begin{bmatrix} \land\ \textit{Change. events} \\ \land\ \lor\ \textit{Execute. step} \\ \quad\ \lor\ \textit{External. change} \end{bmatrix}_{status} \\
\qquad\qquad\quad \land\ WF_{status}(\textit{Execute. step})
\end{array}
$$

Several *External. change*-steps can be taken between two *execute. step*-steps. Their effects will be accumulated, especially $\mathsf{ch}$ events generated implicitly will be accumulated. Hence a variable $\mathsf{w}_j$ can be set to a new value by the environment, implicitly an event $\mathsf{ch}(\mathsf{w}_j)$ is generated, and in a later step $\mathsf{w}_j$ can be set to its previous value. Nevertheless the event $\mathsf{ch}(\mathsf{w}_j)$ will remain present until the next *execute. step*-step. Hence, although each single step is consistent there can be sequences of steps which can be considered as inconsistent.

## 5.3 Non-Deterministic Assignments

In this section we present a slight extension of statecharts by using TLA. Note that this is not an extension w.r.t. expressing temporal properties.

The values assigned to variables and the events generated explicitly when executing an action expression are determined uniquely by the action expression and the status at the beginning of the step. But it is often convenient to provide a non-deterministic choice between several possible values which can be assigned to a variable or between different events which can be generated.

If the set of values is finite, this can be achieved by several compound transitions with the same priority and which are enabled together. With each of the compound transitions, more precisely with the transition segments constituting them, one deterministic action expression can be associated. When taking a step, there will be a non-deterministic choice between the compound transitions and therefore one between their action expressions.

To avoid to add too many compound transitions and to be able to describe a non-deterministic choice from an infinite set we introduce here non-deterministic TLA-actions. We retain the general form of describing action expressions as shown in Equation 5.1, page 92. But we consider $f_{\mathsf{a}}$ and $\mathit{generated.\,events}_{\mathsf{a}}$ as set-valued functions, i.e. $f_{\mathsf{a}}(S_{active}, E_{exists}, w_{in,\mathsf{a}}) \subseteq W_{out}$ and $\mathit{generated.\,events}_{\mathsf{a}}(S_{active}, E_{exists}, w_{in,\mathsf{a}}) \subseteq 2^{E_{expl}}$. We require that $f_{\mathsf{a}}$ and $\mathit{generated.\,events}_{\mathsf{a}}$ are defined when $\mathit{guard}_{\mathsf{a}}$ is valid at a status. One of the values of $f_{\mathsf{a}}$ and one of the set of events of $\mathit{generated.\,events}_{\mathsf{a}}$ is chosen and used as the effect of executing this TLA-action:

$$
\begin{aligned}
&\mathcal{N}_{\mathsf{a}}: \\
&\mathcal{N}_{\mathsf{a}} \equiv \; \mathit{guard}_{\mathsf{a}}(S_{active}, E_{exists}, w_{in,\mathsf{a}}) \\
&\qquad \Rightarrow \wedge\; w'_{out,\mathsf{a}} \subseteq f_{\mathsf{a}}(S_{active}, E_{exists}, w_{in,\mathsf{a}}) \\
&\qquad\quad\;\; \wedge \bigvee_{E_1 \subseteq E_{expl}} \begin{array}{l} \wedge\; E'_{exists} \supseteq E_1 \\ \wedge\; E_1 \in \mathit{generated.\,events}_{\mathsf{a}}(S_{active}, E_{exists}, w_{in,\mathsf{a}}) \end{array}
\end{aligned}
\tag{5.3}
$$

As before, the dependency between the variables occurring in this action and the implicit events written and read is described by the TLA-action $\mathit{Access.\,events}$.

Note that in this case the action of a transition segment is not an action expression of statecharts. It is a TLA-action which can be used directly in the TLA-action $\mathit{Take}_{\mathsf{a}}$.

The restriction, that in a step the tuples of variables $w_{out,\mathsf{a}}$ and $w_{out,\mathsf{b}}$ of two different actions $\mathsf{a}$ and $\mathsf{b}$ have to be disjoint is retained to prevent contradicting assignments to the same variable. Remember that such a contradicting assignment actually is not performed because the corresponding TLA-actions cannot be satisfied together in this step.

As a quite simple example let us assume that we want to describe a lift system including its environment. This can be done by a statechart with the orthogonal substates environment, plant, and controller. A request for the lift cabin to move to floor $i$ is modeled by occurrence of the event $\mathsf{press}_i$. The assumption on the environment, that in each step there can be a new request for the lift cabin on any floor $i$, for $1 \le i \le n$ can be described by the following TLA-action $\mathcal{N}_{\mathsf{a}}$, which is the action expression of a static reaction of the statecharts-state environment:

$$
\begin{aligned}
&\mathcal{N}_{\mathsf{a}}: \\
&\mathcal{N}_{\mathsf{a}} \equiv \bigvee_{E_1 \subseteq E_{expl}} \begin{array}{l} \wedge\; E'_{exists} \supseteq E_1 \\ \wedge\; E_1 \subseteq \{\mathsf{press}_i \mid 1 \le i \le n\} \end{array}
\end{aligned}
$$

Here, as compared to the formula in Equation 5.3, the predicate $\mathit{guard}_{\mathsf{a}}$ is true and there are no output variables. Due to the latter one, $w_{out,\mathsf{a}}$ can be considered as the empty tuple and $f_{\mathsf{a}}$ does not need to be defined.

# Chapter 6

# Temporal Extensions

In the previous chapters we mapped statecharts and TLA to a common semantic domain. This was done by constructing for each statechart a TLA-program with equivalent behaviour. We did not cover the complete syntax of statecharts in this construction to simplify it. For example we did not consider history connectors and we required that no two action expressions executed in one step change the same variables. On the other hand we extended statecharts in two directions. At first we introduced static transitions to be able to express non-determinism between staying in a state of the statechart and leaving it. At second, we used TLA-actions to express non-deterministic assignments when taking a step.

In this chapter we present two extensions of statecharts with TLA w.r.t. expressing temporal properties.

In a first one we use TLA-actions instead of predicates as triggers of transition segments, cf. Section 6.1. The intention is to express synchronisation of compound transitions in a convenient way. But this extension interferes with the definition of steps of a statechart. Especially when priorities are taken into account the statecharts extended this way become difficult to understand.

By the second extension it becomes possible to associate TLA-formulae with states of a statechart. The TLA-formulae are essentially TLA-programs refering to the status of the statechart. The intention of associating a TLA-formula with a state is to restrict those parts of the behaviour in which this state is active. Therefore we call such a TLA-formula a *temporal constraint*. Temporal constraints will be introduced and investigated from Section 6.2 on.

## 6.1 TLA Actions as Conditions

It is often the case that one wants to express that two compound transitions are taken in synchrony. A typical situation is depicted by an example in Section 6.1.1. Synchrony between compound transitions can be expressed by allowing that the

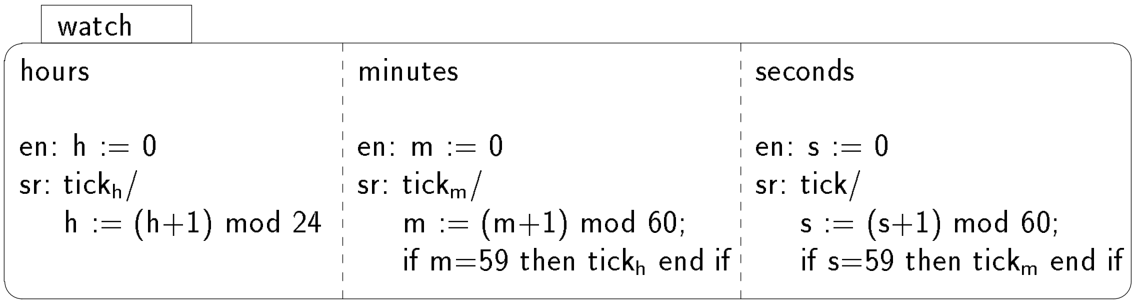



Figure 6.1: Clock, according to [Pnueli and Shalev, 1991]

triggers of transition segments are TLA-actions instead of predicates. The semantics of TLA-actions as triggering conditions is quite simple to define, cf. Section 6.1.2. By using TLA-actions as triggering conditions it can be refered to the change of the status in a step of the statechart. But this interferes with the definition of steps of statecharts which makes the resulting descriptions difficult to understand. This is made clear by an example in Section 6.1.3. Because we consider such descriptions as too difficult to understand we do not pursue this extension further, some conclusions are given in Section 6.1.4.

### 6.1.1 Example: Synchronisation of Compound Transitions

As an example of synchronising compound transitions we consider here a clock similar to [Pnueli and Shalev, 1991] and [Lamport, 1996]. The clock displays seconds, minutes, and hours. The occurrence of an event $\mathsf{tick}$ expresses that one second has passed, as a consequence the displays should be changed accordingly. The problem is to ensure, that changing to the full minute and to the full hour is synchronised between changing seconds and minutes, respectively minutes and hours.

The statechart in Figure 6.1 corresponds to the solution in [Pnueli and Shalev, 1991]. In each of the three parallel substates a variable is initialised and a static reaction is used to change it. To be precise, this means that we synchronise static reactions in this example instead of compound transitions. Changing the seconds, i.e. the variable $\mathsf{s}$, is triggered by occurrence of the event $\mathsf{tick}$. By the same static reaction the event $\mathsf{tick_m}$ is generated if one minute has been completed in this step. $\mathsf{tick_m}$ triggers that the variable $\mathsf{m}$ is changed. Similarly, an event $\mathsf{tick_h}$ is generated to trigger changes of $\mathsf{h}$.

According to the single-step semantics of statecharts used here, changing seconds and minutes takes two steps of the statechart. For example, the tuple of variables $\langle \mathsf{h}, \mathsf{m}, \mathsf{s} \rangle$ will have successively the values $\langle 0, 0, 59 \rangle$, $\langle 0, 0, 0 \rangle$, and $\langle 0, 1, 0 \rangle$. In the step from $\langle 0, 0, 59 \rangle$ to $\langle 0, 0, 0 \rangle$ the event $\mathsf{tick_m}$ is generated, but this can be used only in the next step.

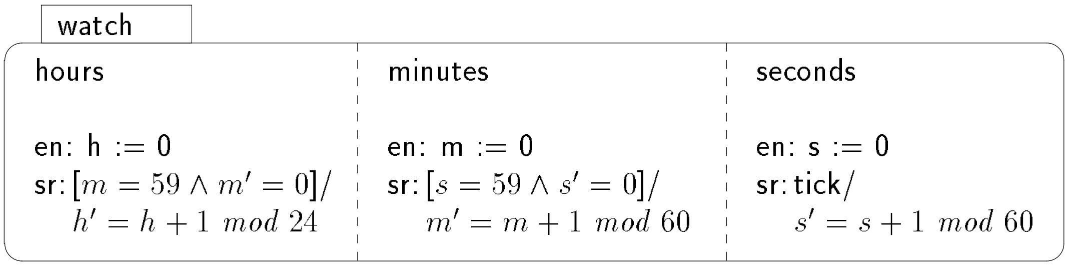



Figure 6.2: Clock: TLA-actions as triggering conditions

On the contrary, in the microstep-semantics [Pnueli and Shalev, 1991] events generated in one step can be used to trigger further non-conflicting compound transitions in the same step. Therefore the event $\mathsf{tick_m}$ becomes effective in the same step, i.e. the value of $\langle \mathsf{h}, \mathsf{m}, \mathsf{s} \rangle$ changes in one step from $\langle 0, 0, 59 \rangle$ to $\langle 0, 1, 0 \rangle$.

Similarly, in the superstep-semantics [Harel and Naamad, 1996] the sequence consisting of the steps from $\langle 0, 0, 59 \rangle$ to $\langle 0, 0, 0 \rangle$ and from $\langle 0, 0, 0 \rangle$ to $\langle 0, 1, 0 \rangle$ is considered as one (super-)step. Remember that by the first step the event $\mathsf{tick_m}$ is generated. This is not visible at the end of the superstep.

In [Lamport, 1996] two different descriptions of a clock are given. In the first one only one TLA-action is used to change minutes and seconds simultaneously. In the other one two distinct TLA-actions are used, in this case there is a similar sequence of TLA-states of $\langle \mathsf{h}, \mathsf{m}, \mathsf{s} \rangle$ as described above. It has to be expressed explicitly that two actions should be taken in synchrony to prevent such intermediate states.

By using TLA-actions instead of predicates as enabling conditions of transitions it becomes possible to express this synchronisation. This can be done as shown in the statechart in Figure 6.2. This is quite similar to the statechart in Figure 6.1, but instead of using events to trigger the static reactions in the states minutes and hours, these are triggered by the *changes* of the variables s and m, respectively.

Note that it is not sufficient to use the conditions $[m = 59]$ and $[s = 59]$ as condition expressions of the static reactions of the states hours and minutes, respectively. Otherwise, these static reactions could be enabled without that the static reaction of the state seconds is enabled.

### 6.1.2 Semantics

The semantics of TLA-actions as triggering conditions is quite simple to define. In this case, each of the predicates $trigger_t$, $event.occurred$, $enabled.transition$, and $executable.transition$ becomes a TLA-action and their declarations have to be changed accordingly.

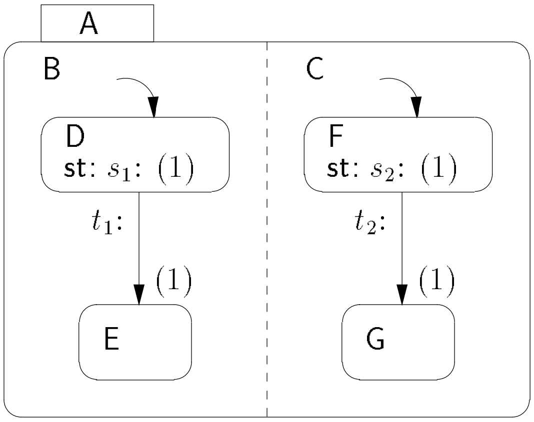



Figure 6.3: Compound transitions to be synchronised

### 6.1.3 Example: Synchronisation of Compound Transitions

In the previous example TLA-actions have been used as triggering conditions to express synchronicity of static reactions in an simple and explicit way. In this section this is considered in more detail by a second example. It is shown, that in contrast to the simplicity of the previous example it is also possible to write statecharts with TLA-actions as triggering conditions which are no longer easy to comprehend.

In this second example the mutual synchronisation of two compound transitions in a step is considered. No data variables are used in it. Starting from the statechart in Figure 6.3 it shall be expressed, that the compound transitions consisting of the transition segments $t_1$ and $t_2$, respectively, can only be taken together. After taking the default transitions the states D and F are active in this statechart. In subsequent *Execute.step*-steps there is a non-deterministic choice between taking the compound transitions consisting of one of the static transitions $s_1$ and $s_2$ and compound transitions $t_1$ and $t_2$, respectively. Each of these compound transitions has the same explicit priority.

In the statechart in Figure 6.4 the (empty) condition expressions of the transition segments have been extended by the TLA-actions $in(\mathsf{F}) \wedge in'(\mathsf{G})$ for $t_1$, and $in(\mathsf{D}) \wedge in'(\mathsf{E})$ for $t_2$. With these triggering conditions only the intended behaviour is possible, i.e. the states D and F can be left only together. But using TLA-actions as triggering conditions has its advantages as well as its disadvantages. On the one hand this statechart expresses the required property in an explicit and precise way. On the other hand one gets the following chain of reasoning to determine whether the compound transition $\{t_1\}$ can be executed in a step: $\{t_1\}$ is enabled, if $\{t_2\}$ is executed in the step. This implies that $\{t_2\}$ is enabled in this step, implying itself that $\{t_1\}$ is executed. Hence $\{t_1\}$ is enabled if it is executed in this step. Such a cycle of causal dependencies makes it difficult to understand a statechart. This is not a problem when considering the question, whether an *Execute.step*-step between two specific statuses of the statechart is possible, because the TLA-actions $in(\mathsf{F}) \wedge in'(\mathsf{G})$

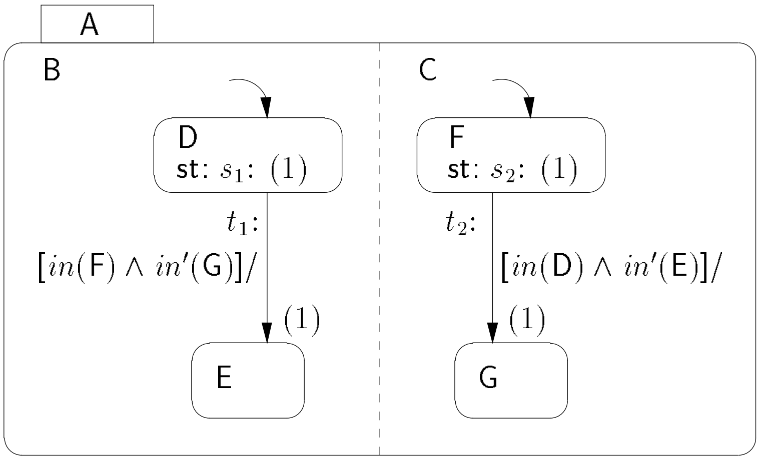



Figure 6.4: Synchronising compound transitions

and $in(\mathsf{D}) \wedge in'(\mathsf{E})$ can be evaluated and there is no cyclic reasoning on the level of the semantics.

In the example in Section 6.1.1 this problem with causality does not occur, because the synchronisation of the static reactions is not symmetrical. Changing the minutes depends on changing the seconds, but not vice versa.

To express the synchronisation between the two compound transitions in this example using statecharts only, one can use additional events as it was done in Section 6.1.1, but again one has the problems with intermediate steps. Another possibility is to split the transition segments $t_1$ and $t_2$ into smaller parts and to connect these with and-connectors as shown in Figure 6.5.[1] A first disadvantage of this approach is, that the structural priority of the compound transitions is changed. The structural priority determinant of the combined compound transition is the state A, of the compound transitions $\{t_1\}$ and $\{t_2\}$ in figures 6.3 and 6.4 these are the states B and C, respectively. As a consequence, the combined compound transition has a higher overall priority than the static compound transitions $\{s_1\}$ and $\{s_2\}$, namely D and F, respectively, and the non-determinism between the combined compound transition and the static ones is lost. A second disadvantage is that the statecharts are cluttered up by the additional transition segments and become more difficult to comprehend. This problem can be circumvented to some amount by diagram connectors, but then it is difficult to comprehend which compound transitions are synchronised at all.

Such an interaction between synchronisation by TLA-actions as triggering conditions and priorities already exists without such an explicit use of and-connectors to combine compound transitions. To investigate this in more detail, we change the

[1]Assigning the static reactions of the statechart in Figure 6.1 to the state watch there is also the possibility to express the synchronisation without further events. This makes it necessary to combine the event and action expressions of the static reactions on a higher level of the hierarchy of states.

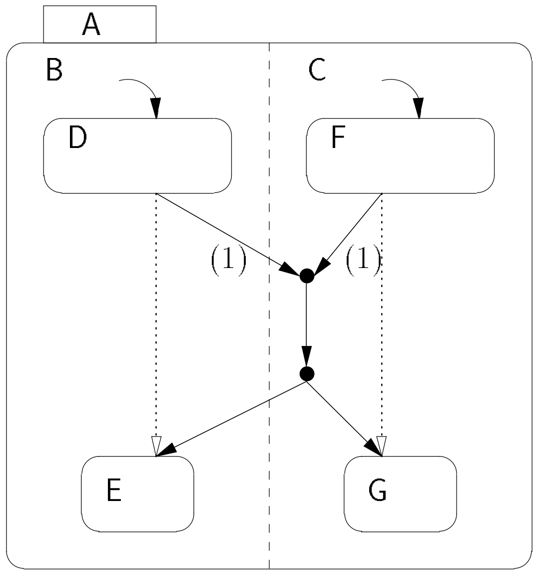



Figure 6.5: Synchronising compound transitions with and-connectors

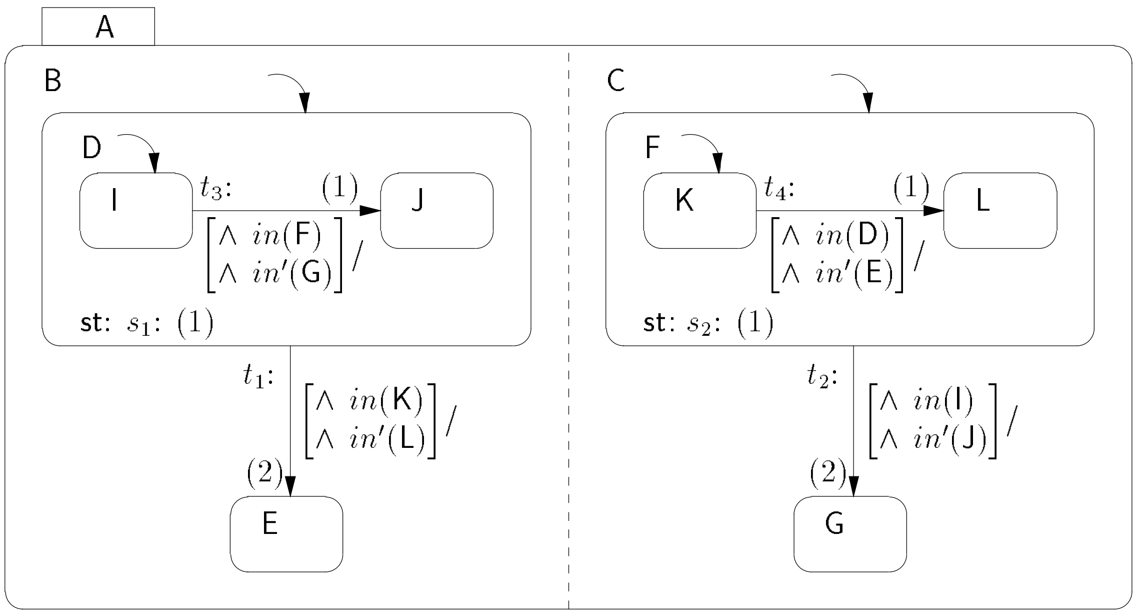



Figure 6.6: Synchronisation versus priority

---

example by refining the states D and F, cf. Figure 6.6. In this statechart compound transitions are synchronised which are comparable with other compound transitions w.r.t. their priority. The compound transition $\{t_1\}$ has higher priority than $\{t_3\}$, and $\{t_2\}$ has higher priority than $\{t_4\}$. In addition, the compound transitions $\{t_1\}$ and $\{t_2\}$ are not in conflict. But due to their triggering conditions the compound transitions $\{t_1\}$ and $\{t_2\}$ cannot be taken together. For example, the compound transition $\{t_1\}$ can be executed only if $\{t_4\}$ is executed. This implies that $\{t_2\}$ cannot be executed because $\{t_4\}$ and $\{t_2\}$ are in conflict. But it is possible to execute the pairs of compound transitions $\{t_1\}$, $\{t_4\}$ and $\{t_2\}$, $\{t_3\}$ together, respectively.

To execute none of the compound transitions $\{t_1\}$ to $\{t_4\}$ is not possible here, because we changed in contrast to the previous examples the explicit priorities of the transition segments $t_1$ and $t_2$ compared with the static transitions $s_1$ and $s_2$.

### 6.1.4 Conclusions

From the examples presented so far, the following conclusions can be drawn. TLA-actions as triggering conditions can be used to express synchronous changes of data variables as well as taking compound transitions in synchrony. By this kind of synchronisation neither the structural nor the explicit priority of compound transitions is changed, which can happen when connecting transition segments by and-connectors. But that priority is not changed appears on the other hand as a disadvantage if there exist further compound transitions which are in conflict to one of the synchronised compound transitions. If their priority w.r.t. the other synchronised compound transitions is different or incompatible, the statecharts become difficult to comprehend because the sets of compound transitions which can be executed in one step are constructed in a non-intuitive way. Thereby it is suggested to use TLA-actions as triggering conditions only, when all synchronised compound transitions can be compared with the same result with other compound transitions w.r.t their priority. Even more, we propose to use TLA-actions as triggering conditions not at all because we will show later on that synchronisation can be achieved by the second kind of extending statecharts with TLA, too. This second extension does not interfere such closely with the definition of a step of the statechart.

Note also, that adding such a TLA-action as a triggering condition does not imply that the behaviour of the statechart without this triggering conditions is restricted. As can be seen from the examples, there are new runs, which did not exist without these conditions. This is due to the fact that the corresponding TLA-actions are used in the definition of the predicate *enabled.transition*.

## 6.2 Temporal Constraints: Examples

In this and the subsequent sections we present a second extension of statecharts with TLA. The intention is to restrict those parts of the runs of a statechart in which a state of the statechart is active. The restrictions are expressed as TLA-formulae. We call these TLA-formulae *temporal constraints*.

In this approach we get less interference between the semantics of a statechart with and without such temporal constraints than by the use of TLA-actions as triggering conditions. But note, that there will also be some problems when such temporal constraints are used in an unrestricted way.

We start with some motivating examples and point out shortly some of these problems. The examples are intended to show how a description with associated TLA-formulae could look like. Typical properties of reactive systems are expressed in these examples. It should be clear, that it is possible to express further properties as well as that it is possible to express some of these properties already in statecharts without temporal constraints, but to our opinion in a less clear manner.

### 6.2.1 Synchronisation of Compound Transitions

As a first example we reconsider synchronisation between compound transitions of parallel substates. In the statechart in Figure 6.3, page 102, there are two parallel substates $\mathsf{B}$ and $\mathsf{C}$ of the and-state $\mathsf{A}$. The states $\mathsf{B}$ and $\mathsf{C}$ have, besides others, the substates $\mathsf{D}$ and $\mathsf{F}$, respectively. In one step of the statechart there is a non-deterministic choice between keeping these states active or leaving them. To express that in one step both or none of them are left, we can associate the TLA-formula $\Box[in(\mathsf{D}) \wedge in'(\mathsf{E}) \;\Leftrightarrow\; in(\mathsf{F}) \wedge in'(\mathsf{G})]_{\langle\rangle}$ with the state $\mathsf{A}$. Taking the semantics of the unrestricted statechart into account the same property can be expressed as an invariant: The formula $\Box(in(\mathsf{D}) \;\Leftrightarrow\; in(\mathsf{F}))$ associated with the state $\mathsf{A}$ restricts the set of runs of the statechart in the same way.

There is another approach by which it can be avoided to use next-state relations as constraints instead of invariants. Assume that the events $\mathsf{de}$ and $\mathsf{fg}$ are generated when taking the transitions from $\mathsf{D}$ to $\mathsf{E}$ and from $\mathsf{F}$ to $\mathsf{G}$, respectively. The restriction can be expressed now as the invariant $\Box(\mathsf{de} \in E_{exists} \;\Leftrightarrow\; \mathsf{fg} \in E_{exists})$. This replacement of restrictions on TLA-actions with invariants will be considered in more detail in the example in Section 7.8.

### 6.2.2 Synchronisation on Data

Let us assume, that we have an or-state $\mathsf{A}$ and a finite set $\{\mathsf{B}_i \mid 1 \le i \le n\}$ of descendants of it. Let us assume further that there are predicates $p_i(w)$, $1 \le i \le n$, on $w$ such that $\neg(p_i \wedge p_j)$, for $i \ne j$, and $\bigvee_{1 \le i \le n} p_i(w)$ are valid. The property we want to express is, that the state $\mathsf{B}_i$ is active if and only if $p_i$ is valid. Furthermore, a state $\mathsf{B}_i$ should be left only, when the corresponding $p_i$ is no longer valid.

This property is expressed as a restriction of the statechart in Figure 6.7, in which the situation is given for $n = 3$. In this statechart there is an or-connector, which is the source of transitions segments to each $\mathsf{B}_i$ and vice versa the target of transition segments from each $\mathsf{B}_i$ and the default transition $d$. Neither event nor action expressions are associated with any of these transition segments. All transition segments and static transitions have the same priority number, viz 1. Hence in each step of the statechart there is a non-deterministic choice between leaving a state $\mathsf{B}_i$ and

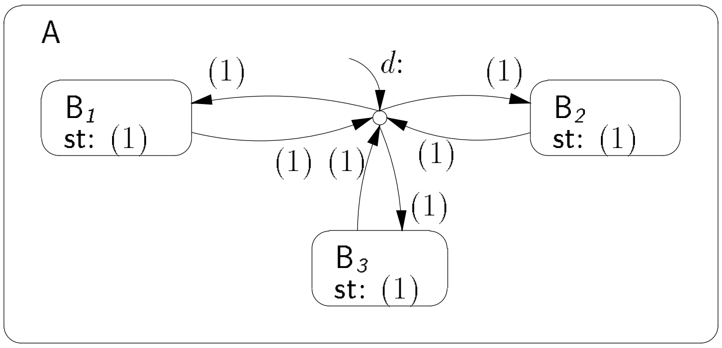



Figure 6.7: Synchronising activity of states with data

entering another (or the same) state $\mathsf{B}_j$ and staying in the state $\mathsf{B}_i$ without leaving and reentering it.

From now on static transitions with empty condition and action will not be mentioned explicitly by the keyword **st**:, only their priority number is retained in the statecharts, cf. Figures 6.8 and 6.10

The required property can be considered as consisting of two separate ones. One expressing the relationship between $\mathsf{B}_i$ and $p_i$ and one expressing that no unnecessary compound transitions are taken. The first property can be expressed as an invariant $\Box\, stay$ with

$$stay \equiv \bigwedge_{1 \le i \le n} \begin{array}{l} in(\mathsf{B}_i) \\ \Rightarrow p_i(w) \end{array}$$

The second property can be expressed as the next-state relation $\Box[Leave]_{\langle\rangle}$ with

$$Leave \equiv \bigwedge_{1 \le i \le n} \begin{array}{l} Exited(\mathsf{B}_i) \\ \Rightarrow \neg p_i'(w) \end{array}$$

Both TLA-formulae are associated with the state $\mathsf{A}$. It is also possible to split *stay* and associate $\Box p_i(w)$ with the state $\mathsf{B}_i$.

The temporal constraints given before differ in two aspects given from the property stated informally. The first difference is, that we have assumed here implicitly, that the state $\mathsf{A}$ is always active and therefore always one of the states $\mathsf{B}_i$ is active, too. If the statechart in Figure 6.7 is part of a larger one, e.g. those in Figure 6.8, then this assumption is no longer valid. Hence, the weaker temporal property

$$\begin{array}{l} \wedge\ \Box in(\mathsf{A})\ \Rightarrow\ stay \\ \wedge\ \Box[Staying(\mathsf{A})\ \Rightarrow\ Leave] \end{array}$$

has been described actually. At second, let us assume, that $w$ is changed by an *external. change*-step which is a $(p_i(w) \wedge \neg p_i'(w))$-step. Due to the semantics of statecharts as described by the TLA-program *statechart* this cannot be an *Execute. step*-step. Hence, the state $\mathsf{B}_i$ should remain active until the next *Execute. step*-step, in which it will be left.

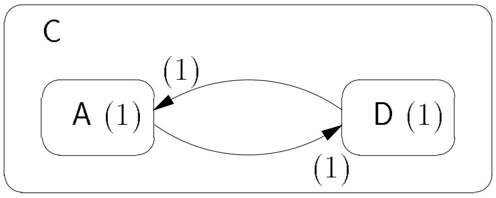



Figure 6.8: Embedding the statechart in Figure 6.7 in another statechart

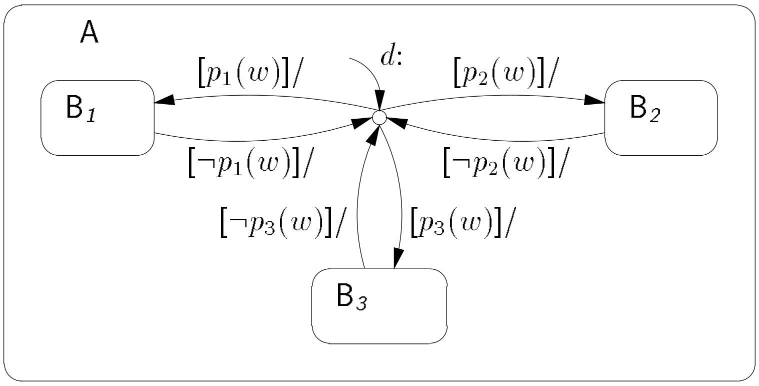



Figure 6.9: Synchronising activity of states with data, using statecharts only

---

This weakening of the properties will be made explicit when defining the precise semantics of associating a temporal constraint with a state in the Sections 6.4 and the following ones.

This behaviour can be described too by using statecharts without temporal constraints. In this case both properties have to be described together. The statechart is quite similar to the one in Figure 6.7, but the static transitions are removed. The priority numbers can be removed, too. Transition segments with a state $\mathsf{B}_i$ as target have $p_i(w)$ as triggering condition, those leaving one of these states have $\neg p_i(w)$ as triggering conditions, see Figure 6.9. Hence a state $\mathsf{B}_i$ can be entered only when $p_i(w)$ is valid and due to the eagerness of the semantics of statecharts the state $\mathsf{B}_i$ does not remain active if $p_i(w)$ is no longer valid.

### 6.2.3 Fairness

Let us assume, that we want to express for the statechart in Figure 6.8 that none of the states A and D remains active forever. This means, that we have a fairness constraint on the steps leaving these states. This can be expressed by associating the fairness constraints $SF_{\langle\rangle}\,(\,in(\mathsf{A}) \wedge \neg\, in'(\mathsf{A}))$ and $SF_{\langle\rangle}\,(\,in(\mathsf{D}) \wedge \neg\, in'(\mathsf{D}))$ with the state C. Another possibility is to associate the leadsto-formulae $in(\mathsf{A}) \leadsto in(\mathsf{D})$ and $in(\mathsf{D}) \leadsto in(\mathsf{A})$ with the state C. Here the property $in(\mathsf{C}) \;\Rightarrow\; (in(\mathsf{A}) \;\Leftrightarrow\; \neg\, in(\mathsf{D}))$ of statecharts is used.

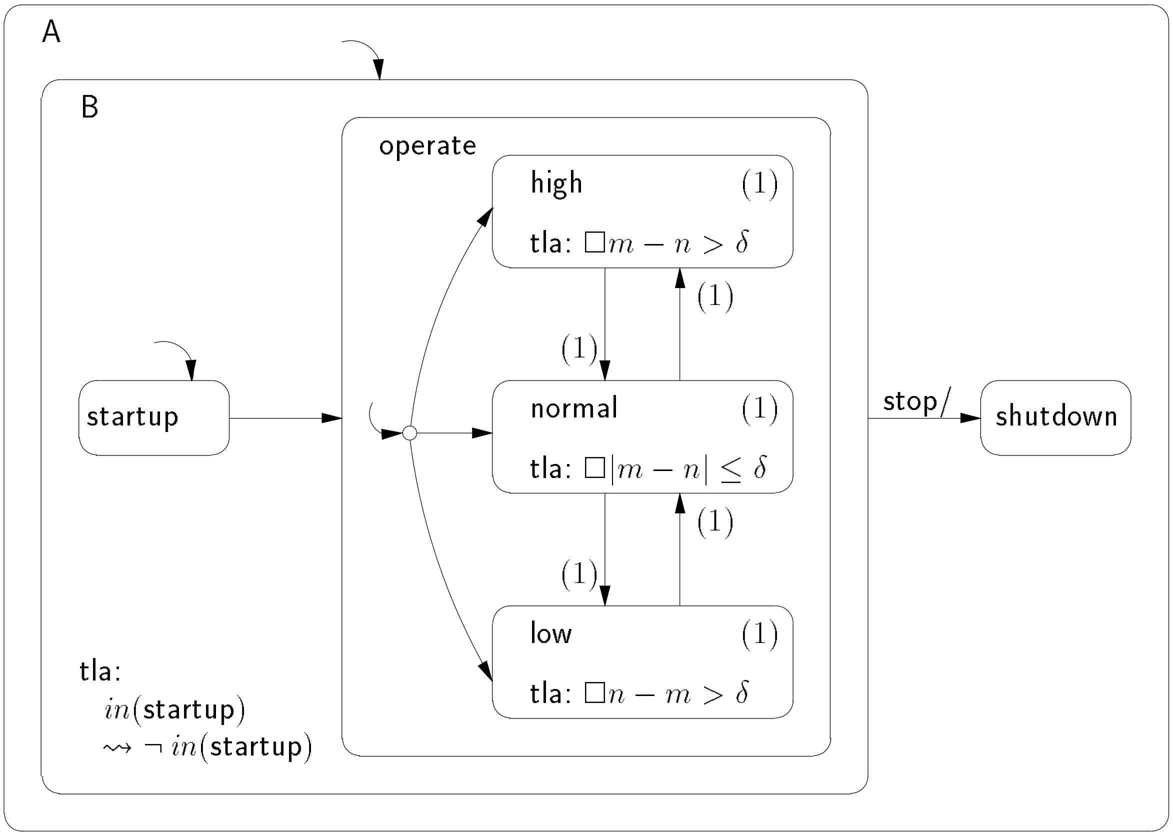



Figure 6.10: Very simple control system

### 6.2.4 Hierarchy

We present here an example in which different properties are expressed by associating TLA-formulae to states on different levels. In this example the modes and the transitions between them of a very simple control system are defined. The statechart in Figure 6.10 contains three states — corresponding to modes of the system — called startup, operate, and shutdown and three substates low, high, and normal of operate. The keyword tla: in some of the states is used to indicate the TLA-formulae associated with a state. We want to express the following properties, stated informally as

1. Certain transitions are not possible at all, e.g. it is not possible to change from low to high in one step.

2. Eventually the state startup is left.

3. The state shutdown is entered on occurrence of the event stop.

4. Whether the system is in one of the modes corresponding low, high, or normal,

depends on whether the difference of the two variables $m$ and $n$ is larger than a predefined value $\delta$ and which one is the larger one.

For the purpose of illustration the properties are partially expressed using only statecharts and are partially expressed by temporal constraints associated with states. In detail

Property 1 is expressed by the static structure of the statechart, i.e. the set of transition segments. For example there are no transition segments from high to low and vice versa.

Property 2 is expressed by a leadsto-formula associated with the state B.

Property 3 is expressed by using stop as triggering event of the corresponding transition segment.

Alternatively it could be expressed by adding a static transition with priority number 1 to the state B and by associating the next-state relation

$$\Box \begin{bmatrix} \wedge\ \mathsf{stop} \in E_{exists} \\ \wedge\ in(\mathsf{B}) \\ \Rightarrow \neg\, in'(\mathsf{B}) \end{bmatrix}_{step.taken}$$

with the state A. By allowing stuttering w.r.t. *step.taken* this action restricts *Execute.step*-steps of the statechart only.

Property 4 is expressed by invariants associated with the states low, high, and normal in the same way as in the example in Section 6.2.2.

## 6.3 Problems of Temporal Constraints

The intention of associating temporal constraints is to exclude some runs from the behaviour of a statechart. In this section we give two examples where temporal constraints lead to somehow unexpected results, i.e. more runs than intended are discarded. Therefore the use of such constraints has to be restricted somehow.

### 6.3.1 Restriction of Inputs

Let us consider the statechart in Figure 6.11. The event e is considered as an input of the statechart. But in each run, in which e occurs after initializing the statechart the state D is reached eventually ($in(\mathsf{C}) \leadsto in(\mathsf{D})$). But this is excluded by the invariant $\Box\neg\, in(\mathsf{D})$. Hence, we have described a set of runs in which the event e does not occur after initializing the statechart. This means that the environment cannot generate the event e, this can be seen as an unintended restriction of the environment.

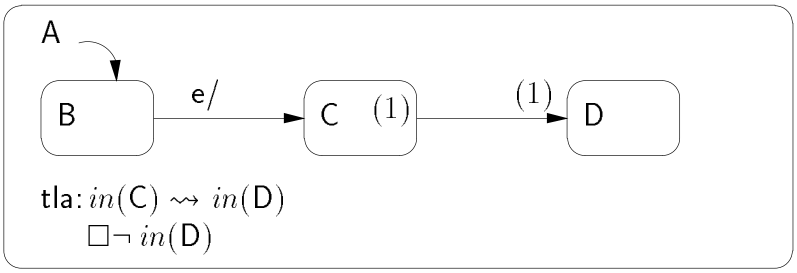



Figure 6.11: Restricting inputs

### 6.3.2 Influencing Ancestor States

Let us reconsider the example of Section 6.2.2, Figure 6.7, and let us assume that $\bigvee_{1\leq i\leq n} p_i(w)$ is not valid and that $\mathsf{A}$ is a substate of another state $\mathsf{C}$ as described in Figure 6.8. If there is a valuation of $w$ such that none of the $p_i$ is valid, then none of the $\mathsf{B}_i$ can be active. This implies, due to the semantics of statecharts, that $\mathsf{A}$ cannot be active either. In the same *Execute.step*-step in which $w$ is changed to this valuation the state $\mathsf{A}$ is left and $\mathsf{D}$ is entered.

Furthermore, the behaviour of this statechart with these temporal constraints contains only runs in which one of the predicates $p_i$ becomes active eventually. Only then it is possible to initialise the statechart and to enter the states $\mathsf{A}$ and $\mathsf{C}$ at all.

## 6.4 Temporal Constraints of a State

By associating a temporal constraint $F$ with a statecharts-state $A$, written $A \rhd F$, we want to express that the subsequences of runs in which $A$ is active satisfy $F$. In the previous sections we have presented some examples of using such temporal constraints and we have already pointed out some problems. We define here the semantics of associating temporal constraints with states.

The semantics of $A \rhd F$ is given by a corresponding TLA-formula, denoted $[\![ A \rhd F ]\!]$, which will be conjoined to the TLA-program *statecharts*. By the definition of $[\![ A \rhd F ]\!]$ it is ensured that $F$ restricts only subsequences of runs in which $A$ is active. The following points have to be taken into account:

- Only *Execute.step*-steps are restricted by temporal constraints. It is clear that *status*-stuttering steps do not need to be restricted. But *External.change*-steps do not have to be restricted either. Restricting a step can imply that the set of active states changes. But this is not possible in an *External.change*-step.

- For a temporal constraint expressing a liveness-property it does not make sense to restrict finite subsequences in which $A$ is active. There have to be infinitely many *Execute. step*-steps in which $A$ remains active. It can be distinguished further between the existence of infinitely many such steps at all and the existence of one subsequence containing infinitely many such steps.

Therefore we define two different semantics of $A \rhd F$. The first one restricts all subsequences of runs in which $A$ is active, it is called the *safety-semantics* of $A \rhd F$. The second one restricts only runs which contain infinitely many *Execute. step*-steps in which $A$ is active, it is called the *liveness-semantics* of $A \rhd F$. The liveness-semantics comes in two variants according to the distinction made above.

When associating a TLA-program with a state $A$ we apply the safety-semantics to its initial condition and its next-state relation. One variant of the liveness-semantics is applied to the fairness-formulae of the TLA-program. Furthermore, we allow to associate leadsto-formulae with a state, the liveness-semantics is applied to them, too.

### 6.4.1 Safety-Semantics

Because we apply the safety-semantics to initial conditions, i.e. to predicates, and to next-state relations, we define the safety-semantics for such TLA-formulae only. Invariants are used quite often, therefore we introduce an abbreviation to associate them with a state. But we do not define the safety-semantics for arbitrary TLA-formulae expressing safety-properties.

The TLA-formulae expressing the safety-semantics of associating a TLA-formula $F$ with a state $A$ is denoted by $[\![ A \rhd F ]\!]_s$. *Execute. step*-steps in which $A$ remains active are denoted by the TLA-action *Execute. stay*$(A)$:

$$\begin{aligned}
&\mathit{Execute.\,stay}\colon S\\
&\mathit{Execute.\,stay}(A) \equiv \land\ \mathit{Execute.\,step}\\
&\qquad\qquad\qquad\qquad\ \ \land\ \mathit{Staying}(A)
\end{aligned}$$

**Predicates** Let $p$ be a predicate, then $A \rhd p$ means, that each *Execute. step*-step beginning a subsequence of *Execute. stay*$(A)$-steps is a $p'$-step, too. Hence we call $p$ an *initial condition* of the statechart-state $A$. As an example consider the run $\sigma$ in Figure 6.12. In this run the statechart-state $A$ becomes active in the *Execute. step*-step from the TLA-state $\sigma_i$ to $\sigma_{i+1}$ and remains so until $\sigma_k$. $A$ becomes active again in the *Execute. step*-step from $\sigma_l$ to $\sigma_{l+1}$, and remains active until $\sigma_m$. In the *Execute. step*-step from $\sigma_m$ to $\sigma_{m+1}$ the state $A$ is left and reentered immediately.

$[\![ A \rhd p ]\!]_s$ has to be defined such that $p$ is valid in the states $\sigma_{i+1}$, $\sigma_{l+1}$, and $\sigma_{m+1}$. I.e. each *Execute. step*-step in which $A$ becomes active has to be a $p'$-step, too. *Execute. stay*$(A)$-steps do not need to be $p'$-steps. We define:

$$[\![ A \rhd p ]\!]_s \equiv \Box[\mathit{Entered}(A) \ \Rightarrow\ p']_{step.taken}$$

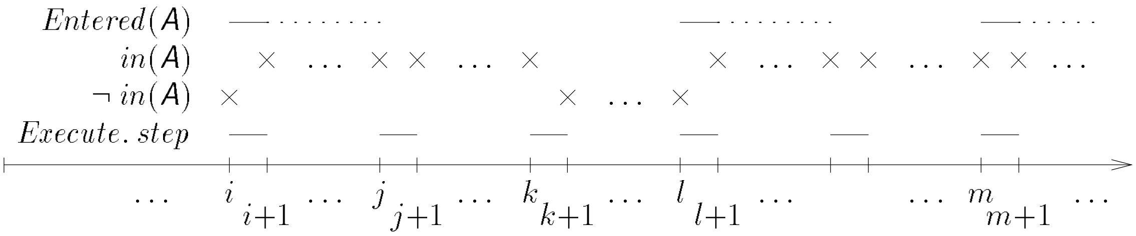



Figure 6.12: Subsequences of a run

This can be expressed in an equivalent way as:

$$[\![ A \triangleright p ]\!]_s \;\Leftrightarrow\; \Box \begin{bmatrix} \wedge\ \mathit{Execute.\,step} \\ \wedge\ \mathit{Entered}(A) \\ \Rightarrow p' \end{bmatrix}_{status}$$

It is assumed here, that in the initial status of the statechart the state $A$ is not active. This assumption is justified by the definition of $\mathit{initial.\,status}$, namely $S_{active} = \emptyset$. Otherwise one would have to conjoin the predicate $in(A) \;\Rightarrow\; p$ to $[\![ A \triangleright p ]\!]_s$.

**Next-State Relations** Using the same example as before, the run $\sigma$ shall satisfy $[\![ A \triangleright \Box[\mathcal{N}]_f ]\!]_s$ for a TLA-action $\mathcal{N}$ and a state function $f$, if each $\mathit{Execute.\,stay}(A)$-step is also a $[\mathcal{N}]_f$-step. Hence $\langle \sigma_j, \sigma_{j+1} \rangle$ in Figure 6.12 has to satisfy $[\mathcal{N}]_f$. But it is not required that $\langle \sigma_m, \sigma_{m+1} \rangle$ satisfies $[\mathcal{N}]_f$ because $A$ is left in this step, although it is reentered immediately again. $\Box[\mathcal{N}]_f$ is called the *next-state relation* of the statechart-state $A$.

Similarly as before we define

$$[\![ A \triangleright \Box[\mathcal{N}]_f ]\!]_s \equiv \Box[\mathit{Staying}(A) \;\Rightarrow\; [\mathcal{N}]_f]_{step.taken}$$

which can be rewritten in an equivalent way, too:

$$[\![ A \triangleright \Box[\mathcal{N}]_f ]\!]_s \;\Leftrightarrow\; \Box \left[ \mathit{Execute.\,stay}(A) \;\Rightarrow\; [\mathcal{N}]_f \right]_{status}$$

Usually, in TLA the state function $f$ in a next-state relation $\Box[\mathcal{N}]_f$ is a tuple of variables containing those occurring in $\mathcal{N}$. Here, we use often the empty tuple $\langle\rangle$ as state function. This expresses that each $\mathit{Execute.\,stay}(A)$-step is actually an $\mathcal{N}$-step.

**Conjunctions** The safety-semantics of a conjunction of temporal constraints is defined simply as the conjunction of the safety-semantics of each temporal constraint.

$$[\![ A \triangleright F \wedge G ]\!]_s \equiv \begin{array}{l} \wedge\ [\![ A \triangleright F ]\!]_s \\ \wedge\ [\![ A \triangleright G ]\!]_s \end{array}$$

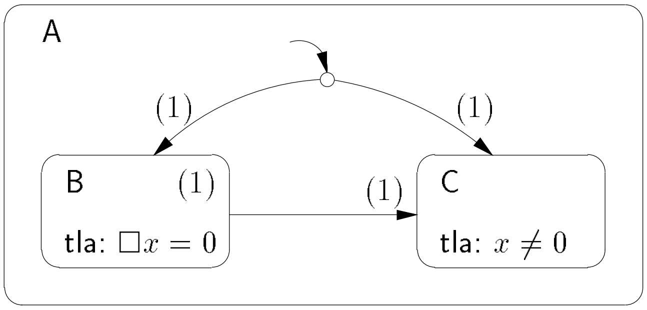



Figure 6.13: Detecting violation of an invariant

**Invariants** We want to express often that a predicate $p$ is valid as long as a state $A$ is active, i.e. that $p$ is an *invariant* of $A$. This can be described by associating the temporal constraint $p \wedge \Box[p \wedge p']_{\langle\rangle}$ with $A$. We define $[\![\, \mathsf{A} \rhd \Box p \,]\!]_s$ as an abbreviation of $[\![\, A \rhd p \wedge \Box[p \wedge p']_{\langle\rangle} \,]\!]_s$. Because we do not apply $[\![\ \ ]\!]_s$ to arbitrary safety-formulae this use of $\Box$ does not collide with the temporal operator $\Box$ of TLA. This definition can be expanded and rewritten to an TLA-formulae as follows:

$$
\begin{array}{rcll}
[\![\, A \rhd \Box p \,]\!]_s & \equiv & [\![\, A \rhd p \wedge \Box[p \wedge p']_{\langle\rangle} \,]\!]_s & \\
& \equiv & \wedge\ [\![\, A \rhd p \,]\!]_s & (\text{Def. } [\![\ \ ]\!]_s) \\
& & \wedge\ [\![\, A \rhd \Box[p \wedge p']_{\langle\rangle} \,]\!]_s & \\
& \equiv & \wedge\ \Box[\mathit{Entered}(A) \ \Rightarrow\ p']_{step.taken} & (\text{Def. } [\![\ \ ]\!]_s) \\
& & \wedge\ \Box[\mathit{Staying}(A) \ \Rightarrow\ [p \wedge p']_{\langle\rangle}]_{step.taken} & \\
& \Leftrightarrow & \Box \begin{bmatrix} \wedge\ \mathit{Entered}(A) \ \Rightarrow\ p' \\ \wedge\ \mathit{Staying}(A) \ \Rightarrow\ [p \wedge p']_{\langle\rangle} \end{bmatrix}_{step.taken} & \begin{array}{l}(\text{TLA STL5}^2, \\ \text{prop. logic})\end{array}
\end{array}
$$

Note that the invariant defined this way is weaker than $\Box(in(\mathsf{B}) \ \Rightarrow\ p)$. It is possible that an invariant $\Box p$ is invalidated by an *External. change*-step. In this case the state $A$ is left when taking the next *Execute. step*-step. This delay cannot be avoided because otherwise it would have to be possible to execute an *External. change*-step together with an *Execute. step*-step. But this is not possible according to the definition of the TLA-program *statechart*.

As an example consider the statechart in Figure 6.13. Assume that $x$ is initially 0, then entering the state A implies that the state B is entered, too. B remains active until $x$ changes its value. In this statechart this can be done only by an *External. change*-step with $x = 0 \wedge x' \neq 0$ and $in(\mathsf{B}) \wedge in'(\mathsf{B})$. I.e. there is a TLA-state which satisfies $in(\mathsf{B}) \wedge x \neq 0$. This can be followed by *status*-stuttering steps or by further *External. change*-steps. Only when taking the next *Execute. step*-step this violation of the invariant is recognised. In this step B becomes inactive.

[2]The proofrules of TLA are shown in Appendix A

**Properties of $[\![\ ]\!]_s$** Concluding the definition of $[\![\ ]\!]_s$ we show that $[\![\ ]\!]_s$ distributes properly over conjunctions of atomic TLA-formulae. This means that associating the conjunction of atomic TLA-formulae with a state restricts the behaviour in the same way as associating both constraints with this state.

**Lemma 6.1** *Let $p, q$ be two predicates, then the following is valid:*

$$[\![ \mathsf{A} \rhd (p \wedge q) ]\!]_s \Leftrightarrow \begin{array}{l} \wedge\ [\![ \mathsf{A} \rhd p ]\!]_s \\ \wedge\ [\![ \mathsf{A} \rhd q ]\!]_s \end{array}$$

**Proof:**

$$\begin{array}{rlll} [\![ \mathsf{A} \rhd (p \wedge q) ]\!]_s & \equiv & \Box[\mathit{Entered}(\mathsf{A}) \ \Rightarrow\ (p \wedge q)']_{step.taken} & (\text{Def. } [\![\ ]\!]_s) \\ & \Leftrightarrow & \Box \begin{array}{l} \wedge\ [\mathit{Entered}(\mathsf{A}) \ \Rightarrow\ p']_{step.taken} \\ \wedge\ [\mathit{Entered}(\mathsf{A}) \ \Rightarrow\ q']_{step.taken} \end{array} & (\text{prop. logic}) \\ & \Leftrightarrow & \begin{array}{l} \wedge\ \Box[\mathit{Entered}(\mathsf{A}) \ \Rightarrow\ p']_{step.taken} \\ \wedge\ \Box[\mathit{Entered}(\mathsf{A}) \ \Rightarrow\ q']_{step.taken} \end{array} & (\text{TLA, STL5}) \\ & \equiv & \begin{array}{l} \wedge\ [\![ \mathsf{A} \rhd p ]\!]_s \\ \wedge\ [\![ \mathsf{A} \rhd q ]\!]_s \end{array} & (\text{Def. } [\![\ ]\!]_s) \end{array}$$

**qed**

Similarly it can be shown

**Lemma 6.2** *Let $\mathcal{N}, \mathcal{M}$ be TLA-actions and $f$ and $g$ be state functions, then the following is valid:*

$$[\![ \mathsf{A} \rhd \Box([\mathcal{N}]_f \wedge [\mathcal{M}]_g) ]\!]_s \Leftrightarrow [\![ \mathsf{A} \rhd \Box[\mathcal{N}]_f \wedge \Box[\mathcal{M}]_g ]\!]_s$$

### 6.4.2 Liveness-Semantics

Liveness formulae have to be interpreted essentially over infinite sequences of TLA-states. Hence, when associating a liveness-formula $F$ with a statecharts-state $\mathsf{A}$, only those runs of the statechart should be constrained which contain infinitely many *Execute. stay*($\mathsf{A}$)-steps. Note, that we do not require, that there is actually an infinite number of such steps. If this is required, it should be stated explicitly by a temporal constraint associated with an ancestor of $\mathsf{A}$.

To define the liveness-semantics of $\mathsf{A} \rhd F$ two further problems have to be considered. On the one hand, $F$ has to be restricted appropriately to *Execute. stay*($\mathsf{A}$)-steps. On the other hand, it can be asked whether runs with an infinite number of *Execute. stay*($\mathsf{A}$)-steps have to be constrained or whether only those with an infinite sequence, where each *Execute. step*-step is an *Execute. stay*($\mathsf{A}$)-step, have to be constrained.

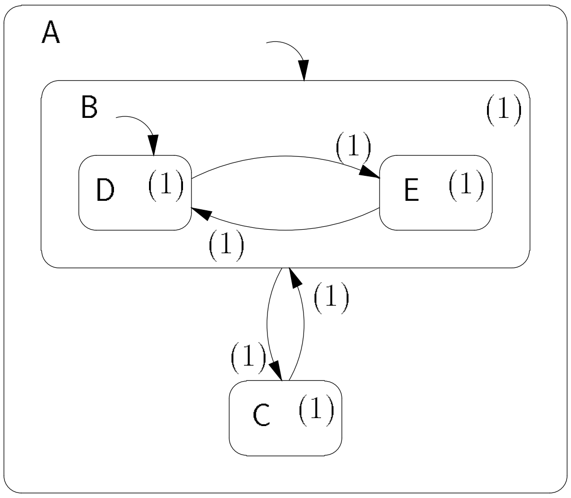



Figure 6.14: Infinitely many *Execute. stay*(B)-steps versus an infinite sequence of *Execute. stay*(B)-steps

---

To explain the first problem, let us assume that the temporal constraint $SF_f(\mathcal{N})$ is associated with the state $A$, where $\mathcal{N}$ is a TLA-action and $f$ a state function. The intended meaning is not only that we have strong fairness for $\mathcal{N}$, provided an infinite number of *Execute. stay*($A$)-steps occurs, but strong fairness for the TLA-action $\mathcal{N} \wedge$ *Execute. stay*($A$). This means that if there is an infinite number of *Execute. stay*($A$)-steps which are also $\mathcal{N}$-steps, then eventually there is an *Execute. step*-step which is also an $\mathcal{N}$-step and in which $A$ remains active. We are not interested in $\mathcal{N}$-steps, when $A$ is not active or which are not *Execute. step*-steps. It cannot be assumed in general that $\mathcal{N}$ implies *Execute. stay*($A$). Therefore it would be appropriate to enforce by the definition of the semantics that only sequences of *Execute. stay*($A$)-steps are restricted. But this cannot be achieved, the problem is that a predicate *Enabled* $\mathcal{N}$, for a TLA-action $\mathcal{N}$, cannot be strengthened to *Enabled*($\mathcal{N} \wedge$ *Execute. stay*($A$)). Note that *Enabled*($\mathcal{N} \wedge$ *Execute. stay*($A$)) is not equivalent to (*Enabled* $\mathcal{N}$) $\wedge$ *Enabled Execute. stay*($A$). As a consequence, the liveness-semantics will have some nice properties only if the condition ((*Enabled* $\mathcal{N}$) $\wedge$ *Enabled Execute. stay*($A$)) $\Rightarrow$ *Enabled*($\mathcal{N} \wedge$ *Execute. stay*($A$)) is valid.

To avoid to require more than the property stated above we use for each kind of TLA-formulae an appropriate premise when defining the semantics of associating it with a state.

For the second problem we consider the statechart in Figure 6.14. This statechart is satisfied by a run $\sigma_1$ in which every second *Execute. step*-step describes a transition from B to C or vice versa, and in which E is never active. It is also satisfied by a run $\sigma_2$, in which B and D remain active forever and neither C nor E become active eventually. Note that these are only two possible runs, there are infinitely many other ones. Now let us assume we want to associate $F \equiv SF_{step.taken}(in(\mathsf{D}) \wedge \neg\, in'(\mathsf{D}))$

with the state $\mathsf{B}$.

It should be clear that the run $\sigma_2$ does not satisfy $\mathsf{B} \rhd F$, but what about the run $\sigma_1$? Because $F$ expresses strong fairness it can be argued that it is not necessary that there is an infinite sequence of steps in which each $Execute.\,step$-step is an $Execute.\,stay(\mathsf{B})$-step and in which an $in(\mathsf{D}) \wedge \neg\, in'(\mathsf{D}))$-step is enabled. Note that there are infinitely many $in(\mathsf{D}) \wedge \neg\, in'(\mathsf{D}))$-steps, but these are not $Execute.\,stay(\mathsf{B})$-steps. Hence $\sigma_1$ should not satisfy $\mathsf{B} \rhd F$ either.

Now consider $G \equiv WF_{step.taken}\,(\,in(\mathsf{D}) \wedge \neg\, in'(\mathsf{D}))$. Here it can be argued the other way round and therefore $\sigma_1$ should satisfy $\mathsf{B} \rhd G$.

Hence, for weak and strong fairness there are preferences whether there is an infinite number or an infinite sequence of $Execute.\,stay(\mathsf{B})$-steps. The situation is less clear for leadsto-formulae, consider e.g. $in(\mathsf{D}) \leadsto in'(\mathsf{E})$. The question is here, whether $\mathsf{E}$ has to become active in the same subsequence of $Execute.\,stay(\mathsf{B})$-steps in which $\mathsf{D}$ became active or not. If yes, only runs with an infinite sequence of $Execute.\,stay(\mathsf{B})$-steps should be constrained.

Because there are different assumptions on the existence of an infinite sequence of such steps for the different fairness properties or there is not even a preference on these assumptions in case of leadsto-formulae we make this distinction explicit by providing two versions of the operator $\rhd$: $\mathsf{A} \rhd_{\Box\Diamond} F$ to assume infinitely many $Execute.\,stay(\mathsf{A})$-steps, and $\mathsf{A} \rhd_{\Diamond\Box} F$ to assume an infinite sequence of $Execute.\,step$-steps, where each one is also an $Execute.\,stay(\mathsf{A})$-step and in which $Execute.\,step$-steps are enabled continuously. To shorten subsequent definitions we define:

$$
\begin{array}{lcl}
Execute.\,stay_{\Box\Diamond}(\mathsf{A}) & \equiv & \Box\Diamond[\,Execute.\,stay(\mathsf{A})]_{status} \\
Execute.\,stay_{\Diamond\Box}(\mathsf{A}) & \equiv & \wedge\ Execute.\,stay_{\Box\Diamond}(\mathsf{A}) \\
 & & \wedge\ \Diamond\wedge\ \Box[Execute.\,step\ \Rightarrow\ Staying(\mathsf{A})]_{status} \\
 & & \qquad \wedge\ \Box\, Enabled\ Execute.\,step
\end{array}
$$

We use $*+$ as a variable ranging over $\{\Box\Diamond, \Diamond\Box\}$.

The liveness-semantics of associating a temporal constraint $F$ with a state $\mathsf{A}$ is denoted by $[\![\,\mathsf{A} \rhd F\,]\!]_l$. $[\![\,\mathsf{A} \rhd F\,]\!]_l$ is defined by recursion on the structure of $F$. The assumptions on the existence of infinite sequences of $Execute.\,stay(\mathsf{A})$-steps are introduced when considering TLA-formulae with $\Box$ as outermost symbol. To ensure that $[\![\,\mathsf{A} \rhd F\,]\!]_l$ distributes properly over negation we use two auxiliary operators $[\![\,\mathsf{A} \rhd F\,]\!]_l^+$ and $[\![\,\mathsf{A} \rhd F\,]\!]_l^-$.

**Predicates** The liveness-semantics is not applied to predicates associated with a state directly. This occurs only as one base case of the recursive definition of $[\![\ \ ]\!]_l$. In contrast to the safety-semantics it is therefore not necessary to distinguish between the cases that $\mathsf{A}$ becomes or remains active. We require that $p$ is valid and an *Execute. stay*($A$)-step is enabled:

$$\begin{aligned}\llbracket A \rhd_{*+} p \rrbracket_l^+ &\equiv (\mathit{Enabled\ Execute.\,stay}(A)) \land p \\ \llbracket A \rhd_{*+} p \rrbracket_l^- &\equiv (\mathit{Enabled\ Execute.\,stay}(A)) \land \neg p\end{aligned}$$

Note that *Enabled Execute. stay*($A$) implies that $A$ is active, i.e. $in(A)$ is valid.

**Next-State Relations** As for predicates, the liveness-semantics is not applied to next-state relations directly. For a TLA-action $\mathcal{N}$ and a state function $f$ the liveness-semantics of associating the next-state relation $\Box[\mathcal{N}]_f$ expresses that each *Execute. step*-step is an *Execute. stay*($A$)-step and an $[\mathcal{N}]_f$-step:

$$\begin{aligned}\llbracket A \rhd_{*+} \Box[\mathcal{N}]_f \rrbracket_l^+ &\equiv \Box[\, \mathit{Staying}(A) \land [\mathcal{N}]_f]_{step.taken} \\ \llbracket A \rhd_{*+} \Box[\mathcal{N}]_f \rrbracket_l^- &\equiv \Diamond\langle\, \mathit{Staying}(A) \land \langle \neg\mathcal{N} \rangle_f \rangle_{step.taken}\end{aligned}$$

**Logical Operators** For two TLA-formulae $F$ and $G$ the semantics of associating $F \land G$ and $\neg F$ with a state $A$ is defined as usual. The assumption on the number of *Execute. stay*($A$)-steps is propagated without change to the recursive cases:

$$\begin{aligned}\llbracket A \rhd_{*+} \neg F \rrbracket_l^+ &\equiv \llbracket A \rhd_{*+} F \rrbracket_l^- \\ \llbracket A \rhd_{*+} \neg F \rrbracket_l^- &\equiv \llbracket A \rhd_{*+} F \rrbracket_l^+\end{aligned}$$

$$\begin{aligned}\llbracket A \rhd_{*+} F \land G \rrbracket_l^+ &\equiv \land\ \llbracket A \rhd_{*+} F \rrbracket_l^+ \\ &\phantom{\equiv}\ \land\ \llbracket A \rhd_{*+} G \rrbracket_l^+ \\ \llbracket A \rhd_{*+} F \land G \rrbracket_l^- &\equiv \lor\ \llbracket A \rhd_{*+} F \rrbracket_l^- \\ &\phantom{\equiv}\ \lor\ \llbracket A \rhd_{*+} G \rrbracket_l^-\end{aligned}$$

It is easy to see, that $\llbracket\ \rrbracket_l^+$ and $\llbracket\ \rrbracket_l^-$ distribute properly over $\lor$. By expanding the definitions of $\lor$, $\llbracket\ \rrbracket_l^+$ and $\llbracket\ \rrbracket_l^-$ we get the following equivalences:

$$\begin{aligned}\llbracket A \rhd_{*+} F \lor G \rrbracket_l^+ &\Leftrightarrow \lor\ \llbracket A \rhd_{*+} F \rrbracket_l^+ \\ &\phantom{\Leftrightarrow}\ \lor\ \llbracket A \rhd_{*+} G \rrbracket_l^+ \\ \llbracket A \rhd_{*+} F \lor G \rrbracket_l^- &\Leftrightarrow \land\ \llbracket A \rhd_{*+} F \rrbracket_l^- \\ &\phantom{\Leftrightarrow}\ \land\ \llbracket A \rhd_{*+} G \rrbracket_l^-\end{aligned}$$

**Temporal Operators** As the last case we define the liveness-semantics of $A \rhd \Box F$ for a TLA-formula $F$. For this temporarily quantified formula we employ the assumption on the number of *Execute. stay*($A$)-steps:

$$\begin{aligned}\llbracket A \rhd_{*+} \Box F \rrbracket_l^+ &\equiv \mathit{Execute.\,stay}_{*+}(A) \\ &\phantom{\equiv}\ \Rightarrow \Box\llbracket A \rhd_{*+} F \rrbracket_l^+ \\ \llbracket A \rhd_{*+} \Box F \rrbracket_l^- &\equiv \mathit{Execute.\,stay}_{*+}(A) \\ &\phantom{\equiv}\ \Rightarrow \Diamond\llbracket A \rhd_{*+} F \rrbracket_l^-\end{aligned}$$

The definition of associating $\Diamond F$ with a state can be derived by using the equivalence $\Diamond F \equiv \neg\Box\neg F$. By expanding the definitions of $\Diamond$, $[\![\ ]\!]_l^+$ and $[\![\ ]\!]_l^-$ we get the following equivalences:

$$\begin{array}{lcl} [\![ \mathsf{A} \rhd_{*+} \Diamond F ]\!]_l^+ & \Leftrightarrow & \mathit{Execute.\,stay}_{*+}(\mathsf{A}) \\ & & \Rightarrow \Diamond[\![ \mathsf{A} \rhd_{*+} F ]\!]_l^+ \\ [\![ \mathsf{A} \rhd_{*+} \Diamond F ]\!]_l^- & \Leftrightarrow & \mathit{Execute.\,stay}_{*+}(\mathsf{A}) \\ & & \Rightarrow \Box[\![ \mathsf{A} \rhd_{*+} F ]\!]_l^- \end{array}$$

**Liveness-Semantics** Finally we define $[\![\_]\!]_l$ by relating this operator to the auxiliary one defining the positive case:

$$[\![ \mathsf{A} \rhd_{*+} F ]\!]_l \equiv [\![ \mathsf{A} \rhd_{*+} F ]\!]_l^+$$

**Properties of $[\![\ ]\!]_l$** In the following we will show under which conditions fairness- and leadsto-formulae distribute properly over the definition of $[\![\ ]\!]_l$.

Let $\mathcal{N}$ be a TLA-action for which fairness is required, then it has to be assumed that if an $\mathcal{N}$-step and an $\mathit{Execute.\,stay}(\mathsf{A})$-step are enabled, then an $\mathit{Execute.\,step}$-step is also enabled which is an $\mathcal{N}$- as well as an $\mathit{Execute.\,stay}(\mathsf{A})$-step. This assumption is satisfied for example when $\mathcal{N}$ implies that a compound transition is taken and that the scope of this compound transition is below $\mathsf{A}$.

Fairness of $(\mathcal{N} \wedge \mathit{Execute.\,stay}(\mathsf{A}))$ is implied by associating $\mathit{WF}_f(\mathcal{N})$ with $\mathsf{A}$ in the case that infinitely many $\mathit{Execute.\,stay}(\mathsf{A})$-steps are taken, i.e. if $\mathit{Execute.\,stay}_{\Box\Diamond}$ is taken as a premise. In the case of taking $\mathit{Execute.\,stay}_{\Diamond\Box}$ as a premise, associating a fairness-formula to $\mathsf{A}$ is equivalent to weak fairness of $(\mathcal{N} \wedge \mathit{Execute.\,stay}(\mathsf{A}))$-steps. This difference occurs also if strong fairness instead of weak fairness is considered.

**Theorem 6.1** *Let $\mathcal{N}$ be a TLA-action, $f$ be a state function, and $\mathsf{A}$ be a state of a statechart such that*

$$\begin{array}{l} \wedge\ \mathit{Enabled}\ \langle \mathcal{N} \rangle_f \\ \wedge\ \mathit{Enabled}\ \mathit{Execute.\,stay}(\mathsf{A}) \\ \Rightarrow \mathit{Enabled} \wedge\ \langle \mathcal{N} \rangle_f \\ \qquad\qquad\quad \wedge\ \mathit{Execute.\,stay}(\mathsf{A}) \end{array}$$

*is valid. Then the following formulae are valid, too.*

1. $[\![ \mathsf{A} \rhd_{\Box\Diamond} \mathit{WF}_f(\mathcal{N}) ]\!]_l \Rightarrow \mathit{Execute.\,stay}_{\Box\Diamond}(\mathsf{A}) \Rightarrow \mathit{WF}_f(\mathcal{N} \wedge \mathit{Execute.\,stay}(\mathsf{A}))$

2. $[\![ \mathsf{A} \rhd_{\Diamond\Box} \mathit{WF}_f(\mathcal{N}) ]\!]_l \Leftrightarrow \mathit{Execute.\,stay}_{\Diamond\Box}(\mathsf{A}) \Rightarrow \mathit{WF}_f(\mathcal{N} \wedge \mathit{Execute.\,stay}(\mathsf{A}))$

3. $[\![ \mathsf{A} \rhd_{\Box\Diamond} SF_f(\mathcal{N}) ]\!]_l \;\Rightarrow\; \begin{array}{l} Execute.\,stay_{\Box\Diamond}(\mathsf{A}) \\ \Rightarrow SF_f(\mathcal{N} \wedge Execute.\,stay(\mathsf{A})) \end{array}$

4. $[\![ \mathsf{A} \rhd_{\Diamond\Box} SF_f(\mathcal{N}) ]\!]_l \;\Leftrightarrow\; \begin{array}{l} Execute.\,stay_{\Diamond\Box}(\mathsf{A}) \\ \Rightarrow SF_f(\mathcal{N} \wedge Execute.\,stay(\mathsf{A})) \end{array}$

**Proof:** The auxiliary lemmata A.4 and A.5 and their proofs are presented in Appendix A.4.

(Theorem 6.1.1)

$$
\begin{array}{lll}
 & [\![ \mathsf{A} \rhd_{*+} WF_f(\mathcal{N}) ]\!]_l & (1) \\
\equiv & \vee\ [\![ \mathsf{A} \rhd_{*+} \Box\Diamond \langle \mathcal{N} \rangle_f ]\!]_l^+ & (2,\ \text{Def. } [\![\ ]\!]_l,\ WF\,()) \\
 & \vee\ [\![ \mathsf{A} \rhd_{*+} \Box\Diamond\neg\, Enabled\, \langle \mathcal{N} \rangle_f ]\!]_l^+ & \\
\equiv & \vee\ \ Execute.\,stay_{*+}(\mathsf{A}) & (3,\ \text{Def. } [\![\ ]\!]_l^+) \\
 & \quad \Rightarrow \Box[\![ \mathsf{A} \rhd_{*+} \Diamond \langle \mathcal{N} \rangle_f ]\!]_l^+ & \\
 & \vee\ \ Execute.\,stay_{*+}(\mathsf{A}) & \\
 & \quad \Rightarrow \Box[\![ \mathsf{A} \rhd_{*+} \Diamond\neg\, Enabled\, \langle \mathcal{N} \rangle_f ]\!]_l^+ & \\
\Rightarrow & \vee\ \ Execute.\,stay_{*+}(\mathsf{A}) & (4,\ \text{Lem. A.4, A.5.1}) \\
 & \quad \Rightarrow \Box\Diamond \langle \mathcal{N} \wedge Execute.\,stay(\mathsf{A}) \rangle_f & \\
 & \vee\ \ Execute.\,stay_{*+}(\mathsf{A}) & \\
 & \quad \Rightarrow \Box\Diamond\neg\, Enabled\, \langle \mathcal{N} \wedge Execute.\,stay(\mathsf{A}) \rangle_f & \\
\Leftrightarrow & \ \ Execute.\,stay_{*+}(\mathsf{A}) & (5,\ \text{prop. logic}) \\
 & \ \ \Rightarrow \vee\ \Box\Diamond \langle \mathcal{N} \wedge Execute.\,stay(\mathsf{A}) \rangle_f & \\
 & \quad\ \ \vee\ \Box\Diamond\neg\, Enabled\, \langle \mathcal{N} \wedge Execute.\,stay(\mathsf{A}) \rangle_f & \\
\equiv & \ \ Execute.\,stay_{*+}(\mathsf{A}) & (6,\ \text{Def. } WF\,()) \\
 & \ \ \Rightarrow WF_f(\mathcal{N} \wedge Execute.\,stay(\mathsf{A})) &
\end{array}
$$

(Theorem 6.1.2) The proof of this case is similar, except that instead of Lemma A.5.1 we use Lemma A.5.2 to conclude the desired equivalence between the formulae (3) and (4).

(Theorems 6.1.3, 6.1.4) The proof is similar as in the case of weak fairness. We use the lemmata A.5.3 and A.5.4 instead of A.5.1 and A.5.2, respectively.

**qed**

For leadsto-formulae a property similar to Theorem 6.1 can be shown only for $p \leadsto q$, where $p$ and $q$ are predicates. Here it makes no difference whether $\rhd_{\Box\Diamond}$ or $\rhd_{\Diamond\Box}$ is considered.

**Theorem 6.2** *Let $p$ and $q$ be two predicates and $\mathsf{A}$ a state of a statechart. Then the following implication is valid*

$$
[\![ \mathsf{A} \rhd_{*+} p \leadsto q ]\!]_l \;\Rightarrow\; \begin{array}{l} Execute.\,stay_{*+}(\mathsf{A}) \\ \Rightarrow\ p \wedge Enabled\ Execute.\,stay(\mathsf{A}) \\ \qquad \leadsto q \wedge Enabled\ Execute.\,stay(\mathsf{A}) \end{array}
$$

**Proof:**

$$
\begin{array}{rcll}
[\![\, A \rhd_{*+} p \leadsto q \,]\!]_l & \equiv & \mathit{Execute.\,stay}_{*+}(A) & \text{(Def. } [\![\ ]\!]_l, \leadsto) \\
 & & \Rightarrow \Box [\![\, A \rhd_{*+} p \;\Rightarrow\; \Diamond q \,]\!]_l^+ & \\
 & \Rightarrow & \mathit{Execute.\,stay}_{*+}(A) & \text{(Lem. A.6)} \\
 & & \Rightarrow \Box\ [\![\, A \rhd_{*+} p \,]\!]_l^+ & \\
 & & \qquad \Rightarrow [\![\, A \rhd_{*+} \Diamond q \,]\!]_l^+ & \\
 & \equiv & \mathit{Execute.\,stay}_{*+}(A) & \text{(Def. } [\![\ ]\!]_l^+) \\
 & & \Rightarrow \Box\ [\![\, A \rhd_{*+} p \,]\!]_l^+ & \\
 & & \qquad \Rightarrow \mathit{Execute.\,stay}_{*+}(A) & \\
 & & \qquad\qquad \Rightarrow \Diamond [\![\, A \rhd_{*+} q \,]\!]_l^+ & \\
 & \Rightarrow & \mathit{Execute.\,stay}_{*+}(A) & \text{(temp. logic)} \\
 & & \Rightarrow \Box\ [\![\, A \rhd_{*+} p \,]\!]_l^+ & \\
 & & \qquad \Rightarrow \Diamond [\![\, A \rhd_{*+} q \,]\!]_l^+ & \\
 & \Leftrightarrow & \mathit{Execute.\,stay}_{*+}(A) & \text{(Def. } [\![\ ]\!]_l, \leadsto) \\
 & & \Rightarrow\ p \wedge \mathit{Enabled\ Execute.\,stay}(A) & \\
 & & \quad \leadsto q \wedge \mathit{Enabled\ Execute.\,stay}(A) &
\end{array}
$$

**qed**

## 6.5 Temporal Constraints of a Statechart

In the previous sections we defined and investigated the safety- and the liveness-semantics of associating temporal constraints with single states. We define here the meaning of associating a TLA-program with a state and the behaviour of a statechart with temporal constraints.

Let $G$ be a TLA-program, i.e. a TLA-formulae of the form $\mathit{init} \wedge \Box[\mathcal{N}]_f \wedge F$, where $\mathit{init}$ is a predicate, $\mathcal{N}$ a TLA-action, $f$ a state function, and $F$ a finite conjunction of weak and strong fairness-formulae. In addition we allow that $F$ contains finitely many leadsto-formulae $p \leadsto q$ for predicates $p$ and $q$. If $G$ does not contain neither fairness- nor leadsto-formulae, then $G$ is called a *safety-constraint*. If $G$ contains only fairness-formulae, then it is called a *fairness-constraint*. Analogously, if $G$ contains only leadsto-formulae, it is called a *leadsto-constraint*. Fairness- and leadsto-constraints are called *liveness-constraints*.

All actions, predicates, and state functions of $G$ can refer to the status of the statechart except the variable *step. taken*. To define the meaning of associating $G$ as a temporal constraint with a state $A$ we apply the safety-semantics to the initial condition and the next-state relation of $G$ and the liveness-semantics to the fairness and leadsto-formulae.

$$
\begin{array}{rl}
[\![\, A \rhd_{*+} G \,]\!] \equiv & \wedge\ [\![\, A \rhd \mathit{init} \wedge \Box[\mathcal{N}]_f \,]\!]_s \\
 & \wedge\ [\![\, A \rhd_{*+} F \,]\!]_l
\end{array}
$$

For a state $A$ of a statechart *constraints*$(A)$ denotes a set of TLA-programs. By $pr(c, A)$ for a temporal constraint $c$ we denote the premise on $Execute.\,stay(A)$-steps when associating $c$ with $A$. We use *true* as a default constraint. Let $\mathcal{TLA}$ denote the set of TLA-formulae, then:

$$constraints\colon S \rightarrow 2^{\mathcal{TLA}}$$

$$pr\colon \mathcal{TLA} \times S \rightarrow \{\Box\Diamond, \Diamond\Box\}$$

$$\begin{aligned}&constrained.\,statechart\colon\\ &constrained.\,statechart \equiv \wedge\ statechart\\ &\qquad\qquad\qquad\qquad\quad \wedge \bigwedge_{A \in S}\ \bigwedge_{c \in constraints(A)} [\![ A \rhd_{pr(c,A)} c ]\!]\end{aligned}$$

Because the meaning of associating temporal constraints with a state is conjoined to the TLA-program *statechart* the behaviour of an unconstrained statechart cannot be enlarged by associating temporal constraints with it. This is in contrast to the use of TLA-actions as triggering constraints, cf. Section 6.1. In the following we will use only temporal constraints.

Remember that a temporal constraint restricts those subsequences of runs in which the state with which the constraint is associated is active. It is quite easy, if not even trivial, to denote by a statechart these subsequences which are restricted by a temporal constraint. Therefore, statecharts has proven to be a good choice as the model-oriented part of our combination of description techniques.

In this and the preceding chapters we have presented a semantics of statechart by translating statecharts to equivalent TLA-formulae and we have defined syntax and semantics of associating temporal constraints with states of a statechart. In the next chapter we use our combination of statecharts and TLA to develop several descriptions of the behaviour of two reactive systems.

# Chapter 7

# Application of the Combination

After defining the semantics of associating TLA-formulae with states of a statechart we present two examples. The first one is a (simple) lift system. This example is considerably larger than the ones presented before, nevertheless the lift system is still a small system. For the lift system we present the different descriptions according to the general considerations in Section 1.2:

**Reference world (observer),** RW-O: (Section 7.1) A description of the behaviour of the uncontrolled plant as seen by an external observer of the lift system. This includes the designations of relevant terms of the problem domain "lift".

**Reference world (machine),** RW-M: (Section 7.2) A description of the plant to describe, how a controller can sense the state of the plant and how a controller can influence the state of the plant.

**Requirements,** REQ: (Section 7.3) A description of the required behaviour of the lift system when the plant is attached to a controller.

**Specification,** SPEC: (Section 7.4) A description of the required behaviour of the controller of the lift system in terms visible to the controller.

**Design of controller,** DES: (Section 7.5) A model of a controller, satisfying the requirements when attached to the plant. Two further variations of the controller are presented in Section 7.6. One is following the synchronous paradigm, the other one is written completely in TLA.

By this example we want to show that temporal constraints can be used in each of these descriptions. One can see also that it is quite convenient to produce REQ by adding temporal constraints to RW-O.

The second example, cf. Section 7.8, presents a part of the well-known case study 'Production Cell' [Lewerentz and Lindner, 1995]. We present the descriptions RW-O and REQ. This example is used to show how the synchronisation of actions can be expressed.

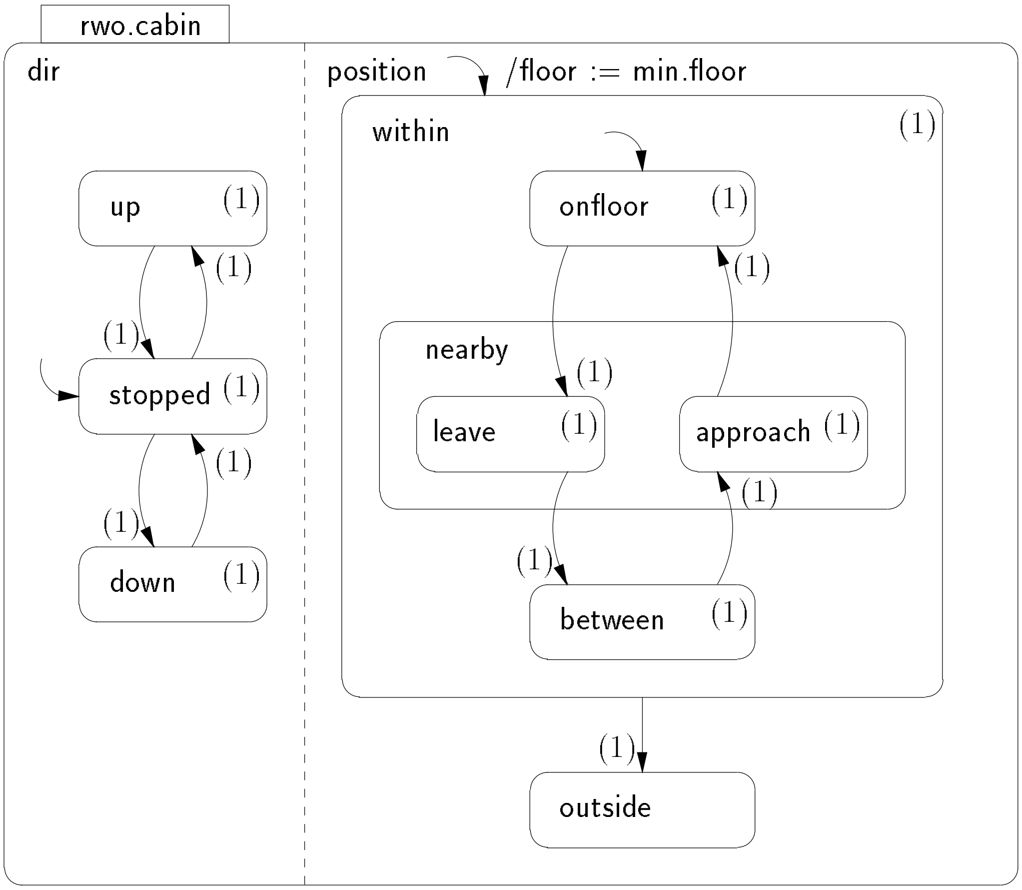



Figure 7.1: Rw-o: Direction and position of the lift cabin

## 7.1 Lift: Reference World (Observer)

The lift system we are considering in this example is a simple one: It has only one lift cabin. The floors of the building are represented by a subrange $[min.floor, max.floor]$ of $\mathbb{Z}$ with $min.floor < max.floor$. The variable $floor \in \mathbb{Z}$ is used to indicate the floor which is currently visited by the lift cabin or has been visited last, when the cabin is not exactly on a floor.

**Lift Cabin** The lift cabin can be moving upwards, downwards, or not at all, this is expressed by the state dir[1] and its substates in the statechart in Figure 7.1. The correspondence between the states and the direction of movement should be obvious. Note, that it is not possible to change in one step from moving upwards to downwards or vice versa, the lift cabin actually stops when changing its direction.

[1]Remember that we shorten the names of states: Instead of writing the complete sequence of names from the root to a specific state we leave out some of the names which are not necessary to identify the state. Which ones are left out depends on the context.

W.r.t. the position of the lift cabin it can be distinguished, whether the lift cabin is exactly on a floor (onfloor), nearby a floor (nearby), or between two floors, without being already nearby, (between). In the state nearby we distinguish whether the lift cabin is leaving a floor but is still nearby it (leave) and whether it is approaching a floor and is already nearby it (approach).

The variable floor is set when entering the state onfloor according to the current direction of movement. This change is expressed as an action expression of the transition segment from nearby.approach to onfloor:

$$\begin{array}{l}\mathsf{if\ in(dir.up)\ then\ floor := floor + 1\ end\ if;}\\ \mathsf{if\ in(dir.down)\ then\ floor := floor\ \text{-}\ 1\ end\ if}\end{array} \tag{7.1}$$

Additionally, we want to distinguish whether the lift cabin is within the intended range of floors, indicated by $min.floor$ and $max.floor$, or whether it is outside of this range. This is expressed by the substates of the state position. Reaching the state outside is considered an abnormal situation which cannot be resolved without external help, hence there is no transition segment leaving this state.

The states dir and position are not unrelated[2]. A first restriction of the behaviour is that the lift cabin is stopped only when it is exactly on a floor, i.e. $\Box(in(\mathsf{dir.stopped}) \Rightarrow in(\mathsf{position.onfloor}))$. Because the position of the lift cabin depends on its movement and not vice versa, this is expressed by associating the invariant $\Box\neg\, in(\mathsf{dir.stopped})$ with the states position.nearby and position.between:

$$\begin{array}{l}\mathsf{position.nearby} \rhd \Box\neg\, in(\mathsf{dir.stopped})\\ \mathsf{position.between} \rhd \Box\neg\, in(\mathsf{dir.stopped})\end{array} \tag{7.2}$$

The problem to which state a temporal constraint should be associated will be discussed in detail in Section 9.5. There it will be discussed, too, why these invariants should be associated with the states nearby and between instead of associating e.g. $\Box(\neg\, in(\mathsf{onfloor}) \Rightarrow \neg\, in(\mathsf{dir.stopped}))$ with the state within.

The position of the lift cabin is represented only coarsely here. By the invariant above it has already been expressed, that the lift cabin does not stop between floors. But it remains to express explicitly that the cabin does not travel forever between two floors. This liveness property can be expressed by a leadsto-formula:

$$\begin{array}{ll}\mathsf{within} \rhd^{3} & \neg\, in(\mathsf{onfloor})\\ & \leadsto in(\mathsf{onfloor})\end{array} \tag{7.3}$$

This property can be expressed in an equivalent way by fairness formulae, too.

$$\begin{array}{l}\mathsf{within} \rhd WF_{\langle\rangle}\,(\, in(\mathsf{between}) \wedge \neg\, in'(\mathsf{between}))\\ \mathsf{within} \rhd WF_{\langle\rangle}\,(\, in(\mathsf{nearby}) \wedge \neg\, in'(\mathsf{nearby}))\end{array} \tag{7.4}$$

[2]Note that this relation has to be realized itself by some kind of control system. We assume this control system to be existing already. Therefore we consider it as a part of the uncontrolled plant, but not of the controller to be developed.

[3]The indices $_{\Box\Diamond}$ and $_{\Diamond\Box}$ of $\rhd$ are omitted because there are only behaviours in which the state within is active forever, or the state outside becomes active. But in this case within is not entered again. As a consequence there is no difference between the two cases indicated by the indices.

The lift cabin passes a floor if it is not stopped. This behaviour can be expressed formally by associating an invariant with the state $\mathsf{onfloor}$. This invariant expresses that the lift cabin is stopped or that the state $\mathsf{onfloor}$ has just been entered. If the lift cabin is stopped this has to be done in the same step in which the state $\mathsf{onfloor}$ becomes active. This property is expressed by the subformula $\mathsf{en(dir.stopped)} \notin E_{exists}$ of the following equation.

$$\begin{aligned} \mathsf{onfloor} \rhd \Box \vee\ & in(\mathsf{dir.stopped}) \\ \vee\ & \mathsf{en(dir.stopped)} \notin E_{exists} \\ \vee\ & \mathsf{en(onfloor)} \in E_{exists} \end{aligned} \tag{7.5}$$

The distinction between the states $\mathsf{within}$ and $\mathsf{outside}$ is defined by an invariant expressing that the lift cabin does not move upwards, when the highest floor has already been reached or downwards, when the lowest one has already been reached:

$$\begin{aligned} \mathsf{within} \rhd \Box\neg \wedge\ & in(\mathsf{dir.up}) \\ \wedge\ & floor \geq max.floor \\ \mathsf{within} \rhd \Box\neg \wedge\ & in(\mathsf{dir.down}) \\ \wedge\ & floor \leq min.floor \end{aligned} \tag{7.6}$$

Vice versa it has to be prevented that the state $\mathsf{outside}$ is entered unnecessarily. We associate the negation of the previous invariants with the state $\mathsf{outside}$ as an initial condition:

$$\begin{aligned} \mathsf{outside} \rhd \vee\ \wedge\ & in(\mathsf{dir.up}) \\ \wedge\ & floor \geq max.floor \\ \vee\ \wedge\ & in(\mathsf{dir.down}) \\ \wedge\ & floor \leq min.floor \end{aligned} \tag{7.7}$$

For simplicity we have chosen the default transitions and their action expressions as shown in the statechart in Figure 7.1. In a real system, the default values of $\mathsf{position}$ have to be taken with care. A controller should not rely upon them. Instead it should try to get the current state of the system in an initialisation phase, which is not considered here.

**Doors** On each floor there is a door between the lift cabin and the floor itself. A door can be opened or closed, if it is opened it will close automatically. This behaviour is depicted by the statechart in Figure 7.2. The fairness constraint is associated with the state $\mathsf{door}_f$ instead of the state $\mathsf{opened}$ because there is no use in associating fairness on an action $\mathcal{N}$ with a state $\mathsf{A}$ such that taking $\mathcal{N}$ implies that $\mathsf{A}$ is left.

**Requests** The lift cabin can be requested to move to floor $f$ by pressing a button, this is modeled as occurrence of the event $\mathsf{press}_f$. We do not distinguish between pressing the button on a floor and pressing a button inside the lift cabin. The

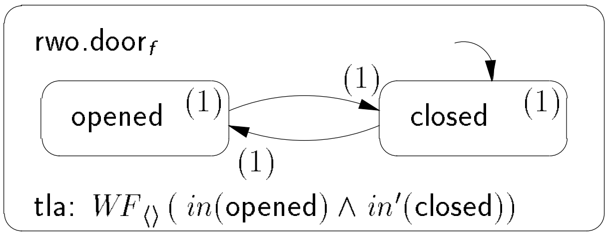



Figure 7.2: RW-O: Behaviour of a door

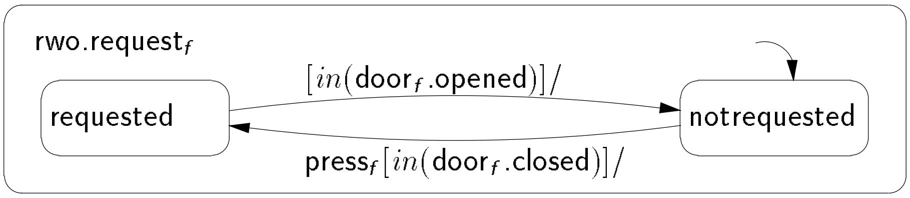



Figure 7.3: RW-O: Outstanding and granted requests

request is considered as granted when the door to the lift cabin opens.[4] This is depicted in the statechart in Figure 7.3. In this statechart it is also shown what happens when the event $\mathsf{press}_f$ is generated as long as the door is opened on the corresponding floor. This situation is not considered as a request.

**Plant** The complete plant consists of a statechart with a root-state plant and orthogonal substates $\mathsf{door}_f$, $\mathsf{request}_f$, for $min.floor \leq \mathsf{f} \leq max.floor$, and cabin, cf. Figure 7.4. It contains also the TLA-formulae of Equations 7.2, 7.3, 7.5, 7.6, and 7.7 associated with the corresponding states and the action expressions of Equation 7.1 of the transition segments from the state position.nearby.approach to position.onfloor.

**Environment** The environment of the plant is able to generate in each step an arbitrary set of the events $\mathsf{press}_f$, $min.floor \leq \mathsf{f} \leq max.floor$. This can be formalised by a static reaction of a state rwo.environment, cf. Figure 7.4. In this static reaction we use the extension of statecharts by non-deterministic TLA-actions, cf. Section 5.3.

**System** RW-O describes the behaviour of the uncontrolled plant and its environment. This is depicted by the statechart in Figure 7.4.

[4]This does not imply that the lift cabin is actually on this floor. This condition is a safety property in the sense of a safe behaviour of the plant and is considered as a requirement.

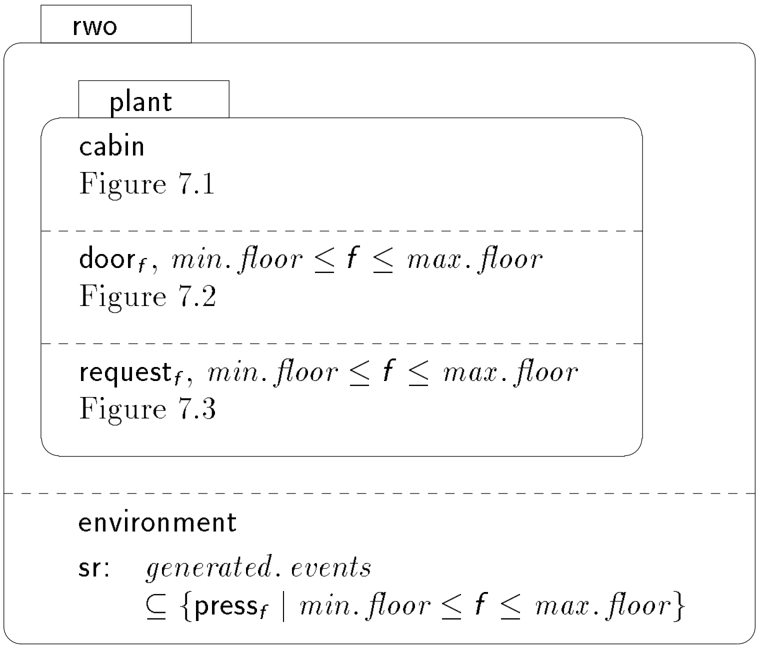



Figure 7.4: RW-O: Complete description

## 7.2 Lift: Reference World (Machine)

In RW-M it is expressed how a controller can influence the plant and how it is informed about the state or changes of the state of the plant. We take a copy of RW-O and extend and change it correspondingly. These changes are described in the following.

**Lift Cabin** To control movement of the lift cabin, a controller can command the lift to move up, to move down, and to stop by generating one of the events moveup, movedown, and stop, respectively. That the cabin has actually stopped is indicated to a controller by occurrence of the event en(stopped). This is the implicit event generated when entering the state dir.stopped. See the definition of *entering. events*.

The state stopped is left, when one of the events moveup or movedown occurs. If both occur there is a non-deterministic choice between taking the compound transitions leaving the state dir.stopped, cf. Figure 7.5. When one of the states dir.up and dir.down is active the events moveup and movedown are ignored.

To stop the lift cabin is more complicated. We assume, that it takes some time to stop the lift cabin physically, hence essentially some assumptions on the speed of the lift cabin or on reaction times of the controller have to be described. Here, we take the following assumptions on the physical realization of the lift cabin: When a stop event occurs within a specific time frame, then the lift cabin stops exactly on the next floor w.r.t. the direction of the lift cabin. We consider a step of the statechart

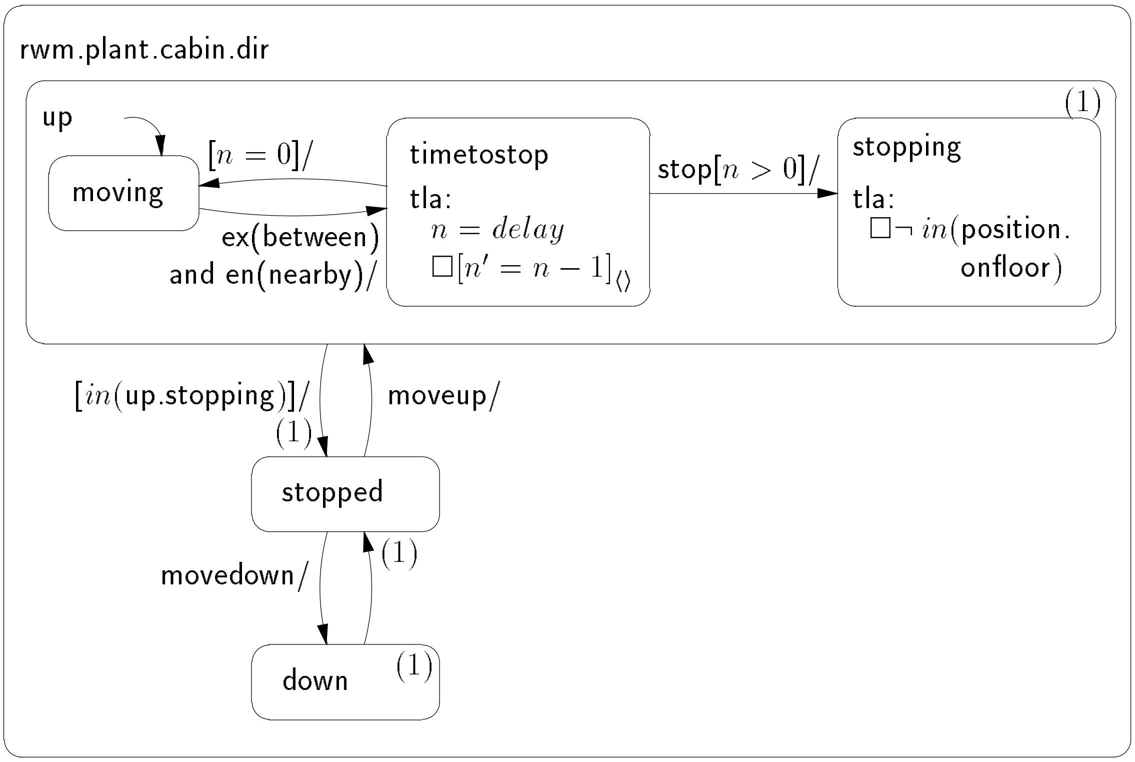



Figure 7.5: RW-M: Starting and stopping the lift cabin

as corresponding to one time unit. The time frame within which the plant reacts to stop events is determined by the starting point of the time frame, i.e. reaching the area nearby a floor, and its duration, described by a constant *delay*, with $delay \geq 0$ time units.

This behaviour can be described formally by refining the states dir.up and dir.down as shown for the state dir.up in Figure 7.5. When entering the state dir.up one waits until the lift cabin enters an area nearby a floor. When this happens, a timer is set to *delay*, counting backwards to 0. If this time passes without receiving a command to stop, i.e. that the event stop occurs, the state moving is reentered. It is no longer possible to stop the lift on the floor the cabin is currently approaching. If the event stop occurs within *delay* time units then the state stopping becomes active.

To prevent that the state up is left too early, the transition from up to stopped is guarded with the condition $in$(up.stopping). On the other hand, to ensure that it is left at all, the invariant $\Box\neg\ in$(onfloor) is associated with the state stopping. The fairness constraints associated with the state within ensure that a floor is reached eventually, which in turn implies that the lift cabin does not pass this floor.

The position of the lift cabin can be monitored because events $\text{nearby}_f$ and $\text{onfloor}_f$ are generated when the lift cabin leaves or enters an area nearby a floor and when it reaches exactly a floor. This can be expressed as action expressions of the transition

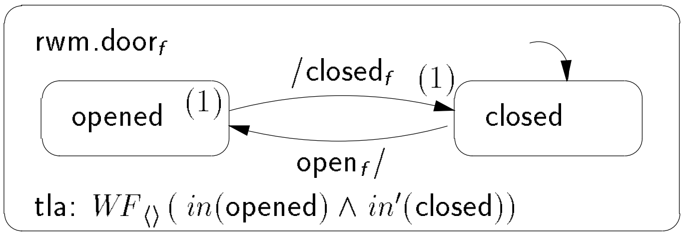



Figure 7.6: Rw-m: Opening and closing a door

segments from the substates of position.nearby to position.between and vice versa. The first ones of the action expressions in the following equations are used with the transition segment from the state position.nearby.leave to the state position.between. The second ones are used with the transition segment from position.between to position.nearby.approach.

$$\begin{array}{l}
\textsf{if floor} = \textsf{f then nearby}_{\textsf{f}} \textsf{ end if} \ , \quad \text{for } min.floor \leq \textsf{f} \leq max.floor \qquad (7.8) \\
\textsf{if floor} = \textsf{f then if in(dir.up) then nearby}_{\textsf{f+1}} \\
\qquad\qquad\qquad\qquad\qquad\qquad \textsf{else nearby}_{\textsf{f-1}} \\
\qquad\qquad\qquad\qquad\qquad\qquad \textsf{end if} \\
\qquad\qquad \textsf{end if} \ , \quad \text{for } min.floor \leq \textsf{f} \leq max.floor
\end{array}$$

Similar action expressions are associated with the transition segments from the state position.nearby.approach to position.onfloor to generate the events onfloor$_{f+1}$ and onfloor$_{f+1}$, respectively.

**Doors** The door on floor $f$ can be commanded to open by generating the event open$_f$, the door itself generates the event closed$_f$ when closing, see the statechart in Figure 7.6. In this statechart the priorities used to express non-determinism between staying in the state closed or leaving it have been removed. Leaving this state is restricted by occurrence of the event open$_f$.

**Requests** With respect to requests, a controller can recognize the events press$_f$, too, but no more information is available to it. The substates request$_f$ are designated terms in Rw-o to express when there is a request, but there is no direct physical counterpart which can be sensed.

## 7.3 Lift: Requirements

We present the requirements on the behaviour of the lift system at first informally. The requirements can be divided into two groups. The first ones are of interest for a user of the lift system. The second ones are of interest for an owner or a company operating the lift system. Some requirements can belong to both groups, in the following they have been stated only once.

From the point of view of an ordinary user of the lift, there are three important requirements:

1. The operation of the lift system shall be safe. This means a door shall be open only if the lift cabin is on the same floor.
2. All requests shall be served eventually.
3. Requests shall be served as fast as possible. This means, that when there is sufficient time to stop the lift cabin on a floor after requesting it, then the lift cabin shall not pass the floor without serving the request.

From the viewpoint of the owner the lift system shall do nothing unnecessary to minimise wear out and maintenance cost

4. The lift cabin shall not move without being requested.
5. A door shall be opened only if there is a request for this floor. This prevents e.g. that the lift cabin stops on every floor and the door is opened on it. This can be seen as a requirement of a user, too, because these unnecessary stops delays that requests are served.
6. The lift cabin shall change its direction of movement only if there is no further request in the current direction.
7. The lift cabin shall not leave the range of floors $[min.floor, max.floor]$.

In the following we give a formalisation of the requirements. Most of them can be expressed by simple temporal constraints associated with states of the statecharts. To formalise the requirements 3 and 6 auxiliary definitions are used. These can be expressed as an extension of the statechart or by using further TLA-programs.

As in Rw-o we associate the temporal constraints with the states in a way such that causality becomes as explicit as possible. This seems to support understanding in a better way than e.g. associating a constraint with the lowest common ancestor of the states refered to in the constraint.

**Requirement 1** is expressed as an invariant for each of the floors $\mathsf{f}$, $min.floor \leq \mathsf{f} \leq max.floor$. The invariant states the the lift cabin has to be on floor $\mathsf{f}$ if the door on floor $\mathsf{f}$ is opened. In addition the lift cabin has to be stopped.

$$\mathsf{door_f.opened} \rhd \Box \begin{array}{l} \wedge\ floor = \mathsf{f} \\ \wedge\ in(\mathsf{position.within.onfloor}) \\ \wedge\ in(\mathsf{dir.stopped}) \end{array} \tag{7.9}$$

**Requirement 2** can be expressed as leadsto-formulae associated with each of the states $\mathsf{request}_f$, $min.floor \leq \mathsf{f} \leq max.floor$:

$$\mathsf{request}_f \triangleright in(\mathsf{requested}) \leadsto \neg\, in(\mathsf{requested}) \tag{7.10}$$

Note that this temporal constraint implies further safety-properties. It implies for example that the state position.outside does not become active, because if this state becomes active then — in conjunction with requirement 1 — no further request can be served.

**Requirement 3** can be expressed by stating that the lift cabin cannot pass a floor when there is a request for it, provided the request was entered early enough to stop the lift on this floor.

It does not become clear from the informal requirements, whether the lift may leave a floor on which it has served a request, the door has closed, and there is a new request before the cabin leaves the floor. We decided that this further request shall be served, too. As a consequence there is the problem, that it might be possible now to hold the lift cabin on a floor by continuously requesting it. In Section 9.8.2 we show that this does not happen. On the other hand, allowing the lift cabin to leave, one would have to pay attention to the situation, when after serving the first of these requests there is no further request for some time, and now a further request has to be served.

Whether there is a request on the floor floor is expressed by the predicate *request. same*:

$$request.\,same \equiv \bigwedge_{min.floor \leq \mathsf{f} \leq max.floor} \begin{array}{l} floor = \mathsf{f} \\ \Rightarrow in(\mathsf{request}_f.\mathsf{requested}) \end{array}$$

Whether a request on the floor floor has been entered early enough is denoted by the predicate *early. request*. Requests are considered as generated early enough when they occur as long as the lift cabin is not nearby or already on the floor requested.

The definition of *early. request* is based on boolean variables $early.\,request_{\mathsf{f}}$ for each of the floors. The values of the variable $early.\,request_{\mathsf{f}}$ can be defined similarly to the refinement of the states dir.up and dir.down, cf. Figure 7.5, as a refinement of $\mathsf{request}_f$.requested. Here we take an alternative approach to show how TLA-programs can be associated with states.

$$early.\,request \equiv \bigwedge_{min.floor \leq \mathsf{f} \leq max.floor} \begin{array}{l} floor = \mathsf{f} \\ \Rightarrow \wedge\ early.\,request_{\mathsf{f}} \\ \quad\ \wedge\ in(\mathsf{request}_f.\mathsf{requested}) \end{array}$$

$$\begin{aligned}
&\mathsf{request}_f.\mathsf{requested} \\
&\rhd \wedge\ \mathit{early.request}_f = \vee\ \wedge\ \mathit{in}(\mathsf{position.onfloor}) \wedge \mathit{in}(\mathsf{dir.stopped}) \\
&\qquad\qquad\qquad\qquad\quad \wedge\ \mathit{floor} = f \\
&\qquad\qquad\qquad\qquad\ \vee\ \wedge\ \mathit{in}(\mathsf{position.nearby}) \wedge \mathit{in}(\mathsf{dir.up}) \\
&\qquad\qquad\qquad\qquad\quad \wedge\ \mathit{floor} + 1 \neq f \\
&\qquad\qquad\qquad\qquad\ \vee\ \wedge\ \mathit{in}(\mathsf{position.nearby}) \wedge \mathit{in}(\mathsf{dir.down}) \\
&\qquad\qquad\qquad\qquad\quad \wedge\ \mathit{floor} - 1 \neq f \\
&\quad \wedge\ \Box \left[ \mathit{early.request}_f{}' = \begin{array}{l} \vee\ \mathit{early.request}_f \\ \vee\ \mathit{in}(\mathsf{position.between}) \end{array} \right]_{\langle\rangle}
\end{aligned} \tag{7.11}$$

By the initial condition associated with $\mathsf{request}_f.\mathsf{requested}$ it is not distinguished whether the lift cabin is nearby the floor $f$ or whether it is nearby the floor directly above or below $f$ and is travelling towards floor $f$. In the second case the predicate $\mathit{in}(\mathsf{between})$ must become true before reaching the floor $f$, as a consequence $\mathit{early.request}_f$ is set to the correct value before reaching the floor $f$. It is ensured by this definition, that there is actually some time to stop the lift. Whether a controller is able to do so, depends on the speed of the controller and the size *delay* of the time frame to generate a $\mathsf{stop}$ event.

The requirement itself can be expressed in a first approach formally by the following next-state relation. Note that *request.same* and *early.request* are relevant only in steps in which the lift cabin leaves a floor. In this situation the variable $\mathsf{floor}$ corresponds exactly to the position of the lift cabin.

$$\mathsf{position} \rhd \Box \left[ \begin{array}{l} \mathit{in}(\mathsf{onfloor}) \wedge \neg\, \mathit{in}'(\mathsf{onfloor}) \\ \Rightarrow \vee\ \neg\, \mathit{request.same} \\ \quad\ \vee\ \wedge\ \mathit{request.same} \\ \qquad\ \ \wedge\ \neg\, \mathit{early.request} \end{array} \right]_{\langle\rangle} \tag{7.12}$$

It can be asked whether this temporal constraint has been associated with the correct state. The question arises because it depends on the movement of the lift cabin, whether the state $\mathsf{position.onfloor}$ is active. It can be argued that the movement of the lift cabin should be restricted directly.

This requirement can be restated such that the states $\mathsf{dir.up}$ and $\mathsf{dir.down}$ cannot remain active when a floor is reached which was requested early enough. The constraints are shown only for $\mathsf{dir.up}$, they have to be associated with $\mathsf{dir.down}$, too.

$$\begin{aligned}
&\mathsf{dir.up} \rhd \neg\, \mathit{request.same} \\
&\mathsf{dir.up} \rhd \Box\neg \wedge\ \mathit{in}(\mathsf{position.onfloor}) \\
&\qquad\qquad\quad \wedge\ \mathit{early.request}
\end{aligned} \tag{7.13}$$

Some remarks can be given on this constraint. At first, because the same formula is associated with the states $\mathsf{dir.up}$ and $\mathsf{dir.down}$, this indicates that a state expressing a common abstraction of $\mathsf{dir.up}$ and $\mathsf{dir.down}$ is missing in the statechart. At second, a mutual dependency between movement and position of the cabin exists here:

$\mathsf{dir.up}$ depends on $\mathsf{position.onfloor}$ now. At third, the requirement is expressed by an invariant now, whereas the constraint in Equation 7.12 was a next-state relation. Nevertheless the behaviours obtained are the same.

To show that the TLA-formulae in Equations 7.12 and 7.13 restrict the behaviour in the same way we distinguish between the cases that the lift cabin is moving or that is has stopped. Let us assume first that the lift cabin is moving and that there is a request for floor $\mathsf{f}$ which was entered early enough. We have to show that the lift cabin cannot pass the floor $\mathsf{f}$. According to Equation 7.12 $\mathsf{position.onfloor}$ cannot be left because there is a actually a request which was entered early enough. In the temporal constraint in Equation 7.13 each of the conjuncts becomes valid in the step leaving $\mathsf{position.nearby}$ and entering $\mathsf{position.onfloor}$. Hence $\mathsf{dir.up}$ cannot remain active and $\mathsf{dir.stopped}$ becomes active. For the same reason $\mathsf{dir.stopped}$ cannot be left as long as the request has not been granted. This implies, by the invariant in Equation 7.2 that $\mathsf{position.onfloor}$ cannot be left either. At second, let us assume that the lift cabin is stopped on floor $\mathsf{f}$. It can be shown that the lift cabin can still leave the floor in the same step in which the request is recognised, i.e. the state $\mathsf{request}_{\mathsf{f}}\mathsf{.requested}$ becomes active. Let $\langle s, t\rangle$ be such a step. It is clear that $\mathsf{position.onfloor}$ can be left according to Equation 7.12 because in the status $s$ this request has not been recognised yet. $\mathsf{position.onfloor}$ can be left according to Equation 7.13 because $early.\,enough_{\,\mathsf{f}}$ is not true at $s$ when $\mathsf{dir.up}$ becomes active. At this point the mutual dependency between movement and position of the lift cabin becomes apparent.

**Requirement 4** is expressed by more specific invariants. The states $\mathsf{dir.up}$ and $\mathsf{dir.down}$ can be active only if there is a request in the direction in which the lift cabin moves. Whether there is a request above or below the current position is expressed by two predicates *request. up* and *request. down*. They are defined as

$$request.\,up \equiv \bigvee_{min.floor \leq \mathsf{f} \leq max.floor} \begin{array}{l} \wedge\ floor < \mathsf{f} \\ \wedge\ in(\mathsf{request}_{\mathsf{f}}\mathsf{.requested}) \end{array}$$

$$request.\,down \equiv \bigvee_{min.floor \leq \mathsf{f} \leq max.floor} \begin{array}{l} \wedge\ floor > \mathsf{f} \\ \wedge\ in(\mathsf{request}_{\mathsf{f}}\mathsf{.requested}) \end{array}$$

The temporal constraints are:

$$\begin{array}{l} \mathsf{dir.up} \rhd \Box\, request.\,up \\ \mathsf{dir.down} \rhd \Box\, request.\,down \end{array} \tag{7.14}$$

**Requirement 5** can be expressed as two initial conditions.

That the lift cabin shall stop only if there is a request on the corresponding floor can be expressed by associating the predicate *request. same* as an initial condition with $\mathsf{dir.stopped}$. Because this state is the target of a default transition it has to be guaranteed that it can be entered if its ancestors become active:

$$\begin{aligned}\mathsf{dir.stopped} \rhd \lor\ & \mathit{request.\,same} \\ \lor\ & \mathsf{en(dir)} \in E_{exists}\end{aligned} \tag{7.15}$$

The door shall be opened on a floor only if there is a request on this floor:

$$\mathsf{door}_f\mathsf{.opened} \rhd \mathit{in}(\mathsf{request}_f\mathsf{.requested}) \tag{7.16}$$

**Requirement 6** refers to the previous direction of the lift cabin before stopping it to serve a request. To this purpose we introduce a variable *dirold* and set it appropriately when entering the states $\mathsf{dir.up}$ and $\mathsf{dir.down}$, respectively.

$$\begin{aligned}&\mathsf{dir.up} \rhd \mathit{dirold} = \mathit{up} \\ &\mathsf{dir.down} \rhd \mathit{dirold} = \mathit{down}\end{aligned} \tag{7.17}$$

The requirement itself can be expressed as

$$\begin{aligned}\mathsf{dir} \rhd \land\ \Box & \begin{bmatrix} \land\ \mathit{in}(\mathsf{stopped}) \land \mathit{dirold} = \mathit{down} \\ \land\ \mathit{in}'(\mathsf{up}) \\ \Rightarrow \neg\, \mathit{request.\,down} \end{bmatrix}_{\langle\rangle} \\ \land\ \Box & \begin{bmatrix} \land\ \mathit{in}(\mathsf{stopped}) \land \mathit{dirold} = \mathit{up} \\ \land\ \mathit{in}'(\mathsf{down}) \\ \Rightarrow \neg\, \mathit{request.\,up} \end{bmatrix}_{\langle\rangle}\end{aligned} \tag{7.18}$$

**Requirement 7** can be expressed by

$$\mathsf{position.outside} \rhd \mathit{false} \tag{7.19}$$

The formulae expressing the requirements can be inserted into a copy of RW-O as depicted in Figure 7.7. This statechart does not contain the changes made in RW-M to RW-O. Some formulae have been simplified slightly and for some formulae which are symmetrical for both directions only the subformula for one direction is shown. Some names have been abbreviated, especially $\mathsf{request}$ by $\mathsf{req}$ and *floor* by *flr*. To distinguish TLA-formulae from the reference world, auxiliary definitions used to state the requirements and the requirements themselves, the latter ones have been preceded by exclamation marks to indicate their optative mood. The exclamation marks are indexed by the number of the corresponding requirement. This does not influence the semantics.

## 7.4 Lift: Specification

The requirements have been expressed in REQ by refering to terms visible to the plant and to an external observer. These terms do not need to be visible to a

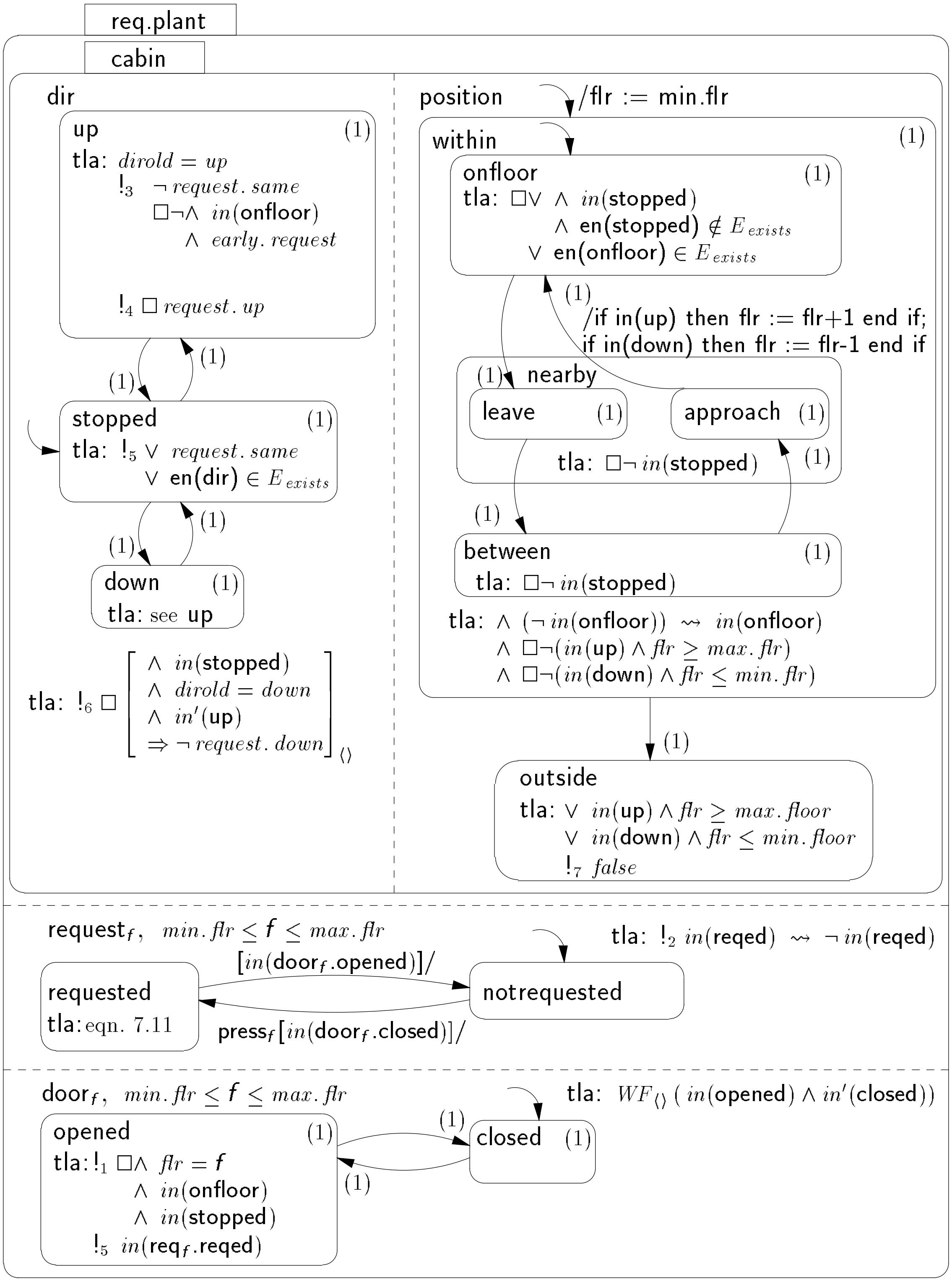



Figure 7.7: REQ: Complete description

controller, therefore in SPEC the requirements have to be restated in terms visible to a controller. To derive SPEC from REQ we use RW-M which describes how a controller is able to monitor and to influence the state of the plant.

SPEC will be similar to REQ, therefore we do not present it here completely. We present only the part of the specification corresponding to the Requirements 1 and 2.

**Requirement 1** (The operation of the lift system shall be safe) To specify this requirement it is sufficient to show that the terms used in the requirements can be made visible to a controller, too. Then the formulation of the requirement can be used again in the specification, it has to be changed only that the references are made to SPEC instead of REQ. In the case of this requirement the terms used are the position of the lift cabin, i.e. whether the state position.onfloor is active and the value of the variable floor, and whether the door on a floor is open or closed. Each of these terms can be made visible to a controller in a way similar to the statecharts in Figures 7.9 and 7.10. These describe a part of a controller which keeps track of the position of the lift cabin and of the doors.

**Requirement 2** (All requests shall be served eventually) It is possible to specify this requirement similar as the one before by showing that the terms refered to can be made visible to a controller, too. We will specify this requirement in a more direct way by refering explicitly to events visible to a controller.

That a new request for the lift cabin is entered means that the event $\mathsf{press}_f$ occurs. The door on floor $f$ opens only if the event $\mathsf{open}_f$ has been generated by a controller. Hence this requirement can be specified by the following temporal constraint:

$$\begin{aligned} &\wedge\ \mathsf{press}_f \in E_{exists} \\ &\wedge\ in(\mathsf{spec.door}_f\mathsf{.closed}) \\ &\leadsto \mathsf{open}_f \in E_{exists} \end{aligned} \tag{7.20}$$

I.e. if a request has to be granted, then the door on the corresponding floor is commanded eventually to open.

Each of the remaining requirements can be specified in a similar way.

## 7.5 Lift: Design of a Controller

In the following a controller of the lift system is described. In contrast to the previous descriptions it is described mainly in statecharts. TLA-formulae are used only to define some auxiliary variables.

The controller is a part of a system which contains also the plant and the environment. The controller must be able to communicate with the plant, hence we use the description of the plant in RW-M, cf. Section 7.2. The environment has not been

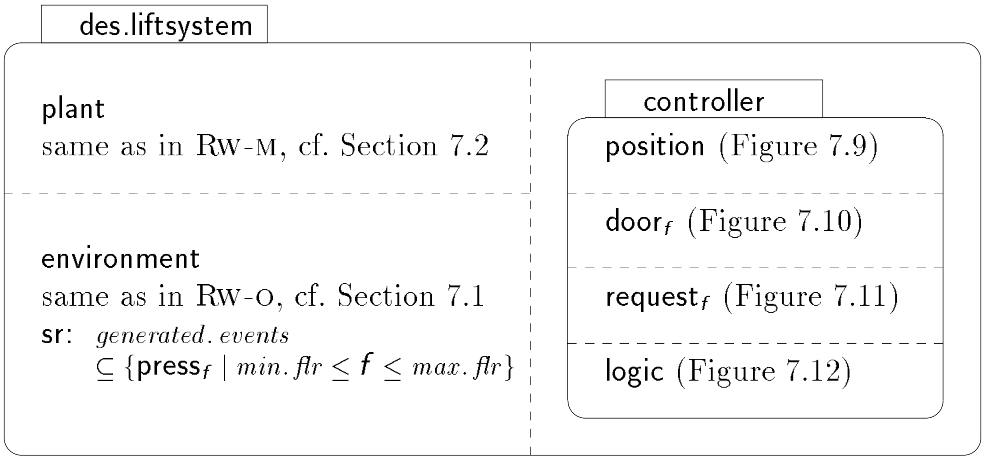



Figure 7.8: DES: Complete system with controller and plant

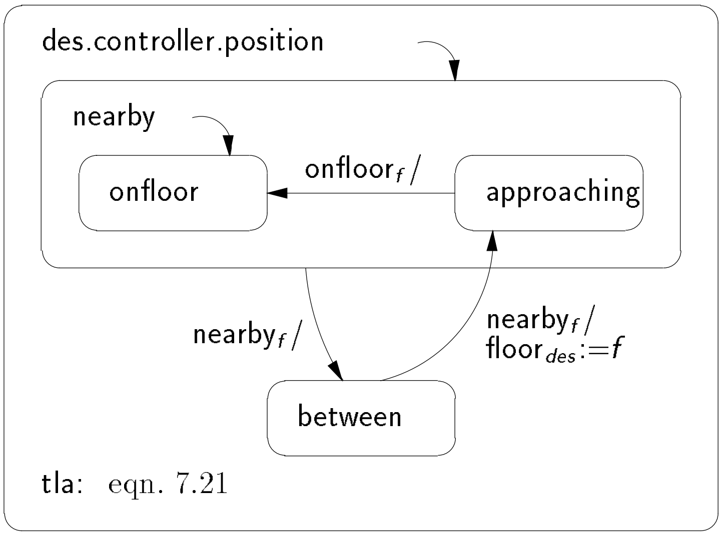
changed in RW-M, hence we use its description in RW-O, cf. Section 7.1. This system is depicted in Figure 7.8. Note that the different parts in this figure refer to already existing parts of the system and to parts of the controller. But the statecharts in this figure do not describe desired behaviour, e.g. requirements.

As depicted in this figure the complete system can be considered as a closed one. No inputs from outside of des.liftsystem are considered. The events $\mathsf{press}_f$ are generated by the static reaction of environment in steps of the statechart, i.e. in *Execute. step*-steps. In this figure the interfaces between the different parts do not become clear. In Section 8.7 we use activities and activity-charts to describe these interfaces explicitly.

There are two different parts of the controller, one keeping track of the state of the plant (des.controller.door$_f$ and des.controller.position) and of requests (des.controller.request$_f$). The other one is the control logic itself (des.controller.logic). It determines what should be done next and generates appropriate events.

**Lift Cabin** The direction of movement and the position of the lift cabin are observed by the controller more coarsely as they are defined in RW-M. For the direction it is only kept track of the last direction of movement by a variable $dirold_{des}$. Its value can be defined by considering the occurrence of the events moveup and movedown. We associate the following next-state relation with the state des.controller.position:

$$\begin{array}{l} \mathsf{des.controller.position} \\ \rhd \wedge\ dirold_{des} = down \\ \quad \wedge\ \Box \begin{bmatrix} \wedge\ \mathsf{moveup} \in E_{exists} \ \Rightarrow\ dirold'_{des} = up \\ \wedge\ \mathsf{movedown} \in E_{exists} \ \Rightarrow\ dirold'_{des} = down \\ \wedge\ \ \{\mathsf{moveup}, \mathsf{movedown}\} \cap E_{exists} = \emptyset \\ \qquad \Rightarrow dirold'_{des} = dirold_{des} \end{bmatrix}_{\langle\rangle} \end{array} \qquad (7.21)$$




The event and action expressions refering to $f$ have to be repeated for each floor $f$.

Figure 7.9: DES: Distinguishing possible positions of the lift cabin

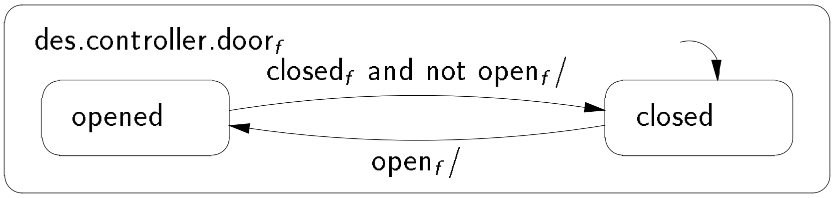



Figure 7.10: DES: Keeping track of the state of a door

---

Note that the definition of $dirold_{des}$ is well-defined only if the events moveup and movedown do not exist together.

W.r.t. the position of the lift we distinguish in the controller the states onfloor, approaching, and between of the statechart in Figure 7.9, and use a variable $floor_{des}$ to model the number of the floor, the lift is on or nearby. In contrast to RW-O this variable is changed already when leaving the state between instead of changing it when entering the state onfloor.

**Doors** To keep track of a the state of a door we use a statechart as depicted in Figure 7.10. Note, that corresponding states of a door in the plant (rwm.plant.door$_f$, Figure 7.6) and the controller (des.controller.door$_f$, Figure 7.10) are not in complete synchrony. When closing a door in the plant an event closed$_f$ is generated in this step, this can be recognised by the controller in the next step of the statechart only. It is possible that the controller generates in the same step, in which the door closes, the event open$_f$. Then the door opens again in the next step. This is the same step in which the controller recognises the event closed$_f$. Therefore, the state

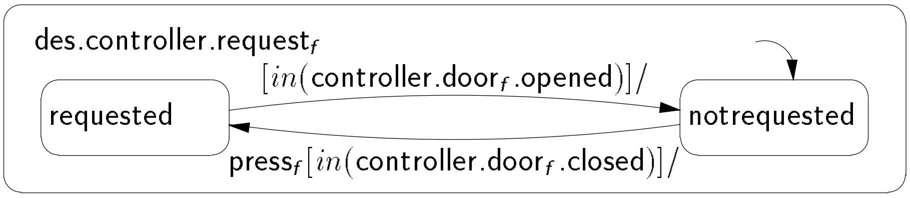



Figure 7.11: DES: Keeping track of requests

des.controller.door$_f$.opened must not be left.

Another possibility to avoid this problem is to write the controller in a way such that it does not generate open$_f$ events as long as it cannot be sure that the door is closed, i.e. it generates open$_f$ events only when the state des.controller.door$_f$.closed is active.

No priorities are needed in this statechart because it is completely determined by the triggering events of the transitions when the states can and have to be left.

**Requests** Whether there is a request on a floor is modeled in the controller similarly as in the definition of the requests in rwo. The difference is that it refers to the model of the doors in the controller instead of to the doors of the plant, cf. Figure 7.11.

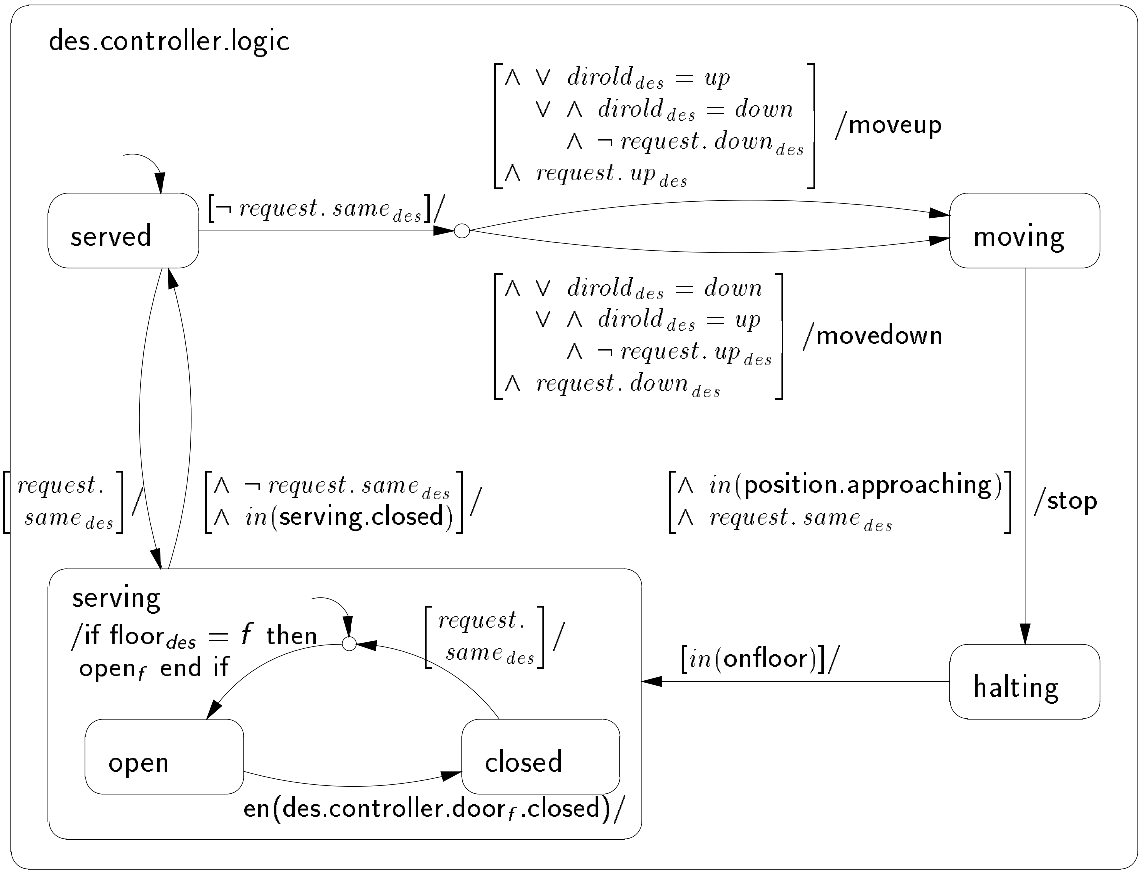
The predicates $request.same_{des}$, $request.up_{des}$, and $request.down_{des}$ are defined as in REQ. Their definitions refer here to the part of the controller keeping track of requests:

$$
\begin{aligned}
& request.same_{des} \\
\equiv & \bigwedge_{min.floor \leq \mathsf{f} \leq max.floor} \quad floor_{des} = \mathsf{f} \Rightarrow in(\text{des.controller.request}_{\mathsf{f}}.\text{requested})
\end{aligned} \tag{7.22}
$$

$$
\begin{aligned}
& request.up_{des} \\
\equiv & \bigvee_{min.floor \leq \mathsf{f} \leq max.floor} \quad \wedge\ floor_{des} < \mathsf{f} \ \wedge\ in(\text{des.controller.request}_{\mathsf{f}}.\text{requested})
\end{aligned}
$$

$$
\begin{aligned}
& request.down_{des} \\
\equiv & \bigvee_{min.floor \leq \mathsf{f} \leq max.floor} \quad \wedge\ floor_{des} > \mathsf{f} \ \wedge\ in(\text{des.controller.request}_{\mathsf{f}}.\text{requested})
\end{aligned}
$$

**Control Logic** The control logic basically cycles through four states. The cycle starts in the state served, cf. the statechart in Figure 7.12. This state is active as long as there is no request. If there is a request on another floor the state moving becomes active. In this state the lift cabin is commanded to start in the appropriate




The event and action expressions indexed with $f$ have to be repeated for each floor $f$.

Figure 7.12: DES: Control logic

direction. Before reaching a requested floor, the lift cabin is commanded to stop and the state halting becomes active. The state halting remains active until the lift cabin is actually on the floor. Then the state serving becomes active. In this state the request is served actually. Thereafter the state served becomes active again.

There are two further short cycles of states. They correspond to serving a request on the same floor as the previous request. One cycle is within the state serving. It is taken to serve a further request immediately after serving a request. The other cycle is from served to serving and back to served. It is taken to serve requests on the floor on which the lift cabin waits if there has been no request for some time.

**Informal Proof** We do not give here a formal proof, that the combination of plant and this controller satisfies the requirements. A complete and formal proof for a similar plant and controller can be found in [Deiß, 1996]. We investigate here only, for which values of *delay* the requirements can be satisfied. Remember that *delay* describes how long it takes for the lift cabin to stop.

For this purpose let us consider that a request is given as late as possible. This means that the lift cabin is moving and that a request is entered just before the lift cabin is nearby the requested floor. We assume that the lift cabin is moving upwards, the case of moving downwards is symmetrical. In the statechart this corresponds to the situation that the state des.plant.cabin.dir.up is active and that the event $\mathsf{press}_f$ has been generated in the step before the state des.plant.position.nearby becomes active. Let the pair of TLA-states $\langle s_1, s_2\rangle$ be this step. In this step the state des.plant.dir.up.timetostop becomes active with $n = delay$ at $s_2$. W.r.t. the controller we have that the state des.controller.logic.moving is active at $s_1$. The event $\mathsf{nearby}_f$ is generated in this step, hence des.controller.position.between is active in $s_1$ and $s_2$. The state des.controller.$\mathsf{request}_f$.requested becomes active in this step, too.

In the next step, denoted by $\langle s_2, s_3\rangle$, the counter $n$ is reduced by 1 to $delay - 1$ in the state des.plant.dir.up.timetostop. In des.controller.position the state nearby.approaching becomes active. The state des.controller.logic.moving remains active.

This state is left in the next step, denoted by $\langle s_3, s_4\rangle$, thereby generating the event stop. $n$ is reduced again by 1, i.e. it has the value $delay - 2$ at $s_4$.

The event stop can be used to enable the compound transition from the state des.plant.dir.up.timetostop to des.plant.dir.up.stopping in the next step. This is denoted $\langle s_4, s_5\rangle$. This compound transition can be taken if $n$ is greater then 0 in $s_4$, this implies that $delay - 2 > 0$ and hence that $delay > 2$ have to be valid such that requirement 3 is satisfied.

Note, that this specific value of *delay* depends on the semantics of statecharts chosen. In the superstep-semantics as described in [Harel and Naamad, 1996] and the microstep-semantics of [Pnueli and Shalev, 1991] the transitions of the observing part of the controller and the control logic would be taken in one step and $delay = 1$ would be sufficient.

## 7.6 Two Variations of the Controller

The description of the controller in Section 7.5 is very close to a pure statechart. Because we will investigate later on the relation to synchronous languages we present in Section 7.6.1 a synchronous controller. As a second variant we present a controller written completely as a TLA-program in Section 7.6.2.

### 7.6.1 A Synchronous Controller

In the controller described before several steps are taken to react to changes in the plant. As example consider that the door on a floor $f$ closes and that there is a request on another floor. Let us assume that this step is denoted by $\langle s_1, s_2\rangle$,

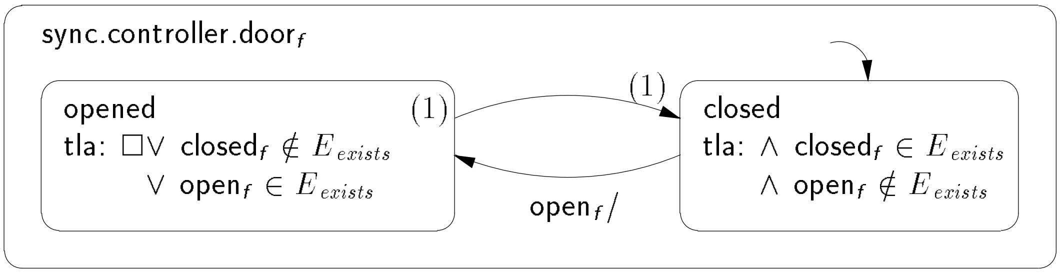



Figure 7.13: DES: Synchronous variant, keeping track of a door

where $s_1$ and $s_2$ are TLA-states. In $s_2$ we have that the states des.plant.door$_f$.closed and des.controller.logic.serving.open are active and that the events press$_f$ and closed$_f$ exist. There is the following sequence of steps and TLA-states: In the next step $\langle s_2, s_3\rangle$ the state des.controller.door$_f$.closed becomes active. In the step $\langle s_3, s_4\rangle$ the state des.controller.logic.serving.closed becomes active. Two further steps are necessary to generate one of the events moveup and movedown: In the step $\langle s_4, s_5\rangle$ the state des.controller.logic.served becomes active, in the step $\langle s_5, s_6\rangle$ the state des.controller.logic.moving becomes active. Finally, in the step $\langle s_6, s_7\rangle$ the state des.plant.dir.stopped is left, i.e. the lift cabin begins to move. Hence it takes five steps of the statechart from entering a request until the lift cabin begins to move.

Each single step counts as one step here. On the contrary, in the micro-step semantics of [Pnueli and Shalev, 1991] one step less is needed because the change in the statechart keeping track of a door, cf. Figure 7.10, and the first step in the state des.controller.logic from serving.open to serving.door can be taken together in one step. In the superstep semantics of [Harel and Naamad, 1996] the complete sequence of steps is considered as one (super-) step.

To avoid a hierarchical notion of time we use a synchronous description of a controller to shorten this sequence of steps to one step. This shows that the combination of TLA and statecharts can be used as a synchronous language.

**Keeping Track of the Plant** The controller has to react synchronously to changes in the plant, hence the statecharts keeping track of the status of the plant have to be made synchronous, too. We will show this for the one keeping track of a door. The original version is shown in Figure 7.10, the synchronous one in Figure 7.13. The state opened can remain active as long as the event closed$_f$ does not occur, or if the event open$_f$ occurs simultaneously. To synchronise on the event closed$_f$ it is necessary to start with a non-deterministic choice between keeping the state opened active or leaving it. This choice is restricted by the invariant associated with the state opened. The negation of the invariant is associated as initial condition with the state closed to prevent that the transition segment is taken without reason. The transition segment from closed to opened is handled differently. It has the event open$_f$ as triggering event. This is sufficient because the plant itself needs a step to

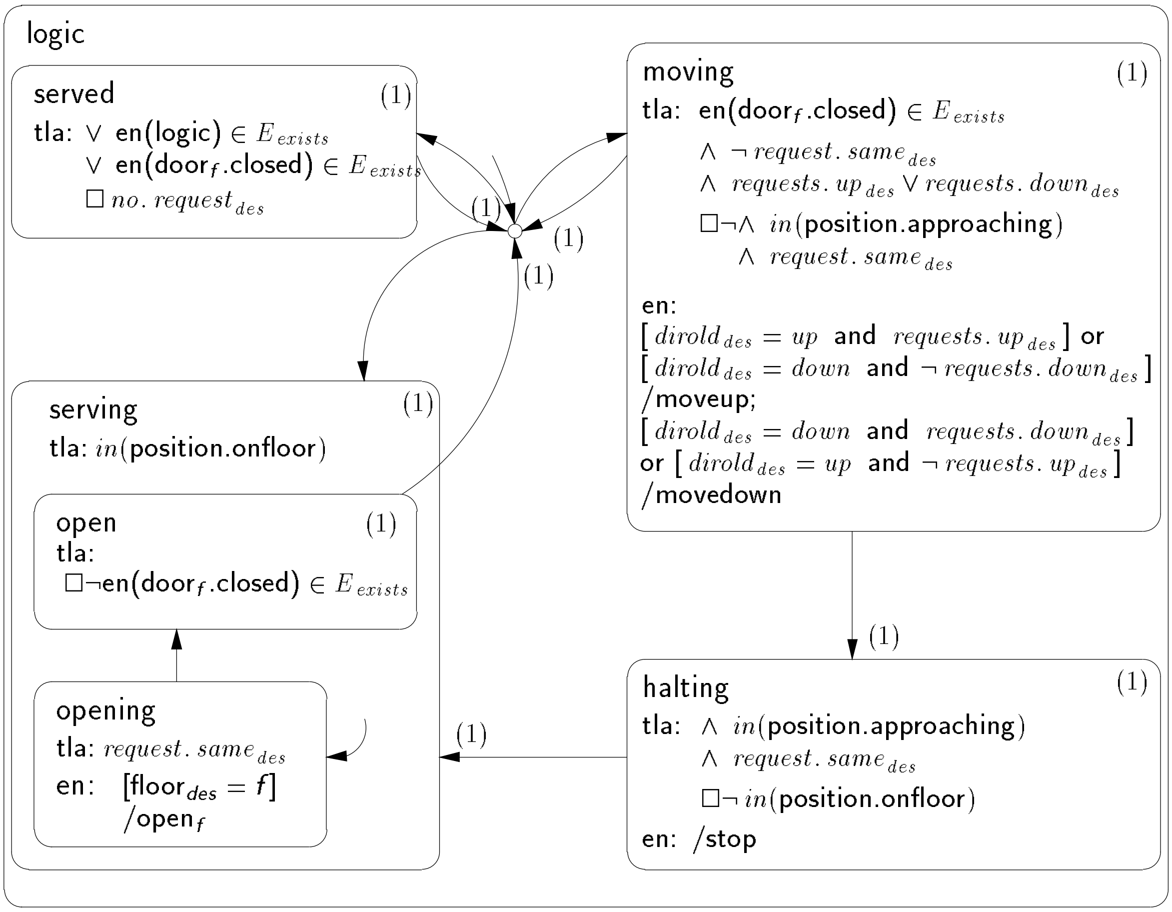



Figure 7.14: DES: Synchronous variant, control logic

react on this event and this is necessary because this event will finally be generated in the same step as the event $\mathsf{close}_f$. Using it as a triggering event inserts a delay of one step, breaking a cycle in the causality relation, cf. Section 9.5.

Keeping track of requests and of the position of the lift cabin can be described in a similar way by statecharts with appropriate constraints.

**Control Logic** The statechart logic is changed in a similar way. In addition further transitions are inserted to shorten sequences of steps. The events used to control the plant are generated when entering an appropriate state. Alternatively, it can be expressed by temporal constraints when they are generated. These temporal constraints have to restrict the generation of the events to steps in which the corresponding states are entered. In this statechart all references to states are made to the synchronous variants of these parts of the controller. In no case they do refer to the plant directly.

In addition to $request.\,same_{des}$, etc., cf. Equation 7.22, we define the predicate

$no.\ request_{des}$. It is valid, if there is no request on any floor:

$$no.\ request_{des} \equiv \bigwedge_{min.floor \leq f \leq max.floor} \neg\, in(\mathsf{sync.controller.request}_f\mathsf{.requested}) \qquad (7.23)$$

In a status corresponding to $s_1$ before, the state sync.controller.logic.serving.open is active. When the door in the plant closes in a step, this is signalled to the controller. Due to the synchronous nature of the statechart sync.controller.door$_f$, cf. Figure 7.13, the state closed becomes active and the event en(sync.controller.door$_f$.closed) is generated. Hence the state sync.controller.logic.serving.open has to be left. Because we assumed that there is no request on the same, but on another floor, neither the state sync.controller.logic.serving.opening can become active, nor can sync.controller.logic.served. Instead, the state sync.controller.logic.moving becomes active and one of the events moveup and movedown is generated. All this happens in one step.

That the sequence of five steps could be reduced to one step was possible because at first the model of the plant, i.e. the states sync.controller.position, sync.controller.door$_f$, and sync.controller.request$_f$, is in synchrony with the plant itself now. At second, the substate open of sync.controller.logic.serving is the second state after entering serving, hence it and serving itself can be left immediately when the door closes instead of activating the state closed for one intermediate step as in Figure 7.12. At third, when sync.controller.logic.serving becomes inactive the state sync.controller.logic.moving can become active in the same step. It has been avoided that sync.controller.logic.served becomes active. Even in a synchronous controller it is not possible to enter and leave a state in one step. Note that the initial conditions of the states serving, moving, and served are a complete and disjoint case distinction.

In the statechart in Figure 7.14 most constraints used to determine which state is active occur twice. Once as an invariant of the source state of a transition segment and a second time negated as an initial condition of the target state. This is caused by the fact that priorities can be used here to express non-determinism between the static transitions keeping a state active and the transition leaving a state. But they cannot be used to express that state remains active as long as its invariant is valid and is left only otherwise. This can be expressed by enabling conditions, but these are not synchronous. It seems to be worthwhile to investigate how priorities can be used on temporal constraints, too.

### 7.6.2 A TLA Controller

The descriptions in the previous examples have been written in statecharts or in the combination of statecharts and TLA. We want to present here a description completely written in TLA as an example of how this fits into our framework. This means to replace the substates of a statechart with a TLA-program which is associated with the root-state. We do this here for the control logic as it is depicted in Figure 7.12.

The TLA-formula is defined along the lines of the controller in Section 7.5. It uses a control variable $\pi$, with $\pi \in \{\mathsf{served}, \mathsf{moving}, \mathsf{halting}, \mathsf{serving}\}$. Changes of the control variable are described by TLA-actions corresponding to the transitions of the statechart. These TLA-actions also describe which events are generated when changing $\pi$. There is an extra TLA-action describing when the control variable does not change. The TLA-formulae already associated with this statechart become part of the TLA-formula, too.

It is possible to instantiate the formula *statechart* with the statechart in Figure 7.12 and use this as the TLA-formula. Instead, we want to give a TLA-formula which can be seen as description of its own. Nevertheless this is defined along the lines of the statechart in Figure 7.12.

As in the previous descriptions we keep initialisation simple. We take served as the initial value of the control variable:

$$control.\,logic.\,init \equiv \pi = \mathsf{served}$$

If there is a request on another floor, but not on the current one, then the cabin is commanded to move. It is taken into account that the direction of movement is kept if possible. By $E_{out}$ we denote the set of possible output events of the controller, i.e. $E_{out} = \{\mathsf{moveup}, \mathsf{movedown}, \mathsf{stop}, \mathsf{open}_\mathsf{f} \mid min.\,floor \leq \mathsf{f} \leq max.\,floor\}$.

$$\begin{aligned} Moveup \equiv\ & \land\ \pi = \mathsf{served} \land \pi' = \mathsf{moving} \\ & \land\ \neg\, request.\,same_{des} \\ & \land\ request.\,up_{des} \\ & \land\ \lor\ dirold_{des} = up \\ & \quad\ \lor\ \neg\, request.\,down_{des} \\ & \land\ E'_{exists} \cap E_{out} = \{\mathsf{moveup}\} \end{aligned}$$

By intersecting $E'_{exists}$ with $E_{out}$ we express which events are generated as well as which ones are not generated. A TLA-action *Movedown* can be defined similarly.

The lift cabin remains moving until it approaches a floor with a request (*Stopping*) and it has actually stopped when the floor has been reached (*Opening*). The TLA-action *Opening* also describes that the default transition of the state serving in Figure 7.12 is taken and the corresponding event is generated.

$$\begin{aligned} Stopping \equiv\ & \land\ \pi = \mathsf{moving} \land \pi' = \mathsf{halting} \\ & \land\ in(\mathsf{des.controller.position.approaching}) \\ & \land\ request.\,same_{des} \\ & \land\ E'_{exists} \cap E_{out} = \{\mathsf{stop}\} \end{aligned}$$

$$\begin{aligned} Opening \equiv\ & \land\ \pi = \mathsf{halting} \land \pi' = \mathsf{serving} \\ & \land\ in(\mathsf{des.controller.position.onfloor}) \\ & \land\ \bigwedge_{min.floor \leq \mathsf{f} \leq max.floor} \begin{array}{l} \mathsf{floor} = \mathsf{f} \\ \Leftrightarrow E'_{exists} \cap E_{out} = \{\mathsf{open}_\mathsf{f}\} \end{array} \end{aligned}$$

The door is considered to be closed again, when a corresponding event from the part of the controller monitoring the plant arrives. Then it is possible to return to the state served. If a new request arrives in the mean time, then control returns to serving. This is a deviation from the statechart in Figure 7.12 in which control does not leave the state serving if the new request arrives before serving is left.

$$\begin{aligned} \mathit{Served} \equiv\ & \wedge\ \pi = \mathsf{serving} \wedge \pi' = \mathsf{served} \\ & \wedge \bigvee_{\mathit{min.floor} \leq \mathsf{f} \leq \mathit{max.floor}} \begin{array}{l} \wedge\ \mathsf{floor} = \mathsf{f} \\ \wedge\ \mathsf{en}(\mathsf{door}_\mathsf{f}.\mathsf{closed}) \in E_{exists} \end{array} \\ & \wedge\ E'_{exists} \cap E_{out} = \emptyset \end{aligned}$$

$$\begin{aligned} \mathit{Reopening} \equiv\ & \wedge\ \pi = \mathsf{served} \wedge \pi' = \mathsf{serving} \\ & \wedge\ \mathit{request.same}_{des} \\ & \wedge \bigwedge_{\mathit{min.floor} \leq \mathsf{f} \leq \mathit{max.floor}} \begin{array}{l} \mathsf{floor} = \mathsf{f} \\ \Leftrightarrow E'_{exists} \cap E_{out} = \{\mathsf{open}_\mathsf{f}\} \end{array} \end{aligned}$$

Up to now we described when the control variable changes. But it has to be described, too, when it does not change. This is the case when the TLA-actions above cannot be taken. This is done for all values of $\pi$ in the action *Stay*:

$$\begin{aligned} \mathit{Stay} \equiv\ & \wedge\ \pi = \pi' \\ & \wedge \vee\ \wedge\ \pi = \mathsf{served} \\ & \qquad \wedge\ \neg\, \mathit{request.same}_{des} \wedge \neg\, \mathit{request.up}_{des} \wedge \neg\, \mathit{request.down}_{des} \\ & \quad \vee\ \wedge\ \pi = \mathsf{moving} \\ & \qquad \wedge\ \neg \wedge\ \mathit{in}(\mathsf{des.controller.position.approaching}) \\ & \qquad\quad\ \ \wedge\ \mathit{request.same}_{des} \\ & \quad \vee\ \wedge\ \pi = \mathsf{halting} \\ & \qquad \wedge\ \neg\, \mathit{in}(\mathsf{des.controller.position.onfloor}) \\ & \quad \vee\ \wedge\ \pi = \mathsf{serving} \\ & \qquad \wedge\ \neg \bigwedge_{\mathit{min.floor} \leq \mathsf{f} \leq \mathit{max.floor}} \begin{array}{l} \mathsf{floor} = \mathsf{f} \\ \Leftrightarrow E'_{exists} \cap E_{out} = \{\mathsf{open}_\mathsf{f}\} \end{array} \end{aligned}$$

The TLA-formula *control.logic* consists of these TLA-actions and the predicate *control.logic.init*. We do not allow stuttering steps of *control.logic* w.r.t. the status of the statechart with which it is associated. Otherwise events might get lost. That we do not allow these stuttering steps is expressed by using the state function $\langle\rangle$ in the next-state relation. This implies that *control.logic* is not a TLA-program in canonical form. Note that the TLA-action *Stay* can be seen as an explicit stuttering step. Also, we do not need fairness properties.

$$control.\,logic \equiv \wedge\ control.\,logic.\,init \\ \wedge\ \Box \begin{bmatrix} \vee\ Moveup \vee MoveDown \\ \vee\ Stopping \\ \vee\ Opening \\ \vee\ Served \\ \vee\ Reopening \\ \vee\ Stay \end{bmatrix}_{\langle\rangle}$$

This TLA-formula can be associated with the state des.controller.logic of the statechart in Figure 7.8.

## 7.7 Lift: Remarks on the Descriptions of the Lift System

It can be observed from the descriptions presented that they are all based upon the model of the plant in RW-O. Different extensions are made to it in the other descriptions. Especially in DES a model of the plant becomes part of the controller. This cannot be identical to the model of the plant in the other descriptions, because the state of the plant must be derived from the visible entities of the plant, this may introduce imprecision and there can be a delay in recognising the correct state of the plant. Nevertheless this model of the plant in RW-O is essential to relate the various descriptions.

As a second point we want to consider the relationship between RW-O and RW-M more closely. RW-O has been used mainly to identify distinguishable modes and to define further terms from these modes. This description is extended in RW-M. To be useful, these descriptions must be closely related. An appropriate relation is, that RW-M refines RW-O. Clearly, additional events and states in RW-M are hidden when comparing these descriptions.

In the example presented here several different changes from RW-O to RW-M have been made:

- The most simple one is that some changes in the plant are explicitly signalled to a controller. This is for example the case for the events $\mathsf{closed}_f$, cf. Figure 7.6.
- Vice versa, events generated by a controller are used as triggering events of transitions segments. This is used to reduce non-determinism between staying in or leaving a state of the statechart. An example can be seen in the same figure, i.e. the event $\mathsf{open}_f$. In this case it is intended that the state has to be left when this event occurs, as a consequence the priorities on the compound and static transitions can be removed.

- A third and more complicated example is the refinement of the states dir.up and dir.down to describe how the lift cabin stops, i.e. how it reacts to occurrence of the event stop. It is described that the cabin cannot stop immediately. Furthermore, in combination with liveness-properties of the state position.within it is described that the cabin actually reacts to occurrence of this event. Note that in this case the state dir.up is not left as soon as possible, only a lower bound on the number of steps throughout which dir.up must remain active after occurrence of the event stop is given. This is actually a refinement of RW-O, if the additional states are hidden.

The model of the lift system used here differs from that in [Deiß, 1996] in one point essentially. There, the lift cabin moved from one floor to the next one in one step of the statechart. Here we use a finer and more realistic model, a sequence of steps is necessary for the lift cabin to move from one floor to the next one. As a consequence it became necessary to describe a real-time property, i.e. the time necessary to stop the lift cabin. Time was modeled simply as the number of steps executed. In this way it is possible to describe simple real-time properties of a reactive system. All other liveness properties can be expressed by fairness- or leadsto-formulae in this example.

Three different versions of a controller have been presented. We consider it as quite natural to describe a controller using only statecharts without temporal constraints. The descriptions in the Sections 7.6.1 and 7.6.2 are intended to show that the controller can be described following other styles of description, too. But we do not claim, that these are more readable than the pure statecharts variant.

Finally let us note, that to compose the controller, the environment, and the model of the plant we embedded these components as substates in an and-state, cf. Figure 7.8. The interface of the individual components is not described explicitly by this kind of composition. For this purpose we will introduce activities in Chapter 8 and apply them to this example.

## 7.8 Second Example: Production Cell

In this section we present an example in which parallel activities are coupled by synchronising their actions. The system which is described is that of the well-known case study 'Production Cell' [Lewerentz and Lindner, 1995]. The descriptions presented here are derived from [Schowalter, 1997]. In contrast to the lift example we present here only a small part of RW-O and REQ.

In the lift example the constraints associated with the states have been mainly invariants and leadsto-formulae. In our description of the production cell actions taken jointly by parallel processes are more important. We show how this synchronisation can be described within our approach. This provides an example of the more abstract description of synchronisation in Section 6.2.1. In [Schowalter, 1997] these joint actions have been described explicitly as TLA-actions and their combinations have been described as TLA-formulae. Here we will show in addition how invariants can be used instead of explicitly combining TLA-actions. This is done by taking the effects of the actions which are visible at the end of a step into account instead of the actions themselves.

From the production cell we consider only a part which is large enough to explain this use and the description of joint actions. The production cell itself and the requirements on its behaviour are introduced in the corresponding sections.

### 7.8.1 Production Cell: Reference World (Observer)

From an abstract point of view, the production cell consists of a sequence of devices which either transport metal plates from one location to another or which carry out some work on the plates. The complete production cell consists of the following devices, given in the order in which they are travelled by the plates: A feed belt ($\mathsf{fb}$), by which the plates enter the cell, an elevating rotary table ($\mathsf{ert}$), by which plates which reached the end of the feed belt can be moved to a position in which they can be grasped by an arm ($\mathsf{arm1}$) of a robot. This robot puts the plates into a press ($\mathsf{press}$) in which they are forged. A second arm of the same robot ($\mathsf{arm2}$) takes the plates from the press and puts them onto a deposit belt ($\mathsf{db}$). In reality the plates leave the production cell via this belt, in the case study the cell is artificially closed and the plates are taken by a crane ($\mathsf{crane}$) from the deposit belt and put again onto the feed belt.

Because we concentrate on the movement of plates from the feed belt to the elevating rotary table and from this to the first arm of the robot we do not need to introduce further detail, as e.g. that the arms of the robot are connected in a fixed angle.

**Feed Belt** Because we are interested in transporting plates from the feed belt to the table it is not necessary to describe what happens at the beginning of the feed belt. For our purposes it is sufficient to describe the feed belt as depicted in Figure 7.15. It is distinguished whether the belt is moving or not ($\mathsf{fb.move}$) and whether there is a plate at the end of the belt ($\mathsf{fb.end}$). The substates of $\mathsf{fb.end}$ become active or inactive only if the feed belt is moving. Otherwise it is neither possible that a plate leaves the feed belt nor enters this specific location at the end of the feed belt.

The TLA-action $Fb.ert.unload$ describes that a plate leaves the end of the belt:

$$Fb.ert.unload \equiv \begin{array}{l} \wedge\ in(\mathsf{fb.end.plate}) \\ \wedge\ in'(\mathsf{fb.end.noplate}) \end{array}$$

Note that there is an implicit assumption that the plates arrive at the end of the feed belt with a sufficient distance. In a complete description of the cell this should be described explicitly.

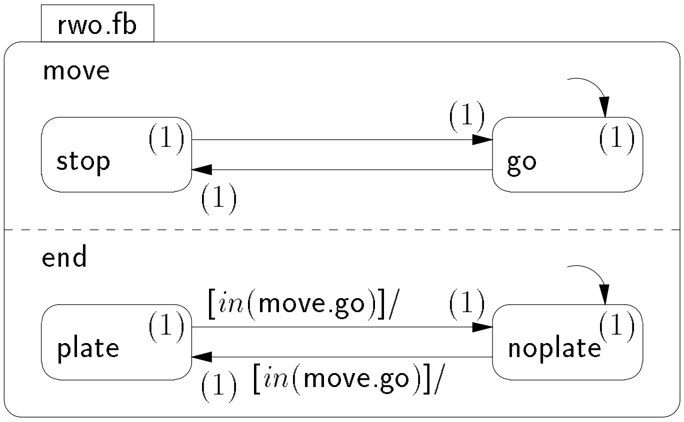



Figure 7.15: RW-O: Feed belt

**Elevating Rotary Table** The table has two designated vertical positions, one corresponding to the height of the feed belt (ert.vertical.fb) and another one corresponding to the height of the first arm of the robot (ert.vertical.arm). The table can also be outside of this range (ert.vertical.outside) which causes damage to the table. Hence in the statechart in Figure 7.16 there is no transition segment leaving this state. Similarly, the table can rotate with corresponding designated positions (ert.rotate). The table itself can be empty, loaded, or overloaded (ert.load). In contrast to [Schowalter, 1997] we allow that an overloaded table can be unloaded again. But note that it is possible to reach the state ert.load.overloaded from ert.load.empty in one step: It is possible that a plate is put on the table from the feed belt as well as from the arm of the robot. That overloading the table is clearly undesired behaviour is not relevant in this description.

To load or unload the table it is necessary, that the table is and remains in the correct position. This can be described by the following TLA-actions:

$$\begin{aligned} \mathit{Ert.fbpos} \equiv\ & \wedge\ in(\mathsf{ert.rotation.fb}) \wedge in'(\mathsf{ert.rotation.fb}) \\ & \wedge\ in(\mathsf{ert.vertical.fb}) \wedge in'(\mathsf{ert.vertical.fb}) \end{aligned}$$

$$\begin{aligned} \mathit{Ert.armpos} \equiv\ & \wedge\ in(\mathsf{ert.rotation.arm}) \wedge in'(\mathsf{ert.rotation.arm}) \\ & \wedge\ in(\mathsf{ert.vertical.arm}) \wedge in'(\mathsf{ert.vertical.arm}) \end{aligned}$$

As before loading or unloading the table is described by TLA-actions. Note that the first conjunct of *Ert.load* corresponds to two compound transitions:

$$\begin{aligned} \mathit{Ert.load} \equiv\ & \vee\ \wedge\ in(\mathsf{ert.load.empty}) \\ & \quad\ \wedge\ \neg\, in'(\mathsf{ert.load.empty}) \\ & \vee\ \wedge\ in(\mathsf{ert.load.loaded}) \\ & \quad\ \wedge\ in'(\mathsf{ert.load.overloaded}) \end{aligned}$$

$$\begin{aligned} \mathit{Ert.unload} \equiv\ & \wedge\ in(\mathsf{ert.load.loaded}) \\ & \wedge\ in'(\mathsf{ert.load.empty}) \end{aligned}$$

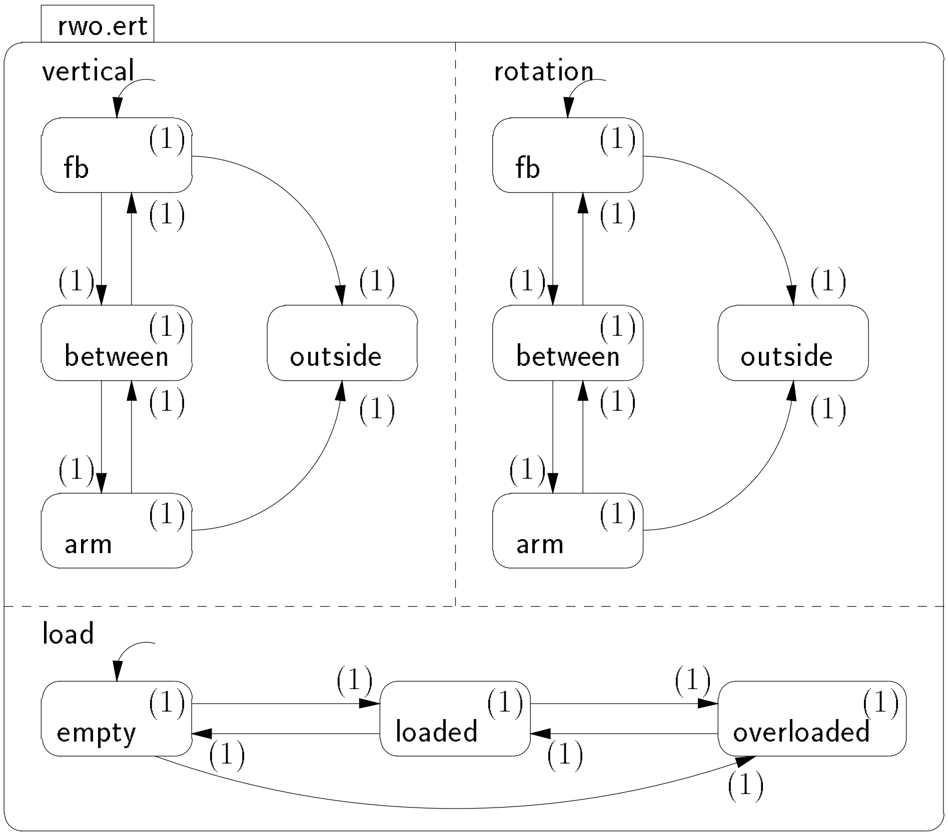



Figure 7.16: RW-O: Elevating rotary table

**Robot Arm** The robot can be described similarly as the table. Because the robot has two arms which are connected in a fixed angle and because the arms can collide with some of the devices its description is actually somewhat larger, but it is not necessary to present it here completely. It is sufficient to have TLA-actions

$$Arm1.\,ertpos$$
$$Arm1.\,load$$
$$Arm1.\,unload$$

which can be defined similarly to the corresponding ones of the table.

**Coupling the Devices** Up to now it was not described how a plate moves from one device to another. This can be done by combining the TLA-actions.

A plate is transported from the feed belt to the table if the feed belt is unloaded and the table is in the correct position:

$$\begin{aligned} Fb.\,to.\,ert \equiv\ & \wedge\ Fb.\,unload \\ & \wedge\ Ert.\,fbpos \\ & \Rightarrow Ert.\,load \end{aligned}$$

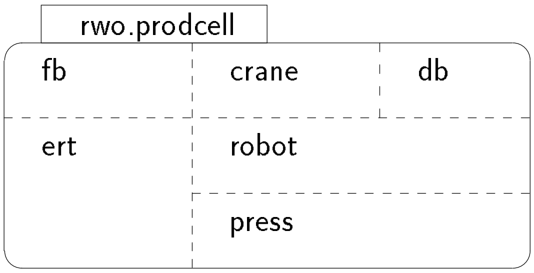



Figure 7.17: RW-O: Production cell

Similarly, the table is unloaded if both itself and the first arm of the robot are in the corresponding positions and the arm is loaded:

$$
\begin{array}{rl}
Ert.to.arm1 \equiv & \wedge\ Arm1.load \\
 & \wedge\ Arm1.ertpos \\
 & \wedge\ Ert.armpos \\
 & \Rightarrow Ert.unload
\end{array}
$$

Vice versa, a plate can be put back on the table by the robot:

$$
\begin{array}{rl}
Arm1.to.ert \equiv & \wedge\ Arm1.unload \\
 & \wedge\ Arm1.ertpos \\
 & \Rightarrow Ert.load
\end{array}
$$

On the other hand, these are the only possibilities to load or unload the table. This is expressed by the following TLA-actions which can be seen as putting a frame around the devices.

$$
\begin{array}{rl}
Ert.load.frame \equiv & Ert.load \\
 & \Rightarrow \vee \wedge\ Fb.unload \\
 & \quad\ \ \wedge\ Ert.fbpos \\
 & \quad \vee \wedge\ Arm1.unload \\
 & \quad\ \ \wedge\ Arm1.ertpos
\end{array}
$$

$$
\begin{array}{rl}
Ert.unload.frame \equiv & Ert.unload \\
 & \Rightarrow \wedge\ Ert.armpos \\
 & \quad\ \wedge\ Arm1.ertpos \\
 & \quad\ \wedge\ Arm1.load
\end{array}
$$

The descriptions of the devices can be combined by considering them as substates of an and-state prodcell, cf. Figure 7.17. The TLA-actions describing how plates are transported can be associated with the state rwo.prodcell.ert, because essentially the behaviour of the table is restricted. I.e. we have:

$$\mathsf{rwo.prodcell.ert} \rhd \begin{array}{l} \wedge\ \Box[\,Fb.\,to.\,ert\,]_{\langle\rangle} \\ \wedge\ \Box[\,Arm1.\,to.\,ert\,]_{\langle\rangle} \\ \wedge\ \Box[\,Ert.\,load.\,frame\,]_{\langle\rangle} \\ \wedge\ \Box[\,Ert.\,to.\,arm1\,]_{\langle\rangle} \\ \wedge\ \Box[\,Ert.\,unload.\,frame\,]_{\langle\rangle} \end{array} \tag{7.24}$$

Note that it is not described for example that a plate leaving the feed belt is put actually on the table, only the situations in which this can happen is described here.

**Invariants Instead of Next-State Relations** In this description we used next-state relations to restrict the behaviour of devices. But this can be done by using invariants, too. To achieve this we generate an event when a step corresponding to the TLA-actions of the devices is taken. For convenience, such an event will have the same name as the TLA-action. As an example, the event fb.ert.unload is generated when a $Fb.\,ert.\,unload$-step is taken. This can be described as part of the TLA-action itself, but this can be done here in a more simple way by generating this event when the compound transition from the state fb.end.plate to fb.end.noplate is taken. This can be done for all of the other actions, too, because each of them corresponds to a compound transition or static transition. In the case of the action $Ert.\,load$ the event has to be generated when taking any of the three transitions corresponding to $Ert.\,load$. The coupling of the actions can be rewritten to:

$$fb.\,to.\,ert_{ev} \equiv \begin{array}{l} \wedge\ \mathsf{fb.unload} \in E_{exists} \\ \wedge\ \mathsf{ert.fbpos} \in E_{exists} \\ \Rightarrow \mathsf{ert.load} \in E_{exists} \end{array}$$

Corresponding invariants can be associated with states as before, e.g.:

$$\mathsf{ert.load} \rhd \Box\, fb.\,to.\,ert_{ev}$$

### 7.8.2 Production Cell: Requirements

We consider here four different requirements on the elevating rotary table. The requirements of the other devices of the production cell are quite similar. There are two further requirements on the complete cell we do not consider at all. The first is related to efficiency, stating that no plate is longer in the cell than a certain amount of time. A second one is related to the implementation, stating that the implementations should be as flexible as possible.

The requirements considered here are:

1. The table must not leave the area between its lower and upper positions corresponding to the height of the feed belt and the arm of the robot, respectively. The rotation of the table has to be restricted similarly.

2. There must not be more than one plate on the table.

3. Plates must not be dropped between the feed belt and the table.

4. A plate must eventually be transported through the system and forged.

The first two requirements are quite easy to express. Both refer to a descendant of ert which must not become active. They can be expressed by associating the initial condition *false* with these states. As in the example of the lift system the initial conditions are inserted in a copy of RW-O.

$$\begin{array}{ll} \mathsf{req.ert.rotation.outside} & \rhd \mathit{false} \\ \mathsf{req.ert.vertical.outside} & \rhd \mathit{false} \\ \mathsf{req.ert.load.overloaded} & \rhd \mathit{false} \end{array} \tag{7.25}$$

The third requirement can be expressed in a similar way as the joint actions in RW-O. It was expressed there how a plate is transported from the feed belt to the table. Here we have to express that if the feed belt is unloaded, then the plate is put actually on the table:

$$\mathsf{req.prodcell} \rhd \Box[\mathit{Fb.\,unload} \;\Rightarrow\; \mathit{Ert.\,load}]_{\langle\rangle} \tag{7.26}$$

Note that this formula allows to overload the table. As in Section 7.8.2 the requirement can be expressed as an invariant on events generated when taking the transitions, too. We associated the formula with the state prodcell, because the behaviour of both devices is restricted actually. This is in contrast to RW-O, where there was causal dependency of the table on the feed belt (and on the robot).

The fourth requirement can be expressed by fairness restrictions on the TLA-actions corresponding to the movement of a plate. These are *Fb. to. ert* and *Ert. to. arm1* for transporting a plate from the feed belt to the arm via the table. We require strong fairness on both actions because it is possible that they are not enabled continuously without transporting the plate actually. As an example consider that the loaded table reaches the position suitable for the first arm of the robot and then starts to move down and up forever. In this case it is infinitely often possible to unload the table, provided that the robot is in the corresponding position and is not loaded itself, but the TLA-action *Ert. to. arm1* is not enabled continuously. Because there is no use in leaving this position without unloading the table it can be argued that this should be forbidden. As a consequence it would be possible to require weak fairness on the actions only. But this is actually stronger than the requirement given informally.

Similarly we require strong fairness on the actions corresponding to the movement of the table from the feed belt position to the robot position, provided that the table is loaded. This still does not imply that the position suitable for the robot to pick a table is reached infinitely often or even at all because it is not required that the vertical and the turning positions are reached together. Hence this has to be required separately by the following temporal constraint:

$$SF_{\langle\rangle}\begin{pmatrix} \wedge\ \neg\wedge\ in(\mathsf{ert.vertical.arm}) \\ \wedge\ in(\mathsf{ert.rotation.arm}) \\ \wedge\ \wedge\ in'(\mathsf{ert.vertical.arm}) \\ \wedge\ in'(\mathsf{ert.rotation.arm}) \end{pmatrix} \tag{7.27}$$

Again, there are alternatives to express the requirement. Instead of fairness-formulae it is possible to use leadsto-formulae to express that a loaded device will be unloaded eventually. But note that by such formulae further safety properties are implied, e.g. that the states outside, which cannot be left again, do not become active.

## 7.9 Summary

In this chapter the combined description technique was used to write several descriptions of two systems, namely a simple lift system and a part of the well-known case study production cell. The aim of this chapter was to give an impression of how descriptions written in the combination of statecharts with temporal constraints could look like.

For the first system, the lift system, we covered the complete range of descriptions from RW-O and REQ to DES. First, we showed that each of these descriptions could make use of both description techniques. Second, we showed that each of these descriptions was based on RW-O. This was most obvious for REQ and RW-M. Both of these descriptions are derived from RW-O by extending the statecharts part and by associating further temporal constraints. In total, we presented three different designs of a controller. The first one was already very close to pure statecharts, TLA-formulae were used mainly to define conditions and actions. The second one used temporal constraints to a large extend as a means of synchronous programming. The control logic of the third one consisted of a TLA-program associated with a rudimentary statechart. This showed that temporal constraints can be larger than just some lines.

For the second system, the production cell, we presented only a part of RW-O and REQ. We showed how temporal constraints could be used to synchronise steps of orthogonal parts of a system.

In the descriptions in this chapter it was not possible to describe the interfaces between parts of a system. Also, it was difficult to describe refinement relations between descriptions because it was not clear which variables are visible and which ones are local to a system. To resolve both problems we introduce activities and activity-charts in the next chapter.

# Chapter 8

# Activities

In the previous chapter the behaviour of a reactive system has been described by statecharts with temporal constraints. There, one description consists essentially of one large statechart. Although we presented parts of the descriptions as separate statecharts, these are not self-contained parts of the description. Especially there is no notion of the interface of a statechart.

*Activities* and *activity-charts* [Harel *et al.*, 1990] allow to describe self-contained parts of a description and the flow of information between these parts. The description of the flow of information can be seen as a description of the interface of activities.

Activities are relevant in the context of this work for two reasons: At first, they allow to distinguish between input, output, and local variables of an activity. The local variables are hidden in the behaviour of an activity. This is relevant when comparing two descriptions w.r.t. refinement. Runs should be compared w.r.t. the input and output variables, but not w.r.t. the local variables. At second, when checking whether descriptions are consistent it is necessary to determine which variable depends on which other one. This information can be derived to some extend from the flow of information between activities. For example, output variables of an activity can depend on input variables of the same activity, but not vice versa.

Therefore we introduce activities and activity-charts in this chapter. We start in Section 8.1 with an informal introduction to activities. In Section 8.2 we define static aspects of activities, i.e. a hierarchy of activities and the flow of information. In Section 8.3 we define when an activity becomes active or inactive and in Section 8.4 we define how the behaviour of an activity is derived from that of its subactivities. Temporal constraints can be associated with activities similar to states of a statechart, this is defined in Section 8.5. The refinement relation between activities is defined in Section 8.6. The lift example is continued in Section 8.7.

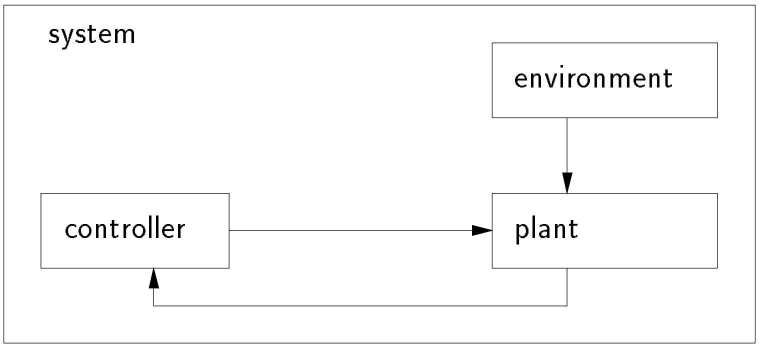



Figure 8.1: Control system described by activities

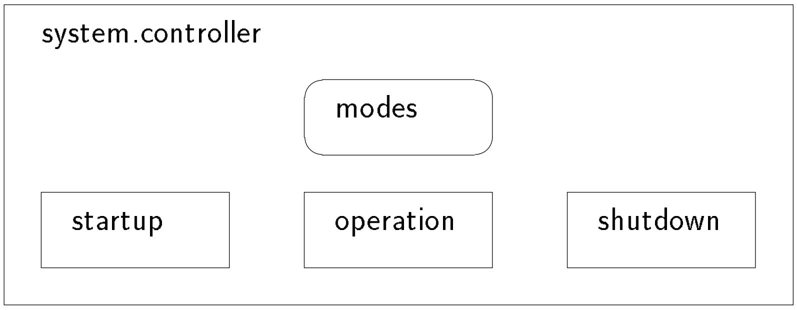



Figure 8.2: Modes of a controller

## 8.1 Informal Introduction to Activities

An activity can be used to describe the behaviour or a part of the behaviour of a system. The *behaviour* of an activity is a set of runs of the activity, where a *run* is a sequence of statuses of the activity. A *status* of an activity is similar to that of a statechart, except that it does not describe which of its subactivities is active. W.r.t. activation a status describes only whether the activity itself is *active* or not. Hence a status describes the values of data variables, which events exist, whether the activity is active, and the value of a variable *step.taken* similar as for statecharts.

As for statecharts there is a hierarchy of activities. As an example see the activity-chart in Figure 8.1. It corresponds to the general view of a control system as shown in Figure 1.1. The activity system consists of three subactivities environment, plant, and controller. These subactivities are active if and only if the activity system is active itself. The behaviour of an activity with subactivities is composed from the behaviour of its subactivities. The behaviour of system is the parallel composition of the behaviour of the subactivities.

It is also possible to describe that a behaviour is composed from the behaviour of one activity, followed by the behaviour of another activity. As an example see the activity-chart in Figure 8.2. In this activity-chart the activity system.controller consists of four subactivities startup, operation, shutdown, and modes. The activity modes is a special activity, called a *control-activity*. In an activity-chart control-activities are distinguished from ordinary activities by the rounded corners of their boxes. A control-activity has the ability to start and stop its siblings explicitly, i.e. modes can start and stop startup, operation, and shutdown. Vice versa a control-activity is able to monitor whether its siblings are active or not.

In this example the activities startup, operation, and shutdown are intended to describe the behaviour of the activity controller in startup mode, operational mode, and in shutdown mode. Under which conditions the mode of the controller can be changed is described in the control-activity modes. In addition, the activity modes has to start and stop the corresponding activities. The control-activity has to ensure in this example that at most one of its siblings is active. This does not need to be the case in general.

The behaviour of a control-activity itself can be described by a statechart or by a so-called *mini-spec*. A mini-spec is similar to a static reaction. Hence we will consider them as a special case of a statechart consisting of one state and its static reaction.

In the control-activities we use statecharts as they have been defined before: Especially we use the extensions by static transitions, cf. Section 4.2.3, by non-deterministic assignments, cf. Section 5.3, and by associating temporal constraints with states, cf. Section 6.4. The syntax of event and action expressions has to be extended to be able to monitor and control the siblings of the control-activity.

If an activity is stopped, all its subactivities are stopped, too. Vice versa, if all subactivities of an activity have been stopped, then the activity itself is stopped, too. An activity is able to stop without being commanded to do so. This is the case if the statechart of its control-activity reaches a special connector, a so-called *termination connector*. When an activity starts, then its direct subactivities are started, too, provided that the activity does not have a control-activity. This is the case in the activity-chart in Figure 8.1. If it has a control-activity this one becomes active in the next step, this means that the corresponding statechart takes a compound transition to reach an initial status. The siblings of the control-activity are not started. This is the case in the activity-chart in Figure 8.2.

In addition to start or to stop an activity it is possible to suspend and to resume it. As long as it is suspended it does not take steps.

An activity which has been started and is not stopped is called *active*. An activity which has been suspended and not resumed is called *hanging*.

The *information flow* between activities is described in an activity-chart by arrows or *flow-lines* between activities labeled with the names of data-variables and events. Let $v$ be the label of such a flow-line, then $v$ is an output of an activity from which the flow-line emanates and an input of an activity to which it points. Similar to transition segments each flow-line can emanate from several activities and it can point to several ones. If several activities have the same variable as output and change its value in the same step, one of these values is chosen non-deterministically.

Similar to connectors of transition segments there are connectors of flow-lines. Furthermore, there are rules how flow-lines ending at the border of an activity are extended from and towards, respectively, its subactivities. In this work neither these connectors nor these rules are relevant.

The flow-lines are used mainly to distinguish between inputs and outputs of an activity. Because the inputs and outputs can be variables as well as events we speak of input and output *entities*. The entities of an activity which are neither input nor output ones are called its local entities.

## 8.2 Static Aspects

An activity-chart consists of a non-empty set $A$ of activities, a set $V$ of variables, and a set $E$ of events. These sets are finite and pairwise disjoint.

The activities are related hierarchically in the form of a tree. In an activity-chart this relation is described by inclusion of the boxes corresponding to the activities. We use the predicate *subactivity* to denote this relation:

$$\mathit{subactivity}\colon A \times A$$

$\mathit{subactivity}(a_1, a_2)$ denotes that $a_1$ is a subactivity of $a_2$. The leaves of this tree are called *basic-activities*. The top-most activity is called the *root-activity*, it is denoted by *root*. The parent of an activity $a$ is denoted by $\mathit{parent}(a)$:

$$\mathit{parent}\colon A \setminus \{\mathit{root}\} \to A$$

Basic-activities have to be control-activities. Vice versa, control-activities do not have subactivities. Later on, the behaviour of an activity will be composed from that of the activities below it. We use a predicate *control. activity* to characterise control-activities:

$$\mathit{control.\ activity}\colon A$$

An activity $a$ can have at most one control activity. If it does not have one this is expressed by $\mathit{empty.\ control}(a)$.

$$\bigwedge_{a,b \in A} \begin{array}{l} \wedge\ a \neq b \\ \wedge\ \mathit{parent}(a) = \mathit{parent}(b) \\ \Rightarrow \neg \wedge\ \mathit{control.\ activity}(a) \\ \quad\quad \wedge\ \mathit{control.\ activity}(b) \end{array} \tag{8.1}$$

$$\mathit{empty.\ control}\colon A$$
$$\mathit{empty.\ control}(a) \equiv \bigwedge_{b \in A} \begin{array}{l} \mathit{subactivity}(b, a) \\ \Rightarrow \neg\, \mathit{control.\ activity}(b) \end{array}$$

A *controlled activity* is an activity, which has a control-activity as sibling. Note that this predicate is valid for control-activities, too.

$$\begin{aligned}&\mathit{controlled.\ activity}\colon A\\&\mathit{controlled.\ activity}(a) \equiv \bigvee_{b\in A} \begin{array}{l}\wedge\ \mathit{parent}(a) = \mathit{parent}(b)\\ \wedge\ \mathit{control.\ activity}(b)\end{array}\end{aligned}$$

The behaviour of a control-activity $a$ is described by a statechart. This statechart is denoted by $\mathit{chart}_a$.

The variables of an activity $a$ are denoted by $\mathit{variables}(a)$, i.e.:

$$\mathit{variables}\colon A \to V$$

In the following we use $\mathsf{v}$ to denote a variable from a set of variables belonging to an activity-chart and $v$ to denote the value of the variable named $\mathsf{v}$.

Instead of introducing flow-lines formally we define directly the interface of an activity and we define an explicit relation *flow* on variables to describe how information is propagated between activities.

The variables of an activity are partitioned into input, output, and local variables. These are denoted by using the following functions:

$$\begin{aligned}&\mathit{inputs}\colon A \to V\\&\mathit{outputs}\colon A \to V\\&\mathit{locals}\colon A \to V\end{aligned}$$

We require that these are pairwise disjoint and that their union is the set of all variables of an activity:

$$\begin{aligned}&\mathit{inputs}(a) \cap \mathit{outputs}(a) = \emptyset\\&\mathit{outputs}(a) \cap \mathit{locals}(a) = \emptyset\\&\mathit{locals}(a) \cap \mathit{inputs}(a) = \emptyset\\&\mathit{inputs}(a) \cup \mathit{outputs}(a) \cup \mathit{locals}(a) = \mathit{variables}(a)\end{aligned} \tag{8.2}$$

In a similar way we partition the events of an activity. $\mathit{events}(a)$ denotes all events of an activity $a$. $\mathit{input.\ events}(a)$, $\mathit{output.\ events}(a)$, and $\mathit{local.\ events}(a)$ denotes correspondingly the input, output, and local events of an activity $a$. As for variables we require that these sets are disjoint and their union is $\mathit{events}(a)$.

$$\begin{aligned}&\mathit{events}\colon A \to E\\&\mathit{input.\ events}\colon A \to E\\&\mathit{output.\ events}\colon A \to E\\&\mathit{local.\ events}\colon A \to E\end{aligned}$$

Furthermore we assume that each activity $a$ has three variables to hold events. These are the variables $E_{in,a} \in \mathit{inputs}(a)$, $E_{out,a} \in \mathit{outputs}(a)$, and $E_{exists,a} \in \mathit{locals}(a)$. We require that the variables $E_{in,a}$ and $E_{out,a}$ reflect correctly the input and output events in the set of all events relevant for an activity:

$$\begin{aligned} \mathit{events.\ relation}_a \equiv\ & \wedge\ E_{in,a} = E_{exists,a} \cap \mathit{input.\ events}(a) \\ & \wedge\ E_{out,a} = E_{exists,a} \cap \mathit{output.\ events}(a) \end{aligned}$$

Similarly, we assume that the local variables of each control-activity $a$ contain a variable $S_{active,a}$ to describe which states of the corresponding statechart are active.

We consider the sets of variables of two activities as disjoint, even if one is a subactivity of the other one. The relation between variables of two activities is denoted by *flow*. This relation is restricted to variables of activities which are hierarchically related: Each input variable of an activity has to be an input or local variable of its parent and each output variable has to be an output or local variable of its parent. A similar condition has to be valid for events. These restrictions on *flow* are captured by the following definitions and conditions.

$$\mathit{flow}: V \times V$$

$$\bigwedge_{a,b \in A, a \neq b} \mathit{variables}(a) \cap \mathit{variables}(b) = \emptyset \tag{8.3}$$

$$\bigwedge_{a,b \in A, v,w \in V} \begin{aligned} & \wedge\ b = \mathit{parent}(a) \\ & \wedge\ v \in \mathit{variables}(a) \\ & \wedge\ w \in \mathit{variables}(b) \\ & \Rightarrow \wedge\ \mathit{flow}(w, v) \ \Rightarrow\ \wedge\ w \in \mathit{inputs}(a) \\ & \qquad\qquad\qquad\qquad \wedge\ v \in \mathit{inputs}(b) \cup \mathit{locals}(b) \\ & \quad\ \ \wedge\ \mathit{flow}(v, w) \ \Rightarrow\ \wedge\ v \in \mathit{outputs}(a) \\ & \qquad\qquad\qquad\qquad \wedge\ w \in \mathit{outputs}(b) \cup \mathit{locals}(b) \end{aligned} \tag{8.4}$$

$$\bigwedge_{a,b \in A} \begin{aligned} & b = \mathit{parent}(a) \\ & \Rightarrow \wedge\ \mathit{flow}(E_{exists,b}, E_{in,a}) \\ & \quad\ \wedge\ \mathit{flow}(E_{out,a}, E_{exists,b}) \end{aligned} \tag{8.5}$$

The relation *flow* is restricted to variables of an activity and those of its parent. Variables of siblings in the hierarchy cannot be related directly. But because changes to variables become visible instantaneously, cf. Section 8.4, the changes can be propagated through chains of variables related by the relation *flow*. As a consequence, this restriction of *flow* does not impose a real restriction when modeling with activity- and statecharts.

$$\bigwedge_{v,w \in V} \bigwedge_{a,b \in A} \begin{aligned} & \wedge\ v \in \mathit{variables}(a) \\ & \wedge\ w \in \mathit{variables}(b) \\ & \wedge\ \mathit{flow}(v, w) \\ & \Rightarrow \vee\ a = b \\ & \quad\ \ \vee\ \mathit{subactivity}(a, b) \\ & \quad\ \ \vee\ \mathit{subactivity}(b, a) \end{aligned} \tag{8.6}$$

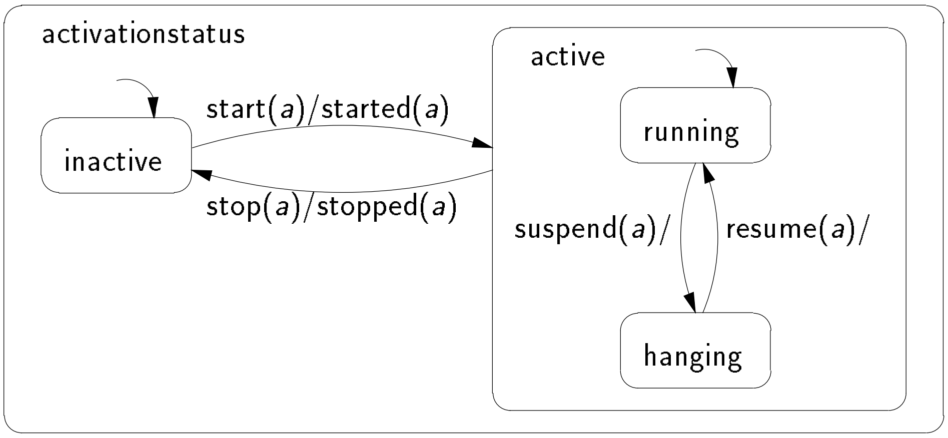



Figure 8.3: Approximation of the activation status of an activity

## 8.3 Active and Inactive Activities

Whether an activity $a$ is active can be influenced by a sibling of $a$ which is a control-activity e.g. by generating special events. Whether $a$ is active depends also on the activation status of other activities in the hierarchy. In this section we define, when an activity is active.

Because the different possibilities to influence whether an activity is active interfere we start in Section 8.3.1 with an informal explanation of how the activation status of an activity can be changed. Some difficult or even pathological cases are explained by examples. The formal definition will be given in Section 8.3.2. Sections 8.3.1 and 8.3.2 are based on the descriptions in [stm91, 1991, Harel and Naamad, 1996], and on the behaviour as implemented in the tool STATEMATE[1], cf. [Harel *et al.*, 1990].

### 8.3.1 Informal Description of Activation Status

A control-activity is able to influence whether one of its siblings $a$ is active by generating one of the following events: start($a$), stop($a$), suspend($a$), and resume($a$). Whether an activity is active depends also on the activation status of other activities in the hierarchy, e.g. when an activity is stopped, all its subactivities are stopped, too.

Which different activation statuses of an activity exist and which transitions between them are possible is depicted roughly by the statechart[2] in Figure 8.3. According to this statechart restarting an already active activity $a$ by generating the event start($a$) or stopping a stopped one does not influence the activity. Especially, a

[1]The version used was STATEMATE MAGNUM 1.0.3

[2]This use of a statechart corresponds to the explanatory use of diagrams in [Lamport, 1995]. It is not used here as basis for describing the semantics of activities formally.

hanging activity remains hanging when the event start(*a*) occurs. Note, that this statechart is only an approximation to illustrate the different activation statuses, it does not illustrate correctly the behaviour, when contradicting events, such as e.g. start(*a*) and stop(*a*), occur simultaneously. Neither does this statechart reflect the relation between the activation status of hierarchically related activities.

In the following, we will consider only whether an activity is active or not. It should become clear, how this can be extended to suspending and resuming activities.

Being more precise, whether an activity *a* is active or not, can be influenced in the following, possibly interfering, ways in activity-charts:

**Hierarchy:** by relations between the activation status of hierarchically related activities,

**Events:** by explicit start(*a*) and stop(*a*) commands,

**Termination Connector:** by reaching a *termination connector* in the statechart of a control-activity,

**Active Throughout:** by relating it to a state *s* of a statechart, i.e. *a* is active throughout *s*.

**Hierarchy** Basically, an activity can be active only if its parent activity is active, too. Furthermore, an activity without active subactivities must be inactive itself. It remains to clarify, what happens when an activity is activated, i.e. which of its subactivities is activated, too? We distinguish between activities with and without a control-activity.

In the case of an empty control-activity, all subactivities become active together with the activity itself. If an activity has a non-empty control-activity, then none of its subactivities is started in this step, not even the control-activity. In the next step of the activity its control-activity is started, which means that its associated statechart takes a compound transition to reach an initial status.[3] The siblings of the control-activity are not started by default. They are started only if commanded explicitly by the control-activity. Note that default transitions of the statechart of the control-activity can have corresponding action expressions. In this case these activities become active when the statechart of the control-activity is initialised. This implies, that siblings of control-activities can become active only one step after activating their parent.

[3]This describes the actual behaviour as can be observed of the tool STATEMATE. In [stm91, 1991, page 4-6] it is stated that '*if the activity that is started has a controlling activity, the action will cause the statechart "running" in parallel to all other statecharts that are active at that time.*'. This seems to indicate, that the statechart of the control-activity enters a default configuration immediately. In the following we use the semantics as implemented and described above.

**Events** The action expressions start($a$) and stop($a$) cause the activity $a$ to start or stop, respectively. According to the literature this happens immediately, i.e. in the same step. Which of the subactivities of $a$ become active too, has been discussed directly above. Because action expressions are executed at the end of a step, an activity commanded to stop still executes its step before stopping.

These action expressions can be executed only by the statechart associated with a control-activity for its siblings, but neither for its parent, for itself, or for subactivities of its siblings.

Starting an already started activity and stopping a stopped one are considered as null operations. It is possible that both action expressions are executed in the same step. In this case we neglect the action expression corresponding to the current activation status and take the other one into account, i.e. an active activity will be stopped and a stopped one will be activated. We decided this way, because changing the activation status twice in a step should be avoided and neglecting both actions cannot be justified.

**Termination Connector** A termination connector, drawn T in a statechart, can be used as a target of a transition segment in a statechart in the same way as an ordinary state. There must not be transition segments leaving it, hence it can be considered a final state. When a termination connector is reached, the parent of the corresponding control-activity is stopped.[4] We consider $\mathsf{T}_a$ as the distinguished name of this termination connector. Although it is called a connector we consider $\mathsf{T}_a$ as a state of this statechart.

Note, that starting a control-activity in a step does not imply, that it is active afterwards, even if its parent remains active. It is possible to use a termination connector as target state of the default transition, as a consequence, the control-activity and its parent become inactive.

**Active Throughout** $a$ throughout $s$ means that upon entrance of a state $s$ the activity $a$ is activated and that it is deactivated when leaving $s$. We take the same approach as in [stm91, 1991]: We associate the action expression start($a$) with the static reaction when entering $s$ and similarly stop($a$) with the one when exiting $s$. The activity $a$ is not restarted if it is already active when entering $s$. It is possible that the activity stops out of other reasons before $s$ is left.

One activity can be active throughout several different states. This is no problem when defining the semantics, but it can be asked, whether this is a useful description.

[4]Another possibility would be to stop only the control-activity. But according to [stm91, 1991, page 4-8] its parent is stopped. In this situation, starting an activity when taking a transition segment to a termination connector, as shown in [stm91, 1991] can be useful only, when the started activity does something just when being activated. This started activity must be a sibling of the control-activity. It will be stopped together with its parent and the control-activity in this step and cannot perform any further steps besides starting up.

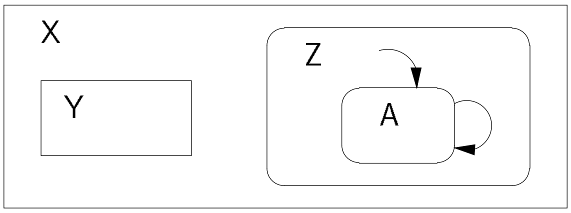



Figure 8.4: Example of pathological behaviour

We do not provide a semantics for *a* within *s*, which describes that the activity *a* becomes active while being in state *s*. This is considered as an intermediate specification in [stm91, 1991]. If necessary, it can be defined formally by associating the action stop(*a*) with the static reaction when exiting *s*. That the activity eventually starts is more problematic to define, because it is possible that the parent of *a* is deactivated before *a* becomes active. We suggest to express this explicitly by associating fairness properties with *s*.

Let us point out, that there can be non-intuitive behaviours using the construct throughout. Consider the activity-chart in Figure 8.4 and assume that Y throughout A. In this figure we have drawn the statechart of the control-activity Z directly in its box. Hence A is a state of a statechart. After entering, the state A is active in each configuration, nevertheless, the activity switches repeatedly between being active and inactive. This behaviour can be derived from the explanations above as follows: In each step, the state A is exited and reentered again, hence both actions start(Y) and stop(Y) have to be performed. But the action corresponding to the current activation status will be ignored, leading to repeatedly changing the activation status of Y.

Because **Active throughout** has been reduced to generate events, only the first three cases have to be considered when combining the different possibilities to influence an activity. Because termination connectors are relevant only if the corresponding control-activity is active already, termination connectors and the hierarchy of activities do not lead to contradictions. Events will be considered only when these do not contradict the other cases, especially the hierarchy of events.[5]

## 8.3.2 Formal Description of Activation Status

One tempting approach to define when an activity is active is to expand an activity-chart to one large statechart and define its semantics as the behaviour of this statechart. For this purpose one could start with a statechart as shown in Figure 8.3 for the root-activity of an activity-chart. The statechart of its control-activity and the recursively expanded subactivities would be inserted as orthogonal substates of

[5]In the tool StateMate it is possible to reach a configuration, in which an activity is active, but its parent is inactive. This situation is detected by the tool and reported as an error. In the formal semantics provided here, this situation does not occur.

the state running. The transition segment from hanging to running would have to be changed to one ending at a deep-history connector within running. By this construction the hierarchical relation of activities and the conditions on the activation status of different activities is mapped to an appropriate hierarchy of states. An activity is active iff the state active corresponding to it is active.

Unfortunately, the semantics obtained in this way differs substantially in two points from the published ones. At first, when suspending and resuming an activity via the deep-history connector, the states of the statecharts of control-activities are explicitly left and entered, respectively. As a consequence, the static reactions associated with leaving and entering these states are taken, but this should not be the case according to [stm91, 1991]. At second, when stopping an activity there would be a delay of one step before the activity actually stops. But according to [Harel and Naamad, 1996] an activity is stopped in the same step in which it is commanded to do so. It would be necessary to introduce intermediate steps handling the stop($a$)-events.

Therefore we take an alternative approach. For each activity $a$ we consider a variable $active_a$. Its value is $true$ if the activity is active. The value of $active_a$ is defined by a TLA-program. This TLA-program considers the values of the corresponding variables of the parent and of the subactivities of $a$, the variable $step.taken_a$, and whether the events start($a$) and stop($a$) are generated in a step.

Accessing the variables $active_b$ of the subactivities of an activity $a$ is hidden by an auxiliary predicate $self.terminated$. An activity $a$ is considered self-terminated, if all its subactivities are inactive, or if it is the parent of a self-terminated control-activity. In the first case we have to take into account that an activity does not need to have a control-activity. A control-activity itself is considered self-terminated, if its associated statechart reached a termination connector.

$$
\begin{array}{l}
self.terminated\colon A \\
self.terminated(a) \equiv \begin{array}[t]{l}
\vee \begin{array}[t]{l}
\wedge\ \neg\ control.activity(a) \\
\wedge\ empty.control(a) \\
\wedge\ \bigwedge\limits_{b \in A} \begin{array}[t]{l} subactivity(b, a) \\ \Rightarrow \neg\ active_b \end{array}
\end{array} \\
\vee \bigvee\limits_{b \in A} \begin{array}[t]{l}
\wedge\ subactivity(b, a) \\
\wedge\ control.activity(b) \\
\wedge\ \neg\ active_b
\end{array} \\
\vee \begin{array}[t]{l}
\wedge\ control.activity(a) \\
\wedge\ in(\mathsf{T}_a)
\end{array}
\end{array}
\end{array}
$$

In the TLA-program it is checked whether the events start($a$) and stop($a$) are contained in the set of events generated in a step to avoid the delay in starting or stopping an activity as described before.

Changing the value of the variable $active_a$ is described by the TLA-action $Active_a$.

The value of $active_a$ changes from $true$ to $false$, if $a$ is self-terminated, if the parent of $a$ becomes inactive, or if $a$ is stopped explicitly. It changes from $false$ to $true$ if the parent of $a$ remains or becomes active. It changes also from $false$ to $true$ if the parent of $a$ becomes active and $a$ is commanded to become active or if $a$ is a control-activity. If the parent of $a$ becomes active then $a$ becomes active, too, if it is not a controlled activity. In all other cases, the value of $active_a$ does not change. Note that the values can change only, when steps of the activity are taken. This is indicated by a change of the variable $step.\,taken_a$.

$Active_a$:

$$
\begin{array}{lll}
Active_a \equiv & \wedge & step.\,taken_a \neq step.\,taken'_a \\
& \wedge & active_a \\
& & \Rightarrow \neg\, active'_a \Leftrightarrow \begin{array}[t]{l} \vee\ self.\,terminated'(a) \\ \vee\ \neg\, active'_{parent(a)} \\ \vee\ \mathsf{stop}(a) \in E'_{in,a} \end{array} \\
& \wedge & \neg\, active_a \\
& & \Rightarrow active'_a \Leftrightarrow \begin{array}[t]{l} \wedge\ active'_{parent(a)} \\ \wedge\ active_{parent(a)} \Rightarrow \begin{array}[t]{l} \vee\ \mathsf{start}(a) \in E'_{in,a} \\ \vee\ control.\,activity(a) \end{array} \\ \wedge\ \neg\, active_{parent(a)} \Rightarrow \neg\, controlled.\,activity(a) \end{array}
\end{array}
$$

Activities are considered to be inactive initially. Hence, the value of the variable $active_a$ is described by the following TLA-program $activation.\,status_a$:

$activation.\,status_a$:

$$
\begin{array}{ll}
activation.\,status_a \equiv & \wedge\ \neg\, active_a \\
& \wedge\ \Box[\,Active_a\,]_{active_a}
\end{array}
$$

Note that we do not require any fairness assumptions in the TLA-program. This cannot be done, because taking steps depends on changing the value of $step.\,taken_a$ and this is not under control of an activity itself. Requiring fairness would be harmful when embedding the activity in another one which does not start the embedded one.

Whether an activity $a$ is active or whether $a$ becomes active or inactive in a step is made visible in an activity-chart by a boolean variable $\mathsf{active}(a)$ and by events $\mathsf{started}(a)$ and $\mathsf{stopped}(a)$. These are output entities of $a$, i.e. $\mathsf{active}(a) \in outputs(a)$ and $\{\mathsf{started}(a), \mathsf{stopped}(a)\} \subseteq output.\,events(a)$.

These variables and events are related to the variable $active_a$ by the following TLA-program. Generating the events is related to changes of $active_a$. This is well-defined only for non-control activities. A non-control activity cannot become active and inactive in one step, which is possible for control-activities. Therefore we use this

TLA-program for non-control activities only.

$$
\begin{array}{l}
\mathit{activation.events}_a\text{:} \\
\mathit{activation.events}_a \\
\equiv \wedge\ E_{exists,\mathsf{a}} \cup \{\mathsf{started}(\mathsf{a}), \mathsf{stopped}(\mathsf{a})\} = \emptyset \\
\quad \wedge\ \Box \begin{bmatrix} \wedge\ \mathsf{started}(\mathsf{a}) \in E'_{out,\mathsf{a}} & \Leftrightarrow & \wedge\ \neg\, \mathit{active}_a \\ & & \wedge\ \mathit{active}'_a \\ \wedge\ \mathsf{stopped}(\mathsf{a}) \in E'_{out,\mathsf{a}} & \Leftrightarrow & \wedge\ \mathit{active}_a \\ & & \wedge\ \neg\, \mathit{active}'_a \end{bmatrix}_{\langle\rangle} \\
\quad \wedge\ \Box\, \mathit{active}(\mathsf{a}) = \mathit{active}_{\mathsf{a}}
\end{array}
$$

The invariant means that the values of the variables $\mathsf{active}(\mathsf{a})$ and $\mathit{active}_{\mathsf{a}}$ coincide.

To suspend and to resume activities can be defined in a similar way. Fewer cases have to be considered because there is no analogous construct to termination connectors. Instead, it has to be taken care of the relation between suspending/resuming and activating/deactivating an activity. As already depicted in the statechart in Figure 8.3 the latter one has priority over the first one. No events are generated when an activity is suspended or resumed.

In the following we assume that variables $\mathit{hanging}_{\mathsf{a}}$ and $\mathsf{hanging}(\mathsf{a})$ for an activity $\mathsf{a}$ are defined and that the changes of their values are described by the TLA-programs $\mathit{activation.status}_{\mathsf{a}}$ and $\mathit{activation.events}_{\mathsf{a}}$, too.

## 8.4 Semantics

In the previous section it has been defined, when an activity is active. In this section the behaviour of an activity is defined. We start with the formal definition in Section 8.4.1. In Section 8.4.2 we consider activities as closed systems. In Section 8.4.3 we give a short overview over another formal definition of the semantics of activities.

### 8.4.1 Behaviour of an Activity

We use a boolean variable $\mathit{step.taken}_{\mathsf{a}}$ for each activity $\mathsf{a}$ to indicate whether it has taken a step. The behaviour of an activity is a set of runs over $\mathit{active}_{\mathsf{a}}$, $\mathit{hanging}_{\mathsf{a}}$, $\mathit{step.taken}_{\mathsf{a}}$, and $\mathit{variables}(\mathsf{a})$.

$$\mathit{status}_{\mathsf{a}} \equiv \langle \mathit{variables}(\mathsf{a}), \mathit{active}_{\mathsf{a}}, \mathit{hanging}_{\mathsf{a}}, \mathit{step.taken}_{\mathsf{a}} \rangle$$

All active activities take their steps simultaneously [Harel and Naamad, 1996]. I.e. *maximal parallelism* is used as model of concurrency, this can be called *lock-step parallelism*, too, because steps in parallel active activities are locked together.

We do not distinguish in this work between the different models of time, i.e. the *synchronous* and *asynchronous* model, described in [Harel and Naamad, 1996]. These are relevant only to determine when an internal clock is advanced with respect to taking steps. Because we do not consider clocks at all it is not necessary to make this distinction. Nevertheless, according to the semantics presented here, activity- and statecharts are reacting to external changes as soon as possible, hence the semantics corresponds to the asynchronous model of time.[6]

**Inactive Activities** It is relatively easy to understand the behaviour of an active activity in terms of its active subactivities. But it is not clear how the behaviour of an inactive activity has to be defined. It is not sufficient to require that its output variables do not change. This is not even sufficient for hanging activities.

Four questions have to be answered for such activities:

1. When are the values of the variables $active_a$ and $hanging_a$ changed?
2. What happens to the values of the (ordinary) output variables?
3. What happens to events generated just before stopping?
4. To reactivate a control-activity a default configuration of it must be entered which implies that its set of active states must be empty before this can be done, cf. Section 4.3. When does this happen?

These questions are not answered clearly in [Harel and Naamad, 1996] or not all because they are specific to the formalisation of the semantics presented here.

The second and third question can be answered based on the assumption, that an activity actually performs a step just before stopping and that the effect of this step must be available to other activities. Hence output variables still have to be changed and output events have to be generated.

In the Sections 4 and 5 the new status, the static reactions and the meaning of temporal constraints have been defined completely in terms based on the compound transitions taken in a step. It would be possible to adapt the definitions of *Step*, *triggered. events*, etc., to take into account when a control-activity $a$ is stopping, i.e. when $active'_a$ is false. But to avoid to change several definitions and to make them less perceptible, we take an alternative approach. We define a separate TLA-action $Reset_a$ to describe all changes to corresponding local variables together in one place. A $Reset_a$-step has to be taken before the next ordinary step of an activity. It is a stuttering step w.r.t. the input and output variables of an activity.[7]

For an activity $a$, a TLA-action $Reset_a$ discards all events which are neither input nor output events. In addition, if $a$ is a control-activity, the set of active states of

[6]Note that asynchronous does not mean here that concurrency is modeled by *interleaving*. Synchrony is related here to take steps in synchrony with advancing an internal clock.

[7]These intermediate steps do not interfere with the temporal constraints associated with states, because it is explicitly required that *execute. step*-steps are taken which enter certain states or keeps them active.

the corresponding statechart is set to the empty set. A $Reset_a$-step does not change the value of $step.\,taken_a$ and it can be taken only if the activity is inactive:

$$
\begin{aligned}
&Reset\text{:}\\
&Reset_a \equiv \wedge\ step.\,taken'_a = step.\,taken_a\\
&\qquad\quad\ \wedge\ \neg\, active_a \wedge \neg\, active'_a\\
&\qquad\quad\ \wedge\ E'_{exists,a} = E_{exists,a} \cap (input.\,events(a) \cup output.\,events(a))\\
&\qquad\quad\ \wedge\ control.\,activity(a)\ \Rightarrow\ S'_{active,chart(a)} = \emptyset
\end{aligned}
$$

The values of the data-variables of an activity are not changed when it is inactive and a step is taken. Remember that the sets of variables of different activities are disjoint. Hence, only an activity itself can change its output variables, even if it is inactive. If the activity is a control-activity the corresponding statechart keeps its set of active states and does not take a step. Newly generated events have to be available for other still active activities, but only until the next step. Hence, the set of output events is explicitly set to the empty set when other activities take steps. This is done for hanging activities, too. Such steps are described by the TLA-action $Clear_a$.

$$
\begin{aligned}
&Clear_a\text{:}\\
&Clear_a \equiv \wedge\ step.\,taken'_a \neq step.\,taken_a\\
&\qquad\quad\ \wedge\ \vee\ hanging_a\\
&\qquad\qquad\ \vee\ \neg\, active_a \wedge \neg\, active'_a\\
&\qquad\quad\ \wedge\ E'_{exists,a} \setminus E'_{in,a} = \emptyset\\
&\qquad\quad\ \wedge \bigwedge_{v \in locals(a)} \begin{array}{l} \vee\ v = E_{exists,a} \\ \vee\ v' = v \end{array}\\
&\qquad\quad\ \wedge \bigwedge_{v \in outputs(a)} v' = v\\
&\qquad\quad\ \wedge\ control.\,activity(a)\ \Rightarrow\ \begin{array}{l} \wedge\ S_{active,chart(a)} = S'_{active,chart(a)} \\ \wedge\ step.\,taken_{chart(a)} = step.\,taken'_{chart(a)} \end{array}
\end{aligned}
$$

Answering the first question last, the values of the variables $active_a$ and $hanging_a$ are changed immediately when taking a step of an activity.

The behaviour of activities is described separately for control-activities and for activities with subactivities.

**Control-Activities** An ordinary step of an active control-activity $a$ can be derived from a step of $chart(a)$. The variables and events of $a$ which are also ones of $chart(a)$ are described by the step of $chart(a)$. The values of further variables of $a$ which are not refered to by $chart(a)$ are left open. These can be restricted e.g. by associating temporal constraints with $a$, cf. Section 8.5.

$$
\begin{array}{l}
\mathit{Control.step}_a\text{:}\\
\mathit{Control.step}_a \equiv \begin{array}[t]{l}
\land\ \mathit{step.taken}'_a \neq \mathit{step.taken}_a\\
\land\ \lor\ \mathit{active}_a\\
\quad\ \lor\ \land\ \neg\,\mathit{active}_a \land \mathit{active}'_a\\
\quad\quad\ \ \land\ S_{active,chart(a)} = \emptyset\\
\land\ \neg\,\mathit{hanging}_a\\
\land\ \mathit{Execute.step}_{chart(a)}\\
\land\ E'_{exists,a} \cap E'_{chart(a)} = E'_{exists,chart(a)}
\end{array}
\end{array}
$$

The first part of this TLA-action describes when a step can actually be taken. Note, that the set of active states of $chart(a)$ is required to be empty when the activity is activated in a step. By the last two conjuncts the effects of taking such a step are described.

The initial status of the control-activity corresponds to the initial status of the statechart.

$$
\mathit{control.initial}_a \equiv \begin{array}[t]{l}
\land\ \mathit{initial.status}_{chart(a)}\\
\land\ E_{exists,a} = \emptyset
\end{array}
$$

The behaviour of a control-activity is described by combining these TLA-actions and predicates with those describing "inactive" behaviours and the activation status. The local variables of $a$ are hidden by quantifying existentially over them. These local variables are those necessary to define the semantics of $chart(a)$[8] and $locals(a)$.

$$
\begin{array}{l}
\mathit{control.activity}_a\text{:}\\
\mathit{control.activity}_a\\
\equiv\ \ \exists\, \mathit{status}_{chart(a)}\\
\quad\ \ \exists v,\ \text{for all } v \in \mathit{locals}(a)\\
\quad\ \ \land\ \mathit{control.initial}_a\\
\quad\ \ \land\ \Box \begin{bmatrix}\land\ \lor\ \mathit{Reset}_a\\ \quad\ \lor\ \mathit{Clear}_a\\ \quad\ \lor\ \mathit{Control.step}_a\end{bmatrix}_{\langle \mathit{status}_a, \mathit{status}_{chart(a)}\rangle}\\
\quad\ \ \land\ \mathit{WF}_{\langle \mathit{status}_a, \mathit{status}_{chart(a)}\rangle}(\mathit{Reset}_a)\\
\quad\ \ \land\ \mathit{activation.events}_a\\
\quad\ \ \land\ \mathit{activation.status}_a
\end{array}
$$

The next-state relation describing effects of taking steps and the next-state relation in the definition of $\mathit{activation.status}_a$ describing when $a$ is active are coupled via the variable $\mathit{step.taken}_a$. Fairness can be required only on the steps necessary to enable reactivation of stopped activities. As explained before, fairness of taking steps of the activity in general must not be required.

[8] I.e. the variables $S_{active,chart(a)}$, $E_{exists,chart(a)}$, and $\mathit{step.taken}_{chart(a)}$.

**Restricting Changes of Statecharts** In Section 5.2.2 it has been left open to define for a statechart that its variables do not change their value without reason and that its events are not generated without reason. It has not been possible to define this restriction there because the distinction into input, output, and local entities has not been available. This distinction is relevant because only output and local entities are controlled by a statechart, but not input ones. We take the distinction of the entities of a control-activity as that of its statechart.

We require that each change of a local or output variable of a statechart $chart(\mathsf{a})$ for a control-activity $\mathsf{a}$ is caused by executing an action expression in a step. We give the definition of a corresponding TLA-action only for the case of action expressions, a similar one can be defined for non-deterministic assignments, cf. Section 5.3.

$$
\begin{array}{l}
No.\,spontaneous.\,change_{chart(\mathsf{a})}\colon 2^{CT \cup SR} \\
No.\,spontaneous.\,change_{chart(\mathsf{a})}(\mathcal{T}) \\
\equiv \bigwedge_{\mathsf{v} \in outputs(\mathsf{a}) \cup locals(\mathsf{a})} \begin{array}{l} v \neq v' \\ \Rightarrow \bigvee_{T_1 \in \mathcal{T}} \bigvee_{t \in T_1} \mathsf{v} \in \mathsf{w}_{\mathsf{out},action(t)} \end{array}
\end{array}
$$

For an explicit output or local event there has to be an action expression by which it is generated. For implicit events this is already restricted by the definitions of *configuration. events* and *Change. events*.

$$
\begin{array}{l}
No.\,spontaneous.\,events_{chart(\mathsf{a})}\colon 2^{CT \cup SR} \\
No.\,spontaneous.\,events_{chart(\mathsf{a})}(\mathcal{T}) \\
\equiv \bigwedge_{\mathsf{e} \in (outputs(\mathsf{a}) \cup locals(\mathsf{a})) \setminus \{\mathsf{E}_{\mathsf{impl},chart(\mathsf{a})}\}} \\
\qquad \begin{array}{l} e \in E'_{exists,chart(\mathsf{a})} \\ \Rightarrow \bigvee_{T_1 \in \mathcal{T}} \bigvee_{t \in T_1} e \in generated.\,events_{action(t)}( \begin{array}{l} S_{active,chart(\mathsf{a})}, \\ E_{exists,chart(\mathsf{a})}, \\ w_{in,action(t)} \,) \end{array} \end{array}
\end{array}
$$

These TLA-actions become part of the definition of *Step* as defined already in Section 5.2.2:

$$
\begin{array}{l}
\mathit{Step}: \\
\mathit{Step} \\
\equiv \bigvee\limits_{\mathcal{T} \subseteq CT} \begin{array}[t]{l}
\wedge\ \mathit{maximal.\,step}(\mathcal{T}, S_{active,chart(\mathsf{a})}, E_{exists,chart(\mathsf{a})}) \\
\wedge\ \mathit{Change.\,configuration}(\mathcal{T}) \\
\wedge\ E'_{exists,chart(\mathsf{a})} \cap E_{conf,chart(\mathsf{a})} = \mathit{configuration.\,events}(\mathcal{T}) \\
\wedge\ \bigwedge\limits_{T_1 \in \mathcal{T}} \bigwedge\limits_{t \in T_1} \mathit{Take}_{action(t)} \\
\wedge\ \bigwedge\limits_{r \in SR} \begin{array}[t]{l} \mathit{enabled.\,static.\,reaction}(r, S_{active,chart(\mathsf{a})}, E_{exists,chart(\mathsf{a})}, \mathcal{T}) \\ \Rightarrow \mathit{Take}_{action(r)} \end{array} \\
\wedge\ \mathit{No.\,spontaneous.\,change}(\mathcal{T}) \\
\wedge\ \mathit{No.\,spontaneous.\,events}(\mathcal{T})
\end{array}
\end{array}
$$

Note that if e.g. the change of the value of a variable is described by a temporal constraint associated with a state, there has to be already an action expression which corresponds to a non-deterministic assignment to this variable.

**Activities with Subactivities** The behaviour of an activity $\mathsf{a}$ with subactivities differs in two aspects from that of a controlled one. At first, the behaviour is based on that of the subactivities instead of the behaviour of a statechart. Basically, all subactivities take steps together with $\mathsf{a}$. Whether these are steps clearing outputs or ordinary steps depends on the activation status of $\mathsf{a}$ and of the subactivities of $\mathsf{a}$. At second, the relation between the values of the variables of $\mathsf{a}$ and of its subactivities has to be described. Problems arise because there can be flow-lines between output variables of two different subactivities and only one variable of $\mathsf{a}$. In the case that both subactivities change these variables to different values in one step, there must be a choice between these values. Simply requiring that variables connected by flow-lines have always the same value would give a contradictory status and the corresponding run would be excluded from the behaviour of $\mathsf{a}$.

Let us assume that there are output variables $\mathsf{v}_i$ of activities $\mathsf{b}_i$ and that these are related to the variable $\mathsf{u}$ of $\mathsf{a}$, i.e. $\mathit{flow}(\mathsf{v}_i, \mathsf{u})$. The variable $\mathsf{u}$ will be written in a step, if at least one of the variables $\mathsf{v}_i$ is written in this step. Whether a variable $\mathsf{x}$ is written can be detected by occurrence of the event $\mathsf{wr}(\mathsf{x})$.[9] In this case the new value of $\mathsf{u}$ is the same as an arbitrary one of the $\mathsf{v}_i$, provided the chosen one has

[9] We require, that if $\mathsf{x}$ is an output variable of $\mathsf{a}$, then $\mathsf{wr}(\mathsf{x})$ has to be an output event of $\mathsf{a}$, too.

been written in this step, i.e.:

$$\begin{array}{l} \mathit{Propagate.\,outputs}_a: \\ \mathit{Propagate.\,outputs}_a \equiv \bigwedge_{u \in \mathit{locals}(a) \cup \mathit{outputs}(a)} \begin{array}{l} \vee \wedge \mathsf{wr}(u) \notin E'_{\mathit{exists},a} \\ \quad \wedge u = u' \\ \quad \wedge \bigwedge_{v \in V} \begin{array}{l} \mathit{flow}(v,u) \\ \Rightarrow \mathsf{wr}(v) \notin E'_{\mathit{out},\mathit{variables}^{-1}(v)} \end{array} \\ \vee \wedge \mathsf{wr}(u) \in E'_{\mathit{exists},a} \\ \quad \wedge \bigvee_{v \in V} \begin{array}{l} \wedge \mathit{flow}(v,u) \\ \wedge \mathsf{wr}(v) \in E'_{\mathit{out},\mathit{variables}^{-1}(v)} \\ \wedge u' = v' \end{array} \end{array} \end{array}$$

where $\mathit{variables}^{-1}(v)$ denotes the activity to which the variable $v$ belongs. This is a subactivity of $a$.

The events $\mathsf{ch}(u)$ have to be generated only, if the variable $u$ changes its value actually:

$$\begin{array}{l} \mathit{Propagate.\,implicit}_a: \\ \mathit{Propagate.\,implicit}_a \equiv \bigwedge_{u \in V} \begin{array}{l} \mathsf{ch}(u) \in E'_{\mathit{exists},a} \\ \Leftrightarrow \wedge u \in \mathit{variables}(a) \\ \quad\; \wedge u \neq u' \end{array} \end{array}$$

Connecting a variable to input variables of subactivities is more simple than connecting to outputs:

$$\begin{array}{l} \mathit{Propagate.\,inputs}_a: \\ \mathit{Propagate.\,inputs}_a \\ \equiv \bigwedge_{u \in \mathit{variables}(a)} \bigwedge_{v \in V} \begin{array}{l} \wedge \mathit{flow}(u,v) \\ \wedge \bigvee_{b \in A} \begin{array}{l} \wedge \mathit{subactivity}(b,a) \\ \wedge v \in \mathit{inputs}(b) \end{array} \\ \Rightarrow \wedge u' = v' \\ \quad\; \wedge \mathsf{rd}(u) \in E'_{\mathit{exists},a} \Leftrightarrow \mathsf{rd}(u) \in E'_{\mathit{out},b} \end{array} \end{array}$$

In contrast to the output variables and the events $\mathsf{wr}(x)$ we do not require, that the events $\mathsf{rd}(x)$ are output events of $a$ if $x$ is an input variable of $a$. Note, that by making these events visible a very detailed description of behaviours can be obtained. This might not always be appropriate, because some knowledge on the internals of an activity can be revealed, e.g. it can be distinguished whether an input variables or a local copy of it is accessed. Hence we suggest that these events are not included automatically in the set of possible output events, but only if they are used somewhere.

The input and output events of subactivities of $a$ are related to the set of events of

$a$ in a simple way. Similar to $events.\,relation_a$ we have

$$
\begin{aligned}
& propagate.\,events_a: \\
& propagate.\,events_a \\
& \equiv \wedge \bigwedge_{b \in \{c \in A \mid subactivity(c,a)\}} E_{in,b} = E_{exists,a} \cap input.\,events(b) \\
& \quad \wedge\ E_{exists,a} \cap \bigcup_{b \in \{c \in A \mid subactivity(c,a)\}} output.\,events(b) = \bigcup_{b \in \{c \in A \mid subactivity(c,a)\}} E_{out,b}
\end{aligned}
$$

The second conjunct of $Propagate.\,events_a$ states that $E_{exists,a}$ contains only output events of its subactivities which have been generated actually by the subactivities in this step.

In Section 8.2 some general conditions on the relationship between the variables of an activity and that of its subactivities have been given already. From the expositions above, we summarise the following further relationships for an activity $a$:

$$
\begin{aligned}
& \{\mathsf{active}(a), \mathsf{hanging}(a)\} \in outputs(a) \\
& \{\mathsf{started}(a), \mathsf{stopped}(a)\} \in output.\,events(a) \\
& \mathsf{wr}(x) \in output.\,events(a) \Leftrightarrow x \in outputs(a) \\
& \mathsf{rd}(x) \in output.\,events(a) \Rightarrow x \in inputs(a) \\
& \mathsf{ch}(x) \in output.\,events(a) \Rightarrow x \in outputs(a)
\end{aligned}
\tag{8.7}
$$

Using the relation *flow* as described above is more general than the use of flow-lines in [stm91, 1991], because it allows e.g. a renaming of variables. But it is possible also to join flow-lines in unnatural ways. For example there can be flow-lines of several output variables of one activity to one variable of its parent, which is not very intuitive.[10] This is especially the case when one would join flow-lines emanating from $\mathsf{active}(a)$ and $\mathsf{hanging}(a)$. But because this generality does not influence the semantics, we do not restrict the relation *flow* further.

A step of an activity $a$ with subactivities can be described formally as the conjunction of the different TLA-actions describing the propagation of events according to the relation *flow*. Furthermore, it is expressed that steps of the activity and of its subactivities are taken together.

$$
\begin{aligned}
& Composed.\,step_a: \\
& Composed.\,step_a \equiv \wedge\ Propagate.\,outputs_a \\
& \qquad\qquad\qquad\quad \wedge\ Propagate.\,implicit_a \\
& \qquad\qquad\qquad\quad \wedge\ Propagate.\,inputs_a \\
& \qquad\qquad\qquad\quad \wedge\ propagate.\,events_a \wedge propagate.\,events'_a \\
& \qquad\qquad\qquad\quad \wedge\ events.\,relation_a \wedge events.\,relation'_a
\end{aligned}
$$

[10] If this would actually have been the desired behaviour, it should be described inside the activity and only one variable should be presented as output.

$$\begin{array}{l}
\mathit{In.synchrony}_a \\
\equiv \bigwedge\limits_{b \in A} \begin{array}[t]{l} \mathit{subactivity}(b, a) \\ \Rightarrow \mathit{step.taken}_a \neq \mathit{step.taken}'_a \;\Leftrightarrow\; \mathit{step.taken}_b \neq \mathit{step.taken}'_b \end{array}
\end{array}$$

Initially, the set of events of $a$ is empty, we get the following TLA-program describing its behaviour. Besides the local variables of $a$, the input and output variables of its subactivities are hidden.

$$\begin{array}{l}
\mathit{composed.activity}_a\text{:} \\
\mathit{composed.activity}_a \\
\equiv\; \exists v,\ \text{for all } v \in \mathit{locals}(a) \\
\quad \exists\, \mathit{active}_b, \mathit{hanging}_b, \mathit{step.taken}_b,\ \text{with } \mathit{subactivity}(b, a) \\
\quad \exists v,\ \text{for all } v \in \mathit{inputs}(b) \cup \mathit{outputs}(b),\ \text{with } \mathit{subactivity}(b, a) \\
\quad \wedge\ \wedge\ E_{exists,a} = \emptyset \\
\quad\quad \wedge\ \mathit{propagate.events}_a \\
\quad\quad \wedge\ \mathit{events.relation}_a \\
\quad \wedge\ \Box \begin{bmatrix} \wedge\ \mathit{In.synchrony}_a \\ \wedge\ \mathit{Reset}_a \vee \mathit{Clear}_a \vee \mathit{Composed.step}_a \end{bmatrix}_{status_a} \\
\quad \wedge\ \mathit{WF}_{status_a}(\mathit{Reset}_a) \\
\quad \wedge\ \mathit{activation.events}_a \\
\quad \wedge\ \mathit{activation.status}_a
\end{array}$$

Finally, $\mathit{activity}_a$ describes the behaviour of an activity $a$ depending on its type.

$$\begin{array}{l}
\mathit{activity}_a\text{:} \\
\mathit{activity}_a \equiv \begin{array}[t]{l} \wedge\ \mathit{control.activity}(a) \Rightarrow \mathit{control.activity}_a \\ \wedge\ \neg\, \mathit{control.activity}(a) \Rightarrow \mathit{composed.activity}_a \end{array}
\end{array}$$

### 8.4.2 Activities as Closed Systems

Up to now we did not impose any fairness requirements on taking steps in activities because it depends on the parent of an activity to take a step. On the other hand, if the parent of an activity is fair w.r.t. taking steps, the activity itself is fair, too. Hence it is sufficient to consider fairness of the root of a hierarchy of activities.

We will have now a closer look at this root-activity. Consider the typical kind of reactive systems we are interested in, then their behaviour can be described by activities as depicted in Figure 8.1, page 158. The root-activity system can be considered as one converting the open systems controller, plant, and environment into a closed one.

When considering system as a closed system the variables $\mathit{active}_{\mathsf{system}}$, $\mathit{hanging}_{\mathsf{system}}$, and $\mathit{step.taken}_{\mathsf{system}}$ are hidden. We require that the activity system is activated and takes steps eventually. Because the variable $\mathit{step.taken}_{\mathsf{system}}$ changes only when one of the TLA-actions $\mathit{Clear}_{\mathsf{system}}$ or $\mathit{Composed.step}_{\mathsf{system}}$ is taken, it is sufficient to require weak fairness for a TLA-action changing $\mathit{step.taken}_{\mathsf{system}}$. This covers also the step in which system is activated. $Controlled.\,step_{\mathsf{system}}$ does not need to be considered here because system is an activity with subactivities, but not a control-activity.

$$
\begin{array}{l}
closed.\,system_{\mathsf{system}}:\\
closed.\,system_{\mathsf{system}}\\
\equiv\ \exists\, active_{\mathsf{system}}, hanging_{\mathsf{system}}, step.\,taken_{\mathsf{system}}\\
\quad \wedge\ activity_{\mathsf{system}}\\
\quad \wedge\ WF_{\langle\rangle}\,(\,step.\,taken_{\mathsf{system}} \neq step.\,taken'_{\mathsf{system}}\,)
\end{array}
$$

Conjoining this fairness assumption with the TLA-formula $activity_{\mathsf{system}}$ describing the behaviour of system should not be confused with associating a temporal constraint with a state as described in Section 6.2. Here, a fairness constraint on taking actually a step is explicitly given, whereas the fairness constraints in Section 6.2 are based on this assumption. In Section 8.5 we associate temporal constraints with activities.

### 8.4.3 Other Semantics of Activities

[Damm *et al.*, 1998] defines a compositional semantics of statecharts and as far as we know the first formal one of activity-charts. Compositionality is achieved by extending the set of observables of a statechart or an activity-chart. This extension allows to add interaction from the environment to a component.

[Damm *et al.*, 1998] focuses on the super-step semantics as it is supported by the simulation tool of StateMate. Because super-steps do not need to terminate divergence of steps is handled. It is distinguished between events generated by the environment and by the system and between data-items local to an activity and shared ones.

In addition to the events it generates or consumes and the variables it shares the interface of an activity is extended by variables which describe whether the system or its environment are willing to perform a step or a super-step. An additional event is used to signal clock ticks. These variables and events are used for synchronising the system and its environment.

Copies of events and variables which can be read and written by an activity-chart or statechart are added. These are used to resolve simultaneous write access to one variable. The actual resolution is done by so-called monitors, one for each variable or event which can be written by two or more activities. These monitors correspond to our TLA-action *Composed. step*.

The semantics presented in [Damm *et al.*, 1998] differs from ours because they use the super-step semantics whereas we use the single-step semantics. It is also more close to [Harel and Naamad, 1996] w.r.t. multiple assignments to one variable in one step of a statechart. In [Damm *et al.*, 1998] even divergent action expressions are allowed, whereas we restrict ourselves to a kind of guarded actions and in one step at most one value is assigned to one variable in a statechart.

The semantics itself is defined by constructing a transition system from a statechart, cf. [Manna and Pnueli, 1992]. This means to define the variables constituting the state space and to define the transition relation. Defining this transition relation directly allows to hide intermediate states as for example those occurring in executing a sequence of action expressions.

## 8.5 Associating TLA-Formulae with Activities

It can be convenient to restrict the behaviour of an activity directly by associating temporal constraints with it instead of restricting the behaviour of statecharts of control-activities below it. The semantics of associating a temporal constraint $F$ with an activity $\mathsf{a}$, written $\mathsf{a} \rhd F$, is defined quite similarly to associating $F$ with a state $\mathsf{s}$. We require that the variables used in $F$ are a subset of $variables(\mathsf{a})$. As a consequence it is neither possible to refer to local variables of subactivities, nor to the variables used to define the semantics, i.e. $active_{\mathsf{a}}$, $hanging_{\mathsf{a}}$, and $step.taken_{\mathsf{a}}$. The latter one is no real restriction because the variables $\mathsf{hanging}(a)$ and $\mathsf{active}(a)$ and the events $\mathsf{started}(a)$, $\mathsf{stopped}(a)$, etc. can be refered to.

To define the semantics of $\mathsf{a} \rhd F$ it is necessary to characterize when an activity $\mathsf{a}$ becomes active and when it remains active. The first of these conditions can be expressed simply as $\neg\, active_{\mathsf{a}} \wedge active'_{\mathsf{a}}$. For the second one it is distinguished between a safety- and a liveness-semantics. For the safety-semantics it is sufficient to require, that $\mathsf{a}$ remains active in steps of $\mathsf{a}$, hence using the safety-semantics temporal constraints have to be valid for hanging activities, too.[11] The definitions of $[\![\, \mathsf{a} \rhd F \,]\!]_s$ are quite similar to that of the safety-semantics of associating temporal constraints with states and are summarized here for completeness. Assume $p$ is a predicate, $\mathcal{N}$ a TLA-action, and $f$ a state function.

$$
\begin{aligned}
[\![\, \mathsf{a} \rhd p \,]\!]_s &\equiv \Box[\neg\, active_{\mathsf{a}} \wedge active'_{\mathsf{a}} \Rightarrow p']_{step.taken_{\mathsf{a}}} \\
[\![\, \mathsf{a} \rhd \Box[\mathcal{N}]_f \,]\!]_s &\equiv \Box[active_{\mathsf{a}} \wedge active'_{\mathsf{a}} \Rightarrow [\mathcal{N}]_f]_{step.taken_{\mathsf{a}}}
\end{aligned}
$$

On the other hand it has to be taken into account by the liveness-semantics, that the activity performs infinitely many steps in which it can do something, i.e. infinitely many steps in which it is active, but not hanging. Similar to the TLA-actions *Execute.stay* we define TLA-actions *Remain.active*:

$$
\begin{aligned}
&Remaining: \mathsf{A} \\
&Remaining(\mathsf{a}) \equiv \wedge\ active_{\mathsf{a}} \wedge active'_{\mathsf{a}} \\
&\qquad\qquad\qquad\quad \wedge\ \neg\, hanging_{\mathsf{a}} \wedge \neg\, hanging'_{\mathsf{a}}
\end{aligned}
$$

[11] This might cause problems when $F$ refers to the set of output events, which is cleared in the next $Clear_{\mathsf{a}}$-step.

$$Remain.\ active\colon \mathsf{A}$$
$$Remain.\ active(\mathsf{a}) \equiv \begin{array}{l} \wedge\ step.\ taken_{\mathsf{a}} \neq step.\ taken'_{\mathsf{a}} \\ \wedge\ Remaining(\mathsf{a}) \end{array}$$

$$remain.\ active_{\Box\Diamond}\colon \mathsf{A}$$
$$remain.\ active_{\Box\Diamond}(\mathsf{a}) \equiv \Box\Diamond\, Remain.\ active(\mathsf{a})$$

$$remain.\ active_{\Diamond\Box}\colon \mathsf{A}$$
$$remain.\ active_{\Diamond\Box}(\mathsf{a}) \equiv \begin{array}{l} \wedge\ Remain.\ active_{\Box\Diamond}(\mathsf{a}) \\ \wedge\ \Diamond\wedge\ \Box\left[\begin{array}{l} step.\ taken_{\mathsf{a}} \neq \neg\, step.\ taken'_{\mathsf{a}} \\ \Rightarrow Remaining(\mathsf{a}) \end{array}\right]_{status_{\mathsf{a}}} \\ \quad\ \wedge\ \Box\, Enabled\ step.\ taken_{\mathsf{a}} \neq \neg\, step.\ taken'_{\mathsf{a}} \end{array}$$

The definition of the liveness-semantics of associating temporal constraints with activities is again similar to that of associating temporal constraints with states, cf. Section 6.4.2. Remember that $*+$ ranges over the set $\{\Diamond\Box, \Box\Diamond\}$.

$$[\![\, \mathsf{a} \rhd_{*+} p\,]\!]_l^+ \equiv (Enabled\ Remain.\ active(\mathsf{a})) \wedge p$$
$$[\![\, \mathsf{a} \rhd_{*+} p\,]\!]_l^- \equiv (Enabled\ Remain.\ active(\mathsf{a})) \wedge \neg p$$
$$[\![\, \mathsf{a} \rhd_{*+} \Box[\mathcal{N}]_f\,]\!]_l^+ \equiv \Box[Remain.\ active(\mathsf{a})\ \Rightarrow\ [\mathcal{N}]_f]_{step.taken_{\mathsf{a}}}$$
$$[\![\, \mathsf{a} \rhd_{*+} \Box[\mathcal{N}]_f\,]\!]_l^- \equiv \Diamond\langle\, Remain.\ active(\mathsf{a})\ \Rightarrow\ \langle \neg\mathcal{N}\,\rangle_f\,\rangle_{step.taken_{\mathsf{a}}}$$
$$[\![\, \mathsf{a} \rhd_{*+} \neg F\,]\!]_l^+ \equiv [\![\, \mathsf{a} \rhd_{*+} F\,]\!]_l^-$$
$$[\![\, \mathsf{a} \rhd_{*+} \neg F\,]\!]_l^- \equiv [\![\, \mathsf{a} \rhd_{*+} F\,]\!]_l^+$$
$$[\![\, \mathsf{a} \rhd_{*+} F \wedge G\,]\!]_l^+ \equiv [\![\, \mathsf{a} \rhd_{*+} F\,]\!]_l^+ \wedge [\![\, \mathsf{a} \rhd_{*+} G\,]\!]_l^+$$
$$[\![\, \mathsf{a} \rhd_{*+} F \wedge G\,]\!]_l^- \equiv [\![\, \mathsf{a} \rhd_{*+} F\,]\!]_l^- \vee [\![\, \mathsf{a} \rhd_{*+} G\,]\!]_l^-$$
$$[\![\, \mathsf{a} \rhd_{*+} \Box F\,]\!]_l^+ \equiv remain.\ active_{*+}(\mathsf{a})\ \Rightarrow\ \Box[\![\, \mathsf{a} \rhd_{*+} F\,]\!]_l^+$$
$$[\![\, \mathsf{a} \rhd_{*+} \Box F\,]\!]_l^- \equiv remain.\ active_{*+}(\mathsf{a})\ \Rightarrow\ \Diamond[\![\, \mathsf{a} \rhd_{*+} F\,]\!]_l^-$$
$$[\![\, \mathsf{a} \rhd_{*+} F\,]\!]_l \equiv [\![\, \mathsf{a} \rhd_{*+} F\,]\!]_l^+$$

This concludes the definition of the semantics of activities and of associating TLA-formulae with it.

## 8.6 Refinement

Given two descriptions one can ask whether one of the descriptions refines the other one, i.e. whether it describes essentially the same, but provides more detail. In our context this means to ask whether an activity $\mathsf{c}$ refines an activity $\mathsf{a}$, i.e. whether the behaviour of $\mathsf{c}$ is a subset of the behaviour of $\mathsf{a}$. We call $\mathsf{c}$ the concrete activity and $\mathsf{a}$ the abstract activity.

In TLA refinement corresponds to logical implication, cf. Section 3.3. Hence, that an activity $c$ refines an activity $a$ can be described essentially by

$$activity_c \Rightarrow activity_a \tag{8.8}$$

This kind of refinement is called *property refinement* in [Broy, 1992].

In this section this definition is adapted to our context by requiring additional properties on the variables of $a$ and $c$. On the one hand, they have to coincide on the distinction between input and output variables. On the other hand, the variables related to the activation status of the activities are different. An appropriate refinement mapping has to be defined.

**Inputs and Outputs** We start by comparing the "ordinary" input and output variables. The variables $\mathsf{hanging}(c)$, $\mathsf{active}(c)$, $E_{in,c}$, and $E_{out,c}$ and the corresponding ones of the activity $a$ are considered separately.

It is possible that the concrete activity $c$ has a larger interface than the abstract one, i.e. it makes more variables visible. This means that the inputs of $a$ are a subset of the inputs of $c$ and similarly for the outputs:

$$\begin{aligned} &inputs(a) \setminus \{E_{out,a}\} \subseteq inputs(c) \setminus \{E_{out,c}\} \\ &outputs(a) \setminus \{\mathsf{hanging}(a), \mathsf{active}(a), E_{out,a}\} \\ &\quad \subseteq outputs(c) \setminus \{\mathsf{hanging}(c), \mathsf{active}(c), E_{out,c}\} \end{aligned} \tag{8.9}$$

Note, that an input variable of $a$ cannot become an output variable of $c$. By Equation 8.9 the variable has to be an input of $c$ and Equation 8.2 states that inputs and outputs of $c$ are disjoint. Analogously, an output of $a$ cannot become an input of $c$.

We require a similar condition as Equation 8.9 on input and output events:

$$\begin{aligned} &input.\,events(a) \setminus \{\mathsf{start}(a), \mathsf{stop}(a), \mathsf{suspend}(a), \mathsf{resume}(a)\} \\ &\quad \subseteq input.\,events(c) \setminus \{\mathsf{start}(c), \mathsf{stop}(c), \mathsf{suspend}(c), \mathsf{resume}(c)\} \\ &output.\,events(a) \setminus \{\mathsf{started}(a), \mathsf{stopped}(a)\} \\ &\quad \subseteq output.\,events(c) \setminus \{\mathsf{started}(c), \mathsf{stopped}(c)\} \end{aligned} \tag{8.10}$$

**Variables Related to the Activation Status** Because the variables related to the activation status differ an appropriate refinement mapping has to be defined. For each such variable $v$ of $a$ we define a state function $\overline{v}$ in terms of the corresponding variable of $c$. In the following equation we use $[e/f]$ to denote substitution of the event $e$ by the event $f$.

$$\begin{aligned}
\overline{\mathsf{active}(a)} &\equiv \mathsf{active}(c) \\
\overline{\mathsf{hanging}(a)} &\equiv \mathsf{hanging}(c) \\
\overline{E_{in,a}} &\equiv (E_{in,c} \cap input.\,events(a))[\ \mathsf{start}(c)/\mathsf{start}(a), \\
&\qquad \mathsf{stop}(c)/\mathsf{stop}(a), \\
&\qquad \mathsf{suspend}(c)/\mathsf{suspend}(a), \\
&\qquad \mathsf{resume}(c)/\mathsf{resume}(a)\ ] \\
\overline{E_{out,a}} &\equiv (E_{out,c} \cap output.\,events(a))[\ \mathsf{started}(c)/\mathsf{started}(a), \\
&\qquad \mathsf{stopped}(c)/\mathsf{stopped}(a)\ ]
\end{aligned} \tag{8.11}$$

The variables $\mathsf{active}(a)$, $\mathsf{hanging}(a)$, $E_{in,a}$, $E_{out,a}$ are local variables of the activity $a$. As usual in TLA they are hidden by quantifying existentially over them. It remains visible whether an activity is active or hanging and whether an activity takes a step. Therefore the variables $active_a$, $hanging_a$, and $step.\,taken_a$ are not hidden. To show that the activity $c$ refines the activity $a$ it is sufficient to show

$$\begin{aligned} activity_c \;\Rightarrow\;& \exists\ \mathsf{hanging}(a), \mathsf{active}(a), E_{in,a}, E_{out,a} \\ & activity_a \end{aligned} \tag{8.12}$$

To show the property in the previous equation we have to define refinement mappings for the existentially quantified variables. To relate the behaviour of the activities $a$ and $c$ we take the refinement mappings as defined in Equation 8.11. This means that to show that the activity $c$ refines the activity $a$ it is sufficient to show

$$activity_c \;\Rightarrow\; \overline{activity_a}^{8.11} \tag{8.13}$$

where $\overline{activity_a}^{8.11}$ is the TLA-program $activity_a$ with all free occurrences of the variables in Equation 8.11 substituted as defined in Equation 8.11.

In the next section we use this refinement relation to express relations between the different kinds of descriptions.

# 8.7 Lift Example (Cont.)

Each of the various descriptions of the lift example in Chapter 7 has been written as one large statechart. In this section we show, how activities can be used to structure the descriptions conveniently and to clarify the interfaces between different parts of the system. Relations between the different descriptions are stated in terms of the refinement relation. The decomposition of the lift system starts from the general view of a control system as described by the activity system in Figure 8.1.

## 8.7.1 Reference World (Observer)

RW-O consists of an activity rwo.system with two subactivities plant and environment. The activity rwo.plant is decomposed into subactivities $\mathsf{floor}_f$ and cabin. These are

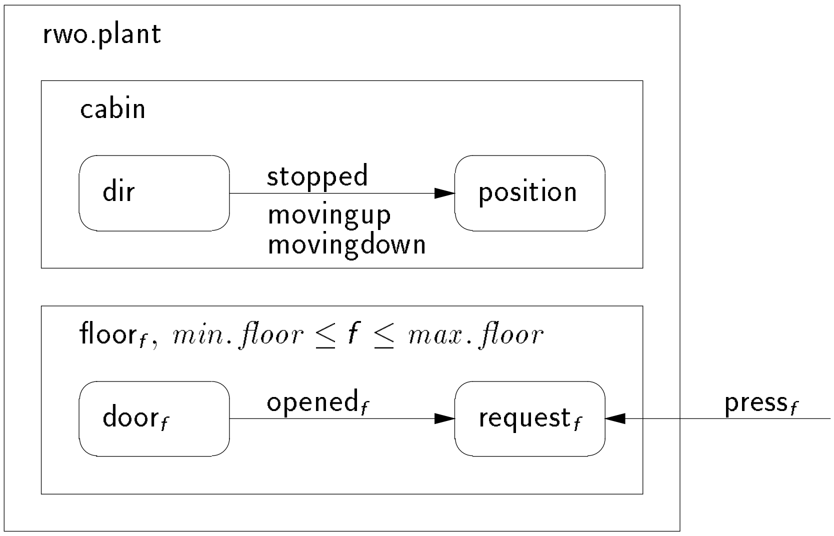



Figure 8.5: Rw-o: Decomposition of the activity plant

decomposed themselves into the activities door$_f$, request$_f$, dir, and position. The hierarchy of activities is depicted in Figure 8.5. The behaviour of the control-activities is described by the statecharts with corresponding names as shown in Section 7.1. The interfaces of the activities are described by the flow-lines.[12] Whether a state of a statechart is active is made visible by corresponding boolean variables which are true if and only if the state is active. This relation can be expressed by associating an invariant with the control-activity. In the case of the state dir.stopped, cf. Figure 7.1, Page 124, we use the variable stopped. The following invariant is associated with the control-activity rwo.plant.cabin.dir:

$$\mathsf{rwo.plant.cabin.dir} \rhd in(\mathsf{dir.stopped}) \;\Leftrightarrow\; \mathsf{stopped}$$

In addition, the control-activity rwo.plant.cabin.dir has the output variables movingup and movingdown. Similarly to the variable stopped these are true if and only if the corresponding states are active.

The temporal constraint in Equation 7.2 can be restated using the variable stopped instead of the predicate $in(\mathsf{dir.stopped})$ and can be associated as before with the states position.nearby and position.between of the statechart level.

$$\begin{array}{l} \mathsf{position.nearby} \rhd \Box\neg\mathsf{stopped} \\ \mathsf{position.between} \rhd \Box\neg\mathsf{stopped} \end{array} \tag{8.14}$$

The temporal constraints in Equations 7.1 through 7.7 can be restated similarly using the variables movingup and movingdown and they can be associated with

[12]The relation *flow* was restricted to relate input and output variables of subactivities and variables of their parent. Hence a flow-line between two subactivities $a$ and $b$ corresponds formally to a flow-line from $a$ to a variable of its parent and and a flow-line from this variable to $b$.

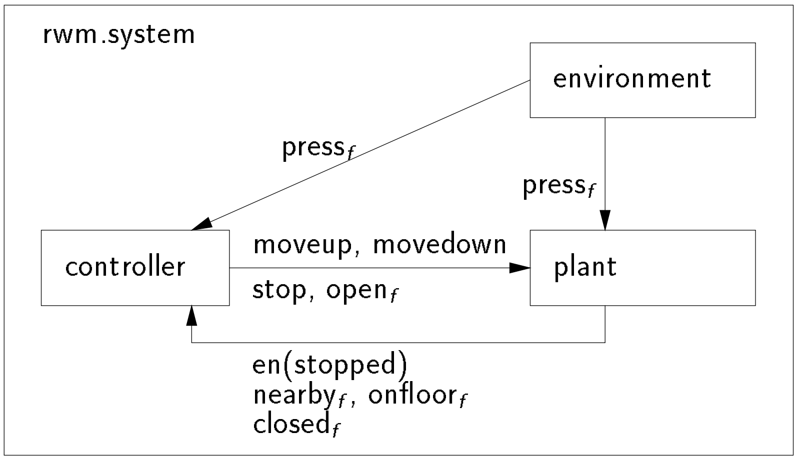



Figure 8.6: RW-M: Decomposition of the lift system

the statechart of the control-activity position and the transition from the state position.nearby and position.onfloor, respectively.

In a similar way the activities $\text{door}_f$ provide boolean variables $\text{opened}_f$ to the activities $\text{requested}_f$. The latter ones have in addition an input event $\text{press}_f$.

The condition expressions of the transition segments of the statechart of the control-activity $\text{rwo.request}_f$ can be restated using the boolean variable $\text{opened}_f$. That the flow-line labeled with $\text{press}_f$ starts outside the activity plant indicates that $\text{press}_f$ is an input event of plant and of all activities between plant and $\text{request}_f$ in the hierarchy of activities. These events are generated by the environment, cf. the activity-chart in Figure 8.6.

### 8.7.2 Reference World (Machine)

The interface between the activities plant and controller can be defined as shown in Figure 8.6. Hence RW-M consists of an activity rwm.system with three subactivities environment, plant, and controller. The subactivity controller is used here only as a target of flow-lines, it does not restrict the behaviour of rwm.system. Hence, the behaviour of rwm.system is described by plant and environment. It should be clear from the description of the events in Section 7.2 how the flow-lines of the activity-chart in Figure 8.6 have to be extended inside plant.

The flow-line from the activity environment to controller labeled with the event $\text{press}_f$ extends slightly the general structure of a control-system as depicted in Figures 1.1 and 8.1. Adhering strictly to this structure, a copy of this event would have to be created within the activity plant because input and output events of one activity have to be disjoint. Because this would add unnecessary and simple activities to the activity plant we prefer to make the events $\text{press}_f$ directly visible to controller.

In addition to the flow-lines between the activities plant and controller the variable onfloor has to be made visible to the activity dir because it is refered to in the TLA-formulae associated with its statechart, cf. Figure 7.5, Page 129. This flow-line is shown in the activity-chart in Figure 8.7.

### 8.7.3 Requirements

REQ consists of an activity req.system with subactivities plant and environment.

In Section 7.3 temporal constraints expressing requirements have been associated with states to restrict the behaviour as shown by RW-O. These constraints have been associated with those states, whose behaviour was restricted by the requirement. References to other states, not related in the hierarchy, have been easily possible because there was no notion of interface and all variables are visible globally. Using activities, the variables of the other activities must be made visible to the activity with which a temporal constraint is associated. In the same way, visibility must be ensured when variables are not refered to in temporal constraints, but used in event and action expressions of transition segments.

A flow-line between two orthogonal subactivities implies that this information becomes visible to all states between the source of the information flow and the least common ancestor of source and target and similarly to all activities between this least common ancestor and the target. To avoid that many interfaces have to be extended with these information flows one could associate TLA-formulae refering to variables of orthogonal activities with the least common ancestor. Then the relevant information has to flow only upwards in the hierarchy of activities. But in this case the flow of information does not coincide with causality. Information should flow from the activity in which the cause takes place to the activity in which the effect becomes visible. Therefore we associate TLA-formulae restricting behaviour with those activities or statecharts whose behaviour is actually restricted, although we have more information flows.

Therefore requirement 1 is expressed by the invariant $\Box(floor = \mathsf{f} \wedge \mathsf{onfloor} \wedge \mathsf{stopped})$ which is associated with the state $\mathsf{door}_f\mathsf{.opened}$ of the statechart of the control-activity $\mathsf{floor}_f\mathsf{.door}_f$, cf. Figures 7.6, page 130, and Figure 8.5. The variables floor, onfloor, and stopped have to be made visible to the activities $\mathsf{floor}_f\mathsf{.door}_f$.

The other requirements can be associated in a similar way with states or control-activities. The corresponding flow-lines are depicted in Figure 8.7 together with the flow-lines of RW-O and RW-M.

### 8.7.4 Combining Interfaces

Both RW-M and REQ are derived from RW-O, but further variables and events have been introduced and the interfaces of the activities have been extended. This means that variables which are local ones in RW-O became visible in the other descriptions.

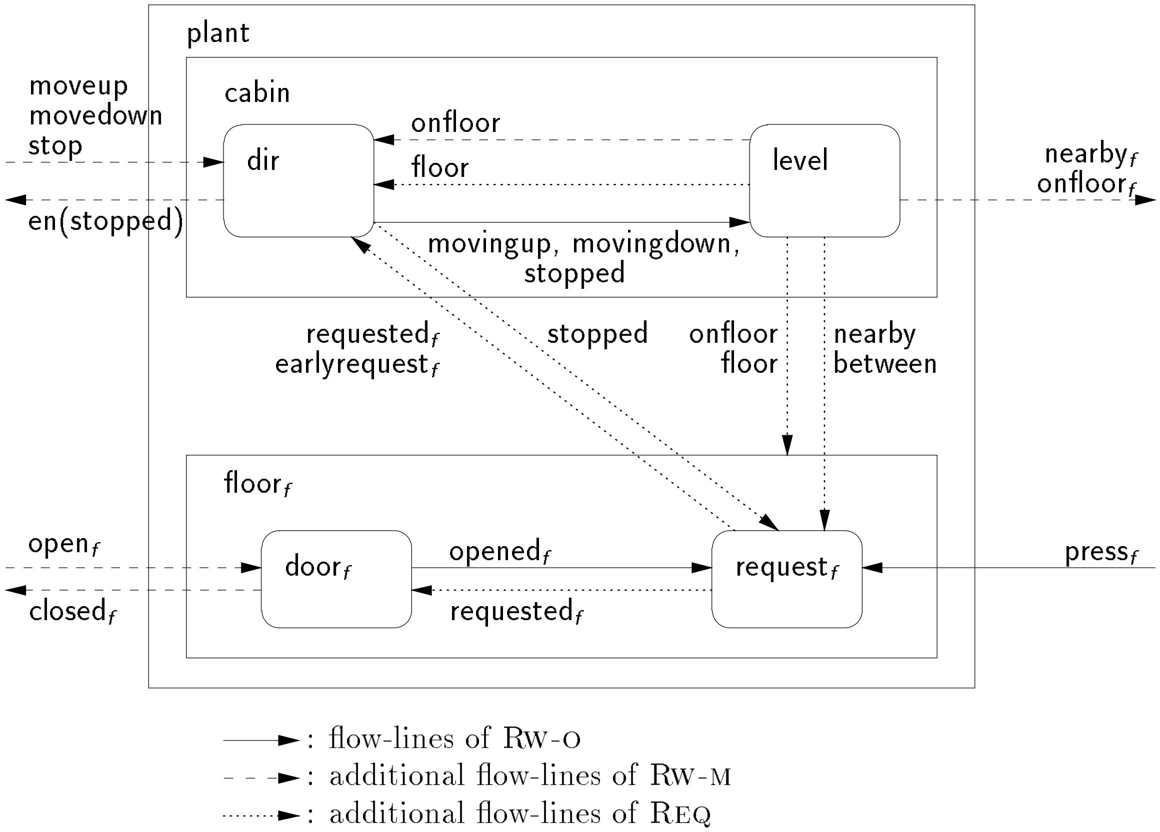



Figure 8.7: Flow-lines of Rw-o, Rw-m, and Req

Rw-m and Req should be refinements of Rw-o. Here we consider the problem that the different interfaces make the same distinctions between inputs and outputs. In our example it is easy to see that Rw-m and Req make the same distinction between input and output variables as Rw-o. In addition, they make the same distinction on those inputs and outputs which do not occur in Rw-o. The flow-lines of all three descriptions are depicted together in Figure 8.7. One can see from this activity-chart, that the activities dir and level are coupled quite closely and that few has been gained in terms of comprehensibility by describing them as separate activities instead of describing them as orthogonal substates of a state as in Section 7.1.

### 8.7.5 Specification and Design of a Controller

Spec and Des can be adapted to the use of activities in a similar way as the other descriptions. Spec is structured similar as Rw-m. It consists of an activity spec.system with subactivities plant, environment, and controller. Similar to Rw-m the activity controller is used here only as target of flow-lines. Des consists only of the activity des.controller.

### 8.7.6 Formal Relationships Between the Descriptions

The descriptions are related in different ways by refinement. In this section we present these relations. But we do not give a formal proof that the descriptions of the lift system actually satisfy these relations.

W.r.t. RW-O these are the following relations. Remember that rwo.system and req.system have two subactivities plant and environment. rwm.system has in addition a subactivity controller, but this does not influence the behaviour of rwm.system. Hence, the behaviour of rwm.system is determined by its subactivities plant and environment, too. Note that the inputs and outputs of the subactivities have to be made visible explicitly at the interface of the activities rwo.system, etc..

$$
\begin{aligned}
activity_{\mathsf{rwm.system}} &\Rightarrow \overline{activity_{\mathsf{rwo.system}}}^{8.11} \\
activity_{\mathsf{req.system}} &\Rightarrow \overline{activity_{\mathit{rwo.system}}}^{8.11}
\end{aligned}
\tag{8.15}
$$

Correctness of a controller means that the composition of DES and RW-M is a refinement of SPEC. The composition of DES and RW-M consists of the activity rwm.system with subactivities plant and environment, but the activity des.controller instead of rwm.system.controller. We denote this composed activity by des||rwm. Correctness of the controller can be stated as:

$$
activity_{\mathsf{des||rwm}} \Rightarrow \overline{activity_{\mathsf{spec.system}}}^{8.11}
\tag{8.16}
$$

It has been stated before, that REQ should be a refinement of RW-O, cf. Equation 8.15. This implies that each sequence of inputs which is possible in REQ has to be possible in RW-O, too. On the other hand, each sequence of inputs which is possible in RW-O should be possible in REQ, too. This means that REQ must not refuse any sequence of inputs, i.e. it is *receptive* [Dill, 1989].

This can be stated as the property that the activity rwm.system.environment is a refinement of REQ. All variables of REQ except the output variables of the activity req.system.environment have to be hidden.

$$
\begin{aligned}
& \exists v,\ \text{f.a. } v \in inputs(\mathsf{rwm.system.environment}) \\
& activity_{\mathsf{rwm.system.environment}} \\
\Rightarrow\ & \exists\ v,\ \text{f.a. } v \in inputs(\mathsf{req.system.environment}) \\
& \quad \exists\ v,\ \text{f.a. } v \in inputs(\mathsf{req.system.requirements}) \\
& \quad \exists\ v,\ \text{f.a. } v \in outputs(\mathsf{req.system.requirements}) \\
& \quad \overline{activity_{\mathsf{req.system}}}^{8.11}
\end{aligned}
\tag{8.17}
$$

## 8.8 Summary

Similar to statecharts we have defined the semantics of activity-charts by translating them to TLA-formulae. The TLA-formulae capture the semantics as presented in

[Harel and Naamad, 1996]. Again similar to statecharts, we have combined activity-charts and TLA: Temporal constraints can be associated with activities. We have defined a refinement relation between activities which respects their interfaces.

It is possible now to write descriptions using statecharts, activity-charts, and TLA. The activity-charts can be used to structure the description. That is, parts of descriptions, their interfaces, and the flow of information among them can be described.

The possibility to describe interfaces of activities will be used in the next chapter, too. The variables and events of a statechart are distinguished into input, output, and local ones according to the interface of the control-activity to which the statechart belongs. This distinction is useful to determine causal dependencies among variables, events, and states. These dependencies can be used to support the proof that a constrained statechart is consistent.

# Chapter 9

# Consistency

In the previous chapters we have defined a combination of statecharts and TLA and have used it in various descriptions of reactive systems. We have shown that the combination of statecharts and TLA allows to describe different properties in an adequate way. In this chapter we consider whether one or more descriptions have certain properties, that is we consider formal analysis of descriptions.

There are two classes of properties of descriptions: Properties which depend on the concrete system described and properties which are independent of the concrete system described. Examples of such dependent properties are

**Reachability:** A specific state can be reached

**Deadlock:** A specific state can be reached which is not left again

Examples of properties which are independent of the described system are

**Consistency:** A description has at least one run satisfying it

**Receptiveness:** [Dill, 1989] For each sequence of inputs there is a run satisfying the description

**Refinement:** Each run of a description satisfies another description

If the behaviour of a system is described by a constrained statechart and a property of the behaviour can be expressed by a TLA-formula, then this TLA-formula can be associated with the root-state of the statechart. To show that the property is valid can be reduced to show that one of the independent properties of the constrained statechart with the additional constraint is valid. It depends on the kind of the dependent property which independent one has to be shown.

Because it is possible to reduce the proof of some of the dependent properties to independent ones we concentrate here on the independent properties. Especially we consider here the question whether a description is consistent. We consider consistency because on the one hand it is an important property of descriptions which does not hold in general for descriptions written in the combination of TLA and statecharts. On the other hand we derive some hints on developing descriptions in an effective way by answering this question. These hints have been used already in the examples in Chapter 7 to determine with which state of a statechart a temporal constraint should be associated. In addition we investigate the refinement relation between two descriptions.

For the sake of simplicity we do not consider the full combination of statecharts and TLA as defined before when investigating properties of descriptions. We restrict ourselves to statecharts with temporal constraints. As temporal constraints we use initial conditions and next-state relations. This includes invariants as temporal constraints. We do not consider fairness- and leadsto-formulae as temporal constraints. We do not use the extension of statecharts with TLA-actions as enabling conditions, cf. Section 6.1. Activities are used only to define the interface of a statechart.

We start this chapter by considering two more technical questions. In Section 9.1 we investigate the question of how empty behaviours can arise. Essentially this is caused by requiring fairness on $Execute.step$-steps of the constrained statechart, whereas a constrained statechart is able to take only $Constrained.step$-steps. In Section 9.2 we investigate whether it is decidable that a constrained statechart is consistent. We show that this problem is undecidable if variables over infinite domains are used and that it is decidable if variables over finite domains are used.

In Section 9.3 we present conditions which guarantee that a statechart with temporal constraints is consistent. These conditions are derived from the property that whenever a step of the statechart is possible, then one satisfying the constraints is possible, too. Starting with the root of the statechart it has to be shown that each active statecharts-state can remain active or can be left. The root itself has to remain active.

Temporal constraints can be used to describe more precisely the set of reachable statuses as well as the set of steps of the constrained statechart. This is considered in Section 9.4. Among others, we show how temporal constraints can be generated from statecharts.

To prove that a constrained statechart is consistent is rendered more difficult because the structure of the statechart cannot be used to partition the proof. This is the case because each variable can be used in each temporal constraint, event- and action-expression. In Section 9.5 we present how a dependency relation between the entities of the statechart can be derived and used to support the proof of consistency. The aim is to derive an acyclic flow of information within the statechart.

In Section 9.6 we present some patterns of constrained statecharts together with conditions which guarantee that instances of these patterns are consistent.

Often descriptions contain liveness-formulae as temporal constraints. These are relevant w.r.t. consistency, too. In Section 9.7 we present conditions for two classes of liveness-formulae to show that statecharts with such constraints are consistent.

In Section 9.8 we show for two of the descriptions of the lift system presented in Chapter 7 that they are consistent. Here we use the heuristics to derive causal dependencies.

In Section 9.9 we consider refinement in addition to consistency. We show there how a statechart without temporal constraints can be extended or changed such that the resulting statechart is consistent and a refinement of the original one.

## 9.1 Machine Closedness

It is not clear immediately why the behaviour of a statechart with temporal constraints can be the empty set. Let $\sigma$ be a run of the statechart which does not satisfy the temporal constraints. Let us assume that a prefix $\sigma_p$ of $\sigma$ satisfies the constraints. From $\sigma_p$ we build a run $\sigma_s$ by repeating the last state of $\sigma_p$ infinitely often. It can be asked now why $\sigma_s$ is not contained in the behaviour of the constrained statechart either. The reason is that we require weak fairness of steps of the unconstrained statechart, but we do not require fairness on steps satisfying the constraints. This means that a statechart with temporal constraints does not need to be machine closed.[1]

In this section we derive a condition which guarantees that a statechart with temporal constraints is machine closed. We consider here only initial conditions and next-state relations as temporal constraints.

We start by characterising steps satisfying temporal constraints in more detail. Let $F$ be an initial condition or a next-state relation and $\mathsf{s} \in S$ be a state of a statechart. The semantics $[\![\, \mathsf{s} \rhd F \,]\!]_s$ of associating $F$ with $\mathsf{s}$ has been defined as a next-state relation, c.f. Section 6.4.1. This means, that $[\![\, \mathsf{s} \rhd F \,]\!]_s$ has been defined as $\Box[\mathcal{N}_{\mathsf{s},F}]_{step.taken}$ for TLA-actions $\mathcal{N}_{\mathsf{s},F}$. A *constrained step* of a statechart $S$ is a step which is restricted by these TLA-actions of all temporal constraints of the statechart.

$$Constrained.\,step \equiv \begin{array}[t]{l} \wedge\ Execute.\,step \\ \wedge \displaystyle\bigwedge_{\mathsf{s}\in S}\ \bigwedge_{F\in constraints(\mathsf{s})} \mathcal{N}_{\mathsf{s},F} \end{array}$$

It is easy to see that by taking constrained steps only the same set of runs is obtained as by the TLA-program *constrained.statechart*, cf. Section 6.5. The definition of *constrained.statechart* considers all runs of the unconstrained statechart, those runs not satisfying the constraints are discarded. This implies that the following formula

[1]If you have skipped Section 3.3.2 on first reading you should read it now.

is valid:

$$constrained.\,statechart \Leftrightarrow \begin{array}{l} \wedge\ initial.\,status \\ \wedge\ \Box \begin{bmatrix} \wedge\ Change.\,events \\ \wedge\ \vee\ Constrained.\,step \\ \quad\ \vee\ External.\,change \end{bmatrix}_{status} \\ \wedge\ WF_{status}(\,Execute.\,step\,) \end{array} \tag{9.1}$$

Because fairness is required on *Execute. step*-steps, but not on *Constrained. step*-steps, it cannot be expected that *constrained. statechart* is machine closed.

To show that the TLA-program *constrained. statechart* is machine closed it is sufficient to show that the TLA-action *Execute. step* is a subaction of the safety part of *constrained. statechart*, cf. Proposition 3.2. The safety part of *constrained. statechart* is

$$\begin{array}{l} \wedge\ initial.\,status \\ \wedge\ \Box \begin{bmatrix} \wedge\ Change.\,events \\ \wedge\ \vee\ Execute.\,step \\ \quad\ \vee\ External.\,change \end{bmatrix}_{status} \\ \wedge\ \bigwedge\limits_{s \in S}\ \bigwedge\limits_{F \in constraints(s)} \Box[\mathcal{N}_{s,F}]_{step.taken} \end{array}$$

By Equation 9.1 the preceding TLA-formula is equivalent to the following one. Hence, to conclude that *constrained. statechart* is machine closed it is sufficient to show that *Execute. step* is a subaction of

$$\begin{array}{l} \wedge\ initial.\,status \\ \wedge\ \Box \begin{bmatrix} \wedge\ Change.\,events \\ \wedge\ \vee\ Constrained.\,step \\ \quad\ \vee\ External.\,change \end{bmatrix}_{status} \end{array}$$

To show that *Execute. step* is a subaction of the preceding TLA-formula we can use Equation 3.4, page 45. That is, we have to show that the following TLA-formulae is valid.

$$\begin{array}{l} \wedge\ initial.\,status \\ \wedge\ \Box \begin{bmatrix} \wedge\ Change.\,events \\ \wedge\ \vee\ Constrained.\,step \\ \quad\ \vee\ External.\,change \end{bmatrix}_{status} \\ \Rightarrow \Box\ Enabled\ Execute.\,step \\ \qquad \Rightarrow Enabled \wedge\ Execute.\,step \\ \qquad\qquad\qquad\quad \wedge \begin{bmatrix} \wedge\ Change.\,events \\ \wedge\ \vee\ Constrained.\,step \\ \quad\ \vee\ External.\,change \end{bmatrix}_{status} \end{array} \tag{9.2}$$

The conjunct of *Execute. step* and the TLA-action describing a step of the constrained statechart in Equation 9.2 can be simplified. It is not possible that an

($\mathit{Execute.step} \wedge \mathit{External.change}$)-step is enabled: $\mathit{External.change}$ requires that $\mathit{step.taken}$ is not changed, whereas $\mathit{Execute.step}$ requires that $\mathit{step.taken}$ is changed. For the same reason it is not possible that an $\mathit{Execute.step}$ is enabled which is stuttering w.r.t. $\mathit{status}$. Furthermore, $\mathit{Execute.step}$ is implied by $\mathit{Constrained.step}$. At last, when taking a step of the statechart it is always possible to create the events as described by $\mathit{Change.events}$. Hence we get the following lemma.

**Lemma 9.1** *The TLA-action* $\mathit{Execute.step}$ *is a subaction of the safety part of* $\mathit{constrained.statechart}$ *for a constrained statechart whose temporal constraints are initial conditions and next-state relations, if the following TLA-formula is valid.*

$$\begin{array}{l} \wedge\ \mathit{initial.status} \\ \wedge\ \Box \begin{bmatrix} \wedge\ \mathit{Change.events} \\ \wedge\ \vee\ \mathit{Constrained.step} \\ \quad\ \vee\ \mathit{External.change} \end{bmatrix}_{\mathit{status}} \\ \Rightarrow \Box\ \mathit{Enabled}\ \mathit{Execute.step} \\ \qquad \Rightarrow \mathit{Enabled}\ \mathit{Constrained.step} \end{array}$$

If it can be shown that Lemma 9.1 holds for a constrained statechart with the corresponding restrictions, then it follows by Proposition 3.2 that $\mathit{constrained.statechart}$ is machine closed.

If $\mathit{constrained.statechart}$ is machine closed, then it is sufficient to consider the safety property described by its initial condition and its next-state relation to determine whether $\mathit{constrained.statechart}$ is consistent. We will see in Section 9.3 that the TLA-formula in Lemma 9.1 can be used again to show that a constrained statechart is consistent.

## 9.2 (Un-)decidability of Consistency

Each statechart without temporal constraints has a non-empty set of runs satisfying it. We show in this section that it is an undecidable problem whether a constrained statechart is consistent. This result is valid only if we use variables over an infinite domain. If we use variables over finite domains the problem is decidable.

**Theorem 9.1** *It is undecidable whether a constrained statechart is consistent.*

**Proof:** We reduce the problem whether a Turing machine accepts every word to consistency of constrained statecharts.

Let us consider a deterministic Turing-machine $T = (Q, q_0, q_a, \Sigma, \delta)$, that is $T$ has a (finite) set of states $Q$, a starting state $q_0 \in Q$, an accepting state $q_a \in Q$, a (finite) alphabet $\Sigma$, and a transition function $\delta\colon Q \times \Sigma \to Q \times \Sigma \times \{l, r\}$. $l$ and $r$ are used to denote whether the head of $T$ moves one position to the left or right in a step. To be more precise we assume that $\delta$ is not defined on $q_a$: $\delta\colon Q \setminus \{q_a\} \times \Sigma \to Q \times \Sigma \times \{l, r\}$.

Hence, $T$ actually stops when reaching the accepting state $q_a$. Otherwise $T$ takes an infinite number of steps. $T$ *accepts* a word $w \in \Sigma^*$ if there is a finite sequence of steps of $T$ which starts with the state $q_0$ and the word $w$ on its tape and reaches the accepting state $q_a$. By $\mathcal{L}(T) \subseteq \Sigma^*$ we denote the set of words accepted by $T$. It is an undecidable problem whether $\mathcal{L}(T) = \Sigma^*$ for a Turing machine $T$, i.e. whether a Turing machine accepts every word.

We construct from $T$ a statechart as follows: We take a set of states $\overline{Q} = \{\overline{q_0}, \dots\}$ corresponding to $Q$ as direct substates of the root of the statechart. The target state of the default transition is $\overline{q_0}$. The contents of the tape and the position of the head are represented by two variables $tape_l, tape_r \in \Sigma^*$ which contain the contents of the tape to the left and the right, respectively, of the position of the head. The transition segments of the statechart can be constructed easily from the definition of $\delta$: For each $q_1, q_2 \in Q$, $a_1, a_2 \in \Sigma$, $d \in \{l, r\}$ with $\delta(q_1, a_1) = (q_2, a_2, d)$ there is a transition segment from $\overline{q_1}$ to $\overline{q_2}$. The enabling condition of this transition segment expresses that $a_1$ is the letter at the position of the head, by convention this is the first letter of $tape_r$, or the blank letter if $tape_r$ represents the empty word. The action expressions of the transition segments change the variables $tape_l$ and $tape_r$ appropriately.

With the default transition of the statechart we associate a TLA-action which chooses non-deterministically a word $w \in \Sigma^*$ and assigns it to $tape_r$. The empty word is assigned to $tape_l$. We associate the invariant $\Box\neg\, in(\overline{q_a})$ with the root of this statechart to prevent that the state $\overline{q_a}$ becomes active.

Without regarding the temporal constraint the state $\overline{q_a}$ becomes active if and only if the word $w$ chosen initially is in $\mathcal{L}(T)$. As a consequence, the statechart has a run in which $\overline{q}_a$ does not become active if and only if $\mathcal{L}(T) \neq \Sigma^*$. This implies that the constrained statechart has a run if and only if $\mathcal{L}(T) \neq \Sigma^*$.

Hence it is undecidable whether a constrained statechart is consistent.

**qed**

The situation changes, when only variables over finite domains are used. In this case consistency of constrained statecharts is decidable. We assume in addition that the predicates and functions occurring in event and action expressions and in temporal constraints are decidable and computable, respectively. At first we consider only initial conditions and next-state relations as temporal constraints.

**Theorem 9.2** *It is decidable whether statecharts with initial conditions and next-state relations as temporal constraints and with data-variables over finite domains are consistent.*

**Proof:** Because the data-variables range over finite domains and the variables $S_{active}$, $E_{exists}$, and *step.taken* can take finitely many values only there are only finitely many different statuses. By the assumption stated before, the predicates and functions occurring in event and action expressions and in temporal constraints are decidable and computable, respectively. Also, the predicates and functions used to define the semantics of a statechart are decidable and computable, respectively. Hence the initial status and the next-state relation of the statechart are described by decidable predicates.

Therefore we can construct the finite graph of statuses which are reachable from an initial status. A node in this graph corresponds to a status of the statechart and an edge corresponds to a step of the statechart, an external, or a stuttering step.

From this graph we remove those nodes and edges which correspond to a violation of the safety-constraints. If the resulting graph contains a maximal strongly connected component[2] which can be reached from an initial status, then the graph contains an infinite path. This path corresponds to a run of the constrained statechart. Because it is a decidable problem, whether a finite graph contains a maximal strongly connected component, it is also decidable, whether the constrained statechart is consistent.

**qed**

Theorem 9.2 can be extended to fairness constraints on subactions of the statechart. Informally, this can be shown as follows. As in the proof of theorem 9.2 a finite graph can be generated for the statechart. It can be checked whether there is a strongly connected component, in which each TLA-action, for which weak fairness is required by a temporal constraint, is either disabled at some status or can be taken without leaving the component, cf. [Manna and Pnueli, 1995]. Similarly, it can be checked for each TLA-action, for which strong fairness is required, whether it is disabled at each status or whether it can be taken without leaving the component. This check can be restricted to the cases that the statecharts-states with which the temporal constraints are associated remain active continuously or for an infinite number of steps.

## 9.3 Existence of Models

In the Sections 9.1 and 9.2 we considered two more technical questions: Under which conditions is a constrained statechart machine closed and is it decidable whether a constrained statechart is consistent? In this section we begin to investigate methods to show that a constrained statechart is consistent.

We start by presenting an approach to show that the TLA-formula of Lemma 9.1 is valid. In Section 9.6 we present patterns of constrained statecharts and conditions which guarantee that instances of these patterns are consistent.

To show that a constrained statechart is consistent it is sufficient to show that a constrained step is enabled whenever an unconstrained one is enabled, too. This means,

[2] A set $S$ of nodes of a graph $G$ is a *strongly connected component* if each node in $S$ can be reached from each other one using edges of $G$ and where each intermediate node is in $S$ itself, cf. [Manna and Pnueli, 1995]. It is a *maximal* one if for each node $s \in G \setminus S$, the set $S \cup \{s\}$ is not a strongly connected component.

that a constrained step is enabled at each status at which an unconstrained step is enabled and which can be reached without violating temporal constraints. This is the condition which has been used already to show that a constrained statechart is machine closed, cf. Lemma 9.1.

**Theorem 9.3** *Assume that the TLA-formula of Lemma 9.1 holds for a constrained statechart, then the statechart is consistent.*

**Proof:** Let $\sigma = \langle s_0, s_1, \ldots, s_n \rangle$, for $n \geq 0$, be a sequence of TLA-states such that the run $\langle s_0, s_1, \ldots, s_n, s_n, \ldots \rangle$ satisfies the safety part of the constrained statechart. This means that each *Execute. step*-step in $\sigma$ is a *Constrained. step*. We show that $\sigma$ can be extended to a run of the constrained statechart.

Because the predicate *initial. status* is satisfiable there is a TLA-state $s_0$ such that $\langle s_0, s_0, \ldots \rangle$ satisfies the safety part of the constrained statechart.

Now let $\sigma = \langle s_0, s_1, \ldots, s_n \rangle$ be a sequence of TLA-states as described before. Assume that the TLA-action *Execute. step* is not enabled at $s_n$. Then we extend $\sigma$ to $\sigma \circ \langle s_n \rangle = \langle s_0, s_1, \ldots, s_n, s_n \rangle$. This implies that *Execute. step* remains disabled. In the case that *Execute. step* is enabled at $s_n$ we conclude from the TLA-formula in Lemma 9.1 that *constrained. step* is enabled at $s_n$, too. This conclusion is valid because $\langle s_0, s_1, \ldots, s_n, s_n, \ldots \rangle$ satisfies

$$\begin{array}{l} \wedge\ \mathit{initial.\,status} \\ \wedge\ \Box \begin{bmatrix} \wedge\ \mathit{Change.\,events} \\ \wedge\ \vee\ \mathit{Constrained.\,step} \\ \quad\ \vee\ \mathit{External.\,change} \end{bmatrix}_{\mathit{status}} \end{array}$$

The preceding formula is the premise of the TLA-formula in Lemma 9.1. We take this step and obtain a sequence $\langle s_0, s_1, \ldots, s_n, s_{n+1} \rangle$ where each *Execute. step*-step is actually a constrained step and where each *Execute. step*-step is taken as soon as possible.

Let $\tau$ be a run satisfying the safety part of a constrained statechart and which has been constructed as defined before, then $\tau$ satisfies fairness on *Execute. step*-steps because either it contains infinitely many *Execute. step*-steps or a TLA-state has been reached in which no *Execute. step*-step is enabled. Because there is no fairness requirement on other TLA-actions, this TLA-state can be repeated forever.

**qed**

Next we look more closely at *Enabled Constrained. step* in the TLA-formula of Lemma 9.1. A constrained step is enabled if an *Execute. step* is enabled and each active state of the statechart can be left or can become active without violating the safety temporal constraints. The root-state of the statechart has to remain active. Whether a state can remain active or can be left has to correspond to the semantics of the statechart. As a consequence, when leaving a state it has to be possible that the target states of the corresponding compound transitions can become active.

**Keeping a State Active** An active statecharts-state $s$ can remain active in a step of the statechart

- if the temporal constraints associated with $s$ are satisfied
- if each active descendant of $s$ can remain active or can be left itself
- if $s$ is not within the scope of an executable compound transition which has to be taken, and without being itself a target of the compound transition
- if there is no executable compound transition leaving $s$, or if $s$ has an executable static transition

These conditions are described by a TLA-action *Can. stay*. Its formal definition can be given in a more simple way because these TLA-actions will be applied top-down in the hierarchy of states. The first case above is handled already when considering the direct substates of the scope of the transition. As in the definition of the TLA-action *Constrained. step* the TLA-action $\mathcal{N}_{s,F}$ is the TLA-action used to define the semantics of the temporal constraint.

$$
\begin{array}{l}
Can.\,stay\colon S \\
Can.\,stay(s) \equiv \land \displaystyle\bigwedge_{F \in constraints(s)} \mathcal{N}_{s,F} \\
\qquad\qquad\qquad \land \displaystyle\bigwedge_{s_1 \in S} \begin{array}{l} \land\ substate(s_1, s) \\ \land\ in(s_1) \\ \Rightarrow \lor\ Can.\,stay(s_1) \\ \quad\ \lor\ Can.\,leave(s_1) \end{array} \\
\qquad\qquad\qquad \land \lor \displaystyle\bigwedge_{T_1 \in FCT} \begin{array}{l} s \in source(T_1) \\ \Rightarrow \neg\, executable.\,transition(T_1, S_{active}, E_{exists}, w) \end{array} \\
\qquad\qquad\qquad\quad \lor \displaystyle\bigvee_{T_1 \in SCT} \begin{array}{l} \land\ s = source(T_1) \\ \land\ executable.\,transition(T_1, S_{active}, E_{exists}, w) \end{array}
\end{array}
$$

**Leaving a State** An active state can be left

- if it is below the scope of an executable compound transition
- if the initial conditions of the states which become active when taking the transition can be satisfied

Again, the formal definition can be kept more simple. Instead of the first condition it is sufficient to consider only those states, which are between — in the hierarchy of states — a source state (inclusively) and the scope (exclusively) of an executable compound transition. Transitions with higher source states do not need to be considered, because for their source states it is checked separately whether they can be left. If these are left actually, then according to the semantics of statecharts their descendants are left, too. Analogously, those states below the scope of a transition which are not above a source state are left, too. These states have to be descendants of an and-state. For an and-state the first condition is checked separately, too.

$$
\begin{array}{l}
Can.\,leave\colon S \\
Can.\,leave(\boldsymbol{s}) \\
\equiv \bigvee_{T_1 \in FCT} \wedge \bigvee_{s_1 \in S} \begin{array}{l} \wedge\ \boldsymbol{s_1} \in source(T_1) \\ \wedge\ \boldsymbol{s} \in ancestors^*(\boldsymbol{s_1}) \cap descendants^+(scope(T_1)) \\ \wedge\ executable.\,transition(T_1, S_{active}, E_{exists}, w) \end{array} \\
\qquad\qquad \wedge \bigwedge_{s_2 \in S} \begin{array}{l} s_2 \in states.\,entered(T_1) \\ \Rightarrow \bigwedge_{F \in constraints(s_2)} \mathcal{N}_{s_2,F} \end{array}
\end{array}
$$

In the second conjunct it is required for each temporal constraint $F$ that the TLA-action $\mathcal{N}_{s_2,F}$ is satisfied. This does not cause a problem because it is distinguished within these actions whether $\boldsymbol{s_2}$ becomes or remains active, cf. Section 6.4.1.

**Showing Consistency** The TLA-action *Can. stay* and *Can. leave* can be used to show that a constrained statechart is consistent. A constrained step is enabled if an unconstrained one is enabled in which the root-state can remain active. Hence we can use the following Lemma instead of Lemma 9.1.

**Lemma 9.2** *A constrained statechart with initial conditions and next-state relations as temporal constraints is consistent if the following TLA-formula holds*

$$
\begin{array}{l}
\wedge\ initial.\,status \\
\wedge\ \Box \begin{bmatrix} \wedge\ Change.\,events \\ \wedge\ \vee\ Constrained.\,step \\ \quad\ \vee\ External.\,change \end{bmatrix}_{status} \\
\Rightarrow \Box\ Enabled\ Execute.\,step \\
\qquad \Rightarrow Enabled \wedge\ Execute.\,step \\
\qquad\qquad\qquad\quad \wedge\ Can.\,stay(root)
\end{array}
$$

The invariant in the conclusion of the TLA-formula of Lemma 9.2 has to be shown for the set of reachable statuses of the constrained statechart. Sometimes this set is difficult to determine itself. It can be easier to show that this conclusion is valid for a larger set of statuses. In Section 9.4 we show how invariants associated with statecharts-states can be used to describe the set of reachable statuses. In addition we show there how temporal constraints can be generated which can be used to determine which constrained step can be taken.

When proving that the TLA-formula of Lemma 9.2 is valid we would like to proceed by using the hierarchical structure and the type of the statecharts-states. For example we would like to check for each descendant of an and-state independent of the other ones whether it can remain active. But this is not possible in general because each temporal constraint and each event or action expression can refer to any variable. Hence it is possible that each of these states can remain active, thereby assuming different values of the variables in the new status, and that these new values cannot be combined to one status.

To prevent this situation we want to make clear which variable or event depends on which other one. If this dependency relation is acyclic it can be used to structure the proof of the TLA-formula of Lemma 9.2 such that the new values of data-variables can be combined to one status. We investigate in Section 9.5 how to derive such a dependency relation.

## 9.4 Using Temporal Constraints as Auxiliary Lemmata

In this section we show which temporal constraints of a constrained statechart can be used to support the proof that the TLA-formula of Lemma 9.2 holds. In Section 9.4.1 we show how invariants associated with statecharts-states can be used to characterise the set of reachable statuses. In Section 9.4.2 we generate temporal constraints from statecharts to restrict the possible steps of a constrained statechart.

### 9.4.1 Relaxing Reachability

The invariant in the conclusion of the TLA-formula of Lemma 9.2 has to be shown for the set of reachable statuses of the constrained statechart. Sometimes the set of reachable statuses is difficult to determine itself. Alternatively one can show

$$\begin{aligned} &\mathit{Enabled}\ \mathit{Execute.\,step} \\ &\Rightarrow \mathit{Enabled} \wedge\ \mathit{Execute.\,step} \\ &\qquad\qquad\quad \wedge\ \mathit{Can.\,stay}(\mathit{root}) \end{aligned} \tag{9.3}$$

for a larger set of statuses. Let $p$ be a predicate on statuses which holds at least for all reachable statuses of the constrained statechart. This means that $\Box p$ is an invariant of *constrained. statechart*. To show that *constrained. statechart* is consistent it is sufficient to show Equation 9.3 for the statuses satisfying $p$.

In the following we present some invariants $\Box p$ which can be derived easily both from the structure of a statechart and from its temporal constraints. We consider here only invariants and initial conditions as temporal constraints, because these refer to single statuses.

The simplest invariant is that a status represents a correct configuration. This property is clearly valid of all reachable statuses.

To determine which temporal constraints can be used it has to be distinguished whether *External. change*-steps can occur or not. If a statechart is considered as a part of an activity, then its environment is also described by an activity. Both activities take their steps together, therefore no *External. change*-steps occur. This implies, that all properties valid at the end of one step are still valid at the beginning of the next one, because only *status*-stuttering steps can be taken between these steps. Hence all temporal constraints can be taken into account. If a statechart is considered without its environment, then *External. change*-steps are possible between consecutive *Execute. step*-steps. By *External. change*-steps invariants refering to input entities can be invalidated. Hence only invariants refering to non-input entities can be taken into account in this case.

Up to now we have determined which temporal constraints can be taken into account at all. Next we point out which temporal constraints of which statecharts-states shall be considered.

At first, only the temporal constraints of the states which are active in each status are considered. These states are the root-state, and if this is an and-state then also its direct descendants. Usually this set of states is a small set of states and therefore only few temporal constraints can be used to describe the set of reachable statuses.

When proving $Can.\,stay(\mathsf{s})$ for a statecharts-state $\mathsf{s}$, there is usually a case distinction on the set of active states below $\mathsf{s}$. In each of the cases there are assumptions which states are active. The temporal constraints of these states, of their ancestors, and of the direct descendants of active and-states above $\mathsf{s}$ can be used. This set of temporal constraints contains the temporal constraints of the states active in all statuses and this set is still easy to derive.

Further invariants can be deduced from these temporal constraints, especially if states of the statechart are related by temporal constraints.

The invariants derived are useful not only to relax the premise in the TLA-formula of Lemma 9.2, but also when proving this equation. In this case they can be used as auxiliary properties.

### 9.4.2 Generating Temporal Constraints from Statecharts

In the previous section we considered only invariants which have been given explicitly as temporal constraints. It is also useful to generate further temporal constraints from the statecharts-parts of a description. Especially we want to use the event expressions of compound transitions which start or end at a state to generate temporal constraints for this state.

Our approach to generate temporal constraints follows the approach presented in [Jeffords and Heitmeyer, 1998]. It is described there how invariants can be generated

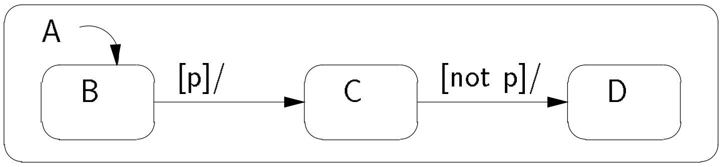



Figure 9.1: Generating temporal constraints from statecharts

from SCR-descriptions. It is not possible to adapt this approach directly because in SCR another semantics of steps is used than our semantics of statecharts. The variant of SCR used in [Jeffords and Heitmeyer, 1998] is a synchronous language. That is, changes of values of conditions can lead to changes of modes — which are the equivalent notion in SCR to states in statecharts — in the same step. On the other hand, in the single-step semantics of statecharts used here changes of values of conditions of event expressions become effective in the next step of the statechart. Therefore the relation between the values of conditions of event expressions and whether a state is active cannot be described by an invariant. Instead of invariants it is necessary to use next-state relations. As an example see the statechart in Figure 9.1. In this statechart the state $\mathsf{C}$ has to be left if the condition $\mathsf{p}$ is no longer valid. But the state $\mathsf{C}$ is left in the step following the one in which $\mathsf{p}$ becomes false. Hence there is a status in which $\mathsf{C}$ is active but $\mathsf{p}$ is not valid. This implies that the invariant $\Box in(\mathsf{C}) \;\Rightarrow\; \mathsf{p}$ is not valid. This separation in time between cause and effect can be taken into account by a next-state relation. In the statechart in Figure 9.1 the temporal constraint $\Box[in'(\mathsf{C}) \;\Rightarrow\; \mathsf{p}]_{\langle\rangle}$ is valid and can be associated with the state $\mathsf{A}$.

$$\mathsf{A} \rhd \Box[in'(\mathsf{C}) \;\Rightarrow\; \mathsf{p}]_{\langle\rangle} \tag{9.4}$$

This means that the state $\mathsf{C}$ can remain active only if $\mathsf{p}$ is valid. We call this temporal constraint a *delayed invariant* of the state $\mathsf{C}$.

Because the temporal constraint in Equation 9.4 is not an invariant it cannot be used to restrict the set of reachable statuses. Nevertheless it is useful to generate delayed invariants because they can be used to describe more precisely the possible steps of a constrained statechart.

To generate delayed invariants of states we consider or-states and their direct descendants. Because we do not consider and-states each compound transition has exactly one source and one target state. We exclude full compound transitions with identical source and target states, but we consider static transitions. For the sake of simplicity we assume that all full and static compound transitions have the same explicit priority. Because the compound transitions have the same structural priority or are not comparable w.r.t. structural priority a compound transition is executable if it is enabled.

A delayed invariant of a state $s$ is intended to describe that the conditions which imply that the state has to be left are not valid. These conditions are those event expressions of compound transitions with source $s$ which consist of a single atomic event or condition expression. To determine a delayed invariant it is considered at first which boolean expressions over these conditions are valid when $s$ becomes active. At second it is considered which boolean expressions over these conditions are valid when the state can remain active when static transitions can be taken although one of the compound transition leaving $s$ is enabled, too.

The functions

$$\begin{array}{ll} enter\colon & S \rightarrow 2^{FCT} \\ exit\colon & S \rightarrow 2^{FCT} \\ stay\colon & S \rightarrow 2^{SCT} \end{array}$$

are used to denote for a state $s$ the set of full compound transitions with target state $s$, the set of full compound transitions with source state $s$, and the set of static compound transitions of $s$, respectively.

By $\mathcal{L}$ we denote the set of atomic event and condition expressions occurring in a statechart and their negations. The elements of $\mathcal{L}$ are called *literals*. The literals which imply that a state $s$ is left if they become valid are called *relevant literals*. For a state $s$ the set of relevant literals is denoted by $rel(s)$. $rel(s)$ is defined as

$$rel\colon S \rightarrow 2^{\mathcal{L}}$$
$$rel(s) \equiv \{l \mid \bigvee_{T_1 \in exit(s)} \{l\} = \bigcup_{t \in T_1} trigger(t)\}$$

That is, a literal $l$ is relevant for a state $s$ if it is the single event expression of a full compound transition with source state $s$.

To restrict the conditions which are valid when a state becomes or remains active to the relevant literals we assume that the event expressions of the corresponding full and static compound transitions are given in disjunctive normal form. That is, the event expressions are written as disjunctions of conjunctions of literals. It is possible to rewrite the event expressions this way, cf. Section 5.1.1. The conditions are restricted to the relevant literals by replacing each non-relevant literal by *true* in these expressions. We denote by $d|_L$ the restriction of a boolean expression to a set of literals $L$. Let $d \equiv \bigvee_{1 \leq i \leq n} c_i$ and $c_i \equiv \bigwedge_{1 \leq j \leq n_i} l_{i,j}$, where the $l_{i,j}$ are literals, and let $L$ be a set of literals. Then $d|_L$ is defined as

$$d|_L \equiv \bigvee_{1 \leq i \leq n} c_i|_L$$

$$c_i|_L \equiv \bigwedge_{1 \leq j \leq n_i} l_{i,j}|_L$$

$$l_{i,j}|_L \equiv \begin{cases} l_{i,j} & \text{if } l_{i,j} \in L \\ true & \text{else} \end{cases}$$

If a state $s$ is active after taking a step of a statechart then it was either possible to enter $s$ or it was not necessary to leave $s$. In the first case one of the compound transitions with target $s$ can be taken. In the second case one of the static compound transitions of $s$ can be taken if one of the full compound transitions with source $s$ can be taken. Restricted to the relevant literals this can be expressed as

$$\begin{array}{l} \vee \bigvee_{T_1 \in enter(s)} trigger(T_1)|_{rel(s)} \\ \vee \bigvee_{l \in rel(s)} l \\ \quad \Rightarrow \bigvee_{T_2 \in stay(s)} trigger(T_2)|_{rel(s)} \end{array}$$

This condition can be rewritten itself to disjunctive normal form and associated with the parent of $s$. We get

$$parent(s) \rhd \Box \left[ in'(s) \;\Rightarrow\; \begin{array}{l} \vee \bigvee_{T_1 \in enter(s)} trigger(T_1)|_{rel(s)} \\ \vee \bigwedge_{l \in rel(s)} \neg l \\ \vee \bigvee_{T_2 \in stay(s)} trigger(T_2)|_{rel(s)} \end{array} \right]_{\langle\rangle} \tag{9.5}$$

In the statechart in Figure 9.1 the literal $\mathsf{not\ p}$ is a relevant literal of the state $\mathsf{C}$. There is only one compound transition with target $\mathsf{C}$ and no static transition of $\mathsf{C}$. By instantiating Equation 9.5 for $\mathsf{C}$ we get

$$A \rhd \Box \left[ in'(\mathsf{C}) \;\Rightarrow\; \begin{array}{l} \vee\ \mathsf{p} \\ \vee\ \neg(\mathsf{not\ p}) \end{array} \right]_{\langle\rangle}$$

which is equivalent to the temporal constraint in Equation 9.4.

The temporal constraint in Equation 9.5 can be strengthened by invariants to take dependencies between validity of the literals into account. As an example we consider the statechart in Figure 7.3, which describes when there is a request on a floor in the lift system. The event expression of the compound transition leading to the state $\mathsf{request}_f\mathsf{.requested}$ is $\mathsf{press}_f[\, in(\mathsf{door}_f\mathsf{.closed})]$, that of the compound transition leaving $\mathsf{request}_f\mathsf{.requested}$ is $[\, in(\mathsf{door}_f\mathsf{.opened})]$. $in(\mathsf{door}_f\mathsf{.closed})$ is the sole relevant literal. The invariant $\Box(\neg\, in(\mathsf{door}_f\mathsf{.opened}) \;\Leftrightarrow\; in(\mathsf{door}_f\mathsf{.closed}))$ is valid here. Hence we get the delayed invariant

$$\mathsf{request}_f \rhd \Box[in'(\mathsf{requested}) \;\Rightarrow\; \neg\, in(\mathsf{door}_f\mathsf{.opened})]_{\langle\rangle}$$

That is, there is a request only if the door on the corresponding floor was not open. The literal $\mathsf{press}_f$ has been replaced by *true* because it is not relevant.

We have shown how the approach of [Jeffords and Heitmeyer, 1998] can be adapted to generate delayed invariants. But there is one difference. The generation of invariants is iterated in [Jeffords and Heitmeyer, 1998]. In each step the invariants of the source states of transitions leading to a state $s$ which are generated in the preceding steps are used to generate an invariant of $s$ itself. Such an iteration cannot be done here because we do not generate invariants but delayed invariants. The delayed invariant of the source state of a compound transition leading to a state $s$ expresses a property of the step preceding the one in which $s$ becomes active. Hence, iterating the generation of delayed invariants implies that not single steps are considered, but increasing sequences of steps.

The delayed invariants generated this way can be used together with the ones associated as temporal constraints with other parts of a constrained statechart as auxiliary lemmata when showing that steps of a constrained statechart can be taken.

## 9.5 Causal Dependency

In this section we investigate a relationship between the entities of a statechart which expresses that one entity depends on another one. That one entity $a$ depends on another entity $b$ means that changing $b$ in a step implies that $a$ has to be changed in the *same* step. It can be checked whether this relationship is acyclic and in this case it can be used to structure the proof that the TLA-formula of Lemma 9.2 is valid.

As in the previous sections we consider constrained statecharts, but not constrained activity-charts. We assume that a statechart is embedded in a control-activity, hence we can distinguish between input, output, and local entities of the statechart. We restrict ourselves to invariants as temporal constraints. It should become clear how the following considerations can be applied to initial conditions, too.

In a statechart without temporal constraints changes of the status in one step become effective in the next step. They do not become effective in the step in which the change occurs. On the other hand, if an invariant associated with a statecharts-state becomes invalid in an *Execute. step*-step, this state cannot remain active in this step, it has to be left. We are concerned with causal dependency between entities used or changed in such instantaneous reactions.

To make the problem more clear we start with an example. Consider the statechart in Figure 9.2. We assume that i and y are input variables and $o_j$ is an output variable. This statechart is intended to express that the current mode of a system, represented by the currently active descendant of the statecharts-state mode, is defined by validity of the predicates $p_A$, $p_B$, and $p_C$. On the other hand the output variable $o_j$ is assigned a value depending on the current mode and the current value of y. Thereby it is not distinguished whether the current mode is A or B.

The dashed lines in the boxes of the states mode.A, mode.B, mode.C, $\text{output}_j$.AB,

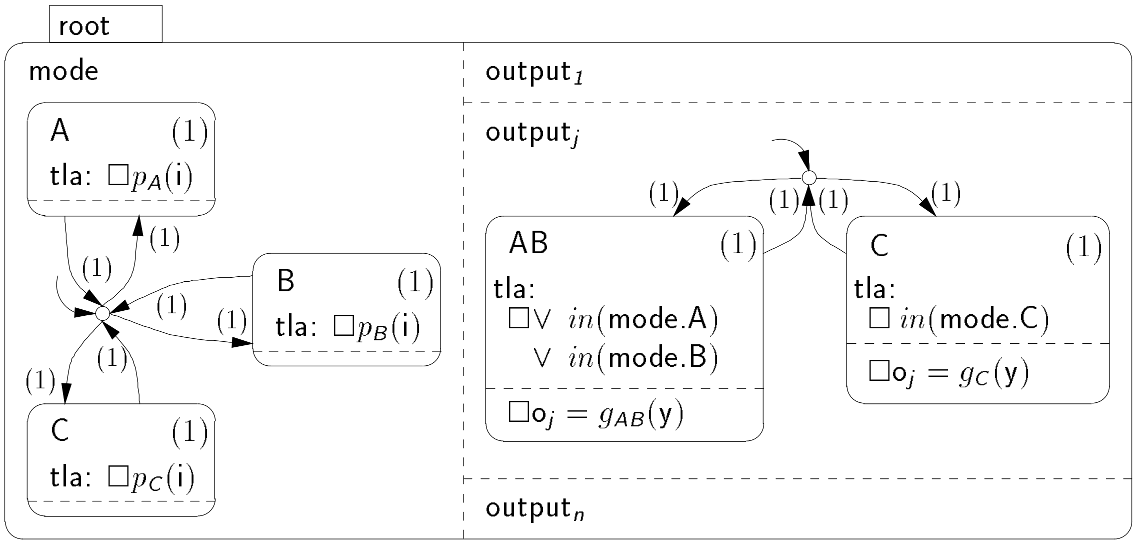



Figure 9.2: Causal dependencies: Example

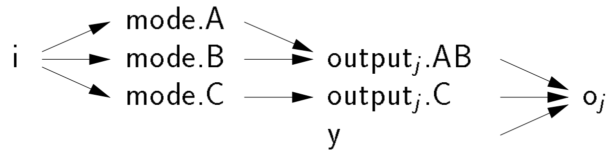



Figure 9.3: Causal dependencies of the statechart in Figure 9.2

and output$_j$.C are introduced in Section 9.5.2 to express causal dependencies. They do not influence the meaning of the statechart.

We assume that the predicates $p_A$, $p_B$, and $p_C$ form a complete case disjunction of i. It is not important here whether this is a disjoint case disjunction.

Let us consider an *Execute. step-* step starting in a status in which the state mode.A is active and in which i is changed such that $p_C(\mathsf{i})$ becomes valid, but $p_B(\mathsf{i})$ does not become valid. This implies that the following TLA-actions are satisfied by this step: $p_A(\mathsf{i}) \wedge p'_C(\mathsf{i}) \wedge \neg p'_B(\mathsf{i})$ and as a consequence $in(\mathsf{mode.A}) \wedge in'(\mathsf{mode.C})$. This implies in turn $in(\mathsf{output}_j.\mathsf{AB}) \wedge in'(\mathsf{output}_j.\mathsf{C})$ and finally $\mathsf{o}_j = g_{AB}(\mathsf{y}) \wedge \mathsf{o}_j{}' = g'_C(\mathsf{y})$. Note that this is not a sequence of statecharts-steps, all these TLA-actions are satisfied by one TLA-step. The sequential argumentation which TLA-actions are satisfied by this TLA-step is closely related to the synchronous execution of microsteps according to [Pnueli and Shalev, 1991], but the different parts of the statechart are not synchronised here by (internal) events, but by the invariants.

The statechart and the temporal constraints have already been written such that the causal dependencies become clear. These relationships are depicted informally in Figure 9.3, where $u \rightarrow v$ denotes that $v$ depends on $u$.

If this relationship contains a cycle then problems can occur. For example, assume that instead of the predicates $p_A$, $p_B$, and $p_C$ on the variable $\mathsf{i}$ we have predicates $q_A$, $q_B$, and $q_C$ on the variables $\mathsf{i}$ and $\mathsf{o}_j$, respectively. Then it depends also on $\mathsf{o}_j$ which substate of mode is active. We consider again a step starting in a status in which the statecharts-state mode.A is active and that $\mathsf{i}$ is changed such that $q_A(\mathsf{i}, \mathsf{o}_j)$ becomes invalid. Assume that there does not exist a value of $\mathsf{o}_j$ such that neither $q_B(\mathsf{i}, \mathsf{o}_j)$ and $\mathsf{o}_j = g_{AB}(\mathsf{y})$ nor $q_C(\mathsf{i}, \mathsf{o}_j)$ and $\mathsf{o}_j = g_C(\mathsf{y})$ are valid, then there is no next status which can be reached. In this step even the statecharts-state root cannot remain active. Note that it is not guaranteed that a next status exists if the causal dependencies are an acyclic relation, but it is easier to show if this is actually the case.

In the remainder of this section we consider the following questions:

1. Which are the entities among which we intend to establish causal dependency? Note that in Figure 9.3 we used single states and variables in the same way as entities, whereas we used the set of active states in the same way as variables when describing the semantics of statecharts.

2. How can the relationship be constructed for a given statechart? For example, how can the information in Figure 9.3 be derived from the statechart in Figure 9.2?

   In this context we present how temporal constraints can be associated with states such that the relationship is acyclic.

3. How can the information about causal relationships be used to structure proofs of consistency?

The first of these questions is answered in Section 9.5.1, the second one in Section 9.5.2, and the third one in Section 9.5.3.

### 9.5.1 Causally Related Entities

To establish a relationship it has to be made clear at first among which entities this relationship shall be established. Given a statechart we determine in this section which constituents of the statechart can be related causally. We distinguish whether an entity is refered to in a temporal constraint or in an event or action expression.

It must be possible to influence the entities which are in this relationship by taking a step. Hence we establish the relationship among those entities which can be part of a status. These are the states, events, and variables of the statechart. We do not consider the connectors, transition segments, and static reactions as entities.

$$En \equiv S \cup E \cup W$$

The relationship shall express that the value of a variable, whether a state is active, and whether an event exists depends on other entities.

The statechart is embedded in a control-activity. The interface of this control-activity allows to distinguish input, output, and local events and variables. We consider the states of the statechart as local entities.

We take the individual states and events as entities instead of the variables $S_{active}$ and $E_{exists}$ because the individual states and events are refered to in temporal constraints, event and action expressions. The variables $S_{active}$ and $E_{exists}$ do not occur in temporal constraints, event and action expressions. Furthermore, $S_{active}$ and $E_{exists}$ as a whole are too large entities. Using them does not allow for example to express that one state depends on another one.

On the other hand we do not consider expressions over $En$ as entities themselves. For example, in the statechart in Figure 9.2 the predicate $p_{\mathsf{A}}(\mathsf{i})$ is neither related to the statecharts-state $\mathsf{mode.A}$ nor to the variable $\mathsf{i}$. If it should be necessary for some reason to express that the value of an expression depends on other entities this can be expressed by using a corresponding variable. The values of the variable and the expression can be coupled by an invariant. For example, in the statechart in Figure 9.2 a boolean variable $p_{\mathsf{A},true}$ can be defined. Its value is related to that of $p_{\mathsf{A}}(\mathsf{i})$ by the invariant $\Box(p_{\mathsf{A},true} \;\Leftrightarrow\; p_{\mathsf{A}}(\mathsf{i}))$ which is associated with the state $\mathsf{mode}$. The invariant $\Box p_{\mathsf{A}}(\mathsf{i})$ of the state $\mathsf{mode.A}$ is replaced by the invariant $\Box p_{\mathsf{A},true}$.

Causal dependency among entities is interesting only for entities which influence other entities when they are changed in a step. Entities which are refered to only in event expressions do not need to be related causally with other entities. Changing these entities in a step does not influence other entities in the same step. Even if an event expression is part of an action expression the entities refered to in the event expression do not need to be included in the causality relation. Remember that condition expressions are considered as event expressions, cf. Section 5.1.1.

### 9.5.2 Deriving Causal Dependency

The causal dependencies among the entities in a constrained statechart are not expressed explicitly. In this section we present some heuristics to derive a relation

$$\rightarrow\ \subseteq En \times En$$

which expresses causal dependency among entities of a constrained statechart. We introduce a syntactical construct to make causal dependencies more explicit. At last we show how to change a constrained statechart such that causal dependencies are inverted and without changing the semantics of the statechart.

It is difficult to make the dependency relation explicit because there are no syntactical means up to now in our combination of statecharts and TLA to express causal dependency. Note that logical operators do not express causal dependency. For example let us consider the implication $p(\mathsf{x}) \Rightarrow q(\mathsf{y})$. This implication does not express that $\mathsf{y}$ depends on $\mathsf{x}$ because it is equivalent to the implication $\neg q(\mathsf{y}) \Rightarrow \neg p(\mathsf{x})$. Similarly, the equality $\mathsf{x} = f(\mathsf{y})$ is symmetrical, it does not express that $\mathsf{x}$ depends on $\mathsf{y}$.

If we consider temporal constraints the situation becomes even more complicated. Assume for example that the invariant $\Box p(I_1, I_2)$ is associated with a statecharts-state $s$. This temporal constraint can be understood in different ways:

1. $s$ can be active only if $p(I_1, I_2)$ is valid.
2. If $s$ is active, then $p(I_1, I_2)$ is valid.
3. The value of $I_2$ depends on $I_1$, as described by $p(I_1, I_2)$, if $s$ is active.
4. The value of $I_1$ depends on $I_2$, as described by $p(I_1, I_2)$, if $s$ is active.

The same behaviour is described in any of the four cases. But the dependency relations are quite different. They can be depicted as (1) $I_1 \to s \leftarrow I_2$, (2) $I_1 \leftarrow s \to I_2$, (3) $s \to I_1 \leftarrow I_2$, and (4) $s \to I_2 \leftarrow I_1$.

It is difficult to separate these cases by using heuristics. Therefore we introduce a syntactical construct to distinguish the first of these cases from the other ones. We use a horizontal dashed line to separate the box of a statecharts-state. Above this line temporal constraints are written which correspond to case 1. Below this temporal constraints are written which correspond to cases 2 to 4. These lines have been used already in the statechart in Figure 9.2. If these cases have to be distinguished for an and-state, we separate the rectangle containing the name of the and-state. As an example see the statechart in Figure 9.12, page 230.

Causal dependency has to be derived for local entities only. For input and output entities we assume that no input entity depends on another entity of this statechart and that no output entity influences another entity of this statechart.

In the following heuristics we use knowledge from the structure and the semantics of a statechart and knowledge which can be seen as good modeling practice.

**Implicit Events** The events generated implicitly when a state becomes active or inactive and when a variable is accessed in a step depend on the corresponding state and variable, respectively. For a state $s$ we have $s \to \mathsf{en}(s)$ and $s \to \mathsf{ex}(s)$. For a variable $u$ we have $u \to \mathsf{ch}(u)$, $u \to \mathsf{wr}(u)$, and $u \to \mathsf{rd}(u)$.

**Action Expressions** By executing an action expression values of variables can be changed and events can be generated. These variables and events are local or output entities. They depend on states which become or remain active when the action expression is executed. Let $a$ be an action expression and $o$ be an entity changed when executing $a$. The state $s$ on which $o$ depends is determined as follows:

- If $a$ is the action expression of a static reaction or an action expression to be executed when leaving or entering a state, then $o$ depends on this state.
- If $a$ is the action expression of a transition segment $t$ which belongs to an initial compound transition, then $o$ depends on the scope of this compound transition. Because $t$ can be part of different compound transitions the entity $o$ can depend on different states.
- If $a$ is the action expression of a transition segment belonging to a continuation compound transition then $o$ depends on the source of this continuation compound transition. As for initial compound transitions, $o$ can depend on several states.
- If $a$ is the action expression of a static transition, then $o$ depends on the source of the static transition. This implies that the entities of an action expression of a static transition and an action expression of a transition segment from a state to the same state depend on different states. In the first case the entity depends on the source of the static transition, in the second case it depends on the scope of the transition. In both cases it depends on the lowest state which is guaranteed to remain active.

We aim at an acyclic dependency relation. If the states as described before depend on these entities of action expressions, then these cycles become apparent.

**Explicit Dependencies** We introduced a horizontal dashed line in boxes in a statechart to distinguish between the cases 1 and the cases 2 to 4 before. Let $\Box p(l_1, \ldots l_n)$ be an invariant which is associated with a state $s$ such that this situation corresponds case 1. In this case the state $s$ depends on each of the variables $l_i$.

In the other cases at least one of the $l_i$ has to depend on $s$. It has to be determined by other heuristics which of the $l_i$ depends on $s$ and how the $l_i$ are related themselves.

This heuristic can be used to determine in the statechart in Figure 9.2 that the substates of the state $\mathsf{mode}$ depend on the input variable $\mathsf{i}$ and that the substates of $\mathsf{output}_j$ depend on those of $\mathsf{mode}$. We obtain the following relations:

$$\begin{array}{lllll} \mathsf{i} & \rightarrow & \mathsf{mode.A} & \rightarrow & \mathsf{output}_j\mathsf{.AB} \\ \mathsf{i} & \rightarrow & \mathsf{mode.B} & \rightarrow & \mathsf{output}_j\mathsf{.AB} \\ \mathsf{i} & \rightarrow & \mathsf{mode.C} & \rightarrow & \mathsf{output}_j\mathsf{.C} \end{array}$$

**(Non-)Dependency on Orthogonal States** Let $\Box p(l_1, \ldots, l_n)$ be an invariant associated with a state $s_1$ such that this corresponds to one of the cases 2 to 4 before. If an entity $l_i$ depends already on a state $s_2$ orthogonal to $s_1$, then we assume that $l_i$ depends neither on the other entities $l_j$ nor on $s_1$. On the other hand, if $l_i$ does not depend on a state $s_2$ orthogonal to $s_1$, then we assume that $l_i$ is restricted by this constraint, that is $l_i$ depends on $s_1$. In addition, if another entity $l_j$ depends on a state orthogonal to $s_1$, then we assume that $l_i$ depends on $l_j$, but not vice versa. We obtain $l_j \rightarrow l_i$. If $l_j$ is an input entity we use it in the same way as an entity depending on a state orthogonal to $s_1$.

This heuristic can be used to derive further dependencies from already existing ones.

Assume that we do not know in the statechart in Figure 9.2 that $o_j$ is an output entity. Because y is an input we can derive by this heuristic that $o_j$ depends on $output_j$.AB, on $output_j$.C, and on y.

$$
\begin{array}{lcl}
output_j.AB & \rightarrow & o_j \\
output_j.C & \rightarrow & o_j \\
y & \rightarrow & o_j
\end{array}
$$

**Dependency and Non-Orthogonal States** We consider it as useful that the dependency relations between an entity and two non-orthogonal states coincide. Remember that two states are non-orthogonal, if one of them is a descendant of the other one or if their least common ancestor is an or-state. This implies that all direct descendants of an or-state coincide on the causal dependencies on the entities to which they are related in common.

Similarly to the heuristic before this one can be used to derive further dependencies from already existing ones.

**Dependency Between Hierarchically Related States** Whether a state is active can depend on other states above or below this state. Let $s$ be a state with the substates $s_i$, $1 \leq i \leq n$. If $s$ is an and-state then $in(s) \Leftrightarrow \bigwedge_{1 \leq i \leq n} in(s_i)$, if $s$ is an or-state, then $in(s) \Leftrightarrow \bigvee_{1 \leq i \leq n} in(s_i)$

It is clear that a substate depends on its parent. The substate cannot be active if its parent is not active. But it is more important to determine whether a state depends on its descendants. To avoid cycles in the dependency relation we do not include in the relation $\rightarrow$ that a state depends on its parent.

As a general rule, a state $s_1$ depends on a substate $r$ if $r$ depends itself on some entity. Before we make this rule more precise we present an example. In Figure 9.4 there are three statecharts which differ in the transition segments leaving the state D. In any of the three statecharts the state D depends on the entity i.

In each of the statecharts in Figures 9.4(a)-9.4(c) the state D depends on the variable i.

In the statechart in Figure 9.4(a) the state A is left if D is left. That A is left does not depend on whether the transition segment with target state B has D or A as its source state. In this statechart A depends clearly on D.

In the statechart in Figure 9.4(b) it is possible that the state C becomes active again if D has to be left. But it is still possible that A is left. Hence we consider that A depends on D. Note that we do not include D $\rightarrow$ C in the dependency relation

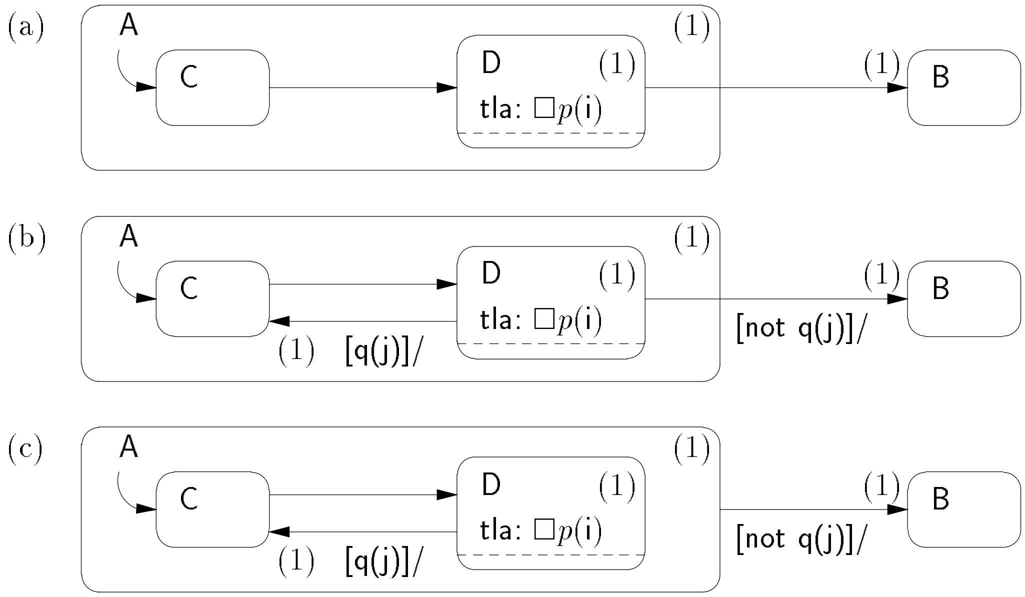



Figure 9.4: Causal dependency between hierarchically related states

because otherwise we get a cyclic relation. If we argue that C depends on D because leaving D can imply that C is entered then it can be argued by the same reasoning that D depends on C. A does not depend on j either because we include in the relation only those entities which influence other entities in the same step if they are changed.

In the statechart in Figure 9.4(c) we assume that A does not depend on D because there is no transition segment leaving A with D as its source state. Nevertheless, if the transition segment from D to C is not enabled then A has to be left if D is left. We do not consider this a well-written description. If it would be intended that A is left if D is left then we propose to use a statechart as in Figure 9.4(b).

By the heuristic we obtain dependencies as follows. Let $s_1$ be a state which depends on an entity $l$ and let $s_2$ be the parent of $s_1$. If $s_2$ is an and-state then in any case $s_2$ depends on $s_1$. If $s_2$ is an or-state, $s_1$ is a source state of a compound transition $T_1$, and the scope of $T_1$ is strictly above $s_2$, then $s_2$ depends on $s_1$ because $s_2$ has to be left if $s_1$ is left by taking $T_1$. Thereby we obtain the dependencies of the statecharts in Figures 9.4(a) and 9.4(b). If $s_2$ is an or-state and $s_1$ is not a source of a compound transition as described before, then $s_2$ does not depend on $s_1$. This corresponds to the dependencies in the statechart in Figure 9.4(c).

In the statechart in Figure 9.2 this heuristic can be used to derive that the states mode and $\mathsf{output}_j$ depend on their substates. In turn, the state root depends on mode and $\mathsf{output}_j$.

Before we investigate how the dependency relationship can be used we show how the semantics of statecharts can be used to break cycles in the dependency relation.

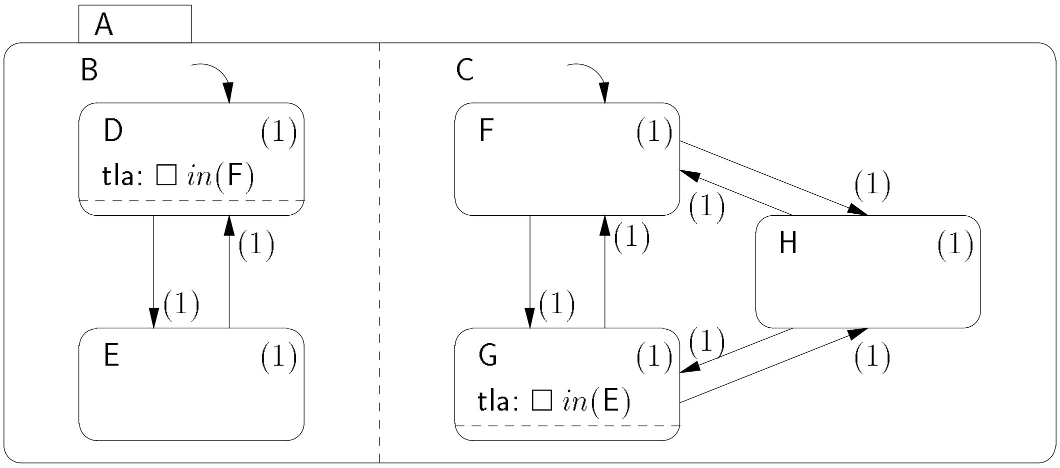



Figure 9.5: Inverting causal dependencies

**Inverting Causal Dependency** By several of the heuristics we derived causal dependencies using the semantics of statecharts. This knowledge can also be used in a completely different way. Note that the following is not a heuristic to derive causal dependencies from a constrained statechart.

The semantics of statecharts can be used to invert causal dependencies. This is useful to break cycles in the dependency relation and to straighten dependencies in opposite directions between orthogonal states.

As an example consider the statechart in Figure 9.5. In this statechart the state D depends on F and the state G depends on E. Hence there are causal dependencies in opposite directions between substates of B and C: $\mathsf{F} \rightarrow \mathsf{D}$ and $\mathsf{E} \rightarrow \mathsf{G}$. These dependencies can be extended to $\mathsf{F} \rightarrow \mathsf{D} \rightarrow \mathsf{B}$ and $\mathsf{E} \rightarrow \mathsf{G} \rightarrow \mathsf{C}$.

In this statechart D can be active only if F is active. Provided that C is active, F is inactive if and only if either G or H is active. Similarly, if B is active then D is inactive if and only if E is active. Because $in(\mathsf{A}) \Leftrightarrow in(\mathsf{B}) \wedge in(\mathsf{C})$ we have

$$\begin{array}{rl} & in(\mathsf{A}) \Rightarrow (in(\mathsf{D}) \Rightarrow in(\mathsf{F})) \\ \Leftrightarrow & in(\mathsf{A}) \Rightarrow (\neg\, in(\mathsf{F}) \Rightarrow \neg\, in(\mathsf{D})) \\ \Leftrightarrow & in(\mathsf{A}) \Rightarrow (in(\mathsf{G}) \vee in(\mathsf{H}) \Rightarrow in(\mathsf{E})) \end{array}$$

Hence, the invariant $\Box\, in(\mathsf{F})$ of D can be removed and the invariant $\Box\, in(\mathsf{E})$ can be associated with the states G and H without changing the meaning of the statechart. After these changes of the invariants no substate of B depends on a substate of C.

Note that inverting causal dependencies as explained here should not be used to obtain dependencies which do not correspond to the dependencies existing in the described system. Instead, this inversion should be used to change descriptions such that they reflect the causal dependencies in the system.

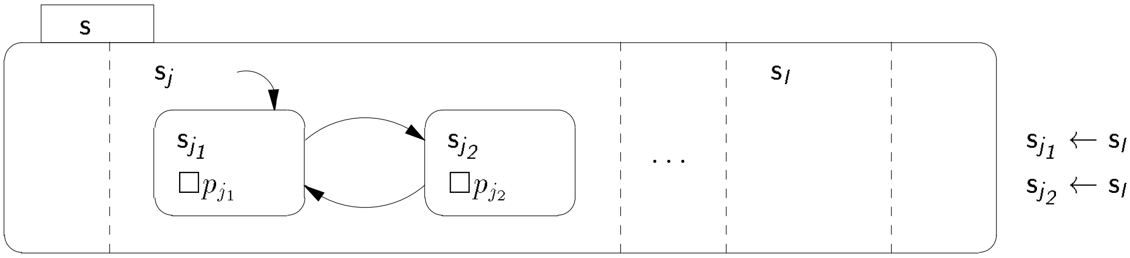



Figure 9.6: Restrictions on siblings

### 9.5.3 Using Causal Relationship

By the heuristics given in Section 9.5.2 a dependency relation between entities of a statechart can be derived. The dependency relation can be used for two purposes.

At first it makes implicit knowledge about a constrained statechart explicit. This eases understanding of the statechart by human readers. For developers of descriptions it can be a hint to check or to change a description if the dependency relation is not acyclic or contains dependencies in opposite directions as in the statechart in Figure 9.5.

The second use of the dependency relation is that it can be used to structure the proof that a constrained statechart is consistent. We use here the transitive closure of the dependency relation. In general the complete statechart with all its variables has to be considered as a whole when it is shown that a constraint is enabled. By using the dependency relation the statechart can be split into smaller parts according to the structure of the statechart. Because and-states are used in statecharts to describe concurrency it is natural to consider one or more substates of an and-state as one part which is examined individually. Let $s$ be an and-state with substates $s_i$, $1 \leq i \leq n$. Then a substate $s_j$ of $s$ can be examined separately of another substate $s_k$ if neither $s_j$ nor any of its descendants depends on $s_k$ or a descendant of $s_k$. If there are mutual dependencies between two substates $s_j$ and $s_k$ of $s$ then it is shown for them together that they can remain active if a step of the statechart is taken.

Showing that a substate $s_j$ can remain active if a step of the statechart is taken can impose further restrictions on siblings of $s_j$ upon which $s_j$ depends. Assume for example that $s_j$ has itself two substates $s_{j_1}$ and $s_{j_2}$ and that each of $s_{j_1}$ and $s_{j_2}$ depend by an invariant on a sibling $s_l$ of $s_j$, cf. Figure 9.6. If a step of $s_j$ is taken in which $s_{j_1}$ becomes inactive and $s_{j_2}$ becomes active, then it has to be possible to take a step of $s_l$ such the invariant of $s_{j_2}$ becomes valid. It can be assumed in this step that the invariant of $s_{j_1}$ is valid at the beginning of the step.

This implies that it is not sufficient to consider arbitrary steps when showing that a substate $s_j$ can remain active. It is necessary to show that steps can be taken which allow that siblings of $s_j$ depending on $s_j$ can take steps, too. Therefore we suggests to work backwards along the dependency relation. Assume that we want to show that a substate $s_j$ can remain active, that $s_j$ depends on a sibling $s_l$, and that compound transitions with scope below $s_l$ have to be taken to show that $s_j$ can remain active, then the steps corresponding to these compound transitions are collected as proof obligations to be shown when examining $s_l$.

In the statechart in Figure 9.2 the state $\mathsf{mode}$ does not depend on $\mathsf{output}_j$. Working backwards we show at first that $\mathsf{output}_j$ can remain active. $\mathsf{output}_j$ can remain active because the disjunction of the invariants of its substates is valid, provided that in the state $\mathsf{mode}$ it is always possible in a step that one of the substates of $\mathsf{mode}$ becomes or remains active. Let us assume that the functions $g_{AB}$ and $g_C$ are computable, then it is always possible to assign a value to $\mathsf{o}_j$.

Because we assumed that $\Box(p_A(\mathsf{i}) \vee p_B(\mathsf{i}) \vee p_C(\mathsf{i}))$ is valid it is always possible that one of the substates of $\mathsf{mode}$ is active. Hence $\mathsf{mode}$ can always remain active and the proof obligations necessary to show that $\mathsf{output}_j$ remains active are valid.

In this simple example we have been able to split the proof of consistency into separate parts and to prove these parts in an order derived from the dependency relation. Each of the parts is an instance of a specific pattern, such patterns are presented in Section 9.6.

Let us note at last that usually in the descriptions REQ and SPEC there exist more dependencies than in the descriptions RW-O and RW-M because the first ones are developed by extending the latter ones. This implies that the descriptions REQ and SPEC usually can be split into fewer and therefore larger parts.

## 9.6 Patterns of Consistent Statecharts

The dependency relation introduced in Section 9.5 can be used to split a statechart into parts and to show for each of the parts individually that a constrained step can be taken. It remains to carry out the corresponding proofs. This task can be simplified if a statechart or a part of it is an instance of a specific pattern for which it has already been shown that it is consistent. In this case it has to be shown only that the statechart is an instance of such a pattern.

In this section we present such patterns and conditions which guarantee that instances of these patterns are consistent. We view these patterns as complete statecharts, but they can be applied to parts of statecharts as identified in Section 9.5.3, too. In the first of these patterns the relation between temporal constraints and the event and action expressions of transition segments is relevant. The second pattern is a special case of the first one. In the third pattern the relation between temporal constraints of hierarchically related states is relevant. As before we consider only invariants as temporal constraints.

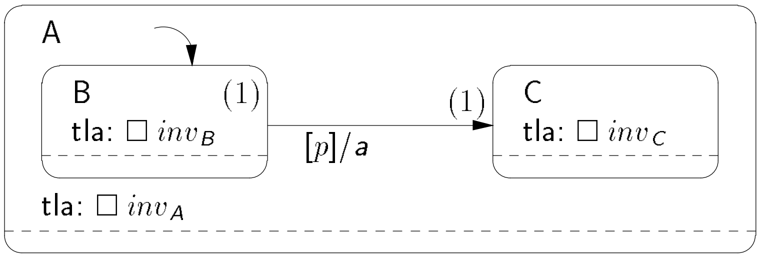



Figure 9.7: Consistent statecharts: Leaving a state

### 9.6.1 Invariants and Conditions of Transitions

In this section we show under which conditions an or-state can remain active. In this case, if a compound transition between descendents of the or-state has to be taken, then a compound transition can be taken such that the invariants of the or-state itself and the target state are satisfied.

We start with a statechart as shown in Figure 9.7. This statechart is extended to the case that there are several compound transitions with source state B. We assume that no *External. change*-steps occur.

In the statechart in Figure 9.7 the state B can remain active as long as its invariant is valid. If $inv_B$ becomes invalid then it has to be possible that the compound transition from B to C is taken. Let $\mathcal{N}_a$ denote the TLA-action corresponding to the action expression $a$. Taking the transition is a $(p \wedge \mathcal{N}_a \wedge inv'_C)$-step. This can be strengthened further because $inv_A$ remains valid in this step.

To show that the statechart is consistent, it has to be shown that steps can be taken and that the statechart can be initialised. To show that steps can be taken it is sufficient to show

$$\begin{aligned} & Enabled(in(\mathsf{B}) \wedge inv_A \wedge inv_B \wedge \neg\, inv'_B) \\ \Rightarrow\ & Enabled \wedge\ p \wedge \mathcal{N}_a \\ & \qquad\quad \wedge\ in'(\mathsf{C}) \wedge inv'_C \\ & \qquad\quad \wedge\ inv_A \wedge inv'_A \end{aligned} \tag{9.6}$$

for all reachable statuses of the statechart. Instead of all reachable statuses weaker characterisations of the statuses can be used, cf. Section 9.3.

Equation 9.6 can be extended to the case that more than one compound transition can be taken to leave B. In this case it is sufficient that one of these compound transitions can be taken. For the sake of simplicity we assume that each of these compound transitions consists of a single transition segment from B to another descendant $\mathsf{C}_i$ of $\mathsf{A}$.

$$\begin{array}{l} \mathit{Enabled}(\mathit{in}(\mathsf{B}) \wedge \mathit{inv}_{\mathsf{A}} \wedge \mathit{inv}_{\mathsf{B}} \wedge \neg\, \mathit{inv}'_{\mathsf{B}}) \\ \Rightarrow \bigvee_{t \in T} \bigvee_{\mathsf{C}_i \in \mathit{descendants}(\mathsf{B})} \begin{array}{l} \wedge\ \mathit{source}(t) = \mathsf{B} \\ \wedge\ \mathit{target}(t) = \mathsf{C}_i \\ \wedge\ \mathit{Enabled} \wedge\ p_i \wedge \mathcal{N}_{\mathsf{a}_i} \\ \qquad\qquad \wedge\ \mathit{in}'(\mathsf{C}_i) \wedge \mathit{inv}'_{\mathsf{C}_i} \\ \qquad\qquad \wedge\ \mathit{inv}_{\mathsf{A}} \wedge \mathit{inv}'_{\mathsf{A}} \end{array} \end{array} \tag{9.7}$$

The predicates and TLA-actions $p_i$, $\mathit{inv}_{\mathsf{C}_i}$, and $\mathcal{N}_{\mathsf{a}_i}$ are defined analogously to the case before.

The statechart in Figure 9.7 can be initialised when $\mathit{inv}_{\mathsf{B}}$ is valid if $\mathsf{A}$ becomes active and $\mathit{inv}_{\mathsf{A}}$ is valid, too.

$$\begin{array}{l} \mathit{Enabled}(\mathsf{en}(\mathsf{A}) \in E'_{\mathit{exists}} \wedge \mathit{inv}'_{\mathsf{A}}) \\ \Rightarrow \mathit{Enabled} \wedge\ \mathsf{en}(\mathsf{A}) \in E'_{\mathit{exists}} \wedge \mathit{inv}'_{\mathsf{A}} \\ \qquad\qquad \wedge\ \mathsf{en}(\mathsf{B}) \in E'_{\mathit{exists}} \wedge \mathit{inv}'_{\mathsf{B}} \end{array} \tag{9.8}$$

If $\mathsf{B}$ itself has substates, then it has to be possible, that a configuration below $\mathsf{B}$ becomes active. These states become active by taking a continuation compound transition because we require that an *Execute.step* is taken.

$$\begin{array}{l} \mathit{Enabled} \wedge\ \mathit{Execute.step} \\ \qquad \wedge\ \mathsf{en}(\mathsf{A}) \in E'_{\mathit{exists}} \wedge \mathit{inv}'_{\mathsf{A}} \\ \Rightarrow \mathit{Enabled} \wedge\ \mathit{Execute.step} \\ \qquad\qquad \wedge\ \mathsf{en}(\mathsf{A}) \in E'_{\mathit{exists}} \wedge \mathit{inv}'_{\mathsf{A}} \\ \qquad\qquad \wedge\ \mathsf{en}(\mathsf{B}) \in E'_{\mathit{exists}} \wedge \mathit{inv}'_{\mathsf{B}} \\ \qquad\qquad \wedge \bigwedge_{s \in \mathit{descendants}^{+}(\mathsf{B})} \begin{array}{l} \mathsf{en}(s) \in E'_{\mathit{exists}} \\ \Rightarrow \mathit{inv}'_{s} \end{array} \end{array} \tag{9.9}$$

Up to now we considered only invariants which influence whether a state is active. It has also to be shown that the invariants written below the dashed line can be satisfied if a step is taken.

Let $\widehat{\mathit{inv}_{\mathsf{C}}}$ be such an invariant of the state $\mathsf{C}$ in the statechart in Figure 9.7. For example, this invariant has to remain valid if a step is taken in which $\mathsf{C}$ is active.

$$\begin{array}{l} \mathit{Enabled}(\mathit{in}(\mathsf{C}) \wedge \mathit{inv}_{\mathsf{C}} \wedge \mathit{inv}'_{\mathsf{C}}) \\ \Rightarrow \mathit{Enabled} \wedge\ \mathit{in}(\mathsf{C}) \wedge \mathit{inv}_{\mathsf{C}} \wedge \mathit{inv}'_{\mathsf{C}} \\ \qquad\qquad \wedge\ \mathit{inv}_{\mathsf{A}} \wedge \mathit{inv}'_{\mathsf{A}} \\ \qquad\qquad \wedge\ \widehat{\mathit{inv}_{\mathsf{C}}} \wedge \widehat{\mathit{inv}'_{\mathsf{C}}} \end{array} \tag{9.10}$$

A similar condition can be given for steps in which $\mathsf{C}$ becomes active.

### 9.6.2 Defining Modes by Invariants

From the statecharts in Section 9.6.1 a special case can be obtained by setting the event and action expressions of the transition segments between the states to *true*. Equations 9.6 and 9.7 can be simplified correspondingly. In this case it depends only on the invariants whether a state can be active. If the states correspond to modes of the system then the modes are defined by the invariants.

Instead of the simplified formulae it is sufficient to show that the disjunction of the invariants of the target states is valid if a compound transition has to be taken.

$$
\begin{aligned}
& inv_{\mathsf{A}} \wedge \neg\, inv_{\mathsf{B}} \\
& \Rightarrow \bigvee_{t \in T} \; \bigvee_{\mathsf{C}_i \in descendants(\mathsf{B})} \begin{array}{l} \wedge\ source(t) = \mathsf{B} \\ \wedge\ target(t) = \mathsf{C}_i \\ \wedge\ inv_{\mathsf{C}_i} \end{array}
\end{aligned}
$$

This implication is valid especially if each substate of A can be reached from each other one and the disjunction of their invariants is a tautology.

To consider initialisation of the statechart let us assume that several substates of A are targets of different continuation compound transitions used to initialise the statechart. Let $\{\mathsf{D}_i | 1 \leq i \leq k\} \subseteq descendants(\mathsf{A})$ be the set of these states. To show that A can be initialised it is sufficient to show that the disjunction of the invariants of these states is valid.

$$
inv_{\mathsf{A}} \;\Rightarrow\; \bigvee_{1 \leq i \leq k} inv_{\mathsf{D}_i}
$$

As before, this implication is valid, if each substate of A can become active when initialising the statechart and if the disjunction of their invariants is a tautology.

### 9.6.3 Relations Between Constraints of And-States

We consider here the relation between the invariants of an and-state and its substates. Let $s$ be an and-state with substates $s_i$, $1 \leq i \leq n$. In addition, we assume that $s_{i+1}$ depends on $s_i$, cf. Section 9.5.3.

The state $s$ can remain active in a step if each of its substates $s_i$ can also remain active. For each of the $s_i$ it has to be shown that its invariants remain active if a step can be taken in which the invariants of $s$ and of the preceding states in the dependency relation remain active.

$$\begin{aligned}\bigwedge_{0\leq i\leq n-1} \quad & \mathit{Enabled} \wedge \mathit{Execute.step} \\ & \quad \wedge\ \mathit{inv}_s \wedge \mathit{inv}'_s \\ & \quad \wedge \bigwedge_{1\leq j\leq i} \mathit{inv}_{s_j} \wedge \mathit{inv}'_{s_j} \\ & \Rightarrow \mathit{Enabled} \wedge \mathit{Execute.step} \\ & \qquad \wedge\ \mathit{inv}_s \wedge \mathit{inv}'_s \\ & \qquad \wedge \bigwedge_{1\leq j\leq i} \mathit{inv}_{s_j} \wedge \mathit{inv}'_{s_j} \\ & \qquad \wedge\ \mathit{inv}_{s_{i+1}} \wedge \mathit{inv}'_{s_{i+1}}\end{aligned} \tag{9.11}$$

In addition it has to be shown that $s$ can be initialised. This can be described analogously to Equation 9.11.

## 9.7 Consistency and Liveness

Up to now we considered only safety-formulae as temporal constraints. But most descriptions contain liveness-formulae as temporal constraints, too. Liveness-formulae are relevant w.r.t. consistency because they can be used to prevent that a status of a statechart is reached but not changed by further steps of the statechart. Such *status*-stuttering-steps[3] are not sufficient to show that a statechart is consistent if liveness-properties are expressed.

In this section we present sufficient conditions for two classes of liveness-constraints to show that a constrained statechart is consistent. The classes are fairness- and leadsto-formulae. The leadsto-formulae are of the form $p \leadsto q$, where $p$ and $q$ are predicates. The main idea is to determine TLA-actions for which it has to be shown that corresponding steps of the constrained statechart can be taken.

For the sake of simplicity we assume that a temporal constraint $L$ has to be valid only if the state with which it is associated remains active continuously, that is $s \rhd_{\Diamond\Box} L$ for a state $s$, cf. Section 6.4.2.

**Fairness-Formulae** Let $L$ be a fairness-formula, that is $L \equiv SF_{\langle\rangle}(\mathcal{N})$ or $L \equiv WF_{\langle\rangle}(\mathcal{N})$ for a TLA-action $\mathcal{N}$, which is associated with a statecharts-state $s$. When showing that $s$ can remain active in a step we show in addition that a constrained $\mathcal{N}$-step can be taken if $\mathcal{N}$ is enabled.

$$\begin{aligned}& \mathit{in}(s) \wedge \mathit{Enabled}\,\mathcal{N} \\ & \Rightarrow \mathit{Enabled}(\mathcal{N} \wedge \mathit{Can.stay}(s))\end{aligned}$$

This implies that $\mathcal{N} \wedge \mathit{Constrained.step}$-steps can be taken without leaving $s$ if they become enabled. Hence the fairness-constraint can be satisfied.

[3] More precisely, these are $\langle S_{active}, E_{exists}, w\rangle$-stuttering steps. The value of the variable $\mathit{step.taken}$ is changed when an $\mathit{Execute.step}$-step is taken.

**Leadsto-Formulae** Let $L$ be a leadsto-formula $p \leadsto q$, where $p$ and $q$ are predicates over $S_{active}$, $E_{exists}$, and $w$, and let $L$ be associated with a state $\mathsf{s}$. To show that a constrained statechart is consistent we split $L$ into a sequence of leadsto-formulae $L_i \equiv p_i \leadsto q_i$, $1 \leq i \leq n$, with $p = p_1$ and $q = q_n$. We assume that there are states $\mathsf{s}_i$ with $\mathsf{s}_i \rhd L_i$ and we require that the states $\mathsf{s}_i$ are active if $\mathsf{s}$ is active, that is $\Box(in(\mathsf{s}) \Rightarrow \bigwedge_{1 \leq i \leq n} in(\mathsf{s}_i))$. To form a sequence of leadsto-formulae we require that $p_{i+1}$ is implied by $q_i$: $\Box((in(\mathsf{s}) \wedge q_i) \Rightarrow p_{i+1})$. If these properties are valid and if the $L_i$ are valid then $L$ is valid, too.

It remains to show that the $L_i$ are valid. We use the proofrule WF1 of TLA, cf. Appendix A.1 to reduce the proof of $L_i$ to the case of fairness-formulae discussed before. We have to show at first weak fairness of $p_i \wedge q_i'$. At second we have to show that if $p_i$ is valid when the state $\mathsf{s}_i$ is active, then each step in which $\mathsf{s}_i$ remains active is a $p_i \wedge (p_i' \vee q_i')$-step. The latter condition means that $p_i$ remains valid until a $(p_i \wedge q_i')$-step is taken.

In Section 9.5.3 we explained how to work backwards along the dependency relation to show that parts of a statechart are consistent. There it was possible that in a state certain steps have to be enabled to ensure that depending states can take their steps, too. For these steps and for the ones identified in this section it has to be shown that they can or have to be taken if the corresponding state remains active.

## 9.8 Consistency of the Lift System

In this section we apply the heuristics to derive causal dependencies, cf. Section 9.5, and the reasoning about liveness-constraints, cf. Section 9.7, to show that descriptions are consistent. We show that the descriptions RW-O and REQ of the lift system, cf. Chapter 7, are consistent.

Because REQ was derived from RW-O by associating further temporal constraints it is a refinement of RW-O. Therefore, if REQ can be shown to be consistent, then RW-O is consistent, too. Nevertheless, to present a more simple example we show explicitly that RW-O is consistent.

In the descriptions in Chapter 7 we did not use the dashed line introduced in Section 9.5 to distinguish whether entities depend on a state or whether they influence it. We consider here auxiliary predicates, for example *request.same*, as entities and we assume that their definitions depend on states. For all other temporal constraints we assume that the entities occurring in them influence the corresponding state.

To show consistency it is sufficient that there exists *one* run satisfying the description. In the examples considered here this could be a run in which there are no requests for the lift cabin at all. Although such a run is sufficient to show consistency, it is a quite useless run. To show that more interesting runs exist we do not make any assumptions on the existence of requests in the following. This implies that we show actually a stronger property than consistency, namely receptiveness of the parts of the descriptions describing the behaviour of the plant. We come back to this point in Section 9.8.3.

### 9.8.1 Consistency of Rw-o

In this section we show that the description Rw-o of the lift system, cf. Section 7.1 and the statecharts in Figures 7.1 through 7.4, is consistent. Because this description contains only a few temporal constraints it is easy to show that it is consistent.

We start by determining the causal dependencies between the entities occurring in the description. The statechart in Figure 7.4 can be considered as a closed system. Therefore there are neither input nor output entities. All entities are local ones. We take into account only those events which occur in event and action expressions, but not those which are generated implicitly without being used.

The dependencies between the entities are depicted in Table 9.1. The dependencies have been derived by using the heuristics presented in Section 9.5.2 from the action expressions and the temporal constraints. Note that the event expressions of the statechart in Figure 7.3 are not used to determine the dependency relation. Changing one of these entities does not influence the step in which they are changed.

Rw-o is consistent if on the one hand rwo can remain active if a step of the statechart can be taken. On the other hand it has to be shown that compound transitions can be taken which contribute to the satisfaction of the fairness constraints. In this example these are the compound transitions which consist of the transition segments between the substates of rwo.plant.cabin.position.within, cf. Figure 7.1, and the transition segments from rwo.plant.door$_f$.opened to rwo.plant.door$_f$.closed, $min.floor \leq \mathsf{f} \leq max.floor$, cf. Figure 7.2.

W.r.t. the states there are dependencies of substates of rwo.plant.cabin.position on those of rwo.plant.cabin.dir. Hence we show that rwo.plant.cabin.position can remain active before we show that rwo.plant.cabin.dir can remain active. For the other substates of rwo it is shown in an arbitrary order that they can remain active. We

| | |
|---|---|
| press$_f$ | $\leftarrow$ environment |
| floor | $\leftarrow$ plant.cabin.position |
| plant.cabin.position.within.nearby | $\leftarrow$ plant.cabin.dir.stopped |
| plant.cabin.position.within.between | $\leftarrow$ plant.cabin.dir.stopped |
| plant.cabin.position.within.onfloor | $\leftarrow$ plant.cabin.dir.stopped |
| | $\leftarrow$ en(plant.cabin.dir.stopped) |
| | $\leftarrow$ en(plant.cabin.position.within.onfloor) |
| plant.cabin.position.within | $\leftarrow$ plant.cabin.dir.up, _down |
| plant.cabin.position.outside | $\leftarrow$ plant.cabin.dir.up, _down |

Table 9.1: Causal dependencies in Rw-o of the lift system

omit the prefix rwo of the names of states subsequently.

environment and plant.request$_f$, cf. Figures 7.4 and 7.3, can remain active in every step of the statechart because no temporal constraints are associated with them or their descendants. For the same reason these states can become active initially.

plant.door$_f$, cf. Figure 7.2. As in the case before these states can become active initally and can remain active when taking a step of the statechart. Because taking the compound transitions from door$_f$.opened to door$_f$.closed is neither restricted by event expressions nor by temporal constraints it can be taken in each step if its source state is active. Hence it is possible to satisfy fairness on these compound transitions.

plant.cabin.position, cf. Figure 7.1, can become active because the invariants of the states which become active initially are valid. If the invariant of the state plant.cabin.position.within becomes invalid in a step then it is still possible to take a step: The state plant.cabin.position.outside can become active in this step. To satisfy the liveness-constraint of plant.cabin.position.within it has to be shown that the compound transitions between the substates plant.cabin.position.within can be taken if plant.cabin.position.within itself remains active in a step. These substates depend on substates of plant.cabin.dir. Therefore, it must be possible either to stay in one of the states plant.cabin.dir.up and plant.cabin.dir.down, to enter them, or to leave them.

plant.cabin.dir, cf. Figure 7.1, can be initialized or can remain active in every step because it is a statechart without temporal constraints. In addition, in each step it is possible to take one of the compound transitions which are necessary to show that plant.cabin.position can remain active.

This shows that RW-O is consistent, even if we do not assume anything on the occurrence of the events press$_f$. Actually, we have shown a stronger property. In RW-O it is not necessary to leave any of the states rwo.plant.cabin.position.within.onfloor and rwo.plant.door$_f$.closed. If these states are never left, then the liveness-constraints are still valid. Nevertheless, it is reasonable to check the preceding cases already in this simple description because they occur also in the proof of consistency of req. If they are not already valid in RW-O, then this potential inconsistency of REQ can be detected with less effort here.

### 9.8.2 Consistency of REQ

In this section we show that REQ of the lift system, cf. Section 7.3 and Figure 7.7, is consistent. Because this description contains more temporal constraints than RW-O the proof is more complex. As before we do not make any assumptions on the occurrence of the events press$_f$.

We concentrate on the state req.plant as shown in Figure 7.7. We omit in the following the prefix req.plant of the names of the states.

The dependencies between entities of RW-O as shown in Table 9.1 are also given in REQ. In addition we derive the dependencies shown in Table 9.2 from the formal description of the requirements. Auxiliary predicates, for example *request. same*, are considered as boolean variables. From the requirements 2 and 7 we do not derive dependencies. Requirement 2 is expressed by a leadsto-formula and requirement 7 does not refer to any entity at all.

There are mutual dependencies between the substates of cabin.dir and cabin.position. Therefore we show for the state cabin at once that it can remain active instead of showing this separately for its substates as has been done for RW-O. Furthermore, each of the states $\mathsf{door}_f$ depends on substates of cabin.position and of $\mathsf{request}_f$ and the state cabin.dir depends on each of the states $\mathsf{request}_f$.requested. Hence we get the dependencies

$$
\begin{array}{ll}
\mathsf{door}_f & \leftarrow \mathsf{cabin.position} \\
\mathsf{door}_f & \leftarrow \mathsf{request}_f \\
\mathsf{cabin.dir} & \leftarrow \mathsf{request}_f
\end{array}
$$

| | | |
|---|---|---|
| Req 1: | $\mathsf{door}_f$.opened | ← floor |
| | | ← position.within.onfloor |
| | | ← dir.stopped |
| Req 3: | *request. same* | ← floor |
| | | ← $\mathsf{request}_f$.requested |
| | *early. request* | ← floor |
| | | ← *early. request*$_f$ |
| | *early. request*$_f$ | ← $\mathsf{request}_f$.requested |
| | | ← position.onfloor, _.nearby, _.between |
| | | ← dir.stopped, _.up, _.down |
| | dir.up, _.down | ← position.within.onfloor |
| | | ← *request. same* |
| | | ← *early. request* |
| Req 4: | *request. up*, _. *down* | ← floor |
| | | ← $\mathsf{request}_f$.requested |
| | dir.up | ← *request. up* |
| | dir.down | ← *request. down* |
| Req 5: | dir.stopped | ← *request. same* |
| | | ← en(dir) |
| | $\mathsf{door}_f$.opened | ← $\mathsf{request}_f$.requested |
| Req 6: | *dirold* | ← dir.up, _.down |
| | dir | ← dir.up, _.down |

Table 9.2: Additional causal dependencies in REQ of the lift system

Therefore we show that the substates or $\mathsf{req.plant}$ can remain active in the order $\mathsf{door}_f$, $\mathsf{cabin}$, and finally $\mathsf{request}_f$.

**Fairness** Before we start to show that the substates of $\mathsf{req.plant}$ can remain active we determine which steps have to be taken to satisfy the fairness constraints. The liveness-constraints associated with substates of $\mathsf{req.plant}$ are

$$\begin{array}{ll} \mathsf{door}_f & \rhd\ WF_{\langle\rangle}(\, in(\mathsf{opened}) \wedge in'(\mathsf{closed})) \\ \mathsf{request}_f & \rhd\ in(\mathsf{requested}) \leadsto \neg\, in(\mathsf{requested}) \\ \mathsf{cabin.position.within} & \rhd\ \neg\, in(\mathsf{onfloor}) \leadsto in(\mathsf{onfloor}) \end{array}$$

For the temporal constraints of $\mathsf{door}_f$ it is clear that it has to be shown that $(in(\mathsf{opened}) \wedge in'(\mathsf{closed}))$-steps can be taken if $\mathsf{opened}$ is active. We will see that the temporal constraint of $\mathsf{cabin.position.within}$ is subsumed by the temporal constraints of the states $\mathsf{request}_f$.

The liveness-constraints of the states $\mathsf{request}_f$ are split into a sequence of leadsto-formulae which consists of three parts.

1. Change the direction of the movement of the lift cabin such that the cabin can move towards a requested floor
2. Move the cabin to the requested floor
3. Open the door on the requested floor

For the sake of simplicity we consider here only the case that the request is on a floor above the current position of the lift cabin.

In the first part of this sequence it is relevant that the direction of movement of the lift cabin can change. That a status is reached in which there is no request below the current position of the lift cabin is guaranteed by the second and third part of this sequence for the case that the lift cabin is moving downwards.

$$\begin{array}{l} \wedge\ request.up \\ \wedge\ \neg\, request.down \\ \wedge\ dirold = down \\ \leadsto \wedge\ request.up \\ \quad\ \ \wedge\ dirold = up \end{array} \tag{9.12}$$

The second part of the sequence consists of several subparts. These correspond to move the lift cabin one floor towards a requested floor by passing through the substates of $\mathsf{cabin.position.within}$.

$$\begin{array}{l} \wedge\ request.up \\ \wedge\ dirold = up \\ \wedge\ in(\mathsf{cabin.position.within.onfloor}) \\ \leadsto in(\mathsf{cabin.position.within.nearby}) \end{array} \tag{9.13}$$

$$
\begin{aligned}
&in(\mathsf{cabin.position.within.nearby.leave}) \leadsto in(\mathsf{\_.between}) \\
&in(\mathsf{cabin.position.within.between}) \leadsto in(\mathsf{\_.nearby.approach}) \\
&in(\mathsf{cabin.position.within.nearby.approach}) \leadsto in(\mathsf{\_.onfloor})
\end{aligned} \tag{9.14}
$$

The leadsto-formulae in Equation 9.14 can be used to show that the liveness-constraint of cabin.position.within can be satisfied.

The third part of the sequence consists of a single leadsto-formula.

$$
\begin{aligned}
&\wedge\ \mathit{request.same} \\
&\wedge\ \mathit{early.request} \\
&\wedge\ \mathit{floor} = \mathsf{f} \\
&\leadsto in(\mathsf{door_f.opened})
\end{aligned} \tag{9.15}
$$

The formula in Equation 9.12 is associated with the state cabin.dir, those in 9.13 and 9.14 with cabin.position.within, and the one in 9.15 with $\mathsf{door}_f$.

Now we show that the substates of req.plant can remain active when taking a step of the statechart.

$\mathsf{request}_f$ Except the liveness-constraints, which will be considered elsewhere, there are no temporal constraints associated with the states $\mathsf{request}_f$ and their descendants. Hence the states $\mathsf{request}_f$ can become or can remain active if an *Execute. step* of the statechart is taken.

cabin We show at first that the state cabin can become active. Thereafter we show for the possible combinations of substates of cabin.dir and cabin.position that cabin can remain active if steps are taken.

It depends on the temporal constraints of cabin.dir.stopped, cabin.position.within, and cabin.position.within.onfloor whether cabin can become active. Each of these temporal constraints becomes valid when the state is initialised. Hence cabin can become active.

To show that cabin can remain active it has to be shown that cabin.position.within can remain active because cabin.position.outside cannot become active. We show for each combination of substates of cabin.dir and cabin.position.within that cabin can remain active. We omit symmetrical cases w.r.t. dir.up and dir.down. It remains to consider the cases that dir.up and any of the substates of position.within are active and the case that dir.stopped and position.within.onfloor are active.

Case $in(\mathsf{dir.stopped}) \wedge in(\mathsf{position.within.onfloor})$ We have to show for the leadsto-formulae in Equations 9.12 and 9.13 that in each step in which the premise of one of the leadsto-formulae is valid that either the premise remains valid or the conclusion becomes valid. In addition, it has to be possible to take a step in which the conclusion becomes valid. Both properties are valid for these leadsto-formulae because it is possible that the lift cabin leaves a floor in the next step of the statechart after reaching it and serving a request on it. That the door on the corresponding floor can open and close fast enough has to be shown when showing that the states $\mathsf{door}_f$ remain active.

Case $in(\mathsf{dir.up}) \wedge in(\mathsf{position.within.nearby.leave})$ The leadsto-formulae in Equation 9.14 are relevant here. It is easy to see that it is possible to take a step in which $\mathsf{position.within.between}$ becomes active. The invariants of $\mathsf{dir.up}$ do not become invalid in such a step, hence $\mathsf{cabin}$ can remain active. These invariants cannot become invalid because the invariants of the states $\mathsf{door}_f\mathsf{.opened}$ imply that the doors are closed. Hence no request can be served in this step.

Case $in(\mathsf{dir.up}) \wedge in(\mathsf{position.within.between})$ Similar to the case before.

Case $in(\mathsf{dir.up}) \wedge in(\mathsf{position.within.nearby.approach})$ It has to be shown that the compound transition with target state $\mathsf{position.within.onfloor}$ can be taken to satisfy the leadsto-formula in Equation 9.14. We distinguish whether there is a request on the floor the lift cabin is approaching, that is whether $request.\,same$ becomes valid when taking this compound transition. If it becomes valid, then the state $\mathsf{dir.stopped}$ can become active, too. If it does not become valid, then there has to be a request above this position. Hence the invariants of $\mathsf{dir.up}$ remain valid in this case and $\mathsf{dir.up}$ can remain active. This implies that $\mathsf{cabin}$ can remain active, too.

Case $in(\mathsf{dir.up}) \wedge in(\mathsf{position.within.onfloor})$ From the invariants of $\mathsf{dir.up}$ we conclude $request.\,up \wedge \neg(request.\,same \wedge early.\,request)$. Hence it is possible to take a step in which $\mathsf{dir.up}$ remains active and $\mathsf{position.within.onfloor}$ is left. This is the step used to satisfy the leadsto-formula in Equation 9.13.

From these cases we conclude that in each $Execute.\,step$-step of the statechart the state $\mathsf{cabin}$ can remain active and that its liveness-constraints and those of its substates can be satisfied.

$\mathsf{door}_f$ It is clear that each of the states $\mathsf{door}_f$ can become active initially. An active state $\mathsf{door}_f\mathsf{.opened}$ can be left in each step. Hence the fairness-constraint of $\mathsf{door}_f$ can be satisfied. On the other hand, if a state $\mathsf{door}_f\mathsf{.closed}$ is active, it can remain active in each step. If there is a request on a floor $f$, which has been entered early enough, then the state $\mathsf{dir.stopped}$ is active, too. Hence the compound transition to $\mathsf{door}_f\mathsf{.opened}$ can be taken. This implies that the leadsto-formula in Equation 9.15 can be satisfied, too.

It remains to show that the door on a floor can open and close "fast enough" as described in the case $in(\mathsf{dir.stopped}) \wedge in(\mathsf{position.within.onfloor})$ before. "Fast enough" means that a door can open in the same step in which the lift cabin reaches the floor and the cabin is stopped and that the door can close in the next step. Because it depends on the invariants of $\mathsf{door}_f\mathsf{.opened}$ this state can become active in the same step in which the invariants become valid. There is no event expression on the transition segment leaving $\mathsf{door}_f\mathsf{.opened}$, hence the door can be closed in the next step after opening it.

This concludes the proof that REQ is consistent. We have shown that the root-state of req can remain active if a step of the statechart is taken. We have shown also that it is possible to take steps such that the liveness-constraints of REQ are satisfied, too. Remember that actually a stronger property than consistency has been shown because no assumptions on the behaviour of environment have been made.

### 9.8.3 Consistency and Receptiveness

By the example at the beginning of Section 9.8 it becomes clear that consistency is not a strong property. If assumptions on the environment can be made then it is often easy to show that a run satisfying the description exists. This can be seen as cooperation of the environment with the plant.

In the examples used here one possible assumption on the behaviour of the environment is that the lift cabin is not requested at all. In this case it is possible to enter an initial status of the plant without leaving it again.

Most often, the assumption that the environment cooperates with the plant is not justified. Instead of consistency, one would like to show that for each run of the environment there is a run of the plant, that is one would like to show receptiveness.

Note that the descriptions of the behaviour both of the environment and the plant can be written using the combination of statecharts and TLA. This has been done already in the descriptions used here. In the case of the lift system the description of the environment was quite simple. It describes that in each step an arbitrary subset of the events $\mathsf{press}_f$ can be generated.

The proofs in Sections 9.8.1 and 9.8.2 contain no assumptions on the occurrence of these events. Hence we have shown actually that for each run of the environment there is one of the plant satisfying the temporal constraints, that is we have shown that these descriptions are receptive.

Alternating-Time Temporal Logic [Alur *et al.*, 1998] can be used to express and show properties of systems, in which the environment plays against the plant, but assumptions justified by the description of the environment can be used to show properties of the plant. But it is beyond the scope of this dissertation to adapt this reasoning to our combination of statecharts and TLA.

## 9.9 Refining Statecharts

In the preceding sections of this chapter we concentrated on consistency. In this section we consider refinement. A refinement relation between constrained statecharts can be defined via the TLA-formulae defining their semantics, cf. Section 8.6. The proofrules of TLA can be used to show that one constrained statechart is a refinement of another one.

When developing the other descriptions from RW-O of a system, RW-O has been

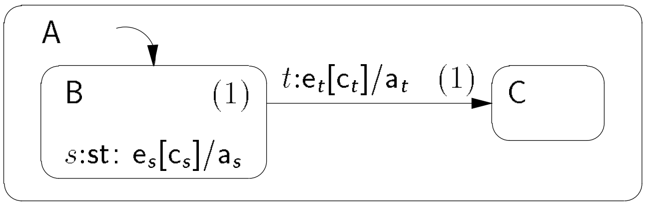



Figure 9.8: Non-deterministic statechart

extended or changed several times in the same way. Such kinds of extensions or changes are presented next. It is argued that extending or changing a statechart correspondingly results in a refinement of the original statechart and that the resulting statechart is consistent. Most of the resulting statecharts are already consistent because they do not contain temporal constraints.

These extensions and changes are applied to statecharts or parts of them without temporal constraints. Therefore they can be applied mainly to the descriptions which are developed at first and which do not contain many temporal constraints. The extensions and changes cannot be used to show that a constrained statechart is refined by one with fewer or no temporal constraints. Such refinements still have to be proven explicitly.

We present these extensions and changes by applying them to the statechart in Figure 9.8. The state B has a static transition $s$ and there is a transition segment $t$ from B to C. Both transition segments have the same structural and explicit priority. If the state B is active and the event expressions $\mathsf{e}_s[\mathsf{c}_s]$ and $\mathsf{e}_t[\mathsf{c}_t]$ are valid at a status, then there is a non-deterministic choice between the compound transitions $\{s\}$ and $\{t\}$. We assume that this statechart is part of another one. Therefore it is possible that a step of the statechart is taken at a status but that the transition segments $s$ and $t$ are not enabled.

### 9.9.1 Invariants

To force that the state B is left, an invariant $\Box p$ can be associated with this state. This is similar to the definition of modes by invariants as shown in Section 9.6.2 and will not be explained further. It has to be possible that if the state A becomes active then it must also be possible that the state B becomes active. This means that the invariant becomes valid if A is entered. Vice versa, if $p$ is no longer valid then the compound transition $\{t\}$ has to be enabled, too, cf. Section 9.6.1.

### 9.9.2 Fairness

Similar to invariants as shown in Section 9.9.1 it is possible to associate a fairness-formulae with the statecharts-state B. The fairness-formulae shall express that the

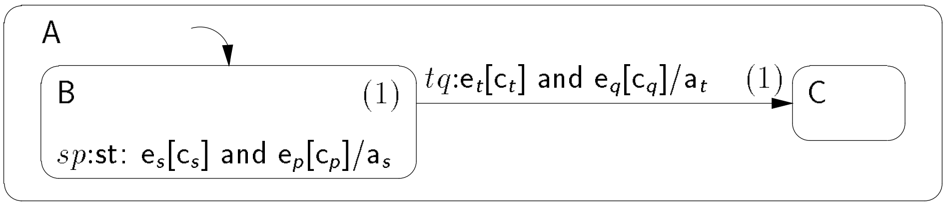



Figure 9.9: Strengthening event expressions

compound transition $\{t\}$ will be taken eventually. In the case that this compound transition is not enabled at every status, this is expressed by strong fairness on the following TLA-action $\mathcal{N}_t$. Remember that $trigger_t$ is valid at a status if the event expression of $t$ is valid, cf. Section 5.1.

$$\mathcal{N}_t \equiv \begin{array}{l} \wedge\ in(\mathsf{B}) \\ \wedge\ trigger_t \\ \wedge\ in'(\mathsf{C}) \end{array}$$

The TLA-formula $SF_{step.taken}(\mathcal{N}_t)$ is associated with the state B. It should be clear that this statechart is a refinement of the one shown in Figure 9.8. Whether it is consistent depends on whether $\mathcal{N}_t$ is a subaction of the larger statechart of which this one is a part of. Assume, that A is below the scope of a compound transition which has to be taken and which is executable whenever $t$ is enabled, then the TLA-action $\mathcal{N}_t$ can be enabled infinitely often without being taken. If this is the case then the refined statechart can be inconsistent.

### 9.9.3 Strengthening Event Expressions

To restrict whether the compound transitions $\{s\}$ and $\{t\}$ can be taken their event expressions can be strengthened. This can be done as shown in Figure 9.9. The event expressions of $s$ and $t$ are strengthened by $\mathsf{e}_p[\mathsf{c}_p]$ and $\mathsf{e}_q[\mathsf{c}_q]$, respectively. The transition segments with the strengthened event expressions will be refered to by $sp$ and $tq$, respectively.

At least one of the compound transitions still has to be enabled if this was the case before strengthening the event expressions:

$$\begin{array}{l} \wedge\ in(\mathsf{A}) \\ \wedge\ \vee\ trigger_s \\ \quad\ \vee\ trigger_t \\ \Rightarrow \vee\ trigger_{sp} \\ \quad\ \vee\ trigger_{tq} \end{array} \tag{9.16}$$

It is necessary that the preceding TLA-formula holds if executing the statecharts-action $\mathsf{a}_s$ makes it visible that the transition segment $s$ has been taken. Assume, that none of the compound transitions $\{sp\}$ and $\{tq\}$ is enabled at a status and that

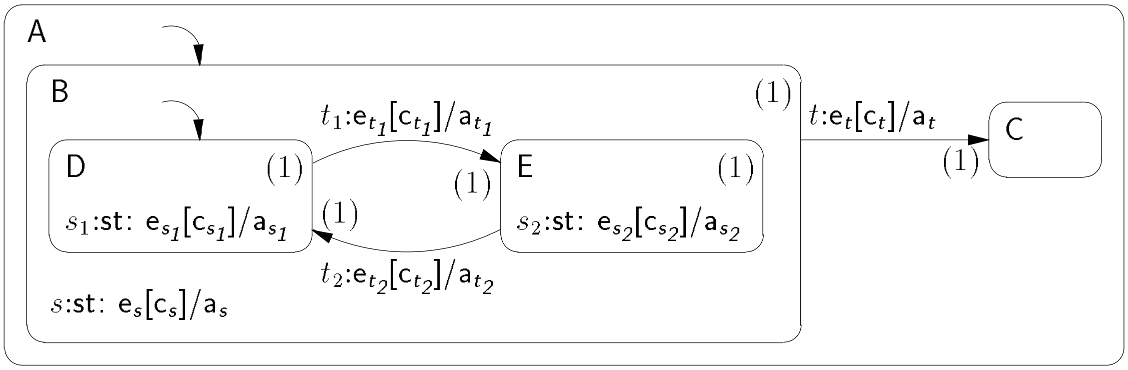



Figure 9.10: Refining a statecharts-state

a step is taken. Then the state B remains active, but $a_s$ is not executed. Such a step was not possible before strengthening the event expressions.

Equation 9.16 ensures that no new runs result from strengthening the event expressions. Hence this statechart is a refinement of the one shown in Figure 9.8 if Equation 9.16 holds. Because no temporal constraints are associated with it, it is consistent.

### 9.9.4 Changing Priorities

In Section 9.9.1 an invariant has been used to ensure that the state B is left if the invariant is no longer valid. On the other hand, by changing priorities it can be achieved that the compound transition $\{t\}$ is taken as soon as it is executable: The priority number of $s$ has to be changed to 0, or alternatively that of $t$ to 2. This implies, that if $\{s\}$ and $\{t\}$ are enabled at the same status then $\{t\}$ will be taken. If only $\{s\}$ is enabled, then $\{s\}$ will be taken as before. Hence this statechart is a refinement of the statechart in Figure 9.8.

Vice versa, if the priority number of $t$ is changed to 0, or that of $s$ to 2, then the state B will be left only, if $\{t\}$ is enabled and $\{s\}$ is not enabled at a status. Note that if $\{s\}$ is not enabled then B does not need to be left.

### 9.9.5 Refining Statecharts-States

In addition to associating temporal constraints with the state B this state can be refined itself by further substates. An example is shown in Figure 9.10 refining Figure 9.9. Whether the compound transitions $\{s\}$ and $\{t\}$ are enabled at a status is not influenced by this change of the statechart. To show that this statechart is a refinement of the statechart in Figure 9.8 it is sufficient to restrict the sets of active states and existing events to those of the one in Figure 9.8.

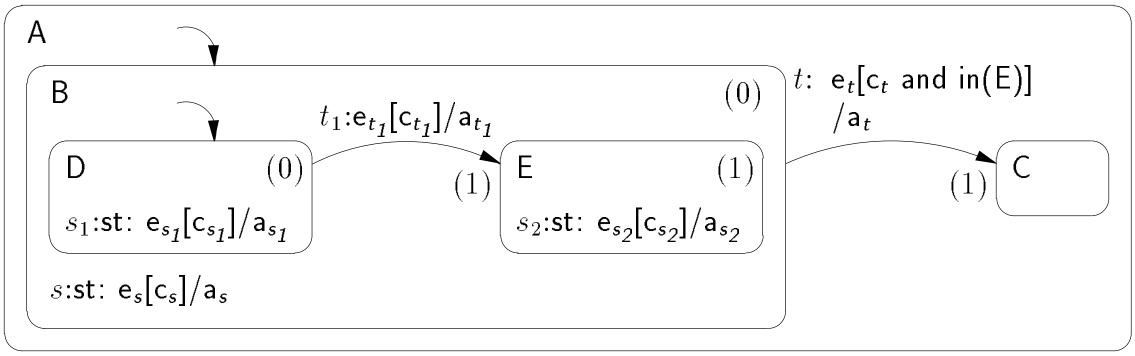



Figure 9.11: Refining a statecharts-state further

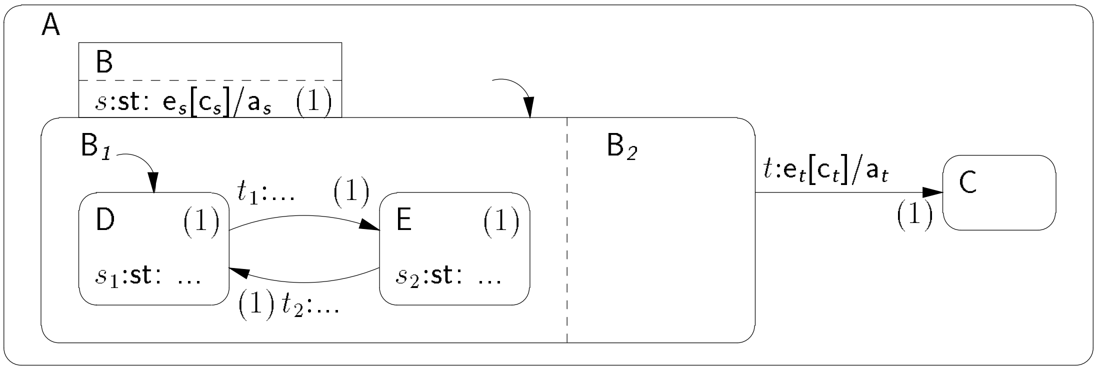



Figure 9.12: Refinement with orthogonal substates

---

The refined statechart can be refined further as shown before. For example, by changing the priority numbers of the static transitions $s$ and $s_1$, strengthening the event expression of $t_2$ with false, and strengthening the event expression of $t$ with in(E) we get the statechart shown in Figure 9.11. The transition segment $t_2$ has been left out because it is never enabled due to its condition expression $e_{t_2}[c_{t_2}]$ and false. This statechart describes that after the state B becomes active, it remains active for at least one further step and that it is left as soon as possible. The condition in Equation 9.16 on strengthening event expressions has to be respected.

### 9.9.6 Adding Orthogonal Statecharts-States

A statechart can also be refined by adding orthogonal substates. In Figure 9.12 we give an example of a refinement of the statechart shown in Figure 9.10. The state B becomes an and-state with two substates $B_1$ and $B_2$. The states below B in the statechart in Figure 9.10 become substates of $B_1$. The static transition $s$ remains a static transition of B. The substate $B_2$ of B can be refined itself by substates, static transitions, and associated temporal constraints.

Because $s$ remains a static transition of $\mathsf{B}$ it is still enabled at the same statuses as before refining the statechart. Vice versa, it is not possible that by enabled compound transitions in $\mathsf{B}_2$ further steps in which $\mathsf{B}$ remains active can be taken although $\{s\}$ is not enabled. At such a status the compound transition $\{t\}$ has to be taken if it is enabled. Again, restricted to the states of the statechart in Figure 9.10 no step can be taken in the refined statechart which cannot be taken in the statechart in Figure 9.10.

## 9.10 Summary

In this chapter we considered properties of descriptions written in the combination of statecharts and TLA. The properties considered did not depend on the concrete system described. We concentrated here on consistency of descriptions.

At the beginning of the chapter we investigated the more technical questions how consistency is related to machine closedness and whether it is decidable whether a constrained statechart is consistent. We derived a sufficient condition to guarantee that a constrained statechart is machine closed. For constrained statecharts with variables over infinite domains we showed that it is not decidable whether such a statechart is consistent. On the other hand, we showed that if only variables over finite domains are used, then it is decidable whether a constrained statechart is consistent.

Thereafter we showed that the sufficient condition for machine closedness implies also that a constrained statechart is consistent. To show that a constrained statechart is consistent it is sufficient to show that if a step of the statechart without its temporal constraints is enabled, then a step of the constrained statechart is enabled, too. This has to be shown for all reachable statuses. To support such proofs we showed

- how the set of reachable statuses can be described more precisely by invariants
- how temporal constraints can be generated from statecharts to describe the possible steps of the statechart more precisely
- how causal dependencies between variables, events, and states of a statechart can be recognizee and used to structure the proofs and to determine with which state a temporal constraint should be associated to obtain an understandable description

In addition we presented patterns of statecharts and conditions on instances of these patterns. These conditions are sufficient to guarantee that the instances are consistent.

Up to this point we took into account temporal constraints which describe only safety properties. But most constrained statecharts contain also temporal constraints which describe liveness properties. We showed for fairness- and leadsto-formulae which steps of a constrained statechart has to be enabled such that a constrained statechart with this kind of temporal constraints is consistent.

For two descriptions of the lift system we provided exemplary proofs that these descriptions are consistent. No assumptions on the environment have been made in these proofs. Therefore, a stronger property than consistency has been proven, namely receptiveness.

At the end of the chapter we considered refinement between descriptions instead of consistency of single descriptions. We showed how to change statecharts without temporal constraints by strengthening event expressions, extending the set of states, and associating temporal constraints with states. We presented conditions which guarantee that the resulting statecharts are refinements of the original ones.

# Chapter 10

# Concluding Remarks

In this chapter we give a summary of the work presented before. We review whether the aims of combining formal description techniques have been achieved, which further questions arise, and whether this approach can be applied to other formal description techniques than the ones used here.

## 10.1 What Has Been Done?

In this work two formal description techniques following different description styles have been combined. One aim was to make available the respective advantages of the description techniques in different descriptions. This was seen mainly w.r.t. the way how the behaviour of a system can be described. A second aim was that by using one formal description technique, namely the combined one, a smooth transition between the different descriptions should be possible.

The formal description techniques statecharts and TLA have been combined. Both are suitable to describe the behaviour of reactive systems. Statecharts have been chosen as a representative of the class of model-oriented description techniques. Statecharts provide a graphical syntax and allow to visualise successions of situations. By the hierarchy of states and the possibility to describe concurrent parts of the system it becomes possible to describe even complex behaviours in a compact way.

TLA has been chosen as a representative of the class of property-oriented description techniques. Its central concept of an action conforms to steps of a statechart. Also, it is possible in TLA to describe properties of the behaviour of a system, for example as invariants or by using fairness-formulae. By using TLA, foundations for considering refinement are provided.

The combination of both formal description techniques is based on constructing an equivalent TLA-program for each statechart. Equivalent means, that a sequence of statuses of the statechart can be seen as a model satisfying the TLA-program, and vice versa models of the TLA-program can be seen as sequences of statuses of the statechart. A substantial amount of the formal description technique statecharts has been covered by this construction.

The syntax of statecharts has been extended by so-called static transitions. The static transitions allow to stay in a statecharts-state, even if there is an executable compound transition leaving the state.

The syntax of statecharts and TLA have been combined by associating one or more TLA-formulae, called temporal constraints, with states of a statechart. It is intended that the temporal constraints are valid as long as the corresponding statecharts-state is active. Fairness-constraints are valid correspondingly if the corresponding state is active infinitely often. The semantics of temporal constraints has been defined by translating them into TLA-formulae, too. The translated TLA-formulae have been conjoined to the TLA-program constructed from the statechart alone. Thereby it has been achieved, that the behaviour of a statechart is only restricted by adding temporal constraints, no new behaviours are created.

A second approach to combine statecharts and TLA consisted of using TLA-actions as enabling conditions of transition segments of statecharts. This influenced how steps of a statechart are built from compound transitions and lead to descriptions which were difficult to understand. Therefore this approach has not been pursued further.

By using the combination of statecharts and TLA several descriptions of a simple lift system have been given. The starting point has been the description RW-O of the behaviour of the uncontrolled plant. This description has been non-deterministic to a large extend and has described sequences of states which shall be avoided by a controller of the plant. The transition segments in this description have been supplemented by events in the description RW-M to describe how a controller is able to influence the plant and how changes of the state of the plant are made visible to a controller. By refining states of the statechart and by associating temporal constraints with statechart-states the effects on the plant caused by events generated by a controller have been described in more detail in RW-M. The description RW-O has been extended also by associating temporal constraints with states to describe the intended behaviour, i.e. the requirements REQ, of the lift system.

Three versions of a controller of the lift system have been given. In each of them the description of the uncontrolled plant could be used again, but in a modified form, as the internal model of the controller of the state of the plant. In the first of the three versions almost pure statecharts has been used. In the second one temporal constraints have been associated with a statechart such that this description can be seen as a synchronous program. The third version consists of a description of the control logic of the controller by one TLA-formula without the use of statecharts.

A specific temporal property, namely synchronisation of steps of concurrent parts of a system, did not occur in the descriptions of the lift system. How this property can be described using the combination of statecharts and TLA has been shown by describing a small part of the well-known case study "production cell". Descriptions of the behaviour of (a part of) the uncontrolled production cell and the requirements on the behaviour have been given.

A formal semantics of activity-charts has been defined similar to that of statecharts by constructing an equivalent TLA-program. Activity-charts are of interest because they allow to describe the interface of a part of a description. Activity-charts have been combined with TLA by associating temporal constraints with activities, this is again similar to the combination of statecharts and TLA. Respecting the interface provided by activities, a refinement relation on activities has been defined. The lift system has been reconsidered by using activities to partition its descriptions.

It is an essential problem whether a constrained statechart is consistent. It has been shown that it is undecidable whether a constrained statechart is consistent. A sufficient condition has been presented to guarantee that a constrained statechart is consistent. In addition, some (simple) patterns of consistent constrained statecharts have been identified. We presented heuristics to make explicit causal dependencies between entities of a constrained statechart. The causal dependencies can be used to support the proof that a constrained statechart is consistent as well as to write more comprehensible descriptions. We provided exemplary proofs that two of the descriptions of the lift system are consistent. Finally we showed how statecharts without temporal constraints can be changed or extended such that the resulting statecharts are consistent and are refinements of the original ones.

## 10.2 Comprehensibility of Descriptions

One of the aims of combining statecharts and TLA was to be able to write comprehensible descriptions of the behaviour of reactive systems. It should be possible to express properties of a behaviour by the most adequate description techniques. We present here our experience gained when writing the example descriptions in Section 7.

Note that our experience does not allow to give a definitive answer on the question whether the combination of statecharts and TLA is a better description technique than statecharts and TLA on their own. "Better" can be seen for example w.r.t. the effort necessary to write descriptions as well as w.r.t. comprehensibility for different kinds of readers of the descriptions. Comprehensibility itself is difficult to measure. One approach to measure comprehensibility can be found in [Finney and Fedorec, 1996]. There, the number of correctly answered questions on a given description are used as a measure. It was beyond the scope of this work to conduct similar empirical studies.

**Using Different Means to Describe Behaviour** It has been possible to write different kinds of descriptions using the combination of statecharts and TLA. These descriptions covered the range from RW-O and REQ as more abstract descriptions to the design of a controller as a more concrete description. In two of the three versions of the design of the controller both formal description techniques have been used in their pure forms. It has been possible to express single parts in different formalisms. In the other descriptions both formal description techniques have been used in a much more mixed form. The requirements have been expressed explicitly by temporal constraints. Mainly invariants and fairness-formulae have been used here. But temporal constraints have also been used in the other descriptions. Especially in RW-M the reactions of the plant on commands from a controller have been described partially by temporal constraints.

We summarise next for some typical properties how they can be expressed. This is independent of the mode — indicative or optative — of the description in which the properties are described.

- The modes of a system and the possible transitions between them can be expressed by statecharts. To express that a mode, i.e. the statecharts-state corresponding to it, can remain active, it is possible to use static transitions. The hierarchy of statecharts-states can be used to express a hierarchy of modes.

- The intended meaning of a mode can be expressed by invariants which are associated with the corresponding state. These invariants can refer to variables as well as to other modes.

- Transitions between the statecharts-states corresponding to modes can be restricted by suitable event expressions of the transition segments.

- The behaviour of a system in individual modes can be expressed by temporal constraints associated with the corresponding statecharts-states. It is also possible to express this behaviour by states and transition segments below these states. A third alternative is to express this for each behaviour specific to a mode in a separate activity.

- Fairness of individual steps of a statechart can be expressed by fairness-formulae on the corresponding TLA-actions.

**Using Several Description Techniques** Obviously, when writing or reading descriptions written in the combination of statecharts and TLA both formal description techniques have to be mastered. To avoid adding further difficulties the semantics of the single formal description techniques has been embedded as faithful as possible in the combination. Also, it is obvious, that besides the single formal description techniques their interplay has to be understood. Therefore it has been important that the use of TLA does not interfere with the construction of steps in statecharts.

**Instantaneous versus Delayed Communication** By associating invariants with statecharts-states it becomes possible that information is communicated instantaneously. This means that changes in one part of a statechart become visible and relevant to other parts of the statechart in the same step. On the other hand the effects of an ordinary step of a statechart cannot influence the step itself.

Surely, it does not contribute in a positive way to comprehensibility of descriptions if both kinds of communicating information are intertwined. From the examples presented here only few hints can be derived on when to use which kind of communication. Communicating instantaneously seems to be useful in restricted parts of a statechart. In this case chains of causal dependencies do not become too long. Long chains should be avoided because otherwise changes in one part of a statechart can affect a large part of the rest of the statechart instantaneously and such large parts can be difficult to grasp. At second, the usage of both kinds of communication should be meaningful in the system being described. Instantaneous communication can be used if communication delays are neglectable in the system. Whether this is the case depends on the granularity in which time is regarded.

**Invariants and Causality** In contrast to action expressions of statecharts it does not become clear which of the entities occurring in an invariant can be considered as being read and which one as being written, and which entity depends on which other ones. Note that there is also a causal dependency between the entities occurring in an invariant and the state with which the invariant is associated. To make this causal dependency more explicit for this case we provided a simple syntactical means to distinguish between the directions of the causal dependencies. Other causal dependencies can be derived by using heuristics starting from the interface provided by activities.

The causal dependencies can be used to define a relation on entities. If this is an irreflexive ordering, this contributes positively to comprehensibility because in this case there is a clear flow of information in the description.

**Consistency** In contrast to pure statecharts it became possible by associating temporal constraints with statecharts-states to describe empty behaviours or to exclude specific sequences of inputs. This problem decreases comprehensibility because it can happen quite easily and without being recognized when writing descriptions. This is especially the case if there are not "contradictions" among the temporal constraints used to describe properties explicitly, but between the temporal constraints and the semantics of the statechart.

Note that this problem occurs already when conjoining an invariant $\Box p$ to a TLA-program $\mathit{init} \wedge \Box[\mathcal{N}]_f \wedge WF_f(\mathcal{M}) \ldots$. Hence it is not specific to the combination of statecharts and TLA considered here.

Clear causal dependencies as described before reduce this problem because it imposes an order on the parts of the statechart. This can be used to check the parts one after another whether they are consistent.

Some simple patterns of statecharts with temporal constraints have been presented which are consistent under certain conditions. More patterns of this kind which are relevant in practice have to be derived. This allows to write consistent and receptive descriptions without having to show this explicitly.

## 10.3 Relations between Descriptions

A further aim was to allow a smooth transition between different descriptions. The description Rw-o has been extended or changed several times in a similar way to obtain other descriptions. These extensions and changes guarantee that the resulting statecharts are refinements of the original ones. On the other hand, when considering refinement between two arbitrary statecharts there is the problem that it is visible that steps of a statechart are taken. Therefore stuttering steps have to be described explicitly in the statechart if they are necessary. These topics will be considered in more detail subsequently.

**Reuse of Descriptions** In the examples different descriptions corresponding to [Zave and Jackson, 1997] have been given. The description Rw-o has been central to these descriptions. The entities which are relevant to describe the behaviour of a reactive system are defined in Rw-o. Especially, the modes of the system which can be distinguished by an external observer are defined. Examples of such modes are whether the doors to the lift cabin on the floors are open or whether the devices of the production cell are in designated positions. In addition the possible transitions between these modes have been given.

It is typical in Rw-o that non-determinism between changing modes and staying in a mode is expressed by static transitions. This non-determinism is restricted only as far as it corresponds to the behaviour of the uncontrolled plant. In the lift system this means for example that the doors on the floors close automatically or that the lift cabin is able to stop exactly on a floor but not between floors.

These descriptions have been refined in two ways. On the one hand the requirements on the behaviour of the system have been described. The description Req refers to the terms visible to an external observer, but it is also possible to introduce auxiliary terms. This can be done by adding auxiliary variables as well as introducing orthogonal statecharts-states. In a second extension of Rw-o, which is independent of Req, the non-determinism is restricted by adding event expressions to the transition segments which correspond to commands of a controller. Corresponding changes of the status are indicated to the controller by corresponding signals. The commands send by a controller and the signals to the controller are described by statecharts-events.

In the combination of both formal description techniques a large variety of means to describe properties is available. This allows to describe the individual extensions of RW-O adequately without changing RW-O substantially. This means, that consistency — in the sense of that a term has the same meaning in different descriptions — is preserved.

Starting from both refinements of RW-O the specification SPEC of a controller can be derived. This means in general that the requirements have to be rewritten in terms visible to a controller.

The description RW-M can be used as a foundation of the part of a controller which checks whether the plant behaves as expected. A corresponding part of the controller was described in the lift example. But it has not been investigated, how this description can be incorporated systematically into a design of a controller, which changes have to be made to it and how this is supported by the combination of statecharts and TLA.

**Recurring Extensions and Changes** When refining RW-O to RW-M and REQ it has been extended or changed several times in a similar way. Applying such extensions and changes to an unconstrained statechart implies that the resulting statechart actually is a refinement of the original ones and that it is consistent.

Using these extensions and changes simplified the transition from RW-O to the other descriptions. On the one hand guarantees on the relation between the descriptions are given. On the other hand the patterns allow to concentrate on the more complex properties.

**Refinement** TLA provides a profound body of work w.r.t. refinement between descriptions. It is reviewed here to which extend this can be utilised to relate statecharts via their equivalent TLA-formulae.

Let $SC_{c(oncrete)}$ and $SC_{a(bstract)}$ be two statecharts. Then $SC_c$ is a refinement of $SC_a$ if every run of $SC_c$ can be interpreted as a run of $SC_a$. This means that from every status of a run of $SC_c$ a status of $SC_a$ can be constructed. In particular, the sets of active states and existing events and the data variables have to be related to each other. This has to be done for the variable *step.taken*, used to indicate steps of the statechart, too. Note that in principle it is possible that $SC_c$ takes several steps, whereas $SC_a$ takes only one.

It is difficult to divide a step of $SC_c$ into several ones of $SC_a$. Between two steps of $SC_c$ there can be external steps. If the set of existing events is extended in such a step, these new events are discarded when taking the next step of $SC_c$. Hence it becomes visible, whether $SC_c$ takes steps. Therefore the more abstract statechart $SC_a$ has to take a step, too. That $SC_a$ can take further steps can be achieved by adding appropriate static transitions in $SC_a$. To add such static transitions corresponds to describe stuttering steps of $SC_a$ explicitly. These additional static transitions do not contribute to comprehensibility of the statechart, because on the one hand it has to be described also that not only these static transitions are taken. On the other hand static transitions are used now for two different purposes: For describing non-determinism and for describing stuttering steps. It cannot be distinguished by a reader of a description which is the intended purpose.

Instead of describing the stuttering steps explicitly it can be tried to incorporate them into the definition of the semantics. For example, by allowing steps of the environment which can change the set of existing events arbitrarily instead of only extending this set. Then for any change of the set events of the concrete statechart $SC_c$ a corresponding change of $SC_a$ is possible, too. But this seems to be be too far away from the usual semantics of statecharts to be useful. Two other possibilities are to use a super-step or a micro-step semantics instead of the single-step semantics used here.

A super-step semantics can be achieved by requiring that steps of the environment can be taken only if some conditions are satisfied. An example of such a condition is that no compound transitions of the statechart are enabled. Hence no sequence of more concrete steps can be interrupted by steps of the environment. It remains as a problem to produce the "results" of this sequence of steps in the last step of this sequence. This has to be done in the last step of such a sequence to ensure that events are not discarded. Also, the main problem of a super-step semantics remains. Namely that the sequences of steps of $SC_c$ do not need to terminate.

It is not necessary to introduce a micro-step semantics explicitly. In one variant of the design of a controller of the lift, it has been shown how some kind of micro-step semantics can be achieved already by associating invariants with statecharts-states. In this case it is necessary to obtain a clear flow of information corresponding to the causal dependencies. A step, more precisely a compound transition, of $SC_a$ can be refined by several compound transitions in orthogonal parts of $SC_c$. These can be taken together in one step, hence they cannot be interrupted by steps of the environment. In addition it is not necessary to pay attention that the results are produced in a "last" step as in a super-step semantics.

It was not possible to use refinement as in TLA directly to relate two statecharts, because by discarding events it becomes visible that a step is taken. In the combination of statecharts and TLA presented here this problem can be eliminated by adding stuttering steps explicitly in the more abstract statechart and by using TLA-formulae to describe synchronous execution of non-conflicting compound transitions of the more concrete statecharts.

## 10.4 Transfer to Other Formal Description Techniques

The work presented here has been based on the combination of statecharts and TLA. Of course there is the question, whether such a combination can be useful for other formal description techniques. In the following we sketch for a combination of SDL [CCITT, 1993] with a property-oriented description technique which properties should be expressible and which problems we expect to occur when carrying out the combination. It is assumed in the following that the reader has knowledge of SDL.

It has to be clarified at first, which kinds of non-determinism is provided by the formal description techniques to be combined. SDL provides mainly the following kinds of non-determinism:

- The different interleavings between the different processes of an SDL-system.
- Within processes there is non-determinism for example by using `any` in decisions, by using `any` to denote an arbitrary value of a data-type, and by continuous signals which are valid simultaneously.
- There can be arbitrary delays when sending signals over channels.

It does not seem to be very promising to restrict the non-determinism within SDL-processes. Especially when using `any` in decisions and when using continuous signals the corresponding predicates can be strengthened already. It seems to be still useful to express fairness between different transitions in an SDL-process.

We consider it as more interesting to restrict the possible interleavings and the delays on the channels. This is motivated by presenting some important properties of a system for discrete simulation of continuous processes [Deiß and Hillenbrand, 1997]. This system is partitioned into some specialised SDL-processes for managing simulation events, for maintaining a specific relation between simulation and real-time, and for controlling the simulator itself. In addition, there is a large number of SDL-processes, called simulation objects. Each of them describes the behaviour of one physical object. The simulation objects can be active, that is they perform some part of the numerical simulation at a specific point of simulation time, or passive, that is they wait until they have to compute again. Some typical properties of this system are:

- If a simulation object is scheduled for some point of simulation time, it will become active before or exactly at this point of time.
- The distance in simulation time between two activations of a simulation object can be bounded from above for each simulation object.
- When the clock process advances simulation time, all simulation objects are passive.
- When a simulation object is computing, then the current simulation time is available to it.
- Each request for the value of a variable of a simulation object at one point of simulation time returns the same value.

These properties refer to internal entities of the SDL-processes — the value of the simulation time the SDL-processes know of and whether they are active or passive — as well as to the signals exchanged between the SDL-processes. The internal entities of the SDL-processes must be made visible to be able to refer to them in temporal constraints.

The properties refering to the signals are more precisely properties of *traces*, that is of sequences of signals. Therefore it has to be possible to describe properties of traces by the property-oriented description technique. In this context it has to be noted that most often SDL is used together with message sequence charts (MSC). Typically, message sequence charts are used to express communication patterns between processes, whereas SDL is used to describe the design of a system showing this behaviour. The meaning of a message sequence chart is a set of traces. Therefore we consider it useful to combine the property-oriented description technique with SDL as well as with MSC. This would allow to take over temporal constraints which restrict a message sequence chart to a SDL-description and to replace the temporal constraints successively by a pure SDL-description. To proceed as described it has to be possible to extend an SDL-process by further signals and by corresponding reactions when receiving them. It is an open problem how such an extension can be achieved.

## 10.5 Future Work

Based on the combination of statecharts and TLA and the examples described using this combination we identify the following areas in which more work is necessary.

**Using and Developing Descriptions** It has been possible to combine the formal description techniques and to use them in a meaningful way to describe one important aspect of reactive systems, namely behaviour. It has also been possible to describe properties of the behaviour adequately by using the combination of statecharts and TLA. Our experience that the combination is meaningful and the combined notation is useful, has to be supported by further empirical evidence.

On the one hand it has to be shown that a description written in the combination is more comprehensible than descriptions written in statecharts without temporal constraints, in TLA, or in another description technique. Comprehensibility can be measured for example by asking whether the behaviour described has specific properties and by counting the number of correct answers.

On the other hand, further and larger case studies have to be conducted to evaluate the combination of statecharts and TLA on a more solid basis. It is necessary to conduct larger case studies to determine whether the combination scales up or whether it can be used only for small systems. We expect that the combination scales up because the structuring mechanisms of activity-charts can be used.

To develop descriptions with reasonable effort more guidelines and methods specific to the combination of statecharts and TLA have to be provided. Among others, these guidelines should state which kinds of properties are to be expressed by statecharts and which ones by temporal constraints. Furthermore, the development of descriptions has to be supported by tools. It seems to be reasonable to adapt existing tools for developing statecharts and activity-charts to handle our extensions of statecharts and temporal constraints.

The description RW-O has been used as a basis for most of the other descriptions. Hence, the terms defined in RW-O can be used in these other descriptions with the same meaning. We expect that this common terminology provides a good starting point to define a traceability relation between descriptions. It has to be investigated how a traceability relation can be defined and whether this is facilitated by this common terminology.

**Relation to Synchronous Languages** By the use of TLA, especially by the use of invariants, aspects of synchronous programming languages have been introduced to the combination of the formal description techniques. To unify whether information is communicated with or without delay it can be helpful to combine variants of statecharts which support the synchrony hypothesis with a property-oriented formal description technique.

**Formal Analysis** As a major problem it remains the question whether a description is consistent. Approaches to write consistent descriptions and to show that descriptions are consistent have been presented, but these are not sufficient. They have to be extended in three directions. At first, the approaches to show consistency have to be developed further themselves. At second, approaches to show that descriptions are receptive, that they have properties specific to the system described, and that one description is a refinement of another one, have to developed. At third, there exist tools to show properties of descriptions written completely in statecharts or in TLA. It has to be investigated how these tools can be used to show properties of descriptions written in the combination of statecharts and TLA. We expect that the tools have to be adapted before they can be used.

# Bibliography


[Abadi and Lamport, 1991] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82:253–284, 1991.

[Abadi and Lamport, 1994] M. Abadi and L. Lamport. An old-fashioned recipe for real time. *ACM Transactions on Programming Languages and Systems*, 16(5):1543–1571, 1994.

[Abadi and Lamport, 1995] M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, 1995.

[Abadi and Manna, 1989] M. Abadi and Z. Manna. Temporal logic programming. *Journal of Symbolic Computation*, 8:277–295, 1989.

[Abadi and Merz, 1996] M. Abadi and S. Merz. On TLA as a logic. In M. Broy, ed., *Deductive Program Design, Proc. of the NATO Advanced Study Institute on Deductive Program Design, 1994*, vol. 152 of *NATO ASI series F*. Springer, 1996.

[Alur *et al.*, 1998] R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-Time Temporal Logic. In W.-P. de Roever, H. Langmaack, and A. Pnueli, eds., *Proc. of Intl. Symp. Compos'97, Compositionality — The Significant Difference*, vol. 1536 of *LNCS*, pages 23–60. Springer, 1998.

[Astesiano and Reggio, 1997] E. Astesiano and G. Reggio. Formalism and method. In M. Bidoit and M. Dauchet, eds., *Proc. of 7th Intl. Joint Conf. CAAP/FASE, Theory and Practice of Software Development, TAPSOFT'97*, vol. 1214 of *LNCS*, pages 93–114. Springer, 1997.

[Bellia and Levi, 1986] M. Bellia and G. Levi. The relation between logic and functional languages: A survey. *The Journal of Logic Programming*, 3:217–236, 1986.

[Benveniste *et al.*, 1994] A. Benveniste, P. Caspi, P. L. Guernic, and N. Halbwachs. Data–flow synchronous languages. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, eds., *Proc. of REX School/Symposium: A Decade of Concurrency: Reflections and Perspectives, 1993*, vol. 803 of *LNCS*, pages 1–45. Springer, 1994.

[Benveniste *et al.*, 1997] A. Benveniste, P. L. Guernic, and P. Aubry. Compositionality in dataflow synchronous languages: Specification and code generation. Publication Interne 1150, IRISA, 1997.

[Bjørner *et al.*, 1997] N. Bjørner, Z. Manna, H. B. Sipma, and T. E. Uribe. Deductive verification of real-time systems using STeP. In M. Bertran and T. Rus, eds., *Transformation-Based Reactive Systems Development, Proc. of 4th Intl. AMAST Workshop on Real-Time Systems and Concurrent and Distributed Systems, ARTS'97*, vol. 1231 of *LNCS*, pages 22–43. Springer, 1997.

[Blom and Jonsson, 1996] J. Blom and B. Jonsson. Constraint oriented temporal logic specification: The Dagstuhl RPC-memory specification problem. In M. Broy, S. Merz, and K. Spies, eds., *Formal Systems Specification, The RPC-Memory Specification Case Study*, vol. 1169 of *LNCS*, pages 161–182. Springer, 1996.

[Broy, 1992] M. Broy. (Inter–)action refinement: The easy way. In *Proc. of Program Design Calculi*, NATO ASI Series. Springer, 1992.

[Brzoska, 1991] C. Brzoska. Temporal logic programming and its relation to constraint logic programming. In V. Saraswat and K. Ueda, eds., *Logic Programming: Proc. of the 1991 Intl. Symp.*, pages 661–677. MIT Press, 1991.

[CCITT, 1993] CCITT. Recommendation Z.100, Specification and Description Language SDL, 1993.

[Chestnut, 1970] H. Chestnut. Information requirements for systems understanding. *IEEE Transactions on Systems Science and Cybernetics*, 6(1):3–12, 1970.

[Courtois and Parnas, 1993] P.-J. Courtois and D. L. Parnas. Documentation for safety critical software. In *Proc. of 15th Intl. Conf. on Software Engineering*, pages 315–323, 1993.

[Damm *et al.*, 1998] W. Damm, B. Josko, H. Hungar, and A. Pnueli. A compositional real-time semantics of STATEMATE designs. In W.-P. de Roever, H. Langmaack, and A. Pnueli, eds., *Proc. of Intl. Symp. Compos'97, Compositionality — The Significant Difference*, vol. 1536 of *LNCS*, pages 186–238. Springer, 1998.

[Day and Joyce, 1993] N. Day and J. J. Joyce. The semantics of statecharts in HOL. In J. J. Joyce and C.-J. H. Seger, eds., *Proc. of 6th Intl. Workshop on Higher Order Logic, Theorem Proving and its Applications*, vol. 780 of *LNCS*, pages 338–351. Springer, 1993.

[Day, 1993] N. Day. A model checker for statecharts (linking case tools with formal methods). Technical report 93-35, Department of Computer Science, Vancouver, B.C. Canada, October 1993.

[Deiß and Hillenbrand, 1997] T. Deiß and T. Hillenbrand. A case study on the use of SDL. SFB 501 Bericht 03/1997, Sonderforschungsbereich 501, Fachbereich Informatik, Universität Kaiserslautern, 1997.

[Deiß *et al.*, 1997] T. Deiß, M. Kronenburg, and D. Zeckzer. A catalogue of criteria for evaluating formal methods and its application. SFB 501 Bericht 04/97, Sonderforschungsbereich 501, Fachbereich Informatik, Universität Kaiserslautern, 1997.

[Deiß, 1996] T. Deiß. Combining a state based formalism with temporal logic. SFB 501 Bericht 05/1996, Sonderforschungsbereich 501, Fachbereich Informatik, Universität Kaiserslautern, 1996.

[Dershowitz and Plaisted, 1985] N. Dershowitz and D. Plaisted. Logic programming cum applicative programming. In *Proc. of IEEE Symp. on Logic Programming*. IEEE Computer Society Press, 1985.

[Dill, 1989] D. L. Dill. *Trace theory for automatic hierarchical verification of speed independent circuits*. ACM distinguished dissertations. MIT Press, 1989.

[Engberg, 1995] U. Engberg. *Reasoning in the Temporal Logic of Actions, The design and implementation of an interactive computer system*. PhD thesis, Department of Computer Science, University of Aarhus, 1995.

[Eschbach *et al.*, 1998] R. Eschbach, T. Deiß, and M. Kronenburg. A framework for the analysis of formal description techniques for timed systems. SFB 501 Bericht 05/98, Sonderforschungsbereich 501, Fachbereich Informatik, Universität Kaiserslautern, 1998.

[Finney and Fedorec, 1996] K. M. Finney and A. M. Fedorec. An empirical study of specification readability. In C. N. Dean and M. G. Hinchey, eds., *Teaching and Learning Formal Methods*, International Series in Formal Methods, chapter 8, pages 117–129. Academic Press, 1996.

[Fischer, 1996] S. Fischer. Real-time Estelle. Reihe Informatik 3/96, Universität Mannheim, Fakultät für Mathematik und Informatik, 1996.

[Glässer and Karges, 1997] U. Glässer and R. Karges. Abstract state machine semantics of SDL. *Journal of Universal Computer Science*, 3(12):1282–1414, 1997.

[Goguen and Burstall, 1992] J. A. Goguen and R. M. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the ACM*, 39(1):95–146, 1992.

[Gordon and Melham, 1993] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[Gurevich, 1995] Y. Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, ed., *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[Halbwachs, 1993] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer, 1993.

[Hanus, 1994] M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19,20:583–628, 1994.

[Harel and Naamad, 1996] D. Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.

[Harel *et al.*, 1990] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. STATEMATE: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, 1990.

[Harel, 1987] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[Heninger, 1980] K. L. Heninger. Specifying software requirements for complex systems: New techniques and their application. *IEEE Transactions on Software Engineering*, 6(1):2–13, 1980.

[Herrmann and Krumm, 1995] P. Herrmann and H. Krumm. Re-usable verification elements for high-speed transfer protocol configurations. In P. Dembinski and M. Sredniawa, eds., *Protocol Specification, Testing and Verification XV*, pages 171–186. Chapman and Hall, 1995.

[Hoare, 1985] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[Huizing and de Roever, 1991] C. Huizing and W. P. de Roever. Introduction to design choices in the semantics of Statecharts. *Information Processing Letters*, 37:205–213, 1991.

[ISO, 1989] I. ISO. Information Processing Systems - Open Systems Interconnection - ESTELLE - A Formal Description technique based on an Extended State Transition Model. ISO/IEC 9074, 1989.

[Jaffar and Maher, 1994] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19,20:503–581, 1994.

[Jeffords and Heitmeyer, 1998] R. Jeffords and C. Heitmeyer. Automatic generation of state invariants from requirements specifications. In *Proc. of 6th Intl. Symp. on Foundations of Software Engineering, FSE-6*, 1998.

[Jourdan *et al.*, 1994] M. Jourdan, F. Lagnier, F. Maraninchi, and P. Raymond. A multiparadigm language for reactive systems. In *IEEE Intl. Conf. on Computer Languages (ICCL), Toulouse, France*, 1994.

[Kavi, 1992] K. M. Kavi, ed. *Real-Time Systems, Abstractions, Languages and Design Methodologies.* IEEE Computer Society Press, 1992.

[Lamport, 1994a] L. Lamport. How to write a long formula. *Formal Aspects of Computing*, 6:580–584, 1994.

[Lamport, 1994b] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

[Lamport, 1995] L. Lamport. TLA in pictures. *IEEE Transactions on Software Engineering*, 21(9):768–775, 1995.

[Lamport, 1996] L. Lamport. Refinement in state-based formalisms. SRC Technical Note 1996-001, digital System Research Center, 1996.

[Lewerentz and Lindner, 1995] C. Lewerentz and T. Lindner, eds. *Formal Development of Reactive Systems, Case Study Production Cell*, vol. 891 of *LNCS*. Springer, 1995.

[Manna and Pnueli, 1992] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems, Specification.* Springer, 1992.

[Manna and Pnueli, 1995] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer, 1995.

[Manna *et al.*, 1994] Z. Manna, A. Anuchitanul, N. Bjørner, A. Browne, E. Chang, M. Colón, L. de Alfaro, H. Devarajan, H. Sipma, and T. Uribe. STeP: the Stanford Temporal Prover. Technical Report stan-cs-tr-94-1518, Department of Computer Science, Stanford University, 1994.

[Maraninchi and Rémond, 1998a] F. Maraninchi and Y. Rémond. Compositionality criteria for defining mixed-styles synchronous languages. In W.-P. de Roever, H. Langmaack, and A. Pnueli, eds., *Proc. of Intl. Symp. Compos'97, Compositionality — The Significant Difference*, vol. 1536 of *LNCS*, pages 424–438. Springer, 1998.

[Maraninchi and Rémond, 1998b] F. Maraninchi and Y. Rémond. Mode-Automata: About modes and states for reactive systems. In C. Hankin, ed., *Proc. of 7th Europ. Symp. on Programming, ESOP'98*, vol. 1381 of *LNCS*, pages 185–199. Springer, 1998.

[Mester and Krumm, 1995] A. Mester and H. Krumm. Composition and refinement based construction of distributed algorithms. In U. H. Engberg, K. G. Larsen, and A. Skou, eds., *Proc. of the Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, number NS-95-2 in BRICS Notes Series, pages 290–303. BRICS, 1995.

[Mester and Mattick, 1996] A. Mester and V. Mattick. Verbindung zwischen Statecharts und TLA zur Spezifikation verteilter Systeme. In U. Herzog and H. Hermanns, eds., *Proc. of 6. GI/ITG Fachgespräch Formale Beschreibungstechniken für verteilte Systeme*, vol. 29 of *Arbeitsberichte des Instituts für Mathematische Maschinen und Datenverarbeitung (Informatik), ISSN 0344-3315*, pages 159–164. Friedrich Alexander Universität Erlangen, Nürnberg, 1996. extended abstract, in german.

[Nazareth *et al.*, 1996] D. Nazareth, F. Regensburger, and P. Scholz. Ministatecharts: A lean version of statecharts. Technical Report TUM-I9610, Technische Universität München, Institut für Informatik, 1996.

[Ostroff, 1989] J. S. Ostroff. *Temporal Logic for Real-Time Systems*. Research Studies Press, 1989.

[Philipps and Scholz, 1997] J. Philipps and P. Scholz. Compositional specification of embedded systems with statecharts. In M. Bidoit and M. Dauchet, eds., *Proc. of 7th Intl. Joint Conf. CAAP/FASE, Theory and Practice of Software Development, TAPSOFT'97*, vol. 1214 of *LNCS*. Springer, 1997.

[Pnueli and Shalev, 1991] A. Pnueli and M. Shalev. What is in a step: On the semantics of statecharts. In T. Ito and A. R. Meyer, eds., *Intl. Conf. TACS'91: Theoretical Aspects of Computer Software*, vol. 526 of *LNCS*, pages 244–264. Springer, 1991.

[Pnueli, 1992] A. Pnueli. System specification and refinement in temporal logic. In R. Shyamasundar, ed., *12th Conf. on Foundations of Software Technology and Theoretical Computer Science*, vol. 652 of *LNCS*, pages 1–38. Springer, 1992.

[Poigné and Holenderski, 1998] A. Poigné and L. Holenderski. On the combination of synchronous languages. In W.-P. de Roever, H. Langmaack, and A. Pnueli, eds., *Proc. of Intl. Symp. Compos'97, Compositionality — The Significant Difference*, vol. 1536 of *LNCS*, pages 490–514. Springer, 1998.

[Prehofer, 1996] C. Prehofer. Some applications of functional-logic programming. In M. M. Chakravarty, Y. Guo, and T. Ida, eds., *Multi-Paradigm Logic Programming, Proc. of the JICSLP'96 Post-Conference Workshop*, number No. 96-28 in Forschungsberichte des Fachbereichs Informatik, pages 35–45. Technische Universität Berlin, Fachbereich 13, Informatik, 1996.

[Prehofer, 1998] C. Prehofer. *Solving higher-order equations: from logic to programming*. Progress in theoretical computer science. Birkhäuser, 1998.

[Rössig, 1994] S. Rössig. *A Transformational Approach to the Design of Communicating Systems*. PhD thesis, Carl von Ossietzky Universität Oldenburg, 1994.

[Schowalter, 1997] A. Schowalter. Modellierung einer Produktionszelle in einer Kombination aus TLA und Statecharts. Projektarbeit, Fachbereich Informatik, Universität Kaiserslautern, 1997. in german.

[Schwarzenbach and Gill, 1978] J. Schwarzenbach and K. F. Gill. *System Modelling and Control*. Edward Arnold, 1978.

[Sowmya and Ramesh, 1998] A. Sowmya and S. Ramesh. Extending statecharts with temporal logic. *IEEE Transactions on Software Engineering*, 24(3):216–231, 1998.

[Spivey, 1992] M. Spivey. *The Z Notation, A Reference Manual*. Prentice Hall, 1992. 2nd edition.

[stm91, 1991] i-Logix, 22 Third Avenue, Burlington, Mass. 01803, USA. *Languages of STATEMATE*, January 1991.

[van der Beeck, 1994] M. van der Beeck. A comparison of statechart variants. In H. Langmaack, W. P. de Roever, and J. Vytopil, eds., *Proc. of 3rd Intl. Symp. Formal Techniques in Real-Time and Fault-Tolerant Systems*, vol. 863 of *LNCS*, pages 128–148. Springer, 1994.

[Weber, 1996] M. Weber. Combining statecharts and Z for the design of safety-critical control systems. In M. C. Gaudel and J. Woodcock, eds., *Proc. of 3rd Intl. Symp. of Formal Methods Europe FME'96*, vol. 1051 of *LNCS*, pages 307–326. Springer, 1996.

[Wing, 1990] J. M. Wing. A specifiers introduction to formal methods. *IEEE Computer*, 23(9):8–24, 1990.

[Zave and Jackson, 1993] P. Zave and M. Jackson. Conjunction as composition. *ACM Transactions on Software Engineering and Methodology*, 2(4):379–411, 1993.

[Zave and Jackson, 1997] P. Zave and M. Jackson. Four dark corners of requirements engineering. *ACM Transactions on Software Engineering and Methodology*, 6(1):1–30, 1997.

[Zave, 1989] P. Zave. A compositional approach to multiparadigm programming. *IEEE Software*, pages 15–25, 1989.

[Zave, 1991] P. Zave. An insider's evaluation of PAISLey. *IEEE Transactions on Software Engineering*, 17(3):212–225, 1991.

# Index

# Appendix A

# Proofs of TLA-Formulae

In this appendix we present the proofs of some auxiliary lemmata used in Section 6.4.2. For the sake of reference we present the proofrules of TLA in Section A.1. Some simple theorems of TLA are proven in Section A.2. In Section A.3 it is explained how the temporal prover STeP was used to prove validity of TLA-formulae. The auxiliary lemmata used to prove properties of the liveness semantics, cf. Section 6.4.2, are shown in Section A.4 together with their proofs.

## A.1 Proofrules of TLA

The proofrules of TLA are shown here for easier reference, they are taken from [Lamport, 1994b, pages 888 and 905].

Let $F$, $G$, $H_c$ be TLA-formulae, $p$, $q$, $i$ be predicates, $\mathcal{A}$, $\mathcal{B}$, $\mathcal{N}$, $\mathcal{M}$ be actions, and $f$, $g$ be state functions.

#### The Rules of Simple Temporal Logic

STL1. $\dfrac{F \text{ provable by propositional logic}}{\Box F}$

STL2. $\dfrac{}{\Box F \Rightarrow F}$

STL3. $\dfrac{}{\Box\Box F \equiv \Box F}$

STL4. $\dfrac{F \Rightarrow G}{\Box F \Rightarrow \Box G}$

STL5. $\dfrac{}{\Box(F \wedge G) \equiv (\Box F) \wedge (\Box G)}$

STL6. $\dfrac{}{(\Diamond\Box F) \wedge (\Diamond\Box G) \equiv \Diamond\Box(F \wedge G)}$

LATTICE. Let $\succ$ be a well-founded partial order on a set $S$

$$\frac{F \wedge (c \in S) \Rightarrow (H_c \leadsto (G \vee \exists d \in S: (c \succ d) \wedge H_d))}{F \Rightarrow ((\exists c \in S: H_c) \leadsto G)}$$

### The Basic Rules of TLA

TLA1. $$\frac{p \wedge (f' = f) \;\Rightarrow\; p'}{\Box p \equiv p \wedge \Box[p \;\Rightarrow\; p']_f}$$

TLA2. $$\frac{p \wedge [\mathcal{A}]_f \;\Rightarrow\; q \wedge [\mathcal{B}]_g}{\Box p \wedge \Box[\mathcal{A}]_f \;\Rightarrow\; \Box q \wedge \Box[\mathcal{B}]_g}$$

### Additional Rules

INV1. $$\frac{i \wedge [\mathcal{N}]_f \;\Rightarrow\; i'}{i \wedge \Box[\mathcal{N}]_f \;\Rightarrow\; \Box i}$$

INV2. $$\frac{}{\Box i \;\Rightarrow\; (\Box[\mathcal{N}]_f \equiv \Box[\mathcal{N} \wedge i \wedge i']_f)}$$

WF1. $$\frac{\begin{array}{l} p \wedge [\mathcal{N}]_f \;\Rightarrow\; (p' \vee q') \\ p \wedge \langle \mathcal{N} \wedge \mathcal{A} \rangle_f \;\Rightarrow\; q' \\ p \;\Rightarrow\; \mathit{Enabled}\,\langle \mathcal{A} \rangle_f \end{array}}{\Box[\mathcal{N}]_f \wedge \mathit{WF}_f\,(\mathcal{A}) \;\Rightarrow\; (p \leadsto q)}$$

WF2. $$\frac{\begin{array}{l} \langle \mathcal{N} \wedge \mathcal{B} \rangle_f \;\Rightarrow\; \langle \overline{\mathcal{M}} \rangle_{\overline{g}} \\ p \wedge p' \wedge \langle \mathcal{N} \wedge \mathcal{A} \rangle_f \wedge \overline{\mathit{Enabled}\,\langle M \rangle_g} \;\Rightarrow\; \mathcal{B} \\ p \wedge \overline{\mathit{Enabled}\,\langle \mathcal{M} \rangle_g} \;\Rightarrow\; \mathit{Enabled}\,\langle \mathcal{A} \rangle_f \\ \Box[\mathcal{N} \wedge \neg\mathcal{B}]_f \wedge \mathit{WF}_f\,(\mathcal{A}) \wedge \Box F \wedge \Diamond\Box\overline{\mathit{Enabled}\,\langle \mathcal{M} \rangle_g} \;\Rightarrow\; \Diamond\Box p \end{array}}{\Box[\mathcal{N}]_f \wedge \mathit{WF}_f\,(\mathcal{A}) \wedge \Box F \;\Rightarrow\; \overline{\mathit{WF}_g\,(\mathcal{M})}}$$

SF1. $$\frac{\begin{array}{l} p \wedge [\mathcal{N}]_f \;\Rightarrow\; (p' \vee q') \\ p \wedge \langle \mathcal{N} \wedge \mathcal{A} \rangle_f \;\Rightarrow\; q' \\ \Box p \wedge \Box[\mathcal{N}]_f \wedge \Box F \;\Rightarrow\; \Diamond\,\mathit{Enabled}\,\langle \mathcal{A} \rangle_f \end{array}}{\Box[\mathcal{N}]_f \wedge \mathit{SF}_f\,(\mathcal{A}) \wedge \Box F \;\Rightarrow\; (p \leadsto q)}$$

SF2. $$\frac{\begin{array}{l} \langle \mathcal{N} \wedge \mathcal{B} \rangle_f \;\Rightarrow\; \langle \overline{\mathcal{M}} \rangle_{\overline{g}} \\ p \wedge p' \wedge \langle \mathcal{N} \wedge \mathcal{A} \rangle_f \;\Rightarrow\; B \\ p \wedge \overline{\mathit{Enabled}\,\langle \mathcal{M} \rangle_g} \;\Rightarrow\; \mathit{Enabled}\,\langle \mathcal{A} \rangle_f \\ \Box[\mathcal{N} \wedge \neg\mathcal{B}]_f \wedge \mathit{SF}_f\,(\mathcal{A}) \wedge \Box F \wedge \Box\Diamond\overline{\mathit{Enabled}\,\langle \mathcal{M} \rangle_g} \;\Rightarrow\; \Diamond\Box p \end{array}}{\Box[\mathcal{N}]_f \wedge \mathit{SF}_f\,(\mathcal{A}) \wedge \Box F \;\Rightarrow\; \overline{\mathit{SF}_g\,(\mathcal{M})}}$$

**Quantification**
Let $x$ be a flexible variable, $c$ be a rigid variable, and $e$ be a constant expression:

E1. $$\frac{}{F(f/x) \Rightarrow \exists x.F}$$

E2. $$\frac{\begin{array}{l} F \Rightarrow G \\ x \text{ does not occur free in } G \end{array}}{(\exists x.F) \Rightarrow G}$$

F1. $$\frac{}{F(e/c) \Rightarrow \exists c.F}$$

F2 $$\frac{\begin{array}{l} F \Rightarrow G \\ c \text{ does not occur free in } G \end{array}}{(\exists c.F) \Rightarrow G}$$

## A.2 Simple Properties of TLA

**Lemma A.1** *Let $\mathcal{N}$ be an action and $f$ be a state function, then*

$$\langle \mathcal{N} \rangle_f \Leftrightarrow \neg[\neg\mathcal{N}]_f$$

**Proof:**

$$\begin{array}{lll} \langle \mathcal{N} \rangle_f & \Leftrightarrow \begin{array}{l} \wedge\ \mathcal{N} \\ \wedge\ f \neq f' \end{array} & \text{(Def. } \langle\ \rangle\text{)} \\ & \Leftrightarrow \neg \begin{array}{l} \vee\ \neg\mathcal{N} \\ \vee\ f = f' \end{array} & \text{(prop. logic)} \\ & \Leftrightarrow \neg[\neg\mathcal{N}]_f & \text{(Def. [ ])} \end{array}$$

**qed**

**Lemma A.2** *Let $F, G_1, G_2$, and $G_3$ be TLA-formulae and given that*

$$(F \Rightarrow \Box G_1) \Leftrightarrow (F \Rightarrow \Box G_2)$$

*is valid, then the following formula is valid, too:*

$$F \Rightarrow \Box \begin{array}{l} \vee\ G_1 \\ \vee\ G_3 \end{array} \Leftrightarrow F \Rightarrow \Box \begin{array}{l} \vee\ G_2 \\ \vee\ G_3 \end{array}$$

**Proof:** Simple temporal logic **qed**

## A.3 Proof Support for TLA

Proving temporal properties manually can be a tedious and error prone task. Therefore, an automated or interactive proof system can be a valuable aid. In the following we describe shortly how the temporal prover STeP [Manna *et al.*, 1994, Bjørner *et al.*, 1997] was used to prove validity of TLA-formulae.

A TLA-formula $F$ is called a propositional temporal formula, if it is constructed from (unprimed) boolean variables, the boolean constants *true* and *false* and the boolean and temporal operators. In such a formula only the empty tuple $\langle\rangle$ of variables can be used as state function. Hence each action $\mathcal{N}$ is actually a predicate and $\Box[\mathcal{N}]_{\langle\rangle}$ is valid if and only if $\Box\mathcal{N}$ is valid. A propositional temporal formula $F$ can be considered as a formula of the temporal logic LTL of [Manna and Pnueli, 1992], too. Note that $\Rightarrow$ and $\Leftrightarrow$ are written as $\rightarrow$ and $\leftrightarrow$, respectively, in LTL. In TLA and in LTL runs are used as models of temporal formulae. It is easy to see, that the axioms and proofrules for propositional LTL, including the past fragment, are valid in TLA. Hence the following theorem is valid:

**Theorem A.1** *Let $F$ be a propositional temporal formulae which is temporarily valid in LTL, then it is valid in TLA, too.*

Because the temporal prover STeP can be used to check temporal validity of propositional temporal formulae, it can be used as a prover for TLA, too. The reason to use STeP instead of tools tailored directly towards TLA, e.g. [Engberg, 1995], was that we already had some experience with STeP.

## A.4 Auxiliary Lemmata

**Lemma A.3** *The variable step.taken changes its value only when Execute.step-steps are taken.*

$$step.taken \neq step.taken' \;\Leftrightarrow\; Execute.step$$

**Proof:** By the definitions of *Execute.step*, *External.change*, and *statechart*. **qed**

**Lemma A.4** *Let $\mathsf{A}$ be a state, $\mathcal{N}$ a TLA-action and $f$ a state function, then the following equivalence is valid*

$$\begin{array}{lcl} \mathit{Execute.stay}_{*+}(\mathsf{A}) & \Leftrightarrow & \mathit{Execute.stay}_{*+}(\mathsf{A}) \\ \Rightarrow \Box[\![\, \mathsf{A} \rhd_{*+} \Diamond \langle \mathcal{N} \rangle_f \,]\!]_l^+ & & \Rightarrow \Box\Diamond \langle \mathcal{N} \wedge \mathit{Execute.stay}(\mathsf{A}) \rangle_f \end{array}$$

**Proof:**

$$\begin{array}{lll}
 & \mathit{Execute.stay}_{*+}(\mathsf{A}) & \\
 & \Rightarrow \Box[\![\, \mathsf{A} \rhd_{*+} \Diamond \langle \mathcal{N} \rangle_f \,]\!]_l^+ & \\
\equiv & \mathit{Execute.stay}_{*+}(\mathsf{A}) & (\text{Def. } [\![\ ]\!]_l^+) \\
 & \Rightarrow \Box[\![\, \mathsf{A} \rhd_{*+} \Box[\neg\mathcal{N}]_f \,]\!]_l^- & \\
\equiv & \mathit{Execute.stay}_{*+}(\mathsf{A}) & (\text{Def. } [\![\ ]\!]_l^-) \\
 & \Rightarrow \Box\Diamond \langle\, \mathit{Staying}(\mathsf{A}) \wedge \langle \mathcal{N} \rangle_f \,\rangle_{step.taken} & \\
\equiv & \mathit{Execute.stay}_{*+}(\mathsf{A}) & (\text{Def. } \langle\ \rangle) \\
 & \Rightarrow \Box\Diamond \wedge\ step.taken \neq step.taken' & \\
 & \qquad\ \wedge\ \mathit{Staying}(\mathsf{A}) & \\
 & \qquad\ \wedge\ \mathcal{N} & \\
 & \qquad\ \wedge\ f \neq f' & \\
\Leftrightarrow & \mathit{Execute.stay}_{*+}(\mathsf{A}) & (\text{Lem. A.3}) \\
 & \Rightarrow \Box\Diamond \wedge\ \mathit{Execute.step} & \\
 & \qquad\ \wedge\ \mathit{Staying}(\mathsf{A}) & \\
 & \qquad\ \wedge\ \mathcal{N} & \\
 & \qquad\ \wedge\ f \neq f' & \\
\Leftrightarrow & \mathit{Execute.stay}_{*+}(\mathsf{A}) & (\text{Def. } \mathit{Execute.stay}) \\
 & \Rightarrow \Box\Diamond \wedge\ \mathit{Execute.stay}(\mathsf{A}) & \\
 & \qquad\ \wedge\ \mathcal{N} & \\
 & \qquad\ \wedge\ f \neq f' & \\
\equiv & \mathit{Execute.stay}_{*+}(\mathsf{A}) & (\text{Def. } \langle\ \rangle) \\
 & \Rightarrow \Box\Diamond \langle \mathcal{N} \wedge \mathit{Execute.stay}(\mathsf{A}) \rangle_f &
\end{array}$$

**qed**

**Lemma A.5** *Let $A$ be a state, $\mathcal{N}$ a TLA-action and $f$ a state function, such that*

$$\begin{array}{l} \wedge\ \mathit{Enabled}\,\langle\, \mathcal{N}\,\rangle_f \\ \wedge\ \mathit{Enabled}\ \mathit{Execute.\,stay}(A) \\ \Rightarrow \mathit{Enabled} \wedge\ \langle\,\mathcal{N}\,\rangle_f \\ \qquad\qquad\quad \wedge\ \mathit{Execute.\,stay}(A) \end{array} \tag{A.1}$$

*is valid. Then the following formulae are valid:*

1. $\mathit{Execute.\,stay}_{\Box\Diamond}(\mathsf{A}) \Rightarrow \Box[\![\, \mathsf{A} \rhd_{\Box\Diamond} \Diamond\neg\, \mathit{Enabled}\,\langle\,\mathcal{N}\,\rangle_f\,]\!]_l^+ \quad \Rightarrow \quad \mathit{Execute.\,stay}_{\Box\Diamond}(\mathsf{A}) \Rightarrow \Box\Diamond\neg\, \mathit{Enabled}\,\langle\,\mathcal{N} \wedge \mathit{Execute.\,stay}(A)\,\rangle_f$
2. $\mathit{Execute.\,stay}_{\Diamond\Box}(\mathsf{A}) \Rightarrow \Box[\![\, \mathsf{A} \rhd_{\Diamond\Box} \Diamond\neg\, \mathit{Enabled}\,\langle\,\mathcal{N}\,\rangle_f\,]\!]_l^+ \quad \Leftrightarrow \quad \mathit{Execute.\,stay}_{\Diamond\Box}(\mathsf{A}) \Rightarrow \Box\Diamond\neg\, \mathit{Enabled}\,\langle\,\mathcal{N} \wedge \mathit{Execute.\,stay}(A)\,\rangle_f$
3. $\mathit{Execute.\,stay}_{\Box\Diamond}(\mathsf{A}) \Rightarrow \Diamond[\![\, \mathsf{A} \rhd_{\Box\Diamond} \Box\neg\, \mathit{Enabled}\,\langle\,\mathcal{N}\,\rangle_f\,]\!]_l^+ \quad \Rightarrow \quad \mathit{Execute.\,stay}_{\Box\Diamond}(\mathsf{A}) \Rightarrow \Diamond\Box\neg\, \mathit{Enabled}\,\langle\,\mathcal{N} \wedge \mathit{Execute.\,stay}(A)\,\rangle_f$
4. $\mathit{Execute.\,stay}_{\Diamond\Box}(\mathsf{A}) \Rightarrow \Diamond[\![\, \mathsf{A} \rhd_{\Diamond\Box} \Box\neg\, \mathit{Enabled}\,\langle\,\mathcal{N}\,\rangle_f\,]\!]_l^+ \quad \Leftrightarrow \quad \mathit{Execute.\,stay}_{\Diamond\Box}(\mathsf{A}) \Rightarrow \Diamond\Box\neg\, \mathit{Enabled}\,\langle\,\mathcal{N} \wedge \mathit{Execute.\,stay}(A)\,\rangle_f$

**Proof:**

1.

$$\begin{array}{lll}
 & \mathit{Execute.\,stay}_{\Box\Diamond}(A) & (1) \\
 & \Rightarrow \Box[\![\, A \rhd_{\Box\Diamond} \Diamond\neg\, \mathit{Enabled}\,\langle\,\mathcal{N}\,\rangle_f\,]\!]_l^+ & \\
\equiv & \mathit{Execute.\,stay}_{\Box\Diamond}(A) & (2, \text{Def. } [\![\ ]\!]_l^+, [\![\ ]\!]_l^-) \\
 & \Rightarrow \Box\ \mathit{Execute.\,stay}_{\Box\Diamond}(A) & \\
 & \qquad \Rightarrow \Diamond[\![\, A \rhd_{\Box\Diamond} \mathit{Enabled}\,\langle\,\mathcal{N}\,\rangle_f\,]\!]_l^- & \\
\Leftrightarrow & \mathit{Execute.\,stay}_{\Box\Diamond}(A) & (3, \text{temp. logic}) \\
 & \Rightarrow \Box\Diamond[\![\, A \rhd_{\Box\Diamond} \mathit{Enabled}\,\langle\,\mathcal{N}\,\rangle_f\,]\!]_l^- & \\
\equiv & \mathit{Execute.\,stay}_{\Box\Diamond}(A) & (4, \text{Def. } [\![\ ]\!]_l^-) \\
 & \Rightarrow \Box\Diamond \wedge\ \mathit{Enabled}\ \mathit{Execute.\,stay}(A) & \\
 & \qquad \wedge\ \neg\, \mathit{Enabled}\,\langle\,\mathcal{N}\,\rangle_f & \\
\Rightarrow & \mathit{Execute.\,stay}_{\Box\Diamond}(A) & (5) \\
 & \Rightarrow \Box\Diamond\neg\, \mathit{Enabled}\,\langle\,\mathcal{N}\,\rangle_f & \\
\Rightarrow & \mathit{Execute.\,stay}_{\Box\Diamond}(A) & (6) \\
 & \Rightarrow \Box\Diamond\neg\, \mathit{Enabled}\,\langle\,\mathcal{N} \wedge \mathit{Execute.\,stay}(A)\,\rangle_f &
\end{array}$$

In the step from (4) to (5) we have an implication, because it cannot be shown here from (5) that infinitely often an *Execute. stay*$(A)$ is enabled and a $\langle\,\mathcal{N}\,\rangle_f$-step is not enabled. The implication from (5) to (6) is valid because if an $\langle\,\mathcal{N}\,\rangle_f$-step is not enabled, then an $(\langle\,\mathcal{N}\,\rangle_f \wedge \mathit{Execute.\,stay}(A))$-step is not enabled either.

2. The first steps of the proof of this case are as above. The implications (4) $\Rightarrow$ (5) and (5) $\Rightarrow$ (6) can be shown as before, too. For the reverse directions we have:

$$
\begin{array}{lll}
 & \mathit{Execute.stay}_{\Diamond\Box}(\mathsf{A}) & (4, \text{Def. } [\![\ ]\!]_l^-) \\
 & \Rightarrow \Box\Diamond \wedge\ \mathit{Enabled}\ \mathit{Execute.stay}(\mathsf{A}) & \\
 & \qquad \wedge\ \neg\, \mathit{Enabled}\ \langle\, \mathcal{N}\, \rangle_f & \\
\Leftarrow & \mathit{Execute.stay}_{\Diamond\Box}(\mathsf{A}) & (5) \\
 & \Rightarrow \Box\Diamond\neg \wedge\ \mathit{Enabled}\ \mathit{Execute.stay}(\mathsf{A}) & \\
 & \qquad \wedge\ \mathit{Enabled}\ \langle\, \mathcal{N}\, \rangle_f & \\
\Leftarrow & \mathit{Execute.stay}_{\Diamond\Box}(\mathsf{A}) & (6, \text{Ass. A.1}) \\
 & \Rightarrow \Box\Diamond\neg\, \mathit{Enabled}\ \langle\, \mathcal{N} \wedge \mathit{Execute.stay}(\mathsf{A})\, \rangle_f &
\end{array}
$$

To conclude (4) from (5) we use the assumption $\mathit{Execute.stay}_{\Diamond\Box}(\mathsf{A})$, which states that $\mathit{Execute.step}$ is enabled continuously from some point of time and as a consequence $\mathit{Execute.stay}(\mathsf{A})$ is enabled continuously from some point of time, too. Hence, it can only be infinitely often not the case that an $\mathit{Execute.stay}(\mathsf{A})$-step and and $\langle\, \mathcal{N}\, \rangle_f$-step is enabled, if infinitely often an $\langle\, \mathcal{N}\, \rangle_f$-step is not enabled although an $\mathit{Execute.stay}(\mathsf{A})$-step is enabled.

To conclude (5) from (6) we use the contraposition of assumption in Equation A.1.

3. The general structure of the proof of this case is as in case 1, but the different definitions of *WF* and *SF* have to be taken into account.

$$
\begin{array}{lll}
 & \mathit{Execute.stay}_{\Box\Diamond}(\mathsf{A}) & (1) \\
 & \Rightarrow \Diamond [\![\, \mathsf{A} \rhd_{\Box\Diamond} \Box\neg\, \mathit{Enabled}\ \langle\, \mathcal{N}\, \rangle_f\, ]\!]_l^+ & \\
\equiv & \mathit{Execute.stay}_{\Box\Diamond}(\mathsf{A}) & (2, \text{Def. } [\![\ ]\!]_l^+) \\
 & \Rightarrow \Diamond\ \mathit{Execute.stay}_{\Box\Diamond}(\mathsf{A}) & \\
 & \qquad \Rightarrow \Box \wedge\ \mathit{Enabled}\ \mathit{Execute.stay}(\mathsf{A}) & \\
 & \qquad\quad \wedge\ \neg\, \mathit{Enabled}\ \langle\, \mathcal{N}\, \rangle_f & \\
\Leftrightarrow & \mathit{Execute.stay}_{\Box\Diamond}(\mathsf{A}) & (3, \text{temp. logic (STeP)}) \\
 & \Rightarrow \Diamond\Box \wedge\ \mathit{Enabled}\ \mathit{Execute.stay}(\mathsf{A}) & \\
 & \qquad \wedge\ \neg\, \mathit{Enabled}\ \langle\, \mathcal{N}\, \rangle_f & \\
\Rightarrow & \mathit{Execute.stay}_{\Box\Diamond}(\mathsf{A}) & (4) \\
 & \Rightarrow \Diamond\Box \vee\ \neg\, \mathit{Enabled}\ \langle\, \mathcal{N}\, \rangle_f & \\
 & \qquad \vee\ \neg\, \mathit{Enabled}\ \mathit{Execute.stay}(\mathsf{A}) & \\
\Rightarrow & \mathit{Execute.stay}_{\Box\Diamond}(\mathsf{A}) & (5) \\
 & \Rightarrow \Diamond\Box\neg\, \mathit{Enabled}\ \langle\, \mathcal{N} \wedge \mathit{Execute.stay}(\mathsf{A})\, \rangle_f &
\end{array}
$$

4. The proof of this case can be derived from the one of case 3 using the same arguments as in the proof of case 2.

**qed**

**Lemma A.6** *Let $p$ and $q$ be two predicates and $\mathsf{A}$ a state, then*

$$\begin{array}{lcl} Execute.\,stay_{*+}(\mathsf{A}) & \Rightarrow & Execute.\,stay_{*+}(\mathsf{A}) \\ \Rightarrow \Box[\![\, \mathsf{A} \rhd_{*+} p \;\Rightarrow\; q \,]\!]_l^+ & & \Rightarrow \Box \;[\![\, \mathsf{A} \rhd_{*+} p \,]\!]_l^+ \\ & & \quad\;\; \Rightarrow [\![\, \mathsf{A} \rhd_{*+} q \,]\!]_l^+ \end{array}$$

**Proof:**

$$\begin{array}{lcll} Execute.\,stay_{*+}(\mathsf{A}) & \equiv & Execute.\,stay_{*+}(\mathsf{A}) & (\text{Def. } [\![\;\;]\!]_l^+) \\ \Rightarrow \Box[\![\, \mathsf{A} \rhd_{*+} p \;\Rightarrow\; q \,]\!]_l^+ & & \Rightarrow \Box \vee\; [\![\, \mathsf{A} \rhd_{*+} p \,]\!]_l^- & \\ & & \qquad \vee\; [\![\, \mathsf{A} \rhd_{*+} q \,]\!]_l^+ & \\ & \Rightarrow & Execute.\,stay_{*+}(\mathsf{A}) & (\text{Lem. A.2, A.7}) \\ & & \Rightarrow \Box \vee\; \neg[\![\, \mathsf{A} \rhd_{*+} p \,]\!]_l^+ & \\ & & \qquad \vee\; [\![\, \mathsf{A} \rhd_{*+} q \,]\!]_l^+ & \\ & equiv & Execute.\,stay_{*+}(\mathsf{A}) & \\ & & \Rightarrow \Box \;[\![\, \mathsf{A} \rhd_{*+} p \,]\!]_l^+ & \\ & & \qquad \Rightarrow [\![\, \mathsf{A} \rhd_{*+} q \,]\!]_l^+ & \end{array}$$

**qed**

**Lemma A.7** *Let $p$ be a formula and $\mathsf{A}$ be a state, then negating $p$ distributes over associating $p$ with $\mathsf{A}$, provided it is temporarily quantified and there are infinitely many Execute. stay($\mathsf{A}$)-steps.*

$$\begin{array}{lcl} Execute.\,stay_{*+}(\mathsf{A}) & \Rightarrow & Execute.\,stay_{*+}(\mathsf{A}) \\ \Rightarrow \Box[\![\, \mathsf{A} \rhd_{*+} p \,]\!]_l^- & & \Rightarrow \Box\neg[\![\, \mathsf{A} \rhd_{*+} p \,]\!]_l^+ \end{array}$$

**Proof:**

$$\begin{array}{cll} & Execute.\,stay_{*+}(\mathsf{A}) \;\Rightarrow\; \Box[\![\, \mathsf{A} \rhd_{*+} p \,]\!]_l^- & \\ \equiv & Execute.\,stay_{*+}(\mathsf{A}) \;\Rightarrow\; \Box \wedge\; Enabled\ Execute.\,stay(\mathsf{A}) & (\text{Def. } [\![\;\;]\!]_l^-) \\ & \qquad\qquad\qquad\qquad\quad\;\; \wedge\; \neg p & \\ \Rightarrow & Execute.\,stay_{*+}(\mathsf{A}) \;\Rightarrow\; \Box\neg p & (\text{prop., temp. logic}) \\ \Leftrightarrow & Execute.\,stay_{*+}(\mathsf{A}) \;\Rightarrow\; \Box\neg \wedge\; Enabled\ Execute.\,stay(\mathsf{A}) & (\text{prop., temp. logic}) \\ & \qquad\qquad\qquad\qquad\qquad\;\; \wedge\; p & \\ \equiv & Execute.\,stay_{*+}(\mathsf{A}) \;\Rightarrow\; \Box\neg[\![\, \mathsf{A} \rhd_{*+} p \,]\!]_l & (\text{Def. } [\![\;\;]\!]_l^+) \end{array}$$

**qed**

# Lebenslauf

## Persönliche Daten

| | |
|---|---|
| Name | Thomas Deiß |
| Geburtsdatum | 8. Februar 1964 |
| Geburtsort | Reutlingen |
| Familienstand | verheiratet, drei Kinder |
| Staatsangehörigkeit | deutsch |

## Schulbesuch

| | |
|---|---|
| 1970-1974 | Grundschule in Pfullingen |
| 1974-1983 | Friedrich-Schiller-Gymnasium in Pfullingen<br>Abschluß: Abitur |

## Grundwehrdienst

| | |
|---|---|
| 10/83-09/84 | in Landsberg/Lech |

## Studium

| | |
|---|---|
| 10/84-05/90 | Informatik mit Nebenfach Wirtschaftswissenschaften an der Universität Kaiserslautern<br>Abschluß: Diplom |

## Berufstätigkeit

| | |
|---|---|
| 08/90-12/94 | Wissenschaftlicher Mitarbeiter in der Arbeitsgruppe "Grundlagen der Informatik" von Prof. Dr. K. Madlener Universität Kaiserslautern |
| 01/95-06/99 | Wissenschaftlicher Mitarbeiter im Sonderforschungsbereich 501, Teilprojekt C1 "Formale Beschreibungstechniken" von Prof. Dr. J. Avenhaus und Prof. Dr. K. Madlener |