University of Sheffield

Department of Computer Science

Automated testing of Harel's statecharts

Kirill Bogdanov

Submitted towards the degree of
Doctor of Philosophy
January 2000

Contents

List of Figures

Abstract

The work described below attempts to provide a flexible method to improve the quality of software development, by improvement in testing. It shows how a formal testing method can be applied to the statechart notation widely used in industry. The method focuses on testing implementations against statechart designs and, subject to certain specific requirements, allows us to prove that the implementation is behaviourally equivalent to the design, by testing. The described testing method is based on the X-machine testing method and utilises the 'divide and conquer' strategy to testing different features of statecharts.

A design of a software system can be built top-down in a sequence of refinements, i.e. constrained additions of some detail. If the implementation is developed in parallel, certain faults become impossible. This allows us to reduce the size of a test set dramatically.

The prototype implementation of the testing method has been built and case studies evaluating its applicability to parts of real systems were successful.

Acknowledgement

I would like to thank my supervisor, Prof. M. Holcombe, for his guidance throughout the three and a half years. Many thanks to members of the Verification and Testing Research Group in the Department of Computer Science, the University of Sheffield and everyone in FT3/SM group in the DaimlerChrysler, whom I met. Special thanks to H. Dörr, M. Fairtlough, J. Hiemer, C. Jordan, B. Norton, S. Sadeghipour and H. Singh for useful discussions, to R. Büssow and W. Grieskamp for help with $\mu\mathcal{SZ}$ and to R. Hierons for help with his testing methods.

This project is sponsored by the DaimlerChrysler.

Chapter 1

Introduction

1.1 Software testing

In this section we introduce the reader to the field of software testing and related problems, outline approaches to testing and show where the work described in the thesis can help to improve software quality. Much of the background material is from the `swtest-discuss` mailing list [Fau99] which is in the opinion of the author the best main list for discussions on the quality assurance at present.

1.1.1 Poor quality software

Much of the software available today is of rather poor quality. Crashing with possible data loss seems to be as normal as rain and despite seemingly many complaints, things are not getting better. The main reason for poor quality of mass-marketed software seems to be that the number of features and time to market outweighs benefits of quality. Even more, companies include explicit disclaimers that they are not liable for whatever happens to the person using their software and sometimes a prohibition for a customer to publish benchmarks of their products without their consent. Presently they seek almost complete protection from a customer with a law proposal in the USA [Kan99].

Industry involved in building safety-critical systems, appears to be quite different to the one described above. Not only do they seem to care more about quality, they are required by law to comply with certain standards. Unfortunately, these requirements concern the process they follow and companies seem to be able to get certification on the basis of process documents rather than the product itself.

1.1.2 Testing means different things to different people

The term 'testing' is heavily overloaded. It could mean anything from ad-hoc 'breaking' the system to generation of test sets using a formal design; load or stress testing is also referred to as a form of testing. The same holds outside computing too: the author has seen a tobacco advertisement banner ending with words 'test it'; whether it meant to evaluate the taste of the smoke[1], assess the quality of packing, properties of the filter, the number of milligrams of tobacco packed, or whether it causes fatal diseases or not, it did not say.

1.1.3 Testing as a part of a process

Testing has been traditionally viewed as a way to find faults in software, by exposing it to inputs deliberately chosen to cause a malfunction [Mye79, Pre94]. This rather destructive activity generally caused relations between testers and developers to be poor and thus testers were advised to acquire people skills to communicate problems without fighting the ego of developers. Thus, testing cannot in general be viewed on its own, but as a part of a process.

Other parts of the process include bug/feature reporting, inspections of different kind, configuration management and other activities. One can talk endlessly about how these can be organised, but every such management approach has to be unique to an organisation using it since it is dependent on the in-company culture, background of employees and other hard-to-capture aspects. How well companies do in this respect, can be assessed with Capability Maturity Model [PCCW93] but universally-applicable recommendations are hard to make; one can only expect that the lower a company is, the easier it is for it to climb up, but whether it would improve the quality of the product they build is uncertain. In the thesis we do not cover process-related aspects of quality assurance; instead, we look at the one very specific part of it: testing.

1.1.4 Types of testing

Here we briefly describe different types of testing and their place in the software lifecycle.

Software lifecycle

In this section we provide an overview of the waterfall software development model.

[1]the author must admit he is a non-smoker thus some of the statements could be slightly inaccurate.

1. Initially, developers of a software product talk to their customers and derive a set of *requirements*. They state what functionality is needed and what additional properties such as reliability the software to be developed has to achieve. Compliance of these requirements to what is needed could be assured by prototyping, for example by construction of an interface part of the system to be built and observation of a few customers using it. This activity of making sure developers know they are building the right product is called *validation*.

2. In consultation with the requirements, a design is built which is a high-level description of the functionality the system is supposed to have. *Verification* is the process of making sure that this design is compliant with the requirements. It can be done by model-checking which is essentially an enumeration of all possible behaviours of the system and comparison of them to requirements, or theorem-proving, during which a formal proof is made that the design satisfies requirements. The former could be difficult to apply due to big size of the application's data space; the latter is generally hard to automate.

3. *Coding*, or *implementing* is the process of translation of a design into code in some programming language. Errors introduced at this stage can be discovered by testing. Testing of an implementation to assure its correct behaviour w.r.t its design is the subject of this thesis.

Different types of testing

Testing is a process of supplying a system under test with some values and making conclusions on the basis of its behaviour. In order to prepare those values, test cases are usually derived and then populated with test data. Test cases correspond to groups of data values which could be used; during selection of test data we derive specific values. For example, a test case could be $x > 0$ and we might choose 1 as test data.

Testing might be used to investigate whether a system under test satisfies some properties. As well as many others, these could be stability under load, ease of use (usability testing) or the one we consider further, correct implementation of a design. Similar to the CMM model, there is one for maturity of testing in an organisation [KP98].

It is possible to distinguish white- and black-box testing. White box test methods include different ways of code coverage where test cases are supposed to cover paths through the code without consideration of the design. Black-box testing derives test cases and data from specification and as such is better at uncovering differences between intended and implemented behaviour. This is the type of testing the author's research has been focusing upon.

While most testing methods are informal [Mye79, DN84, WTF94, RR93], those making formal claims on the basis of testing not revealing faults have been developed for finite-state machines [Cho78, FvBK$^+$91], X-machines [HI98, IH98b, IH98a, IH97] and CSP specifications [PS96a]. Such testing methods make specific assumptions which together with results of testing allow one to show correct behaviour of implementations. Unlike informal methods, where one could also try to state such assumptions, formal methods make them precise and easy to verify. Not all methods strictly fall in either of these two groups: [DB94] features a unique combination of formal and informal approaches.

1.1.5 Problems with testing and quality assurance

Many problems with the type of testing considered are mainly due to the following two causes:

1. Lack of effective methods and tools to do testing.

   (a) For testing based on a design, we need to construct tests which could demonstrate the presence of the specified behaviour and absence of undesirable behaviour. A variety of testing methods developed for this purpose exist. Although authors might claim great results, they may only be valid in a particular area. For this reason, there is an approach to have some kind of 'maturity' for such empirical studies [HHH$^+$99].

   Additionally, many methods are hindered by an absence of a complete specification and supported by tools which are no more than prototypes and are not ready for general use. Finally, some specification languages used by tool authors while nice on their own, are not likely to be widely acceptable due to limited scope of their application or complex notation.

   (b) Generating test cases could be difficult by hand even for moderately sized systems; some automation is often necessary. A classical example of a snake oil includes capture-playback tools for testing graphical interfaces. The main problem with them is that when the layout of a screen changes, such as when a button is moved to another screen, much of the test set of it has to be captured once again. This type of problems can be solved by using a system where testers specify test cases in some generic way and the system derives test data for it. For example, we could specify the test case for the button considered by clicking it rather than an area on some screen and then testing the dialog which appeared. In this case, it is applicable regardless which screen the button is used on.

Scalable tool support is often difficult to achieve since at the early stages of automation simple tools (such as capture-playback) are easy to deploy while more complex ones (tools involving programming) require considerably more effort to start with. As a project grows, simple tools may become a burden such that all time would be spent maintaining an existing test suite while the second class of tools could then provide ease of maintainability.

2. Absence of a proper process of software development.

    (a) In general, the waterfall model, while perhaps useful for contract development, appears to be too restrictive in many cases. The process could be significantly improved by, for example, replacing the waterfall model with the spiral one [Pre94]. Nevertheless, in many cases companies are unwilling to change the way they do things in order to improve quality and instead try to obtain 'magic bullet' test tools which are supposed to solve their problems.

    (b) Testing has to be aimed: we could wish to find problems with the system [Mye79] (such as security holes), ensure correct operation under use from a typical user [HT90] (who is not expected, for instance, to press more than 10 keys at the same time), or make certain that the system does not catastrophically malfunction (includes incorrect data as well as load testing).

1.1.6 Achievements of the work to be described

In this thesis we focus on testing as a way to ensure correct implementation of a system, refer to item 1 above. The method has been developed for automated testing of implementation of systems specified with statecharts; reference tool support has been built. The author hopes that this work would be useful to overcome the stated problems as described below.

- Statecharts is a widely-used language for the specification and design of reactive systems [LHHR94, FJW97, BGK98]. Its graphical nature making it easy to understand, and the support of the Statemate tool contributed to its success.

- Statecharts are easier to test than many conventional languages. This is a result of having a state transitions system and transitions defined separately, making separate testing of them possible.

- The testing method provides provable correctness. Based on the assumptions taken during test set generation, and testing not revealing faults in an implementation, we can show behavioural equivalence of the implementation to the design. The possibility of this for the developed testing method, compared to many others, is its great advantage.

Had the method been developed a few years ago, a situation [Fel98] where a collision-avoidance system, described in [LHHR94, HL96, CAB+98], nearly caused two crashes in one day, could probably be avoided.

- The testing method developed lacks complete automation. This is inevitable since many problems of test derivation are computationally very difficult. In addition, complete testing means infinite testing and consequently some assumptions have to be made by a human tester to reduce the size of a test set to a manageable size.

  In the context of safety-critical systems reliability is paramount and thus complete automation is not necessary.

  In non-safety critical software industry, provable correctness is not always necessary. In such cases, many conditions could be relaxed. We still gain benefits from its properties such that after relaxing some assumptions, we can derive the type of faults which does not get detected by the modified method, thus helping a tester to make decisions.

- Refinement is a process of incremental addition of new functionality to a system without significant changes to the present one. In terms of testing, constrained changes of a system allow unchanged parts of it not to be re-tested again. Testing of refined systems is described in [Ipa95, IH98a] for X-machines and in [SCW98, DB98, HHS86, Spi92] for the Z notation [Toy98, Spi92, WD96].

- The *TestGen* tool supporting the method is not ready for industrial use yet, although work on it continues in the followup project and TestGen will be improved to allow Daimler-Chrysler AG to try it on its projects. If this proves to be useful, a company will be contracted to build a commercial version of the tool and it will be used to improve the quality of software in the products of Daimler-Chrysler AG.

1.1.7 Formal verification and testing

Formal methods are used in industry mostly for verification of properties of a design by model-checking and/or theorem-proving where they were of benefit to a number of projects [CW96]. A lot of the work is focused at verifying whether a design has some required properties [Cor96, BCM+92, BCCZ98, CGH94, CJ95]. As for ascertaining that the code is correct with respect to a design, most of the time either a compiler is proven correct or the code generated by it is proven to be correct [BKN, Sac98, PSS98, Shi95]. Unfortunately, this does not shed much light on what will happen when that code is actually run since when proving correctness we assume that an operating system and hardware are behaving in an expected way. In practice, such assumptions may be wrong but defects are hard to detect;

they may nevertheless cause a system to fail. For example, the author's P233MMX processor behaves well at the clock speed of 233MHz but when overclocked to 266MHz, certain programs fail unexpectedly while others behave properly. Testing of a program in the context of a target system could help us to reduce a risk of developing a system which is unreliable.

In the area of statecharts, a great amount of effort was invested in model-checking [CAB$^+$98, Day93, CK96, PS97b, BW98, WVF95], most of which is based on Binary Decision Diagrams (BDD) [And94]. Relatively informal approaches can also be used [HL96]; usage of verification techniques to generate test cases is provided in [FJJV96, GH99]. Unfortunately, very little has been done on formal testing for designs done in statecharts. Apart from the testing method described in this thesis which is based on X-machine testing method, alternative approaches exists. [Bur98] focuses on the conversion of statecharts into Z and using a Disjunctive Normal Form (DNF) approach to test them; a later work [Bur99] described testing of behaviour of transitions with DNF, illustrated by a case study. [OA99] performs different forms of design coverage while [KHC$^+$99] additionally applies dataflow testing; none of the two are formal. The method described here was presented at conferences and workshops [Bog98, BHS98a, BHS98b], described in a report [BH98] and published in conference proceedings [BHS99]. Earlier work on it was provided in [Bog97, HB97, BH97, Bog96].

1.2 Introduction to the notation used in the thesis

We will work with either detailed specifications or high-level designs, to which the method developed is most applicable. For this reason, an implementation is tested to conform with its *design* or, in other words, *model*.

Further, a number of notations will be used:

- X-machines, introduced in Sect. 1.3 on p. 10. This is a very powerful extension of finite-state machines without a considerable increase in the complexity of testing.
- Harel's statecharts, introduced in Sect. 1.4 on p. 12. Statecharts introduce many constructs to finite-state machines and thus are difficult to test. This kind of statecharts has been implemented with some additions in the Statemate(tm) tool by Ilogix.
- $\mu\mathcal{SZ}$, introduced in Sect. 1.4.11 on p. 24. This notation combines statecharts with Z and temporal logic, replacing some of the constructs and adding new ones.
- Z, described in [Spi92, Toy98, WD96]. We expect the reader to be familiar with it and do not provide an introduction.

Due to the complexity of the notation and, in parts, some confusing semantics, not all constructs of Statemate and $\mu\mathcal{SZ}$ statecharts are covered. The testing method considers main features of syntax and semantics, present in the same form in both Statemate statecharts and $\mu\mathcal{SZ}$ without changes. These elements are what we call Harel's statecharts. Other features are covered briefly. For this reason, the term *statechart* generally refers to this subset considered and features specific to one of the notations are marked as such. Due to the author's greater familiarity with Statemate rather than $\mu\mathcal{SZ}$ statecharts, specific features are almost always Statemate ones. Usage of the Z notation in the thesis essentially follows the $\mu\mathcal{SZ}$ notation [BGGK97] with slight changes to facilitate presentation.

We write 'refer to Sect. 1.2.3 on p. 45' to mean a reference to something within the subsection 3 of the section 2 of chapter 1. The page number gives the location of an element referred to. For references to theorems, propositions, testing requirements, definitions, chapters and appendices, we use the following abbreviations:

element	abbreviation	example
theorem	Th.	Th. 1
proposition	Prop.	Prop. 1
requirement	Req.	Req. 1
definition	Def.	Def. 1
chapter	Chap.	Chap. 1
appendix	App.	App. 1

All requirements for the test method are described in Chap. 5 on p. 79.

1.3 An introduction to X-machines

Finite-state machines (*FSM*) are widely used to teach people programming as this form of graphical notation is easy to understand. Non-trivial systems cannot be modelled in this way due to lack of data representation, because an approach to model data with states leads to an unmanageable expansion in the number of them, referred to as state explosion. X-machines are similar to finite-state machines and have an explicit mechanism for operations on data. As a result, they are vastly more powerful and at the same time resemble finite-state machines such that testing methods developed for them can be applied to X-machines. In this section we provide an introduction to X-machines; statecharts and $\mu\mathcal{SZ}$ will be covered in those that follow and the testing method — in Chap. 2.

Consider a finite-state machine given in Fig. 1.1. Transitions have a triggering input (before /) and an output (after /), such that $a/0$ is triggered by a and produces 0 as well as changing a state. When this machine is supplied with an input sequence, every element of it triggers a single transition

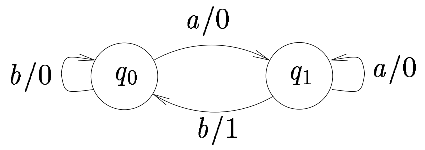


Figure 1.1: FSM which detects $a\ b$ pairs in its input sequence by producing an output of 1 upon detection

in the machine and the output of it can be observed by a user (we assume that all states are final), but not the next state. For instance, a sequence $a\ a\ b\ b$ generates output $0\,0\,1\,0$ and traverses states

$$q_0 \overset{a/0}{\to} q_1 \overset{a/0}{\to} q_1 \overset{b/1}{\to} q_0 \overset{b/0}{\to} q_0$$

Consider a more complex example where we try to count the number of times $a\ b$ occurred in the input sequence. In this case, we have states corresponding to 0th count, 1, 2 and so on, resulting in almost twice as many states as the maximal count we allow. This is depicted in Fig. 1.2. X-

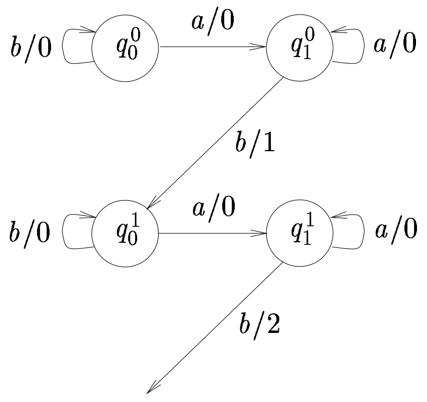


Figure 1.2: A finite-state machine which counts the number of $a\ b$ pairs in its input sequence

machines avoid this state explosion by using explicit data and functions on transitions. It means that instead of an input/output pair on a transition, we could have something more complicated, which modifies an internal *memory* of a machine. A transition diagram for the X-machine is depicted in Fig. 1.3. Functions get_a, b_ignore and b_count are defined as follows:

$$\begin{aligned} get\_a(counter, a) &= (0, counter) \\ b\_ignore(counter, b) &= (0, counter) \\ b\_count(counter, b) &= (counter+1, counter+1) \end{aligned}$$

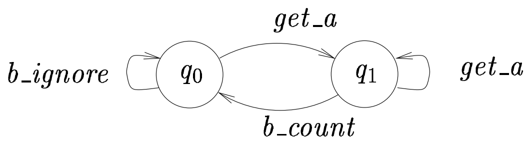


Figure 1.3: An X-machine which counts the number of *a b* pairs in its input sequence

where σ is the current element of the input sequence, and *counter* is the memory variable representing the counter. The result of every function consists of an output and the new counter value. Formally X-machines are defined in [HI98, Ipa95, Lay92] and Sect. 6.3.2 on p. 169.

Usage of an X-machine allows us to avoid the great increase in the number of states and provide a seemingly easy to understand and test example. Other nice properties of X-machines relate to languages and computability [BGG98, IH96, Ipa95].

1.4 An introduction to statecharts and $\mu\mathcal{SZ}$

Statecharts is a specification language derived from finite-state machines. The language is rather rich in features including state hierarchy and concurrency. Transitions can perform nontrivial computations unlike finite-state machines where they contain at most input/output pairs. In this section we describe Statemate statecharts [Har87, HPSS87, HN96, NH95, HLN$^+$90, MLPS97, Har97, HG96, Ilo95b]. Specific constructs, the testing of them and some alternative semantics are given in Chap. 3 on p. 46.

1.4.1 A tape recorder

Consider a simple tape recorder model capable of doing all the standard functions like *play*, *rewind*, *fast forward*, *stop* and *record* as well as changing a side of a tape when the button $\underline{play}$ is pressed during the playback or when a tape ends. The statechart is shown in Fig. 1.4. *STOP* is the initial state, indicated by the transition from the blob (described in Sect. 1.4.5 on p. 16). Transition names are selected to reflect user actions, i.e., *play* occurs when the user presses the $\underline{play}$ button, *rew_or_ff* occurs if either $\underline{rew}$ or $\underline{ff}$ buttons. The *direction* transition is triggered by the $\underline{play}$ button to change the side of a tape or by the $\underline{tape\_end}$ event when the current side has ended.

In the following, the `typewriter` font is used for TestGen implementation-related details such as class names; $\underline{underline}$ font — to denote input and output events and variables. Transition names are given in *italics* and state names are CAPITALISED.

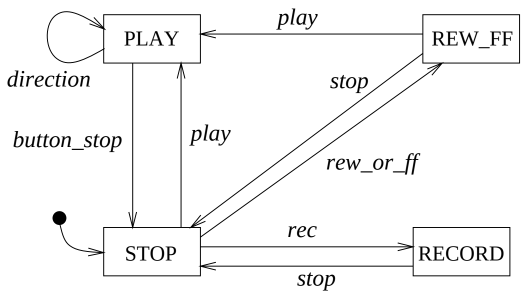


Figure 1.4: A tape recorder modelled as a statechart

The tape recorder communicates with a keyboard and a tape drive. It serves as a controller which interprets user's button presses and sends appropriate commands to a tape drive. Input variables are events $\underline{play}$, $\underline{stop}$, $\underline{rec}$, $\underline{rew}$, $\underline{ff}$ and $\underline{tape\_end}$. Output variables are $\underline{ff\_direction}$ and $\underline{operation}$. They are the commands which tell a tape drive what to do. The $\underline{operation}$ output can have one of the following values: $stop, play, record, move$. The boolean variable $\underline{ff\_direction}$ specifies the direction of a tape, with *true* meaning forward tape movement. During playback or recording it also implies the side, with side A being played or recorded forward and B — backward. The communication to a tape drive is bidirectional: the controller is notified when a tape has stopped using the event $\underline{tape\_end}$.

1.4.2 Transitions

It is possible to specify functions on transitions of our tape recorder as

$$\begin{aligned}
stop:&\quad \mathsf{df}\ \underline{stop} \lor \mathsf{df}\ \underline{tape\_end} / \mathsf{df}\ \underline{operation}' \land \mathsf{vl}\ \underline{operation}' = stop\\
button\_stop:&\quad \mathsf{df}\ \underline{stop} / \mathsf{df}\ \underline{operation}' \land \mathsf{vl}\ \underline{operation}' = stop\\
direction:&\quad \mathsf{df}\ \underline{play} \lor \mathsf{df}\ \underline{tape\_end} / \underline{ff\_direction}' = \neg\ \underline{ff\_direction}\\
play:&\quad \mathsf{df}\ \underline{play} / \mathsf{df}\ \underline{operation}' \land \mathsf{vl}\ \underline{operation}' = play\\
rec:&\quad \mathsf{df}\ \underline{rec} / \mathsf{df}\ \underline{operation}' \land \mathsf{vl}\ \underline{operation}' = record\\
rew\_or\_ff:&\quad \mathsf{df}\ \underline{rew} \lor \mathsf{df}\ \underline{ff} / \mathsf{df}\ \underline{operation}' \land \mathsf{vl}\ \underline{operation}' = move
\end{aligned}$$

The part before the '/' sign means the *trigger*, i.e. the *precondition* which is required to become *true* for a transition to occur. The *stop* transition will occur if event $\underline{stop}$ is generated; the predicate that an event is generated is expressed via $\mathsf{df}\ \underline{event\text{-}name}$. When a transition executes (we also use the term *fires*), operation carried out is called an *action*. It is specified after the '/' sign. For the stop transitions above, actions set $\underline{operation}$ to *stop* in order to stop the tape; the predicate specifying the value of an event is given by $\mathsf{vl}\ \underline{event\text{-}name}' = value$; $\underline{ff\_direction}$ is assigned directly since it is an ordinary variable rather than an event. The difference between the two is such that after events are generated, they hold their value for a short time and then revert to being undefined; ordinary variables do not lose their value. We thus call events (such as $\underline{operation}$ or $\underline{stop}$) *event variables* and ordinary variables (such as *direction*) — *persistent variables*. df, vl and events holding values are defined in $\mu\mathcal{SZ}$.

Following the Z notation, the prime sign after variable names means values of variables after completion of an operation. For instance, $\underline{ff\_direction'} = \neg\ \underline{ff\_direction}$ means that the new value of $\underline{ff\_direction}$ is an inverse of the old one.

With respect to event generation, we consider Statemate semantics where transitions may only generate events, but cannot remove those already generated[2]. In principle, removal of events is not necessary since if a transition does not generate one, it will become undefined on its own.

Transition functions whose precondition is satisfied are further referred to as *triggered*. In order to make some functions triggered, we have to generate events and change memory; this operation is referred to as *triggering*. A transition with triggered label may only occur when a statechart is in its source state; such transitions are referred to as *enabled*. For example, if we are in the *STOP* state and press $\underline{play}$, then both the *play* and *direction* transitions will be triggered, but only *play* will be enabled.

Transitions could be triggered by time. Considering tapes no longer than C120 we could stop a tape after 1.2hr of playback since it would mean a malfunction in the tape mechanism. This could be written as

$$\begin{aligned} stop:\quad & \underline{stop} \vee \underline{tape\_end} \vee tm(\underline{play}, 4320)/ \\ & \mathsf{df}\ \underline{operation'} \wedge \mathsf{vl}\ \underline{operation'} = stop \end{aligned}$$

The $tm(\underline{play}, 4320)$ expresses an event occurring 4320 seconds after $\underline{play}$. We use the notation of Statemate for *tm*.

1.4.3 The notation for labels of transitions

In the rest of the thesis, except for the hi-fi case study, we use the abbreviation for notation such that an event name on its own means $\mathsf{df}\ event$ if used on the left of / and event generation ($\mathsf{df}\ event'$) — on the right. For instance, $a \wedge b/c \wedge d$ stands for $\mathsf{df}\ a \wedge \mathsf{df}\ b/\mathsf{df}\ c' \wedge \mathsf{df}\ d'$. Events with values are not abbreviated.

Predicate or event expressions such as $aa \wedge bb$ can also be used on transitions. This shortcut can be confusing since aa in $aa/$ could mean either an event or a predicate named aa. We assume usage of / to mean the former.

[2]This is necessary for Def. 6.3.9 on p. 175.

This notation for labels of transitions is used throughout the thesis since it is seemingly easier to understand for very simple labels than the X-machine one. Complex ones are given in the notation of $\mu\mathcal{SZ}$ as it is thought by the author to be more clear for the representation of the examples considered.

Often, we used the term 'label' to express both the name of a transition and its functionality. Which of the two is meant will be clear from the context.

1.4.4 Connectors

Statecharts may contain transitions with much functionality in common. Special graphical *connectors* can be used to make a statechart more understandable; using **C**, **S** and junction connectors it is possible to separate common parts. In the example, shown in Fig. 1.5, the **C** connector is used in state *REW_FF* to enter an appropriate state when requested by a user. To

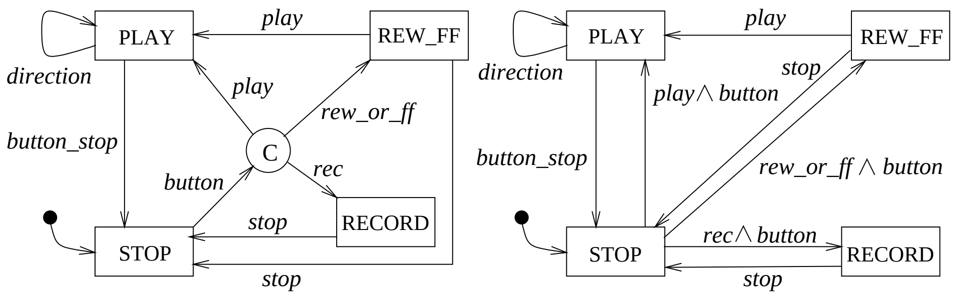


Figure 1.5: The tape recorder with a **C** connector

avoid drawing all that is shown on the right, including event expressions on transitions, we can use the connector shown on the left. This can make the statechart easier to understand and modify. If, for example, we decided to change the trigger of *button*, only one transition would have to be modified for the statechart on the left of Fig. 1.5, compared to three for the right part of the figure.

The choice connector **C** allows us to enter it with a transition and continue from it using any of outgoing transitions, depending on labels on them. In our example, we enter the connector when a user presses a button and then follow the transition corresponding to the button pressed. The **S** one is similar to the choice connector. The junction connector can be viewed as the reverse of the above two; we can have a number of transitions entering a junction and one leaving.

Transitions can be assembled from parts separated by connectors as shown in Fig. 1.5; they are called *compound transitions* (abbreviated *CT*). For example, the effective transition entering the *REW_FF* state from the

STOP one on the left of Fig. 1.5 is shown on the right of the same figure between those states, with label *rew_or_ff* $\wedge$ *button*.

Since the considered three connectors are such that we can only enter them with one and only one transition, and proceed by taking another one and only one transition, they are referred to as *OR* connectors.

In addition to **C**, **S** and junction connectors, forks and joints can be used to structure parts of transitions entering concurrent states. These connectors are described in Sect. 1.4.6 on p. 19.

History connectors allow the system to remember a state (or a number of states) in a statechart within some state upon an exit of that state and afterwards be able to return to it. These connectors are considered in Sect. 3.5 on p. 60.

Diagram connectors are provided in Sect. 3.7 on p. 62.

1.4.5 State hierarchy

An example of a state hierarchy is shown in Fig. 1.6. There is a statechart in

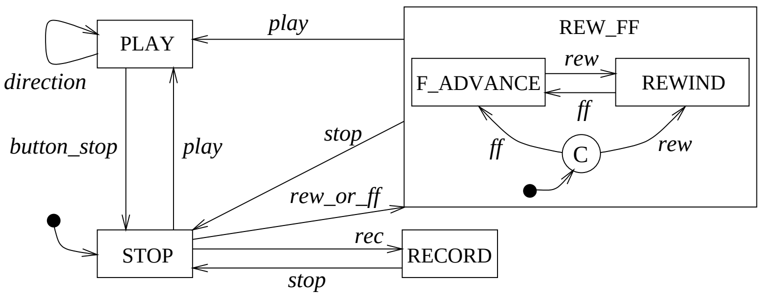


Figure 1.6: The tape recorder with a substate statechart

the state *REW_FF* which describes a behaviour of the tape recorder when it is in that state. When we enter *REW_FF* by taking the *rew_or_ff* transition, the transition terminates at the border of *REW_FF* and does not lead to any of the states inside. The blob indicates the beginning of a *default transition* which is taken in this case. Usually it just points at some state to be entered. Our case is more complicated as the default transition enters a **C** connector such that the state eventually entered depends on the button pressed by a user. A state containing a statechart is referred to as an *OR* state while that without any — a *basic* one. The transition to the *REWIND* state consists of two parts: the one entering *REW_FF* and the one entering *REWIND*. The first one is a compound transition but it is not split using **C**, **S** and junction connectors. Compound transitions going from states are called *initial compound transitions* and those going from default connectors — *continuation compound transitions* because when taken, they must follow initial compound transitions.

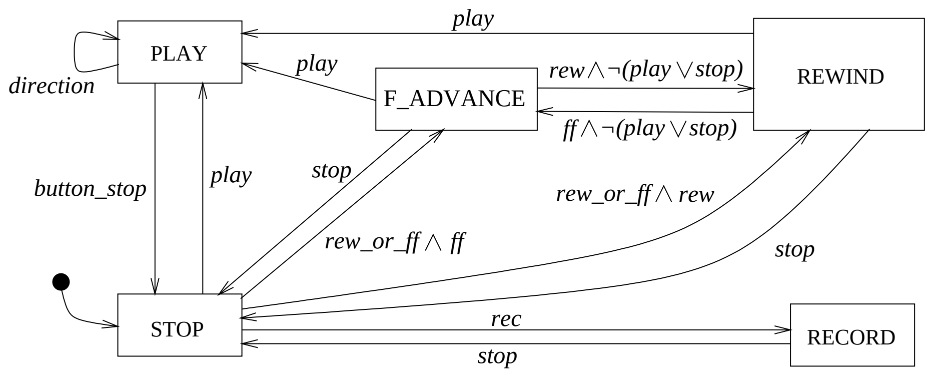


Figure 1.7: The tape recorder with the flattened statechart

A statechart within a state (we call it a *substate statechart*) is left when a transition from a state it is in, is taken. For example, if we take the *play* transition, the *REW_FF* state is left regardless of the state, *F_ADVANCE* or *REWIND*, we were in. The equivalent statechart to that in Fig. 1.6 is shown in Fig. 1.7 where the hierarchy and connectors of Fig. 1.6 are removed. To do that, the state *REW_FF* has to be replaced by its contents. The two outgoing transitions *play* and *stop* and the incoming *rew_or_ff* one need to be replaced by the four corresponding transitions. Hierarchy of states imposes priorities on transitions; to retain these priorities, transitions between *F_ADVANCE* and *REWIND* have been appropriately modified in Fig. 1.7. Furthermore, all transitions of the statechart are free of connectors.

Transitions in Fig. 1.7 represent those which are taken during steps in the original statechart in Fig. 1.6 because, according to the semantics of statecharts [NH95], while taking transitions, we expect to enter basic states. Such transitions are called *full compound* (abbreviated *FCT*) and consist of an initial compound transition followed by a number of continuation CTs. In the case where a statechart has no connectors, like the one in Fig. 1.7, all its transitions are full compound. At any moment, we can take only one full compound transition in every concurrent part of a statechart design (described further in Sect. 1.4.6 on p. 19).

Fig. 1.8 shows a statechart with a number of transitions, on the left of Fig.1.9 compound transitions of it are shown and on the right of Fig.1.9 — full compound ones. The blob above the choice connector in Fig. 1.8 is a junction one.

Informally, sets of states which are left and entered by full compound transitions are called *configurations*. It is formally defined in Def. 6.1.3 on p. 111. Sequences of transitions (not necessarily full compound or even those which could be taken) are called *paths* and given in Def. 6.1.65 on p. 155.

Realistic designs may involve constructions of statecharts with many

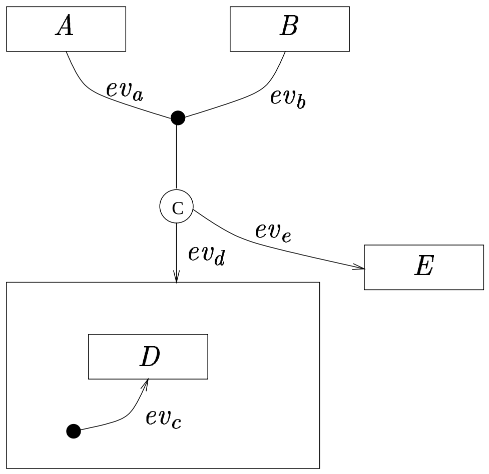


Figure 1.8: An illustration of OR-connectors

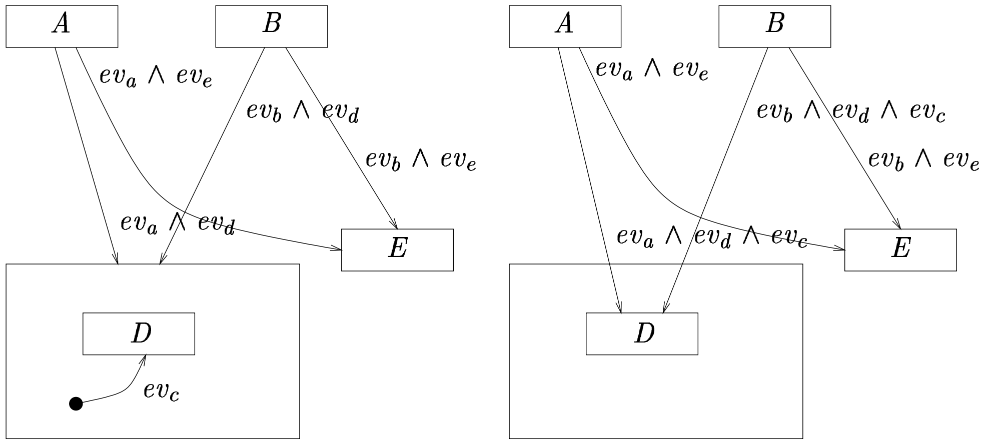


Figure 1.9: Compound and full compound transitions of Fig. 1.8

states and transitions. These could be split into a number of statecharts, by drawing the contents of an OR-state separately. Diagram connectors are syntactical elements which allow us to have transitions between such separately drawn statecharts which are parts of the same big one.

Restating what was said above, full compound transitions are considered to consist of compound transitions, which terminate at or go from default connectors. Compound transitions, in turn, consist of transitions separated by **C**, **S**, junction, fork and joint connectors. This notation is slightly different from [NH95] where compound transitions are considered to consist of transition segments; the described notation is used because it seems to be more appropriate to the author. In the test case generation we consider only compound and full compound transitions; connectors listed above are expected to be removed. For this reason, in subsequent chapters we can also refer to a compound transition as 'transition'; transitions which comprise a CT are then (infrequently) referred to as 'individual transitions'. When mainly focusing on FCTs, compound transitions they consist of are referred to as 'individual CTs'.

1.4.6 Concurrency

Statecharts have constructs to express concurrency. For example, consider an extension of the tape recorder which provides a search facility. A user can get the tape to advance forward or backward not only when the tape recorder is idle but also when it is playing or recording. The statechart is depicted in Fig. 1.10. States containing concurrently executing statecharts are called *AND* states. In this statechart we can have more than one full compound

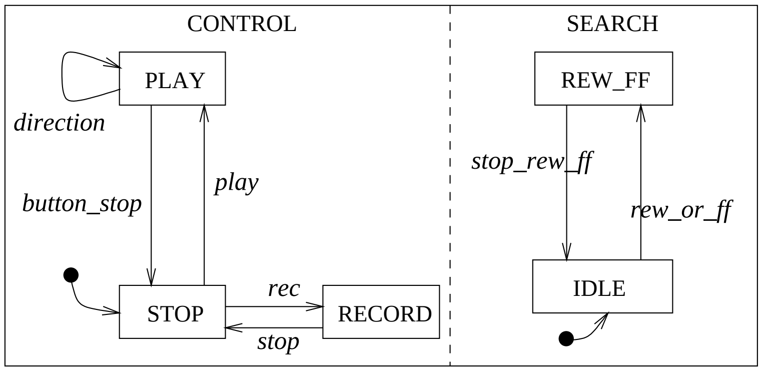


Figure 1.10: The tape recorder with the search function

transition execute concurrently, provided they reside in concurrent states; for example, we can take *button_stop* and *stop_rew_ff*.

Fork and joint connectors, referred to as AND connectors, can be used similarly to **C**, **S** and junction connectors in order to structure transitions entering or exiting a number of concurrent states. The joint connector is used for exiting such a state, fork — for entering one. An example of usage of a fork connector is shown in Fig. 1.11. The upper part is an enhancement of our tape recorder where a user can press *button_play* from a standby mode to make the device play. In the case the timer has not been set up, it will also enter the *set_time* state where a user would be able to set the time. At the bottom of the figure the equivalent statechart is presented. Dashed forked lines in the bottom statechart represent the full compound transition $button\_play \wedge \neg\ time\_set$ while solid lines — $button\_play \wedge time\_set$. If we enter a fork connector, we must take all outgoing transitions; for a joint one, we have to take all incoming ones to enter it. Note that the usage of AND connectors as described relies on the fact that we consider compound transitions to go from or enter multiple states.

1.4.7 Broadcast communication

Statecharts feature broadcast communication, meaning that an event generated by some transition is accessible to labels in the whole statechart. For example, changes in the *direction* could trigger transitions in order to change 'A' to 'B' on the status display of the tape recorder as well as, possibly, a *stop* transition if recording was in progress but the new side is write-protected.

1.4.8 Static Reactions

Statecharts considered are always *complete*, i.e. their behaviour is defined for every input. If there is no transition enabled from a given state, a *static reaction* in that state may be executed. An explicit static reaction can be defined for a state using a trigger/action pair similar to that of transitions. The difference between the two is that a static reaction does not leave or enter any state and thus its execution involves no continuation CTs.

If no transition or static reaction is enabled, nothing happens. In this case we can assume that there is an implicit static reaction which does not do anything. We call such a static reaction a 'do nothing' one. In the case of the tape recorder, 'do nothing' static reactions are in all the states since the controller is essentially idle until an event combination triggering a transition occurs.

In a more complicated model of a tape recorder we could have a counter. It can be incremented/decremented in static reactions of states $PLAY$, $RECORD$ and REW_FF (the direction of tape movement is given by the *ff_direction* variable) and zeroed when tape is removed in the $STOP$ state. The trigger for them has to involve a timeout, implying that these static reactions are executed periodically while we remain in appropriate states. The reason for this is that a tape is expected to be moving all the time when the system is not in the $STOP$ state. Such explicit static reactions are

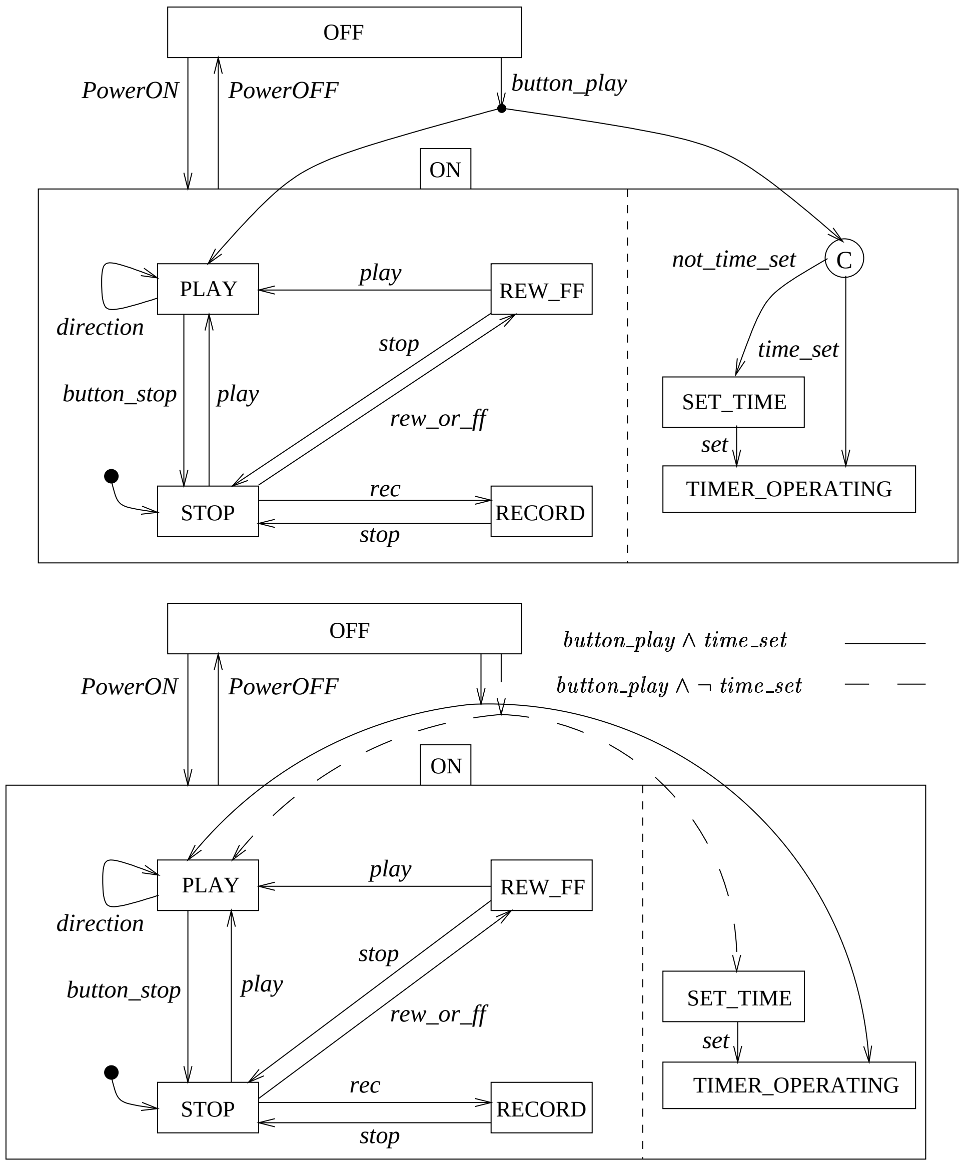


Figure 1.11: Fork connectors and their expansion

depicted in Fig. 1.12. The static reaction named *counter* can be defined as

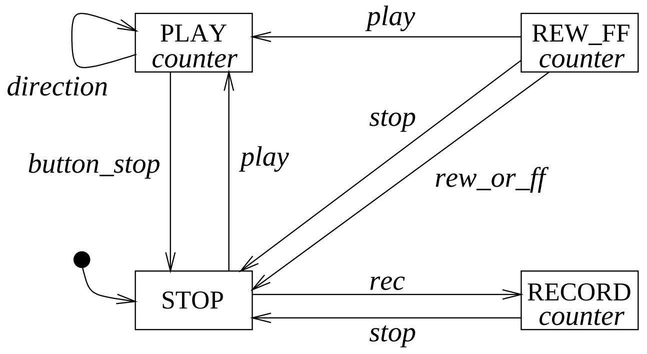


Figure 1.12: The tape recorder with a counter using static reactions

$$\begin{aligned} counter: \quad & \mathsf{df}\ tm(counter\_event, delay)/ \\ & tape\_counter' = tape\_counter + tonumTapeDirc\ \underline{ff\_direction} \wedge \\ & \mathsf{df}\ counter\_event' \end{aligned}$$

Here we use the counter_event event to cause *counter* to be executed periodically. *tonumTapeDirc* converts the direction of tape movement to 1 or -1.

1.4.9 History connectors

For a substate statechart, history connectors allow us to remember which state we were in when we exited the statechart. Later, when we return back, we enter this last state. For example, we could remember the state in the *REW_FF* statechart and return to the correct tape movement state after a pause. Deep history connectors are a variation of those described which deal with the whole hierarchy under a given state including substate statecharts, substate statecharts of substate statecharts etc. Instead of a single state, they record a configuration.

1.4.10 Step semantics

In this section we describe the Statemate step semantics given in [NH95]. Observed difference between that provided and implemented in Statemate is noted in Sect. 1.4.10 on p. 23.

Harel's (Statemate) statecharts follow one of the two types of semantics: asynchronous and synchronous. We consider them in turn.

Asynchronous step semantics

Assume that an environment the statechart is running in generates some events or changes variables which enable transitions, for example, a user could press the 'play' button. We then essentially take all enabled full compound transitions and static reactions we can. Such execution of enabled transitions is called a *step*. These transitions may in turn generate events and make changes to variables. During the step we collect all changes and apply them after the step has ended; all events active in the step which were not generated again such as $\underline{\mathit{play}}$ are discarded. Consider *play* in Fig. 1.10 to make changes which trigger *stop_rew_ff*. According to step semantics, *stop_rew_ff* will become enabled in the next step. Additionally, since the $\underline{\mathit{play}}$ event was removed, transition *direction* will not occur in the next step. The same procedure is applied for the following step.

After a number of steps, no transitions or static reactions will be enabled and a *superstep* is considered to be finished. A superstep is a reaction of a statechart to an external stimulus; in asynchronous step semantics it is assumed to complete instantaneously to an environment the statechart is operating in, i.e. much faster than an environment may notice. In principle, infinite supersteps are possible but are not useful. This may happen, for instance, if a static reaction is always enabled, but will not occur in Fig. 1.12 since *counter* refers to time which does not advance during a superstep.

Transitions taken in a step are those which are enabled and do not *conflict*[3]. For example in Fig. 1.6, if *rew* and *stop* transitions are both enabled, we take *stop* because it goes on a higher level in a state hierarchy. We say that *stop* has a higher priority than *rew*. Also, *stop_rew_ff* and *button_stop* can be taken at the same time in Fig. 1.10 because they reside in concurrent states. Conflicting transitions are those which are enabled and have the same priority such as *rec* and *play* from the *STOP* state; this implies nondeterminism which we prohibit (Req. 1b on p. 79).

Specifics of Statemate step semantics

Implementation of statecharts in the Statemate tool has a minor difference from the semantics described here, specifically, it makes an empty step when no transition or static reaction occurs at the end of a superstep. This, however, does not change the observed behaviour and thus can be ignored.

Synchronous step semantics

In this step semantics a system's reaction to the environment is a single step rather than a complete sequence of them. Events at every step are

[3]Refer to Def. 6.1.30 on page 131 for the formal definition.

those which were generated at the previous step and those generated by the environment.

This semantics does not involve a possibility of a statechart entering an infinite sequence of steps within the same time moment. Since a rather limited number of transitions occur in response to stimuli by environment, it makes statecharts easier to manipulate and observe. For this reason, statecharts are required to exhibit this semantics during testing (refer to Sect. 5.2.7 on p. 95 and Sect. 6.6 on p. 201 for details).

The two types of semantics of statecharts

Unlike many other types of semantics based on instantaneous feedback [PS91], Statemate semantics can be considered to exhibit both conjunctional and compositional parts while [PS91] only has a conjunctional one. Here conjunctional semantics means that we consider all transitions taken as a conjunction. As such, conflicting assignments to variables are disallowed which also eliminates possibilities for an effect of a transition execution to be contradictory to its cause (as in $a/\neg\, a$, [vdB94]). Statemate semantics is conjunctional within a step which means that an order of execution of parts of full compound transitions in the same step does not matter since all those parts are expected not to overwrite each other's changes. On the superstep level, semantics is compositional: the behaviour of a statechart is a composition of steps it consists of. With appropriate restrictions on transitions, they will exhibit the same behaviour in both semantics. Generalising on this, a new semantics of statecharts can be developed which could then be restricted to comply with a least two semantics, Statemate and [Per95]. Here we omit the description of this generalisation, called 'Independence Theory' (the title reflects that transitions that produce the same behaviour in both conjunctional and compositional semantics, are called *independent*). It is expected to be completed and published in the future.

1.4.11 An introduction to $\mu\mathcal{SZ}$

$\mu\mathcal{SZ}$ notation has been developed in order to combine formal and informal approaches [BG98, GHD98, Web96, BGGK97] in the design of safety-critical reactive systems. $\mu\mathcal{SZ}$ uses Statemate semantics of statecharts but data transformations are specified in Z unlike Statemate which uses a combination of a special notation with a Pascal-like programming language.

$\mu\mathcal{SZ}$ considers software systems to consist of interconnected *process classes.* Data flow and aggregation between them is specified in the *structural view*, which corresponds to activity-charts in Statemate. Behaviour of every process class is given by a statechart; Z is used to express data and its transformations. The former aspect of behaviour is given by the *dynamic view* and the latter — the *data view.*

Consider the tape recorder provided above in Sect. 1.4.1 on p. 12. The communication with users and the tape controller is handled through interfaces called ports, which can be described as follows:

taperecorder

PORT Keyboard

$\underline{play}, \underline{stop}, \underline{rew}, \underline{ff}, \underline{rec} : \mathsf{signal}$

INPUT $\underline{play}, \underline{stop}, \underline{rew}, \underline{ff}, \underline{rec}$

The port definition is placed inside the process class *taperecorder* it is a part of. *INPUT* declares the above events as inputs to the tape recorder.

The mechanism handling tapes can be given the following commands:

taperecorder

$OPERATIONTYPE ::= stop \mid play \mid move \mid record$

The boolean type, used by the $\underline{ff\_direction}$ variable, does not exist as such in Z; for this reason, we define a free type provided below:

$\mathbb{B} ::= TRUE \mid FALSE$

Then we can write

taperecorder

PORT Controller

$\underline{ff\_direction} : \mathbb{B}$

$\underline{operation} : \text{event } OPERATIONTYPE$

$\underline{tape\_end} : \mathsf{signal}$

INPUT $\underline{tape\_end}$

signal above is a μSZ boolean event type, i.e. $\underline{tape\_end}$ is an event variable which does not have a value when defined. Unlike it, $\underline{operation}$ does; its value in this case is a command to a tape drive.

With the provided definition of $\underline{ff\_direction}$, we have to express assignments and negations of it explicitly, for instance $\underline{ff\_direction'} = \neg\ \underline{ff\_direction}$ from Sect. 1.4.2 on p. 13 has to be rewritten as

$$\begin{aligned} \underline{ff\_direction} = TRUE &\Rightarrow \underline{ff\_direction'} = FALSE \\ \underline{ff\_direction} = FALSE &\Rightarrow \underline{ff\_direction'} = TRUE \end{aligned}$$

We can specify transitions using Z schemas as follows:

taperecorder

stop

$\Xi Keyboard$

$\Delta(\underline{operation})\,Controller$

$(\mathsf{df}\ \underline{stop} \lor \mathsf{df}\ \underline{tape\_end}) \land \mathsf{df}\ \underline{operation}' \land \mathsf{vl}\ \underline{operation}' = stop$

Ξ above means that the *stop* transition is using keyboard events but is not generating any (and it cannot anyway due to the *INPUT* declaration above). $\Delta(\underline{operation})\,Controller$ means that only $\underline{operation}$ event is modified, i.e. generated by the *stop* transition and everything else in the *Controller* port is unaffected. Other transitions could be specified similarly.

Data of the tape recorder, such as a counter, can be defined as

taperecorder

DATA counterDATA

$counter : \mathbb{N}$

INIT counterDATA

$counterDATA$

$counter = 0$

Counter is defined in the schema *counterDATA* which is marked to be a part of a data space of a class *taperecorder* using the *DATA* keyword. The initial value of data is described in the schema marked with *INIT*.

μSZ contains many other elements, most of which we shall not consider in this thesis. Understanding the differences between μSZ and Z is not important for understanting the rest of the thesis.

1.5 Summary of the notation used

When talking about a statechart, 'main statechart' means an enclosing statechart. Consider the *TAPE_RECORDER* state in Fig. 1.13; when talking about Fig. 1.6, we would refer to it as the main statechart[4].

The term 'substate statechart' is used to denote a statechart consisting of immediate substates of a given state. Which state is meant will be clear from the context. For example, *REW_FF* is a substate statechart of *TAPE_RECORDER* in Fig. 1.13 and of the main statechart in Fig. 1.6. For substate statechart *F_ADVANCE*, *REW_FF* can be considered as the main statechart.

[4]It is essentially a synonym to the *root* state of the state hierarchy as described in Sect. 6.1 on p. 108 but when we talk about merging rules, main statechart refers to a parent of some state which would not generally be the root one.

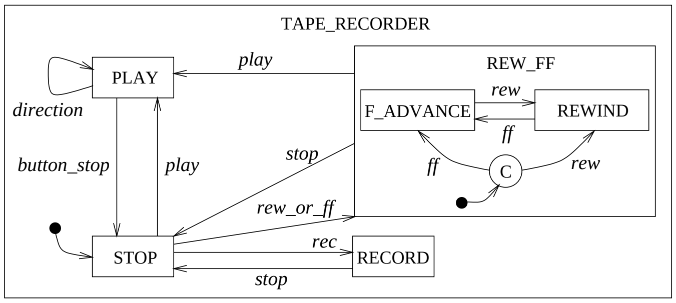


Figure 1.13: The *TAPE_RECORDER* state of the tape recorder with a substate statechart

In the thesis, we use terms automaton and finite automaton as a synonym to a finite-state machine. An acceptor is used to refer to an automaton without outputs.

'X-machine' can be used to refer to simple statecharts (this is made possible by Prop. 6.3.4 and Th. 6.3.10) and *extended finite-state machines (EFSM)* , mentioned in [TS95, LFHH].

URLs of the form `http://(espress/...)` refer to the private area of the BSCW server used by some members of the ESPRESS project.

In predicates we assume intuitive priorities of operations, such that $\wedge$ taking precedence over $\vee$; $\Leftrightarrow$ and $\Rightarrow$ having one of the lowest priorities. For set operators $\cup$, $\cap$ and $\setminus$ we assume equal priority of them and left-associativity. Most of the time, brackets are used to remove ambiguity and end-of-line conjunctions separate unrelated predicates. The number of Z constructs utilised is kept to a minimum, for instance, λ, μ and schema expressions were generally avoided.

Definitions end with the horizonal rule (__________), proofs of theorems and propositions — the box ($\Box$) symbol.

Other notational details are given in:

Sect. 1.2 on p. 9 — overview of formal notations and abbreviations of references to different parts of the thesis,

Sect. 1.4.1 on p. 12 — usage of underlining, italics, capitalisation and the typewriter font to refer to input/output variables, transitions, states and TestGen implementation - related details respectively,

Sect. 1.4.2 on p. 14 — usage of Statemate notation for the timeout function,

Sect. 1.4.3 on p. 14 — abbreviations of transition labels,

Sect. 1.4.5 on p. 19 — usage of the 'individual' term.

Chapter 2

Test case generation for simple statecharts

In this chapter we describe the testing method and how it can be applied to simple statecharts which do not contain state hierarchy and concurrency. Such statecharts (Def. 6.3.1) are behaviourally-equivalent to X-machines (follows from Prop. 6.3.4 and Th. 6.3.10). For this reason, the testing method for X-machines can be used for simple statecharts without changes. Test data generation is described in Chap. 4.

2.1 Introduction

The testing method for X-machines [IH97] has Chow's W method [Cho78] as a foundation. The testing method is based on a separation of function and transition diagram testing. It concentrates on testing of the transition diagram; transitions of an X-machine are assumed to be tested before, which can be done, for example, using the disjunctive normal form (DNF) approach (Sect. 2.3.2 on p. 38). As a result of the testing not revealing faults, the implementation[1] is proven to be behaviourally-equivalent to the designed one. In this chapter we describe the testing method for simple statecharts which is the same as that for X-machines.

According to the method, we test every transition by visiting every state and generating events to trigger it from that state. Then we should check the state that that transition has led us to, against the designed one. For example, to test the *play* transition from the state *REW_FF* to *PLAY* in Fig. 1.4, we should:

- Starting from the initial state *STOP*, enter *REW_FF* by generating $\underline{rew}$. The $\underline{operation}$ should change to *move*, assuming that it means a command to a tape drive to begin tape movement.

[1]We assume it can be modeled as an EFSM.

- Trigger *play* by generating <u>*play*</u> and observe the <u>*operation*</u> changing to *play*.
- Test that we entered the *PLAY* state. This can be done by generating the <u>*tape_end*</u> event to trigger *direction* as transition *direction* exists only from the *PLAY* state. After triggering it we should observe the modification in the <u>*ff_direction*</u> variable.

Note that for every executed transition we need to observe certain output changes which make us confident that we executed the right transition.

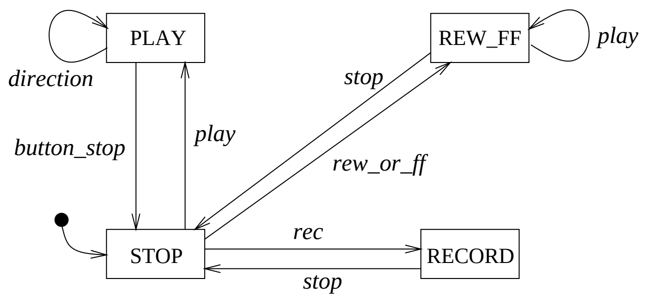


Figure 2.1: Faulty implementation of the tape recorder

For example, if the implementation is faulty as shown in Fig. 2.1, the *play* transition does not change a state from *REW_FF* to *PLAY*. Thus when trying to trigger *direction* with the <u>*tape_end*</u> event after invoking *play*, we invoke the *stop* transition. It will produce an output which is different from the output of *direction* (<u>*ff_direction*</u> will not change) and we shall be able to report the fault.

In addition to testing *play* between those two states, we need to test its existence between *STOP* and *PLAY* as well as to make sure that it does not emanate from any other state. The latter test is needed because in a faulty implementation the transition *play* could exist from some state other than *REW_FF* and *STOP*. To perform this test, we need to visit *PLAY* and *RECORD* states and generate the <u>*play*</u> event. This should invoke the *direction* transition in *PLAY* and a 'do nothing' type of static reaction in *RECORD*.

2.2 Test Case Generation

During test case generation for statecharts, we treat them as finite state acceptors with inputs being transitions labels. The result of the test case generation is a set of sequences of transition names such as *rew_or_ff play*. In order to construct it, we shall need to build auxiliary sets. There are three of them:

Set of transitions (denoted by Φ) is the set of transition labels of a statechart. We need to know it since the test method tests an equivalence between transition structures of the design and an implementation. For our example,

$$\Phi = \{\mathit{stop}, \mathit{play}, \mathit{rec}, \mathit{rew\_or\_ff}, \mathit{direction}, \mathit{button\_stop}\}.$$

State cover (denoted by C). To perform testing, we need to visit all states. A state cover C is a set of sequences of transition labels, such that we can find an element from this set to reach any desired state starting from the initial one. For our example this will be

$$C = \{1, \mathit{play}, \mathit{rew\_or\_ff}, \mathit{rec}\}.$$

Here we use 1 to denote an empty sequence of transitions. In the Tab. 2.1 the list of states is shown together with the corresponding elements of C. If the state *RECORD* is reachable only by going through

State	Sequence
STOP — the initial state	1
PLAY	*play*
REW_FF	*rew_or_ff*
RECORD	*rec*

Table 2.1: The state cover for the tape recorder example

the state *PLAY*, C would be

$$C_{rp} = \{1, \mathit{play}, \mathit{rew\_or\_ff}, \mathit{play}\ \mathit{rec}\}.$$

Characterisation set (denoted by W). A characterisation set allows us to check the state arrived at when triggering some transition, i.e. if it is the one expected. Above, in Sect. 2.1 on p. 28, we had to check if we arrived at the *PLAY* state or not. We did that by trying to follow a *path* (a sequence of transitions) which exists from *PLAY* and not from any other state. For every pair of states, we can construct a path which exists from one of them and not from the other. For example, for states *STOP* and *REW_FF* we may select *stop* as it exists from *REW_FF* and not from *STOP*. Note that the *play* transition exists from both *STOP* and *REW_FF* and thus cannot distinguish them. Such sequences for every pair of states comprise a characterisation set. In our case,

$$W = \{\mathit{stop}, \mathit{play}\}.$$

Each element of this W is a sequence consisting of a single transition.

A more complex W has to be constructed for a statechart depicted in Fig. 2.2. In order to distinguish between B and C, we have to try $b\,b$. Thus $W = \{a, b\,b\}$.

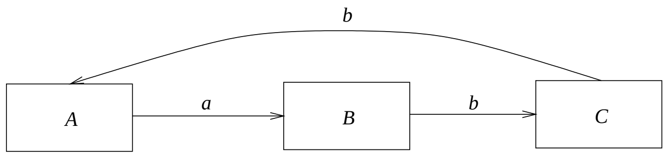


Figure 2.2: An example of a statechart with its W containing sequences of labels

With sets Φ, C and W, we can construct a set of test cases to be

$$T \quad = \quad (C \cup C * \Phi) * (\{1\} \cup \Phi \cup \Phi^2 \cup \ldots \cup \Phi^{m-n}) * W \qquad (2.1)$$

where

n is the number of states in a design (4 in our example),

m is the maximum expected number of states in the implementation, to be described later in this section,

$C \cup C * \Phi$ is referred to as a transition cover,

The set multiplication operation $A * B$ for sets of sequences A, B is defined to be $A * B = \{a\,b \mid a \in A, b \in B\}$, $a\,b$ means a concatenation of sequences a and b.

For testing we further assume that the design of a system does not contain redundant states (having the same behaviour as some others), or those with no transitions leading to them. We call this property *minimality*. m above refers to the maximal number of states in a minimal implementation. We do not require the implementation to possess this property. It is still possible to estimate the number of states in a behaviourally-equivalent one which does satisfy it.

If we consider testing a faulty implementation which may have one or more missing states, it is possible to assume that the maximal number of states in such an implementation is at most n. With $m = n$,

$$T = (C \cup C * \Phi) * W.$$

This test set deals with visiting every state (the C part of the transition cover) and verifying it (by applying W). We are then trying every transition from it ($C * \Phi$ part of the transition cover) and checking the expected state (by applying W).

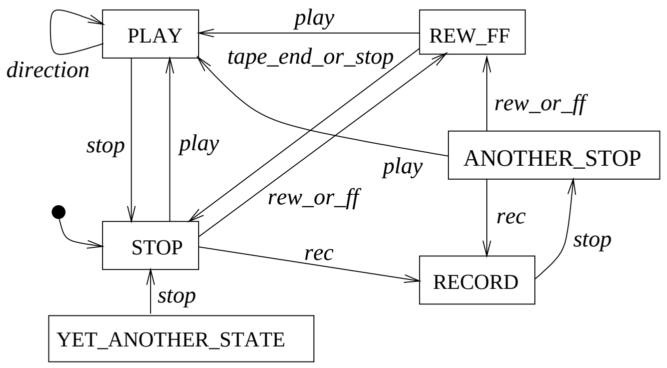


Figure 2.3: A possible implementation of the sample tape recorder

The way of testing implementations which may contain more states than the appropriate design, $m > n$ is illustrated below. An implementation in Fig. 2.3, has 6 states but m is 4 as the state *yet_another_state* is unreachable and *another_stop* is behaviourally equivalent to *stop*. Note that this statechart cannot be used as a design for our method.

Consider the faulty statechart as shown in Fig. 2.4. The transition from

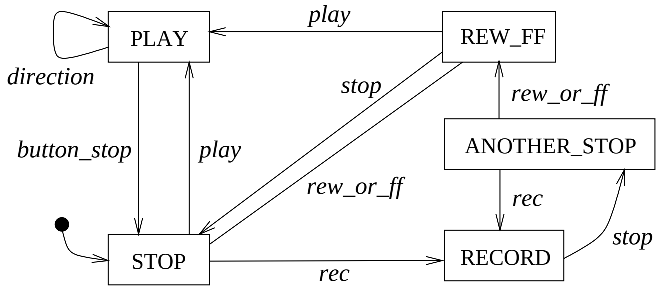


Figure 2.4: The faulty implementation with extra states

$ANOTHER\_STOP$ to $PLAY$ is missing. The extra state $ANOTHER\_STOP$ is reachable only from the $RECORD$ one. When testing transitions of this statechart using $(C \cup C * \Phi) * W$, we try them all from all states we can reach using the state cover. Consequently, we do not try all transitions from $ANOTHER\_STOP$. In order to cope with extra states we have to try sequences of more than one transition, such as all pairs of transitions, from every state. From the state $RECORD$ one of the pairs of transitions to be tried will be *stop rew_or_ff* which will not bring us to the REW_FF state as expected. Thus a fault will get revealed. The part of the set of test cases where we try all pairs of transitions can be expressed as $C * \Phi * \Phi * W$. In the case where we assume more than one extra state, we need to make these sequences of transitions longer. For the possibility of $(m - n)$ extra states we arrive at Eqn. 2.1 for generating sets of test cases which can be rewritten as

$$T = C * (\{1\} \cup \Phi \cup \Phi^2 \cup \ldots \cup \Phi^{m-n+1}) * W \qquad (2.2)$$

The size of the set of test cases can be computed (following [Ipa95]) as a function of n, assuming[2] $\mid \Phi \mid > 1$, as follows:

$$\begin{aligned} Size_T &= \mid C * (\{1\} \cup \Phi \cup \ldots \cup \Phi^{m-n+1}) * W \mid \\ &= \mid C \mid * \mid (\{1\} \cup \Phi \cup \ldots \cup \Phi^{m-n+1}) \mid * \mid W \mid \\ &= \mid C \mid * \frac{\mid \Phi \mid^{m-n+2} - 1}{\mid \Phi \mid - 1} * \mid W \mid \\ &\leq \mid C \mid * \mid \Phi \mid^{m-n+2} * \mid W \mid & (2.3) \\ &\leq n * \mid \Phi \mid^{m-n+2} * (n-1) \\ &\leq n^2 * \mid \Phi \mid^{m-n+2} & (2.4) \end{aligned}$$

The size of C has to be n since we visit every state with it; the maximal size of W is $n-1$ (Sect. 2.4.2 on p. 42).
In practice, however, $\mid \Phi \mid$ would be much greater than 1 and thus $Size_T$ is more likely to be at the order of

$$n^2 * \mid \Phi \mid^{m-n+1}$$

For $m = n$ in practice,

$$Size_T \leq n^2 * \mid \Phi \mid$$

For our tape recorder, we get $Size_T = 4 * (1 + 6) * 2 = 56$ (here and further, numbers for sizes of sets of test cases are computed using precise formulas) under the assumption that $m = n$. The actual set of test cases generated by the TestGen tool uses $W = \{play, rew\_or\_ff, direction\}$ and contains $4 * (1+6) * 3 = 84$ sequences; after removal of those, which coincide with beginnings of others, 63 remained. Tab. 2.2 provides these sequences.

2.3 Related testing methods

2.3.1 Finite-state machine testing methods

Here some of the known testing methods for deterministic finite-state machines will be outlined. They could potentially be extended for statecharts in order to make a set of test cases smaller. An application of the *Wp* testing method is described in App. C on p. 276.

References to publications in descriptions of methods refer to papers in which a considered method is provided and not necessarily the one where it was first published.

[2]For a set S, we use $\mid S \mid$ to mean $card(S)$, i.e. the number of elements in S.

1	*rec*	*play*	*play*
1	*rec*	*rew_or_ff*	*play*
1	*rec*	*direction*	*play*
1	*play*	*play*	*play*
1	*play*	*rew_or_ff*	*play*
1	*play*	*direction*	*play*
1	*rew_or_ff*	*play*	*play*
1	*rew_or_ff*	*rew_or_ff*	*play*
1	*rew_or_ff*	*direction*	*play*
1	*direction*	*play*	
1	*rec*	*rec*	*play*
1	*rec*	*rec*	*rew_or_ff*
1	*rec*	*rec*	*direction*
1	*rec*	*play*	*rew_or_ff*
1	*rec*	*play*	*direction*
1	*rec*	*stop*	*play*
1	*rec*	*stop*	*rew_or_ff*
1	*rec*	*stop*	*direction*
1	*rec*	*button_stop*	*play*
1	*rec*	*button_stop*	*rew_or_ff*
1	*rec*	*button_stop*	*direction*
1	*rec*	*direction*	*rew_or_ff*
1	*rec*	*direction*	*direction*
1	*rec*	*rew_or_ff*	*rew_or_ff*
1	*rec*	*rew_or_ff*	*direction*
1	*play*	*rec*	*play*
1	*play*	*rec*	*rew_or_ff*
1	*play*	*rec*	*direction*
1	*play*	*play*	*rew_or_ff*
1	*play*	*play*	*direction*
1	*play*	*stop*	*play*
1	*play*	*stop*	*rew_or_ff*
1	*play*	*stop*	*direction*
1	*play*	*button_stop*	*play*
1	*play*	*button_stop*	*rew_or_ff*
1	*play*	*button_stop*	*direction*
1	*play*	*direction*	*rew_or_ff*
1	*play*	*direction*	*direction*
1	*play*	*rew_or_ff*	*rew_or_ff*
1	*play*	*rew_or_ff*	*direction*
1	*rew_or_ff*	*rec*	*play*
1	*rew_or_ff*	*rec*	*rew_or_ff*
1	*rew_or_ff*	*rec*	*direction*
1	*rew_or_ff*	*play*	*rew_or_ff*
1	*rew_or_ff*	*play*	*direction*
1	*rew_or_ff*	*stop*	*play*
1	*rew_or_ff*	*stop*	*rew_or_ff*
1	*rew_or_ff*	*stop*	*direction*
1	*rew_or_ff*	*button_stop*	*play*
1	*rew_or_ff*	*button_stop*	*rew_or_ff*
1	*rew_or_ff*	*button_stop*	*direction*
1	*rew_or_ff*	*direction*	*rew_or_ff*
1	*rew_or_ff*	*direction*	*direction*
1	*rew_or_ff*	*rew_or_ff*	*rew_or_ff*
1	*rew_or_ff*	*rew_or_ff*	*direction*
1	*stop*	*play*	
1	*stop*	*rew_or_ff*	
1	*stop*	*direction*	
1	*button_stop*	*play*	
1	*button_stop*	*rew_or_ff*	
1	*button_stop*	*direction*	
1	*direction*	*rew_or_ff*	
1	*direction*	*direction*	

Table 2.2: The set of test cases for the simple tape recorder in Fig. 1.4

Most of considered testing methods are based on the assumption that an implementation cannot have more states than specification. Although it is quite restrictive in general, those methods have been successfully used in testing of network protocols [SL89, SLD90, Ber94, RDT95b].

It is questioned in [FvBK$^+$91] if it is good to limit the number of possible extra states in an implementation. Although all existing methods assume that, combining multiple test sequences into one is recommended as often faults are detected only by relatively long sequences of inputs. It means that instead of resetting the system after application of a test sequence to it, we take a transition to the expected initial state or to the one in which inputs should be applied. Usage of this technique requires a machine to be strongly connected. Minimisation of the length of such a path is described in [SLD90, Ura92, Hie97b].

It is suggested in [FvBK$^+$91] for an implementation to contain methods of state identification and/or special transitions to bring the implementation into the required state. The identification could be implemented in a form of 'loopback' transitions leaving and entering the same state and having a unique output for every state. This is similar to the approach of using status information for state identification (App. C.4 on p. 282). In principle, sequences of transitions used for state identification have also to be verified. This is done in [Ura92] for the UIOv method and in [FvBK$^+$91] for the Wp one.

Usually, W and Wp methods are described regarding an implementation potentially having more states than specification and others — having the same or less. It is claimed [RDT95b] that they could probably be easily extended for that.

Unlike the rest of the thesis, in this section we use terms 'specification' and 'implementation'.

Transition tour method

The *transition tour method* (TT) [SL89] method is the most simple among those to be described and provides the shortest test sequences and the least coverage. Using this method, one just traverses all transitions in the specification without trying to identify target states. An efficient algorithm has been proposed to generate a minimum-length sequence in [Ura92].

Distinguishing sequence method (DS)

We begin with the following definition:

Definition 2.3.1 (distinguishing sequence). *An input string x is said to be a* distinguishing sequence *of a finite-state machine $\mathcal{M}$ if the output sequence produced by $\mathcal{M}$ in response to x is different for each state.*

Using the distinguishing sequence (DS) method [SL89, Ura92], we traverse states applying all possible inputs and checking the resulting states via DS, for every state. We could use reset transitions to bring a machine into the initial state after applying DS. Alternatively, it is possible find the minimal length tour to cover the required sequences of transitions.

Unique input output sequence method

UIO method [SL89] involves deriving a *unique input-output (UIO)* sequence for each state of a machine to be tested. It is a sequence of inputs and expected outputs which reflects the behaviour of only that state. Consequently, states reached can be checked by application of the UIO for the expected state. The testing method involves construction of test sequences such that every transition in an automaton is followed by a UIO for the expected entered state.

The size of a test set generated by this method can be reduced by construction of a minimal-length tour containing all necessary sequences [Ura92, **SUIO** method].

An approach of using adaptive distinguishing sequences was proposed [LY94]. Such sequences can be much shorter than UIO ones, applicable to a wider range of them and there is a more memory-efficient way of constructing adaptive distinguishing sequences than that for UIO ones [LY94].

UIOv method

This is an improvement of the UIO one. As in a faulty implementation UIO derived for a design may loose its state identification properties, an improvement, called UIOv, is presented in [Ura92]. The method consists of the following steps:

1. Application of the UIO for a state after the machine has been brought to that state. This verifies that the implementation contains all states of its specification.

2. For every state, the UIO for a different state is applied. This helps to ensure state identification capability of UIO sequences for the implementation machine (for those states where respective UIOs have different input parts).

3. Transitions not checked in the state identification part above, are checked as given in the description of the UIO method.

MUIO method

In the UIO method (Sect. 2.3.1 on p. 36), we have a single UIO sequence to verify a state; instead, we could be able to construct a number of them.

During construction of a tour, these sequences can be used to reduce the overall length of the resulting test sequence [SLD90].

A significant improvement of this method using invertible sequences of transitions has been described in [Hie96, Hie97b]. A path ending at some state is called *invertible* if it is the only one with the specific sequence of input/output pairs on it, to enter that state. This property of invertibility helps us verify states as well as merge test sequences. Consequently, the length of the test set can be reduced in comparison to testing without consideration of invertibility.

DD method

The brief description of this method is from [RDT95b]. We consider implementations with at most one fault and satisfying the so-called TULD property. The method is claimed to find all faults.

The DD method uses UIO sequences to identify states and a state cover C to reach all of them. During testing, we need to verify applicability of both C and UIOs to an implementation. This is hindered by transitions shared between C and UIO sets, since in order to verify transitions in one of them, we need to use another one. FSMs satisfying the TULD property are such that no transitions are shared between them. The authors claim most network protocols satisfy it.

The main advantage of the considered method is that its ability to locate a fault, i.e. the one 1-fault resolution capability (Def. 4.4.1 on p. 78) is the best of all finite-state machine methods considered.

SW method

This method, outlined in [FvBK$^+$91], is a modification of the original W one. After an application of a test sequence, SW-method avoids using resets, instead taking a sequence of transitions to enter the desired state. No proof of its ability to detect all faults is given.

C method

The characterising sequence method [RDT95a] is a modification of the DS one (Sect. 2.3.1 on p. 35), introducing a set of sequences to identify a state. This set is similar to a W_i one of the Wp method (App. C.3 on p. 281) but unlike it no extra states are considered and the state identification capabilities W_i are not verified.

Comparison between the methods described

It is possible to express coverage of the implementation w.r.t design achieved by every of described methods [RDT95b]. The results are as follows (denoting by '<' sign the weakness relation between methods and '=' — equivalent fault detection capability of them),

$$TT < UIO < C < DS = W = Wp$$

$$UIO = SUIO = MUIO < UIOv < W$$

It is clear that TT is the weakest method. The UIO method uses a sequence of transitions to enter a source state of a transition to be tested without verifying elements of this sequence, while the C method does [RDT95a]. From construction of UIOv, it is better than UIO but still below the *W* method (Chap. 2 on p. 28).

2.3.2 Disjunctive Normal Form Approach

A set of input data supplied to a program is not often uniformly treated by it. For this reason, it is commonplace to split the domain of the system under test into a number of non-intersecting parts and generate a test case for each of them [Mye79, Rop94]. This approach to testing is often called category partition testing. This is a rather informal method assuming that behaviour of the program in every partition of its domain is such that either its response to every element of it is correctly implemented or it is not to all of them. This allows us to reason about the correctness of an implementation based on a single test case for every partition. Since in practice this assumption is seldom true, a number of test cases can be generated.

When used for the Z notation, partition testing is referred to as *Disjunctive Normal Form (DNF)* testing method [HNS, Meu98]. [DF93, Hie97c] consider constructing an automaton from partitions and generating test sequences from it.

Comparison between X-machine and DNF testing methods

A simple example Consider an X-machine in Fig. 2.5.

We introduce variables s and s' to designate old and new states for a transition, $\underline{e_1} \ldots \underline{e_4}$ for input events and $\underline{ee'_1} \ldots \underline{ee'_4}$ for output ones. Transition functions $f_1 \ldots f_4$ can be written as follows:

$$\begin{aligned}
f_1 &: \underline{e_1} \wedge \underline{ee'_1} \wedge s = 1 \wedge s' = 2 \\
f_2 &: \underline{e_2} \wedge \underline{ee'_2} \wedge s = 2 \wedge s' = 2 \\
f_{3_1} &: \underline{e_3} \wedge \underline{ee'_3} \wedge s = 2 \wedge s' = 3 \\
f_{3_2} &: \underline{e_3} \wedge \underline{ee'_3} \wedge s = 3 \wedge s' = 3 \\
f_4 &: \underline{e_4} \wedge \underline{ee'_4} \wedge s = 3 \wedge s' = 1
\end{aligned}$$

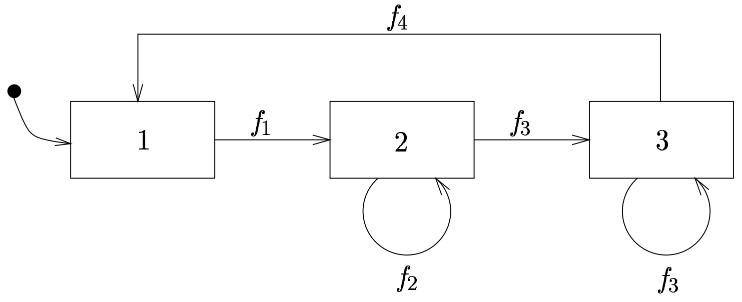


Figure 2.5: The sample machine to compare Z and X-machine testing methods

Combining the above, the machine can be expressed as:
$\underline{e_1} \wedge \underline{ee_1'} \wedge s = 1 \wedge s' = 2 \vee \underline{e_2} \wedge \underline{ee_2'} \wedge s = 2 \wedge s' = 2 \vee$
$\underline{e_3} \wedge \underline{ee_3'} \wedge s = 2 \wedge s' = 3 \vee \underline{e_3} \wedge \underline{ee_3'} \wedge s = 3 \wedge s' = 3 \vee$
$\underline{e_4} \wedge \underline{ee_4'} \wedge s = 3 \wedge s' = 1$

When applying the DNF approach, we pick all the disjuncts. Consider the second of them, $\underline{e_2} \wedge \underline{ee_2'} \wedge s = 2 \wedge s' = 2$. In order to take this one, we have to enter state 2, generate $\underline{e_2}$ and verify the s'. Note that we have to have some way to make s become 2 initially and an approach to verification of it at the end. This is the purpose of C and W sets in the W method. Due to its data-orientation, the DNF approach lacks explicit mechanisms to achieve this.

The main differences between the DNF and X-machine-based approaches are summarised below:

1. DNF does not try all the transitions from all states, i.e., disjuncts like $\underline{e_3} \wedge \underline{ee_3'} \wedge s = 1 \wedge s' = 1$. Also, extra states are not checked for.

2. DNF does not verify if transitions occurred or not, i.e. $\underline{e_2} \wedge s = 2 \wedge s' = 2$ can potentially be selected for a test with the DNF method even though this transition does not produce any output and thus we cannot tell it apart from the 'do nothing' static reaction. As a result, we cannot always be sure whether the expected transition occurred or not.

3. In order to enter a state, we cannot always set a variable because a state of a system is generally a product of a subset of its data space and a subset of a set of control states. In order to force the control into some state, something like C has to be applied. This problem is not the case with DNF since all variables are believed to be in the data subset.

As a result, we can see that while useful for testing data transformations, DNF falls short of specialised methods for testing transition structures.

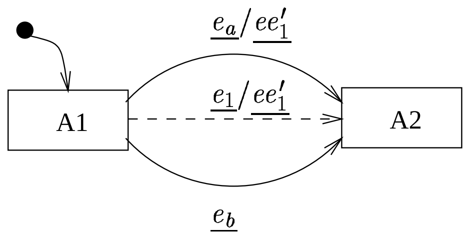

$\underline{e_a}/\underline{ee'_1}$
$\underline{e_1}/\underline{ee'_1}$
A1
A2
$\underline{e_b}$

Figure 2.6: Testing individual transitions

A more complicated example In the example above, transitions did not perform any data transformations. Consider a more complex example, where

$$f_1 = ((\underline{e_1} \vee \underline{e_a}) \wedge \underline{ee'_1} \vee \underline{e_b}) \wedge s = 1 \wedge s' = 2$$

Here DNF would be

$$\begin{aligned}&\underline{e_1} \wedge \underline{ee'_1} \wedge s = 1 \wedge s' = 2 \vee \\ &\underline{e_a} \wedge \underline{ee'_1} \wedge s = 1 \wedge s' = 2 \vee \\ &\underline{e_b} \wedge s = 1 \wedge s' = 2\end{aligned}$$

This type of testing could be accomplished by an extension of our testing method where we DNF-partition the operation of a transition. We could then enter a state from which this transition emanates and try all disjuncts. The approach has similarity to the CFTT testing method (Sect. 2.3.5 on p. 42).

With this approach we could:

1. apply the test method described in the thesis, to the most of the transition structure, using inputs/outputs, selected or introduced for testing, which exercise 'trusted' disjuncts of every transition.

2. test the rest of the disjuncts by

   - entering a state from which they are going (and since the previous stage completed, expected transitions actually do go from that state),
   - try disjuncts and check outputs,
   - verify the entered state (since they could lead to different states).

The described approach is depicted in Fig. 2.6. Dashed transition represents the part of the transition which is exercised by the test method for the most of the transition structure. Disjuncts on the top and bottom are tested after that. A problem with the approach is that it could be difficult to ensure that non-dashed transitions are not triggered during testing of the dashed ones.

The implementation of the statechart testing method supports marking some transitions to be ignored for testing and thus could be utilised to support the first part of the method, where after splitting every transition into a number of disjuncts, only trusted ones are allowed to participate in testing.

The described approach may also be applicable to testing of systems where implementation does not comply with testing requirements, for instance,

- transitions may get split into a number of them. With the knowledge of possible splitting (this may follow from definitions of transitions) and which of the resulting transitions can be assumed to be correctly implemented, we can apply the described method reasonably directly.

- a state may get split into a number of states (described in Sect. 8.2 on p. 232. In this case, usage of the approach leads to unreachable states in the first stage of testing. Derivation of appropriate constraints to prevent this is left for future work (p. 235).

2.3.3 Testing of process-algebraic specifications

Traditionally, concurrent systems were specified using process-algebraic languages [SVG, Mil89] and appropriate testing methods were developed for them [Pel96, PS96a, PS96b, PF90, Wez90] (potentially, these methods can be applied to testing FSMs). It is also possible to convert LOTOS specifications to extended finite-state machines [KP92] and communicating machines [Kar92]. In the former case, the X-machine testing method can be applied; for the latter, the testing methods developed for communicating automata [Hie97d] as well as the one described in the thesis (Sect. 3.3 on p. 54), could be usable. Similar work is described in [TPvB97].

An interesting model for communicating X-machines has been developed [BGG$^+$99] and the X-machine testing method was applied to it.

2.3.4 Dataflow-oriented testing for X-machines

An EFSM can be tested using a dataflow-oriented approach [BDAR97], in contrast to the X-machine one [HI98].

For every variable in a program, we have statements which assign values to it and those which use it but not modify it. A path in the considered EFSM, in the beginning of which a variable is assigned to and not modified until the end of the path, is called *du-path* (this stands for definition-usage path). With a specific coverage criterion for du-paths, we can construct a set of test cases. To complete it, tests for transitions not covered by the described paths are added.

Dataflow testing methods can also be applied to LOTOS [TS95, vdSU95]. Being informal, dataflow testing does not allow provable correctness as a result of testing.

2.3.5 CFTT approach

In [Sad98, SS98], an approach combining DNF testing with a transition tour method is presented.

The method considers automata with transitions specified in Z. They are partitioned using a DNF approach and these partitions are used to split states. After that, an automaton is constructed in which split states are connected by partitions of original transitions (this is similar to [Hie97c]). For such an automaton, a transition tour testing method is applied. While providing a small test set, the method seems to be difficult to extend to handle hierarchy and concurrency of statecharts; correct behaviour of an implementation also cannot be assured by testing.

2.4 Reduction of a test case set size

Here we give the minimal and maximal sizes of sets C and W. After that, a summary of approaches to reduce the size of the set of test cases without sacrificing the conclusions which could be made if the test set does not reveal faults, is provided. The growth of the number of test sequences could be a lot slower than that of test cases, refer to Sect. 4.2.3 on p. 74 for details.

2.4.1 Minimisation of C

It is possible to construct C in order to minimise the length of sequences in it. The minimal length of every sequence is 1; the maximal is $n-1$. Everything depends here on how closely states are linked together. If every state is connected to every other, we get the minimal length. For a statechart which is a sequence of states, $n-1$.

2.4.2 The size of a characterisation set

The characterisation set has its minimal and maximal size depend on the number of states. The minimal size is given by the $\log_2 n$ and maximal by $n-1$ where n is the number of states in the statechart considered. We describe them and also provide a way to construct a distinguishing sequence; the actual size of W and existence of a distinguishing sequence depend on a given statechart, such that considered lower bounds are purely theoretical.

The smallest W

For every pair of states, we can have to have a transition which exists from one of them and not from the other. Thus, at best we have a transition which exists from half of the states of a statechart and does not from another half. If this transition exists from more or less than a half of states, it would not be able to separate states of a statechart in two equal groups which is our intention, to make each such group as small as possible. After separating states into the two groups, we could use another transition to split those groups into two and so on. Thus we arrive at the lower bound for W being $\log_2 n$. Such division by two approach could be adopted in the method for W generation but is not currently implemented in the *TestGen* tool.

Consider, for example, a statechart in Fig. 1.4. Here transition *stop* can be considered to separate states into groups $\{PLAY, STOP\}$ and $\{REW\_FF, RECORD\}$. After that, *play* can be used to split every of these two groups of states into $\{PLAY\}$, $\{STOP\}$ and $\{REW\_FF\}$, $\{RECORD\}$. Thus, the smallest W for the tape recorder is $\{play, stop\}$, which is of the same size as the theoretical minimum described.

The biggest W

The upper bound is where the groups of states considered above are of significantly different size, meaning that one state contains only one state and the other one contains the rest of them. Consequently, every additional transition reduces the size of the big group by 1 which makes the biggest size of the W set to be $n - 1$. This is also the case when we try to use the union of sequences distinguishing individual states of a statechart from all other states, as a W.

Distinguishable sequence

Since an associated automaton is an acceptor, we can only find out whether some sequence exists or not. After test data generation, however, the sequence of test inputs will terminate after the first transition which does not exist and we would be able to see how 'much' of our sequence exists. Consequently, the length of it could be used to distinguish states. From one state the first element of it would not exist, from the second one — the first would but the second would not and so on. Such a W is easy to construct by a simple extension of the method described above. After we find a sequence which exists from one group of states and not from another, we could then extend the existing sequence such that it would split states of the first group. This sequence could then be extended further. The approach for the construction of optimal sequences can be considered for future work.

Summary

In addition to C and W minimisation methods above, optimisations to reduce the size of a test set are given in Sect. 4.2.2 on p. 73; an application of the Wp method [FvBK+91] is outlined in App. C.3 on p. 281. Here we provide a summary of additional approaches to the reduction of test set size and/or improvement of the speed of test set generation and application:

- Instead of resetting a machine each time a sequence from a W set has been applied in order to apply another one, we could use transfer transitions from the arrived state, which seems to be very similar to the SW method[3] (Sect. 2.3.1 on p. 37).
- Usage of invertible transitions and sequences of them [Hie96, Hie97b].
- Removing unreachable states in the case of testing concurrent statecharts using state multiplication (Chap. 3.3 on p. 54). Such a removal is quite a complicated operation, involving the determination of whether some path can be followed and in general is undecidable.

2.5 Verification and validation of statecharts

Usually, verification of a design should involve checking for the required properties such as those needed by the testing method (Chap. 5 on p. 79). Such checking could be done using a variety of model-checking methods developed for statecharts (Sect. 1.1.7 on p. 8) and/or by theorem proving. Validation can be done by running a simulation of the design and making sure it does what is expected from it. The validation necessity was the reason why the hi-fi model (Sect. 8.3 on p. 236) was built in pure Statemate and then translated into $\mu\mathcal{SZ}$.

The Statemate Analyser tool [Ilo95a] provides an automated approach to verification of certain properties of statecharts. This is accomplished by exhaustive search by supplying the statecharts with all possible sequences of inputs and checking its behaviour. A panel presenting the test complexity is displayed when such a test is executed. The complexity is described in terms of a function (such as a logarithm) of a number of possible system configurations to be evaluated. Users can set the maximum one as well as predefined values of external variables, which are not changed by the analyser during analysis. A part of a statechart model can be analysed separately too.

There are two possible (interchangeable) approaches to usage of Statemate Analyser: filling a form where all conditions to look for are specified, and using a *watchdog* statechart. A watchdog statechart is a statechart executed

[3]The paper describing it is written in Japanese which the author is not familiar with; and instead followed a single paragraph in [FvBK+91].

in parallel with a design under test to monitor and drive it as well as access internal elements of it. There could be several watchdogs. As an example of its usage, it is possible to have an error state included in a watchdog statechart which will be entered when the system has arrived in an unsafe state.

Although the test method for statecharts described in this chapter was developed to test an implementation against its design, it can potentially be applied at any level of abstraction. We could even try to test a design to correspond to a specification or try to validate the design. In this case, test sequences would be applied to a model and its behaviour would be checked for compliance with requirements or user needs. In some sense this is useful since designers would likely to build the system under the assumption of 'sane' usage and an application of a test set could reveal undesired behaviour. Additionally, this approach could be easier to apply than formal verification; on the other hand, we would have to do a hard job finding legitimate inputs to follow transitions we desire instead of relying on design for test. This follows from the fact that we are trying to verify/validate behaviour of the system here rather than assessing a transition diagram.

Chapter 3

Test case generation for complex statecharts

In the previous chapter we have described testing of simple statecharts. Here we consider more complicated ones. Every feature of statecharts will have a section devoted to it.

The most simple approach to testing state hierarchy and concurrency deals with flattening a statechart, i.e. turning it into a behaviourally-equivalent one without substate statecharts and AND states. For example, Fig. 1.7 depicts a result of flattening of the statechart in Fig. 1.6. Default connectors are also removed after flattening, leaving a simple (Def. 6.3.3 on p. 168) but, in practice, huge statechart. Generation of a set of test cases for it is an easy operation as described in Chap. 2 on p. 28.

As an alternative to the above, we propose an approach of incremental test case development. It has the advantage of following the design process and thus providing a possibility of updating the set of test cases to reflect design changes made.

In order to test state hierarchy and concurrency, we begin with the generation of a tuple $(\Phi, C, W, \mathbf{DE})$ with the first three members described in Chap. 2 on p. 28 and the last one, $\mathbf{DE}$, helping to handle default transitions, described later in Sect. 3.2.1 on p. 48. $(\Phi, C, W, \mathbf{DE})$, called *test case basis* (abbreviated TCB), is generated for the main statechart and every non-basic state considering all its substates as basic ones. Afterwards, we combine (we say 'merge') the tuples in a way described below. From the resulting tuple $(\Phi, C, W, \mathbf{DE})$, a set of test cases can be constructed following the Eqn. 2.2 in Sect. 2.2 on p. 29 or a slightly modified one (Eqn. 3.6), to follow.

3.1 Testing of statecharts with connectors

In order to test an implementation of the tape recorder shown in Fig. 1.5, left, we can remove all **C**, **S**, junction, fork and joint connectors and construct *compound transitions* as shown in Fig. 1.5, right. It essentially means that we wish to eliminate all connectors for testing and treat all transitions as 'simple'. Such removal of connectors described is possible due to Req. 4a on p. 81. The testing methods described further rely on only default connectors present in the design. The process of connector removal is rather mechanical and can be easily automated; for this reason absence of all connectors apart from default ones is not listed in the requirements for the testing method (Chap. 5 on p. 79).

3.2 Hierarchy — OR-states

3.2.1 General Approach.

Here the case when no interlevel transitions exist, is considered. Element **DE** of a test case basis will be covered later. The tape recorder example is used to illustrate the approach.

Merging rules

We start traversing states in both REW_FF and the main statechart from the default connector; with that, the elements of the test case basis for REW_FF are given by:

$$\begin{aligned}
\Phi_{REW\_FF} &= \{ff, rew\},\\
C_{REW\_FF} &= \{rew, ff\},\\
W_{REW\_FF} &= \{ff\}.
\end{aligned}$$

The elements of the test case basis for the main statechart of the tape recorder (considering the state REW_FF to be a basic one) are as follows:

$$\begin{aligned}
\Phi_{MAIN\,STATECHART} &= \{play, stop, direction, rec, rew\_or\_ff, button\_stop\},\\
C_{MAIN\,STATECHART} &= \{1, 1\,play, 1\,rec, 1\,rew\_or\_ff\},\\
W_{MAIN\,STATECHART} &= \{stop, play\}.
\end{aligned}$$

To get the elements of the resulting tuple (Φ, C, W), we propose the following merging rules for the above elements with explanation to follow:

$$\begin{aligned}
\Phi &= \Phi_{MAIN\,STATECHART} \cup \Phi_{REW\_FF}, && (3.1)\\
C &= C_{MAIN\,STATECHART} \cup \\
&\quad \{\text{path in } C_{MAIN\,STATECHART} \text{ to enter } REW\_FF\} *_1 C_{REW\_FF}\\
&\quad \setminus \{\text{path in } C_{MAIN\,STATECHART} \text{ to enter } REW\_FF\}, && (3.2)\\
W &= W_{MAIN\,STATECHART} \cup W_{REW\_FF}. && (3.3)
\end{aligned}$$

Above, multiplication $*_1$ is similar to concatenation $*$, but the last element of the first sequence and the first element of the second one are united such that, for example, $\{1\,\mathit{rew\_or\_ff}\} *_1 \{\mathit{rew\,ff}\} = \{1\,\mathit{rew\_or\_ff\text{-}rew\,ff}\}$; refer to Sect. 6.1.6 on p. 155 for details. Proofs are given in Chap. 6.

Merging sets Φ we obtain:

$$\begin{aligned}
\Phi^{merged} &= \Phi_{MAIN\,STATECHART} \cup \Phi_{REW\_FF} \\
&= \{\mathit{play}, \mathit{stop}, \mathit{direction}, \mathit{rec}, \mathit{rew\_or\_ff}, \mathit{button\_stop}\} \cup \{\mathit{ff}, \mathit{rew}\} \\
&= \{\mathit{play}, \mathit{stop}, \mathit{direction}, \mathit{rec}, \mathit{rew\_or\_ff}, \mathit{button\_stop}, \mathit{ff}, \mathit{rew}\}
\end{aligned}$$

The path in $C_{MAIN\,STATECHART}$ to enter REW_FF is the one to the default connector of REW_FF, i.e. $1\,\mathit{rew\_or\_ff}$. With this, merging sets for C gives us:

$$\begin{aligned}
C^{merged} &= C_{MAIN\,STATECHART} \cup \\
&\quad \{\text{path in } C_{MAIN\,STATECHART} \text{ to enter } REW\_FF\} *_1 C_{REW\_FF} \\
&\qquad \setminus\{\text{path in } C_{MAIN\,STATECHART} \text{ to enter } REW\_FF\} \\
&= \{1, 1\,\mathit{play}, 1\,\mathit{rec}, 1\,\mathit{rew\_or\_ff}\} \cup \{1\,\mathit{rew\_or\_ff}\} *_1 \{\mathit{rew}, \mathit{ff}\} \\
&\qquad \setminus\{1\,\mathit{rew\_or\_ff}\} \\
&= \{1, 1\,\mathit{play}, 1\,\mathit{rec}, 1\,\mathit{rew\_or\_ff}\} \cup \{1\,\mathit{rew\_or\_ff\text{-}rew}, 1\,\mathit{rew\_or\_ff\text{-}ff}\} \\
&\qquad \setminus\{1\,\mathit{rew\_or\_ff}\} \\
&= \{1, 1\,\mathit{play}, 1\,\mathit{rec}, 1\,\mathit{rew\_or\_ff\text{-}rew}, 1\,\mathit{rew\_or\_ff\text{-}ff}\}.
\end{aligned}$$

In the set C above, transitions which have to be taken in the same step, such as *rew_or_ff-rew* to enter the REW_FF state, are separated by dashes. The path to the default connector of REW_FF is removed since its presence in $C_{MAIN\,STATECHART}$ is irrelevant because REW_FF is no longer considered a basic state.

The generation of W involves simply uniting the sets,

$$\begin{aligned}
W^{merged} &= W_{MAIN\,STATECHART} \cup W_{REW\_FF} \\
&= \{\mathit{stop}, \mathit{play}\} \cup \{\mathit{ff}\} \\
&= \{\mathit{stop}, \mathit{play}, \mathit{ff}\}
\end{aligned} \tag{3.4}$$

Note that transitions of Φ^{merged} are not all FCTs, for example, *rew_or_ff* is included in it. Expansion of Φ, C and W to make their transitions full compound by addition of default continuations, is explained in the following.

Consideration of default transitions

After merging, sets Φ, C and W may contain some non-full compound transitions which we have to expand to full compound ones. This is necessary for the test set we later generate to be applicable to an implementation. The element **DE** (stands for 'default entrances') of a TCB is the set of sets of labels which correspond to all possible default completions for a transition entering the state. For our tape recorder, $\mathbf{DE}_{REW\_FF} = \{\{rew\}, \{ff\}\}$. Usage of this set helps us to resolve the stated difficulty.

For our tape recorder, the requirement (Req. 4b on p. 81) means that an implementation may contain transitions *rew_or_ff-rew*, *rew_or_ff-ff*, but not *play-rew* because *play* does not enter *REW_FF* state in the correct implementation. Consequently, when expanding labels in Φ and W into those corresponding to full compound transitions, only those which label transitions leading to states with default connectors in the designed statechart have to have them added. For the tape recorder, we would expand transitions entering *rew_or_ff* but not *play*. If an erroneous transition goes to an unexpected state, such as *play* to *REW_FF*, it would have a number of default continuation transitions implicitly with no triggers on them, due to the stated requirement.

Since default transitions cannot be present on their own, but only as part of full compound transitions and we assume correct implementation of those transitions (Req. 4b), we do not have to include them separately in Φ. Labels of default transitions are present in Φ_{REW_FF} of the tape recorder because these labels are also used on non-default transitions.

In general, we cannot simply replace transitions in Φ and W with their full compound equivalents. The reason for this is that the continuation transition taken depends on the state and there is no state information in TCB. The necessity of expansion of transitions follows from the statechart in Fig. 3.1. In order to distinguish states A and B, we can use sequence

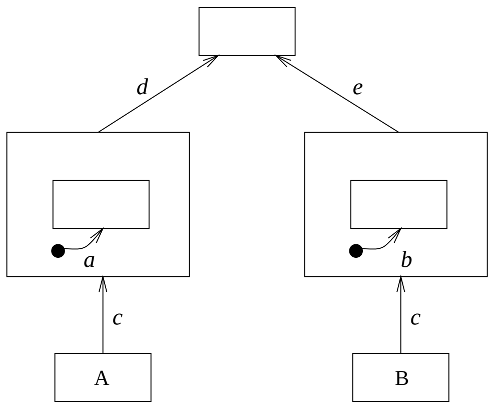


Figure 3.1: An illustration for the description of function *defaultComplete*

$c\ d$. Unfortunately, c cannot be applied in both A and B since we also have to trigger default completions. Consequently, we replace c in W with two transitions $c\text{-}a, c\text{-}b$ and get $c\text{-}a\ d, c\text{-}b\ d$ as an expansion of $c\ d$.

Similar problems occur in Φ since in the test set, we try to invoke transitions starting with those in Φ; the solution provided for W is possible in this case too.

The state cover set C is different because there we know which state we are entering and could make an appropriate entering transition full compound; for all other transitions of it we can take any possible default continuation contained in **DE** of a state entered by it. In the case of our tape recorder, only one sequence of $C_{MAIN\,STATECHART}$ enters a non-basic state and it is REW_FF where it terminates. For this reason, nothing is needed to be done for it. A sequence *rew_or_ff play* would have been expanded to *rew_or_ff-rew play*. The formal definition of merging for C (Def. 6.4.11) takes this into account.

In making transitions of Φ and W full compound, we can use the following rule:

$$expand(lbl) = \{lbl\} *_1 \mathbf{DE}_{sub_1} \cup \{lbl\} *_1 \mathbf{DE}_{sub_2} \cup \ldots \cup \{lbl\} *_1 \mathbf{DE}_{sub_k}$$

for all states sub_1, $sub_2 \ldots sub_k$ which any transition with label lbl enters. Thus,

$$rew\_or\_ff_{expanded} = \{rew\_or\_ff\} *_1 \mathbf{DE}_{REW\_FF} = \{rew\_or\_ff\text{-}rew, rew\_or\_ff\text{-}ff\}$$

since it enters REW_FF. All other transitions are unaffected by expansion; for instance, $play_{expanded} = play *_1 \mathbf{DE}_{PLAY} = \{play\}$ since the $PLAY$ is a basic one and thus $\mathbf{DE}_{PLAY} = \{1\}$.

With this, we can write the set of test cases as

$$\begin{aligned}
\Phi^{merged}_{final} &= expand(\Phi) \\
&= \{play, stop, direction, rec, rew\_or\_ff\text{-}ff, rew\_or\_ff\text{-}rew, \\
&\qquad button\_stop, ff, rew\}, \\
W^{merged}_{final} &= expand(W) = \{stop, play, ff\}
\end{aligned}$$

where *expand* when applied to a set of sequences means that every element of every sequence is expanded. The described approach to making transitions in Φ^{merged} and W^{merged} full compound is formalised in Def. 6.4.16.

Statecharts with default transitions having no label

The type of statecharts where default transitions have no trigger or action, is a considerable simplification of what was described above. Since such statecharts are believed to be rather common and most of those considered in the case studies (Chap. 8 on p. 229) satisfy this condition, we provide merging rules for statecharts whose default transitions have empty labels.

First of all, we note that due to the requirement of determinism, every OR state will contain only one default transition. As none of them will have a label, the **DE** set introduced above will contain a single set $\varnothing$ and this allows us not to consider expansion of transitions. It means that merging rules introduced in Sect. 3.2.1 on p. 47 will already generate full compound transitions.

Results

We apply merging rules in the bottom-up fashion until TCB for the root state is constructed. The resulting tuple $(\Phi, C, W, \mathbf{DE})$ can be used to generate the set of test cases for the whole system. For example, if the state *F_ADVANCE* had a substate statechart, we would have to 'merge' its test case basis with sets constructed for *REW_FF* before merging TCB of *REW_FF* with that for the main statechart.

It can be observed that the generation process above resulted in the sets possible for the flattened statechart in Fig. 1.7, i.e. C allows us to visit every state, W-to distinguish every pair of them and Φ has the beginnings of all full compound transitions. Only transitions to be triggered are included in the set of test cases; those not to be, corresponding to negated transitions in Fig. 1.7, are expected not to be triggered by the t_completeness requirement (Req. 2).

The size of the set of test cases can be calculated using Eqn. 2.4 as follows:

$$\begin{aligned} Size_T &\le (n_{MAIN\,STATECHART} + n_{REW\_FF})^2 * \\ &\quad \mid \Phi_{MAIN\,STATECHART} + \Phi_{REW\_FF} \mid^{m_{MAIN\,STATECHART}+m_{REW\_FF}-} \\ &\qquad {}^{n_{MAIN\,STATECHART}-n_{REW\_FF}+2} \\ &\le (n_{MAIN\,STATECHART} + n_{REW\_FF})^2 * \\ &\quad \mid \Phi_{MAIN\,STATECHART} + \Phi_{REW\_FF} \mid^{2*(m_{mr\,max}-n_{mr\,min})+2} \\ &\le 4 * n_{mr\,max}^2 \Phi_{mr\,max}^{2*(m_{mr\,max}-n_{mr\,min})+2} \end{aligned} \tag{3.5}$$

where

$$\begin{aligned} n_{mr\,max} &= max(n_{MAIN\,STATECHART}, n_{REW\_FF}), \\ m_{mr\,max} &= max(m_{MAIN\,STATECHART}, m_{REW\_FF}), \\ n_{mr\,min} &= min(n_{MAIN\,STATECHART}, n_{REW\_FF}), \\ \Phi_{mr\,max} &= max(\mid \Phi_{MAIN\,STATECHART} \mid, \mid \Phi_{REW\_FF} \mid). \end{aligned}$$

For our tape recorder, we get $Size_T = 5 * (1 + 9) * 3 = 150$ under assumption of $m = n$ for both main statechart and substate statechart.

3.2.2 Interlevel transitions

Above, an approach for statecharts with no interlevel transitions has been presented. Here we describe how to deal with interlevel transitions.

When constructing sets C and W of the test case basis, our requirements allow us to ignore all interlevel transitions but include them in Φ for the statechart being the lowest common OR-state ancestor of the initial and final states of the interlevel transition concerned, i.e. the *scope* (Def. 6.1.24). The merging procedure is the same as the one for statecharts without interlevel transitions.

For example, consider the statechart in Fig. 3.2 which is equivalent to the one in Fig. 1.6. The test case basis for the whole statechart will be as follows:

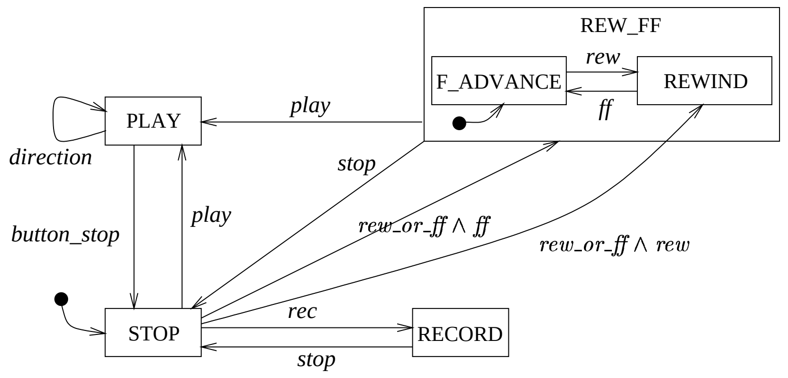


Figure 3.2: The tape recorder with an interlevel transition

$$\begin{aligned}
C_{MAIN\,STATECHART} &= \{1, 1\,play, 1\,rec, 1\,rew\_or\_ff \wedge ff\},\\
W_{MAIN\,STATECHART} &= \{stop, play\},\\
\Phi_{MAIN\,STATECHART} &= \{play, stop, direction, rec, rew\_or\_ff \wedge ff,\\
&\qquad rew\_or\_ff \wedge rew, button\_stop\}.
\end{aligned}$$

In the above, the transition to enter REW_FF is *rew_or_ff* ∧ *ff* and the interlevel transition is *rew_or_ff* ∧ *rew*. The latter is included in $\Phi_{MAIN\,STATECHART}$ but not in $C_{MAIN\,STATECHART}$ or $W_{MAIN\,STATECHART}$.

Note that due to our treatment of interlevel transitions, we cannot remove all connectors from statecharts under test (Sect. 3.1 on p. 46) as this would replace transitions entering states with interlevel ones. Since those transitions would be the only way to enter states and we ignore them, states of a substate statechart would become unreachable, rendering our method not directly applicable.

3.2.3 OR-state refinement

A design is usually constructed gradually. Initially we may have only a skeleton, then step-by-step fill it with details as shown in Fig. 3.3. The way we handle consistency between a design and implementation makes a big difference. We could either develop a design in a stepwise manner and having finished it, generate code, or we could have both design and implementation developed in parallel. The latter means that each change in a design is accompanied by an appropriate change in the implementation. In the case of parallel development, we can avoid leaving all testing to the final pre-release stage. Additionally, we can make certain kinds of implementation faults

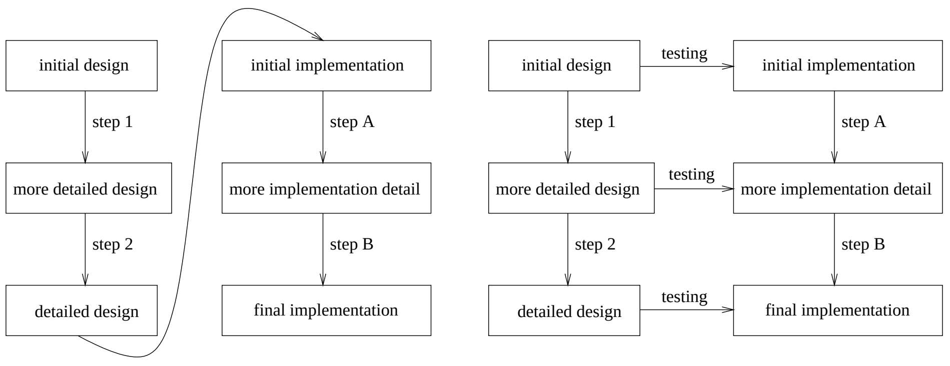


Figure 3.3: Parallel development of a design and an implementation

impossible, like the one depicted in Fig. 3.4. Its faults are that transitions *play* and *stop* are not present from the *REWIND* state and *rew_or_ff* cannot enter the *REWIND* state. These faults become impossible if initially the design in Fig. 1.4 is built and implemented and then the statechart 'placed' in the *REW_FF* state (Fig. 1.6). This is due to the *play* and *stop* transitions being on the higher level of hierarchy than transitions inside the statechart within *REW_FF* and thus they would always take precedence over those in *REW_FF*.

Similar types of refinement (master-slave in [IH98a, Ipa95] and X-machine-let one [Lay92]) additionally assume that the substate statechart does not change the internal variables other than its own. If it is not so, the substate statechart could generate the $\underline{play}$ event and thus interfere with user's intention for rewinding or fast forwarding. For our type of testing where we test the transition diagram, assuming transition labels to be correctly implemented, the assumption of non-interference is not required.

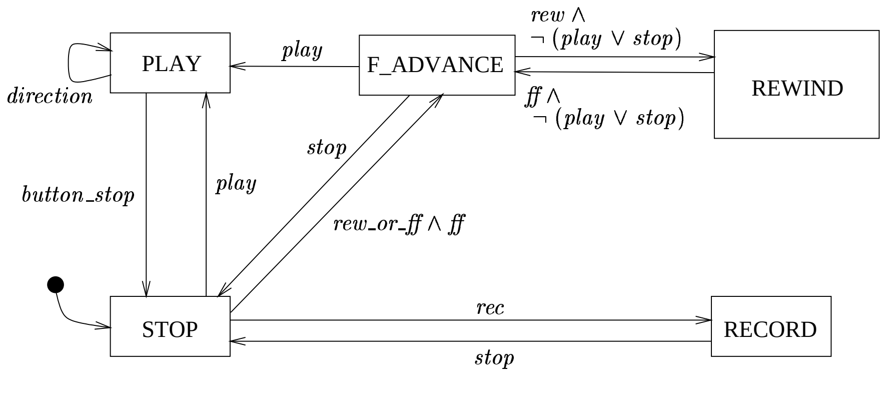


Figure 3.4: The faulty implementation which cannot happen if we keep an implementation consistent with every change in a design

With the described restrictions, the test set for such a refinement can be greatly reduced even for our simple example compared to the case when we consider the statechart in the *REW_FF* state to be 'just' a geometrical hierarchy. Faults related to not entering or exiting the state *REW_FF* are not possible; we need to consider only the set of transitions defined inside *REW_FF* when testing it. This influences the rules we use to construct the set of test cases using Eqn. 2.2 for a statechart with a refined state. The set of test cases is given below:

$$\begin{aligned} T \quad = \quad & C_{MAIN\,STATECHART} * (\{1\} \cup \Phi_{MAIN\,STATECHART} \cup \ldots \cup \Phi_{MAIN\,STATECHART}^{m_1-n_1+1}) * \\ & W_{MAIN\,STATECHART} \\ \cup & \\ & \{1\, rew\_or\_ff\} *_1 C_{REW\_FF} * (\{1\} \cup \Phi_{REW\_FF} \cup \ldots \cup \Phi_{REW\_FF}^{m_2-n_2+1}) * W_{REW\_FF} \end{aligned} \tag{3.6}$$

The first part of the union tests the main statechart treating the state *REW_FF* as a basic one. The second — enters the substate statechart (via *rew_or_ff*) and tests it separately from the main one. Numbers m_1, n_1 correspond to the expected number of states in the implementation of the main statechart and the number of states in the design of it (4 in our example); m_2, n_2 are the corresponding numbers for the statechart in the *REW_FF* state.

The size of the above set of test cases can be estimated using Eqn. 2.3 to be

$$\begin{aligned} Size_T \quad = \quad & Size_{T_{MAIN\,STATECHART}} + Size_{T_{REW\_FF}} \\ \leq \quad & n_{MAIN\,STATECHART}^2 * \mid \Phi_{MAIN\,STATECHART} \mid^{m_{MAIN\,STATECHART}-n_{MAIN\,STATECHART}+2} + \\ & n_{REW\_FF}^2 * \mid \Phi_{REW\_FF} \mid^{m_{REW\_FF}-n_{REW\_FF}+2} \\ \leq \quad & 2 * n_{mr\ max}^2 * \Phi_{mr\ max}^{m_{mr\ max}-n_{mr\ min}+2} \end{aligned} \tag{3.7}$$

We have $Size_T = 4 * (1 + 6) * 2 + 2 * (1 + 2) * 1 = 62$ (computed without approximation), which is less than a half of that computed using Eqn. 2.2 from the test case basis for our example when we do not treat a hierarchy as a refinement and do not expect extra states in the implementation ($5 * (1+9) * 3 = 150$). If we do, i.e. all m are greater compared to corresponding n, the difference can be much bigger.

3.3 Concurrency — AND-states

There are three methods of testing concurrent statecharts: via state multiplication and that with two different refinements. We consider these approaches in turn.

3.3.1 State multiplication.

Using the state multiplication approach results in an exponential state growth w.r.t a number of concurrent statecharts which could be potentially eliminated by a requirement for developers to construct models with a small number of concurrent states.

Merging rules provided below do not consider default transitions. For this reason, one has to construct the **DE** set and turn transitions into full compound as described in Sect. 3.2.1 on p. 48.

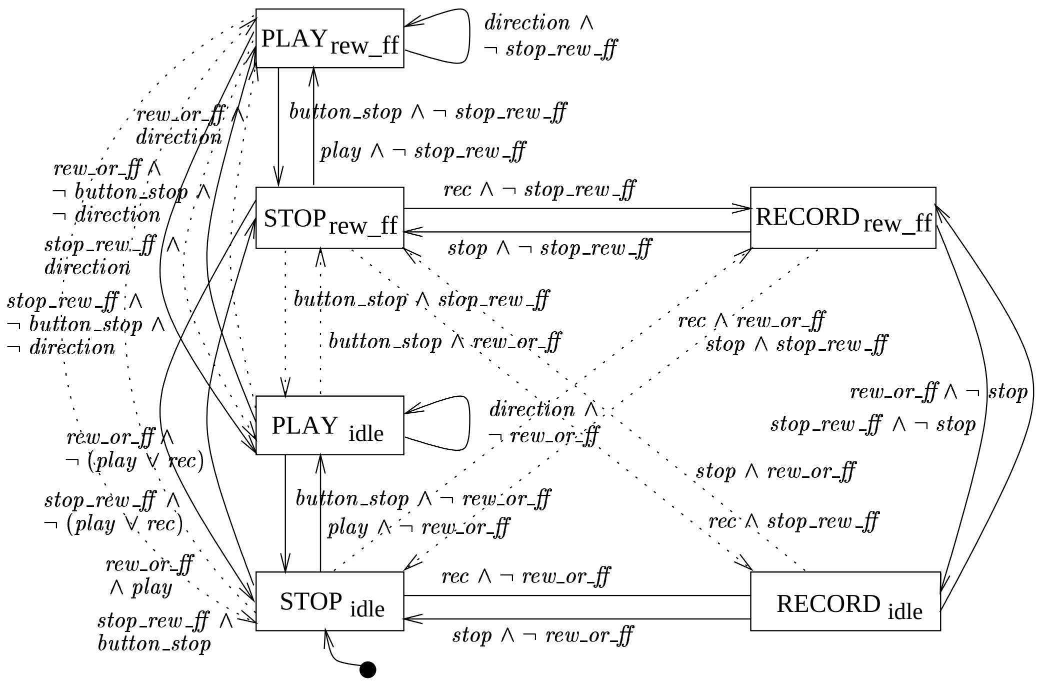


Figure 3.5: Testing AND-states via state multiplication

We begin with the generation of a test case basis for each component of the AND state in Fig. 1.10. The elements of the test case basis for the left concurrent component ($CONTROL$) are:

$$\begin{aligned}\Phi_{CONTROL} &= \{direction, stop, button\_stop, play, rec\},\\ C_{CONTROL} &= \{1, 1\,play, 1\,rec\},\\ W_{CONTROL} &= \{direction, rec\}.\end{aligned}$$

The corresponding elements for the right concurrent component ($SEARCH$) are:

$$\begin{aligned}\Phi_{SEARCH} &= \{rew\_or\_ff, stop\_rew\_ff\},\\ C_{SEARCH} &= \{1, 1\,rew\_or\_ff\},\\ W_{SEARCH} &= \{stop\_rew\_ff\}.\end{aligned}$$

To build a test case basis for the whole statechart, we need to 'merge' TCBs of individual components. In order to visit all states and consider all transitions, sets C and Φ have to be multiplied; W sets get united,

$$\begin{aligned}
\Phi &= (\{1\} \cup \Phi_{CONTROL})\underline{*}(\{1\} \cup \Phi_{SEARCH}) \setminus \{1\} = \{1, \mathit{direction}, \mathit{stop},\\
&\quad \mathit{button\_stop}, \mathit{play}, \mathit{rec}\}\underline{*}\{1, \mathit{rew\_or\_ff}, \mathit{stop\_rew\_ff}\} \setminus \{1\} =\\
&\quad \{\mathit{direction}, \mathit{stop}, \mathit{button\_stop}, \mathit{play}, \mathit{rec}, \mathit{rew\_or\_ff}, \mathit{stop\_rew\_ff},\\
&\quad \mathit{direction\text{-}rew\_or\_ff}, \mathit{stop\text{-}rew\_or\_ff}, \mathit{button\_stop\text{-}rew\_or\_ff},\\
&\quad \mathit{play\text{-}rew\_or\_ff}, \mathit{rec\text{-}rew\_or\_ff}, \mathit{direction\text{-}stop\_rew\_ff}, \mathit{play\text{-}stop\_rew\_ff},\\
&\quad \mathit{stop\text{-}stop\_rew\_ff}, \mathit{button\_stop\text{-}stop\_rew\_ff}, \mathit{rec\text{-}stop\_rew\_ff}\},\\
C &= C_{CONTROL}\underline{*}C_{SEARCH} = \{1, 1\,\mathit{play}, 1\,\mathit{rec}\}\underline{*}\{1, 1\,\mathit{rew\_or\_ff}\} =\\
&\quad \{1, 1\,\mathit{play}, 1\,\mathit{rec}, 1\,\mathit{rew\_or\_ff}, \mathit{play\text{-}rew\_or\_ff}, \mathit{rec\text{-}rew\_or\_ff}\},\\
W &= W_{CONTROL} \cup W_{SEARCH} = \{\mathit{direction}, \mathit{rec}\} \cup \{\mathit{stop\_rew\_ff}\} =\\
&\quad \{\mathit{direction}, \mathit{rec}, \mathit{stop\_rew\_ff}\}.
\end{aligned}$$

One can observe that the sets constructed are appropriate for the statechart in Fig. 3.5 (which is the flattening of Fig. 1.10 on p. 19). C allows us to visit every state, W-to distinguish every pair of them and Φ contains all initial compound transitions.

Multiplication $\underline{*}$ is used for AND-multiplication in order to produce a shortest possible sequence of transitions. We could in principle use a sequential multiplication $*$ or OR-multiplication $*_1$ instead of the concurrent one $\underline{*}$ but then the resulting sequence would be the sum of their lengths rather than the longest of them.

Sizes of elements of the merged test case basis are

$$\begin{aligned}
\mathit{Size}_\Phi &= \mid (1 + \Phi_{CONTROL}) * (1 + \Phi_{SEARCH}) - 1 \mid\\
&= (1+ \mid \Phi_{CONTROL} \mid) * (1+ \mid \Phi_{SEARCH} \mid) - 1\\
&= \mid \Phi_{CONTROL} \mid * \mid \Phi_{SEARCH} \mid + \mid \Phi_{CONTROL} \mid + \mid \Phi_{SEARCH} \mid\\
\mathit{Size}_C &= \mid C_{CONTROL}\underline{*}C_{SEARCH} \mid\\
&= \mid C_{CONTROL} \mid * \mid C_{SEARCH} \mid\\
\mathit{Size}_W &= \mid W_{CONTROL} \cup W_{SEARCH} \mid\\
&= \mid W_{CONTROL} \mid + \mid W_{SEARCH} \mid
\end{aligned}$$

In practice we expect $\mid \Phi_{CONTROL} \mid * \mid \Phi_{SEARCH} \mid$ to be much greater than $\mid \Phi_{CONTROL} \mid + \mid \Phi_{SEARCH} \mid$ and thus in the following calculations we shall approximate Size_Φ by $\mid \Phi_{CONTROL} \mid * \mid \Phi_{SEARCH} \mid$.
From Eqn. 2.3 it follows that the size of a set of test cases can be estimated be

$$\begin{aligned}
\mathit{Size}_T &\leq \mathit{Size}_C * \mathit{Size}_\Phi^{m_{\mathit{flattened}} - n_{\mathit{flattened}} + 2} * \mathit{Size}_W\\
&\approx n_{CONTROL} * n_{SEARCH} *
\end{aligned}$$

$$\begin{aligned}
& (\mid \Phi_{CONTROL} \mid * \mid \Phi_{SEARCH} \mid)^{m_{CONTROL} * m_{SEARCH} - n_{CONTROL} * n_{SEARCH} + 2} * \\
& (n_{CONTROL} + n_{SEARCH}) \\
\leq\ & max(n_{CONTROL}, n_{SEARCH})^2 * \\
& (max(\mid \Phi_{CONTROL} \mid, \mid \Phi_{SEARCH} \mid))^{2*(max(m_{CONTROL}, m_{SEARCH})^2 - min(n_{CONTROL}, n_{SEARCH})^2 + 2)} * 2 * max(n_{CONTROL}, n_{SEARCH}) * \\
\leq\ & 2 * n_{cs\,max}^3 * \Phi_{max}^{2*(m_{cs\,max}^2 - n_{cs\,min}^2 + 1)} \qquad (3.8)
\end{aligned}$$

where

$$\begin{aligned}
n_{cs\,max} &= n_{CONTROL} + n_{SEARCH}, \\
\Phi_{cs\,max} &= max(\mid \Phi_{CONTROL} \mid, \mid \Phi_{SEARCH} \mid), \\
m_{cs\,max} &= max(m_{CONTROL}, m_{SEARCH}), \\
n_{cs\,min} &= min(n_{CONTROL}, n_{SEARCH}).
\end{aligned}$$

For state multiplication, $Size_T$ grows significantly faster than that in Eqn. 2.4. For our tape recorder under assumption of an implementation containing no more states than the design, we get $Size_T = 6 * (1 + 17) * 3 = 324$.

In the following we cover the AND-states being a result of refinement and how such refinement can reduce the size of a set of test cases.

3.3.2 Communication of statecharts

Different statecharts may communicate with each other by triggering transitions in each other. It is most often used between concurrent states. Although we can test every concurrent component individually and apply some approach to their communication, interaction is expected to be tested at the level of transitions. Communication faults are that some transitions do not trigger expected ones or trigger those they should not. It could mean that an output of some label is wrong, an input of another one or the relation between shared data in concurrent components is not a bijection.

Our approach essentially requires communication to be disabled during testing since it could cause sequences of transitions to occur, contradicting Req. 3c. For some statecharts, it could be possible to test the core transition structure with transitions which do not communicate and then test the remaining ones, following the approach described in [Hie97d].

3.3.3 Weak AND-state refinement

Description

The development process where we design and implement concurrent states and then place appropriate statecharts in them (refinement), allows us to eliminate faults where the state entered (or behaviour exhibited) by some transition is different if we take a transition in the different concurrent component in the same step. For example, such a fault could involve transition *rec* erroneously entering the *PLAY* state when executed at the same time as *rew_or_ff*, and the correct state *RECORD* otherwise. For the case of refinement, we test *rec* and *rew_or_ff* but not both of them at the same time. Consequently, it is possible to merge sets Φ as

$$\Phi = \Phi_{CONTROL} \cup \Phi_{SEARCH}$$

and thus we get the following:

$$\begin{aligned}
\Phi &= \Phi_{CONTROL} \cup \Phi_{SEARCH} = \\
&\{direction, stop, button\_stop, play, rec\} \cup \{rew\_or\_ff, stop\_rew\_ff\} = \\
&\{direction, stop, button\_stop, play, rec, rew\_or\_ff, stop\_rew\_ff\}, \\
C &= C_{CONTROL} \underline{*} C_{SEARCH} = \{1, 1\,play, 1\,rec\} * \{1, 1\,rew\_or\_ff\} = \\
&\{1, 1\,play, 1\,rec, 1\,rew\_or\_ff, play\text{-}rew\_or\_ff, rec\text{-}rew\_or\_ff\}, \\
W &= W_{CONTROL} \cup W_{SEARCH} = \{direction, rec\} \cup \{stop\_rew\_ff\} = \\
&\{direction, rec, stop\_rew\_ff\}. \qquad (3.9)
\end{aligned}$$

The size of the set of test cases can be estimated to be (using the derivation of Eqn. 3.8):

$$\begin{aligned}
Size_T &\le 2 * n^3_{cs\,max} * (\Phi_{CONTROL} + \Phi_{SEARCH})^{m^2_{cs\,max} - n^2_{cs\,min} + 2} \\
&\le 2 * n^3_{cs\,max} * (2 * \Phi_{cs\,max})^{m^2_{cs\,max} - n^2_{cs\,min} + 2} \\
&= 2 * n^3_{cs\,max} * \Phi_{cs\,max}^{(m^2_{cs\,max} - n^2_{cs\,min} + 2)*(1+\log_{\Phi_{cs\,max}} 2)}. \qquad (3.10)
\end{aligned}$$

For absence of extra states in the implementation, we get $Size_T = (3*2)*(1+5+2)*(2+1) = 144$ which is 2 times less than that for multiplication of sets of transition labels (Eqn. 3.8). The transitions on the flattened statechart which we do not take, are given in dotted lines in Fig. 3.5.

Rationale

States can be considered to be implemented using variables, such as internal CPU registers like `PC` or `CS:EIP`. State refinement of a state into a number of concurrent states means that each newly created concurrent component is given a *unique state space* and a possibly non-unique data space. It changes state by modifying data responsible for the state space and communicates with other components via shared variables in the data space.

Consider the case when a number of transitions produce a different result when taken at the same time rather than when taken one-after-one (in different steps). For correct implementation of transitions and transition diagram, this implies racing which is not possible for variables which are not shared and for shared ones is prohibited by Req. 3b. For this reason, states entered by some transitions cannot depend on whatever transitions in concurrent statecharts are taken in the same step (note that only one full compound transition from every concurrent component can be taken in a step). Consequently, we do not have to trigger groups of full compound transitions from concurrent components; triggering individual ones is enough.

Consider a system which is specified as a concurrent one but implemented via state multiplication. Incorrect implementation transitions could make our refinement assumptions to be not satisfied, meaning that transitions corresponding to multiple orthogonal transitions taken in the same step may produce totally unexpected result.

This type of refinement is not the same as testing concurrent components separately (Sect. 3.3.4 on p. 59) in that it does not preclude missing transitions. In a system designed as concurrent and implemented using state multiplication, we could have missing transitions since an implementor could simply 'forget' to add some transitions to some states. For example, some implementations of multiple concurrent transitions taken in the same step, could be left out.

3.3.4 Strong AND-state refinement

In truly concurrent implementations, concurrent components are placed into units which either run in parallel on different processors or utilise an operating system to provide the same effect. Consequently, no concurrent component may cause missing transitions in some other one (the case considered above) or misdirected transitions (if no refinement has taken place). Also, if some event is generated by some action, both concurrent components should 'see' it. Absence of racing eliminates a possibility for a transition to behave differently when taken in the same step as some other (concurrent) transition.

We assume that only one transition from a concurrent component occurs in a step during testing. If it is not so, the outputs from them could superimpose and hide each other; any two transitions have also to have mutually exclusive outputs (i.e. satisfy the output-distinguishability requirement). This problem is solved by the 'no racing' requirement, Req. 3b.

For this type of refinement, the test case set is a union of the test case sets of concurrent parts, similar to the one described in the OR-state refinement, Sect. 3.2.3 on p. 52. The similarity is not coincidental — the two types of refinement are essentially the same. For the statechart in Fig. 3.5, we get

$$\begin{aligned} T \;=\; & C_{CONTROL} * (1 \cup \Phi_{CONTROL} \cup \ldots \cup \Phi_{CONTROL}^{m_{CONTROL}-n_{CONTROL}+2}) * W_{CONTROL} \cup \\ & C_{SEARCH} * (1 \cup \Phi_{SEARCH} \cup \ldots \cup \Phi_{SEARCH}^{m_{SEARCH}-n_{SEARCH}+2}) * W_{SEARCH} \qquad (3.11) \end{aligned}$$

where we test the left component forgetting about the right one, and vice-versa. $m_{CONTROL}$, $n_{CONTROL}$ and m_{SEARCH}, n_{SEARCH} correspond to numbers of states in an implementation and design of the first and second concurrent statechart.

The advantage of this method is a much smaller set of test cases compared with testing via state and transition multiplication. It can be estimated (using the Eqn. 2.4), to be

$$\begin{aligned} Size_T &\leq n^2_{CONTROL} * \mid \Phi_{CONTROL} \mid^{m_{CONTROL}-n_{CONTROL}+2} + \\ & \quad n^2_{SEARCH} * \mid \Phi_{SEARCH} \mid^{m_{SEARCH}-n_{SEARCH}+2} \\ &\leq 2 * n^2_{cs\,max} * \Phi_{cs\,max}^{m_{cs\,max}-n_{cs\,min}+2} \end{aligned} \tag{3.12}$$

For $m_{CONTROL} = n_{CONTROL}$ and $m_{SEARCH} = n_{SEARCH}$, we get $Size_T = 3 * (1 + 5) * 2 + 2 * (1 + 2) * 2 = 42$ which is 8 times less than that for multiplication of sets of transition labels (Eqn. 3.8).

3.4 Static reactions

Static reactions which do have some functionality and consequently are explicitly added to the system design, have to be treated as transitions concurrent to the functionality of the states they are defined in.

In the example of the tape recorder with a counter (Sect. 1.4.8), we allow merging rules and include counter-updating static reactions in the set Φ. Testing of them is necessary as an incorrectly implemented tape recorder may decide to traverse from *RECORD* to the *PLAY* state upon invocation of a static reaction. We also have to make static reactions satisfy the design for test condition. They also could be used to distinguish states as described in App. C.4 on p. 282 and Sect. 5.2.1 on p. 83. Since testing of explicitly added static reactions is similar to testing transitions, static reactions were not elaborated on in proofs.

Implicit 'do nothing' static reactions, which occur when no specified transitions or explicit static reactions are enabled (Sect. 1.4.8 on p. 20), do not correspond to any code and thus their inclusion in the set of test cases would be artificial. In our testing method we assume them to be correctly implemented (follows from Req. 4a).

3.5 History connectors

In the simplest case of non-deep history connectors, we could treat them as a number of transitions. This is shown in Fig. 3.6, where we can treat a history connector as a **C** connector.

In order to do testing of statecharts with history connectors, we need to:

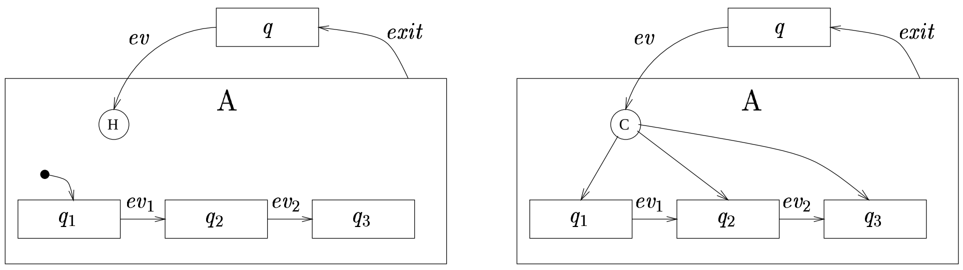


Figure 3.6: Testing a non-deep history connector

- Apply the testing method, avoiding usage of the history connector. This can be accomplished either via an appropriate design for test or with the approach described in [Hie97d].
- Test the history connector itself. For every state in a statechart with a history connector we could do the following:
  - enter the state,
  - exit the statechart,
  - re-enter it using the history connector,
  - verify the entered state.

  For example, we could try the sequence $\{ev\ ev_1\ exit\ ev\} * W_{q_2}$ to test if the history connector works for the state q_2. $ev\ ev_1$ enters it, $exit\ ev$ exits and re-enters through the history connector. We then need to apply a characterisation set to make sure we are in q_2 again.

For testing of deep history connectors, we need to visit every configuration rather than single states. Further research into testing of history connectors could be done in the future.

3.6 Generic statecharts

Some statecharts could be used as templates; they are created with placeholder variables and during instantiation (construction of a statechart following the template), these variables are replaced with real ones. The concept behind template statecharts is similar to template classes in C++. We test such statecharts separately using state refinement described in Sect. 3.2.3 on p. 52; refer to [Ove94] for description of informal approaches to testing of generic classes.

3.7 Off-page statecharts and diagram connectors

Statecharts for complex designs may get very large. In order to split them into manageable parts, off-page statecharts were introduced. An off-page statechart is the substate statechart for some state which is drawn separately rather than being placed inside that state. Semantically these two statecharts are parts of the same one.

Diagram connectors are syntactical elements which allow us to have transitions between statecharts and the off-page ones which they contain. Transitions go from the source state to the diagram connector and then from the corresponding diagram connector in the off-page chart to the target state, or the other way. They are used in the hi-fi case study, described in Sect. 8.3 on p. 236.

Such transitions can be treated as interlevel ones and are easy to handle using the given testing method.

3.8 Aspects of statecharts which are not considered

3.8.1 Syntactic elements of statecharts

Statecharts contain some other elements such as scheduling of actions and events which occur when variables are accessed and/or modified. These constructs are not considered in the thesis since they are used in transition labels while the testing method primarily focuses on testing of transition diagram assuming labels to be correctly implemented. Generating test data for transitions with such constructs could be difficult; triggers may be added explicitly as described in Sect. 4.1 on p. 65.

Events generated as a result of a statechart entring or exiting states, the condition, allowing to test if a statechart is in a specific state, and time are not generally considered since these elements are only used in labels. Augmentation of labels using timeouts is described in App. B.3 on p. 273.

3.8.2 Non-Statemate semantics of statecharts

Statecharts as proposed originally by Harel [Har87] did not have a clearly defined semantics. Many people interpreted what he said slightly differently and added their own improvements which gave rise to a variety of semantics [Ilo95b, HG94, HCB92, KP92, LC96, LHHR94, MSP96, MST97, NRS96, Per95, PM94, Sch96, Shi95, US94, JM94, vdB93, BGK98, HRdR92, MLS97]. An excellent summary of many of them can be found in [vdB94]. Each of the semantics has its advantages and disadvantages. Here we provide an overview of the most often used Pnueli and Shalev's type of semantics [PS91] and the UML one, presently gaining popularity in industry [Rat97]. Syntax of statecharts is the similar in all of them to the one described above but for most of them is seriously restricted.

Semantics with instantaneous feedback

This type of semantics has a completely different view of what a step is. Unlike Statemate, where events generated by transitions can trigger transitions in the next step only, here they can do that in the same step under restriction that only one transition from every concurrent component may occur in a step. As a result, transitions fired may cause other transitions; those transitions — yet more. The whole chain reaction occurs in the same step. This type of feedback of generated events is referred to as 'instantaneous' [NRS96]; in Statemate it is 'delayed' according to [NRS96]. It is claimed [PS97a] that this semantics is better for a high-level specification of reactive systems rather than for the detailed one or a design.

While rather intuitive, instantaneous feedback possesses a number of problems. Consider, for example, a transition a/a. It can be thought to trigger itself; in Statemate semantics it will occur only if a was generated and will generate a itself. Another problem involves transitions of the form $\neg\ a/a$. This transition is self-contradictory in the considered semantics since it requires a to be not triggered to occur and then generates it after being taken. We could say that then it should have not been taken at all.

The chain reaction of transitions triggering others can be expressed using a fixed point. In semantics of [PS91], transition triggers are required to consist of conjunctions and negations only. This guarantees the concavity of triggers w.r.t. a set of events generated. With this assumption, a theorem is proven which relates denotational and operational semantics. It says that a sequence of transitions is a step if and only if the set of them cannot be separated into two parts such that transitions in one of them do not trigger those in another one. Transitions violating the condition, i.e. those containing disjunctions, could be decomposed into a number of transitions satisfying the property.

In terms of transition structure, this group of semantics considers a simplification of Statemate's one where any connectors are not used and default transitions are replaced by default states [PS91].

Hybrid statecharts derived from semantics with instantaneous feedback [LC96, KP92] contain two types of transitions: instantaneous and those taking time.

Usage of our statechart testing method for Pnueli and Shalev's statecharts does not meet significant problems due to Req. 3c and Req. 3b, effectively eliminating instantaneous feedback. The necessary modification to the method and proofs could be done in the future.

UML

UML [Rat97] stands for the Unified Modeling Language and represents a notation created by putting a number of notations together. We restrict our consideration of UML to statecharts in it. UML imposes severe limitations on them compared to Statemate statecharts. For instance, triggers on default transitions are not allowed; only one transition may go from a default connector; full compound transitions consisting of multiple compound transitions cannot have more than one of them containing a trigger/action pair.

Priorities of transitions in UML statecharts are reversed: transitions with their scope lower in the state hierarchy have priority over higher-level ones. It is argued that this provides a way to use substate statecharts as subroutines. Unlike Statemate semantics, UML does not have step semantics. While a transition is executing, multiple transitions in concurrent states can be taken and complete before it finishes; such semantics attempts to capture the behaviour of real systems where assumptions of step semantics cannot be used. Overall, the behaviour of UML statecharts is similar to asynchronous semantics, i.e. the response of a statechart to an external stimuli is instantaneous but takes a number of transitions to complete.

The testing method developed for Statemate statecharts in this thesis could be applicable to UML provided we can guarantee synchronous behaviour under test (Req. 3c) and 'reverse' the concept of refinement. The first can be guaranteed by making sure only expected transitions are going to fire. Refinement could be considered to be bottom-up rather than top-down one (Sect. 3.2.3 on p. 52). More work on this could be done in the future.

Chapter 4

Test data generation

4.1 Design for Test

When applying our test method, we assume that all transitions are implemented correctly. It means that a design may be implemented with a different number of states, transitions could traverse them in possibly random ways, but triggers and actions (Sect. 1.4.2) of transitions are correctly implemented (Req. 4a). Moreover, we assume that we can trigger (i.e. satisfy the precondition of) every transition label by supplying an appropriate input, which involves making changes to externally accessible variables of the system. The pair of a triggering input and an output from a transition has to uniquely identify it. This is needed because, having triggered a transition, we need to be sure which one occurred. The requirement of being able to trigger is called *t_completeness* (Req. 2) and the one of the input-output pair to identify a transition — *output-distinguishability* (Req. 3a). When these two (and some others we shall talk about later in Sect. 4.1.4 on p. 72) are satisfied, we can say that design satisfies the *design for test* condition. This definition is different from that described in [HI98, IH98b, IH97, Ipa95] in that it considers a number of additional conditions, specific to statecharts. The idea of making a design testable is also mentioned in [FS97]; design for test seems to solve the problem of easily testable software failing more often [BS96] than the one which is hard to test.

Note that 'design for test' term is used for both a part of design process and a requirement (Chap. 5 on p. 79).

4.1.1 Introduction

Here we shall describe how t_completeness and output-distinguishability conditions apply to our sample tape recorder and possible problems where transitions have to be augmented in order to make them compliant with it.

Triggering transitions

For our tape recorder to be *t_complete*, we need an ability to generate $\underline{\textit{tape\_end}}$ and events generated by buttons. Additionally, having triggered a transition, we should be able to ensure that we did that with the expected one (see the note about *direction*, *play* and Fig. 2.1 on p. 29). In our example this requirement is satisfied, however for more complicated designs we may have to explicitly add some inputs and outputs for testing.

As an example of the triggering problem, consider the transition

$$\textit{TapeCounter} = 1035/\underline{\textit{operation}}' = \textit{stop}$$

which occurs only when an externally inaccessible variable is set to some value. Consequently, we have to artificially modify[1] the label associated with the transition by adding an extra input as follows:

$$\underline{\textit{trigger}} \vee \textit{TapeCounter} = 1035/\underline{\textit{operation}}' = \textit{stop}$$

where event *trigger* is selected such that it does not belong to the original set of events. In some cases a transition may be triggered by different inputs depending on the internal variables of the process. We will have then to track values of that internal data while generating sequences of inputs. For

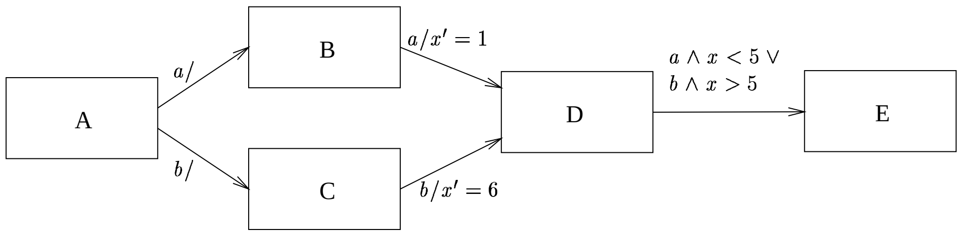


Figure 4.1: The case when we need to keep internal data in mind when triggering transitions

example, consider a statechart depicted in Fig. 4.1. Depending on whether we enter the state D from B or from C, we need to choose a different input to reach E. In simple cases like the one depicted it is not difficult but with complex transition preconditions and operations it could be. We can eliminate this problem by adding triggers to labels which could be otherwise difficult to trigger, at the design stage.

When testing is finished we can forget that we have extra variables introduced for testing and the behaviour of the system will remain the same as before any testing.

[1]This process can be described by the word *augment*. It is used to designate a change to a design by adding new inputs and/or outputs which does not affect the behaviour restricted to the original sets of possible inputs and outputs.

Outputs from transitions

One might also have to introduce an extra output if the pair $\underline{trigger}$, $\underline{operation} = stop$ does not uniquely identify a transition. This is the case for *rew_or_ff* and *rew*. If we use $\underline{rew}$ to trigger *rew_or_ff*, we cannot distinguish between it and *rew*, thus we could add an output $\underline{rew\_or\_ff}$ to *rew_or_ff*. Since we would prefer not to make any changes to an implementation in order to do testing, the design for test condition is best considered at the design time.

4.1.2 Selection of externally accessible inputs and outputs

Inputs and outputs used for testing have to be externally accessible. To achieve this in $\mu\mathcal{SZ}$, we have to declare them in some exported ports. Additionally, a special port *TESTPORT* can be created with testing purpose.

PORT *TESTPORT*

$TEST\_IN\_PROGRESS : \mathbb{B}$
$trigger_1, trigger_2, \ldots :$ signal
$output_1, output_2, \ldots :$ signal

Variables which are internal to a process class could be made accessible to the testing process by listing them in this port. New inputs and outputs introduced for testing can also be added there. When testing is finished we can either eliminate this port or connect it to a stub such that $TEST\_IN\_PROGRESS = FALSE$ (the boolean type is introduced in Sect. 1.4.11 on p. 24).

4.1.3 T_completeness and output-distinguishability for statecharts

Now we describe how statecharts can be made *t_complete* and *output-distinguishable* by augmentation. Further, the term *transition* will often be used to mean a label of a transition.

General approach to augmentation

It is possible to use the same input as a trigger for more than a single transition as shown in Fig. 4.2. On the top, we have the original design where all transitions are dependent on inaccessible data. They thus have to be augmented. We can assign the same testing input $trigger_1$ to more than one transition, as shown in the middle, provided in the correct machine they do not both emanate from the same state. In a faulty one, this will also work because we expect the implementation to implement the augmented design and be deterministic (Req. 1b, 4a). The testing output $output_1$ is assigned

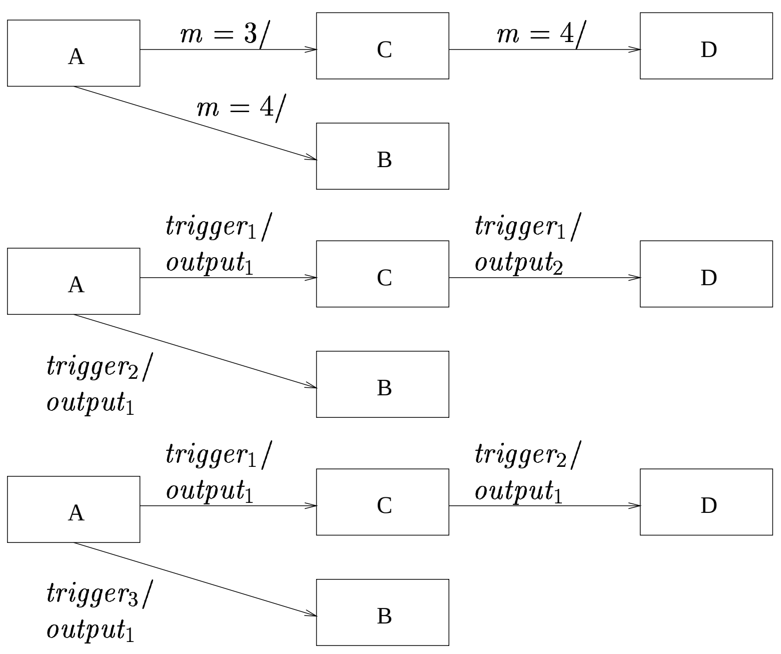


Figure 4.2: An example of augmentation where the same testing input is used for two transitions

to two transitions too. This is possible because any of the transitions can be identified by its trigger/output pair. In some cases we might introduce a unique testing input to every transition as shown at the bottom of Fig. 4.2. This would make the testing method more robust in cases when the implementation of one or more transitions is faulty. For example, transition $trigger_1/output_1$ could erroneously generate $output_2$ and thus we would fail to distinguish between states A and C. We are saying 'more robust' because no guarantees can be made in this case but a wider variety of faults will get detected, at the expense of a bigger set of test inputs. Unique test outputs may also be assigned to cope with faulty transitions.

If we are not using augmentation, there could be transitions which are in practice difficult to trigger independently, such as those with timeout triggers (App. B.3 on p. 273). At the same time, we could decide not to distinguish between them in which case an approach described in Sect. 5.3 on p. 102 is appropriate.

The determination of whether a design complies with the requirements of the testing method is undecidable, due to the undecidability of the predicate calculus. In the App. B.1 on p. 269 a heuristic algorithm is proposed in an attempt to get close to the optimal solution.

Design and implementation should remain deterministic after augmentation (a part of Req. 1b)

When augmenting transition labels to satisfy conditions considered, no nondeterminism has to be introduced. For example, consider triggering the transition *play* using some newly-introduced testing input $button_{test}$. If this input is also used to trigger *rec* transition, then we obtain a nondeterministic statechart in which we cannot selectively choose to follow a desired path from the *STOP* state.

Transitions with empty triggers (Sect. 5.2.5 on p. 90) may also cause this requirement to be not satisfied.

Augmentation of full compound transitions

Augmentation of transition labels, i.e., an addition of a testing input/output is proposed to be done in essentially the same way as before (Sect. 4.1 on p. 65), meaning that all parts (i.e. compound transitions) of full compound transitions get augmented. For example, to trigger a transition to the *REWIND* state from the *STOP* one (Fig. 1.6), we need to supply it with events to trigger both *rew_or_ff* and *rew* transitions and observe outputs from both of them. Full compound transitions constructed from augmented parts generally satisfy the t_completeness and output-distinguishability conditions too. Additional requirements are to have these parts triggerable at the same time and not mask each other's outputs (Req. 3b). For example, if *rew_or_ff* is triggerable by $\underline{rew\_new}$ then in order to trigger *rew_or_ff-rew* both events $\underline{rew\_new}$ and $\underline{button\_rew}$ have to be generated. For transitions with multiple parts, test set construction remains the same as for single-part transitions, but instead of a single event or variable we used to trigger a transition in Chap. 2 on p. 28, we may have to use a number of them. In addition, we must be able to confirm that all triggered parts occurred. More details are given below in the subsubsection on augmentation of individual transitions.

Using the augmentation approach in general, we need to make sure that it does not cause requirements for a statechart (Chap. 5 on p. 79) to fail. Possible problems are described above and in Sect. 4.1.3 on p. 70. We also may have to augment transitions to satisfy other requirements than t_completeness and output-distinguishability. This is described in Chap. 5 on p. 79 and in Sect. 8.3 on p. 236.

Note that we would be able to talk about *full compound transitions* rather than *compound transitions* being augmented as a whole only if the requirement Req. 4b were stating that full compound transitions are assumed to be implemented consisting of all the expected parts rather having a subset of them implemented. Since it does not, we have to focus at CTs rather than FCTs.

Augmentation of individual transitions of compound transitions

For compound transitions, we may wish to ensure that all its transitions have been executed. Consequently, we could work with individual connector-to-connector transitions rather than with a higher-level compound ones and ensure the output-distinguishability condition for such individual transitions. Note that here we are talking about **C** and other connectors which are not considered by the testing method and assumed to be absent at the stage of test case generation; in future we could extend the testing method to such lower-level transitions (Sect. 6.8 on p. 206). Alternatively, we can assume that connector-related part of the statechart is implemented correctly and make sure that compound transitions are output-distinguishable as a whole, as described above. Problems with CTs related to augmentation of individual transitions are very similar to those with FCTs caused by augmentation of CTs, since in both cases transitions occur in the same step. Parts of compound transitions are given with **C** connector between them (and in some figures here, such as Fig. 4.3, the corresponding CT is drawn); for full compound transitions, we would have CTs separated by a state boundary. Due to the noted similarity, we describe both types of problems here.

The two concepts, augmentation of compound transitions as a whole and on a connector-to-connector basis, are illustrated by an example. Fig. 4.3 depicts transitions and a connector and the equivalent compound one.

Augmentation results are shown in Fig. 4.4 for modification of the com-

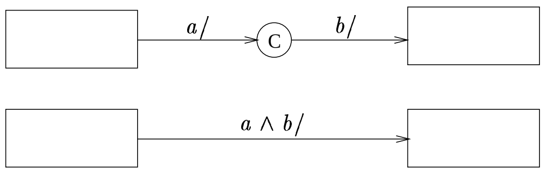


Figure 4.3: Connector-to-connector transitions and the corresponding compound one

pound transition in Fig. 4.3; connector-to-connector transitions shown at the top of the Fig. 4.3 are augmented in Fig. 4.5 with equivalent compound transitions given at the bottom of it.

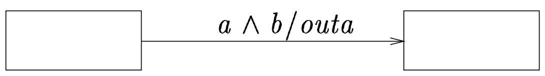


Figure 4.4: A result of augmentation of a compound transition

Both types of augmentation can be accomodated by the proposed approach with augmentation of parts of individual compound transitions being

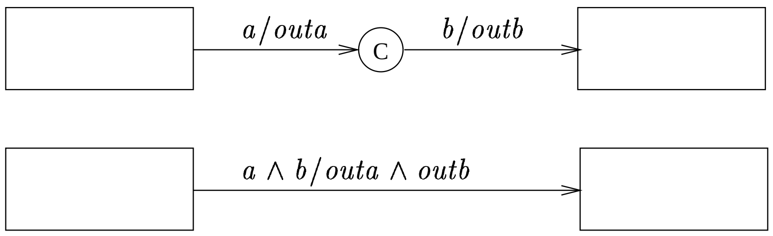


Figure 4.5: A result of augmentation of a connector-to-connector transition

probably simpler. Since it is not clear which of the two is best in general, both were described.

Individual augmentation of transitions may also lead to our inability to trigger groups of them. For example, if we have transitions where negation is used in triggers, in concurrent components of a statechart, the result of augmentation may fail to be t_complete. For instance, for $a \vee b/q_1$, $a \vee \neg b/q_2$ we should select a rather than decide to trigger b for the former and not for the latter, which is not possible if we would like to take them at the same time, (i.e. in the same step). The same problem may occur between independently augmented compound transitions.

A problem where augmenting individual transitions caused CTs to be not output-distinguishable is depicted in Fig. 4.6, which shows the result of augmentation. If we take the compound transition from state A to B, the

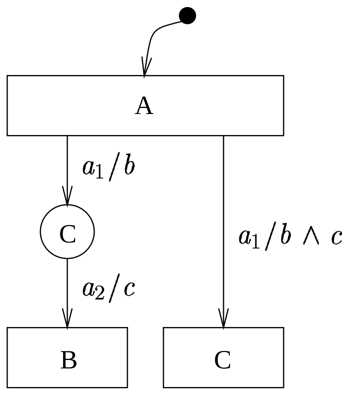


Figure 4.6: Transitions rendered not output-distinguishable as a result of augmentation

output produced will be the same as that of the transition from A to C and thus we would not be able to tell which one occurred.

Another example where augmentation of a transition individually has lead to our inability to reason whether both of them occurred or not, is given in Fig. 4.7. If we trigger the depicted transitions on their own, we

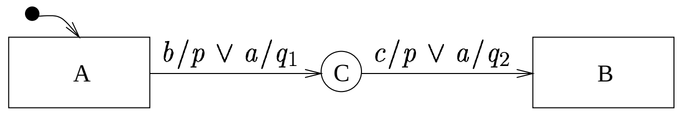


Figure 4.7: Independent augmentation of transition does not lead to output-distinguishability

could select the b event to be a trigger of the first one and p — the expected output as well as c and p for the second one. When we execute them, the two p events do not allow us to reason if both transitions fired or only one of them. Note that selection of event a would eliminate the problem and allow for a smaller set of events to be generated for triggering, specifically $\{a\}$ instead of $\{b, c\}$.

Additional problems possible with augmenting individual transitions are described in Sect. 5.2.6 on p. 92.

4.1.4 Other design for test requirements

T_completeness (Req. 2) and output-distinguishability (Req. 3a) are not the only requirements that a statechart has to comply with. The whole group of requirements is called design for test because a statechart can be made compliant with it by proper augmentation which has been described above for the two conditions.

Synchronicity of behaviour (Req. 3c) can be enforced not only by appropriate augmentation which is not described here but also by embedding a tester statechart into the one under test. Both approaches are provided in Sect. 5.2.7 on p. 93.

Structural requirements (Req. 1e, 4b, 4e, 4f, 4g) are not considered in detail; they are believed to be relatively easily diagnosed and appropriate statecharts augmented. This will be illustrated by the hi-fi case study in Sect. 8.3 on p. 236.

4.2 Generation of a test set

Having constructed a set of test cases as described above in Chap. 3 on p. 46, we need to generate concrete test data for each of them. The idea is to generate a sequence of test data for each sequence of transitions where every element of this sequence serves as an input for triggering the corresponding transition in the test case under consideration. For a statechart satisfying the design for test condition (Sect. 4.1 on p. 65), the task is relatively easy. Often, we can replace each transition name with an input corresponding to that transition. For example, a possible input sequence corresponding to *rew_or_ff-play play* in Fig. 1.10 is $\{\underline{rew}, \underline{play}\}\,\{\underline{play}\}$. We could also use $\underline{ff}$ instead of $\underline{rew}$ there.

Triggering and observation of the effect of transitions is done using *changes* to variables. Such changes include a modification of a persistent variable or an event generation. Details are in Sect. 6.1.2 on p. 118. The test set consists of sets of sequences of sets of changes to persistent and event variables to be applied to a system under test. The above illustration is one such sequence. Every element of it is expected to be executed in a separate step.

For a sequence of sets of transition labels *lpath*, function $t(lpath)$ can be defined to return the sequence of sets of inputs to follow the considered path, with optimisations considered in Sect. 4.2.2 on p. 73. This t is called *fundamental test function.* It is an extension of the one given in [Ipa95] for sequences of sets of labels rather than just sequences of labels.

4.2.1 Simulation

To construct expected outputs, we compute (i.e. simulate) the reaction of the design to the input data sequence. In the case where no transition or explicit static reaction gets triggered by some input, we expect a 'do nothing' static reaction to occur and the implementation to produce an empty output, corresponding to changes to no variables. For example, a test sequence $\underline{rew}\,\underline{play}$ applied in the initial *STOP* state corresponds to *operation* changing to *move* and then to *play*.

Determination of transition outputs by simulation can be achieved using the language called SCP. It is a Pascal-like language built into Statemate supporting complete control over its execution environment. It means that it is possible to write a program which will read a sequence of test inputs from a file, apply it and write outputs to another file.

In $\mu\mathcal{SZ}$, Z schemas are converted into C programming language and interfaced to Statemate via a C or a JAVA library. This provides an alternative way to apply testing inputs to the statechart considered and extract outputs.

4.2.2 Simple optimisations

Due to cutting of test sequences after inputs corresponding to transitions which should not exist and prefix removal, both described below, the resulting test set can be much smaller than the set of test cases. In the worst case, however, they are of the same size. Optimisations are performed after test data generation since we could use the same input for more that one transition, thereby possibly 'collapsing' a number of test cases into a single sequence of inputs.

Prefix removal

Some simple optimisation of the constructed test set can be performed. If we have sequences of inputs like $\underline{rew}\,\underline{play}$ and $\underline{rew}\,\underline{play}\,\underline{stop}$, we can eliminate the former as it is the same as the beginning of the latter (a *prefix*). It can be defined formally following [Spi92]:

$$\begin{array}{|l}
[AA] \\
\hline
\_\mathit{prefix}\_ : \mathrm{seq}\,AA \leftrightarrow \mathrm{seq}\,AA \\
\hline
\forall f, \mathit{seq} : \mathrm{seq}\,AA \bullet f\ \mathit{prefix}\ \mathit{seq} \Leftrightarrow (\exists\, c : \mathrm{seq}\,AA \bullet f \frown c = \mathit{seq}) \\
\hline
\end{array}$$

Note that this kind of optimisation may also be applied to the W set, which is made possible by its usage as the last element in the multiplication during test case generation (Eqn. 2.2).

Cutting test sequences

Consider an invocation of the *play* transition in the *RECORD* state. If it exists, it is a fault in the implementation regardless of the entered state; otherwise the implementation passes this test case. Consequently, when making sure that some transition is not implemented in some state, the test case sequence may be cut after an input corresponding to that transition. For transitions which have to exist in the implementation, such as the *stop* transition from the *RECORD* state, we need to test not only their existence but also whether they terminate at the expected state. As the W set is constructed using existence or nonexistence of some sequence of transitions, we can also terminate test sequences after an input corresponding to the first transition which should not exist.

4.2.3 Loops and growth of a test set w.r.t. $m - n$

The size of a test set is often estimated for $m - n$ without regard to the set of labels. Taking it into account reveals that the test set may be much smaller. Here we try to estimate its expected size.

Consider a simple statechart which looks like a tree, i.e., with the limited path length. It is depicted in Fig. 4.8. For this statechart, the test set does not grow when m increases.

What makes the test set grow is loops, as if we can traverse more states than there are in a statechart in the same sequence then we have to visit one of them more than once. Consider a statechart in Fig. 4.9 which contains a single loop. The resulting test set growth is directly proportional to $m - n$. 'directly' follows from that we can only repeat a single transition many times. In the statechart in Fig. 4.10, we have two loops and the test set size is expected to be 2^{m-n}. It could be possible to state the following:

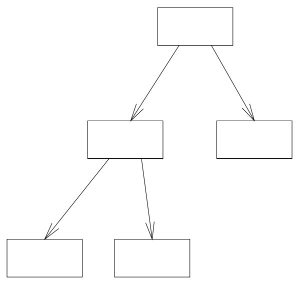


Figure 4.8: A tree-like simple statechart

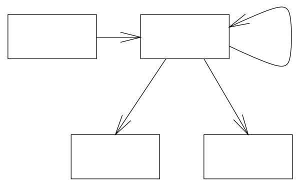


Figure 4.9: A simple statechart with a single loop

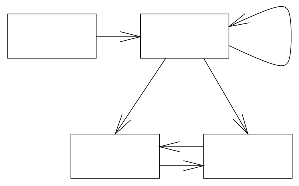


Figure 4.10: A simple statechart with two loops

Conjecture. *The growth of the test set for an X-machine w.r.t.* $m - n$ *is* $min(\mid \Phi \mid,$ *number of loops). It means that the test set size is of an order of* $min(\mid \Phi \mid,$ *number of loops*$)^{m-n}$ *(or it could be just a number of loops without min).*

No research was done in this area such that the formula presented above could actually be reduced to (number of loops)$^{m-n}$. It is also unclear how to estimate the number of loops and whether their length matters. We could try to obtain the number of loops and then try to tackle the task of obtaining it from the transition graph. Looking into relations between a number of loops and growth of a set of test cases depending on $m - n$ is a possible direction for future work.

4.2.4 Reduction of the number of inputs to trigger transitions reduces the test set

Consider a finite-state machine shown in Fig. 4.11. We can construct a test

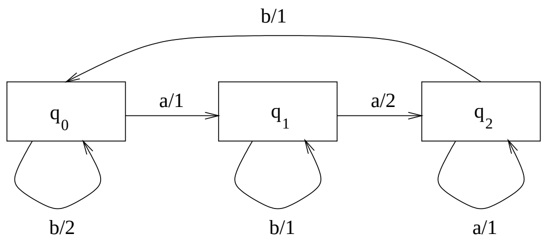


Figure 4.11: A finite-state machine with outputs

set for it using the original Chow's W method [Cho78]. Treating it as an X-machine, it is possible to generate another test set. Test case bases for these are shown in Tab. 4.1. From the table it can be observed that the set

	X-machine	finite-state machine
C	$1, a/1, a/1\,b/2$	$1, a, a\,a$
W	$b/2, a/2, b/1\,b/2$	$b/2, a/2, b\,b/1\,2$
	$\Phi = \{a/1, a/2, b/1, b/2\}$	$\Sigma = \{a, b\}$

Table 4.1: Comparison of the test case bases for the same machine considered to be an X-machine and a finite-state machine

of test cases for an X-machine will be much bigger, compared to the test set for the FSM. The situation changes when we generate a test set, because we use input a to trigger labels $a/1$ and $a/2$ and similar for b. Consequently, the two test sets have the same size.

This is the reason of the difference between the size of the set of test cases and that of the test set in the example. The test set can be reduced further if we group labels and use one input for each group. Consider a partition $P = \{\Phi_1, \Phi_2, \ldots\}$ of Φ such that

$$\begin{aligned} \forall\, \sigma_i, \sigma_j : P \bullet i \neq j \Rightarrow \forall\, \phi_a \in \sigma_i, \phi_b \in \sigma_b &\quad \bullet \quad \text{test\_input}(\phi_a) \neq \text{test\_input}(\phi_b) \\ \forall\, \sigma : P \bullet \forall\, \phi_i, \phi_j : \sigma &\quad \bullet \quad \text{test\_input}(\phi_i) = \text{test\_input}(\phi_j) \end{aligned}$$

Partitioning presented is similar to that of [Ipa95] where Ipate was constructing a set of test outputs for augmentation. It is possible to have an associated automaton with inputs being inputs to trigger any label in a partition and outputs — outputs from the expected label. It means that instead of trying to trigger every label $\phi_{i\,1} \ldots \phi_{i\,n_i} \in \sigma_i$, we use their common input. The output will allow us to reason which of the labels has been actually triggered, based on the assumption of the correct implementation of Φ.

Although finite-state acceptors may seem to be simpler than machines with outputs, usage of outputs in an associated automaton does not lead to an apparent reduction of a test set size. This is evident from looking at our example where the only difference between TCB for an FSM and that for an X-machine is that of the set of Φ v.s. Σ which gets reduced to zero once we consider inputs.

The time to construct a test set can be considerably reduced if we construct a set of inputs, capable of triggering all transitions. In a simple case where this set is state-independent, it can be used in the construction of a set of test cases instead of Φ (with inputs being clearly distinguished from labels). For Fig. 4.11, it will be $\{a^{input}, b^{input}\}$ for the set of labels $\{a/1, a/2, b/1, b/2\}$.

Theoretically, we could test X-machines by supplying them with all possible inputs, i.e. by using Σ instead of Φ. This will in most cases make a test set much bigger. With finite-state machines treated as X-machines, as we have observed, this is not always true.

In practice it may be useful to balance the two, i.e. use inputs generated from labels as in X-machine testing method as well as try a subset of a set of inputs. This would allow us to weaken our assumptions of correct implementation of labels.

4.3 Test set execution and monitoring

In order to supply a test statechart with inputs and observe outputs, the tester has to communicate to the statechart. We can either test it with a separate statechart connected to the process under test via ports or have it embedded in a statechart being tested (App. A on p. 267). Problems related to Req. 3c being not satisfied are described in Sect. 5.2.7 on p. 93.

4.4 Test result analysis

Test result analysis is not a trivial task in case a test reveals a fault by producing a different output from the expected one. In order to locate a fault, we shall have to perform extensive further testing, work has been done on that [Hie97a, RDT95b].

The ability of testing methods to detect a single fault can be defined as follows [RDT95b]:

Definition 4.4.1 (t-fault resolution capability of level k). *A test sequence selection method has t-fault resolution capability of level k if for any implementation with at most t faulty transitions, a test sequence selected by the method can localise at least one faulty transition to within a set of k transitions provided the implementation is faulty.*

In the area of fault diagnosis for different testing methods, we can consider implementations having only one fault, i.e. *1-fault resolution capability.* Wp method (App. C.3 on p. 281) is shown by [RDT95b] to be one of the best but inferior to the DD one (Sect. 2.3.1 on p. 37).

In the case where a test does not discover a fault, we may have to estimate the reliability achieved. The latter has to be done from a probabilistic model of failures. For example, if we estimate m to be 6 for our example, it is possible to compute a probability for it to be 7 or more. There could also be some distant chances for the *reset* transition to fail in the implementation or some other requirement be not satisfied. The method for statistical evaluation of the quality of a tested product might be developed in future.

Chapter 5

Requirements for the test method

In order to generate the test set and draw conclusions from this test set not revealing faults, a number of conditions have to be satisfied. We group them according to the phase of testing where they apply.

5.1 Summary of the requirements

Here we provide all the requirements a statechart has to satisfy in order for the developed testing method to be useful for testing its implementation. In practice, some of these requirements may be not satisfied. In such cases the method might have to be slightly modified in order to be applicable and/or it could lose some of its fault detection ability. A description of why the following requirements are needed and what could be done if a system does not comply with them is given in Sect. 5.2 on p. 82.

1. Test cases can be generated if the following conditions can be asserted:

   (a) minimality of the design: the statechart in every OR-state is minimal, i.e. all states are reachable from its default connector, restricted to non-interlevel transitions; no two of them have equivalent behaviour (again restricting the statechart to non-interlevel transitions only). This is formalised in Def. 6.4.6. As shown in Prop. 6.4.25, with this requirement we can also claim minimality of an X-machine, behaviourally equivalent to the statechart considered (Prop. 6.3.4 and Th. 6.3.10).

   Consideration of implementations with states being split is provided in Sect. 8.2 on p. 232 and Sect. 2.3.2 on p. 41.

   (b) design and implementation are deterministic (explained in Sect. 5.2.2 on p. 83 and in Sect. 6.1.4 on p. 153).

   (c) existence of an upper bound on the number of extra states in an implementation (explained in Sect. 2.2 on p. 31); this uses Req. 4.
   (d) a decision was made whether or not we consider all labels to be candidates for usage in C, W, Φ, or we exclude some. For example, 'do nothing' static reactions can be ignored. The description is provided in Sect. 5.2.3 on p. 85.
   (e) no shared labels exist between transitions in statecharts in different states, except that default transitions are allowed to have empty triggers (which implies that they are shared). This is described in Sect. 5.2.4 on p. 86 and formalised in Def. 6.1.35.
   (f) non-default transitions cannot have empty triggers. This is described in Sect. 5.2.5 on p. 90 and described formally in Def. 6.6.1.
   (g) there should be a unique initial configuration in the statechart design. A default transition entering this configuration should have no trigger. This is used in Sect. 6.3.3 on p. 173.

2. For a set of labels, we have to make certain that:

   (a) labels we wish to trigger can be triggered regardless of memory value and
   (b) those we do not wish to trigger, will not be triggered.

   This requirement, referred to as t_completeness, is formalised in Def. 6.6.2. A description of the importance of the second clause of it is provided in Sect. 5.2.7 on p. 95.

3. Test outputs can be generated if:

   (a) every transition label is identifiable by its input/output pair (output-distinguishability), described in Chap. 4.
   (b) transitions taken in the same step should not assign values to the same variable. It also means that outputs from transitions occurring in the same step do not mask each other. This requirement, called 'racing' [NH95] is described in Sect. 5.2.6 on p. 92 and Sect. 4.1.3 on p. 70.
   (c) synchronous behaviour under test, implying that no unexpected chains of transitions may occur, due to Req. 2. This requirement is described in Sect. 5.2.7 on p. 93.

   These requirements are formally defined in Def. 6.6.2.

4. In order to draw conclusions from the results of testing, we have to ensure the following (requirements 4e-4g are formally defined as a part

of definition for the transition structure of a statechart and Req. 4h — in Def. 6.1.26):

(a) an implementation can be modelled as a statechart with the same sets of labels on compound transitions as those in the original statechart. The design has been built with Statemate statechart semantics in mind; the step semantics is correctly implemented,

(b) a reasonably correct implementation of full compound transitions. This means that subsets of sets of labels comprising full compound transitions are implemented as single transitions with initial compound transitions never left out. Naturally, if a proper subset of any FCT is implemented, such an implementation is to be declared faulty during testing. This is described in Sect. 5.2.9 on p. 98.

(c) refinements assumed took place.

(d) both design and implementation have a reset transition, which we assume to end all our test sequences with.

(e) a transition cannot enter a default connector explicitly. This is described in Sect. 5.2.10 on p. 101.

(f) transitions from default connectors cannot leave the state within which they begin. This is explained in Sect. 5.2.11 on p. 101.

(g) There should not be any transitions from an OR-state being an immediate substate of an AND-state. This is described in Sect. 5.2.12 on p. 102.

(h) Default transitions are non-interlevel. This is believed to be unnecessary but was assumed in order to simplify proofs of merging rules given in Sect. 6.4 on p. 178. Note that even if we allow interlevel default transitions, in every OR state there would have to be at least one non-interlevel default transition or otherwise states within that state would be unreachable during construction of C.

When t_completeness, (Req. 2), output-distinguishability (Req. 3a), synchronous behaviour (Req. 3c) as well as structural requirements Req. 1e, 4b, 4e-4g are satisfied, we say that the *design for test* condition holds. The name follows from a possibility to make a statechart comply with these conditions by an appropriate augmentation (p. 66). From the above requirements comprising design for test, Req. 2 and Req. 3a are to do with labels and the rest — with structure of statecharts. Requirements which are not a part of the design for test are considered by the author to be necessary for well-behaved statecharts.

5.2 Explanation of requirements for statecharts

Here we describe the need for the requirements provided above and what can be done if they are not satisfied.

5.2.1 Minimality of the design (Req. 1a)

Applying minimality of an X-machine [HI98, p. 171] to simple statecharts, we get that a statechart has to have all its states accessible by a sequence of transitions and none of them be behaviourally-equivalent. Here we treat statecharts as finite-state acceptors with inputs being transition labels. Minimality makes it possible to construct a state cover C and a characterisation set W. If all simple statecharts of a complex statechart are minimal, a flattened statechart corresponding to the complex one is minimal too (Prop. 6.4.25). Here we shall discuss problems occurring with non-minimal statecharts.

Statecharts without interlevel transitions

The set W cannot always be constructed using our incremental approach. In such cases we might either look inside OR states or introduce static reactions in order to distinguish states. We consider these two in turn.

An approach of looking inside OR states is illustrated in Fig. 5.1 where we might have to consider transitions b and c of the statechart in the state B to distinguish it from C. Since every two statecharts have non-intersecting sets

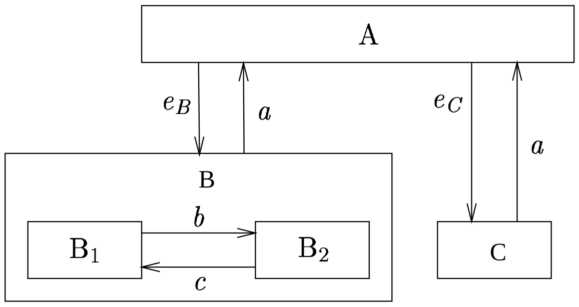


Figure 5.1: The case when we have to look inside an OR state

of transitions, we could in principle use any transition inside B to identify it in the main statechart even when it is possible to distinguish it using transitions of the main statechart. This approach allows one to reduce the usage of transitions defined on higher levels in the state hierarchy, compared to those at the lower level. Unfortunately, distinguishing states in such a way leads to a considerable increase in the size of the set of test cases. This is caused by the fact that while testing the main statechart we do not know which state of B we are in. Consequently, we need to try transitions specific to B until one occurs and this would potentially require the whole Φ_B to be included in W. In the example in Fig. 5.1, both b and c have to be included. This problem can be eliminated, if for every state in the substate statechart there is a transition in $W_{SUBSTATE\,STATECHART}$ which exists from it. For the statechart in Fig. 5.1, it means that for every state in B, i.e. B_1, B_2, there is a transition in W from them, $\{b, c\} \subseteq W$. This could be the case if B had more than two substates and b, c were used to distinguish them. The proof of the described modification of W is similar to that of Prop. C.2.5 on page 280.

Static reactions can be introduced in order to distinguish states. Consider Fig. 5.2.

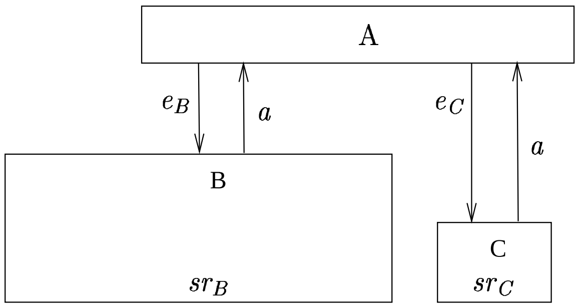


Figure 5.2: The case when we have to consider static reactions

We have states B and C behaviourally equivalent and thus indistinguishable unless we add static reactions sr_B and sr_C to them. Usage of static reactions can lead to a reduction of a test set size as described in App. C.4 on p. 282.

Statecharts with interlevel transitions

Construction of a test case basis for a substate statechart is not always possible as a substate statechart might be very closely related to the parent one via interlevel transitions. It is illustrated in Fig. 5.3 where C and W for the statechart in the F state have to include interlevel transitions and we prohibit them from appearing there. In practice we shall consider flattening such states.

5.2.2 Nondeterministic designs (Req. 1b)

In this subsection we consider nondeterministic statecharts. Cases when nondeterminism is introduced as a result of augmentation are described in Sect. 5.2.2 on p. 83; racing as a cause of nondeterminism is given in Sect. 5.2.6 on p. 92.

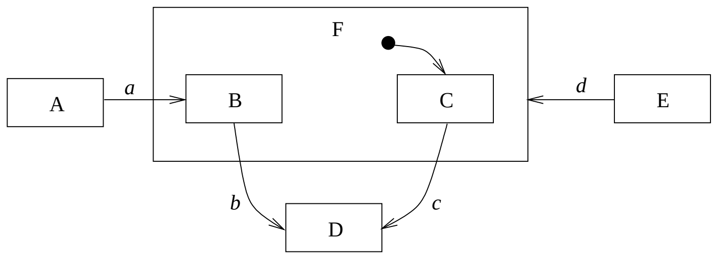


Figure 5.3: A substate statechart with nonexistent test case basis

Structural and functional determinism

The concept of determinism in the thesis means structural determinism. In the following we give an example of a simple statechart which is structurally nondeterministic but exhibits deterministic behaviour. The diagram is given in Fig. 5.4; labels are defined as

$$\begin{aligned} \mathit{init}: &\quad /m' = 0 \\ a: &\quad /m' = m + 1 \\ b: &\quad /m' = m + 2 \\ c: &\quad /\mathsf{df}\ \underline{\mathit{output}}' \wedge \mathsf{vl}\ \underline{\mathit{output}}' = m \end{aligned}$$

This simple statechart is structurally nondeterministic, but produces a deterministic result ($\underline{\mathit{output}}$=3).

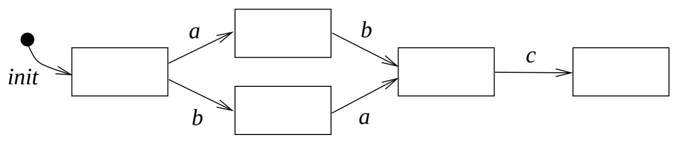


Figure 5.4: Non-deterministic simple statechart producing deterministic output

Testing of nondeterministic statecharts

The testing method allows us to test the transition diagram but in general requires deterministic behaviour during test execution. Note that as a result of testing of nondeterministic statecharts we would only be able to show the equivalence of relations computed by them rather than of functions as is the case for deterministic systems.

In order to apply the testing method we could augment transitions with nondeterminism, such that during testing the implementation will behave deterministically. In some cases, however, this could be impossible; an example is given in Fig. 5.5. The reason is that both *trans* transitions are

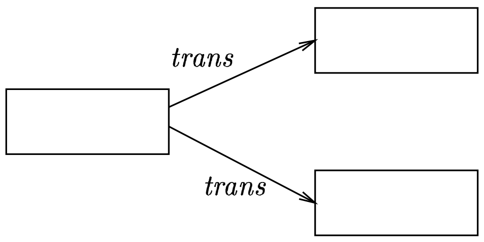


Figure 5.5: A statechart which is always nondeterministic

always either enabled or not and nondeterminism is always present unless these transitions can be augmented independently, refer to Sect. 5.3 on p. 102 for details on that. In such cases of nondeterminism, we could have difficulty constructing sets C and W for a model.

An alternative approach is similar to [LvBP94, IH99] where a complete-testing assumption is used. This assumption states that nondeterminism is 'fair', i.e. if from some state a number of transitions can be nondeterministically chosen, then by taking a test which triggers them eventually all choices would be used.

Testing nondeterministic statecharts could be considered in future.

5.2.3 Excluding some transitions from the test (Req. 1d)

Some transitions may be assumed correct and excluded from testing. For instance, this could be done with implicit 'do nothing' static reactions as augmentation of them may be considered difficult because it affects a large number of states. Transitions excluded from testing should not be added to any set of TCB.

Some transition labels possess properties, preventing them from being used in C and W, although they can be included in Φ. This is the case for interlevel transitions or those which are shared between different statecharts. Although the latter contradicts Req. 1e, the testing method could still be applied if we do not use such transitions in C and W.

Assume that E is the set of transitions of the considered statechart which should not be included in C and W. Construction of appropriate C and W sets can be accomplished by building an auxiliary statechart, with transitions of the original statechart excluding those in E. For such a statechart, we can try to compute C and W sets. In the case it is possible, the result will satisfy the condition of not using shared and interlevel transitions.

Unfortunately, in some cases it is not possible as shown in Fig. 5.6 because after removal of 'bad' transitions the statechart becomes disconnected as shown in Fig. 5.7.

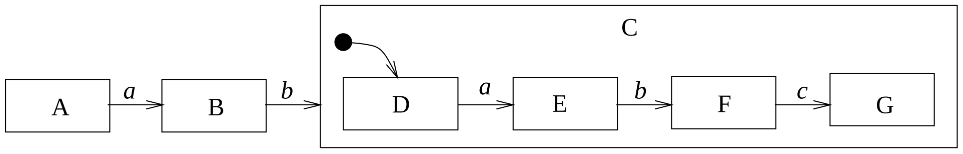


Figure 5.6: Another case when we have to look inside an OR state

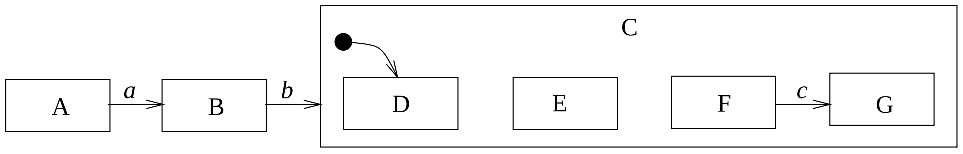


Figure 5.7: A disconnected statechart resulting from removal of shared transitions from that in Fig. 5.6

5.2.4 Problems related to shared transitions (Req. 1e)

Here we describe problems which occur when we consider statecharts with shared transitions, for statecharts with and without interlevel transitions.

Note that transitions with empty triggers in different states are inherently shared; this is not a problem in our case since non-default transitions cannot have empty triggers by Req. 1f and default transitions with them are excluded from consideration.

Statecharts without interlevel transitions

The construction of a set of test cases as proposed in Chap. 3 on p. 46 may encounter a problem depicted in Fig. 5.8 where *rew_or_ff* in Fig. 1.6 is replaced by *ff*. With both the main statechart and the one in the state *REW_FF* complying with 'design for test' condition, transitions with the same label will bear the same test input and output (more on it in Sect. 5.3 on p. 102). If we used *rew_or_ff* in W to tell *STOP* and *REW_FF* apart in the main statechart, we shall now fail to distinguish between *STOP* and *F_ADVANCE*. This follows from the input event $\overline{button\_ff}$ being used to distinguish *STOP* from states *PLAY*, *REC*, *REW_FF* in the main statechart and *REWIND* from *F_ADVANCE* within the *REW_FF* state. Note that the problem exists only because transitions to distinguish states in *REW_FF* contain transitions from the main statechart. In other words,

$$TRANSITIONS(W_{REW\_FF}) \cap TRANSITIONS(W_{MAIN\,STATECHART}) \neq \varnothing$$

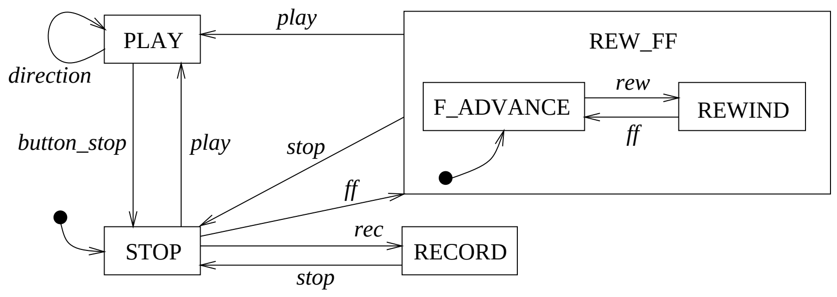


Figure 5.8: An illustration why every statechart may have to be augmented

as *ff* belongs to both of them. $TRANSITIONS(W)$ (Def. 6.1.18 on page 125) means the set of transitions participating in the set W. Above, the set of transitions in W_{REW_FF} essentially coincides with W_{REW_FF}. It will not necessarily be the case if W contains sequences of transitions. For the statechart in Fig. 2.2, $W = \{a, c, b\ a, b\ c\}$, $TRANSITIONS(W) = \{a, b, c\}$. To cope with the problem, we can use a different $W_{MAIN\,STATECHART}$ which would use *rec* to distinguish between $STOP$ and REW_FF.

Statecharts with interlevel transitions

Consider the design of the tape recorder which contains an additional *rec* transition from the *rewind* state in Fig. 5.9. The set of test cases for such

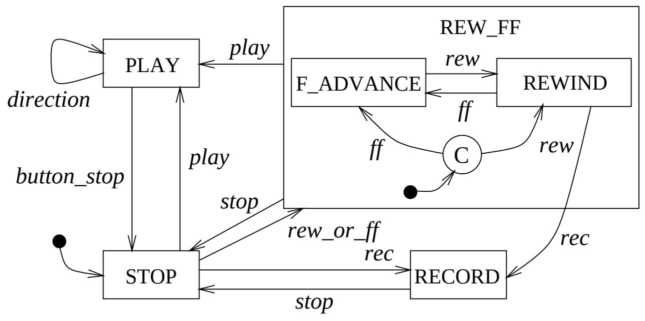


Figure 5.9: An example of an interlevel transition

a statechart coincides with that constructed for the statechart in Fig. 1.6. According to the rules for the construction of a test case basis of the main statechart, we might like to use the *rec* transition to distinguish REW_FF from the $STOP$ state. For this reason, during testing we would trigger *rec* while in REW_FF. In the case we were in the $REWIND$ state, the interlevel transition would occur and we might confuse the $STOP$ and REW_FF states. Even with the knowledge of the behaviour of the *rec* transition it cannot be used in test case basis construction for the main statechart because it may occur only when REW_FF is in the $REWIND$ state and not in $F\_ADVANCE$.

The most simple solution to this problem is either to flatten REW_FF or not use *rec* in $C_{MAINSTATECHART}$ and $W_{MAINSTATECHART}$.

AND-states — testing by multiplication of states and transitions

We can apply our incremental test case construction approach only as long as concurrent parts do not have transitions with the same trigger. An example

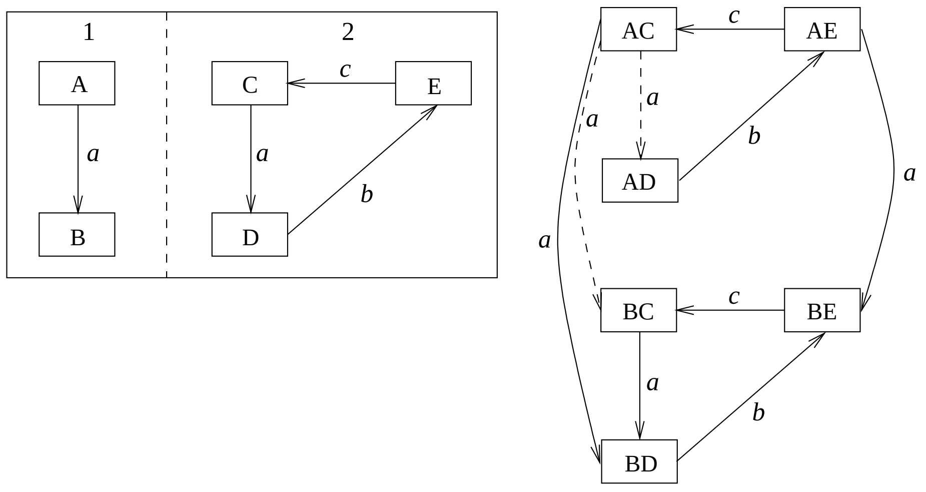


Figure 5.10: The problem with incrementally testing AND-states via state multiplication

of a problem if this does not hold is shown in Fig. 5.10, where we have the following sets for the second concurrent component:

$$\begin{aligned} C_2 &= \{1, a, a\,b\}, \\ W_2 &= \{a, b\}, \\ \Phi_2 &= \{a, b, c\}. \end{aligned}$$

Further,

$$C = C_1 \underline{*} C_2 = \{1, a\} \underline{*} \{1, a, a\,b\} = \{1, a, a\,a, a\,b, a\,a\,b\}$$

This is not quite what we would like, just observe that $a\,a$ is the same as a (by cutting sequences, Sect. 4.2.2 on p. 74), making $C = \{1, a, b\}$. It definitely cannot allow us to visit six states. The right part of Fig. 5.10 depicts the flattened statechart equivalent to the one on the left. When event a is generated, both transitions $A \to B$ and $C \to D$ take place making state AD unreachable. In order to get from C to D we have to invoke a transition triggered by a, but this will move us out of the state A. In fact, to remain in the state C of the second concurrent component after triggering

a, we have to take path $b\,c$. We could thus search for paths which allow us to return to a state which was left 'accidentally', which is not easy.

AND-states — separate testing

The method meets the same difficulty as above when there are transitions in concurrent components with the same trigger. Consider testing of the faulty implementation on the right of Fig. 5.11, against the design on the left. When we trigger a in state C, the transition from A may occur. As

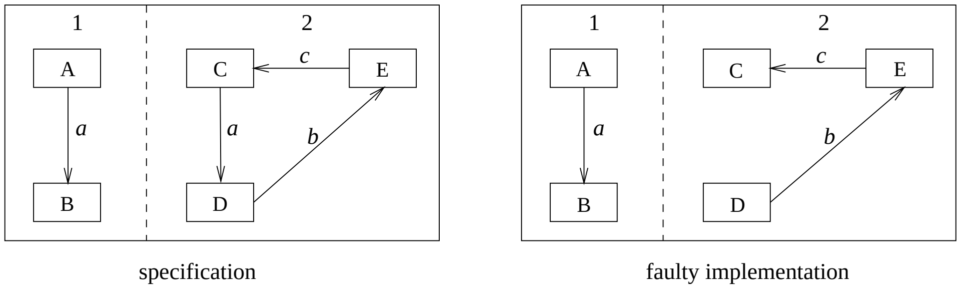


Figure 5.11: The problem with incrementally testing AND-states with each state tested separately

we cannot tell whether a occurred from A or from C, such a fault will be undetected. This problem can be solved by the augmentation making transitions in different states behave differently under test, the description to follow.

Augmentation of shared transitions in different statecharts

The proposed set multiplication above works if we replace a with some transition having a different trigger in each concurrent component. We could easily enforce this behaviour by an appropriate augmentation. This kind of modification may involve adding a test input and a variable as shown below:

$$\begin{array}{|l}
\textit{LABEL\_a} \\
\hline
\Xi\, \textit{TestPort} \\
\hline
\textit{TEST\_IN\_PROGRESS} \wedge \\
\quad (\mathsf{df}\ a \wedge 1 \in \textit{INSTATE} \vee \mathsf{df}\ d \wedge 2 \in \textit{INSTATE}) \vee \\
(\mathsf{df}\ a \wedge \neg\, \textit{TEST\_IN\_PROGRESS}) \\
\hline
\end{array}$$

Above, $\textit{INSTATE}$ is the set of states the statechart is in, i.e. a configuration; $\textit{INSTATE}$ is a set defined in $\mu\mathcal{SZ}$.

Under test, this transition behaves differently in each concurrent component. The augmentation shown can be done automatically. The disadvantage of this approach is a bigger set of extra inputs/outputs and the same transitions having different triggers if they happen to be in different statecharts. Proposed augmentation of transitions can be done for shared transitions in OR-states too. If we are trying to reuse schemas from transitions in the main statechart within the *REW_FF* state, this approach becomes difficult to apply. More details are given in Sect. 5.3 on p. 102.

In some cases, we might also wish to augment outputs of shared transitions to provide information on the concurrent state they occurred in. This could be done similarly to the described input augmentation.

5.2.5 Empty transition triggers (Req. 1f)

In statecharts, transitions may have an empty trigger. It means that they are always enabled (with domains of their *label*s being the whole of *DATA*, refer to Sect. 6.1.2 on p. 118 for details) as long as a statechart is in the source state of such a transition. These transitions may cause complications to test set generation since

- in states with such transitions we cannot trigger any other non-interlevel transitions without causing nondeterminism (Req. 1b is not satisfied),
- chains of transitions may occur (Req. 3c may be not satisfied).

This problem could be solved by augmenting transitions containing empty triggers to make them 'well-behaved' (i.e. not constantly triggered). Here we try to look at the problem and identify cases when such augmentation is not needed. What we say also applies to transitions which have their trigger enabled nearly always.

In the following we describe problems outlined above and then provide an approach to the augmentation of transitions to eliminate those with empty triggers.

No other transition can be triggered without causing nondeterminism

Consider a statechart in Fig. 5.12. Since transitions taken in a step are full compound, the described problem only occurs if all parts of them other than default transitions are empty-triggered. This is the case with the transition from state A to C.

According to the testing method (Sect. 2.2 on p. 29), we trigger every transition of a statechart from every state. Since the transition from A to C is always enabled, triggering any other transition, such as the $A \rightarrow B$ one, from A causes nondeterminism contradicting Req. 1b (nondeterminism is described in Sect. 5.2.2 on p. 83). Note that we can still 'trigger' transitions

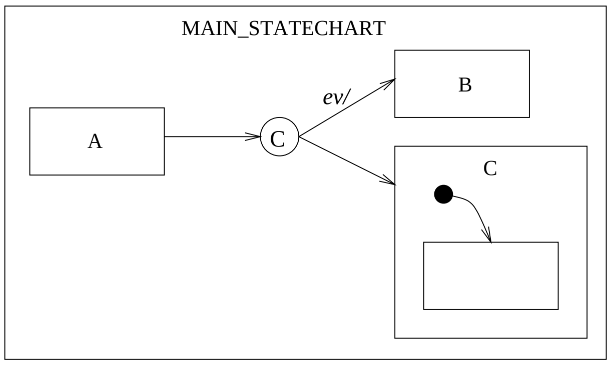


Figure 5.12: An example of the two full compound transitions with one of them without a trigger

with empty triggers by supplying a statechart with inputs which cause no transition with nonempty triggers to become enabled.

Unexpected chains of transitions are possible

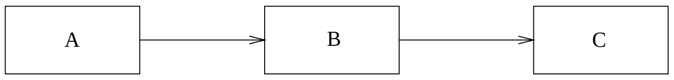


Figure 5.13: Full compound transitions without triggers

If we have a full compound transition which has no trigger (Fig. 5.13), the first transition is immediately followed by another one and thus the statechart takes two transitions instead of one. As a consequence, states B and C essentially ‘squash’ into one and we can neither apply other transitions from B nor identify with W or visit it with state cover. Such behaviour contradicts Req. 3c. A possible solution is described in Sect. 5.2.7 on p. 93 (a description of Req. 3c).

Augmentation to remove empty triggers

Non-default transitions of full compound transitions with empty triggers can be augmented to make empty-triggered transitions well-behaved. Default connectors cannot be augmented since we try to apply every non-default transition from all states (Sect. 2.2 on p. 29). The transition between A and C will be applied from states B and C and thus would still have an empty trigger. In addition, default transitions with triggers may cause problems (Req. 4b) and would generally lead to a larger test set (Sect. 3.2.1 on p. 48).

5.2.6 Racing between transitions is not allowed (Req. 3b)

Racing [NH95, p. 24] is an erroneous condition during execution of a statechart model when two or more transitions taken in the same step change the same variable[1]. An example is provided in Fig. 5.14 where variable a is assigned contradictory values by two transitions (a' means the value of a after execution of the transition, refer to Sect. 1.4.3 on p. 14 for details). Such changes are considered to be in a contradiction since transitions taken

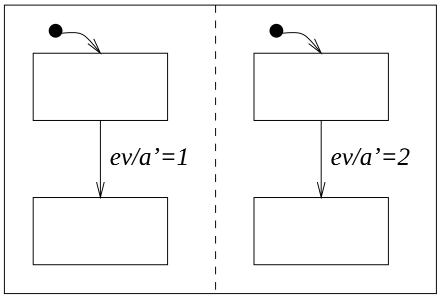


Figure 5.14: An example of racing

in a step do not have a specific order of execution. For this reason, such a statechart can be viewed as nondeterministic. Apart from FCTs, racing may also occur between parts of the same FCT or parts of a compound transition.

Cases of racing cause the following problems:

- outputs from transitions may mask each other, be it full compound transitions in concurrent statecharts or parts of a full compound transition; they can also do so if statecharts behave asynchronously, violating Req. 3c.
- racing between concurrent transitions may cause nondeterminism, because an order in which racing transitions are taken would affect the result. In some sense, this is the type of nondeterminism which we could allow as long as events or variables which are used during testing are not affected by racing. Note that the behaviour exhibited by a statechart when not under test could contain no racing since paths which are taken under test could be infeasible in normal operation. Despite this, we prohibit this phenomenon.

Although in general assignment of the same value to a variable by more than one transition taken in a step does not introduce the described problem, such a case is also prohibited; this is the behaviour of the Statemate tool.

[1]The case when a transition uses a variable modified by another one in the same step, is not considered erroneous.

The requirement for no racing is such that for every path in a statechart, transitions taken in the same step should exhibit no racing. In principle, it is possible to test for this kind of behaviour by taking complex full compound and concurrent transitions. If there is racing then the behaviour would be nondeterministic which could be detected assuming the fairness assumption described in Sect. 5.2.2 on p. 83 is satisfied. Statemate Analyzer can be used too.

5.2.7 Unexpected chains of transitions may not occur (Req. 3c)

We begin with the description of different approaches to test set application. Synchronicity requirement, as a possible approach to eliminate the problems encountered, is then described. Some alternatives to it are also provided, even though they were discarded.

Communication between a tester and a system under test

Consider the statechart in Fig. 5.15. The top statechart is the one under test and the bottom one — the statechart applying test sequences and monitoring changes. The big circle is the communication port between them and the double lines show system boundaries.

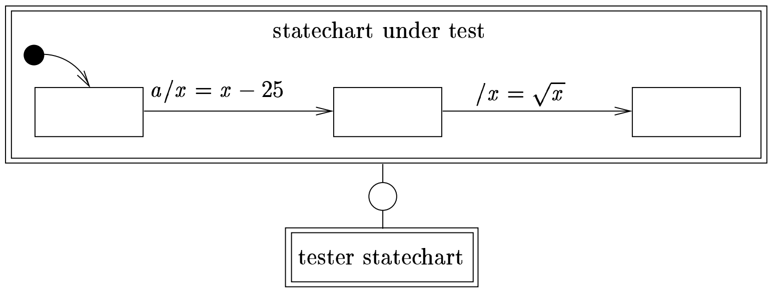


Figure 5.15: The tester which is not embedded in a statechart but communicates with it through a port

If we supply the statechart under test with some input, both transitions will take place and the output will be a combination of them. We cannot tell the sequence of transitions taken from outputs unless we introduce the *step* variable and augment every transition as shown in Fig. 5.16.

In general, statecharts may be allowed to exhibit some sequences of transitions, provided they do not mask changes made by each other. Racing covers transitions taken in the same step as depicted in Fig. 5.17[2] and is considered erroneous (Req. 3b); here we are talking about transitions taken

[2]Fig. 5.17 differs from Fig. 5.15 by having a **C** connector instead of a state in the middle of it.

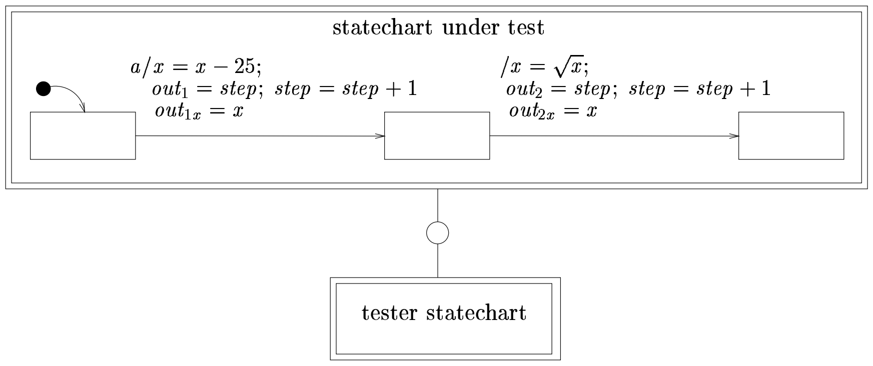


Figure 5.16: An augmentation to tell a sequence of transitions from outputs

in separate steps. If this holds, then provided we can enter every state and then try every transition from it, such statecharts can be tested. An extension of the testing method in this case may follow [Hie97d] and can be done in future.

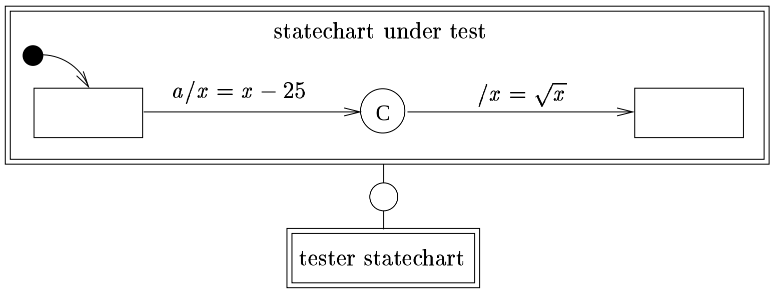


Figure 5.17: An illustration of masking between two transitions

For test application shown in Fig. 5.15, it rather difficult to tell a sequence of transitions occurred from outputs as well as to force a statechart through a desired path. The task becomes significantly easier if we embed the tester into the statechart under test as shown in Fig. 5.18 and have a static reaction in the tester statechart generate and sense variables at every step[3]. Such embedding allows the tester to observe changes and generate inputs on a step-by-step basis rather than a superstep one. Such embedding would require considerations of specifics of statemate semantics, refer

[3]When talking about testing in general, we suppose a tester to interact with a statechart under test via externally available variables. Embedding it in the statechart under test does not contradict it. First of all, an interface allows us to reason which variables may be affected by the tester and possibly set them to some values when testing is finished. Second, embedding is only supposed to allow the tester to sense the ports it has access to on the step rather than superstep basis.

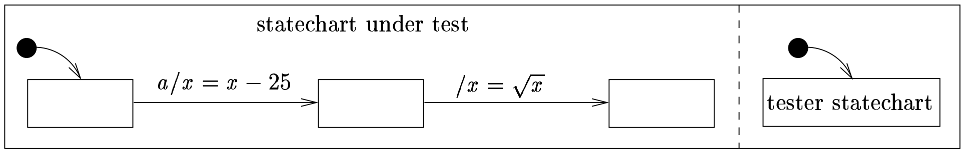


Figure 5.18: The tester embedded in the statechart under test

to Sect. 1.4.10 on p. 23 for details. The primary disadvantage of the approach of embedding is that it is rather intrusive. We expect to be able to interact with the implementation at the very low level. Above, the chain of transitions is shown. Applying the approach of embedding, we can cause a different path to occur which is the prime advantage of embedding. On the other hand, it can make a statechart behave contrary to the step semantics.

In this work we introduce a requirement of a synchronous operation of a statechart under test and apply the test set without embedding. The summary of the different approaches to embedding a tester in a statechart under test is given in App. A on p. 267.

The requirement of a synchronous behaviour under test

When we apply an element of a test sequence, the necessary labels get triggered and we observe the output. These labels may generate events and change variables, triggering other labels. In asynchronous semantics, chains of transitions occur while in synchronous we are given a chance to influence the execution. The problem of chains of transitions is solved by an introduction of a requirement of a synchronous behaviour of a statechart under test. This requirement does not prevent labels of transitions we do not wish to take from being triggered; t_completeness is responsible for that.

When deciding how to trigger a set of labels corresponding to the next element of a test sequence, we have to consider modifications made by labels executed when we triggered those of the previous element of the test sequence. This is necessary to ensure that no labels other than those we expect to take, are triggered (note that here we talk about triggerability rather than enableness since in a faulty implementation any transition may go from any state). In the examples given above, such as the one in Fig. 5.15, it is not possible without an appropriate design for test.

Making triggered labels not triggered is included in the t_completeness requirement (Req. 2); indeed, it says 'those we do not wish to trigger, will not be triggered'. For this reason, having a set of labels to trigger and those already triggered, we can find a set of events to generate and variables to change such that the desired set of labels and only this set will be triggered. In order to make this possible, we could augment labels with negative events, such as $\neg\ \underline{not\_triggered} \wedge label$. In this case, we can generate the $\underline{not\_triggered}$ event to prevent this label from being triggered even if the precondition of *label* is satisfied. An alternative approach involves providing the tester with an ability to make events undefined and thus influence which labels are triggered. This approach violates the semantics of statecharts in addition to being a rather intrusive one.

Note that when deciding which variables (persistent and volatile) to change in order to trigger a given set of labels, we have to consider all labels Φ of the considered statechart, irrespective of state transition diagram. This follows again from a need to accommodate a faulty implementation where any transition may go from any state (subject to refinement).

Alternative approaches to eliminate chains of transitions

The t_completeness requirement is significantly more strict than the one for X-machines, where the clause about not triggering other transitions is absent. It was unnecessary to have it for X-machines since their behaviour is strictly synchronous on input symbols, i.e. an X-machine cannot trigger a transition without an input being supplied while a statechart can, regardless whether we consider synchronous or asynchronous step semantics.

In order to tackle the problem of chain reactions without usage of the power of t_completeness to stop transitions from being triggered, we can introduce a notion of a *supersynchronous* semantics where transitions taken in a step do not trigger any others and thus in some sense we are free to trigger those we would like to. We say ‘in some sense’ because, although no labels are triggered, we cannot say anything about undesired labels when we trigger those we would like to. Consider, for instance, labels $a \wedge b \vee c/$ and $b/$. We could decide to use c and b respectively to trigger them. Unfortunately, if event a was generated in the previous step, then, although no label is triggered, the first one will be when we generate event b. For this reason, even in this case we have to consider already generated events when deciding how to trigger labels.

A further restriction of semantics of statecharts can require statecharts under test not to make any changes to variables which we would have to take into consideration when triggering labels to be taken in the next step. This may be accomplished by the tester being able to remove changes to variables which may cause a dependency, prior to applying those necessary to trigger the desired labels. For example, for the above two labels the tester would stop the a event from being generated (provided it is not necessary for triggering the desired label) because it introduces the possibility of the described problem. Consider a model with not only the above two labels but also the third one, $a \wedge d/$, which we would like to trigger, but none of the former two. In this case, both a and d events have to be generated and thus a does not have to stopped from being generated.

Even in the case where we can make the above assumption, it is not possible to eliminate the second clause of t_completeness requirement since statecharts may have transitions with labels having intersecting preconditions on different levels in state hierarchy. When triggering one of them we would also have to ensure that another label is not triggered.

From the description of the two alternative approaches to handling the problem of undesired chains of transitions, we can observe that neither allows us to reduce the number of requirements for testing or relax existing ones. For this reason, they are not considered in the rest of the thesis.

5.2.8 Correct implementation of transition labels and step semantics (Req. 4a)

Correctness of the implementation of individual labels

The testing method requires the correct implementation of labels of transitions. If it is not so, the method might fail. Consider the following faulty implementation of the *play* transition:

Faulty play

$\Xi\, UserButtons$

$\Delta(\underline{ff\_direction}, operation)\, CommunicationToMechanics$

$\mathsf{df}\ \underline{button\_play} \land \neg\ \mathsf{df}\ \underline{button\_stop}$

$operation' = play$

$TapeCounter < 2000 \Rightarrow \underline{ff\_direction'} = \underline{ff\_direction}$

$TapeCounter \geq 2000 \Rightarrow \underline{ff\_direction'} = FALSE$

An implementation with such a *play* transition is likely to pass testing while being faulty. We can, however, test all Z labels separately before using the proposed testing method. It could reveal this fault.

Some implementations may split transitions and errors may lead to different parts of the same transition entering different states. With that, the possible two-stage approach is described in Sect. 2.3.2 on p. 41.

A few groups of third year students were asked to choose a simple system, design it with X-machines, implement and evaluate the ability of the testing method to find seeded faults. Results have shown [CT98] that selection of test inputs which may occur during the operation of the system, rather than artificially added ones, allows us to find many faults even in the implementation of transition labels. It also appeared [KE98] to be better to combine test sequences together rather than reset an implementation after application of every of them which is very similar to what is said on that in [FvBK+91]. For instance, it is harder to find a fault of an alarm clock not going to 0:0 after 23:59 if we do not run it long enough.

Correct implementation of compound transitions

As the method is developed under assumption that **C**, **S**, junction, fork and joint connectors are absent (Sect. 3.1 on p. 46), we have to assume that compound transitions are implemented correctly. In future, this restriction can be removed; the outline of work is given in Sect. 6.8 on p. 206.

Transitions implemented with wider domains

Some transitions may be implemented to occur on a wider set of inputs (i.e. have *wider domains*) than stated on an original design. Let us denote some label in a design as f and in an implementation as f'. Then this can be put down as

$$\forall m : \mathsf{dom}\, f \bullet m \in \mathsf{dom}\, f' \wedge f'(m) = f(m)$$
$$\mathsf{dom}\, f \subset \mathsf{dom}\, f'$$

Although statecharts are completely specified, completeness is implicit via 'do nothing' static reactions. If all static reactions are made explicit we can show that wider domains cannot occur in an implementation. Indeed, from every state with an outgoing transition labeled by f, there is a transition g in a design such that $\mathsf{dom}\, g \cap (\mathsf{dom}\, f' \setminus \mathsf{dom}\, f) \neq \varnothing$ leading to nondeterminism because g is expected to be implemented without narrowing of its domain. Note once again that such a conclusion is only the case when all static reactions are made explicit, i.e. there are no 'do nothing' ones.

Correct implementation of step semantics

In addition to correct implementation of transition labels, the testing method requires correct implementation of step semantics. This could be tested to a certain extent by triggering multiple conflicting transitions in the same step and verifying that the one with the higher priority is taken. Testing of step semantics could be considered as a subject for further research.

5.2.9 Correct implementation of full compound transitions (Req. 4b)

In this section we shall illustrate certain problems related to default connectors and different approaches for handling them.

The test method described uses an assumption that an implementation may have subsets of full compound transitions implemented but initial compound transitions are always included. Consider the statechart in Fig. 5.19. We assume that the full compound transition

$$rew\_or\_ff\text{-}\ rew\text{-}\ init\_mechanism$$

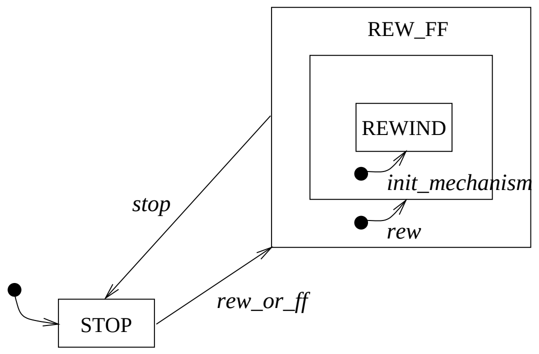


Figure 5.19: An example of a statechart with deeply nested states

can be implemented as any of

$$\begin{array}{l} rew\_or\_ff \text{ - } rew \text{ - } init\_mechanism \\ rew\_or\_ff \text{ - } rew \\ rew\_or\_ff \end{array}$$

or even

$$rew\_or\_ff \text{ - } init\_mechanism$$

but not either of

$$\begin{array}{l} stop \text{ - } rew \\ rew \text{ - } init\_mechanism \end{array}$$

since in our statechart *rew* cannot be a part of any full compound transition beginning from $STOP$ and an implementation of a full compound transition has to contain an original initial transition; *rew*, *init_mechanism* are both continuation CTs.

The considered requirement means that if there is a default transition in some state, it is not implemented on its own but only as a part of transitions which are supposed to enter the considered state. The following example demonstrates a possible problem when this assumption is not true.

In the Fig. 5.20, there is an incorrect implementation on the right which does not get detected by the test method under our assumptions, because even though we trigger the *rec* transition from the $RECORD$ state, the REW_FF state does not get entered because the default transition is not triggered.

From the considered requirement it follows that, for example, if there is an interlevel transition entering the REW_FF state in a faulty implementation, it would enter some state within it and not go through any default

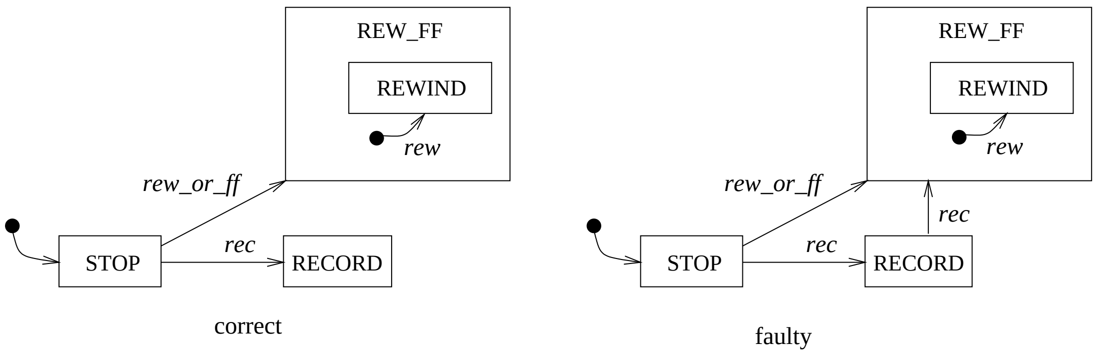


Figure 5.20: An example of a problem with default connectors

transition. This implies that triggering it, but none of the default transitions, would allow us to take such a transition. For this reason, *stop-rew* was considered to be absent from an implementation of the design in Fig. 5.19.

If we wish to remove this assumption (Req. 4b) and assume that any combinations of ordinary transitions with default ones are possible, we have to do either of the following:

- make it possible for test sequences to enter every state we would like, since as shown it is not sufficient just to be able to trigger an appropriate full compound transition (happily, we can always leave any state we are in). This can be done by making default transitions enabled as a part of the design for test, but may lead to nondeterminism. For instance, if we do that for our tape recorder, both *rew* and *ff* labels would have to be always triggered since they label default transitions. Unfortunately, we then get a chain reaction in the *REW_FF* state where the considered labels are also used on ordinary transitions.

- keep default transitions in mind when making a set of test cases. This means we essentially 'multiply' all labels with non-empty labels of default transitions, denoted $deflt = \{default_1, default_2, \ldots, default_n\}$. More formally, let $TST(\Phi) = \{t : \Phi \bullet \langle\{t\}\rangle\}$. Then

$$\Phi_{ok} = (TST(\Phi) \cup TST(deflt))\underline{*}TST(deflt)\underline{*}\ldots\underline{*}TST(deflt)$$

  where we have to assume some number of elements in the multiplication, related to the depth of the state hierarchy tree of a statechart under test. In most cases, the size of the product will render this approach infeasible (the above is the same as testing many concurrent states without refinement!).

- use a combination of the two approaches above, i.e. design for test some labels and multiply the remaining ones.

- use an assumption that if we trigger a transition which enters a state with no default transition triggered, an error is reported. For instance, if we trigger *rew_or_ff* in our example of the tape recorder, it would enter the *REW_FF* state and terminate with an error if none of *rew* or *ff* is enabled.

  The code generator of Statemate tool works the described way. The author also expects triggers on default transitions to be used to express preconditions[4] and their actions to initialise a substate statechart. This provides some justification for such an assumption.

Removal of the requirement 4b could be considered in future work.

5.2.10 A transition cannot enter a default connector explicitly (Req. 4e)

Consider the statechart in Fig. 5.21. It does not make much sense since a transition can be drawn to the border of the *REW_FF* state. If we wish to express that a precondition of an operation performed by a default transition should be made explicit, this can be accomplished as given on the right of Fig. 5.21. This requirement is also stated by Harel [NH95, p.9].

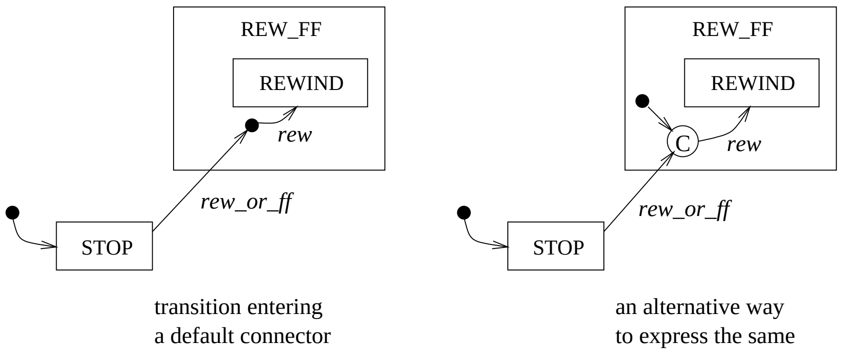

REW_FF
REWIND
rew
rew_or_ff
STOP
transition entering a default connector
REW_FF
REWIND
C
rew
rew_or_ff
STOP
an alternative way to express the same

Figure 5.21: A transition cannot explicitly enter a default connector

Potentially, it is possible to allow such transitions entering default connectors explicitly. They will then be considered interlevel by the testing method; alternatively, we could consider the entering transition to terminate at the state border and test it as a non-interlevel one.

5.2.11 Transitions from default connectors cannot leave the state within which they begin (Req. 4f)

This restriction is not documented explicitly in [NH95, MLPS97]. Without it we would be able to enter and exit a state in the same step as shown in

[4]similar to the `assert` statement in programming languages `C` and `C++`.

Fig. 5.22 and such behaviour is not allowed in general [NH95, p.30]. The restriction is also the behaviour of the Statemate tool.

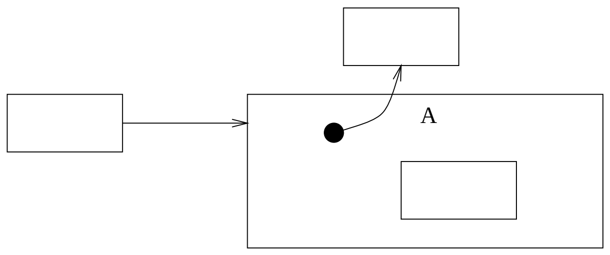


Figure 5.22: The prohibited default transition

5.2.12 Transitions also cannot go from or terminate at immediate substates of an AND-state (Req. 4g)

This requirement is not described in [MLPS97] but is seemingly obvious since we cannot even draw transitions which go from a specific concurrent component such as *CONTROL* or *SEARCH* and not the whole AND-state (Fig. 1.10).

5.3 Transitions which could be treated the same

In this section we shall describe the special case of statecharts which causes violation of a number of testing requirements and a possible solution to it.

5.3.1 i_same transitions

Usually, transitions which have the same label are assumed to be implemented by the same code (which could be accomplished with generic statecharts or using template classes in C++) and thus have identical behaviour. Changes to one of them by augmentation will affect another one, e.g. if we add a trigger to one of them, the other one will become triggerable by it too. Here we call such transitions *i_same* (*i*mplementation-*same*). Note that transitions which have the same functionality do not have to be i_same since one of them could be represented by a replica of another transition rather than by shared code. Consequently, we have to decide which transitions can be considered to be implemented by the same code. This is shown in Fig. 5.23. Transitions a b between states A, B and C, D could be considered to be i_same. A user can help a tool here since treating different transitions as i_same or those which are i_same as different ones can be disadvantageous:

- if we erroneously decide a pair of transitions to be i_same, we assume that we cannot distinguish between them and they will have the same

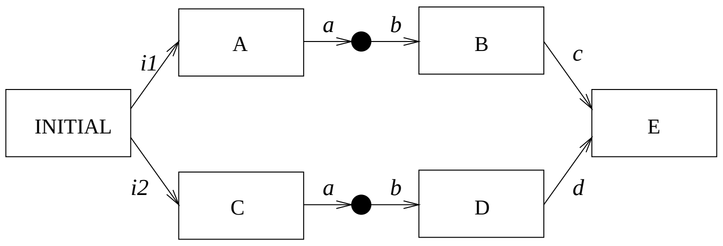


Figure 5.23: An example of transitions which could implemented by the same code

trigger and output. Thus in a statechart with such transitions, the characterisation set W could be longer. For example, in order to distinguish between states A and C in Fig. 5.23, we have to use the sequence a-b c instead of a_{top}-b_{top} where the a_{top}-b_{top} denotes the upper compound transition consisting of the a-b transitions.

- if a pair of transitions which is i_same is considered to be not i_same, it may become very difficult to select the triggering input and an appropriate output. For example, if transitions a-b in the above statechart, on top and bottom, are treated as different, a_{top}-b_{top} could be used to distinguish between states A and B. If they are, in fact, i_same, they would have to behave differently when supplied with the same input (triggering the top or bottom one). Consequently, we would have to make such transitions 'aware' of which, top or bottom transition is being triggered which could make augmentation for the design for test complicated. This approach is described in Sect. 5.2.4 on p. 89.

5.3.2 Consideration of parts of compound transitions

So far, we illustrated issues related to treating transitions i_same or not, using CTs. What has been said also applies to parts of them, as shown in Fig. 5.24. It is possible to treat transitions with label b in both transitions

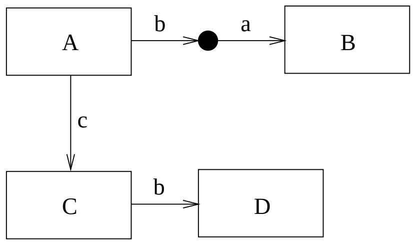


Figure 5.24: Parts of compound transitions which could be treated i_same

to be i_same. Using such a fine-grained control mechanism over the parts of transitions could allow users to make better decisions. For example, for the statechart to comply with the design for test condition, the output from the transition a has to be different from that of b and triggers could be the same. If we consider b-a as a whole transition, this decision could be more difficult to arrive at. In real statecharts, transitions could consist of even more parts making such operations on compound transitions relatively more complex than when looking at individual transitions. It is possible to go even lower where individual disjuncts in the DNF representation of transitions are considered, treating identical ones to be i_same and using this to judge if individual transitions or compound ones are the same.

5.4 A summary of solutions to problems with shared transitions and non-minimality

A number of problems caused by shared transitions have been pointed out in sections 5.2.1 and 5.2.4. The incremental approach proposes construction of auxiliary sets for parts of the design and combining them. Such auxiliary sets can be constructed for every substate statechart, ignoring the content of its substates. Sometimes, the substate statecharts cannot be considered separately because they have a strong relation to their parent statecharts. It could mean that many transitions in them share their schemas with that of the main statechart, interlevel transitions are overused, much processing is going on in static reactions or in substates. In this section we describe what can be done to solve these problems. Since all the proposed approaches are considering statecharts prohibited by requirements, not much detail on solutions is provided.

1. Flattening the whole design solves all stated problems. Regarding what has been said in Sect. 1.4.5 on p. 16 about the incremental approach and refinement (Sect. 3.2.3 on p. 52), we seek methods to avoid global flattening.

2. We could flatten OR states containing shared transitions. This is done without looking inside that substate's OR states. For example, when flattening C in Fig. 5.8, we would flatten *REW_FF* but not *F_ADVANCE* and *REWIND*, even if they were not basic. The test case basis would then be recomputed for the resulting statechart.

3. If considered transitions with the same label can be augmented individually, we no longer need to consider them to be shared; refer to Sect. 5.3 on p. 102 for details.

4. Sets C and W could be constructed such as not to include shared transitions. In such a case, the testing method could be applied and would provide some of the expected benefits. This is mentioned in Req. 1d.

5. As an extreme case, one could only design statecharts with all their transitions having different labels. In order to distinguish states of such statecharts, we would have to include one transition from every state but one in W; all such transitions would be different resulting in W containing $n - 1$ elements. For example, for a 3-state machine we would need W containing 2 sequences of one transition each. This W could be much bigger than the theoretically possible smallest one (Sect. 2.4.2 on p. 42).

Chapter 6

Proofs for the testing method

In order to do formal testing, we need to formalise statecharts. Unfortunately, despite many formal semantics including that of automata compositions [Mar92, MLS97] modecharts [PSM96] and statecharts [MLPS97, GK96, MSPT96, HGdR88, Day93, HMLS98, HRdR92] being developed, all of them consider aspects of statecharts other than those we are interested in. The difference is mainly that usually formalisation serves a purpose of model-checking specifications and details of behaviour of transitions are included in formalisations. In our case, we do not consider behaviour of transitions in detail since it is assumed to be correct (Req. 4a) and look in detail at the transition structure. This focus on transition diagrams and default connectors is absent from most formalisations considered, only [GK96] bears some similarity to this work but lacks many useful propositions. For this reason, another formalisation was developed in order to prove the method to guarantee correctness on the basis of testing not revealing faults under testing assumptions. It is based on the higher-level semantics [MLPS97]. Static reactions and history connectors are not considered.

In order to show that the set of test cases generated by the incremental approach has the same fault detection ability as the set of test cases constructed from the flattened statechart, we prove that the test case basis constructed as a result of merging possesses the required properties when applied to the flattened chart as illustrated by Fig. 6.1. Test case generation is described in Sect. 2 on p. 28, merging — in Sect. 3 on p. 46 and in Sect. 6.4.3 on p. 182 and the flattening process is formalised further in Sect. 6.2 on p. 157. Then, the behavioural equivalence between transition systems of the design and implementation essentially follows from the Chow's proof [Cho78]. Since the test data construction method presented in the thesis essentially follows the requirements described in [IH97] (refer to proofs in Sect. 6.6 on p. 201), we can directly apply the proof of [IH97] which guar-

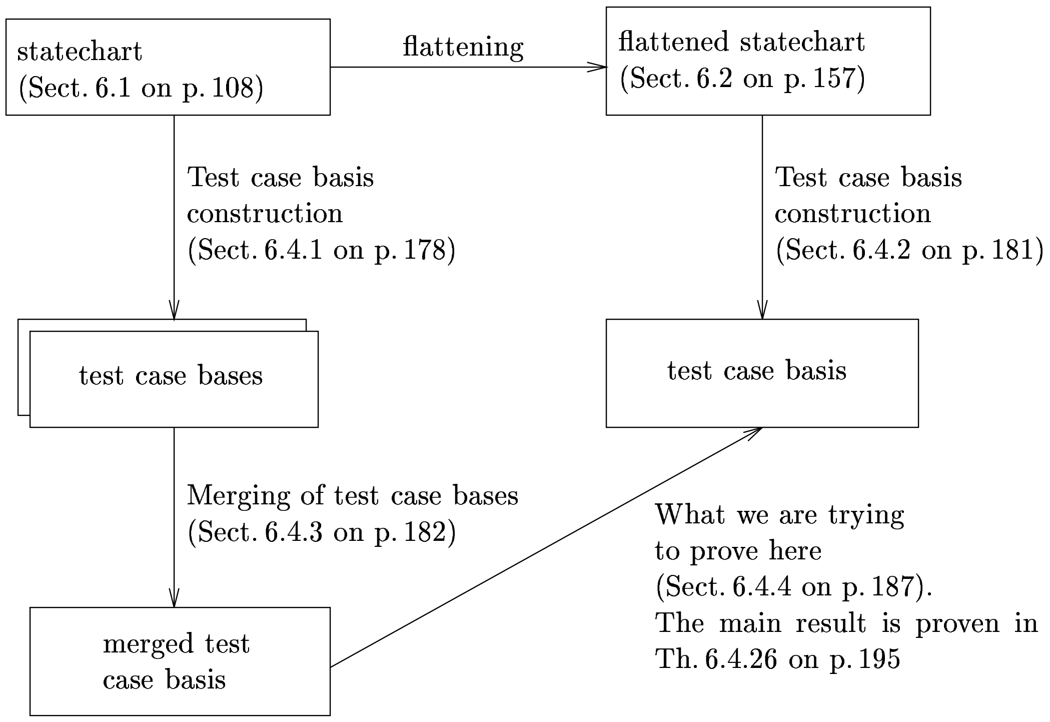

statechart
(Sect. 6.1 on p. 108)
flattening
flattened statechart
(Sect. 6.2 on p. 157)
Test case basis construction (Sect. 6.4.1 on p. 178)
Test case basis construction (Sect. 6.4.2 on p. 181)
test case bases
test case basis
Merging of test case bases (Sect. 6.4.3 on p. 182)
What we are trying to prove here (Sect. 6.4.4 on p. 187). The main result is proven in Th. 6.4.26 on p. 195
merged test case basis

Figure 6.1: The diagram which is to be shown to commute

antees the correct operation of the implementation w.r.t design. While providing an outline of it, we clarify the difference between an FSM used by Chow and an acceptor used by Ipate [IH97] and the author.

Z is used for expressions in order to facilitate comparison with other work and make type-checking possible. Below in proofs we use some extensions of Z [Toy98], namely expressions of the form $\exists_1 \, aa == expr$. This means that aa is an abbreviation of $expr$ such that, for instance, if we later write $c = func(aa)$, it is the same as $c = func(expr)$. In declarations of functions we have given types of arguments while predicate only considered meaningful values of variables of these types. For example, in Def. 6.1.2, we write

$$defaultfrom : SSet \rightarrow\!\!\!\!\!+ \; SSet$$

but then define it for $sset : \mathbb{F}\,\Sigma$ since it does not make sense to consider states not belonging to the considered statechart. Not all statements proven are necessary to show the main result. Unnecessary ones are useful to ascertain intuitive properties of statecharts.

6.1 Formalisation of the state transition system of a statechart

In this section we describe the formalisation [MLPS97] of the state transition system of a statechart and its extension to non-full compound transitions.

6.1.1 State hierarchy

States of a statechart are considered to be of type $STATE$ as given in [NH95, MLPS97] and form a tree with a root state *root*. *default*-type states are default connectors. The presence of the last two is necessary here to include non-full compound transitions[1]. The child relation of a state tree $\rho : STATE \twoheadrightarrow\!\!\!\!\!+ \; \mathbb{F}\,STATE$ provides a set of substates of a given state. The state tree for our tape recorder (Fig. 1.6) is given in Fig. 6.2 and the one for the AND-state in Fig. 1.10 — in Fig. 6.3. In Fig. 6.2, we make the main statechart state to be the root one while in Fig. 1.10, the state containing $CONTROL$ and $SEARCH$ states is an AND-one and we thus have to introduce a higher-level OR-state, which we will call $TAPE\_RECORDER$.

For a state, we might wish to consider it and a set of all states below it; the construct ρ^* is used for that. For example, in Fig. 6.2,

$$\begin{aligned} \rho^*(root) &= \{root, default\_TAPE\_RECORDER, PLAY, REW\_FF, STOP, \\ &\quad RECORD, default\_REW\_FF, F\_ADVANCE, REWIND\} \\ \rho^*(PLAY) &= \{PLAY\} \end{aligned}$$

[1][MLPS97] has only one *init* at the top level since it does not consider non-full compound transitions.

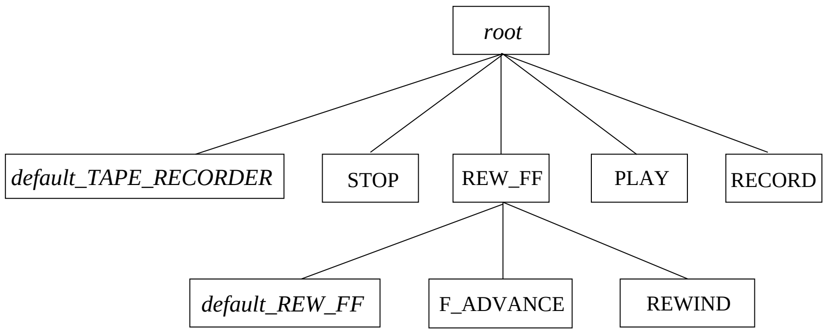


Figure 6.2: The state tree for the tape recorder

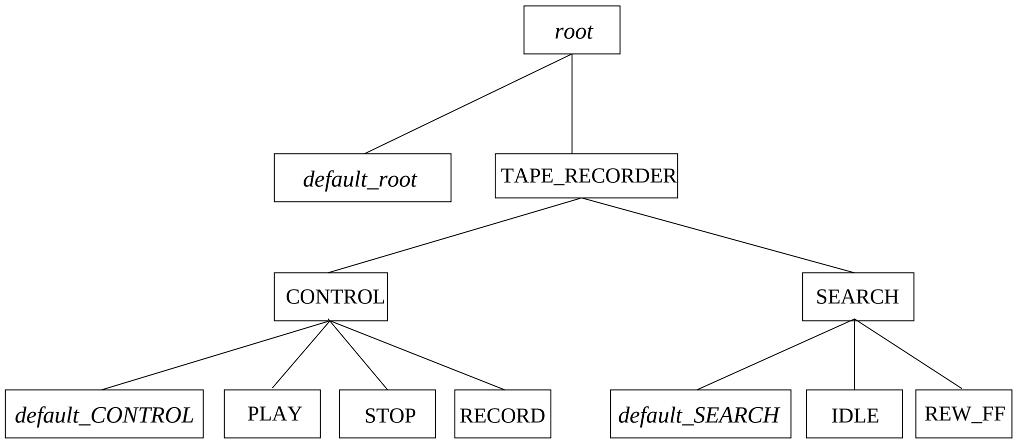


Figure 6.3: The state tree for the tape recorder with an AND-state

ρ^+ is different from ρ^* in that it does not include the state it was applied to:

$$\begin{aligned}\rho^+(root) &= \{default\_TAPE\_RECORDER, PLAY, REW\_FF, STOP,\\ &\qquad RECORD, default\_REW\_FF, F\_ADVANCE, REWIND\}\\ \rho^+(PLAY) &= \varnothing\end{aligned}$$

We postulate the type $STATE$ and possible types of states; $CONNECTOR$ type implies non-default connectors which we do not consider until Sect. 6.8 on p. 206.

$[STATE]$
$TYPE ::= stateBASIC \mid stateAND \mid stateOR \mid$
$\quad CONNECTOR \mid connectorDEFAULT$

The definition of the state tree as given in [MLPS97] includes predicates which ensure that the state tree is indeed a tree. They are provided here with appropriate changes to reflect the statecharts we operate on. The type of a state $TYPE$ is given by the function ϕ.

Definition 6.1.1. *A state tree of a statechart is defined as follows:*

$$
\begin{array}{|l}
root : STATE \\
\rho, \rho^*, \rho^+ : STATE \twoheadrightarrow\!\!\!\!\!\!\!\!\;\,\, \mathbb{F}\, STATE \\
\phi : STATE \twoheadrightarrow TYPE \\
parent : STATE \twoheadrightarrow STATE \\
\Sigma : SSet \\
\hline
\mathrm{dom}\,\rho \setminus \bigcup(\mathrm{ran}\,\rho) = \{root\} \wedge \bigcup(\mathrm{ran}\,\rho) \subseteq \mathrm{dom}\,\rho \wedge \phi(root) = stateOR \\
\forall\, set : \mathbb{F}(\bigcup(\mathrm{ran}\,\rho)) \bullet (\exists\, el : set \bullet (\forall\, st : set \bullet el \notin \rho(st))) \\
\forall\, s : \bigcup(\mathrm{ran}\,\rho) \bullet (\exists_1\, anc : \mathrm{dom}\,\rho \bullet s \in \rho(anc) \wedge parent(s) = anc) \\
\mathrm{dom}\, parent = \bigcup(\mathrm{ran}\,\rho) \\
\\
\forall\, st : STATE \bullet \\
\quad (\phi(st) = stateAND \Rightarrow \\
\quad\quad (\forall\, s : \rho(st) \bullet \phi(s) = stateOR)) \wedge \\
\quad (\phi(st) = stateOR \Rightarrow (\exists_1\, d : \rho(st) \bullet \phi(d) = connectorDEFAULT)) \wedge \\
\quad ((\phi(st) = stateBASIC \vee \phi(st) = connectorDEFAULT \vee \\
\quad\quad \phi(st) = CONNECTOR) \Leftrightarrow \rho(st) = \varnothing) \\
\\
\Sigma = \mathrm{dom}\,\rho \wedge \mathrm{dom}\,\rho = \mathrm{dom}\,\phi \\
\forall\, st : STATE \bullet \rho^*(st) = \{st\} \cup \bigcup\{s : \rho(st) \bullet \rho^*(s)\} \wedge \\
\rho^+(st) = \rho^*(st) \setminus \{st\}
\end{array}
$$

The definition consists of three parts, the first one making sure ρ forms a tree (from [MLPS97]), the second asserting properties of states and the last one defining Σ-the set of states of a statechart, ρ^+, ρ^* and domains of tree operations. State properties are those given in [MLPS97] with addition of connectors. An AND-state should have OR substates and every OR — state should have a default connector. BASIC states and connectors cannot have substates. The set Σ is further assumed to be finite.

The following function helps to determine a default connector of a statechart in a state, passed to it as a parameter.

Definition 6.1.2.

$$
\begin{array}{|l}
\mathit{defaultfrom} : \mathit{SSet} \twoheadrightarrow\!\!\!\!\!\!\!\!\;\shortmid\;\;\;\; \mathit{SSet} \\
\mathit{defaultFROM} : \mathit{STATE} \twoheadrightarrow\!\!\!\!\!\!\!\!\;\shortmid\;\;\;\; \mathit{STATE} \\
\hline
\forall \mathit{sset} : \mathbb{F}\,\Sigma \bullet \\
\qquad \mathit{defaultfrom}(\mathit{sset}) = \{s : \mathit{sset} \mid \phi(s) = \mathit{connectorDEFAULT}\} \\
\forall s : \Sigma \mid \phi(s) = \mathit{stateOR} \bullet \\
\qquad \exists_1\, \mathit{st} : \rho(s) \bullet \phi(\mathit{st}) = \mathit{connectorDEFAULT} \wedge \mathit{defaultFROM}(s) = \mathit{st}
\end{array}
$$

A *configuration* is a set of states, excluding default ones, a statechart can be in simultaneously. For example, if we enter *F_ADVANCE* state, *FF_REW* should also be entered since it is a parent of *F_ADVANCE* in the state hierarchy. Additionally, no more than a single substate of an OR-state can be entered, for example we cannot be in *STOP* and *PLAY* states at the same time. Thus, possible configurations in Fig. 6.2 are $\{root, F\_ADVANCE\}$ and $\{root, STOP\}$ but $\{F\_ADVANCE, STOP\}$ is not. For concurrent components, some states in all of them should be entered; for example, if we enter the statechart in Fig. 1.10, a state inside each of the *CONTROL* and *SEARCH* states, such as *STOP* and *IDLE* also have to be entered. For this reason, allowed configurations include

$$\{root, TAPE\_RECORDER, CONTROL, SEARCH, STOP, IDLE\}$$

but not

$$\{root, CONTROL, STOP\}$$

These rules are given by the *configuration* relation between the top state of a state hierarchy and a set of states:

Definition 6.1.3. *Configuration of a statechart (essentially from [MLPS97])*

$$
\begin{array}{|l}
\mathit{configuration}\,\_ : \mathbb{F}_1(\mathit{STATE} \times \mathbb{F}_1\,\mathit{STATE}) \\
\hline
\forall \mathit{top} : \mathit{STATE};\ \mathit{conf} : \mathbb{F}_1\,\mathit{STATE} \bullet \\
\qquad \mathit{configuration}(\mathit{top}, \mathit{conf}) \Leftrightarrow \mathit{top} \in \mathit{conf} \wedge \mathit{conf} \subseteq \Sigma \wedge (\forall \mathit{state} : \mathit{conf} \bullet \\
\qquad\qquad \phi(\mathit{state}) \in \{\mathit{stateOR}, \mathit{stateAND}, \mathit{stateBASIC}\} \wedge \\
\qquad\qquad (\phi(\mathit{state}) = \mathit{stateOR} \Rightarrow \#(\rho(\mathit{state}) \cap \mathit{conf}) = 1) \wedge \\
\qquad\qquad (\phi(\mathit{state}) = \mathit{stateAND} \Rightarrow \rho(\mathit{state}) \subseteq \mathit{conf}) \wedge \\
\qquad\qquad (\mathit{state} = \mathit{top} \vee \mathit{state} \neq \mathit{top} \wedge (\exists\, \mathit{parent} : \mathit{conf} \bullet \mathit{state} \in \rho(\mathit{parent}))))
\end{array}
$$

The above definition excludes all connectors from a configuration. Since configuration is defined such that it contains a root state, it is nonempty and further, when talking about a configuration, we write, for instance,

$$conf : \mathbb{F}_1\,\Sigma \mid configuration(root, conf)$$

Proposition 6.1.4. *A parent of a non-root state in a configuration belongs to the configuration.*

$$\begin{aligned}&\forall\, rootstate : \Sigma;\ conf : \mathbb{F}_1\,\Sigma \mid configuration(rootstate, conf) \bullet\\ &\qquad (\forall\, s : \Sigma \bullet s \in conf \land s \neq rootstate \Rightarrow parent(s) \in conf)\end{aligned}$$

Proof. Is a restatement of the last line of the definition of a configuration (Def. 6.1.3). □

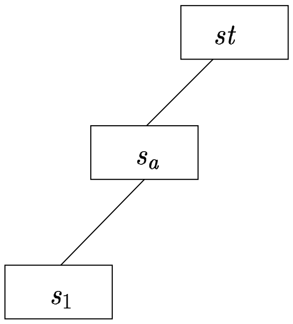


Figure 6.4: An illustration of the Prop. 6.1.5

Proposition 6.1.5.

$$\begin{aligned}&\forall\, st : \Sigma \bullet\\ &\qquad (\forall\, s : \Sigma \bullet s \in \rho^{+}(st) \Rightarrow (\exists_1\, s_a : \rho(st) \bullet s \in \rho^{*}(s_a)))\end{aligned}$$

This proposition is illustrated by Fig. 6.4.

Proof. By definition of ρ^{+}, there is $s : \rho(st) \bullet s_1 \in \rho^{*}(s)$. By the definition of the tree (Def. 6.1.1, line 3 of the predicate part), this s_1 is unique. □

Some useful relations from [MLPS97] are *Anc* (ancestor), *SAnc* (strict ancestor) *lca* (lowest common ancestor), *lcoa* (lowest common strict OR-ancestor) and *orth* (whether states are orthogonal or the same) are defined as follows.

Definition 6.1.6.

$$
\begin{array}{|l}
Anc\ \_, SAnc\ \_ : \mathbb{F}_1(STATE \times \mathbb{F}_1\, STATE) \\
lca, lcoa : \mathbb{F}_1\, STATE \twoheadrightarrow STATE \\
orth\ \_ : \mathbb{F}_1(STATE \times STATE) \\
orthset\ \_ : \mathbb{F}_1(\mathbb{F}\, STATE) \\
\hline
\forall\, anc : \Sigma;\ sts : \mathbb{F}_1\, \Sigma \bullet \\
\quad (Anc(anc, sts) \Leftrightarrow sts \subseteq \rho^*(anc)) \wedge \\
\quad (SAnc(anc, sts) \Leftrightarrow sts \subseteq \rho^+(anc)) \\
\forall\, sts : \mathbb{F}_1\, \Sigma;\ anc : \Sigma \bullet \\
\quad (lca(sts) = anc \Leftrightarrow Anc(anc, sts) \wedge \\
\qquad (\forall\, st : \Sigma \bullet Anc(st, sts) \Rightarrow Anc(st, \{anc\}))) \wedge \\
\quad (lcoa(sts) = anc \Leftrightarrow SAnc(anc, sts) \wedge \\
\qquad (\forall\, st : \Sigma \bullet SAnc(st, sts) \wedge \phi(st) = stateOR \Rightarrow Anc(st, \{anc\}))) \\
\forall\, s_1, s_2 : \Sigma \bullet orth(s_1, s_2) \Leftrightarrow (s_1 \notin \rho^*(s_2) \wedge s_2 \notin \rho^*(s_1) \wedge \\
\quad \phi(lca(\{s_1, s_2\})) = stateAND) \\
\forall\, sset : \mathbb{F}_1\, \Sigma \bullet orthset(sset) \Leftrightarrow \\
\quad (\forall\, s_1, s_2 : sset \mid s_1 \neq s_2 \bullet orth(s_1, s_2))
\end{array}
$$

From Def. 6.1.3, we can show the following:

Proposition 6.1.7. *Any set of orthogonal states is possible in some configuration,*

$$
\begin{array}{l}
\forall\, states : \mathbb{F}_1\, STATE \mid orthset(states) \bullet \\
\quad \exists\, conf : \mathbb{F}_1\, STATE \bullet configuration(root, conf) \wedge states \subseteq conf
\end{array}
$$

Proof. *configuration* restricts which states could be a part of it only with respect to OR-states, $\phi(state) = stateOR \Rightarrow \#(\rho(state) \cap conf) = 1$, this is always satisfied for orthogonal states. □

From Def. 6.1.6, propositions 6.1.8-6.1.10 follow:

Proposition 6.1.8. *Consider* $st = lca(s_1, s_2) \wedge st \neq s_1$*, then* $\exists_1\, s_a : \rho(st) \bullet s_1 \in \rho^*(s_a)$.

Proof. From $st = lca(s_1, s_2)$, it follows by definition of *lca* that $s_1 \in \rho^*(st)$. Since $st \neq s_1$, then $s_1 \in \rho^+(st)$ and the result follows from Prop. 6.1.5. □

We can also show the following:

Proposition 6.1.9. *If p and q are unrelated states, a child of p is unrelated to q.*

$$\begin{aligned}&\forall\, p, q : \Sigma \mid p \notin \rho^*(q) \wedge q \notin \rho^*(p) \bullet \\ &\quad (\forall\, s : \rho^*(p) \bullet s \notin \rho^*(q) \wedge q \notin \rho^*(s))\end{aligned}$$

Proof. The proof is illustrated with Fig. 6.5.

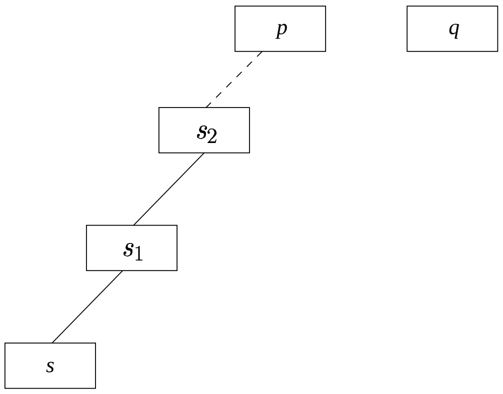


Figure 6.5: An illustration of the proof of Prop. 6.1.9

If $s = p$, the result follows from the assumption of the proposition.

Consider $s \in \rho^+(p)$, $s \in \rho^*(q)$. If $s = q$ then $q \in \rho^+(p)$ contradicting the assumption of the proposition. Thus, $s \in \rho^+(q)$ and by Prop. 6.1.5, $\exists_1\, s_1 = parent(s)$, $s_1 \in \rho^*(q)$, $s_1 \in \rho(p)$; by assumption of the proposition, $s_1 \neq p$, consequently, $s_1 \in \rho^+(p)$.

Treating s_1 similarly to s above, we get that $s_1 \in \rho^+(p) \wedge s_1 \in \rho^+(q)$ and $\exists_1\, s_2 = parent(s_1)$. This sequence of s_i is bounded by p since $\forall\, i \bullet s_i \in \rho^+(p)$ and $s_i \in \rho(s_{i+1})$. This leads us to a contradiction as the tree is finite.

Consider $s \in \rho^+(p)$, $q \in \rho^*(s)$. By transitivity of ρ^* (obvious from its definition), $q \in \rho^*(p)$ — a contradiction. □

Proposition 6.1.10. *Consider $st = lca(s_1, s_2)$, then if $s_1 \neq st \wedge s_2 \neq st$ then $\exists_1\, s_a, s_b : \rho(st)$ such that $s_1 \in \rho^*(s_a) \wedge s_2 \in \rho^*(s_b) \wedge s_a \neq s_b$. This is illustrated in Fig. 6.6.*

Proof. Let $st = lca(s_1, s_2)$. From Prop. 6.1.8, $\exists_1\, s_a, s_b : \rho(st) \bullet s_1 \in \rho^*(s_a) \wedge s_2 \in \rho^*(s_b)$. The inequality $s_a \neq s_b$ is from the definition of lca. □

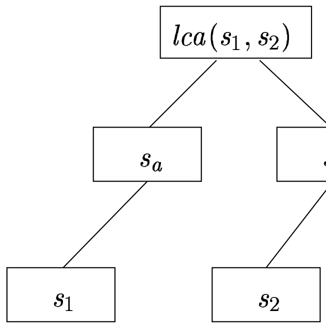


Figure 6.6: An illustration of the proof of Prop. 6.1.10 and Prop. 6.1.11

Proposition 6.1.11. *orth possesses the following properties:*

$\forall\, s : \Sigma \bullet \neg\, orth(s, s)$
$\forall\, s_1, s_2 : \Sigma \bullet orth(s_1, s_2) \Leftrightarrow orth(s_2, s_1)$
$\forall\, s_1, s_2 : \Sigma \bullet orth(s_1, s_2) \Rightarrow (\forall\, sub : \rho^*(s_1) \bullet orth(sub, s_2))$

Proof. The first two directly follow from the definition of *orth*.

Consider $and = lca(\{s_1, s_2\})$, then, from definitions of *lca* and *orth*, $\phi(and) = stateAND$, $s_1 \in \rho^*(and) \wedge s_2 \in \rho^*(and) \wedge s_1 \notin \rho^*(s_2) \wedge s_2 \notin \rho^*(s_1)$, from which follows that $s_1 \neq and \wedge s_2 \neq and$. Thus, by Prop. 6.1.10, $\exists\, s_a, s_b : \rho(and) \bullet s_1 \in \rho^*(s_a) \wedge s_2 \in \rho^*(s_b) \wedge s_a \neq s_b$.

$Anc(and, \{sub, s_2\})$ is satisfied for all $sub \in \rho^+(s_1)$ by definition of *Anc*. We now show that *and* is the lowest ancestor of them from the contrary. Consider the lower one l, such that $Anc(l, \{sub, s_2\}) \wedge l \in \rho^+(and)$, then $\exists\, st : \rho(and) \bullet l \in \rho^*(st)$ (Prop. 6.1.5); since $\forall\, s_c : \rho(and) \setminus \{s_b\} \bullet s_c \notin \rho^*(s_b) \wedge s_b \notin \rho^*(s_c)$, then by Prop. 6.1.9 we get that $s_2 \notin \rho^*(s_c)$. Thus for all those s_c, $s_2 \notin \rho^*(s_c)$ and consequently it only has to be $l \in \rho^*(s_b)$. By Prop. 6.1.9, $\forall\, sub : \rho^*(s_1) \bullet sub \in \rho^*(s_a) \wedge sub \notin \rho^*(s_b)$. We get that l cannot be an ancestor of *sub* and as a result, $and = lca(sub, s_2)$. □

In the following, the term *route* is used for sets of states in the state hierarchy, every two of them related with ρ^+.

Definition 6.1.12. *Route*

$$
\begin{array}{|l}
route : \Sigma \times \Sigma \twoheadrightarrow SSet \\
\hline
\forall\, rootstate, s : \Sigma \mid s \in \rho^*(rootstate) \bullet \\
\qquad s = rootstate \Rightarrow route(rootstate, s) = \{rootstate\} \land \\
\qquad s \neq rootstate \Rightarrow route(rootstate, s) = route(rootstate, parent(s)) \cup \{s\}
\end{array}
$$

From the definition of *configuration*, we get that

$$\forall\, conf : \mathbb{F}_1\, \Sigma \mid configuration(root, conf) \bullet (\forall\, s : conf \bullet route(root, s) \subseteq conf)$$

Consider a set of states *states* such that there is a configuration to which all of them belong. It may be that there is a number of such configurations; it is interesting to find out which states of *states* actually determine the set of possible configurations and for those states, which belong to a single configuration only, try to construct the smallest subset of them uniquely determining the configuration. With that in mind, a given set of states could be minimised without changes in the set of possible configurations containing it. The above questions are answered in the following theorem.

Theorem 6.1.13. *For a set of states in a valid configuration, we can try to define as small a subset of them from which that configuration can be reconstructed, as possible. The following subset is further considered:*

$$\{s : states \mid \rho^+(s) \cap states = \varnothing\}$$

The function reducing a given set of states is called treereduced. *Every pair of states in the set obtained as a result of an application of the treereduced function is orth. States in this set are basic.*

$$
\begin{array}{|l}
treereduced : SSet \twoheadrightarrow SSet \\
\hline
\forall\, states : \mathbb{F}_1\, \Sigma \bullet treereduced(states) = \\
\qquad \{s : states \mid \rho^+(s) \cap states = \varnothing\} \\
\forall\, conf : \mathbb{F}_1\, \Sigma \mid configuration(root, conf) \bullet \\
\qquad \phi(\!|\, treereduced\ conf \,|\!) = \{stateBASIC\} \land orthset(treereduced\ conf)
\end{array}
$$

Note that are not claiming that treereduced produces the smallest set.

Proof. We begin with a justification of the definition of *treereduced*.

- By definition of a configuration, if an AND-state is in a configuration, then all its substates are. If any child of any substate of an AND-state is in the configuration, the AND-state is and all its substates.

- For OR-states, one and only one of its substates can be entered. Consider $s : \Sigma$ such that $\phi(s) = \mathit{stateOR}$. If there is a substate s_1 of s in a configuration conf, then by Prop. 6.1.5, $\exists_1\, s_a \in \rho(s) \bullet s_1 \in \rho^*(s_a)$ implying that all states above s_1 up to the root (i.e. $\mathit{route}(\mathit{root}, s_1)$) are in the configuration. Consequently, there is more than one configuration possible for a set of states if there are some OR-states in it such that no states below them are entered.

States which are *treereduced* are *orth*: they are not related by ρ^* by construction and have their *lca* of type *stateAND* as otherwise there would be no valid configuration containing them.

States which are *treereduced* are basic: for a set of states defining a unique configuration, all OR-states in it have one of their substates entered. If we consider an AND-state entered, then one child of all substates of it should be entered. Descending from the root state, we get that such lowest-level states are basic ones.

Now we show that the reduced set of states defines a configuration uniquely. Assume that there is more than one configuration with the given set of basic states. We denote *states* to be the considered set of states, $\mathit{conf}_{\mathit{orig}}$ — the configuration it was constructed from
($\mathit{states} = \mathit{treereduced}(\mathit{conf}_{\mathit{orig}})$) and $\mathit{conf}_{\mathit{new}}$ — some different configuration, such that $\mathit{conf}_{\mathit{orig}} \neq \mathit{conf}_{\mathit{new}} \wedge \mathit{states} \subseteq \mathit{conf}_{\mathit{new}}$. *root* state will be contained in both of $\mathit{conf}_{\mathit{orig}}$ and $\mathit{conf}_{\mathit{new}}$ by definition of a configuration (Def. 6.1.3). We then descend from *root*, following states which are in $\mathit{conf}_{\mathit{orig}} \cap \mathit{conf}_{\mathit{new}}$ until we find a non-basic state $s : \mathit{conf}_{\mathit{orig}} \cap \mathit{conf}_{\mathit{new}}$ such that $\rho(s) \cap \mathit{conf}_{\mathit{orig}} \neq \rho(s) \cap \mathit{conf}_{\mathit{new}}$. Such a state s has to exist as otherwise $\mathit{conf}_{\mathit{orig}} = \mathit{conf}_{\mathit{new}}$. Moreover, s has to be an OR-state as connectors are not contained in configurations, basic states do not have substates and for an AND-state, $\rho(s) \cap \mathit{conf}_{\mathit{orig}} = \rho(s) \cap \mathit{conf}_{\mathit{new}} = \rho(s)$. As s is an OR-state, there are states $s_1, s_2 : \rho(s) \bullet s_1 \neq s_2$, such that $s_1 \in \mathit{conf}_{\mathit{orig}} \cap \rho(s) \wedge s_2 \in \mathit{conf}_{\mathit{new}} \cap \rho(s)$ since only a single substate of an OR-state may be included in a configuration. Sets $\rho^+(s) \cap \mathit{conf}_{\mathit{orig}}$ and $\rho^+(s) \cap \mathit{conf}_{\mathit{new}}$ are both non-empty (due to s_1 and s_2) and non-intersecting from definitions of a configuration and a state tree (Def. 6.1.1). It means that *treereduced* function, applied to $\mathit{conf}_{\mathit{new}}$ has to include either s_2 or a state below it, i.e. $\rho^*(s_2) \cap \mathit{treereduced}(\mathit{conf}_{\mathit{new}}) \neq \varnothing$. At the same time, since $s_2 \notin \mathit{conf}_{\mathit{orig}}$, $\rho^*(s_2) \cap \mathit{conf}_{\mathit{new}} = \varnothing$ and consequently $\rho^*(s_2) \cap \mathit{treereduced}(\mathit{conf}_{\mathit{orig}}) = \varnothing$, which implies that the two configurations cannot be reduced to the same set of basic states. □

6.1.2 Transitions — basic definitions and properties

Data

The description of data of statecharts below was developed by the author to mimic that of X-machines. It explicitly models both persistent and event variables.

Transitions are activated by some input and manipulate variables of a statechart. Input and output are considered to be of type *CHANGE* and internal data — *DATA*. This *DATA* consists of variables of both persistent and event types; *CHANGE* is essentially an event which expresses changes to them. This notation is particularly useful when we try to express behaviour of multiple transitions taken in the same step. Due to step semantics, they have to operate on original data and the modifications made by all of them are combined. After an application of the combined changes to data, the result becomes an original data for the next step. For this reason, functional composition of behaviour of transitions cannot be used. In what follows we show (Th. 6.3.10) that the behaviour of a statechart under our testing assumptions is the same as that of some X-machine.

$[SPACE]$

SPACE is the combined data space of internal data and changes.

For any given statechart, we can define these sets. For example, consider a statechart with the following data, consisting of two variables, persistent and event one,

$$\begin{array}{|l}
SPACE^{sample}, DATA^{sample}, CHANGE^{sample} : \mathbb{P}_1\, SPACE^{sample} \\
len_{DATA^{sample}} : \mathbb{N} \\
\hline
DATA^{sample} = \mathbb{N} \times (\mathbb{N} \cup \{\bot_{sample}\}) \\
CHANGE^{sample} = (\mathbb{N} \times \{\bot_{sample}\}) \cup (\{\bot_{sample}\} \times \mathbb{N}) \\
len_{DATA^{sample}} = 2
\end{array}$$

where

$$\begin{array}{|l}
\bot_{sample} : \mathbb{Z} \\
\bot^{sample} : \mathbb{Z} \times \mathbb{Z} \\
\hline
\bot_{sample} \notin \mathbb{N} \\
\bot^{sample} = (\bot_{sample}, \bot_{sample})
\end{array}$$

and

$$SPACE^{sample} == (\mathbb{N} \cup \{\bot_{sample}\}) \times (\mathbb{N} \cup \{\bot_{sample}\})$$

$DATA^{sample}$ is the internal data; $\mathbb{N} \cup \{\perp_{sample}\}$ part of it represents an event. $CHANGE^{sample}$ is defined such that we can either set a variable or leave it unchanged.

For persistent variables, assignment means using a value of an appropriate type and $\perp$ — leaving a variable unchanged (we assume that $\perp$ is not a part of a type of any variable). Event variables can be set and if they are not, they lose their values in the following step. Note that if a statechart sets a value and environment then overrides it, this is not a case of racing since environment makes its changes after the statechart completes its changes.

We do not have to differentiate between ordinary variables and events since changes can be applied to both of them in the same way.

CHANGE is meant to be used for a modification of a single variable; multiple changes are supposed to be in a set of changes, *CSet*. No changes mean an empty set of changes.

$[SPACE]$

$| \; DATA, CHANGE : \mathbb{P}_1 \, SPACE$

$\perp : SPACE$
$CSet == \mathbb{F} \, CHANGE$
$CSetSet == \mathbb{F}(\mathbb{F} \, CHANGE)$

For example, environment's changes could be $\{(3, \perp_{sample}), (\perp_{sample}, 6)\}$ which means setting the persistent variable to 3 and generating the event with the value of 6.

For any given statechart, it is possible to define filtering, racing and modifications of data for changes. For that, we define function π which expresses a projection of a part of *SPACE* with the given number. The *index* function retrieves the number of the element of a *CHANGE* which is assigned by it, i.e. the one for which $\pi(index(change), change) \neq \perp$. From definition of *CHANGE* we can derive that

$$\forall \, i : \mathbb{N} \mid i \leq len_{SPACE} \wedge i \neq index(data) \bullet \pi(i, change) = \perp$$

Definition 6.1.14.

$$
\begin{array}{|l}
len_{SPACE} : \mathbb{N} \\
\pi : \mathbb{N} \times SPACE \nrightarrow SPACE \\
index : CHANGE \rightarrow \mathbb{N} \\
filter : CSet \times \mathbb{F}\,\mathbb{N} \nrightarrow CSet \\
racing\ \_ : \mathbb{F}_1(CSet \times CSet) \\
modify : CSet \times SPACE \nrightarrow SPACE \\
\hline
\forall\, cset : CSet;\ what : \mathbb{F}\,\mathbb{N} \mid what \subseteq 1\,..\,len_{SPACE} \bullet \\
\quad filter(cset, what) = \{c : cset \mid index(c) \in what\} \\
\forall\, cset_1, cset_2 : CSet \bullet racing(cset_1, cset_2) \Leftrightarrow \\
\quad (\exists\, c_1 : cset_1;\ c_2 : cset_2 \bullet index(c_1) = index(c_2)) \\
\forall\, cset : CSet;\ data : SPACE \bullet \\
\quad (\forall\, c : cset \bullet \pi(index(c), modify(cset, data)) = \pi(index(c), c)) \wedge \\
\quad (\forall\, i : 1\,..\,len_{SPACE} \mid \neg\,(\exists\, c : cset \bullet i = index(c)) \bullet \\
\qquad \pi(i, modify(cset, data)) = \pi(i, data))
\end{array}
$$

We cannot define racing with a single set of changes since if we have those generated by two transitions, identical changes to the same variable have to be flagged as erroneous. Uniting sets of such changes will make such errors go undetected.

For the example of the two variables given above, we can define

$$
\begin{array}{|l}
\pi^{sample} : \mathbb{N} \times SPACE^{sample} \pfun CHANGE^{sample} \\
index^{sample} : CHANGE^{sample} \rightarrow \mathbb{N} \\
modify^{sample} : \mathbb{F}\, CHANGE^{sample} \times DATA^{sample} \pfun DATA^{sample} \\
\hline
\forall e : SPACE^{sample} \bullet \pi^{sample}(1, e) = (first(e), \bot_{sample}) \land \\
\qquad \pi^{sample}(2, e) = (\bot_{sample}, second(e)) \\
\forall c : CHANGE^{sample} \bullet \\
\qquad (c \in \mathbb{N} \times \{\bot_{sample}\} \Rightarrow index^{sample}(c) = 1) \land \\
\qquad (c \in \{\bot_{sample}\} \times \mathbb{N} \Rightarrow index^{sample}(c) = 2) \\
\forall cset : \mathbb{F}\, CHANGE^{sample};\ data : DATA^{sample} \bullet \\
\qquad ((\exists c : cset \bullet index^{sample}(c) = 1 \land \\
\qquad\qquad first(modify^{sample}(cset, data)) = first(c)) \lor \\
\qquad \neg (\exists c : cset \bullet index^{sample}(c) = 1) \land \\
\qquad\qquad first(modify^{sample}(cset, data)) = first(data)) \\
\qquad \land \\
\qquad ((\exists c : cset \bullet index^{sample}(c) = 2 \land \\
\qquad\qquad second(modify^{sample}(cset, data)) = second(c)) \lor \\
\qquad \neg (\exists c : cset \bullet index^{sample}(c) = 2) \land \\
\qquad\qquad second(modify^{sample}(cset, data)) = second(data))
\end{array}
$$

Transition labels

LABEL contains preconditions and data transformations performed by transitions. A transition can fire if a statechart is in its source state and the supplied input is in the domain of that transition. The transition would indeed occur provided no transition with a higher priority is also enabled as outlined in Sect. 1.4.5 on p. 16 and formalised in Sect. 6.1.4 on p. 153.

$LABEL == DATA \pfun CSet$

Definition 6.1.15. *We use an and operation on labels and have true- and false-equivalent labels defined for them.*

$$
\begin{array}{|l}
and : LABEL \times LABEL \rightarrow LABEL \\
andTRUE, andFALSE : LABEL \\
\hline
\forall data : DATA \bullet data \in \operatorname{dom} andTRUE \land andTRUE(data) = \varnothing \\
\neg (\exists data : DATA \bullet data \in \operatorname{dom} andFALSE) \\
\forall l_1, l_2 : LABEL;\ data : DATA \bullet (and(l_1, l_2))(data) = l_1(data) \cup l_2(data)
\end{array}
$$

The above definition uses 'and' since it is later used in Def. 6.1.37 to combine labels which should be taken in the same step. It is possible to unite them because no racing between such labels is allowed by Req. 3b.

An empty label can be defined as the *andTRUE* function.

The definitions above differ from [MLPS97] where only event variables are considered while above ordinary variables are permitted as well. Due to the extension, the concept of *changes* was introduced which is unnecessary if no persistent variables are allowed. Another difference between this work and the referenced paper is that no events occurring when states are entered or left, variables accessed or modified, are considered. These were omitted but can be introduced relatively easily into actions of transitions.

An approach to represent data as a big Cartesian product described above can be given an alternative representation as a partial function from names of variables to their values. For example, for the above example we could have $values : NAME \nrightarrow VALUE$ and

$$values = \{(variable, 3), (event, \perp_{sample})\}$$

The functionality of such a function is essentially provided by functions len_{SPACE}, π and *index*. Since the whole data space of a system is actually a Cartesian product of sets of values of variables used in it, the tuple representation $\mathbb{N} \times (\mathbb{N} \cup \{\perp_{sample}\})$ rather than its alternative has been used.

The basic definition of a transition

Definition 6.1.16. *TRANSITION type represents a compound transition which further in proofs will be referred to as transition; it can be defined (from [MLPS97]) as:*

```
— TRANSITION ——————————————————
source, target : 𝔽₁ STATE
label : LABEL
———————————————————————————————
```

TRANSITION does not have its source and target configuration uniquely defined. Non-unique source configuration is possible for the *stop* transition which exists from both *F_ADVANCE* and *REWIND* in Fig. 1.6. Non-unique target configuration is possible for transitions in concurrent states of the statechart in Fig. 1.10 where *play* can be taken together with *rew_or_ff* or separately.

Now we provide a few useful definitions which are used in different parts of this chapter.

A few useful abbreviations are defined below:

$SSet == \mathbb{F}\, STATE$

$TSeq == \text{seq}\, TRANSITION$
$TSet == \mathbb{F}\, TRANSITION$
$LSeq == \text{seq}\, LABEL$
$LSet == \mathbb{F}\, LABEL$
$TSeqSet == \text{seq}\, TSet$
$TSetSet == \mathbb{F}\, TSet$
$LSeqSet == \text{seq}\, LSet$
$LSetSet == \mathbb{F}\, LSet$
$TSetSeq == \mathbb{F}\, TSeq$
$LSetSeq == \mathbb{F}\, LSeq$
$TSetSeqSet == \mathbb{F}\, TSeqSet$
$LSetSeqSet == \mathbb{F}\, LSeqSet$

Now we define a function which takes a singleton set and returns the only element of it.

$[AA]$

$FromSet : \mathbb{F}_1\, AA \twoheadrightarrow AA$

$\forall A : \mathbb{F}_1\, AA \mid \#A = 1 \bullet \exists a : A \bullet FromSet(A) = a$

In order to apply a function to sequences, we define the following function:

$[AA, BB]$

$apply : \text{seq}\, AA \times (AA \twoheadrightarrow BB) \twoheadrightarrow \text{seq}\, BB$

$\forall seqAA : \text{seq}\, AA;\ func : AA \twoheadrightarrow BB \bullet$
$\quad apply(seqAA, func) = \{i : \text{dom}\, seqAA \bullet i \mapsto func(seqAA\ i)\}$

Composition is an auxiliary function used in definitions of Υ_f, C^{merged} and Φ^{merged} among others. For example,

$$Composition(ADD, \{1, 2, 3\}) = ADD(1, ADD(2, 3))$$

assuming selection of elements from the set in order of appearance; *Composition* is defined as follows:

$$
\begin{array}{|l}
[AA] \\
\hline
Composition : (AA \times AA \rightarrow AA) \times \mathbb{F}_1(AA) \rightarrow AA \\
\hline
\forall srcSet : \mathbb{F}_1(AA);\ with : AA;\ Operation : (AA \times AA \rightarrow AA) \bullet \\
\quad \# srcSet = 1 \Rightarrow Composition(Operation, srcSet) = FromSet(srcSet) \land \\
\quad \# srcSet > 1 \Rightarrow (\exists\, el : srcSet \bullet \\
\qquad Composition(Operation, srcSet) = \\
\qquad\quad Operation(el, Composition(Operation, srcSet \setminus \{el\})))
\end{array}
$$

The function to multiply sets can be defined as follows:

$$
\begin{array}{|l}
[AA] \\
\hline
setMULT : \mathbb{F}(\mathbb{F}\, AA) \times \mathbb{F}(\mathbb{F}\, AA) \rightarrow \mathbb{F}(\mathbb{F}\, AA) \\
\hline
\forall setset_a, setset_b : \mathbb{F}(\mathbb{F}\, AA) \bullet setMULT(setset_a, setset_b) = \\
\quad \{set_a : setset_a;\ set_b : setset_b \bullet set_a \cup set_b\}
\end{array}
$$

Definition 6.1.17. *Multiplication of sets of sequences of sets is defined as follows:*

$$
\begin{array}{|l}
[AA] \\
\hline
ASetSeqSet == \mathbb{F}(\mathrm{seq}(\mathbb{F}\, AA))
\end{array}
$$

$$
\begin{array}{|l}
[AA] \\
\hline
multOR : ASetSeqSet[AA] \times ASetSeqSet[AA] \twoheadrightarrow ASetSeqSet[AA] \\
\hline
\forall A, B : ASetSeqSet \bullet multOR(A, B) = \{a : A;\ b : B \bullet a \frown b\}
\end{array}
$$

$$
\begin{array}{|l}
[AA] \\
\hline
multOR1 : ASetSeqSet[AA] \times ASetSeqSet[AA] \twoheadrightarrow ASetSeqSet[AA] \\
\hline
\forall A, B : ASetSeqSet \bullet multOR1(A, B) = \\
\quad \{a : A;\ b : B \bullet front\ a \frown \langle last\ a \cup head\ b \rangle \frown tail\ b\}
\end{array}
$$

$$
\begin{array}{|l}
[AA] \\
\hline
RaiseToPower : ASetSeqSet[AA] \times \mathbb{N} \twoheadrightarrow ASetSeqSet[AA] \\
\hline
\forall A : ASetSeqSet \bullet \\
\quad RaiseToPower(A, 0) = \{\langle\rangle\} \land \\
\quad (\forall n : \mathbb{N}_1 \bullet RaiseToPower(A, n) = \\
\qquad multOR(A, RaiseToPower(A, n - 1)))
\end{array}
$$

$$
\begin{array}{|l}
[AA] \\
\hline
multAND : ASetSeqSet[AA] \times ASetSeqSet[AA] \pfun ASetSeqSet[AA] \\
\hline
\forall A, B : ASetSeqSet \bullet \\
\qquad A \neq \emptyset \land B \neq \emptyset \Rightarrow multAND(A, B) = \{a : A;\ b : B \bullet Unite(a, b)\} \\
\hline
\end{array}
$$

Above we used the following function

$$
\begin{array}{|l}
[AA] \\
\hline
Unite : \mathrm{seq}(\mathbb{F}\, AA) \times \mathrm{seq}(\mathbb{F}\, AA) \pfun \mathrm{seq}(\mathbb{F}\, AA) \\
\hline
\forall a, b : \mathrm{seq}(\mathbb{F}\, AA) \bullet Unite(a, b) = \\
\qquad \{n : 1 \mathbin{..} min(\{\#a, \#b\}) \bullet n \mapsto (a(n) \cup b(n))\} \cup \\
\qquad (1 \mathbin{..} min(\{\#a, \#b\})) \ndres a \cup (1 \mathbin{..} min(\{\#a, \#b\})) \ndres b \\
\hline
\end{array}
$$

Two useful conversion functions are defined below:

$$
\begin{array}{|l}
[AA] \\
\hline
SeqtoSeqSet : \mathrm{seq}\, AA \pfun \mathrm{seq}(\mathbb{F}\, AA) \\
SetSeqtoSetSeqSet : \mathbb{F}(\mathrm{seq}\, AA) \pfun \mathbb{F}(\mathrm{seq}(\mathbb{F}\, AA)) \\
\hline
\forall seq : \mathrm{seq}\, AA \bullet SeqtoSeqSet(seq) = \\
\qquad \{i : 1 \mathbin{..} \#seq \bullet i \mapsto \{seq\ i\}\} \\
\forall setseq : \mathbb{F}(\mathrm{seq}\, AA) \bullet SetSeqtoSetSeqSet(setseq) = \\
\qquad \{seq : setseq \bullet SeqtoSeqSet(seq)\} \\
\hline
\end{array}
$$

Transitions involved in a path are given by the following function:

Definition 6.1.18.

$$
\begin{array}{|l}
[AA] \\
\hline
TRANSITIONS : ASetSeqSet[AA] \pfun \mathbb{F}\, AA \\
\hline
\forall asetseqset : ASetSeqSet \bullet \\
\qquad TRANSITIONS(asetseqset) = \\
\qquad\qquad \bigcup\{aseqset : asetseqset \bullet \bigcup(\mathrm{ran}\ aseqset)\} \\
\hline
\end{array}
$$

Definition 6.1.19. *The enable predicate can be used to check if a transition can be taken from a given configuration and its precondition is satisfied. The trigger predicate only verifies satisfiability of the precondition of a transition.*

$$
\begin{array}{|l}
\mathit{trigger}\ \_ : \mathbb{F}_1(\mathit{LABEL} \times \mathit{DATA}) \\
\mathit{triggerSET}\ \_ : \mathbb{F}_1(\mathit{LSet} \times \mathit{DATA}) \\
\hline
\forall\, l : \mathit{LABEL};\ \mathit{data} : \mathit{DATA} \bullet \mathit{trigger}(l, \mathit{data}) \Leftrightarrow \mathit{data} \in \mathsf{dom}\, l \\
\forall\, \mathit{lset} : \mathit{LSet};\ \mathit{data} : \mathit{DATA} \bullet \\
\quad \mathit{triggerSET}(\mathit{lset}, \mathit{data}) \Leftrightarrow (\forall\, l : \mathit{lset} \bullet \mathit{trigger}(l, \mathit{data}))
\end{array}
$$

Here we define basic requirements for transitions of a statechart,

Definition 6.1.20. *Valid transitions (transitionVALID). The set of valid transitions is τ; the set of valid transitions actually used in a statechart is Υ, which includes default transitions. transitionDEFAULT is a helper function to verify that a transition is a default one. We assume that Υ is finite.*

$$
\begin{array}{|l}
\mathit{transitionVALID}\ \_ : \mathbb{F}_1(\mathit{TRANSITION}) \\
\Upsilon, \tau : \mathbb{F}\ \mathit{TRANSITION} \\
\mathit{transitionDEFAULT}\ \_ : \mathbb{F}_1(\mathit{TRANSITION}) \\
\hline
\forall\, \mathit{tr} : \mathit{TRANSITION} \bullet \mathit{transitionVALID}(\mathit{tr}) \Leftrightarrow \\
\quad \mathit{tr.source} \cup \mathit{tr.target} \subseteq \Sigma \wedge \\
\quad \mathit{root} \notin (\mathit{tr.source} \cup \mathit{tr.target}) \wedge \\
\quad \#\mathit{tr.source} > 0 \wedge \#\mathit{tr.target} > 0 \wedge \\
\quad (\forall\, s_1, s_2 : \mathit{tr.source} \bullet s_1 \neq s_2 \Rightarrow \mathit{orth}(s_1, s_2)) \wedge \\
\quad (\forall\, s_1, s_2 : \mathit{tr.target} \bullet s_1 \neq s_2 \Rightarrow \mathit{orth}(s_1, s_2)) \wedge \\
\quad (\neg\, (\mathit{transitionDEFAULT}(\mathit{tr}) \vee \mathit{tr.source} \subseteq \rho(\mathit{root})) \Rightarrow \\
\qquad \mathit{connectorDEFAULT} \notin \phi(\!|\, \mathit{tr.source} \,|\!) \cup \phi(\!|\, \mathit{tr.target} \,|\!)) \wedge \\
\quad (\mathit{transitionDEFAULT}(\mathit{tr}) \Rightarrow \#\mathit{tr.source} = 1 \wedge \\
\qquad \mathit{tr.target} \subseteq \rho^{+}(\mathit{parent}(\mathit{FromSet}\ \mathit{tr.source}))) \wedge \\
\quad (\forall\, \mathit{st} : \Sigma \mid \phi(\mathit{st}) = \mathit{stateAND} \bullet ((\mathit{tr.source} \cup \mathit{tr.target}) \cap \rho(\mathit{st})) = \varnothing) \\
\tau = \{\mathit{tr} : \mathit{TRANSITION} \mid \mathit{transitionVALID}(\mathit{tr})\} \wedge \Upsilon \subseteq \tau \\
\exists\, s : \rho(\mathit{root});\ \mathit{tr} : \Upsilon \bullet \phi(s) = \mathit{connectorDEFAULT} \wedge \mathit{tr.source} = \{s\} \\
\forall\, \mathit{tr} : \tau \bullet \\
\quad \mathit{transitionDEFAULT}(\mathit{tr}) \Leftrightarrow \phi(\!|\, \mathit{tr.source} \,|\!) = \{\mathit{connectorDEFAULT}\} \wedge \\
\qquad \neg\, (\mathit{tr.source} \subseteq \rho(\mathit{root}))
\end{array}
$$

Valid transitions should not begin or terminate at the root state and should have both source and target states orthogonal. According to the Req. 4e, we also restrict states entered by transitions to exclude default connectors. Additionally, the definition formalises Req. 4f and Req. 4g (explained in Sect. 5.2.11 on p. 101 and Sect. 5.2.12 on p. 102).

Definition 6.1.21. *Transition is enabled if it is triggered and a statechart is in its source states*

$$\begin{array}{|l}
enable\,\_ : \mathbb{F}_1(TRANSITION \times DATA \times SSet) \\
\hline
\forall\, t : \Upsilon;\ data : DATA;\ conf : \mathbb{F}_1\,\Sigma \mid configuration(root, conf) \bullet \\
\quad enable(t, data, conf) \Leftrightarrow \\
\qquad t.source \subseteq conf \land trigger(t.label, data)
\end{array}$$

The definition of the *transitionDEFAULT* does not treat the transition entering the whole statechart as default. This reflects the fact that we consider statecharts either not embedded in any other statecharts or entered by some transition to which the one from the default substate of the *root* state is a continuation. In both cases such a transition from the default substate of the *root* state does not have a known initial one and thus should be treated as an initial one. This allows us to construct a full compound transition entering an initial configuration of a statechart. Such treatment is consistent with Req. 4a, from which it follows that initial CT are always not default transitions; continuation CTs are always default.

Proposition 6.1.22. *Every non-default transition can go from some configuration to another one.*

$$\begin{array}{l}
\forall\, tr : \Upsilon \bullet \exists\, src, tgt : \mathbb{F}_1\,\Sigma \bullet \\
\quad configuration(root, src) \land configuration(root, tgt) \land \\
\quad tr.source \subseteq src \land tr.target \subseteq tgt
\end{array}$$

Proof. Follows from Prop. 6.1.7 since source and target states of a transition are orthogonal.

A configuration is invalid if one of the properties of Def. 6.1.3 is not satisfied. Since we are talking about a subset of a configuration, missing states do not count, only those which cannot be in a valid configuration together do. Such states are the following:

1. $\phi(s) \neq stateAND \lor \phi(s) \neq stateOR$ — cannot be true for non-default valid transitions.

2. For *states* being source or target states of a considered transition, $\phi(s) = stateOR \land \#(\rho(s) \cap states) > 1$ — contradicts *orth*.

3. $states \setminus \Sigma \neq \varnothing$ — contradicts *transitionVALID*.

□

Proposition 6.1.23. *For every valid configuration there is a valid transition going from it and terminating at it. Note: such a transition does not have to be in Υ, but it will be in τ.*

$$\forall\, conf : \mathbb{F}_1\,\Sigma \mid configuration(root, conf) \bullet$$
$$(\exists\, tr_s : \tau \bullet tr_s.source \subseteq conf) \land$$
$$(\exists\, tr_d : \tau \bullet tr_d.target \subseteq conf)$$

Proof. A configuration of a non-empty statechart contains at least one basic state (for empty statecharts — only the root one). We can construct a transition going from it which would satisfy *transitionVALID*. Similarly for target configuration. □

The scope of a transition is defined (from [MLPS97]) as follows:

Definition 6.1.24. *Scope of a transition*

$$scope : TRANSITION \twoheadrightarrow STATE$$
$$\forall\, tr : \tau \bullet scope(tr) = lcoa(tr.source \cup tr.target)$$

Transitions may be interlevel and non-interlevel; a non-interlevel transition and a sequence of them can be defined as follows:

Definition 6.1.25. *Interlevel and a sequence of interlevel transitions*

$$transitionNI\,\_ : \mathbb{F}_1\, TRANSITION$$
$$TSetNI\,\_ : \mathbb{F}_1\, TSet$$
$$TSeqNI\,\_ : \mathbb{F}_1\, TSeq$$
$$\forall\, t : \Upsilon \bullet transitionNI(t) \Leftrightarrow (\exists\, s : \Sigma \bullet t.source \subseteq \rho(s) \land t.target \subseteq \rho(s))$$
$$\forall\, tset : TSet \bullet TSetNI(tset) \Leftrightarrow (\forall\, tr : tset \bullet transitionNI(tr))$$
$$\forall\, tseq : TSeq \bullet TSeqNI(tseq) \Leftrightarrow TSetNI(\mathrm{ran}\, tseq)$$

From the definition above, it is easy to see that non-interlevel transitions have a single source and target state,

$$\forall\, tr : \Upsilon \mid transitionNI(tr) \bullet \#tr.source = 1 \land \#tr.target = 1$$

In the proofs for the testing method (Sect. 6.4 on p. 178), we assume that default transitions are not interlevel.

Definition 6.1.26. *Default transitions are not interlevel*

$$\forall\, tr : \Upsilon \bullet transitionDEFAULT(tr) \Rightarrow transitionNI(tr)$$

Note that default transitions can potentially be interlevel. Having all of them this way though contradicts Req. 1a since states in the statechart they are in would be unreachable. In a similar case, states inside the $PLAY$ one in the statechart depicted in Fig. 8.16 on p. 251 are unreachable due to transitions entering them explicitly rather than entering an enclosing state. This is formalised in the following proposition.

Proposition 6.1.27. *If an OR state has a default transitions with no label (empty label aka andTRUE one), it has only one such default transition*

Proof. Follows from determinism of a statechart (Req. 1b) □

The set of labels of transitions including interlevel ones within some state but not inside its children can be defined as

Definition 6.1.28. *Transitions within some state*

$$\begin{array}{|l}
TR : STATE \twoheadrightarrow TSet \\
T : STATE \twoheadrightarrow LSet \\
\hline
\forall\, s : \Sigma \bullet \\
\qquad TR(s) = \{tr : \Upsilon \mid s = scope(tr)\} \wedge \\
\qquad T(s) = \{tr : \Upsilon \mid s = scope(tr) \bullet tr.label\}
\end{array}$$

Above, $s = scope(tr)$ captures the fact that we include interlevel transitions in the set of transitions for their enclosing states; for non-interlevel transitions it reduces to having source and target states of tr within s.

In the following we shall often consider sets of labels which correspond to non-interlevel transitions within some state but not within its children states.

Definition 6.1.29. *Non-interlevel transitions within some state*

$$
\begin{array}{|l}
TR^{ni} : STATE \twoheadrightarrow\!\!\!\!\!+\; TSet \\
T^{ni} : STATE \twoheadrightarrow\!\!\!\!\!+\; LSet \\
\hline
\forall\, s : \Sigma \bullet\ TR^{ni}(s) = \{tr : \Upsilon \mid s = scope(tr) \land transitionNI(tr))\} \\
\forall\, s : \Sigma \bullet\ T^{ni}(s) = \{tr : \Upsilon \mid s = scope(tr) \land transitionNI(tr) \bullet tr.label\}
\end{array}
$$

Non-conflicting transitions and their properties

Definition 6.1.30. *A sequence of transitions can be taken in a step if transitions are pairwise non-conflicting. We define the set of states which is being exited (function exit) and entered (enter) by a transition and then give the definition of non-conflicting transitions from [MLPS97].*

$$\begin{array}{|l}
Uexit, Uenter : TRANSITION \twoheadrightarrow STATE \\
\hline
\forall\, tr : \Upsilon \bullet \\
\quad \exists\, theUexit, theUenter : \rho(scope(tr)) \bullet \\
\qquad tr.source \subseteq \rho^*(theUexit) \land Uexit(tr) = theUexit \land \\
\qquad tr.target \subseteq \rho^*(theUenter) \land Uenter(tr) = theUenter
\end{array}$$

$$\begin{array}{|l}
exit, enter : TRANSITION \times \mathbb{F}_1\, STATE \twoheadrightarrow \mathbb{F}_1\, STATE \\
\hline
\forall\, tr : \Upsilon \bullet \\
\quad \forall\, conf : \mathbb{F}_1\, \Sigma \mid configuration(root, conf) \bullet \\
\qquad exit(tr, conf) = conf \cap \rho^*(Uexit(tr)) \land \\
\qquad (\exists\, entconf : \mathbb{F}_1\, STATE \bullet configuration(Uenter(tr), entconf) \land \\
\qquad\quad enter(tr, conf) = entconf \land tr.target \subseteq entconf)
\end{array}$$

$$\begin{array}{|l}
nonconflict\,\_ : \mathbb{F}_1(TRANSITION \times TRANSITION \times \mathbb{F}_1\, STATE) \\
\hline
\forall\, tr_1, tr_2 : \Upsilon \bullet \\
\quad \forall\, conf : \mathbb{F}_1\, \Sigma \mid configuration(root, conf) \bullet \\
\qquad nonconflict(tr_1, tr_2, conf) \Leftrightarrow \\
\qquad\quad (tr_1 \neq tr_2 \Rightarrow exit(tr_1, conf) \cap exit(tr_2, conf) = \varnothing)
\end{array}$$

$$\begin{array}{|l}
orthogonal\,\_ : \mathbb{F}_1(TSet) \\
\hline
\forall\, tset : TSet \bullet orthogonal(tset) \Leftrightarrow \\
\quad \#tset \leq 1 \lor \\
\quad (\forall\, t_1, t_2 : tset \bullet \\
\qquad (\forall\, s_1 : t_1.source;\ s_2 : t_2.source \bullet orth(s_1, s_2)) \land \\
\qquad (\forall\, s_1 : t_1.target;\ s_2 : t_2.target \bullet orth(s_1, s_2)))
\end{array}$$

Note that in the definition of *Uexit* and *Uenter*, the corresponding states always exist in $\rho(scope(tr))$ as *scope* is defined to be a strict ancestor for source and target states. This is illustrated in Fig. 6.7 where *lca* for highlighted transitions is B but *lcoa* is A.

We show a few properties of the above.

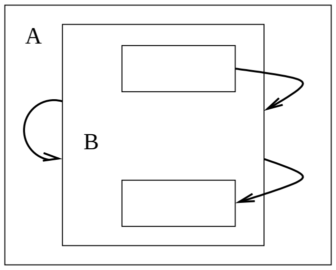


Figure 6.7: An illustration of *lcoa* returning a higher-level state than *lca*

Theorem 6.1.31. *Transitions are not in conflict iff they have orthogonal scopes,*

$$\begin{array}{l}\forall\, t_1, t_2 : \Upsilon \bullet \\ \quad (\forall\, conf : \mathbb{F}_1\, \Sigma \mid configuration(root, conf) \bullet \\ \qquad nonconflict(t_1, t_2, conf) \Leftrightarrow orth(scope(t_1), scope(t_2)))\end{array}$$

Proof.
$\Longrightarrow$
The proof is illustrated by Fig. 6.8. Consider configuration *conf* where t_1

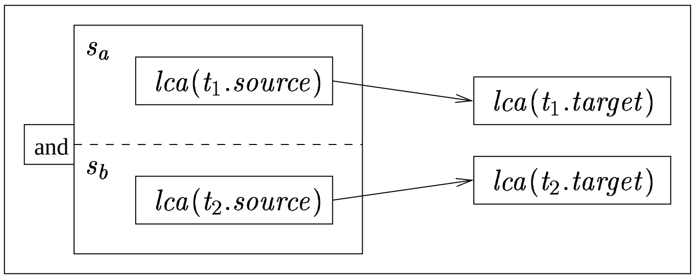


Figure 6.8: An illustration of the proof of Th. 6.1.31

and t_2 are both enabled and $l_1 = lca(t_1.source)$, $l_2 = lca(t_2.source)$ and $and = lca(t_1.source \cup t_2.source)$.

If $\#t_1.source = 1$, then $l_1 = FromSet(t_1.source)$. For the case of $\#t_1.source > 1$ we get $l_1 \notin t_1.source$. Since $\phi(l_1) = stateOR$ contradicts validity of t_1, we get that $\phi(l_1) = stateAND$. Consequently, from definition of *scope*, $l_1 \in \rho^+(scope(t_1))$ and from definition of *Uexit* — $l_1 = Uexit(t_1)$. As we consider t_1 to be enabled and parent states of states in a configuration also belong to it, $l_1 \in conf$ and from the definition of $exit(t_1)$, $l_1 \in exit(t_1)$. A similar result can be shown for t_2.

The above proves that l_1 and l_2 are not related by ρ^* as otherwise transitions t_1 and t_2 would be in conflict.

Due to validity of transitions t_1 and t_2, $\#t_1.source > 0 \wedge \#t_2.source > 0$ and thus $\#(t_1.source \cup t_2.source) > 1$. Using the proof about the source states of them, we have that $and \notin (t_1.source \cup t_2.source)$. For this reason, $\phi(and) = stateAND$ as if it were $stateOR$, there would be no valid configuration in which both t_1 and t_2 are enabled.

$a = lca(l_1, l_2)$ is an ancestor of all states in $t_1.source \cup t_2.source$ by transitivity of ρ^*. Since l_1 and l_2 are not related by ρ^*, no state $s : \rho^*(l_1)$ is an ancestor of any among $t_2.source$ and vice-versa. As a result, $\{l_1, l_2\} \subseteq \rho^+(and)$ which proves $a = and$, i.e. $lca(l_1, l_2) = lca(t_1.source \cup t_2.source)$.

From Prop. 6.1.10 we get that $\exists\, s_a, s_b \in \rho(and) \wedge s_a \neq s_b \wedge l_1 \in \rho^*(s_a) \wedge l_2 \in \rho^*(s_b)$.

Assume that $t_1.target \not\subseteq \rho^*(s_a)$. Denoting $s = lca(t_1.source \cup t_1.target)$, we get from this assumption that $\forall\, ss : \rho^*(s_a) \bullet \neg\, Anc(ss, t_1.target)$ and thus $and \in \rho^*(s)$ (if $and \notin \rho^*(s)$, we get from $t_1.source \subseteq \rho^*(s) \wedge t_1.source \subseteq \rho^*(s_a)$ that $s \in \rho^*(s_a)$ contradicting that $s = lca(t_1.source \cup t_1.target)$). By definition of $scope$, $scope(t_1) = lcoa(t_1.source \cup t_1.target)$ and thus $s \in \rho^*(scope(t_1))$. Since $scope$ is defined to be a strict ancestor and $and \in \rho^*(scope(t_1))$, we get that $and \in \rho^+(scope(t_1))$; from $t_1.source \subseteq \rho^*(and)$, $and \in exit(t_1)$ follows. Considering that $t_1.source \subseteq conf$, $and \in conf$ as a parent of a state in $conf$. As a result, $and \in conf \cap exit(t_1)$. A similar result can be shown for t_2. It follows that if $t_1.target \not\subseteq \rho^*(s_a) \wedge t_2.target \not\subseteq \rho^*(s_b)$, transitions t_1 and t_2 are in conflict.

Assume that $t_2.target \subseteq \rho^*(s_b)$, then $scope(t_2) \in \rho^*(s_b)$. If $t_1.target \not\subseteq \rho^*(s_a)$, we get from $s_b \in \rho(and)$, $s_b \in conf$ and $and \in conf \cap exit(t_1)$ that $t_2.target \subseteq (conf \cap exit(t_1) \cap exit(t_2))$. A result similar to this one can be shown for $t_1.target \subseteq \rho^*(s_b) \wedge t_2.target \not\subseteq \rho^*(s_b)$.

We have just proven that in order to avoid conflict between t_1 and t_2, we have to allow $t_1.target \subseteq \rho^*(s_a) \wedge t_2.target \subseteq \rho^*(s_b)$ only.

Since $s_a, s_b \in \rho(and)$ are immediate substates of an AND-state and transitions cannot start or terminate in such states (Req. 4g), we get that $t_1.target \subseteq \rho^+(s_a)$, $t_2.target \subseteq \rho^+(s_b)$, $t_1.source \subseteq \rho^+(s_a)$ and $t_2.source \subseteq \rho^+(s_b)$. Thus, $scope(t_1) \in \rho^*(s_a)$ and $scope(t_2) \in \rho^*(s_b)$ which gives us $orth(scope(t_1), scope(t_2))$

$\Longleftarrow$

If scopes are orthogonal, $orth(scope(t_1), scope(t_2))$, from the definition of $orth$ we get that $and = lca(scope(t_1), scope(t_2))$ is an AND-state and through Prop. 6.1.10 that $\exists\, s_a, s_b \in \rho(and) \wedge s_a \neq s_b \wedge scope(t_1) \in \rho^*(s_a) \wedge scope(t_2) \in \rho^*(s_b)$. This implies that $\rho^*(Uexit(t_1)) \cap \rho^*(Uexit(t_2)) = \varnothing$. This implies that t_1 and t_2 are not in conflict. $\square$

The result proven is also mentioned as property in the sketch of the proof of well-definedness of a step (Def. 6.2.8) in [MLPS97]. [PS91], on the other hand, puts the orthogonality of the areas (a similar concept to scope) in the definition.

Proposition 6.1.32. *Enabled nonconflicting transitions are orthogonal. Here we assume that transitions have their triggers satisfied and hence only consider state-related 'enableness'.*

$$\begin{aligned}
&\forall\, t_1, t_2 : \Upsilon;\ conf : \mathbb{F}_1\, \Sigma \mid configuration(root, conf) \bullet \\
&\quad t_1.source \subseteq conf \land t_2.source \subseteq conf \land nonconflict(t_1, t_2, conf) \Rightarrow \\
&\qquad orthogonal(\{t_1, t_2\})
\end{aligned}$$

Proof. If two transitions are nonconflicting, they have orthogonal scopes from Th. 6.1.31, $orth(scope(t_1), scope(t_2))$. From the definition of $orth$ we get that $and = lca(scope(t_1), scope(t_2))$ is an AND-state and through Prop. 6.1.10) that $\exists\, s_a, s_b \in \rho(and) \land s_a \neq s_b \land scope(t_1) \in \rho^*(s_a) \land scope(t_2) \in \rho^*(s_b)$ and $t_1.source \subseteq \rho^*(s_a) \land t_2.source \subseteq \rho^*(s_b) \land t_1.target \subseteq \rho^*(s_a) \land t_2.target \subseteq \rho^*(s_b)$ from which orthogonality of source and target states follows by definition of orthogonality. □

Proposition 6.1.33. *nonconflict does not depend on configuration, i.e.,*

$$\begin{aligned}
&\forall\, tr_1, tr_2 : \Upsilon \bullet \\
&(\exists\, conf : \mathbb{F}_1\, \Sigma \bullet configuration(root, conf) \land (nonconflict(tr_1, tr_2, conf) \Rightarrow \\
&\quad (\forall\, c : \mathbb{F}_1\, \Sigma \mid configuration(root, c) \bullet nonconflict(tr_1, tr_2, c)))) \\
&\land \\
&(\exists\, conf : \mathbb{F}_1\, \Sigma \bullet configuration(root, conf) \land \neg\, nonconflict(tr_1, tr_2, conf) \Rightarrow \\
&\quad (\forall\, c : \mathbb{F}_1\, \Sigma \mid configuration(root, c) \land tr_1.source \subseteq c \land tr_2.source \subseteq c \bullet \\
&\qquad \neg\, nonconflict(tr_1, tr_2, c)))
\end{aligned}$$

Proof. Two transitions are nonconflicting iff they have orthogonal scopes from Th. 6.1.31. The result follows from scopes and orthogonality of them not depending on a particular configuration. □

We further use a definition

Definition 6.1.34. *A set of transitions with orthogonal scopes,*

$$\begin{array}{|l}
orthscope\ \_ : \mathbb{F}_1(TSet) \\
\hline
\forall\, tset : \mathbb{F}_1\, \Upsilon \bullet orthscope(tset) \Leftrightarrow \\
\quad (\forall\, t_1, t_2 : tset \mid t_1 \neq t_2 \bullet orth(scope(t_1), scope(t_2)))
\end{array}$$

Definition 6.1.35. *We can assert the requirement Req. 1e as follows:*

$$\forall s_1, s_2 : \Sigma \mid s_1 \neq s_2 \bullet (\forall tr : TR(s_1) \cap TR(s_2) \bullet \\ \quad transitionDEFAULT(tr) \land tr.label = andTRUE)$$

Proposition 6.1.36. *There exists a function giving a scope state to all non-default transitions with a given non-andTRUE label,*

$$\begin{array}{|l} getSCOPE : LABEL \twoheadrightarrow STATE \\ \hline \forall lbl : LABEL \bullet (\exists tr : \Upsilon \bullet \\ \quad tr.label = lbl \Rightarrow getSCOPE(lbl) = scope(tr)) \end{array}$$

Proof. By definition of T (Def. 6.1.28), and Def. 6.1.35 we get that

$$\forall state : \Sigma \bullet (\forall lbl : T(state) \mid lbl \neq andTRUE \bullet \\ \quad (\forall tr : \Upsilon \mid tr.label = lbl \bullet scope(tr) = state))$$

The above statement shows that for a given label there is only one scope state, which proves that *getSCOPE* is indeed a function. □

Full compound transitions taken in the same step should enter the same group of states in concurrent components

From Prop. 6.1.32 it appears that transitions which could be taken together as a single full compound transition, cannot be taken in the same step if they are parts of separate transitions. The generalised semantics described below corresponds to usage of orthogonality of transitions to decide whether some transitions can be taken in the same step or not. It is required for parts of full compound transitions in [NH95]. Whether two full compound transitions may be taken in the same step is more restrictive as given in Th. 6.1.31.

Consider a statechart in Fig.6.9. In some sense, we can say that transitions a and b can be taken both separately and together (this is prohibited in Statemate semantics since they are conflicting). The same can be said about those in Fig. 6.10. In Fig.6.11 full compound transitions starting with a and b are both entering the same state D and thus could be theoretically taken together. This is not the case for a and c which are entering different OR-states. The difference between these two cases is the default state. A modification to make it point to C would make a and c possible and a, b — not. Since such a modification does not modify the exiting states of the transitions considered and its effects are rather subtle, the autor thinks

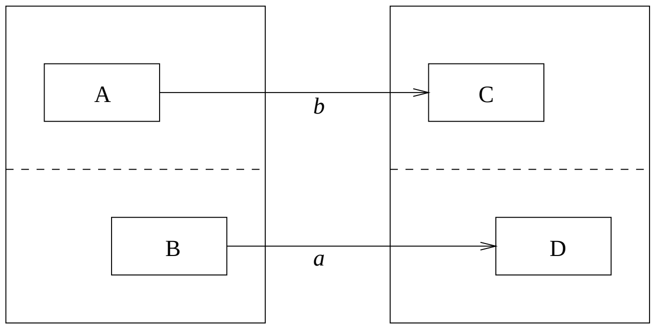


Figure 6.9: Transitions from concurrent states entering concurrent states

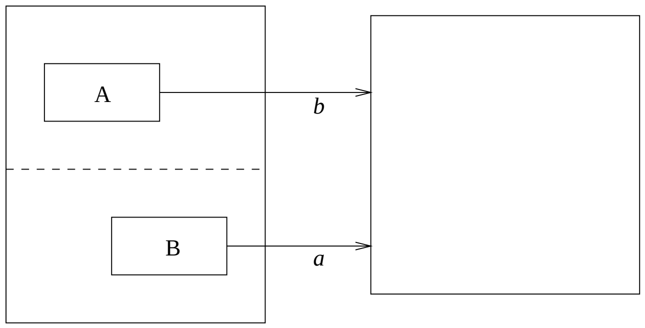


Figure 6.10: Transitions from concurrent states entering a single state

it is better to prohibit such a case and require that transtions which could be taken in the same step should enter either the same state or orthogonal states.

Note that we could state the same requirement for transitions leaving states, i.e. require them to exit orthogonal states or the same one.

Combining transitions

Given a set of orthogonal transitions or full compound transitions, we can unite them into a single transition as shown below, using auxiliary functions *landALL* and *andALL*. The former combines provided labels, the latter — labels of transitions given to it.

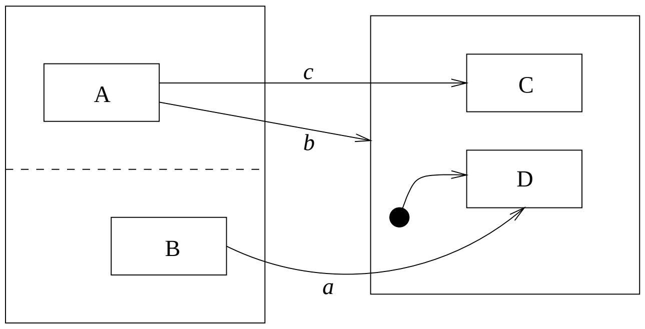


Figure 6.11: Transitions from concurrent states entering an OR-state and its substates

Definition 6.1.37.

$$\begin{array}{l}
landALL : LABEL \times LSet \rightarrow LABEL \\
andALL : TSet \rightarrow LABEL \\
\hline
\forall l : LABEL; lset : LSet \bullet \\
\quad lset = \varnothing \Rightarrow landALL(l, lset) = l \land \\
\quad lset \neq \varnothing \Rightarrow (\exists lbl : lset \bullet \\
\quad\quad landALL(l, lset) = landALL(and(l, lbl), lset \setminus \{lbl\})) \\
\forall l : LABEL; tset : TSet \bullet \\
\quad andALL(tset) = landALL(andTRUE, \{t : tset \bullet t.label\})
\end{array}$$

$$\begin{array}{l}
toTRANSITION : TSet \nrightarrow TRANSITION \\
\hline
\forall tset : \mathbb{F}_1 \Upsilon \mid orthogonal(tset) \land \neg (\exists tr : tset \bullet transitionDEFAULT(tr)) \bullet \\
\quad toTRANSITION(tset) = \\
\quad\quad \langle\!| source == \bigcup\{t : tset \bullet t.source\}, \\
\quad\quad target == \bigcup\{t : tset \bullet t.target\}, \\
\quad\quad label == andALL(tset) |\!\rangle
\end{array}$$

This definition does not include default connectors in the set of source states. The transition constructed can be proven to be a valid transition if the set of transitions supplied to *toTRANSITION* contains orthogonal and valid transitions. This property (Prop. 6.1.38) is used later in Sect. 6.2 on p. 157 for flattened statecharts.

Proposition 6.1.38. *For a set of transitions which are valid, orthogonal and non-default, toTRANSITION generates a valid transition.*

$$\forall\, tset : \mathbb{F}_1\, \Upsilon \mid orthogonal(tset) \land \neg\, (\exists\, tr : tset \bullet transitionDEFAULT(tr)) \bullet$$
$$orthogonal(tset) \Rightarrow transitionVALID(toTRANSITION(tset))$$

Proof. Since transitions are orthogonal, $\forall\, s_1 : enter(tr_1);\ s_2 : enter(tr_2) \bullet orth(s_1, s_2)$ and $\forall\, s_1 : exit(tr_1);\ s_2 : exit(tr_2) \bullet orth(s_1, s_2)$. This proves that *toTRANSITION* generates a valid transition. From validity of transitions of *tset*, we get that $root \notin source \cap target$, $connectorDEFAULT \notin \phi(\!|source|\!)$, $connectorDEFAULT \notin (\!|target|\!)$, where we denote $source = toTRANSITION(tset).source$ and $target = toTRANSITION(tset).target$, and finally $\forall\, st : \Sigma \mid \phi(st) = stateAND \bullet ((tr.source \cup tr.target) \cap \rho(st)) = \varnothing$ □

Transitions which can be taken in the same step

From Prop. 6.1.32 and the above description we get a hierarchy of conditions transitions must satisfy, such that we could combine them with *toTRANSITION* in valid transitions or take at the same time in a step as shown in Tab. 6.1.

requirement	description
transitionVALID	The basic requirement for valid transitions.
orthogonal	Allows to combine transitions into one with *toTRANSITION*. This allows combining compound transitions into a full compound transition.
nonconflicting, i.e. *orthscope*	Allows to take transitions at the same time in the same step.

Table 6.1: The hierarchy of requirements for transitions which could be taken in the same step

6.1.3 Full compound transitions

The definition of a full compound transition is adapted from [MLPS97] with a modification such that a transition from a default connector can be full compound only if it starts in the default connector at the top level. This change is included in the definition of *transitionDEFAULT* (Def. 6.1.20).

Definition 6.1.39. *The definition of a full compound transition from [MLPS97] is essentially reproduced as follows:*

$$
\begin{array}{|l}
\mathit{transitionFCT\_MLPS97}\,\_ : \mathbb{F}_1\ \mathit{TRANSITION} \\
\hline
\forall\, tr : \Upsilon \bullet \mathit{transitionFCT\_MLPS97}(tr) \Leftrightarrow \\
\quad (\exists\, \mathit{EnterState} : \rho(\mathit{scope}(tr)) \bullet \\
\quad\quad (\exists\, \mathit{conf} : \mathbb{F}_1\ \mathit{STATE} \bullet \mathit{configuration}(\mathit{EnterState}, \mathit{conf}) \wedge \\
\quad\quad\quad tr.\mathit{target} \subseteq \mathit{conf} \wedge \phi(\!| tr.\mathit{target} |\!) = \{\mathit{stateBASIC}\} \wedge \\
\quad\quad\quad (\forall\, s_1, s_2 : tr.\mathit{source} \bullet s_1 \neq s_2 \Rightarrow \mathit{orth}(s_1, s_2))))
\end{array}
$$

One might expect that a full compound transition enters a complete configuration such that no continuation transitions are needed. This is indeed the case because continuation transitions are only needed if entered states are OR ones while the above definition requires them to be basic states. Since orthogonality of source and target states is stated in Def. 6.1.20 and existence of appropriate configuration is shown in Prop. 6.1.22, the above definition can be reduced as given below:

Definition 6.1.40. *A full compound transition is the one satisfying transitionFCT.*

$$
\begin{array}{|l}
\mathit{transitionFCT}\,\_ : \mathbb{F}_1\ \mathit{TRANSITION} \\
\hline
\forall\, tr : \Upsilon \bullet \mathit{transitionFCT}(tr) \Leftrightarrow \\
\quad \mathit{transitionVALID}(tr) \wedge \\
\quad \neg\ \mathit{transitionDEFAULT}(tr) \wedge \phi(\!| tr.\mathit{target} |\!) = \{\mathit{stateBASIC}\}
\end{array}
$$

From hierarchy in Tab. 6.1, we can show that full compound transitions which are orthogonal, can be united into a single full compound transition.

Proposition 6.1.41. *For a set of transitions which are orthogonal and full compound toTRANSITION generates a full compound transition.*

$$
\begin{array}{l}
\forall\, \mathit{tset} : \mathbb{F}_1\ \Upsilon \bullet \\
\quad (\forall\, tr : \mathit{tset} \bullet \mathit{transitionFCT}(tr)) \wedge \mathit{orthogonal}(\mathit{tset}) \Rightarrow \\
\quad\quad \mathit{transitionFCT}(\mathit{toTRANSITION}(\mathit{tset}))
\end{array}
$$

Proof. From Prop. 6.1.38 we get the validity of the result of *toTRANSITION*. Since states entered by all $tr : tset$ are basic, states entered by their union will be so too. As no transitions in *tset* are default, no default connectors will appear in the result of *toTRANSITION*. □

In the following, we define the operational semantics of full compound transitions which will be used later and show its equivalence to the denotational one given above. In some sense, we can say that Prop. 6.1.41 gives soundness of *FULL_COMPOUND* (Def. 6.1.54) and completeness follows from Prop. 6.1.60.

We begin with the definition of the default completion function. This function essentially corresponds to the *Complete* one in [CAB+98].

Definition 6.1.42 (defaultEntranceComplete).

$$
\begin{array}{l}
\mathit{defaultEntranceComplete} : \mathit{STATE} \times \mathit{SSet} \twoheadrightarrow\!\!\!\!\!\!\!+ \;\; \mathit{SSet} \\
\hline
\forall\, \mathit{rootstate} : \Sigma;\ \mathit{states} : \mathbb{F}\,\Sigma \mid \\
(\exists\, \mathit{conf} : \mathbb{F}_1\, \mathit{STATE} \bullet \mathit{configuration}(\mathit{rootstate}, \mathit{conf}) \wedge \mathit{states} \subseteq \mathit{conf}) \bullet \\
\quad \mathit{rootstate} \in \mathit{defaultEntranceComplete}(\mathit{rootstate}, \mathit{states}) \wedge \\
\quad (\forall\, s : \mathit{defaultEntranceComplete}(\mathit{rootstate}, \mathit{states}) \bullet \\
\qquad (\phi(s) = \mathit{stateAND} \Rightarrow \\
\qquad\quad (\rho(s) \subseteq \mathit{defaultEntranceComplete}(\mathit{rootstate}, \mathit{states}))) \wedge \\
\qquad (\phi(s) = \mathit{stateOR} \Rightarrow \\
\qquad\quad (\rho^{+}(s) \cap \mathit{states} = \varnothing \Rightarrow \\
\qquad\qquad (\exists_1\, \mathit{def} : \rho(s) \bullet \phi(\mathit{def}) = \mathit{connectorDEFAULT} \wedge \\
\qquad\qquad\quad \mathit{def} \in \mathit{defaultEntranceComplete}(\mathit{rootstate}, \mathit{states}))) \wedge \\
\qquad\quad (\rho^{+}(s) \cap \mathit{states} \neq \varnothing \Rightarrow (\exists\, \mathit{sl} : \rho(s) \bullet \rho^{*}(\mathit{sl}) \cap \mathit{states} \neq \varnothing \wedge \\
\qquad\qquad\quad \mathit{sl} \in \mathit{defaultEntranceComplete}(\mathit{rootstate}, \mathit{states})))))
\end{array}
$$

We allow *defaultEntranceComplete* to be applied only to sets from which a valid configuration can be constructed.

Proposition 6.1.43. *configuration can be defined recursively.*

$$
\begin{array}{l}
\forall\, \mathit{conf} : \mathbb{F}_1\, \Sigma \mid \mathit{configuration}(\mathit{root}, \mathit{conf}) \bullet \\
\quad \forall\, s : \mathit{conf} \bullet \mathit{configuration}(s, \rho^{*}(s) \cap \mathit{conf})
\end{array}
$$

Proof. Follows from the definition of *configuration*. □

Proposition 6.1.44. *In the line*

$$\exists\, \mathit{sl} : \rho(s) \bullet \rho^{*}(\mathit{sl}) \cap \mathit{states} \neq \varnothing$$

of the Def. 6.1.42, sl is unique, $\exists_1\, \mathit{sl} : \rho(s) \bullet \rho^{*}(\mathit{sl}) \cap \mathit{states} \neq \varnothing$.

Proof. Follows from the definition of configuration (Def. 6.1.3). □

Proposition 6.1.45. *All states supplied to defaultEntranceComplete are included in the result (we assume that* $(rootstate, states) \in \mathsf{dom}\ defaultEntranceComplete$*),*

$$\forall\, rootstate : \Sigma;\ states : \mathbb{F}\,\Sigma \mid states \subseteq \rho^*(rootstate) \bullet$$
$$states \subseteq defaultEntranceComplete(rootstate, states)$$

Proof. Due to Prop. 6.1.44, *defaultEntranceComplete* follows a route to all states in *states*. First of all, $\rho^*(rootstate) \cap states \neq \varnothing$. If for some OR-states s we have that $\rho^+(s) \cap states \neq \varnothing$, then $\exists_1\, s_1 : \rho(s) \bullet \rho^*(s_1) \cap states \neq \varnothing)$. Such $s_i \in \rho(s_{i-1})$ and there is a limit (belonging to *states*), due to finiteness of a state tree it will be reached. All such s_i will be included in the set of states returned by *defaultEntranceComplete*. □

Proposition 6.1.46. *defaultEntranceComplete has the following recursive property:*

$$\forall\, states : \mathbb{F}\,\Sigma \mid$$
$$(\exists\, conf : \mathbb{F}_1\,\Sigma \bullet configuration(root, conf) \land states \subseteq conf) \bullet$$
$$(\forall\, \sigma : states \bullet defaultEntranceComplete(root, states) \cap \rho^*(\sigma) =$$
$$defaultEntranceComplete(\sigma, states \cap \rho^*(\sigma)))$$

Note that since $\rho^*(\sigma) \cap states$ *are the only states which can be children of* σ *(from definition of* ρ^**), we have to restrict states to* $\rho^*(\sigma) \cap states$ *when the root state passed to defaultEntranceComplete is* σ.

Proof. $defaultEntranceComplete(\sigma, states \cap \rho^*(\sigma))$ will start from σ which will also be included in the result of $defaultEntranceComplete(rootstate, states)$ by Prop. 6.1.45. As *defaultEntranceComplete* from a root state does the same job as the one with σ, results (when restricted to those within $\rho^*(\sigma)$) will be the same. □

Theorem 6.1.47. *defaultEntranceComplete works fine with respect to configuration, i.e.*

1. *result does not change regardless how many times we apply defaultEntranceComplete, as long as we do once.*

$$\begin{aligned}&\forall\, rootstate : \Sigma;\ states : \mathbb{F}\,\Sigma \mid\\ &\quad (\exists\, conf : \mathbb{F}_1\, STATE \bullet configuration(rootstate, conf) \land states \subseteq conf) \bullet\\ &\qquad (defaultEntranceComplete(rootstate,\\ &\qquad\quad defaultEntranceComplete(rootstate, states)) =\\ &\qquad\quad defaultEntranceComplete(rootstate, states))\end{aligned}$$

2. *for a set of states which is a valid subset of some configuration, defaultEntranceComplete with default connectors excluded is a subset of the same configuration and will include those states.*

$$\begin{aligned}&\forall\, rootstate : \Sigma;\ states : \mathbb{F}\,\Sigma \mid states \subseteq \rho^*(rootstate) \bullet\\ &\quad (\exists\, conf : \mathbb{F}_1\, STATE \bullet configuration(rootstate, conf) \land\\ &\qquad states \setminus defaultfrom(states) \subseteq conf \Rightarrow\\ &\qquad (defaultEntranceComplete(rootstate, states) \setminus\\ &\qquad\quad defaultfrom(states) \subseteq conf) \land\\ &\qquad states \subseteq defaultEntranceComplete(rootstate, states))\end{aligned}$$

3. *if we pass it a complete configuration, the result will be the same configuration.*

$$\begin{aligned}&\forall\, rootstate : STATE;\ conf : \mathbb{F}_1\, STATE \mid configuration(rootstate, conf) \bullet\\ &\quad defaultEntranceComplete(rootstate, conf) = conf\end{aligned}$$

4. *In a state tree returned by defaultEntranceComplete, default connectors are orthogonal.*

$$\begin{aligned}&\forall\, rootstate : STATE;\ states : \mathbb{F}\,\Sigma \mid\\ &\quad (\exists\, conf : \mathbb{F}_1\, STATE \bullet configuration(rootstate, conf) \land states \subseteq conf) \bullet\\ &\qquad (\forall\, s_1, s_2 : defaultEntranceComplete(rootstate, states) \bullet\\ &\qquad\quad \phi(s_1) = connectorDEFAULT \land\\ &\qquad\quad \phi(s_2) = connectorDEFAULT \Rightarrow orth(s_1, s_2))\end{aligned}$$

5. *defaultEntranceComplete can be applied after any transition is taken.*

$$\begin{aligned}&\forall\, tr : \Upsilon \bullet (\forall\, conf : \mathbb{F}_1\, STATE \mid tr.target \subseteq conf \land configuration(root, conf) \bullet\\ &\quad (root, conf) \in \mathrm{dom}\ defaultEntranceComplete)\end{aligned}$$

Proof. We consider the above five statements in turn.

1. Follows from definition of *defaultEntranceComplete* (it tries to complete on all states it includes).

2. That all *states* are included is shown in Prop. 6.1.45. The definition of *defaultEntranceComplete* is similar to the definition of configuration for AND-states. For OR-states, there are two cases,

   (a) for some state $\sigma \in \mathit{states}$, no states lower it in the state tree are included in *states*. In such a case, a default connector is added. Since we remove it, the inclusion between $\mathit{defaultEntranceComplete}(\sigma, \mathit{states}) \setminus \mathit{defaultfrom}(\mathit{states})$ and *conf* follows.

   (b) there are states lower than σ included in *states*. Since states form a tree, there are unique routes from σ to those $\mathit{states} \cap \rho^*(\sigma)$. All states along these routes will be included in the result of *defaultEntranceComplete*. Now we show that if something gets included which cannot be in a configuration, we have a contradiction. If there is an OR-state s such that

   $$\#(\rho(s) \cap \mathit{defaultEntranceComplete}(s, \mathit{states})) > 1$$

   then from definition of $\mathit{defaultEntranceComplete}(s, \mathit{states})$,

   $$\exists\, s_1, s_2 : \rho(s) \bullet \rho^*(s_1) \cap \mathit{states} \neq \varnothing \wedge \rho^*(s_2) \cap \mathit{states} \neq \varnothing$$

   which contradicts that *states* is a subset of a valid configuration. When considering $\mathit{defaultEntranceComplete}(s, \mathit{states})$, we rely on Prop. 6.1.46.

   The set of possible configurations $\mathit{conf} \bullet \mathit{states} \subseteq \mathit{conf}$ is unchanged by *defaultEntranceComplete*. According to Th. 6.1.13, the set of configurations encompassing the given set of states is determined by

   $$\mathit{treereduced}(\{s : \Sigma \mid \phi(s) \neq \mathit{stateDEFAULT}\})$$

   which is unchanged by Prop. 6.1.45 and item 2b of this theorem proven above.

3. follows from item 2 of this proposition proven above.

4. Two default connectors cannot be related by ρ^* due to them having no substates. Default connectors can have their lowest common ancestor *lca* to be either an OR-state or an AND-one. If it is an OR one, *lca* would have more than one of its substates entered by Prop. 6.1.10 and this contradicts that *defaultEntranceComplete* enters a subset of a valid configuration. Consequently, *lca* of every two default connectors entered should be an AND-state in which case the result follows from definition of *orth*.

5. Follows from the fact that every configuration is in the domain of *defaultEntranceComplete* and a transition always enters a set of states possible for some configuration by Prop. 6.1.22.

□

Now we define a configuration entered when we take a transition (*confENTERED*) and whether some transition is a continuation one.

Definition 6.1.48.

$$
\begin{array}{|l}
\mathit{confENTERED} : \mathit{TRANSITION} \times \mathbb{F}_1\, \mathit{STATE} \twoheadrightarrow\!\!\!\!\!\!\mapsto \mathbb{F}_1\, \mathit{STATE} \\
\hline
\forall\, tr : \Upsilon;\ \mathit{conf} : \mathbb{F}_1\, \mathit{STATE} \mid \mathit{configuration}(\mathit{root}, \mathit{conf}) \bullet \\
\qquad \mathit{confENTERED}(tr, \mathit{conf}) = \mathit{defaultEntranceComplete}(\\
\qquad\qquad \mathit{root}, \mathit{conf} \setminus \rho^*(\mathit{Uexit}(tr)) \cup tr.\mathit{target})
\end{array}
$$

$$
\begin{array}{|l}
\mathit{continuationDEF}\ \_ : \mathbb{F}_1(\mathit{TRANSITION} \times \mathit{TRANSITION}) \\
\hline
\forall\, tr_1, tr_2 : \Upsilon \bullet \\
\qquad \mathit{continuationDEF}(tr_1, tr_2) \Leftrightarrow tr_1 \neq tr_2 \land \\
\qquad\qquad \mathit{transitionDEFAULT}(tr_2) \land \mathit{transitionNI}(tr_2) \land \\
\qquad\qquad tr_2.\mathit{source} \subseteq \mathit{defaultEntranceComplete}(\mathit{Uenter}(tr_1), tr_1.\mathit{target})
\end{array}
$$

---

continuationDEF relates two transitions if the first one enters a non-basic state; in this case transitions from default connectors have to be taken to enter a configuration. Since such transitions are only considered with respect to states entered by the transition taken, i.e. $\rho^*(\mathit{Uenter}(tr))$, we do not need to consider a configuration from which the initial transition tr_1 is taken and thus this configuration is not included in the definition.

Definition 6.1.49. *The validity of the set of transitions from which a full compound transition is defined as follows:*

$$
\begin{array}{|l}
tsetVALID\ \_ : \mathbb{F}_1(TSet) \\
\hline
\forall\, tset : \mathbb{F}_1\, \Upsilon \bullet tsetVALID(tset) \Leftrightarrow \\
\quad orthogonal(\{t : tset \mid \neg\ transitionDEFAULT(t)\}) \land \\
\quad (\forall\, tr_c : tset \mid transitionDEFAULT(tr_c) \bullet \\
\qquad \exists\, tr_s : tset \bullet continuationDEF(tr_s, tr_c)) \land \\
\quad (\forall\, tr_s : tset \bullet (\exists\, tr_c : tset \bullet continuationDEF(tr_s, tr_c) \Rightarrow \\
\qquad (\exists_1\, tr_c : tset \bullet continuationDEF(tr_s, tr_c))))
\end{array}
$$

The definition states that all nondefault transitions of tset should be orthogonal; we cannot include default transitions in tset without those to which they serve as a continuation. The last line states that there should be no nondeterminism in selection of continuations.

Definition 6.1.50. *$toTRANSITION_D$ is defined similarly to $toTRANSITION$ (Def. 6.1.37 on p. 137). Its purpose is to combine transitions with their continuations.*

$$
\begin{array}{|l}
toTRANSITION_D : TSet \mathrel{+\!\!\!\twoheadrightarrow} TRANSITION \\
\hline
\forall\, tset : \mathbb{F}_1\, \Upsilon \mid tsetVALID(tset) \bullet \\
\quad toTRANSITION_D(tset) = \\
\qquad \langle\!|\ source == \bigcup\{t : tset \mid \neg\ transitionDEFAULT(t) \bullet t.source\}, \\
\qquad target == \bigcup\{tr_s : tset \\
\qquad\quad \neg\ (\exists\, tr_c : tset \bullet continuationDEF(tr_s, tr_c)) \bullet tr_s.target\}, \\
\qquad label == andALL(tset)\ |\!\rangle
\end{array}
$$

Proposition 6.1.51. *$toTRANSITION_D$ possesses all properties of $toTRANSITION$, i.e.*

$$
\begin{array}{l}
\forall\, tset : \mathbb{F}_1\, \Upsilon \mid orthogonal(tset) \land (\forall\, t : tset \bullet \neg\ transitionDEFAULT(t)) \bullet \\
\quad toTRANSITION(tset) = toTRANSITION_D(tset)
\end{array}
$$

Proof. Since $toTRANSITION$ is only defined for non-default transitions, comparison of definitions of the two gives the result. □

Proposition 6.1.52. *Transitions related by continuationDEF cannot be orthogonal and their scopes are related with* ρ^+*,*

$$\forall\, tr_c, tr_s : \Upsilon \mid transitionDEFAULT(tr_c) \wedge continuationDEF(tr_s, tr_c) \bullet$$
$$\neg\ orthogonal(\{tr_s, tr_c\}) \wedge scope(tr_c) \in \rho^+(scope(tr_s))$$

Proof. We shall prove that for tr_s and tr_c considered, their target states are not *orth*. This proof is illustrated by Fig. 6.12. Consider a continuation

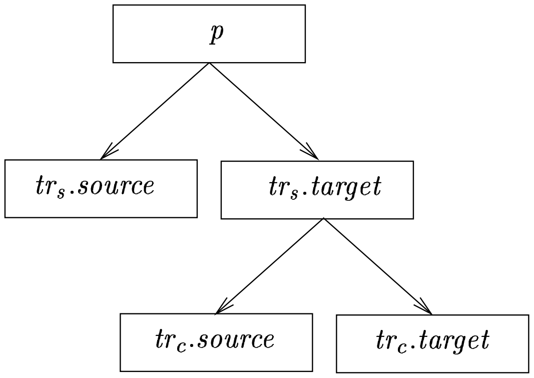


Figure 6.12: An illustration for the proof of Prop. 6.1.52

transition tr_c such that $\exists\, s : tr_s.target \bullet FromSet(tr_c.source) \in \rho(s)$. From Req. 4f, we get that $tr_c.target \subseteq \rho^+(s)$ implying $\exists\, s_c : tr_c.target \bullet s_c \in \rho^+(s)$ which contradicts the definition of orthogonality of tr_s and tr_c
$\forall\, s_1 : tr_s.target;\ s_2 : tr_c.target \bullet orth(s_1, s_2)$.

In order to prove the result about scopes, we observe that the enclosing state Sc of a default transition tr_c is its scope:

$$Sc = parent(FromSet\ tr_c.source) = scope(tr_c)$$

This follows from that $FromSet\ tr_c.source \in \rho(Sc)$ by defition of a default transition; $tr_c.target \subseteq \rho^+(Sc)$ from Req. 4f, giving $Sc = scope(tr_c)$.

By definition of *continuationDEF*, $\exists\, s : tr_s.target \bullet Sc \in \rho^*(s)$. Since $s \in \rho^+(scope(tr_s))$, $scope(tr_c) \in \rho^+(scope\ tr_s)$. □

Proposition 6.1.53. *A scope of a full compound transition is related to the scope of an initial transition with* ρ^**.*

$$\forall\, initial : \Upsilon;\ tset : \mathbb{F}_1\, \Upsilon \mid \neg\ transitionDEFAULT(initial) \wedge$$
$$(\forall\, tr : tset \bullet transitionDEFAULT(tr)) \bullet$$
$$(\exists_1\, fct == toTRANSITION_D(\{initial\} \cup tset) \bullet$$
$$transitionFCT(fct) \Rightarrow scope(fct) \in \rho^*(scope(initial)))$$

Proof. Consider

$$bs = lcoa(initial.source \cup initial.target \cup \bigcup_{t:tset} t.source \cup t.target)$$

It differs from $scope(fct)$ in that it includes states corresponding to default connectors. From Prop. 6.1.52, we get that $bs = scope(initial)$. Since we exclude some states from the above set of states, the $lcoa$ might potentially go down, $scope(fct) \in \rho^*(scope(initial))$. Fig. 6.13 gives an example that $scope(fct) \in \rho(scope(initial))$. $\square$

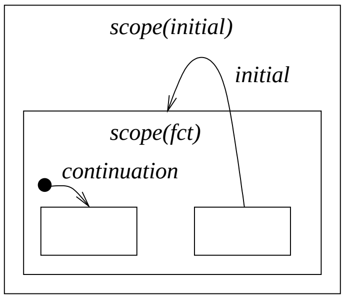


Figure 6.13: An example of $scope(fct) \in \rho(scope(initial))$

Following step semantics, we can construct a full compound transition from a number of initial ones by adding continuation transitions related by $continuationDEF$. This is given by the $FULL\_COMPOUND$ function. Here we use an assumption that default transitions are just part of full compound ones, rephrasing Req. 4b.

Definition 6.1.54.

$$
\begin{array}{|l}
FULL\_COMPOUND : TSet \times (STATE \nrightarrow TRANSITION) \nrightarrow TSet \\
\hline
\forall tset : \mathbb{F}_1\, \Upsilon;\ nondetres : \Sigma \nrightarrow \Upsilon \mid tsetVALID(tset) \bullet \\
\quad \exists result : TSet \bullet FULL\_COMPOUND(tset, nondetres) = result \land \\
\qquad tset \subseteq result \land \\
\qquad (\forall tr_c : result \setminus tset \bullet \exists tr_s : result \bullet \\
\qquad\quad tr_c \in nondetres(\!| tr_s.target |\!) \land continuationDEF(tr_s, tr_c) \land \\
\qquad\quad \neg\, (\exists tr_t : tset \bullet tr_t.source = tr_c.source)) \land \\
\qquad \neg\, (\exists tr_c : \Upsilon \setminus result;\ tr_s : result \bullet \\
\qquad\quad tr_c \in nondetres(\!| tr_s.target |\!) \land continuationDEF(tr_s, tr_c))
\end{array}
$$

Since we could have multiple transitions going from a default connector, we need to decide which one of them will be used. Considering that there can be only one default connector in a state, this can be expressed by $\Sigma \nrightarrow \Upsilon$. $tset$

may have transitions in it which are related by continuation, i.e. *rew_or_ff-ff*. This is needed if we wish to enter a specific state rather than just some valid configuration under a given state. Transitions of *tset* are treated to have priority over those which are selected via *nondetres*. This is expressed with line $\neg\ (\exists\, tr_t : tset \bullet tr_c.source = tr_t.source)$.

Well-definedness of *FULL_COMPOUND* is related to *nondetresolution* function passed to it. This function decides which state is entered from a default connector, if more than one transition exists such as *ff* and *rew* from the default connector in state *REW_FF* in Fig. 1.6.

Proposition 6.1.55. *FULL_COMPOUND is well-defined.*

Proof. Assume that there are two sets of resulting transitions, $result_1$ and $result_2$ for which the predicate part of the definition of *FULL_COMPOUND* is satisfied. We shall now show that they are the same.

Consider $tr_c \in result_1 \setminus tset$. Then there is

$$tr_{s\,a} \in result_1 \bullet continuationDEF(tr_{s\,a}, tr_c)$$

We can then take a $tr_{s\,b} \in result_1 \bullet continuationDEF(tr_{s\,b}, tr_{s\,a})$ and so on until we reach a transition in *tset*. This will eventually occur since transitions related by *continuationDEF* have their scopes related with ρ^+ (Prop. 6.1.52) and thus when taking $tr_{s\,b}$, $tr_{s\,c}$ etc we have a sequence of their scopes going higher on the state hierarchy, which is bounded by the *root* state.

Let us assume that $tr_{s\,a} \in tset$, then tr_c is contained in $result_2$ since otherwise the property

$$\begin{array}{l}\neg\ (\exists\, tr_c : \Upsilon \setminus result;\ tr_s : result \bullet \\ \quad tr_c \in nondetres(\!|\, tr_s.target\,|\!) \wedge continuationDEF(tr_s, tr_c))\end{array}$$

will be not satisfied (we assume *nondetres* is a function).

We can take another transition tr_{c2} from $result_1$ such that there is $tr_{s2} : tset \cup \{tr_c\}$ and similarly show that tr_{c2} is included in $result_2$ too. Thus, we get that $result_2 = result_1$. □

Although the definition Def. 6.1.54 given above seems to be relatively easy to understand, it is cumbersome in proofs. We rectify this with the following proposition:

Proposition 6.1.56. *The FULL_COMPOUND function can be given an operational definition*

$$\begin{array}{|l}
gettrc : TSet \times (\Sigma \mathrel{+\!\!\!\rightarrow\!\!\!\rightarrow} \Upsilon) \leftrightarrow TRANSITION \\
FULL\_COMP : TSet \times (STATE \mathrel{+\!\!\!\rightarrow\!\!\!\rightarrow} TRANSITION) \mathrel{+\!\!\!\rightarrow\!\!\!\rightarrow} TSet \\
\hline
\forall tset : \mathbb{F}_1\, \Upsilon;\ nondetres : \Sigma \mathrel{+\!\!\!\rightarrow\!\!\!\rightarrow} \Upsilon \mid tsetVALID(tset) \bullet \\
\quad (\exists tr_c : \Upsilon \setminus tset;\ tr_s : tset \bullet \\
\qquad tr_c \in nondetres(\!| tr_s.target |\!) \land continuationDEF(tr_s, tr_c) \Rightarrow \\
\qquad gettrc(tset, nondetres) = tr_c) \\
\forall tset : \mathbb{F}_1\, \Upsilon;\ nondetres : \Sigma \mathrel{+\!\!\!\rightarrow\!\!\!\rightarrow} \Upsilon \mid tsetVALID(tset) \bullet \\
\quad ((tset, nondetres) \notin \mathsf{dom}\, gettrc \Rightarrow FULL\_COMP(tset, nondetres) = tset) \\
\quad \land \\
\quad ((tset, nondetres) \in \mathsf{dom}\, gettrc \Rightarrow FULL\_COMP(tset, nondetres) = \\
\qquad FULL\_COMP(tset \cup \{gettrc(tset, nondetres)\}, nondetres))
\end{array}$$

Proof. First of all, we observe that by construction of *FULL_COMP*, all transitions added to *tset* by *FULL_COMP* are satisfying

$$tr_c \in nondetres(\!| tr_s.target |\!) \land continuationDEF(tr_s, tr_c)$$

and not in conflict with existing ones,

$$\neg\, (\exists tr_t : tset \bullet tr_t.source = tr_c.source)$$

Upon termination of it (if any), we get for
$result = FULL\_COMP(tset, nondetres)$ that

$$\begin{array}{l}
\neg\, (\exists tr_c : \Upsilon \setminus result;\ tr_s : result \bullet \\
\quad tr_c \in nondetres(\!| tr_s.target |\!) \land continuationDEF(tr_s, tr_c))
\end{array}$$

This result also follows by construction of *FULL_COMP*.

The result of the *FULL_COMP* may only be affected by an order continuation transitions are added to *tset* (i.e. taking some transition would not allow us to take another one), if for some set of initial transitions and their continuations, there are continuations possible which are not orthogonal. This can only happen if there is more than one continuation transition possible from a default connector, which is rendered impossible by *nondetres* being a function. This gives well-definedness of *FULL_COMP*.

From the results proven above we get that the result of *FULL_COMP* satisfies the property of *FULL_COMPOUND*. It remains to show that everything constructed by *FULL_COMPOUND* can also be constructed using

FULL_COMP. Since domains of the two are the same, we only need to concern ourselves with well-definedness of *FULL_COMPOUND* which is proven in Prop. 6.1.55. Consequently, we get that

$$FULL\_COMPOUND(tset, nondetres) = FULL\_COMP(tset, nondetres)$$

□

Proposition 6.1.57. *FULL_COMP terminates.*

Proof. Follows from the fact that it only adds transitions and the number of them is finite. □

Due to *FULL_COMPOUND* and *FULL_COMP* being the same well-defined terminating function, in the rest of this chapter we will often refer to *FULL_COMPOUND*, but use the operational definition of *FULL_COMP*.

A set of transitions *tset* constructed by *FULL_COMPOUND* is equivalent to a full compound transition with source and target states united by *toTRANSITION* as shown below.

Proposition 6.1.58. *FULL_COMPOUND generates full compound transitions with source states being the union of source states of non-default transitions of tset,*

$$\begin{aligned}
&\forall\, tset : TSet;\ nondetres : STATE \twoheadrightarrow TRANSITION \mid tsetVALID(tset) \bullet \\
&\quad transitionFCT(toTRANSITION_D(FULL\_COMPOUND(tset, nondetres))) \land \\
&\quad (toTRANSITION_D(FULL\_COMPOUND(tset, nondetres))).source = \\
&\qquad \bigcup\{t : tset \mid \neg\, transitionDEFAULT(t) \bullet t.source\}
\end{aligned}$$

Proof. Orthogonality of target states of the considered FCT follows from Prop. 6.1.47, item 4 because all default connectors entered by *defaultEntranceComplete* are *orth* and from Req. 4f, as due to this requirement *toTRANSITION* returns target states which are *orth*. Source states are *orth* from definition of *tsetVALID*.

Let us denote

$$target = toTRANSITION_D(FULL\_COMPOUND(tset, nondetres)).target$$

and

$$source = toTRANSITION_D(FULL\_COMPOUND(tset, nondetres)).source$$

Due to Req. 4f, $root \notin target$. Since all transitions in *tset* are valid,

$$(\forall\, st : \Sigma \mid \phi(st) = stateAND \bullet (source \cup target) \cap \rho(st) = \varnothing)$$

and $source \neq \varnothing \wedge target \neq \varnothing$. From the definition of $toTRANSITION_D$ (Def. 6.1.50) and validity of transitions of *tset*, $source \cap target$ does not contain default transitions. This proves the validity of transitions generated by *FULL_COMPOUND*. Since only default transitions are removed from the union of source states by $toTRANSITION_D$,

$$source = \bigcup_{t:tset \mid \neg\, transitionDEFAULT(t)} t.source$$

By definition of $toTRANSITION_D$, only basic states are present in $toTRANSITION_D(FULL\_COMPOUND(t)).target$ which completes the proof. □

Since *FULL_COMPOUND* adds default transitions to the set of those it was supplied with, one would expect to be able to apply it once again, which is proven in the following proposition.

Proposition 6.1.59. *FULL_COMPOUND returns a set of transitions which is tsetVALID; if applied to a result of FULL_COMPOUND, FULL_COMPOUND returns the same set.*

$$\begin{aligned}
&\forall\, tset : TSet;\ nondetres : \Sigma \twoheadrightarrow \Upsilon \mid tsetVALID(tset) \bullet \\
&\quad tsetVALID(FULL\_COMPOUND(tset, nondetres)) \wedge \\
&\quad FULL\_COMPOUND(FULL\_COMPOUND(tset, nondetres), nondetres) = \\
&\qquad FULL\_COMPOUND(tset, nondetres)
\end{aligned}$$

Proof. We consider three statements of the definition of *tsetVALID* in turn,

1. From definition of *FULL_COMPOUND*, it only adds default transitions and thus the set of non-default ones remains the same. For this reason,

   $orthogonal(\{t : FULL\_COMPOUND(tset) \mid \neg\, transitionDEFAULT(t)\})$

2. Follows from definition of *FULL_COMPOUND*.

3. Follows from Prop. 6.1.55.

$FULL\_COMPOUND(FULL\_COMPOUND(tset)) = FULL\_COMPOUND(tset)$ follows from the definition of *FULL_COMPOUND* since default transitions already included in *tset* have precedence over those which could be added by *FULL_COMP* and the inner *FULL_COMP* has added all of those which could be added. □

As shown in Tab. 6.1, transitions which could be taken in the same step (*nonconflict*) are orthogonal and thus could be united into a single transition with *toTRANSITION*. Default continuations could also be added to them and the result will be a full compound transition. Since we cannot split single full compound transitions into parts all of which could be taken in the same step (*orthogonal* $\not\Rightarrow$ *nonconflict*) as described in Sect. 6.1.2 on p. 135, it would be nice to be able to apply *FULL_COMPOUND* to non-conflicting compound transitions and have the result 'splittable' into non-conflicting full compound transitions. This is indeed so as we show below.

From the definition of *FULL_COMPOUND* (Def. 6.1.54), initial transitions of *tset* are preserved in the output and as only default ones are added, we can extract those initial transitions from the result of it. Since FCTs have a single initial compound transition (Sect. 1.4.5 on p. 16), each such initial transition is a beginning of a separate full compound one.

Proposition 6.1.60. *The result of FULL_COMPOUND can be split into a number of full compound transitions each of which starting from a different initial one and all of them non-conflicting if initial transitions are.*

$$
\begin{array}{l}
\forall\, tset : TSet \mid tsetVALID(tset) \wedge \\
\quad orthscope(\{t : tset \mid \neg\, transitionDEFAULT(t)\}) \bullet \\
\quad \exists\, fcts : TSetSet \bullet \\
\qquad \bigcup fcts = FULL\_COMPOUND(tset, nondetres) \wedge \\
\qquad (\forall\, fct_1, fct_2 : fcts \bullet fct_1 \cap fct_2 = \varnothing) \wedge \\
\qquad (\forall\, fct : fcts \bullet \\
\qquad\quad (\exists_1\, t : fct \bullet t \in tset \wedge \neg\, transitionDEFAULT(t))) \wedge \\
\qquad\qquad orthscope(\{fct : fcts;\ t : TRANSITION \mid \\
\qquad\qquad\quad t \in fct \wedge \neg\, transitionDEFAULT(t) \bullet t\})
\end{array}
$$

Proof. Consider the following function *fcti* which takes a set of transitions *tset*, an initial one *initial* and extracts all continuation transitions for *initial* from *tset*:

$$
\begin{array}{|l}
fcti : TSet \times TRANSITION \twoheadrightarrow TSet \\
\hline
\forall\, tset : \mathbb{F}_1\, \Upsilon;\ initial : \Upsilon \mid tsetVALID(tset) \wedge initial \in tset \bullet \\
\quad initial \in fcti(tset, initial) \wedge \\
\quad (\forall\, tr_c : fcti(tset, initial) \setminus \{initial\} \bullet (\exists\, tr_s : fcti(tset, initial) \bullet \\
\qquad continuationDEF(tr_s, tr_c))) \wedge \\
\quad \neg\, (\exists\, tr_s : fcti(tset, initial);\ tr_t : FULL\_COMPOUND(tset, nondetres) \bullet \\
\qquad continuationDEF(tr_s, tr_t))
\end{array}
$$

Consider $fcts = \{t : tset \mid \neg\, transitionDEFAULT(t) \bullet fcti(tset, t)\}$. From similarity to Def. 6.1.54, we get that all transitions originally included in *tset* will appear in $\bigcup fcts$ since *FULL_COMPOUND* is well-defined (Prop. 6.1.55).

Since all the initial transitions in $fcti(tset, t)$ above are expected to be initial in full compound transitions which can be taken in the same step, from Th. 6.1.31 we get that scopes of all of them are orthogonal.

We show that there is no duplication of transitions in different *fcts*, $\forall fct_1, fct_2 : fcts \bullet fct_1 \cap fct_2 = \varnothing$. Initial transitions $initial_1, initial_2$ are such that $orth(scope(initial_1), scope(initial_2))$. Consider a shared continuation transition, tr_c, then

$$\exists\, tr_{s1} : fct_1;\ tr_{s2} : fct_2 \bullet continuationDEF(tr_{s1}, tr_c) \wedge continuationDEF(tr_{s2}, tr_c)$$

and

$$scope(tr_c) \in \rho^+(scope(initial_1)) \wedge scope(tr_c) \in \rho^+(scope(initial_2))$$

from Prop. 6.1.52. This contradicts orthogonality of scopes of $initial_1$ and $initial_2$ as it implies their relation by ρ^*.

Every $fct : fcts$ above is *tsetVALID*: it has only one non-default transition (*initial*) and thus $orthogonal(\{initial\})$; the second and third statements of Def. 6.1.49 are satisfied by construction of *fct*.

From orthogonality of scopes of initial transitions and Prop. 6.1.53, we get from Prop. refth:orth-properties that scopes of full compound transitions in the set *fcts* are orthogonal. By Th. 6.1.31, we get that they are not in conflict. □

Note that the above theorem also is true for initial transitions which are interlevel and enter a set of states.

6.1.4 Transition priorities and structural determinism

When multiple conflicting transitions are enabled, the conflict may be resolved via a concept of priorities. It means that transitions are given priorities and a transition with the highest one is taken; if a number of transitions with the highest priority is enabled, nondeterminism is reported. The concept of transition priority, related to state hierarchy, is absent in X-machines, where if more than one transition is enabled, this always means nondeterminism.

Definition 6.1.61. *Priority is determined by scope such that the transition with a higher-level scope state is of higher priority,* $SAnc(scope(t_1), scope(t_2))$*. For example, transition* ff *between* REWIND *and* F_ADVANCE *states in Fig. 1.6 has its scope* REW_FF *and* play *transition from the* REW_FF *state — the main statechart.*

It is nice to know that for a given set of changes the response received from a system will only depend on its internal state. This is one of the requirements for the testing method. It is formalised below.

Definition 6.1.62. *A statechart is structurally deterministic if in any valid configuration there are no conflicting (Def. 6.1.30) transitions of the same priority [MLPS97].*

$$\begin{array}{l}\forall\, conf : \mathbb{F}_1\,\Sigma;\ tset : TSet \mid configuration(root, conf) \wedge tset \subseteq \Upsilon \bullet \\ \quad (\forall\, t_1, t_2 : tset \bullet t_1.source \subseteq conf \wedge t_2.source \subseteq conf \wedge \\ \quad (\neg\, nonconflict(t_1, t_2, conf) \Rightarrow \\ \qquad (SAnc(scope(t_1), \{scope(t_2)\}) \vee SAnc(scope(t_2), \{scope(t_1)\}))))\end{array}$$

Definition 6.1.63. *Implementation is considered deterministic if every sequence of changes and memory values correspond to one sequence of transitions.*

Although the proof may seem to be showing that no nondeterminism may occur during testing, it does not consider possible faults in an implementation. The above definition, being a reflection of Req. 1b, can be seen as a remedy. While looking too restrictive, it can be justified from the implementation point of view such that conditional operators in most programming languages exhibit deterministic behaviour, due to precisely defined evaluation order. From test set construction, we can also expect deterministic execution as follows from the following proposition.

Proposition 6.1.64. *Conflicting transitions are never triggered provided they are not generated to be in the set of test cases.*

Proof. Since we trigger only non-conflicting transitions we wish to take and not others (definition of t_complete, Def. 6.6.1), and there are no shared transitions on a different level of hierarchy (Def. 1e), we get that no conflicting transitions will be triggered. □

This proposition also allows us not to consider priorities of transitions, for test case generation. Priorities are taken into account when we talk about refinement (Sect. 6.5 on p. 195). Conflicting transitions are later shown not to be in the set of test cases (Prop. 6.4.18).

6.1.5 Paths

A path is a sequence of sets of transitions. Paths which *exist* must have their sets contain transitions which may be executed in the same step and together comprise a full compound transition. Intuitively, an existing path can be taken by a statechart in a superstep consisting of a number of steps.

In each of these steps, a set of transitions of a single element in the path is executed; transitions in such a set have to be enabled. For example, the transition to *F_ADVANCE* from the initial state is *rew_or_ff-ff* and a path there can be written as $\langle\{rew\_or\_ff, ff\}\rangle$, because we have to take both *rew_or_ff* and *ff* in the same step. The sequence from the initial state to the *PLAY* state through the *F_ADVANCE* one is *rew_or_ff-ff play* which is $\langle\{rew\_or\_ff, ff\}\,\{play\}\rangle$. Existence of a path is given by the *pathEXISTS* predicate.

Definition 6.1.65. *A path being a sequence of sets of full compound transitions can be taken in a step in some configuration if the pathEXISTS predicate is satisfied.*

$$
\begin{array}{|l}
pathEXISTS\ \_ : \mathbb{F}_1(TSeqSet \times \mathbb{F}_1\ STATE) \\
\hline
\forall\, path : TSeqSet;\ conf : \mathbb{F}_1\ STATE \mid configuration(root, conf) \bullet \\
\quad path = \langle\rangle \land pathEXISTS(path, conf) \lor \\
\quad path \neq \langle\rangle \land (\exists_1\ trans == toTRANSITION_D(head\ path) \bullet \\
\qquad orthscope(head\ path) \land \\
\qquad trans.source \subseteq conf \land \\
\qquad pathEXISTS(tail\ path, confENTERED(trans, conf)))
\end{array}
$$

6.1.6 Multiplication types

If we wish to construct a set of test cases from a test case basis, we use the sequential set multiplication which essentially concatenates sequences. For merging test case bases from those of OR or AND states, we take appropriate transitions in the same step. This can be illustrated by the tape recorder with a clock shown in Fig. 6.14. Results of multiplication of

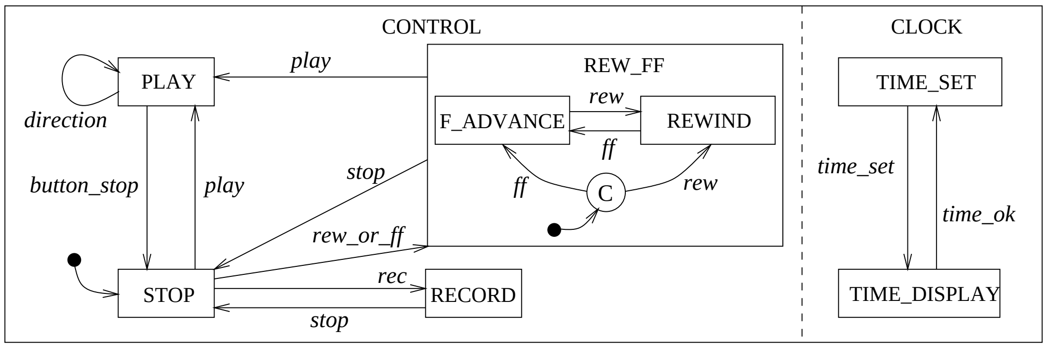


Figure 6.14: The tape recorder with a clock

$\{$*rew_or_ff-ff play*$\}$ by $\{$*time_set time_ok*$\}$ are given in Tab. 6.2. Sequences

oper.	result of multiplication
$\underline{*}$	{ *rew_or_ff* - *ff* - time_set play - time_ok }
$*_1$	{ *rew_or_ff* - *ff* play - time_set time_ok }
$*$	{ *rew_or_ff* - *ff* play time_set time_ok }

Table 6.2: Results of multiplication of $\{$*rew_or_ff-ff play*$\}$ by $\{$*time_set time_ok*$\}$, depending on the type of multiplication

to be AND-multiplied do not have to be of the same length,

$$\{play\ direction\}\underline{*}\{time\_set\} = \{play\text{-}time\_set\ direction\}$$

In the merging of test bases and test case generation we use operators $*$, $*_1$, $\underline{*}$ which correspond to functions *multOR*, *multOR*1, and *multAND*. Operators are easier to understand in explanations while functions are easier to use in proofs.

Consider multiplications of transitions in Def. 6.1.17, then from Prop. 6.1.38 it follows, that *multAND*[*TSetSeqSet*] generates valid paths when supplied with valid ones. Other multiplication functions always generate valid transitions because they do not combine them in the way what *Unite*[*TRANSITION*] does in $a(n) \cup b(n)$.

Proposition 6.1.66. *Consider a set of transitions which can be taken in the same step, i.e. orthscope. For a result of multAND[TSetSeqSet], we can always reconstruct original transitions if all sequences multAND[TSetSeqSet]ed were of length 1.*

Proof. Follows from Prop. 6.1.60. □

In the rest of the thesis, we shall not give *multAND* a type explicitly since the type of sets of sequences of sets it is applied to, shall be clear from the context. It can be either *TSetSeqSet* or *LSetSeqSet*.

6.2 Flattening of a statechart

The flattened statechart can be described as:

Definition 6.2.1.

$$\begin{array}{|l} \Sigma_f : \mathbb{F}\, SSet \\ \Upsilon_f : TSet \\ execTRANS : TRANSITION \mathrel{+\!\!\!\!\twoheadrightarrow} LABEL \end{array}$$

where Σ_f corresponds to a configuration of the original statechart and Υ_f-to a set of transitions which could be taken in the same step. Note that we do not try to find infeasible sequences of transitions since the complexity of that in real applications is expected to far outweigh difficulties with augmentation in order to make a statechart satisfy the testing requirements. For a label *label*, *execTRANS*(*label*) is the behaviour of a transition with label *label*. The difference of it from *label* is the consideration of priorities of transitions.

6.2.1 Flattening of state hierarchy

stateFLATTEN partial function gives the flattening of a statechart by relating a configuration to an appropriate state of a flattened statechart.

Definition 6.2.2.

$$\begin{array}{|l} stateFLATTEN : STATE \times \mathbb{F}_1\, STATE \mathrel{+\!\!\!\!\twoheadrightarrow} \mathbb{F}_1\, STATE \\ \hline \forall\, s : \Sigma;\ conf : \mathbb{F}_1\, STATE \mid configuration(s, conf) \bullet \\ \qquad (\phi(s) = stateBASIC \Rightarrow stateFLATTEN(s, conf) = \{s\}) \wedge \\ \qquad (\phi(s) = stateOR \Rightarrow \\ \qquad\qquad (\exists\, s_{sub} : conf \cap \rho(s) \bullet stateFLATTEN(s, conf) = \\ \qquad\qquad\qquad stateFLATTEN(s_{sub}, conf))) \wedge \\ \qquad (\phi(s) = stateAND \Rightarrow \\ \qquad\qquad stateFLATTEN(s, conf) = \bigcup(stateFLATTEN(\!|\rho(s) \times \{conf\}|\!))) \end{array}$$

Proposition 6.2.3. *stateFLATTEN provides substates for all lowest-level OR-states of a configuration,*

$$\begin{aligned} &\forall\, conf : \mathbb{F}_1\, \Sigma \mid configuration(root, conf) \bullet stateFLATTEN(root, conf) = \\ &\qquad \{s : conf \mid \rho^{+}(s) \cap conf = \varnothing \wedge \phi(parent(s)) = stateOR\} \end{aligned}$$

In other words, stateFLATTEN is an operational description of treereduced.

Proof. The set of basic states generated by *stateFLATTEN* provides unique substates of all OR-states which follows from definition of *stateFLATTEN* (Def. 6.2.2) (if it does not, i.e.

$$\exists\, s : conf \bullet \phi(s) = stateOR \wedge conf \cap \rho^{+}(s) = \varnothing,$$

from Def. 6.1.3 *conf* is not a valid configuration). □

Now when considering full compound transitions in flattened statecharts beginning with a set of transitions *tset* and followed by continuation transitions, we can use

$$stateFLATTEN(root, toTRANSITION(tset).source)$$

to describe the source state in the flattened statechart and

$$stateFLATTEN(root, toTRANSITION(FULL\_COMPOUND(tset)).target)$$

for the target one. Using the *stateFLATTEN* function, we can define the flattened statechart as follows.

Definition 6.2.4.

$$
\begin{array}{|l}
allFULLCOMPOUND : TSet \twoheadrightarrow\!\!\!\!\!\!\mapsto TSetSet \\
toFCT : TSet \twoheadrightarrow\!\!\!\!\!\!\mapsto TSetSet \\
\hline
\forall\, tset : \mathbb{F}_1\, \Upsilon \mid tsetVALID(tset) \bullet \\
\quad allFULLCOMPOUND(tset) = Composition(setMULT, \\
\qquad \{tr : tset \bullet \{ndr : \Sigma \twoheadrightarrow\!\!\!\!\!\!\mapsto \Upsilon \mid (\forall\, s : \Sigma \mid \phi(s) = stateOR \bullet \\
\qquad\quad scope(ndr(s)) \in \rho(s) \wedge transitionDEFAULT(ndr(s))) \bullet \\
\qquad\qquad FULL\_COMPOUND(\{tr\}, ndr)\}\} \\
\qquad) \\
\forall\, trs : \mathbb{F}_1\, \Upsilon \bullet toFCT(trs) = \\
\quad \bigcup\{tset : TSet \mid tset \subseteq trs \wedge \\
\qquad \neg\,(\exists\, tr : tset \bullet transitionDEFAULT(tr)) \wedge orthscope(tset) \bullet \\
\qquad allFULLCOMPOUND(tset)\}
\end{array}
$$

$$
\begin{array}{|l}
\Sigma_f : \mathbb{F}\, SSet \\
\Upsilon_f : TSet \\
execTRANS : TRANSITION \twoheadrightarrow\!\!\!\!\!\!\mapsto LABEL \\
\hline
\Sigma_f = \{conf : \mathbb{F}_1\, \Sigma \mid configuration(root, conf) \bullet stateFLATTEN(root, conf)\} \\
\Upsilon_f = \{t : toTRANSITION_D(\!| toFCT(\Upsilon) |\!);\ conf : \mathbb{F}_1\, \Sigma \mid \\
\quad configuration(root, conf) \wedge t.source \subseteq conf \bullet \\
\quad \langle\!| \ source == stateFLATTEN(root, conf), \\
\quad target == stateFLATTEN(root, confENTERED(t, conf)), \\
\quad label == t.label \ |\!\rangle\} \\
\exists_1\, trans == toTRANSITION_D(\!| toFCT(\Upsilon) |\!) \bullet \\
\quad \forall\, t : trans;\ conf : \mathbb{F}_1\, \Sigma \mid \\
\qquad configuration(root, conf) \wedge t.source \subseteq conf \bullet \\
\qquad\quad execTRANS(t) = (\bigcup\{tr : trans \mid \\
\qquad\qquad tr.source \subseteq conf \wedge scope(t) \in \rho^+(scope(tr)) \bullet \\
\qquad\qquad \mathsf{dom}\ tr.label\}) \\
\qquad\qquad \lhd t.label
\end{array}
$$

All possible completions of a set of transitions are given by *allFULLCOMPOUND*, which makes use of all possible *ndr* — default completion (stands for nondeterminism resolution) functions. It completes every transition in the *tset* given to it in all possible ways. We then take such a set of completions of all transitions in *tset* and multiply all that together. This means combining every possible completion of one transition with that of another one and so on. The process described relies on the following fact:

Proposition 6.2.5.

$$\begin{array}{l} \forall a, b : \Upsilon \mid orthscope(\{a, b\}) \land \\ \quad \neg\, transitionDEFAULT(a) \land \neg\, transitionDEFAULT(b) \bullet \\ \qquad FULL\_COMPOUND(\{a, b\}, nondetres) = \\ \qquad\quad FULL\_COMPOUND(\{a\}, nondetres) \cup \\ \qquad\quad FULL\_COMPOUND(\{b\}, nondetres) \end{array}$$

Proof. Follows from Prop. 6.1.60 as it explains how to extract $FULL\_COMPOUND(\{a\})$ and $FULL\_COMPOUND(\{b\})$ from $FULL\_COMPOUND(\{a, b\})$. □

The *execTRANS* in Def. 6.2.4 resolves some cases of nondeterminism resulting from flattening conflicting transitions. This function takes a transition and constructs a label from that of it by removing domains of higher-level transitions from its domain. This is similar to priority resolution computations in [Bur99]. In our case, a non-*andTRUE* label may be used in only one state which makes it possible to define *execTRANS* as *execLABEL* (using the *getSCOPE* function).

Usage of *Composition* with *setMULT* is made well-defined by Req. 3b.

In order to convert states in Σ_f to configurations, we define

Definition 6.2.6.

$$\begin{array}{|l} toCONFIGURATION : SSet \twoheadrightarrow\!\!\!\!\!\!\!\!\;\;| \;\; SSet \\ \hline \forall st : \Sigma_f \bullet (\exists_1\, conf == \bigcup\{s : st \bullet route(root, s)\} \bullet \\ \quad configuration(root, conf) \land conf = toCONFIGURATION(st)) \end{array}$$

Proposition 6.2.7. *toCONFIGURATION is well-defined and is essentially a reverse of stateFLATTEN.*

$$\begin{array}{l} \forall st : \Sigma_f \bullet stateFLATTEN(root, toCONFIGURATION(st)) = st \\ \forall conf : \mathbb{F}_1\, \Sigma \mid configuration(root, conf) \bullet \\ \quad toCONFIGURATION(stateFLATTEN(root, conf)) = conf \end{array}$$

Proof. Follows from Prop. 6.2.3 and Th. 6.1.13. □

Now we adapt the definition of a step of statecharts from [MLPS97].

Definition 6.2.8.

$$
\begin{array}{|l}
\textit{MLPS\_Step} \\
\hline
data, data' : DATA \\
conf, conf' : \mathbb{F}\,\Sigma \\
\hline
(\exists_1\, ET == \{tr : \Upsilon \mid enable(tr, data, conf)\} \bullet \\
(\exists_1\, HPT == \{etr : ET \mid (\forall\, tr : ET \bullet \neg\, SAnc(scope(tr), \{scope(etr)\}))\} \bullet \\
(\exists_1\, MNS == (\mu\, ncs : \mathbb{F}(\mathbb{F}\, HPT) \mid (\forall\, tset : ncs \bullet orthscope(tset)) \land \\
\quad (\forall\, tset : ncs;\ t : HPT \bullet orthscope(\{t\} \cup tset) \Rightarrow t \in tset)) \bullet \\
\quad (\#MNS = 0 \Rightarrow conf' = conf \land data' = data) \land \\
\quad (\#MNS \neq 0 \Rightarrow (\exists\, EN : MNS \bullet \\
\qquad conf' = (conf \setminus \bigcup\{t : EN \bullet exit(t, conf)\}) \cup \\
\qquad\quad \bigcup\{t : EN \bullet enter(t, conf)\} \land \\
\qquad data' = modify(andALL(EN)\ data, data)))))) \\
\hline
\end{array}
$$

Abbreviations:

ET	*enabled transitions*
HPT	*enabled transitions such that no two transitions which are in conflict which can be resolved via priorities are included*
MNS	*sets of maximal non-conflicting sets of transitions, which can be taken in a step*

The following proposition gives us confidence that flattened statechart has the same behaviour as the original one. It also shows our compliance to the semantics described in Def. 6.2.8 above.

Proposition 6.2.9. *A transition $t : \Upsilon_f$ will be enabled in $s : \Sigma_f$, if and only if the set of transitions corresponding to it is enabled in $toCONFIGURATION(s)$. There exists a set tset which is the member of the $toFCT(\Upsilon)$ set in Def. 6.2.4, from which t is constructed, such that $toTRANSITION_D(tset) = t$. In addition, t reflects a maximal non-conflicting set of transitions from a given configuration s.*

Proof. From Def. 6.2.4, transition $t : \Upsilon_f$ is triggered if all transitions of the corresponding set *tset* are and none of the higher-priority transitions is triggered. Conversely, if t is triggered, the corresponding set of transitions will be. Enabledness follows from Prop. 6.2.7.

Maximality of the set of transitions represented by $t : \Upsilon_f$ follows from construction of Υ_f and *execTRANS*. In the definition of the latter, for any transition $tr : \Upsilon_f$ we modify triggers of all lower-priority ones such they will not be triggered if the one under consideration is. Transitions from which t is constructed are not in conflict by construction of $toFCT(\Upsilon)$. □

6.2.2 Flattening of paths

Since a path is a sequence of sets of transitions with each of the sets taken in the same step, flattening corresponds to *FULL_COMPOUND*ing each set as provided below.

Definition 6.2.10.

$$\mathit{flattenPATH} : \mathit{TSeqSet} \pfun \mathit{TSeqSet}$$

$$\begin{array}{l}
\forall\, \mathit{path} : \mathit{TSeqSet} \mid \\
\quad (\exists\, \mathit{conf} : \mathbb{F}_1\, \Sigma \bullet \mathit{configuration}(\mathit{root}, \mathit{conf})) \bullet \exists\, \mathit{nondetres} : \Sigma \pfun \Upsilon \bullet \\
\quad \mathit{path} = \langle\rangle \land \mathit{flattenPATH}(\mathit{path}) = \langle\rangle \land \\
\quad \mathit{path} \neq \langle\rangle \land \mathit{flattenPATH}(\mathit{path}) = \\
\qquad \langle \mathit{FULL\_COMPOUND}(\mathit{head}\, \mathit{path}, \mathit{nondetres})\rangle \\
\qquad\quad {}^\frown \mathit{flattenPATH}(\mathit{tail}\, \mathit{path})
\end{array}$$

The result of *flattenPATH* is a sequence of transitions from Υ_f (follows from comparison of this definition and the one for *pathEXISTS*). An example of flattening is given in Fig. 6.15. Numbers 1-3 give transitions we take in steps 1, 2 and 3.

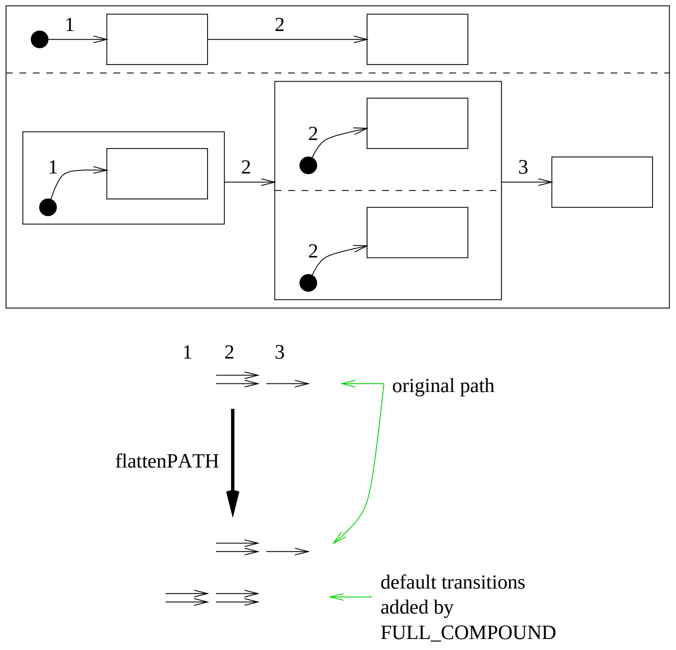


Figure 6.15: An example of path flattening

In a deterministic statechart (Req. 1b), we can also expect

$$\forall\, \mathit{path} \bullet \#\{\mathit{nondetres} : \Sigma \pfun \Upsilon \bullet \mathit{flattenPATH}(\mathit{path}, \mathit{nondetres})\} = 1$$

where *nondetres* is used as a parameter for *FULL_COMPOUND* used by *flattenPATH*. For this reason, *nondetres* is not a parameter of *flattenPATH*.

We shall prove that when paths are flattened, this does not affect our ability to verify existent/nonexistent paths in Sect. 6.4 on p. 178. First of all, definitions of conversions between sequences of labels (as used in characterisation set) and paths are given:

Definition 6.2.11. *The definition describes conversion functions between sets of labels and transitions (toLABELSET, toTRANSITIONSET), as well as between sequences of such sets (toLPATH, toPATH).*

$$\begin{array}{|l}
toLABELSET : TSet \twoheadrightarrow LSet \\
\hline
\forall tset : TSet \bullet toLABELSET(tset) = \{t : tset \mid t.label \neq andTRUE \bullet t.label\}
\end{array}$$

$$\begin{array}{|l}
toTRANSITIONSET : LSet \times \mathbb{F}_1\, STATE \twoheadrightarrow TSet \\
\hline
\forall lset : LSet;\ conf : \mathbb{F}_1\, STATE \mid configuration(root, conf) \bullet \\
\quad \exists\, tset : TSet \bullet pathEXISTS(\langle tset \rangle, conf) \wedge \\
\quad toTRANSITIONSET(lset, conf) = tset \wedge toLABELSET(tset) = lset \wedge \\
\quad \neg\, (\exists\, tset2 : TSet \bullet \#tset2 < \#tset \wedge \\
\qquad pathEXISTS(\langle tset \rangle, conf) \wedge toLABELSET(tset2) = lset)
\end{array}$$

$$\begin{array}{|l}
toLPATH : TSeqSet \twoheadrightarrow LSeqSet \\
\hline
\forall path : TSeqSet \bullet \\
\quad toLPATH(path) = apply(path, toLABELSET)
\end{array}$$

$$\begin{array}{|l}
toPATH : LSeqSet \times \mathbb{F}_1\, STATE \twoheadrightarrow TSeqSet \\
\hline
\forall lpath : LSeqSet;\ conf : \mathbb{F}_1\, \Sigma \mid configuration(root, conf) \bullet \\
\quad lpath = \langle\rangle \Rightarrow toPATH(lpath, conf) = \langle\rangle \wedge \\
\quad lpath \neq \langle\rangle \Rightarrow \\
\qquad (\exists_1\, trans == toTRANSITIONSET(head\ lpath, conf) \bullet \\
\qquad\quad toPATH(lpath, conf) = \langle trans \rangle \frown toPATH(tail\ lpath, \\
\qquad\qquad confENTERED(toTRANSITION(trans), conf)))
\end{array}$$

The term *lpath* stands for a sequence of labels, thus it starts with the small letter 'l'. It may correspond to an existing path (sequence of transitions) or not.
The

$$\neg\, (\exists\, tset2 : TSet \bullet \#tset2 < \#tset \wedge \\ pathEXISTS(tset) \wedge toLABELSET(tset2) = lset)$$

lines in the definition of *toTRANSITIONSET* above together with

$$pathEXISTS(\langle tset\rangle, conf)$$

statement in the same definition make certain that all relevant default transitions with empty labels are included in the set of transitions returned by *toTRANSITIONSET*. This relies on Prop. 6.1.27.

Proposition 6.2.12. *For all sets of labels corresponding to a set of transitions possible from a given configuration, toTRANSITIONSET is well-defined.*

Proof. Existence of *tset* such that

$$pathEXISTS(\langle tset\rangle, conf) \wedge toLABELSET(tset) = lset$$

follows from *lset* being a set of transitions which can be taken in the same step. Default transitions with empty triggers do not make existence of multiple maximal paths with the same set of explicit labels possible due to Prop. 6.1.27. Thus, uniqueness of the resulting path follows from the determinism of a statechart (Req. 1b). □

Due to the above proposition, we can consider *toLABELSET* and *toTRANSITIONSET* to be an inverse of one another and the same holds for *toLPATH* and *toPATH*. If path exists, *toPATH* is well-defined.

For statecharts within some state, forgetting about interlevel transitions and structure of compound states, *clfollowPATH* gives the state which can be entered by a given sequence of labels from the default connector of the considered state. *clpathEXISTS* checks for existence of a given *lpath* from a given state. The prefix 'c' in front of each considered function/predicate means 'local' within some OR-state, where we do not consider interlevel transitions and treat all substates as basic.

In a similar way to *pathEXISTS* we can define *lfollowPATH* and *lpathEXISTS* which give the entered state and whether the given *lpath* exists, for a flattened statechart.

Definition 6.2.13.

$$
\begin{array}{|l}
\mathit{clfollowPATH} : \mathit{LSeq} \times \mathit{STATE} \twoheadrightarrow\!\!\!\!\shortmid\;\; \mathit{STATE} \\
\mathit{clpathEXISTS}\ \_ : \mathbb{F}_1(\mathit{LSeq} \times \mathit{STATE}) \\
\mathit{lfollowPATH} : \mathit{LSeqSet} \times \mathit{SSet} \twoheadrightarrow\!\!\!\!\shortmid\;\; \mathit{SSet} \\
\mathit{lpathEXISTS}\ \_ : \mathbb{F}_1(\mathit{LSeqSet} \times \mathit{SSet}) \\
\hline
\forall\, \mathit{lpath} : \mathit{LSeq};\ \mathit{st} : \Sigma \bullet \\
\quad \mathit{lpath} = \langle\rangle \Rightarrow \mathit{clfollowPATH}(\mathit{lpath}, \mathit{st}) = \mathit{st} \wedge \\
\quad \mathit{lpath} \neq \langle\rangle \Rightarrow (\exists_1\, \mathit{trans} : \mathit{TRANSITION} \bullet \\
\qquad \mathit{trans.label} = \mathit{head}\ \mathit{lpath} \wedge \#\mathit{trans.source} = 1 \wedge \\
\qquad \mathit{FromSet}(\mathit{trans.source}) = \mathit{st} \wedge \\
\qquad (\exists_1\, \mathit{next} == \mathit{FromSet}(\mathit{trans.target}) \bullet \\
\qquad\quad \mathit{next} \in \rho(\mathit{parent}(\mathit{st})) \wedge \\
\qquad\quad \mathit{clfollowPATH}(\mathit{lpath}, \mathit{st}) = \mathit{clfollowPATH}(\mathit{tail}\ \mathit{lpath}, \mathit{next}))) \\
\forall\, \mathit{lpath} : \mathit{LSeq};\ \mathit{st} : \Sigma \bullet \\
\quad \mathit{clpathEXISTS}(\mathit{lpath}, \mathit{st}) \Leftrightarrow (\mathit{lpath}, \mathit{st}) \in \mathrm{dom}\ \mathit{clfollowPATH} \\
\forall\, \mathit{lpath} : \mathit{LSeqSet};\ \mathit{st} : \Sigma_f \bullet \\
\quad \mathit{lpath} = \langle\rangle \Rightarrow \mathit{lfollowPATH}(\mathit{lpath}, \mathit{st}) = \mathit{st} \wedge \\
\quad \mathit{lpath} \neq \langle\rangle \Rightarrow \\
\qquad (\exists_1\, \mathit{trans} : \Upsilon_f \bullet \mathit{trans.label} = \mathit{landALL}(\mathit{andTRUE}, \mathit{head}\ \mathit{lpath}) \wedge \\
\qquad\quad \mathit{lfollowPATH}(\mathit{lpath}, \mathit{st}) = \mathit{lfollowPATH}(\mathit{tail}\ \mathit{lpath}, \mathit{trans.target})) \\
\forall\, \mathit{lpath} : \mathit{LSeqSet};\ \mathit{st} : \Sigma_f \bullet \\
\quad \mathit{lpathEXISTS}(\mathit{lpath}, \mathit{st}) \Leftrightarrow (\mathit{lpath}, \mathit{st}) \in \mathrm{dom}\ \mathit{lfollowPATH}
\end{array}
$$

Theorem 6.2.14. *Different existing paths map to different ones and same ones map to the same ones if sets of transitions in all OR-states of a statechart are disjoint,*

$$
\begin{array}{l}
(\forall\, A, B : \Sigma \bullet T^{ni}(A) \cap T^{ni}(B) = \varnothing) \Rightarrow \\
\quad (\forall\, \mathit{path}_1, \mathit{path}_2 : \mathit{TSeqSet};\ \mathit{conf} : \mathbb{F}_1\,\Sigma \mid \mathit{configuration}(\mathit{root}, \mathit{conf}) \wedge \\
\qquad \mathit{pathEXISTS}(\mathit{path}_1, \mathit{conf}) \wedge \mathit{pathEXISTS}(\mathit{path}_2, \mathit{conf}) \bullet \\
\qquad\quad \mathit{toLPATH}(\mathit{path}_1) \neq \mathit{toLPATH}(\mathit{path}_2) \Leftrightarrow \\
\qquad\qquad \mathit{toLPATH}(\mathit{flattenPATH}(\mathit{path}_1)) \neq \\
\qquad\qquad\quad \mathit{toLPATH}(\mathit{flattenPATH}(\mathit{path}_2)))
\end{array}
$$

Proof. Consider the diagram in Fig. 6.16; we shall show that it commutes.

First of all, *(a)-(c)* are well-defined, by the determinism of a statechart, properties of *FULL_COMPOUND* and definition of transitions correspondingly. The correctness of exchanging *toLPATH* and *Unite* is explained below.

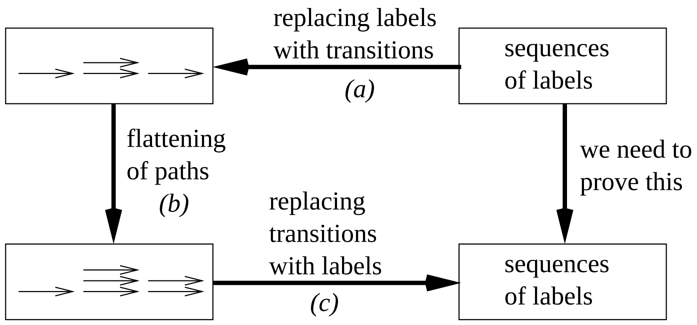


Figure 6.16: A commutative diagram for the proof that different paths map to different ones

Following Prop. 6.1.56, it is possible to express the behaviour of $FULL\_COMPOUND$ in the following way:

$$\begin{aligned}
&\forall\, tset : TSet \mid tsetVALID(tset) \bullet \\
&\quad \exists\, addition : TSet \bullet FULL\_COMPOUND(tset, nondetres) = \\
&\qquad tset \cup addition = Unite(\langle tset\rangle, \langle addition\rangle)\ (1)
\end{aligned}$$

Considering operations on paths, i.e. sequences of sets, we can write that:

$$\begin{aligned}
&\forall\, path : TSeqSet \mid \\
&\quad (\exists\, conf : \mathbb{F}_1\,\Sigma \bullet configuration(root, conf) \land pathEXISTS(path, conf)) \bullet \\
&\quad \exists\, cont : TSeqSet \bullet flattenPATH(path) = Unite(path, cont) \land \\
&\quad \#path = \#cont
\end{aligned}$$

and

$$\begin{aligned}
&toLPATH(Unite(path, cont)) \\
&\quad = apply(\{i : \mathsf{dom}\, path \bullet i \mapsto (path\ i \cup cont\ i)\}, toLABELSET) \\
&\quad = \{i : \mathsf{dom}\, path \bullet i \mapsto toLABELSET(path(i) \cup cont(i))\} \\
&\quad = \{i : \mathsf{dom}\, path \bullet i \mapsto toLABELSET(path(i)) \cup toLABELSET(cont(i))\} \\
&\quad = Unite(toLPATH(path), toLPATH(cont))
\end{aligned}$$

We can write $flattenPATH(path) = Unite(path, cont)$ since from Def. 6.2.10 *flattenPATH* applies $FULL\_COMPOUND$ pointwise to elements of the path and $FULL\_COMPOUND$ can be considered to provide an *addition* to every such element. *cont* can thus be constructed of such additions to every element.

Using the above result, the proof itself is rather simple. Consider an *lpath* in the main statechart (a set of labels involved being $labelset = \bigcup \mathsf{ran}\ lpath$), then

$$\begin{array}{l}
\forall\, lpath : LSeqSet;\ conf : \mathbb{F}_1\, STATE \mid configuration(root, conf) \land \\
\quad (lpath, conf) \in \mathrm{dom}\ toPATH \bullet \\
\quad\quad (\exists_1\, seq == toPATH(lpath, conf);\ cont : TSeqSet \bullet \\
\quad\quad toLPATH(flattenPATH(toPATH(lpath, conf))) \\
\quad\quad\quad = toLPATH(Unite(seq, cont)) \\
\quad\quad\quad = Unite(toLPATH(seq), toLPATH(cont)) \\
\quad\quad\quad = Unite(lpath, toLPATH(cont)))
\end{array}$$

where $\mathrm{ran}\ lpath \subseteq T^{ni}_{MAINSTATECHART}$ and $\mathrm{ran}\ toLPATH(cont) \cap T^{ni}_{MAINSTATECHART} = \varnothing$ from $T^{ni}(A) \cap T^{ni}(B) = \varnothing$. Consequently, we get that

$$Unite(lpath, toLPATH(cont)) \rhd T^{ni}_{labelset} = lpath$$

which concludes the proof. □

The above theorem only considers paths which exist; nonexisting ones are not misbehaving either, as follows.

Theorem 6.2.15. *An lpath exists in a flattened statechart iff it exists in the original one.*

$$\begin{array}{l}
(\forall\, A, B : \Sigma \bullet\ T^{ni}(A) \cap T^{ni}(B) = \varnothing) \Rightarrow \\
\quad (\forall\, lpath : LSeqSet;\ conf : \mathbb{F}_1\, \Sigma \mid configuration(root, conf) \bullet \\
\quad\quad (lpath, conf) \in \mathrm{dom}\ toPATH \Leftrightarrow \\
\quad\quad\quad lpathEXISTS(lpath, stateFLATTEN(root, conf)))
\end{array}$$

Proof. Follows from the comparison of definitions of $toPATH$ and $lfollowpath$. □

6.2.3 Restrictions on an implementation by Req. 4b

The considered requirement imposes certain specific constraint on possible errors in an implementation. Our handing of default transitions (the element **DE** of a test case basis), depends of this constraint.

Definition 6.2.16. *Req. 4b can be formalised as follows:*

$$\begin{array}{|l}
possibleImpl : TSet \twoheadrightarrow \mathbb{F}_1\, LSet \\
\hline
\forall\, tset : \mathbb{F}_1\, \Upsilon \bullet\ possibleImpl(tset) = \\
\quad \{trs : allFULLCOMPOUND(tset) \bullet \\
\quad\quad \{str : \mathbb{F}_1\, \Upsilon \mid str \subseteq trs \land (\forall\, t : trs \bullet \\
\quad\quad\quad \neg\, transitionDEFAULT(t) \Rightarrow t \in str) \bullet\ andALL(str)\} \\
\quad \}
\end{array}$$

Proposition 6.2.17. *If we trigger a full compound transition, only a part of which is implemented, this part will be triggered too.*

$$\begin{array}{l} \forall\, lset : LSet;\ m : DATA \bullet triggerSET(lset, m) \Rightarrow \\ \quad (\forall\, ls : LSet \mid ls \neq \varnothing \wedge ls \subseteq lset \bullet triggerSET(ls, m)) \end{array}$$

Proof. Follows from the definition of *triggerSET*. □

This allows us to construct a test set without consideration that only subsets of set of CTs comprising full compound transitions may be implemented.

6.3 Behaviour of statecharts and X-machines

Here we provide definitions of the behaviour of statecharts, of X-machines and relate the two. Specifically, we show that flat statecharts under our testing requirements are behaviourally equivalent to X-machines with almost the same transition diagram and the same labels on transitions.

6.3.1 Simple statecharts

Definition 6.3.1. *A statechart is* simple *if the root state is the only OR-state in it and it does not contain AND-states and has no default transitions (as the only one transition which could be default due to Req. 1g is not treated as such due to Def. 6.1.20).*

$$\begin{array}{l} \forall\, s : \Sigma \bullet (\phi(s) \neq stateBASIC \Rightarrow s = root) \wedge \\ \quad (\exists_1\, t : \Upsilon \bullet transitionDEFAULT(t)) \end{array}$$

Proposition 6.3.2. *For simple statecharts we have that*

$$\begin{array}{l} \forall\, s : \Sigma \setminus \{root\} \bullet \rho(s) = \varnothing \\ \rho(root) = \Sigma \setminus \{root\} \end{array}$$

Proof. Follows from Def. 6.3.1 and Def. 6.1.1. □

Proposition 6.3.3. *An expanded statechart is simple.*

Proof. Follows from Def. 6.2.4 and Req. 1g. □

Proposition 6.3.4. *Expanded simple statechart is essentially the same as the original one, the two related by an isomorphism as given below:*

$$\begin{array}{|l} statetoORIG : \mathbb{F}_1\ STATE \twoheadrightarrow\!\!\!\!\!\!\!\!\;\mapsto \mathbb{F}_1\ STATE \\ transitiontoORIG : TSet \twoheadrightarrow\!\!\!\!\!\!\!\!\;\mapsto TRANSITION \\ \hline \forall\, sset : \mathbb{F}_1\ \Sigma \bullet statetoORIG(sset) = \{FromSet(sset), root\} \\ \forall\, tset : TSet \bullet (\exists_1\, tr == FromSet(tset) \bullet \\ \quad transitiontoORIG(tset) = \langle\!|\ source == statetoORIG(tr.source), \\ \quad\quad target == statetoORIG(tr.target), label == tr.label\ |\!\rangle) \end{array}$$

Proof. From definition of *stateFLATTEN* (Def. 6.2.2), we get that it will preserve all states, i.e.

$$\forall\, conf : \mathbb{F}_1\ \Sigma \mid configuration(root, conf) \bullet stateFLATTEN(conf) = conf \setminus \{root\}$$

This shows that *statetoORIG* is a bijection and exhibits the same behaviour as *toCONFIGURATION*.

All transitions of a simple statechart are not default and have no default continuations, since all states below the root one are basic and the initial transition is not default by definition Def. 6.1.20. Since all transitions have the same priority, *FULL_COMPOUND* will preserve sets of labels of full compound transitions of the considered simple statechart. This implies that *transitiontoORIG* is a bijection.

From above, we get that *statetoORIG* and *transitiontoORIG* give an isomorphism of an original simple statechart and a flattened one. □

Behaviour of statecharts w.r.t step semantics described in Sect. 1.4.10 on p. 22. We give this semantics in terms of X-machines in Sect. 6.3.3 on p. 173.

6.3.2 X-machines

In this section we give the formal definition for stream X-machines from [HI98, Ipa95]. Only stream X-machines are considered in the thesis.

Definition 6.3.5 (Stream X-machine). *In the original notation developed for X-machines, a stream X-machine is a tuple* $\mathcal{M} = (\Sigma, \Gamma, Q, M, \Phi, F, I, T, m_0)$, *where:*

1. Σ *is the set of* inputs.
2. Γ *is the set of* outputs.
3. Q *is the finite set of* states.
4. M *is a possibly infinite set called* memory.

5. *Φ is a finite set of partial functions, used on transitions*

$$\phi : M \times \Sigma^* \to \Gamma^* \times M$$

   Each transition function removes the head of the input stream and adds an element to the rear of the output stream. Thus, a function cannot use information from the tail of the input stream or any of the output one.

6. *F is the next state partial function $F : Q \times \Phi \to Q$. It is often depicted as a state- transition diagram.*

7. *I and T are the sets of initial and termination states respectively, $I \subseteq Q$, $T \subseteq Q$. We assume that I is a singleton and $Q = T$.*

8. *$m_0 \in M$ is the initial memory value. This is the memory value what the machine starts operating with.*

A stream X-machine takes a symbol from the input and takes a transition if the current symbol and memory value satisfy any function from the current state. The output of the transition is appended to the output and memory modified in accordance to the what the function does. If no transition can be taken for a given input and memory value, an X-machine halts.

This notation while conventional in the X-machine world, contradicts our one for statecharts, by the choice of symbols. In the following we relate the two and will further use our statechart notation for X-machines.

Proposition 6.3.6. *The notation X-machines is in one-to-one correspondence with that for statecharts.*

Proof.

- Input and output sets Σ and Γ are equivalent to sets of changes $CSet$.
- The set of states Q can be described as a subset of $STATE$.
- Memory M corresponds to $DATA$ in statecharts.
- The set of functions Φ corresponds to labels on transitions of statecharts, $\{tr : \Upsilon \bullet tr.label\}$. Both are defined very similarly; in fact, the only difference is an order of arguments, in X-machines it is $(memory, input)$ and in statecharts — $(input, memory)$.
- The next-state function given by the state transition diagram can be unchanged between simple statecharts and X-machines.

- I (the set of initial states) is a singleton of the set of states in both stream X-machines and simple statecharts. From the definition of an X-machine (Def. 6.3.5) and testing requirements (Req. 1g) respectively.
- T (the set of terminal states) is equal to the set of states for both of the two (except for the *root* state).
- m_0 (initial memory value) is implicit in statecharts where it can be specified on the definition of the initial transition.

  For a given X-machine, a simple statechart can be constructed such that its data is initialised to m_0 on its initial transition. Conversely, for a simple statechart, we can construct an equivalent X-machine with initial memory value m_0 of it being the result of an action of its default transition.

□

Definition 6.3.7. *Behaviour of an X-machine can be defined following [HI98].*

```
── xmSTATUS ──────────────
s : Σ
m : DATA
──────────────────────────
```

Since transitions in the statechart notation are defined to take DATA and produce CSet, a few conversion functions between this representation and an X-machine one are needed.

$$
\begin{array}{|l}
len_{IO} : \mathbb{N} \\
emptySPACE : SPACE \\
\hline
len_{IO} \leq len_{SPACE} \\
\forall i : 1 \mathinner{..} len_{IO} \bullet \pi(i, emptySPACE) = \perp
\end{array}
$$

len_{IO} is the number of variables in the input tuple. SPACE is essentially an equivalent of X [Ipa95], a generalised type on which X-machine transitions can be defined to operate. It simply has input/output and memory parts together, i.e. variables with indices $1 \mathinner{..} len_{IO}$ are from input/output ones and the remaining $len_{IO} + 1 \mathinner{..} len_{SPACE}$ — represent memory. We consider memory to be of type DATA and input/output — of type CHANGE, both of which are potentially infinite. emptySPACE is described and used below.

$$
\begin{array}{|l}
xmtoX : SPACE \times SPACE \rightarrow SPACE \\
\hline
\forall IO : CHANGE;\ M : SPACE \bullet \\
\quad (\forall i : 1 \mathinner{..} len_{IO} \bullet \pi(i, xmtoX(IO, M)) = \pi(i, IO)) \wedge \\
\quad (\forall i : len_{IO} \mathinner{..} len_{SPACE} \bullet \pi(i, xmtoX(IO, M)) = \pi(i, M))
\end{array}
$$

xmtoX combines an input and data spaces into an X.

$$\begin{array}{|l}
InputtoX : CSet \times DATA \pfun SPACE \\
\hline
\forall m : DATA;\ chset : CSet \mid \\
\quad (\forall i : 1 \mathinner{\ldotp\ldotp} len_{IO} \bullet \#(\pi(\!|\{i\} \times chset|\!)) > 1) \bullet \\
\qquad InputtoX(chset, m) = xmtoX(modify(chset, emptySPACE), m)
\end{array}$$

An input to an X-machine is of type CHANGE with indices for non-$\perp$ elements of it ranging between 1 and len_{IO}. In relating X-machines to statecharts, it is more convenient to operate on a set of changes. Combining them together into a single tuple is done with $modify(chset, emptySPACE)$ where emptySPACE is the tuple corresponding to no changes to any variable[2].

After the machine has taken a transition, we need to obtain an output and new memory value. This is provided in a similar way to statecharts, i.e. using changes. XtoChanges splits an individual change to X into parts corresponding to output and changes to memory.

$$\begin{array}{|l}
XtoChanges : CHANGE \pfun (CHANGE \times CHANGE) \\
\hline
\forall X : CHANGE \bullet \\
\quad (\exists_1\, i : 1 \mathinner{\ldotp\ldotp} len_{SPACE} \bullet \pi(i, X) \neq \perp \wedge XtoChanges(X) = (xmtoX^{\sim})(\pi(i, X)))
\end{array}$$

where $xmtoX^{\sim}$ is a relational inverse of xmtoX.

The initial state and memory are given by the following schema:

$$\begin{array}{|l}
\hline
xmSTATUS\_Initial \\
\hline
init_s : \Sigma \\
init_m : DATA \\
\hline
\end{array}$$

It is then used to initialise an X-machine:

$$\begin{array}{|l}
\hline
xmINIT \\
\hline
\Delta xmSTATUS \\
xmSTATUS\_Initial \\
\hline
s' = init_s \wedge m' = init_m \\
\hline
\end{array}$$

[2] *emptySPACE* is not of type *CHANGE* since absence of changes is described with the set of changes being empty while individual elements of such a set have to provide those changes.

A transition taken by an X-machine implies performing the following operation:

$$
\begin{array}{|l}
\textit{xmTRANSITION} \\
\hline
\Delta \textit{xmSTATUS} \\
\textit{in}, \textit{in}', \textit{out}, \textit{out}' : \mathrm{seq}\, \textit{CSet} \\
\hline
\exists\, \textit{tr} : \Upsilon \bullet \\
\quad (\exists_1\, \textit{allchanges} == \textit{XtoChanges}(\!|\, \textit{tr}.\textit{label}(\textit{InputtoX}(\textit{head in}, m))\,|\!) \bullet \\
\quad (\exists_1\, \textit{output} == \{\textit{cht} : \textit{allchanges} \bullet \textit{first}(\textit{cht})\}; \\
\qquad \textit{chmemset} == \{\textit{cht} : \textit{allchanges} \bullet \textit{second}(\textit{cht})\} \bullet \\
\qquad \textit{last out}' = \textit{output} \land m' = \textit{modify}(\textit{chmemset}, m) \land \\
\quad \textit{in}' = \textit{tail in} \land \textit{front out}' = \textit{out} \land \textit{tr}.\textit{source} = \{s\} \land \textit{tr}.\textit{target} = \{s'\})) \\
\hline
\end{array}
$$

Behaviour of an X-machine can be expressed as

$$\textit{xmTRANSITION} \,⨟\, \textit{xmTRANSITION} \,⨟ \ldots ⨟\, \textit{xmTRANSITION} \,⨟\, \textit{xmINIT}$$

until sequence in becomes empty,

$$
\begin{array}{l}
\exists\, \textit{result} : \mathbb{F}[\textit{xmSTATUS};\ \textit{in}, \textit{in}', \textit{out}, \textit{out}' : \mathrm{seq}\, \textit{CSet} \mid \textit{true}] \bullet \\
\quad \textit{result} = ((\textit{xmINIT} \setminus \textit{xmSTATUS\_Initial}) ⨟ \\
\quad \textit{xmTRANSITION} ⨟ \ldots ⨟ \textit{xmTRANSITION} ⨟ \\
\quad [\textit{in}' : \mathrm{seq}\, \textit{CSet} \mid \textit{in}' = \langle\rangle]) \setminus (s', m')
\end{array}
$$

Definition 6.3.8. *An X-machine is complete and deterministic if for any state, any memory value and input symbol, there is one and only one function enabled.*

$$
\begin{array}{l}
\forall\, s : \Sigma;\ \textit{in} : \textit{CSet};\ m : \textit{DATA} \bullet \\
\quad (\exists_1\, \textit{tr} : \Upsilon \bullet \textit{InputtoX}(\textit{in}, m) \in \mathsf{dom}\, \textit{tr}.\textit{label})
\end{array}
$$

6.3.3 Behaviour of statecharts

Here we provide the formalisation of the behaviour of statecharts, informal description of which was given in Sect. 1.4.10 on p. 22. The description somewhat follows [Bog97].

Since there are two types of semantics to consider, we show how a simple statechart can comply with the synchronous one and then introduce the second machine, which serves as an intermediary between the environment and the first machine. This second one implements the asynchronous step semantics. The described testing method under our testing assumptions would then test the first machine; the second one is always assumed correct. Synchronous step semantics is implemented by labels of transitions. The second machine and the communication between the two are shown in Fig. 6.17 in the notation similar to that of Sect. 5.2.7 on p. 93.

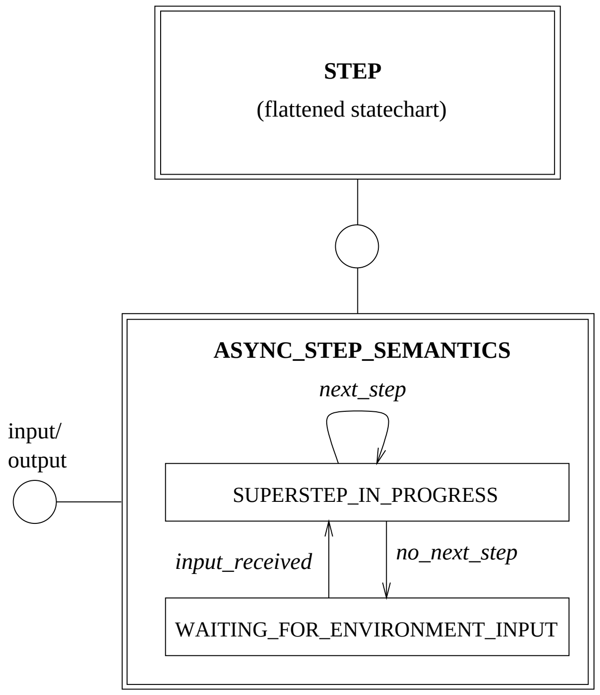


Figure 6.17: Behaviour of statecharts expressed using X-machines

A statechart has input, output ports, holds data and a configuration, in addition to a state-transition structure given in Def. 6.1.1.

stSTATUS

$inport, outport : \mathbb{F}_1 \mathbb{N}$
$data : DATA$
$conf : \Sigma_f$

inport and *outport* are the sets of indices of variables from the global data space of a statechart, appearing in input and output ports, respectively. They are non-empty as otherwise we either cannot supply a statechart with inputs or observe outputs. In either case, testing of such a system (as defined in Sect. 1.1.4 on p. 5) is impossible.

The initial values of variables of a statechart are defined as

$$\begin{array}{|l}
\textit{stSTATUS\_Initial} \\
\hline
\textit{init\_data} : \textit{DATA} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\textit{stINIT} \\
\hline
\Delta \textit{stSTATUS} \\
\textit{stSTATUS\_Initial} \\
\hline
\exists\, \textit{nondetres} : \textit{STATE} \twoheadrightarrow\!\!\!\!\!\!\!\!+\;\; \textit{TRANSITION};\ \textit{init\_tr} : \Upsilon \bullet \\
\quad (\exists_1\, \textit{init\_fct} == \textit{toTRANSITION}_D(\\
\qquad \textit{FULL\_COMPOUND}(\{\textit{init\_tr}\}, \textit{nondetres})) \bullet \\
\qquad \textit{enable}(\textit{init\_fct}, \textit{init\_data}, \textit{defaultEntranceComplete}(\textit{root}, \{\textit{root}\})) \wedge \\
\qquad \textit{data} = \textit{modify}(\textit{init\_fct}.\textit{label}(\textit{init\_data}), \textit{init\_data}) \wedge \\
\qquad \textit{conf} = \textit{stateFLATTEN}(\textit{root}, \textit{init\_fct}.\textit{target})) \\
\hline
\end{array}$$

This gives the initial configuration and memory of a statechart. We assume that a full compound transition to the initial configuration is always enabled and any *init_data* : *DATA* is in its domain; as such full compound transition is unique, we get well-definedness of the result of this initialisation schema.

According to the semantics of statecharts, events hold their value for one step only. This can be described with

$$\begin{array}{|l}
\textit{ev\_index} : \mathbb{F}\,\mathbb{N} \\
\hline
\pi(\!|\, \textit{ev\_index} \times \textit{CHANGE} \,|\!) = \pi(\!|\, \textit{ev\_index} \times \textit{DATA} \,|\!)
\end{array}$$

Above, *ev_index* is a set of indices of event variables in the space of a statechart. For events, sets of changes and values coincide.

Definition 6.3.9. *event_discard takes the set of changes, and data; its behaviour is similar to modify, apart from that event_discard removes events which were not generated. It is supposed to be applied to a set of changes generated by transitions in a step.*

$$\begin{array}{|l}
\textit{event\_discard} : \textit{CSet} \times \textit{DATA} \rightarrow \textit{DATA} \\
\hline
\forall\, \textit{chset} : \textit{CSet};\ m : \textit{DATA} \bullet \\
\quad (\forall\, i : 1 \mathinner{..} \textit{len}_{\textit{SPACE}} \bullet \\
\qquad i \in (1 \mathinner{..} \textit{len}_{\textit{SPACE}}) \setminus \textit{ev\_index} \cup \textit{index}(\!|\, \textit{chset} \,|\!) \Rightarrow \\
\qquad\quad \pi(i, \textit{event\_discard}(\textit{chset}, m)) = \pi(i, \textit{modify}(\textit{chset}, m)) \wedge \\
\qquad i \notin (1 \mathinner{..} \textit{len}_{\textit{SPACE}}) \setminus \textit{ev\_index} \cup \textit{index}(\!|\, \textit{chset} \,|\!) \Rightarrow \\
\qquad\quad \pi(i, \textit{event\_discard}(\textit{chset}, m)) = \perp)
\end{array}$$

event_discard does not affect changes which it was supplied with; only events which were not generated in the previous step are removed. Consequently, its behaviour does not affect racing. Output-distinguishability is not affected according to Prop. 6.6.3.

Theorem 6.3.10. *If a statechart is deterministic (Req. 1b) and required to behave synchronously (Req. 3c, Req. 1f), an X-machine behaviourally equivalent to the considered statechart can be constructed.*

Proof. Consider a statechart; determinism of it implies well-definedness of the result of the *STEP* function and thus of the labels of the X-machine.

Comparing the definition of *STEP* with the *xmTRANSITION* schema in Def. 6.3.7, we can see that *STEP* can be used as a function in an X-machine on a transition between states *t.source* and *t.target*. Note that it differs from the label of a statechart such that for a statechart label *label*, the function of an X-machine will be $event\_discard(label(m), m)$. Consequently, is possible to construct an X-machine corresponding to the considered statechart which has the same transition diagram but labels are replaced with the result of their composition with the *event_discard* function.

As for input and output behaviour of the constructed X-machine, we introduce the *xmTRANSITION* (Def. 6.3.7)-equivalent schema of it as follows:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\text{— } stSTEP \text{ —}} \\
\Delta(data, conf)stSTATUS \\
in, in', out, out' : \mathrm{seq}\ CSet \\
\hline
\exists_1\, t : \Upsilon_f \bullet (\exists_1\, newdata == modify(filter(head\ in, inport), data) \bullet \\
\quad enable(t, newdata, conf) \land conf' = t.target \land \\
\quad (\exists_1\, changes == (execTRANS(t)\ newdata) \bullet \\
\quad last\ out' = filter(changes, outport) \land \\
\quad data' = event\_discard(changes, newdata) \land \\
\quad in' = tail\ in \land front\ out' = out)) \\
\hline
\end{array}
$$

In the above, we write $conf' = t.target$ since transitions of the flattened statechart are defined such as to go from a configuration to a configuration (Def. 6.2.4).

With this, we have an X-machine behaviourally equivalent to the simple statechart considered by construction.

Note that synchronicity (Req. 3c) is used in the X-machine construction since we put a single $execTRANS(t)$ on a transition rather than a composition of them corresponding to a superstep. □

For complex statecharts, for construction of an X-machine we can use flattened statecharts introduced in Sect. 6.2 on p. 157 due to Prop. 6.2.9 and

Prop. 6.3.3.

To say that again, the top machine in Fig. 6.17 expresses what happens during a step; the bottom one feeds it with changes from the environment and then waits until it has finished processing them. The communication mechanism is described in Sect. 5.2.7 on p. 93. For a simple statechart the top machine in the figure is the same as the transition diagram of that statechart; complex statecharts have their diagrams flattened.

The initial state of the *ASYNC_STEP_SEMANTICS* machine is the *WAITING_FOR_ENVIRONMENT_INPUT* one. Communication between it and the *STEP* machine is handled by the *COMMUNICATE* function.

$$\begin{array}{|l}
COMMUNICATE : CSet \rightarrowtail\!\!\!\!\!\!+ \; CSet \\
STEP, PERFORM\_STEP : CSet \nrightarrow CSet \\
\hline
PERFORM\_STEP = COMMUNICATE \mathbin{⨾} STEP \mathbin{⨾} COMMUNICATE
\end{array}$$

where *STEP* expresses the behaviour of the *STEP* machine and *COMMUNICATE* models the communication channel.

Transitions of the machine representing step semantics are given as follows (all three are communicating in terms of [BGG+99]):

$$\begin{array}{|l|}
\hline
input\_received \\
\hline
env_{in}, changes' : CSet \\
\hline
\mathsf{df}\ env_{in} \land changes' = PERFORM\_STEP(env_{in}) \\
\hline
\end{array}$$

$$\begin{array}{|l|}
\hline
step \\
\hline
changes, changes' : CSet \\
\hline
changes \in \mathsf{dom}\ PERFORM\_STEP \land \\
changes' = PERFORM\_STEP(changes) \\
\hline
\end{array}$$

$$\begin{array}{|l|}
\hline
no\_next\_step \\
\hline
changes : CSet \\
\hline
changes \notin \mathsf{dom}\ PERFORM\_STEP \\
\hline
\end{array}$$

we assume that the *STEP* machine can notify us if changes are within its domain or not.

During testing, the tester communicates to the *STEP* machine bypassing the *ASYNC_STEP_SEMANTICS* one as described in Sect. 5.2.7 on p. 93. If testing does not reveal faults, we can guarantee correct behaviour of the *STEP* machine and of the whole system, assuming correct implementation of *ASYNC_STEP_SEMANTICS* and *COMMUNICATE*.

6.4 Proofs of the merging rules without refinement

Here we provide formal definitions of the sets used in test case generation introduced in Chap. 3 on p. 46 and the merging rules for them. We begin with definitions of sets used in Chap. 2 on p. 28 and then show how they can be generalised for complex statecharts. Proofs for merging rules are then provided under assumption that default transitions are non-interlevel, Req. 4h.

6.4.1 TCB for a substate of an OR state

Here we define sets comprising TCB for a substate of an OR-state, considering all its states basic. Such a substate statechart is an extension of a simple statechart in that it may have multiple default transitions and interlevel transitions. Since, due to the consideration of all substates being basic, such statecharts have no concurrency in them, an element of every sequence of labels in the corresponding Φ, C or W is a single label rather than sets of labels.

We define *StateCoverElement* to contain a state and a label-path leading to that state. This simplifies merging for C. Definitions follow from those in [HI98].

Definition 6.4.1.

$$
\begin{array}{|l}
\mathit{StateCoverElement} == [\mathit{lpath} : \mathit{LSeq};\ \mathit{state} : \mathit{STATE}] \\
\mathit{cStateCover}\ \_ : \mathbb{F}_1(\mathit{STATE} \times \mathbb{F}[\mathit{lpath} : \mathit{LSeq};\ \mathit{state} : \mathit{STATE}]) \\
\hline
\forall\, C : \mathbb{F}\,\mathit{StateCoverElement};\ \mathit{st} : \Sigma \bullet \\
\quad \mathit{cStateCover}(\mathit{st}, C) \Leftrightarrow \\
\quad \phi(\mathit{st}) \neq \mathit{stateOR} \land C = \varnothing \lor \\
\quad \phi(\mathit{st}) = \mathit{stateOR} \land (\forall\, S : \rho(\mathit{st}) \bullet \\
\quad (\exists\, \mathit{element} : C \bullet \mathit{element.state} = S \land \mathit{clfollowPATH}(\\
\qquad \mathit{element.lpath}, \mathit{defaultFROM}(\mathit{st})) = S))
\end{array}
$$

Definition 6.4.2.

$$
\begin{array}{|l}
\mathit{cCharacterisationSet}\ \_ : \mathbb{F}_1(\mathit{STATE} \times \mathit{LSetSeq}) \\
\hline
\forall\, W : \mathit{LSetSeq};\ st : \Sigma \bullet \\
\qquad \mathit{cCharacterisationSet}(st, W) \Leftrightarrow \\
\qquad \phi(st) = \mathit{stateBASIC} \Leftrightarrow W = \varnothing \vee \\
\qquad \phi(st) = \mathit{stateOR} \wedge \\
\qquad \#\{s : \rho(st) \mid \phi(s) \neq \mathit{connectorDEFAULT}\} = 1 \wedge W = \{\langle\rangle\} \vee \\
\qquad \#\{s : \rho(st) \mid \phi(s) \neq \mathit{connectorDEFAULT}\} > 1 \wedge \\
\qquad\qquad (\forall\, S_1, S_2 : \rho(st) \bullet (\exists\, \mathit{lpath} : W \bullet \\
\qquad\qquad\qquad (\mathit{clpathEXISTS}(\mathit{lpath}, S_1) \wedge \neg\ \mathit{clpathEXISTS}(\mathit{lpath}, S_2)) \vee \\
\qquad\qquad\qquad (\mathit{clpathEXISTS}(\mathit{lpath}, S_2) \wedge \neg\ \mathit{clpathEXISTS}(\mathit{lpath}, S_1))))
\end{array}
$$

Since transitions without triggers cannot be used to distinguish states, we include $\{\langle\rangle\}$ in W for OR-states with only one substate such that when constructing a set of test cases we get nonempty result. This $\{\langle\rangle\}$ has to be removed during the merging process provided the resulting W is going to contain non-empty sequences of labels.

For a state st, the *defaultTransitionLabels* function gives all possible default entrances into st and is recursively defined as follows:

Definition 6.4.3.

$$
\begin{array}{|l}
\mathit{defaultTransitionLabels} : \mathit{STATE} \twoheadrightarrow \mathit{LSetSet} \\
\hline
\forall\, st : \Sigma \bullet \\
\qquad (\phi(st) = \mathit{stateBASIC} \Rightarrow \mathit{defaultTransitionLabels}(st) = \{\varnothing\}) \wedge \\
\qquad (\phi(st) = \mathit{stateOR} \Rightarrow \mathit{defaultTransitionLabels}(st) = \\
\qquad\qquad \bigcup\{tr : TR^{ni}(st) \mid \mathit{transitionDEFAULT}(tr) \bullet \\
\qquad\qquad\qquad \mathit{Composition}(\mathit{setMULT}, \{ \\
\qquad\qquad\qquad\qquad (\textbf{if}\ tr.\mathit{label} = \mathit{andTRUE}\ \textbf{then}\{\{\varnothing\}\}\ \textbf{else}\{\{tr.\mathit{label}\}\}), \\
\qquad\qquad\qquad\qquad \mathit{defaultTransitionLabels}(\mathit{FromSet}(tr.\mathit{target})) \\
\qquad\qquad\qquad \}) \\
\qquad\qquad \}) \wedge \\
\qquad (\phi(st) = \mathit{stateAND} \Rightarrow \mathit{defaultTransitionLabels}(st) = \\
\qquad\qquad \mathit{Composition}(\mathit{setMULT}, \{s : \rho(st) \bullet \mathit{defaultTransitionLabels}(s)\}))
\end{array}
$$

The meaning of the *Composition* function is illustrated by the Fig. 6.18. Dashed lines represent two out of many paths connecting {}s. Every path must take a single *LSet* from every *LSetSet*; sets on a path get united by *setMULT* during the execution of *Composition*.

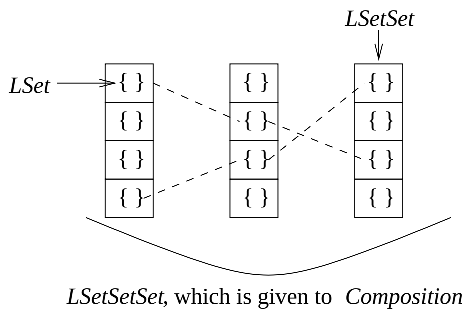


Figure 6.18: An illustration of the *Composition* function

Putting the above definitions together, the whole TCB for a statechart has to satisfy the following:

Definition 6.4.4.

$$
\begin{array}{|l}
C : STATE \twoheadrightarrow\!\!\!\!\!\shortmid\;\; \mathbb{F}\, StateCoverElement \\
W : STATE \twoheadrightarrow\!\!\!\!\!\shortmid\;\; LSetSeq \\
\Phi, \boldsymbol{DE} : STATE \twoheadrightarrow\!\!\!\!\!\shortmid\;\; LSetSet \\
\hline
\forall\, st : \Sigma \bullet \\
\quad cStateCover(st, C(st)) \land \\
\quad cCharacterisationSet(st, W(st)) \land \\
\quad \Phi(st) = \{l : T(st) \bullet \{l\}\} \land \\
\quad \boldsymbol{DE}(st) = defaultTransitionLabels(st)
\end{array}
$$

Note that when viewed as functions, $C(st)$, $W(st)$, $\Phi(st)$ are not well-defined, i.e. we could have different sets all qualifying for a valid TCB. In the merging rules we assume that for every state we decided what the TCB is before merging.

Proposition 6.4.5. *For all states s and every element of DE(s), we have a unique continuation transition TSet (consisting of individual transitions).*

$$\forall\, st : \Sigma \bullet (\exists\, bijection : LSet \rightarrowtail\!\!\!\!\!\shortmid\;\; TSet \bullet \boldsymbol{DE}(st) = \mathsf{dom}\, bijection)$$

Proof. We need to show that exclusion of transitions with *andTRUE* labels from *DE* in Def. 6.4.3 does not introduce ambiguity and thus *bijection* is a function. From construction of **DE** it follows that labels of transitions selected correspond to a valid path from some transition entering state *st* and continuing from the default connector. Prop. 6.2.12 gives us the desired result.

The $bijection^{\sim}$ (the relational inverse of *bijection*) is a function since *toLABELSET* is defined for every *TSet*. This completes the proof. □

Definition 6.4.6. *Minimality of statecharts (Req. 1a) can be asserted as follows:*

$$\begin{array}{l}
\forall\, s : \Sigma \mid \phi(s) = \mathit{stateOR} \bullet \\
\quad \exists\, \mathit{lpath} : \mathit{LSeq} \;\bullet\; \mathit{clfollowPATH}(\mathit{lpath}, \mathit{defaultFROM}(s)) = s \\
\forall\, s : \Sigma \mid \phi(s) = \mathit{stateOR} \bullet \\
\quad \forall\, s_1, s_2 : \rho(s) \mid s_1 \neq s_2 \bullet \\
\qquad \exists\, \mathit{lpath} : \mathit{LSeq} \bullet \\
\qquad\quad (\mathit{clpathEXISTS}(\mathit{lpath}, s_1) \wedge \neg\, \mathit{clpathEXISTS}(\mathit{lpath}, s_2)) \vee \\
\qquad\quad (\neg\, \mathit{clpathEXISTS}(\mathit{lpath}, s_1) \wedge \mathit{clpathEXISTS}(\mathit{lpath}, s_2))
\end{array}$$

It corresponds to the minimality of an associated automaton in [Ipa95].

For a minimal substate statechart for some state, we can construct a C set to visit all states and a W set to distinguish all of them. In both cases this is possible without usage of interlevel transitions.

Proposition 6.4.7. *A simple statechart is minimal iff C and W exist for it.*

Proof. Existence of C and W for a minimal statechart follows from comparison of definitions of C, W (Def. 6.4.1, Def. 6.4.2) and minimality of a statechart (Def. 6.4.6).

Conversely, if C exists, we can visit all states and thus all of them are reachable. Existence of W ensures that no states have the same behaviour. These two imply minimality. □

6.4.2 TCB for a flattened statechart

Here we define sets comprising TCB for a flattened statechart. This essentially involves replacement of *LSeq* by *LSeqSet*, Υ by $\mathbb{F}\,\Upsilon$ in the TCB for a substate of an OR state and removal of "c" in front of names.

Definition 6.4.8.

$$
\begin{array}{|l}
StateCover\ \_ : \mathbb{F}_1(LSetSeqSet) \\
\hline
\forall\, C : LSetSeqSet \bullet \\
\quad StateCover(C) \Leftrightarrow \\
\quad \Sigma_f = \varnothing \wedge C = \varnothing \vee \\
\quad \Sigma_f \neq \varnothing \wedge (\forall\, S : \Sigma_f \bullet \\
\quad (\exists\, element : C \bullet lfollowPATH(element, \\
\qquad stateFLATTEN(root, defaultEntranceComplete(root, \varnothing))) = S))
\end{array}
$$

Definition 6.4.9.

$$
\begin{array}{|l}
CharacterisationSet\ \_ : \mathbb{F}_1(LSetSeqSet) \\
\hline
\forall\, W : LSetSeqSet \bullet \\
\quad CharacterisationSet(W) \Leftrightarrow \\
\quad \#\Sigma_f = 0 \wedge W = \varnothing \vee \\
\quad \#\Sigma_f = 1 \wedge W = \{\langle\{andTRUE\}\rangle\} \vee \\
\quad \#\Sigma_f > 1 \wedge (\forall\, S_1, S_2 : \Sigma_f \bullet \\
\qquad \exists\, lpath : W \bullet \\
\qquad\quad (lpathEXISTS(lpath, S_1) \Rightarrow \neg\, lpathEXISTS(lpath, S_2)) \wedge \\
\qquad\quad (lpathEXISTS(lpath, S_2) \Rightarrow \neg\, lpathEXISTS(lpath, S_1)))
\end{array}
$$

Proposition 6.4.10. *If we consider a simple statechart, the above definitions are almost identical to those for the test case basis (TCBs for flattened and original statecharts are related by SetSeqtoSetSeqSet[LABEL] which is a bijection in our case).*

Proof. Prop. 6.3.4 states that expanded simple statechart is almost identical to the original one. The conclusion of the theorem follows from comparison of the definitions of *StateCover* and *CharacterisationSet* with *cStateCover* and *cCharacterisationSet* above. □

6.4.3 Merging rules

We begin with the outline of the whole proof. As stated earlier, here we consider the case with no refinement; refinement is dealt with in Sect. 6.5 on p. 195.

Definition 6.4.11.

$$
\begin{array}{|l}
\mathit{completeTransition} : \mathit{TRANSITION} \twoheadrightarrow \mathit{LSet} \\
\hline
\forall\, \mathit{trans} : \Upsilon \bullet (\exists\, \mathit{lset} : \boldsymbol{DE}(\mathit{FromSet}(\mathit{trans.target})) \bullet \\
\qquad \mathit{completeTransition}(\mathit{trans}) = \{\mathit{trans.label}\} \cup \mathit{lset})
\end{array}
$$

$$
\begin{array}{|l}
\mathit{completePath} : \mathit{LSeq} \times \mathit{STATE} \twoheadrightarrow \mathit{LSeqSet} \\
\hline
\forall\, \mathit{lseq} : \mathit{LSeq};\ \mathit{state} : \Sigma \bullet \\
\qquad \mathit{lseq} = \langle\rangle \Rightarrow \mathit{completePath}(\mathit{lseq}, \mathit{state}) = \langle\rangle \wedge \\
\qquad \mathit{lseq} \neq \langle\rangle \Rightarrow (\exists\, \mathit{tr} : \Upsilon \bullet \\
\qquad\qquad \mathit{tr.source} = \{\mathit{state}\} \wedge \mathit{transitionNI}(\mathit{tr}) \wedge \\
\qquad\qquad\qquad \mathit{completePath}(\mathit{lseq}, \mathit{state}) = \langle \mathit{completeTransition}(\mathit{tr})\rangle ^\frown \\
\qquad\qquad\qquad\quad \mathit{completePath}(\mathit{tail}\ \mathit{lseq}, \mathit{FromSet}(\mathit{tr.target})))
\end{array}
$$

$$
\begin{array}{|l}
\mathit{multC} : \mathit{LSetSeqSet} \times \mathit{LSetSeqSet} \twoheadrightarrow \mathit{LSetSeqSet} \\
\hline
\forall\, \mathit{lsetseqset}_1, \mathit{lsetseqset}_2 : \mathit{LSetSeqSet} \bullet \\
\qquad \mathit{multC}(\mathit{lsetseqset}_1, \mathit{lsetseqset}_2) = \\
\qquad\qquad \{\mathit{lseqset}_1 : \mathit{lsetseqset}_1;\ \mathit{lseqset}_2 : \mathit{lsetseqset}_2 \bullet \\
\qquad\qquad\qquad \mathit{Unite}(\mathit{lseqset}_1, \mathit{lseqset}_2)\}
\end{array}
$$

$$
\begin{array}{|l}
C^{merged} : \mathit{STATE} \twoheadrightarrow \mathit{LSetSeqSet} \\
\hline
\forall\, \mathit{st} : \Sigma \bullet \\
\qquad (\phi(\mathit{st}) = \mathit{stateBASIC} \Rightarrow C^{merged}(\mathit{st}) = \{\langle\varnothing\rangle\}) \wedge \\
\qquad (\phi(\mathit{st}) = \mathit{stateOR} \Rightarrow C^{merged}(\mathit{st}) = \\
\qquad\qquad \bigcup\{\mathit{coverel} : C(\mathit{st}) \bullet \mathit{multOR1}(\\
\qquad\qquad\qquad \{\mathit{completePath}(\mathit{front}\ \mathit{coverel.lpath}, \mathit{defaultFROM}(\mathit{st})) \cup \\
\qquad\qquad\qquad \{\#\mathit{coverel.lpath} \mapsto \{\mathit{last}\ \mathit{coverel.lpath}\}\}\}, \\
\qquad\qquad\qquad C^{merged}(\mathit{coverel.state}))\}) \wedge \\
\qquad (\phi(\mathit{st}) = \mathit{stateAND} \Rightarrow C^{merged}(\mathit{st}) = \\
\qquad\qquad \mathit{Composition}(\mathit{multC}, \\
\qquad\qquad\qquad \{s : \rho(\mathit{st}) \bullet \{\mathit{cel} : C(s) \bullet \mathit{SeqtoSeqSet}(\mathit{cel.lpath})\}\}))
\end{array}
$$

In the above definition, we have $C^{merged}(st) = \{\langle\varnothing\rangle\}$ for basic states, since for all non-basic ones, we have a nonempty path to substates; when this path is merged with $C^{merged}(st)$, an $\varnothing$ disappears. A statechart always contains at least one non-basic state — the root one. Usage of $\mathit{FromSet}(tr.target)$ in the *completeTransition* function is justified by our usage of only non-interlevel transitions in C. The expression

$$\begin{array}{l} \mathit{Composition}(\mathit{multC}, \\ \qquad \{s : \rho(st) \bullet \{\mathit{cel} : C(s) \bullet \mathit{SeqtoSeqSet}(\mathit{cel.lpath})\}\}) \end{array}$$

corresponds to usage of *multAND* to multiply *cel.lpath* as

$$\mathit{multAND}(\{\mathit{SeqtoSeqSet}\ \mathit{lpath}_1\}, \mathit{multAND}(\{\mathit{SeqtoSeqSet}\ \mathit{lpath}_2\} \ldots))$$

Every path in C is made to consist of full compound transitions using the *completePath* and *multOR*1 functions. The *completePath* function makes a path given to it consist of FCTs. The last element of *coverel.lpath* is made full compound by virtue of *multOR*1 uniting it with default transitions in *coverel.state*, contained in $C^{merged}(\mathit{coverel.state})$.

completePath function uses *completeTransition* which picks any valid default completion for a state, entered by a given transition. Only transitions from C are supplied to *completeTransition* which are non-interlevel by construction of C.

We define Φ^{merged} for merging without refinement (multiplication of transitions) and that with the weak one (union of transitions) Φ^{merged}_{union}:

Definition 6.4.12.

$$\begin{array}{|l} \Phi^{merged}, \Phi^{merged}_{union} : \Sigma \twoheadrightarrow \mathit{LSetSet} \\ \hline \forall\, st : \Sigma \bullet \\ \qquad \phi(st) = \mathit{stateBASIC} \Rightarrow \Phi^{merged}(st) = \varnothing \wedge \Phi^{merged}_{union}(st) = \varnothing \wedge \\ \qquad \phi(st) = \mathit{stateOR} \Rightarrow \\ \qquad\qquad (\Phi^{merged}(st) = \Phi(st) \cup \bigcup\{s : \rho(st) \bullet \Phi^{merged}(s)\} \wedge \\ \qquad\qquad \Phi^{merged}_{union}(st) = \Phi(st) \cup \bigcup\{s : \rho(st) \bullet \Phi^{merged}_{union}(s)\}) \wedge \\ \qquad \phi(st) = \mathit{stateAND} \Rightarrow \\ \qquad\qquad (\Phi^{merged}(st) = \mathit{Composition}(\mathit{setMULT}, \\ \qquad\qquad\qquad \{s : \rho(st) \bullet \{\varnothing\} \cup \Phi^{merged}(s)\}) \setminus \{\varnothing\} \wedge \\ \qquad\qquad \Phi^{merged}_{union}(st) = \bigcup\{s : \rho(st) \bullet \Phi^{merged}_{union}(s)\}) \end{array}$$

Above, we are dealing with sets of sets of labels rather than with sets of sequences of sets as we do in W^{merged}. For this reason, we use $\{\varnothing\}$ instead of $\{\langle\rangle\}$ in *Composition* above.

Note that $\forall\, st : \Sigma;\ \phi : \Phi^{merged}_{union}(st) \bullet \#\phi = 1$.

Definition 6.4.13.

$$W^{merged} : \Sigma \mathrel{+\!\!\!\!\twoheadrightarrow} LSetSeq$$

$$\begin{array}{l}
\forall\, st : \Sigma \bullet (\exists_1\, union == W(st) \cup \bigcup\{s : \rho(st) \bullet W^{merged}(s)\} \bullet \\
\quad (\#union > 1 \lor union \neq \{\langle\rangle\} \Rightarrow W^{merged}(st) = union \setminus \{\langle\rangle\}) \land \\
\quad (\#union = 1 \land union = \{\langle\rangle\} \Rightarrow W^{merged}(st) = union))
\end{array}$$

Completion of compound transitions in Φ^{merged} and W^{merged} to full compound ones is done using the *defaultComplete* function, in turn using the *transitionDefaultComplete* function to complete every individual set of transitions.

Definition 6.4.14.

$$convtoTRANSITIONSET : LSetSet \mathrel{+\!\!\!\!\twoheadrightarrow} TSetSet$$

$$\begin{array}{l}
\forall\, lsetset : LSetSet \bullet \\
\quad convtoTRANSITIONSET(lsetset) = \{lset : lsetset \bullet \\
\qquad \{l : lset;\ tr : \Upsilon \mid \\
\qquad\quad transitionDEFAULT(tr) \land scope(tr) = getSCOPE(l) \bullet tr\} \\
\quad \}
\end{array}$$

$$allFCTComplete : LABEL \mathrel{+\!\!\!\!\twoheadrightarrow} TSetSet$$

$$\begin{array}{l}
\forall\, l : LABEL \bullet allFCTComplete(l) = \\
\quad \bigcup\{tr : \Upsilon \mid \neg\, transitionDEFAULT(tr) \land tr.label = l \bullet \\
\qquad Composition(setMULT, \{s : tr.target \bullet \\
\qquad\quad convtoTRANSITIONSET(\boldsymbol{DE}(s))\})\}
\end{array}$$

$$transitionDefaultComplete : LABEL \mathrel{+\!\!\!\!\twoheadrightarrow} LSetSet$$

$$\begin{array}{l}
\forall\, l : LABEL \bullet transitionDefaultComplete(l) = \\
\quad \{tset : allFCTComplete(l) \bullet \{t : tset \mid t.label \neq andTRUE \bullet t.label\}\}
\end{array}$$

$$defaultComplete : LSetSeq \mathrel{+\!\!\!\!\twoheadrightarrow} LSetSeqSet$$

$$\begin{array}{l}
\forall\, lsetseq : LSetSeq \bullet defaultComplete(lsetseq) = \\
\quad \bigcup\{lseq : lsetseq \bullet \\
\qquad \{resseq : LSeqSet \mid (\forall\, i : 1 \mathinner{..} \#lseq \bullet \\
\qquad\quad resseq\ i \in apply(lseq, transitionDefaultComplete)\ i)\} \\
\quad \}
\end{array}$$

---

convtoTRANSITIONSET converts labels of **DE** into corresponding transitions, by picking those with the appropriate label. From determinism of a statechart, only one default transition may exist with a given label in a state. Note that default connectors without labels (i.e. with label *andTRUE*) are ignored in both Def. 6.4.14 and **DE** which is possible due to Prop. 6.4.5.

The *allFCTComplete* takes all compound transitions labelled with the given label ($tr : \Upsilon \mid tr.label = l$) and returns a set of all default possible completions, for each such transition. *Composition* is necessary for interlevel transitions entering AND-states where in each of the concurrent components there could be multiple default completions. $transitionDefaultComplete(l)$ takes a label l and computes the unified label. As a result of this computation, we get a set of labels which can trigger all possible full compound transitions containing an initial transition with a non-empty label l. Note that no default transition completing the given one may also have this label due to the absence of shared transitions. An initial transition in the same statechart with that label does not cause a problem as well because FCTs entering the statechart and those starting within it will have different labels.

defaultComplete takes every sequence *lseq* of a given *lsetseq* and looks at all possible full compound transitions which could exist in a statechart starting from every its element $l : \mathsf{ran}\, lseq$. Taking one element from each of them, we get a possible path in the statechart which has its full compound transitions begin with initial transitions of *lset*. A set of all such paths makes a completion of a sequence and a union of such paths for all $lseq : lseqset$ is returned by *defaultComplete*. We also have that

Proposition 6.4.15.
$\forall\, lpathset_1, lpathset_2 : LSetSeq \bullet defaultComplete(lpathset_1 \cup lpathset_2) =$
$\quad defaultComplete(lpathset_1) \cup defaultComplete(lpathset_2)$

Proof. Follows from Def. 6.4.14. □

Definition 6.4.16.

$$
\begin{array}{|l}
expandedPhi : LSetSet \pfun LSetSet \\
W^{merged}_{final}, \Phi^{merged}_{final}, \Phi^{merged}_{union,final} : \Sigma \pfun LSetSeqSet \\
\hline
\forall st : \Sigma \bullet W^{merged}_{final}(st) = defaultComplete(W^{merged}(st)) \\
\forall lsetset : LSetSet \bullet expandedPhi(lsetset) = \\
\quad \bigcup\{lset : lsetset \bullet Composition(setMULT, \\
\quad\quad \{l : lset \bullet transitionDefaultComplete(l)\}) \\
\quad \} \\
\forall st : \Sigma \bullet \\
\quad \Phi^{merged}_{final}(st) = \{lset : expandedPhi(\Phi^{merged}(st)) \bullet \langle lset \rangle\} \wedge \\
\quad \Phi^{merged}_{union,final}(st) = \{lset : expandedPhi(\Phi^{merged}_{union}(st)) \bullet \langle lset \rangle\}
\end{array}
$$

The expression

$$Composition(setMULT, \{l : lset \bullet transitionDefaultComplete(l)\})$$

does the default transition expansion on every initial transition in *lset* and set-multiplies results. Instead of doing this, we could apply *transitionDefaultComplete* to every transition in $\Phi(st)$ at the stage of construction of Φ^{merged}.

6.4.4 Proofs for the merging rules

Proposition 6.4.17. *Sequences of labels multiplied by multAND in Φ merging for AND-states, correspond to transitions which can be taken in the same step.*

Proof. From Def. 6.4.12 and Prop. 6.1.36, we get that

$$
\begin{array}{l}
\forall s : \Sigma \bullet (\forall lbl : \Phi^{merged}(s) \bullet \\
\quad (\forall tr : \Upsilon \mid tr.label \in lbl \bullet scope(tr) \in \rho^*(s)))
\end{array}
$$

For this reason, transitions *multAND*ed together in the part of Def. 6.4.12, corresponding to AND-states, satisfy *orthscope* and by Th. 6.1.31 they can be taken in the same step. □

The above proposition can be extended to all labels generated by $\Phi^{merged}(root)$ as follows.

Proposition 6.4.18. *All transitions with labels from $\Phi^{merged}(root)$ are not conflicting.*

$$\begin{array}{l}\forall\, tset : TSet \mid (\forall\, tr : tset \bullet \neg\, transitionDEFAULT(tr)) \wedge \\ \quad \{tr : tset \bullet tr.label\} \in \Phi^{merged}(root) \bullet \\ \quad orthscope(tset)\end{array}$$

Proof. From construction of Φ for a state, it contains individual compound transitions in that state (rather groups of them). As a result, conflicting transitions are never included in Φ. Merging rules for Φ either merge sets or multiply them, implying that conflicting transitions may emerge only due to multiplication which forms groups of transitions to be taken in the same step. Prop. 6.4.17 shows this not to be the case. Consequently, all transitions which carry sets of labels belonging to $\Phi^{merged}(root)$, can be taken in the same step. □

As a consequence of the above proposition and Prop. 6.1.64, we get that all transitions triggered during test application are non-conflicting.

In both merging rules and definition of Υ_f, we expand transitions to make them full compound. The following proposition shows that the two different approaches to expansion, introduced for convenience, are equivalent.

Proposition 6.4.19. *allFULLCOMPOUND is essentially the same as allFCTComplete,*

$$\begin{array}{l}\forall\, l : LABEL \bullet \bigcup\{tr : \Upsilon \mid \\ \quad \neg\, transitionDEFAULT(tr) \wedge tr.label = l \bullet \\ \qquad allFULLCOMPOUND(\{tr\})\} = allFCTComplete(l)\end{array}$$

Proof. The proof consists of the five steps.

1. From construction of **DE** where we choose all possible non-interlevel default transitions from every OR-state, we get that

$$\begin{array}{l}\{nd : \Sigma \twoheadrightarrow \Upsilon \mid \\ \quad (\forall\, s : \Sigma \mid \phi(s) = stateOR \bullet (\exists_1\, ndl == (nd\ s).label \bullet \\ \qquad (ndl \neq andTRUE \vee (\exists\, lset : \mathbf{DE}(s) \bullet ndl \in lset)) \wedge \\ \qquad transitionNI(nd\ s) \wedge transitionDEFAULT(nd\ s)))\} \\ = \{ndr : \Sigma \twoheadrightarrow \Upsilon \mid (\forall\, s : \Sigma \mid \phi(s) = stateOR \bullet \\ \quad scope(ndr(s)) \in \rho(s) \wedge transitionDEFAULT(ndr(s)))\}\end{array}$$

The statement

$$(ndl \neq andTRUE \lor (\exists\, lset : \mathbf{DE}(s) \bullet ndl \in lset))$$

follows from usage of *Composition* in Def. 6.4.3 for OR states. Since in every state labels of default transitions are all different, from statement $tr : TR^{ni}(st) \mid transitionDEFAULT(tr)$ in the definition of *defaultTransitionLabels*, it follows that all default transitions are considered and none of others. Hence our first set above is equal to the second one.

This proves that **DE** essentially contains all possible *nondetresolution* functions (the second set above is the one used in the definition of *allFULLCOMPOUND*).

2. From the definition of *defaultEntranceComplete*, we get that

$$\begin{aligned}&\forall\, tr : \Upsilon \bullet defaultEntranceComplete(root, tr.target) = \\ &\qquad \bigcup\{s : tr.target \bullet route(root, defaultFROM(s))\}\end{aligned}$$

For every target state s of transition tr, we can identify default completions. Each of them, denoted *top*, leaves the $defaultFROM(s)$ of an appropriate s, from which it follows that all these *top* are related to tr by *continuationDEF*.

3. Consider the following function

$$\begin{array}{|l}defaultTransitions : STATE \twoheadrightarrow TSetSet \\ \hline \forall\, st : \Sigma \bullet \\ \quad (\phi(st) = stateBASIC \Rightarrow defaultTransitions(st) = \{\varnothing\}) \land \\ \quad (\phi(st) = stateOR \Rightarrow defaultTransitions(st) = \\ \qquad \bigcup\{tr : TR^{ni}(st) \mid transitionDEFAULT(tr) \bullet \\ \qquad\quad \{tset : defaultTransitions(FromSet(tr.target)) \bullet \{tr\} \cup tset\}\}) \land \\ \quad (\phi(st) = stateAND \Rightarrow defaultTransitions(st) = \\ \qquad Composition(setMULT, \{s : \rho(st) \bullet defaultTransitions(s)\}))\end{array}$$

This function differs from *defaultTransitionLabels* (Def. 6.4.3), i.e. that of **DE**, only by usage of $\{tr\}$ in the case of $\phi(st) = stateOR$ rather than $\{tr.label\}$.

From construction of $defaultTransitions(st)$ it follows that

$$\begin{array}{l}
\forall\, tset : defaultTransitions(st) \bullet \exists_1\, top : \Upsilon \bullet scope(top) = st \land \\
\quad \neg\, (\exists\, t : tset \setminus \{top\} \bullet continuationDEF(t, top)) \land \\
\quad (\forall\, tr_c : tset \bullet transitionNI(tr_c) \land transitionDEFAULT(tr_c) \land \\
\quad (\exists_1\, tr_s : tset \setminus \{tr_c\} \bullet continuationDEF(tr_s, tr_c))) \land \\
\quad \neg\, (\exists\, tr_s : tset;\ tr_c : \Upsilon \setminus \bigcup(defaultTransitions\ st) \bullet \\
\qquad continuationDEF(tr_s, tr_c))
\end{array}$$

The existence of such *top* transition follows from consideration of OR-states by *defaultTransitions*, since all further transitions are also default non-interlevel and lower-level to it. Consequently, **DE** produces labels of transitions related by *continuationDEF* as required by *FULL_COMPOUND*.

4. This shows that

$$\begin{array}{l}
\forall\, tr : \Upsilon;\ nondetresolution : \Sigma \twoheadrightarrow \Upsilon \bullet \\
\quad (\exists\, cont : \Sigma \twoheadrightarrow TSet \bullet (\forall\, s : tr.target \bullet \\
\qquad toLABELSET(cont(s)) \in \mathbf{DE}(s)) \land \\
\qquad (\forall\, s : \Sigma \bullet nondetresolution(s) \in cont(s) \land \\
\qquad FULL\_COMPOUND(\{tr\}, nondetresolution) = \\
\qquad\quad \{tr\} \cup \bigcup\{s : tr.target \bullet cont(s)\}))
\end{array}$$

Existence of *cont* such that

$$\begin{array}{l}
toLABELSET(cont(s)) \in \mathbf{DE}(s) \land \\
\quad (\forall\, s : \Sigma \bullet nondetresolution(s) \in cont(s))
\end{array}$$

follows from the fact that **DE** essentially contains all possible *nondetresolution* functions. We can express *FULL_COMPOUND* in the described way due to the result proven in the above two steps of the proof.

Note that for a transition *tr* where $s_1 \in tr.target \land s_2 \in tr.target \land orth(s_1, s_2)$, we have $cont(s_1) \cap cont(s_2) = \varnothing$ because transitions related by *continuationDEF* are not orthogonal.

5. The above step shows that *allFCTComplete* generates sets of labels for results of *FULL_COMPOUND* for all possible *nondetresulution* functions. This proves the conclusion of the theorem.

□

We can use expansion to construct Φ^{merged}_{final} and W^{merged}_{final} given in Def. 6.4.16 due to the following proposition:

Proposition 6.4.20. *A transition with initial CT lbl exists iff any of the expanded lpaths generated by transitionDefaultComplete(lbl) exist.*

$$\begin{array}{l}
\forall\, conf : \mathbb{F}_1\,\Sigma;\ lbl : LABEL \mid \\
\quad configuration(root, conf) \wedge defaultEntranceComplete(root, conf) = conf \bullet \\
\qquad (\{lbl\}, conf) \in \mathsf{dom}\ toTRANSITIONSET \Leftrightarrow \\
\qquad (\exists\, lset : transitionDefaultComplete(lbl) \bullet \\
\qquad\quad lpathEXISTS(\langle lset\rangle, stateFLATTEN(root, conf)))
\end{array}$$

Proof. This essentially follows from the proof of Prop. 6.4.19 as follows.

For a label *lbl*, *transitionDefaultComplete* constructs the set of all possible continuations of it. This means that if a transition with the *lbl* label exists from a considered configuration, it will have a continuation and this continuation will be included in at least one element of *transitionDefaultComplete*(*lbl*).

In case it does not, no transition with labels from *transitionDefaultComplete*(*lbl*) may exist since it would have to begin with the considered initial transition *lbl* and no transition with such label exists from the considered configuration by our assumption. □

Nondeterminism after expansion of transitions cannot occur due to requirement of deterministic behaviour of an implementation.

Proposition 6.4.21. *Any set of labels of non-default transitions related by orthscope is contained in* $\Phi^{merged}(root)$.

Proof. First of all, we note that when labels *lset* correspond to some transitions related by *orthscope*, a set of states produced by *getSCOPE*(*lset*) (Prop. 6.1.36) has every pair of states in it orthogonal (i.e. related by *orth*). Further in the proof we operate with this set of scopes, which we call *group*.

Consider a state $top = lca(group)$. As all states in *group* are scope states, $\forall\, s : group \bullet \phi(s) = stateOR$. If $top \in group$, then $\forall\, s : group \mid s \neq top \bullet top = lca(top, s)$ which contradicts that states *top* and *s* are orthogonal because *orth* requires $\phi(lca(top, s))$ to be an AND-state. Non-orthogonality of *top* and *s* in turn contradicts *orthscope*-property of *group*. We thus have $top \notin group$.

From the proof of Prop. 6.1.10, we have that there are states $s_1, s_2, \ldots, s_p : \rho(top)$ such that $\forall\, s : group \bullet (\exists\, s_i : \rho(top) \bullet s \in \rho^*(s_i))$. From definition of *lca* for *top*, it follows that there is more than one such $s_i : \rho(top)$.

Let us partition states in *group* into gr_i, such that all states in gr_i are under the corresponding s_i, $\forall\, s : gr_i \bullet s \in \rho^*(s_i)$. We can build a tree of states with the root of it being an AND-state *top*. Note that taking a single state from each gr_i, we get a set of states for which *top* is an *lca* (this follows from the fact that *top* is an ancestor and s_i prevent any lower state to be such).

Partitions gr_i are disjoint since they are contained in $\rho^*(s_i)$, for different states s_i.

Now we consider every partition gr_i individually. In case it is not a singleton, every pair of states in it is orthogonal, from the definition of *orthscope*. We can then construct a node $top_1 = lca(gr_i)$ and follow the partitioning described above for states in gr_i. Doing this for all gr_i gives us a set of nodes for our tree, each corresponding to gr_i. These nodes are the lower-level ones to *top*. For a singleton gr_i, we have a leaf node $FromSet(gr_i)$ under *top*.

The process of identifying a top state and partitioning can be repeated with smaller partitions. Finally (and this will occur since partitions are smaller in size than sets which were split), we arrive at a tree with leaf nodes being our scope states and non-leaf nodes of the tree being AND-states.

The definition of merging for Φ, Def. 6.4.12 unites sets of labels for OR-states and multiplies those for AND-ones. In the latter case, the result always contains original sets, due to the inclusion of $\{\varnothing\}$ in the multiplication. Following the tree constructed from bottom to top, we can see from Def. 6.4.12 that Φ^{merged} will contain the set of labels *lset*. □

Proposition 6.4.22 (Merging rule for Φ). *Merging rules generate TCB which is valid. For an expanded statechart Υ_f is essentially*

$$flattenPATH(toTRANSITIONSET(lpath))$$

The equality between the two is not, strictly speaking, correct since toTRANSITIONSET takes a configuration as a parameter. Consequently, the result to prove is somewhat more complicated:

$$\begin{aligned}
&\Upsilon_f = \{lseqset : LSeqSet;\ st : \Sigma_f \mid \\
&\quad lseqset \in \Phi^{merged}_{final}(root) \land lpathEXISTS(lseqset, st) \bullet \\
&\quad toTRANSITION_D(flattenPATH(\\
&\qquad \langle toTRANSITIONSET(lseqset(1), toCONFIGURATION(st))\rangle)(1))\} \\
&\forall\, lseqset : \Phi^{merged}_{final}(root) \bullet (\exists\, st : \Sigma_f \bullet lpathEXISTS(lseqset, st))
\end{aligned}$$

Proof. We need to show that labels of all full compound transitions are captured by the Φ^{merged}_{final}, and no more than that is in there.

First of all, we observe that *flattenPATH* when applied to a sequence consisting of a single element is the same as *FULL_COMPOUND* and thus we have to prove that *lset*s from $\Phi^{merged}_{final}(root)$, expanded with

$$FULL\_COMPOUND(toTRANSITIONSET(lset, toCONFIGURATION(st)))$$

generate all full compound transitions of the statechart.

From proof of Prop. 6.4.19, we get that for a set of non-default transitions

$$tset : TSet \mid \{tr : tset \bullet tr.label\} \in \Phi^{merged}(root),$$

$\Phi^{merged}_{final}(root)$ will contain labels of all possible default continuations. From determinism of the statechart, we get that for any such label and any *nondetresolution* function, $FULL\_COMPOUND(tset)$ would return the same set. From Prop. 6.4.18 and Prop. 6.4.19 we have that such a *tset* should be returned by *toFCT* and thus corresponding transitions will be included in Υ_f. Since *FULL_COMPOUND* generates valid transitions (Prop. 6.1.58) and as such transitions in Υ_f are valid, we get that

$$\exists\, st : \Sigma_f \bullet pathEXISTS(tset, toCONFIGURATION(st))$$

From a transition $tr : \Upsilon$ we can extract initial transitions of it due to Prop. 6.1.60. These transitions will be *orthscope* since they can be taken in the same step. Taking labels of them and applying Prop. 6.4.21, we get that this set will be constructed by the result of merging of Φ. The full compound transition *tr* will then have labels belong to Φ^{merged}_{final} due to Prop. 6.4.19. $\square$

Proposition 6.4.23 (Merging rule for C).
$StateCover(C^{merged}(root))$

Proof. Consider a lowest-level OR state *st*, which only contains basic substates. $C^{merged}(st) = C(st)$ (from Def. 6.4.11) gives paths to every substate of it, from the default connector. Such paths would also consist of full compound transitions (if considered within the state) as basic states have no default transitions. Each path in $C(st)$ will begin with a default transition.

A higher-level state $sp = parent(st)$ would have $C(sp)$ visit every state, including *st*. A path visiting this state would be multiplied with $C^{merged}(st)$, such that paths entering every substate statechart of *st* from the default connector of *sp* are formed and can be represented in the form (from Def. 6.4.11 and Def. 6.1.17)

$$\begin{array}{l}
\forall\, st : \Sigma;\ path_{sp} : LSeqSet;\ Cst : LSetSeqSet \mid \\
\quad \phi(st) = stateOR \land Cst = \{cel : C(st) \bullet SeqtoSeqSet(cel.lpath)\} \bullet \\
\quad multOR1(\{path_{sp}\}, Cst) = \{path_{st} : Cst \bullet \\
\qquad front\ path_{sp} \frown \langle last\ path_{sp} \cup head\ path_{st} \rangle \frown tail\ path_{st}\}
\end{array}$$

These paths would consist of full compound transitions. The part $tail\ path_{st}$ has all transitions full compound as shown above. $\langle last\ path_{sp} \cup head\ path_{st} \rangle$ is an FCT too as it contains a transition entering an OR-state and a default transition within that state, entering a basic state. This proves that transitions entering OR- or basic states in sp or st get expanded to full compound as a result of merging. The same can be said for all other OR-substates of sp.

From the definition of $completePath(coverel.path, defaultFROM(st))$ we get that $front\ path_{sp}$ consists of full compound transitions too. Consequently, for OR-states C^{merged} consists of full compound transitions and enters every state.

From construction of C^{merged} for AND-states and a result just shown, we get that all combinations of states are entered by full compound transitions. As a result, C^{merged} produces a result complying with definition of the $StateCover$ (Def. 6.4.8). □

Proposition 6.4.24 (Merging rule for W).
$CharacterisationSet(W^{merged}_{final}(root))$

Proof. For every $S_1, S_2 : \Sigma_f \mid S_1 \neq S_2$, we can construct $conf_1$, $conf_2$ using $toCONFIGURATION$. We then have $root \in conf_1 \cap conf_2$. Consider a non-basic state $s : conf_1 \cap conf_2$ such that $\rho(s) \cap conf_1 \cap conf_2 = \varnothing$, then this s is an OR-state (otherwise $\rho(s) \subseteq conf_1 \cap conf_2$ from the definition of configuration) and

$$\exists\, s_1, s_2 : \rho(s) \bullet s_1 \neq s_2 \wedge s_1 \in conf_1 \wedge s_2 \in conf_2.$$

If there is no such s, $conf_1 = conf_2$ (again from Def. 6.1.3) and thus $S_1 = S_2$, which contradicts our choice of S_1 and S_2. Having shown that there exists a state s with properties above, we may note that there could be more than one such state, for instance, when in some AND-state substates of multiple concurrent states are different between $conf_1$ and $conf_2$.

The characterisation set W_s for the state s can distinguish between s_1 and s_2 and is included in W^{merged} by construction of it. Th. 6.2.14 and Th. 6.2.15 generalise this to W_s distinguishing between $conf_1$ and $conf_2$; the rest follows from Prop. 6.4.20. □

Proposition 6.4.25. *If for every state, its substate statechart is minimal, then flattened statechart will be minimal too.*

Proof. From Prop. 6.4.7 follows the existence of C and W for substate statechart in every state. From Prop. 6.4.23 and Prop. 6.4.24 we get their existence for the flattened statechart and thus using Prop. 6.4.7, arrive at the conclusion of the proposition. □

Theorem 6.4.26 (Merging rules). *Merging rules work.*

Proof. Proven in Prop. 6.4.22, 6.4.23, 6.4.24. □

The theorem we just proven is the main result of the proofs: it shows that merging rules described in Chap. 3 on p. 46 allow provable correctness of an implementation of a system w.r.t its design under assumptions provided in Chap. 5 on p. 79.

6.5 Refinement of statecharts

In this section we provide proofs of the merging rules and test case construction for the three types of refinements, refinement of OR-states, weak and strong refinement of AND-states.

6.5.1 Refinement of OR-states

In this subsection we describe a state refinement of OR states. In the process of refinement, we add a statechart inside a state of a design and make appropriate modification to an implementation.

As described in Sect. 1.4.5 on p. 16, transitions have priorities associated with them such that a transition between the higher-level states has a precedence over a transition between lower states in the state hierarchy. Such priorities are related to the scope of a transition, i.e. the *lcoa* of the union between the source and target states of the transition (Def. 6.1.24). Due to priorities, transitions entering the refined state enter a default connector of it and those leaving take priority over those inside the state. This allows us to eliminate testing of transitions exiting the state, from all its substates.

The refinement considered is subject to restrictions that a design and an implementation of the main statechart is expected to contain no transitions with labels of transitions of a substate statechart and the same for the substate statechart for labels of the main statechart. Interlevel transitions between the two are not allowed either.

Definition 6.5.1. *A statechart in state state (further often referred to as a substate statechart) is a state refinement of a statechart main statechart if an implementation of the whole system satisfies the following property:*

$$
\begin{aligned}
&\forall\, state : \Sigma \bullet \\
&\quad (\exists_1\, lblsub == toLABELSET(\{tr : \Upsilon \mid scope(tr) \in \rho^*(state)\}) \bullet \\
&\qquad (\forall\, tr : \Upsilon \bullet tr.label \in lblsub \Leftrightarrow tr.source \cap \rho^+(state) \neq \varnothing)) \wedge \\
&\quad (\forall\, tr : \Upsilon \bullet ((tr.source \cup tr.target) \cap \rho^+(state)) \neq \varnothing \Leftrightarrow \\
&\qquad tr.source \cup tr.target \subseteq \rho^+(state))
\end{aligned}
$$

In addition, we require that there are no transitions labelled with labels of the substate statechart lblsub from extra states of the main statechart, i.e. states accounted for in $m - n$ during test case generation.

Theorem 6.5.2. *The set of test cases*

$$
\begin{aligned}
T \;=\;& C_{\text{MAIN STATECHART}} * (\{1\} \cup \Phi_{\text{MAIN STATECHART}} \cup \Phi^2_{\text{MAIN STATECHART}} \cup \\
&\qquad \cup \ldots \cup \Phi^{m_{\text{MAIN STATECHART}} - n_{\text{MAIN STATECHART}} + 1}_{\text{MAIN STATECHART}}) * W_{\text{MAIN STATECHART}} \\
\cup& \\
&\{path\ to\ state\} *_1 C_{\text{SUBSTATE STATECHART}} * (\{1\} \cup \Phi_{\text{SUBSTATE STATECHART}} \cup \\
&\qquad \Phi^2_{\text{SUBSTATE STATECHART}} \cup \ldots \cup \\
&\qquad \Phi^{m_{\text{SUBSTATE STATECHART}} - n_{\text{SUBSTATE STATECHART}} + 1}_{\text{SUBSTATE STATECHART}}) * W_{\text{SUBSTATE STATECHART}}
\end{aligned}
$$

(an example of which is given in Eqn. 3.6 on p. 54) is adequate for provable correctness of an implementation to a design as a result of testing not revealing faults.

Proof. Using multiplication operators rather than functions for clarity, we have for the set of test cases,

$$
\begin{aligned}
T \;=\;& C * (\{1\} \cup \Phi \cup \Phi^2 \cup \ldots \cup \Phi^{m-n+1}) * W \\
=\;& (C_{MAIN\,STATECHART} \cup \{\text{path to } state\} *_1 C_{SUBSTATE\,STATECHART}) * \\
&*(\{1\} \cup (\Phi_{MAIN\,STATECHART} \cup \Phi_{SUBSTATE\,STATECHART}) \cup \\
&\qquad \cup \ldots \cup (\Phi_{MAIN\,STATECHART} \cup \Phi_{SUBSTATE\,STATECHART})^{m-n+1}) * \\
&\qquad\quad *(W_{MAIN\,STATECHART} \cup W_{SUBSTATE\,STATECHART})
\end{aligned}
$$

Where $C_{MAIN\,STATECHART}$ visits all states of the main statechart and $C_{SUBSTATE\,STATECHART}$ — all of the substate statechart.

Using Def. 6.5.1, we have that

$$
\begin{aligned}
&C_{MAIN\,STATECHART} * \Phi_{SUBSTATE\,STATECHART} = \\
&\qquad \{\text{path to some initial configuration of } state\} * \Phi_{SUBSTATE\,STATECHART}
\end{aligned}
$$

since transitions with labels of substate statechart do not exist from any state other than those in $\rho^*(\mathit{state})$ by Req. 1e (Def. 6.1.35). The resulting set is contained in

$$\{\text{path to } \mathit{state}\} *_1 C_{SUBSTATE\,STATECHART} * \Phi_{SUBSTATE\,STATECHART}$$

As there are no transitions of the main statechart having any specific states of the *state* statechart as their source state, every transition leaving a configuration of *state* would leave any other configuration of it. Due to Def. 6.5.1, $\{\text{path to } \mathit{state}\} *_1 C_{SUBSTATE\,STATECHART} * \Phi_{MAIN\,STATECHART}$ can be reduced to $\{\text{path to } \mathit{state}\} *_1 C_{MAIN\,STATECHART} * \Phi_{MAIN\,STATECHART}$.

For sets of transitions

$$\begin{aligned}&\Phi_{MAIN\,STATECHART} * \Phi_{SUBSTATE\,STATECHART} = \\ &\quad \{\text{initial configuration of } \mathit{state}\} * \Phi_{SUBSTATE\,STATECHART}\end{aligned}$$

where initial configuration of *state* is the configuration entered by *state* for transitions terminating at the border of it. This follows from the fact that multiplication only yields pairs of transitions which could be followed if the first one enters the substate statechart, i.e. a loopback or a transition entering an extra state of main statechart, having a transition system with labels of substate statechart in it. The latter is made impossible by Def. 6.5.1; from the fact that the initial configuration has to be included in $C_{SUBSTATE\,STATECHART}$, we get that $\Phi_{MAIN\,STATECHART} * \Phi_{SUBSTATE\,STATECHART}$ is contained in $C_{SUBSTATE\,STATECHART} * \Phi_{SUBSTATE\,STATECHART}$.

$\Phi_{SUBSTATE\,STATECHART} * \Phi_{MAIN\,STATECHART}$ is contained in $\Phi_{MAIN\,STATECHART} * \Phi_{MAIN\,STATECHART}$ for similar reasons to those given in description of $C_{MAIN\,STATECHART} * \Phi_{MAIN\,STATECHART}$.

For sets W we get similar results to those of Φ.

The above has shown that any multiplications of elements of the merged test case basis from different statecharts is contained in or can be reduced to that for one or another statechart. Consequently, we can discard all such multiplications without impact on fault distinguishing ability of the set of test cases, giving the following:

$$\begin{aligned}T \;=\;& C_{MAIN\,STATECHART} * (\{1\} \cup \Phi_{MAIN\,STATECHART} \cup \Phi^2_{MAIN\,STATECHART} \cup \\ &\quad \cup \ldots \cup \Phi^{m-n+1}_{MAIN\,STATECHART}) * W_{MAIN\,STATECHART} \\ \cup& \\ & \{\text{path to } \mathit{state}\} *_1 C_{SUBSTATE\,STATECHART} * \\ & *(\{1\} \cup \Phi_{SUBSTATE\,STATECHART} \cup \Phi^2_{SUBSTATE\,STATECHART} \cup \\ &\quad \cup \ldots \cup \Phi^{m-n+1}_{SUBSTATE\,STATECHART}) * W_{SUBSTATE\,STATECHART}\end{aligned}$$

Consider sequences of transitions from $\Phi_{MAIN\,STATECHART}$ which will check for extra states in main statechart. Due to Def. 6.5.1 and Req. 1e, their verification ability will not be affected by existence of extra states in the substate statechart. In the same way, $\Phi_{SUBSTATE\,STATECHART}$ will verify extra states in substate statechart unhindered by those of main statechart, which follows from the clause of Def. 6.5.1 concerning extra states of main statechart. The just proven fact allows us to separate testing for the two statecharts even more: we can assume different numbers of extra states. This completes the proof. □

6.5.2 Weak refinement of AND-states

With weak refinement of AND-states we assume that transitions taken in concurrent states in the same step will enter the same configuration and exhibit the same behaviour as those taken sequentially, in any order.

We begin with the definition of a helper function which is an extension of *confENTERED*, operating on sequences of sets of transitions rather than on single transitions. It returns the entered configuration following the sequence from a given configuration. Another helper function is called *labelCOMPUTED*, which computes a label corresponding to a sequence of transitions. Both are defined below:

Definition 6.5.3. *Auxiliary functions confENTEREDSEQ and labelCOMPUTED.*

$$
\begin{array}{|l}
\mathit{confENTEREDSEQ} : \mathit{TSeq} \times \mathbb{F}_1\,\Sigma \times (\mathit{STATE} \twoheadrightarrow\!\!\!\!\!+ \ \mathit{TRANSITION}) \twoheadrightarrow\!\!\!\!\!+ \ \mathbb{F}_1\,\Sigma \\
\hline
\forall\, \mathit{tseq} : \mathit{TSeq};\ \mathit{conf} : \mathbb{F}_1\,\Sigma;\ \mathit{nondetres} : \mathit{STATE} \twoheadrightarrow\!\!\!\!\!+ \ \mathit{TRANSITION} \mid \\
\quad \mathit{configuration}(\mathit{root}, \mathit{conf}) \bullet \\
\quad \mathit{tseq} = \langle\rangle \Rightarrow \mathit{confENTEREDSEQ}(\mathit{tseq}, \mathit{conf}, \mathit{nondetres}) = \mathit{conf} \land \\
\quad \mathit{tseq} \neq \langle\rangle \Rightarrow \\
\qquad \mathit{confENTEREDSEQ}(\mathit{tseq}, \mathit{conf}, \mathit{nondetres}) = \\
\qquad\quad \mathit{confENTEREDSEQ}(\mathit{tail}\ \mathit{tseq}, \\
\qquad\quad \mathit{confENTERED}(\mathit{toTRANSITION}_D(\\
\qquad\quad \mathit{FULL\_COMPOUND}(\{\mathit{head}\ \mathit{tseq}\}, \mathit{nondetres})), \mathit{conf}), \mathit{nondetres})
\end{array}
$$

$$
\begin{array}{|l}
\mathit{labelCOMPUTED} : \mathit{TSeq} \times \mathit{DATA} \times (\mathit{STATE} \twoheadrightarrow\!\!\!\!\!+ \ \mathit{TRANSITION}) \twoheadrightarrow\!\!\!\!\!+ \ \mathit{CSet} \\
\hline
\forall\, \mathit{tseq} : \mathit{TSeq};\ m : \mathit{DATA};\ \mathit{nondetres} : \mathit{STATE} \twoheadrightarrow\!\!\!\!\!+ \ \mathit{TRANSITION} \bullet \\
\quad \mathit{tseq} = \varnothing \Rightarrow \mathit{labelCOMPUTED}(\mathit{tseq}, m, \mathit{nondetres}) = \varnothing \land \\
\quad \mathit{tseq} \neq \varnothing \Rightarrow \mathit{labelCOMPUTED}(\mathit{tseq}, m, \mathit{nondetres}) = \\
\qquad (\mathit{toTRANSITION}_D(\mathit{FULL\_COMPOUND}(\{\mathit{head}\ \mathit{tseq}\}, \mathit{nondetres}))).\mathit{label}\ m \cup \\
\qquad\quad \mathit{labelCOMPUTED}(\mathit{tail}\ \mathit{tseq}, m, \mathit{nondetres})
\end{array}
$$

The above definition of labelCOMPUTED does not guarantee identical behaviour of transitions when taken in the same step and in different ones; due to semantics of events, behaviour will almost always differ. labelCOMPUTED is not necessary for the weak AND-state refinement which focuses on transition diagram only and was included here only to make obvious 'oddities' of behaviour impossible such as a tape recorder exploding if user presses both $\underline{\text{rew}}$ *and* $\underline{\text{play}}$ *buttons at the same time.*

Definition 6.5.4. *For an AND-state (further referred to as state), a weak refinement means that for an implementation considered*

$$
\begin{array}{l}
\forall\, tset : TSet;\ state : \Sigma;\ conf : \mathbb{F}_1\,\Sigma \mid configuration(root, conf) \land state \in conf \land \\
\quad (\forall\, tr : tset \bullet scope(tr) \in \rho^*(state) \land \neg\, transitionDEFAULT(tr)) \land orthscope(tset) \bullet \\
\quad (\exists\, tseq : TSeq \mid \\
\qquad \mathsf{ran}\, tseq = tset \land \#tseq = \#tset \land orthscope(\mathsf{ran}\, tseq) \bullet \\
\quad \forall\, nondetres : STATE \twoheadrightarrow TRANSITION;\ m : DATA \mid \\
\qquad enable^{design}(toTRANSITION_D(\\
\qquad\qquad FULL\_COMPOUND(tset, nondetres)), m, conf) \land \\
\qquad enable(toTRANSITION_D(\\
\qquad\qquad FULL\_COMPOUND(tset, nondetres)), m, conf) \bullet \\
\qquad\qquad confENTERED(toTRANSITION_D(\\
\qquad\qquad\quad FULL\_COMPOUND(tset, nondetres)), conf) = \\
\qquad\qquad confENTEREDSEQ(tseq, conf, nondetres) \land \\
\qquad\qquad labelCOMPUTED(tseq, m, nondetres) = (toTRANSITION_D(\\
\qquad\qquad\quad FULL\_COMPOUND(tset, nondetres))).label\ m)
\end{array}
$$

---

the above definition assumes that any sequentialisation of *tset* will deliver the same target configuration which will coincide with that for taking all those transitions in the same step.

Theorem 6.5.5. *The set of test cases in Eqn. 3.9 on p. 58 is adequate for provable correctness of an implementation to a design as a result of testing not revealing faults.*

Proof. Testing of weak refinements of AND-states is done similarly to a non-refined case, but instead of $\Phi^{merged}(state)$, we use $\Phi^{merged}_{union}(state)$ in *expandedPhi* (Def. 6.4.16).

1. Using Prop. 6.4.17 and Prop. 6.1.66, we get that the set $\Phi^{merged}(state)$ for some state *state* can be reconstructed back from set $\Phi^{merged}_{final}(state))$. $\Phi^{merged}(state)$ consists of sets of sets of labels. For every label in any of such sets (if there is more than one label in a set, such a set corresponds to Φ^{merged} of an AND-state) we can construct FCTs corresponding to it with *transitionDefaultComplete*. The resulting FCTs could be 'put back' into construction of Φ^{merged} and the result will not differ from $\Phi^{merged}_{final}(state))$, due to Prop. 6.1.60. The term 'put back' means that we replace transitions in $\Phi(st)$ used in construction of $\Phi^{merged}(state)$, by sets of their full compound equivalents, generated using *defaultTransitionComplete*.

This proves that we can decompose parts of $\Phi^{merged}_{final}(state)$, corresponding to concurrent transitions and these constituent transitions will be included in $\Phi^{merged}_{union,final}(state)$.

2. We now show that transitions which are not tested under the weak refinement assumption (Def. 6.5.4) are behaving correctly (i.e. what this theorem is supposed to demonstrate).

   Consider $\Phi^{merged} \setminus \Phi^{merged}_{union}$. It is equal to a set of sets of labels, each set corresponding to multiple transitions taken concurrently in the same step (follows from Def. 6.4.12), with full compound equivalents (using Def. 6.4.16)

   $$\Phi^{merged}_{final}(st) \setminus \Phi^{merged}_{union,final}(st) = \\ \{lset : expandedPhi(\Phi^{merged}(st) \setminus \Phi^{merged}_{union}(st)) \bullet \langle lset \rangle\}.$$

   The transition diagram, including all the transitions taken concurrently, can be tested using $\Phi^{merged}_{union,final}$. Using the weak refinement assumption (Def. 6.5.4), we get from testing with $\Phi^{merged}_{union,final}$ not revealing faults that behaviour of those transitions taken in the same step will be identical to that when they are taken sequentially and thus this behaviour will be correct; taking them concurrently will also lead to the expected state.

□

6.5.3 Strong refinement of AND-states

Strong refinement assumes separate implementation of concurrent states and absence of communication between them under test. With this, we need to explore neither state nor transition space when testing. Consequently, separate testing of an AND-state is the same as testing of OR-state refinement and the proof for it can be found in Sect. 6.5.1 on p. 195.

6.6 Test data generation

Here we give definitions for testing requirements related to data and show how to use the testing theorem developed for X-machines, for statecharts.

We begin with definitions of helper functions aiding us to define testing requirements.

Definition 6.6.1.

$$\begin{array}{|l}
noemptytrigger\ \_ : \mathbb{F}_1(LSet) \\
\hline
\forall\, lset : LSet \bullet noemptytrigger(lset) \Leftrightarrow \\
\quad (\forall\, l : lset \mid (\exists\, tr : \Upsilon \bullet tr.label = l \land \neg\, transitionDEFAULT(tr)) \bullet \\
\qquad (\exists\, m : DATA \bullet \neg\, trigger(l, m)))
\end{array}$$

$$
\begin{array}{|l}
t\_complete\ \_ : \mathbb{F}_1(LSet) \\
stSTATUS \setminus (data, conf) \\
\hline
\forall lset : LSet \bullet t\_complete(lset) \Leftrightarrow \\
\quad (\forall ls : LSet \mid ls \neq \varnothing \land ls \subseteq lset \bullet \\
\quad\quad (\forall m : DATA \bullet (\exists cset : CSet \bullet \\
\quad\quad\quad triggerSET(ls, modify(filter(cset, inport), m)) \land \\
\quad\quad\quad \neg (\exists l_f : lset \setminus ls \bullet trigger(l_f, m)))))
\end{array}
$$

$$
\begin{array}{|l}
noracing\ \_ : \mathbb{F}_1(LSet) \\
\hline
\forall lset : LSet \bullet noracing(lset) \Leftrightarrow \\
\quad (\forall l_1, l_2 : lset;\ m : DATA \mid l_1 \neq l_2 \land triggerSET(lset, m) \bullet \\
\quad\quad \neg\ racing(l_1(m), l_2(m)))
\end{array}
$$

$$
\begin{array}{|l}
output\!-\!distinguishable\ \_ : \mathbb{F}_1(LSet) \\
stSTATUS \setminus (data, conf) \\
\hline
\forall lset : LSet \bullet output\!-\!distinguishable(lset) \Leftrightarrow \\
\quad (\forall l_1, l_2 : lset;\ m : DATA \mid l_1 \neq l_2 \land triggerSET(lset, m) \bullet \\
\quad filter(l_1(m), outport) \neq filter(l_2(m), outport))
\end{array}
$$

Above, in *noemptytrigger*, *t_complete*, *noracing* and *output–distinguishable*, we talk about compound transitions of a statechart. In the definition of *noracing* we use *trigger* rather than *enable*. This is needed as we are trying to verify all transitions from every state; if there are some which mask each other, an implementation error could be missed.

Requirements for the test data generation can be put down as follows:

Definition 6.6.2. *A statechart has to satisfy the requirements of t_completeness, output-distinguishability, absence of racing, empty triggers and presence of synchronous behaviour under test.*

$$
\begin{array}{l}
\exists_1\ lset == \{tr : \Upsilon \bullet tr.label\} \bullet \\
\quad t\_complete(lset) \land noemptytrigger(lset) \land \\
\quad output\!-\!distinguishable(lset) \land noracing(lset)
\end{array}
$$

Note that this definition is rather strict; indeed, it requires completeness and output-distinguishability of *every* compound transition. In practice, this could be limited to those used for testing. For instance, we do not have to require output-distinguishability of default transitions which cannot be taken in the same step by Req. 4b. Removal of some unnecessary constraints could be a subject of future work.

Proposition 6.6.3. *Output-distinguishable transitions remain output-distinguishable after an application of event_discard (rather than modify) to them.*

Proof. Follows since changes from transitions are not affected by *event_discard*. □

Proposition 6.6.4. *All changes made by full compound transitions in response to our triggering them, are observable provided transitions satisfy the output-distinguishability condition for individual transitions (Sect. 4.1.3 on p. 70) and a statechart is running synchronously (Req. 3c).*

Proof. An output from a transition will not be observable if it is masked by some other transition or a number of them. This can potentially happen when more than one transition is executed in the same superstep; without loss of generality we can consider two transitions, the second of which masks changes made by the first one. Consider the following cases:

- *These two transitions execute in different steps.* This contradicts the requirement of synchronous execution under test (Req. 3c),
- *Transitions execute in the same step, either in concurrent states or as a part of a single full compound transition.* Racing is prohibited (Req. 3b).

□

Although the above proposition shows observability, it does not prove output-distinguishability, which together with t_completeness and other requirements can be ensured as a part of design for test described in Sect. 4.1.3 on p. 70.

6.7 Testing theorem

In this section we essentially present the testing theorem from [Ipa95] with some clarifications, therefore only an outline proof of the main result (Th. 6.7.3) is provided.

Consider a finite-state machine $\mathcal{A}$ corresponding to a simple statechart, such that the two have the same number of states. For every transition tr of an original simple statechart, we have the corresponding $tr_{\mathcal{A}}$ with the same source and target states as tr but with the label $tr.label/1$. This automaton has the set of inputs being the set of labels of transitions Φ in our simple statechart and the output set of $\{0, 1\}$. We make $\mathcal{A}$ complete, i.e.

have a transition defined for every possible input from all states, by adding transitions from every state to the *SINK* state with output 0. Function computed by $\mathcal{A}$ is denoted $\mathcal{A}(lseq)$ for a sequence of labels $lseq : LSeq$.

For example, for the main statechart of our tape recorder, we have $\mathcal{A}$ depicted in Fig. 6.19[3]. Transitions with output 0 are drawn in thick lines.

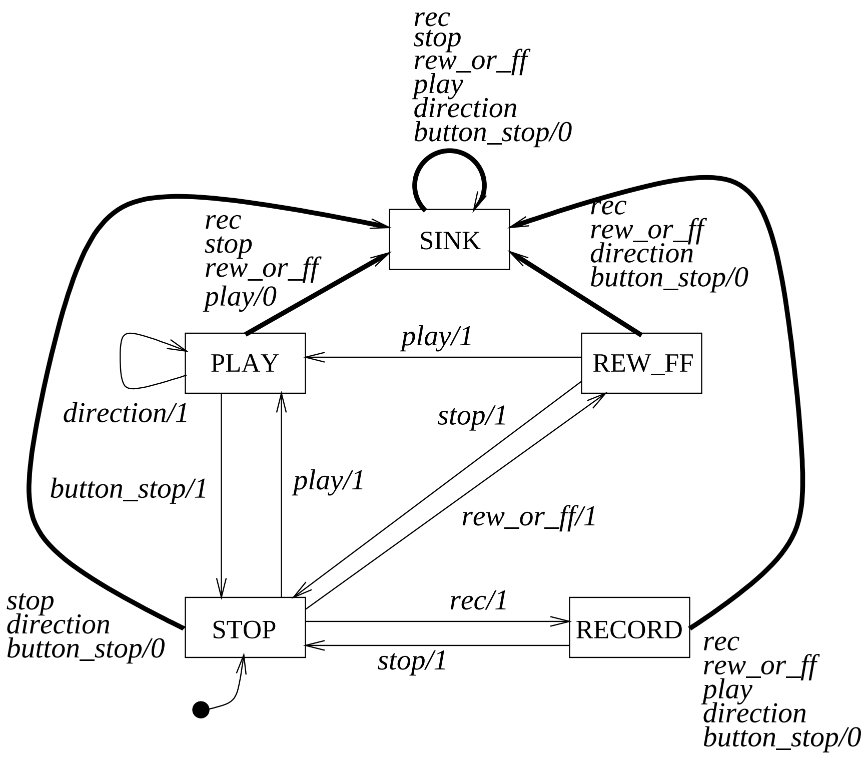


Figure 6.19: The tape recorder FSM with a sink state

Multiple transitions between the same pair of states with the same output are drawn as a single transition. For example, between *PLAY* and *SINK* we have *rew_or_ff*/0, *stop*/0, *rec*/0, *play*/0.

Proposition 6.7.1. *There is the following relation between paths of the simple statechart and $\mathcal{A}$:*

$$\forall\, lseq : LSeq \bullet lpathEXISTS(lseq) \Leftrightarrow 0 \notin \mathsf{ran}\,\mathcal{A}(lseq)$$

Proof. If a path in the simple statechart labelled *lseq* exists, then no transition to the *SINK* state in $\mathcal{A}$ will happen and thus there would be no 0 in the output sequence. If there is no path, then the *SINK* state will be entered and the output sequence will contain 0. Conversely, if there is no 0 in the

[3]In actual fact, the *SINK* state can be expected to exhibit divergence for any input.

output sequence, no transition to *SINK* was taken and the path exists in the original simple statechart; if there was 0, no such path would exist by construction of $\mathcal{A}$. □

Proposition 6.7.2. *W constructed above is valid for $\mathcal{A}$; C visits all states but* SINK.

Proof. Follows from Prop. 6.7.1. □

Theorem 6.7.3. *Consider a statechart design satisfying the design requirements of Chap. 5 on p. 79, and an implementation of the considered system satisfying the implementation-related requirements of the same chapter. Then if the implementation delivers the expected output for $t(T)$ where t is the fundamental test function and T the set of test cases as follows*

$$\begin{aligned} T \;=\; & \mathit{MultOR}(\mathit{MultOR}(C^{merged}, \{\langle\rangle\} \cup \\ & \mathit{RaiseToPower}(\Phi^{merged}_{final}, 1) \cup \mathit{RaiseToPower}(\Phi^{merged}_{final}, 2) \cup \\ & \cup \ldots \cup \mathit{RaiseToPower}(\Phi^{merged}_{final}, m-n+1)),\, W^{merged}_{final}), \end{aligned}$$

then we have the behavioural equivalence between the considered design and implementation.

Proof. We provide a sketch of the proof.

From requirements for statecharts, Def. 6.6.2, and Prop. 6.2.17 it follows that in an implementation, when we trigger a transition, we can always tell if it occurred or not. [Ipa95] proves that provided an X-machine satisfies output-distinguishability and t_completeness, the t-fundamental test function, converting a sequence of labels to that of input/output pairs, allows to reason based on the output of the system under test, that the *lpath* considered exists there or not.

Let us assume that an implementation contains an implicit *SINK* state such that if a transition with some label does not exist from a state, it is assumed to lead to that state. With this assumption, the Chow's theorem [Cho78] can be applied to show the equivalence of behaviour between $\mathcal{A}$ and implementation. With the help of Prop. 6.7.2 and Th. 6.4.26, we get the behavioural equivalence between design and implementation. This follows from a proof in [Ipa95] for X-machines and that simple statecharts are behaviourally-equivalent to X-machines (Prop. 6.3.10). □

6.8 Future work — OR- and AND-connectors

C, **S**, junction and diagram connectors could be included in the general axiomatisation of statecharts described above. It is based on the introduction of a special type for states called *CONNECTOR*. Valid transitions are restricted such that

$$\begin{array}{l}\forall\, tr : \tau \bullet \\ \quad (CONNECTOR \notin \phi(\!| tr.source |\!) \vee \\ \quad \{CONNECTOR\} = \phi(\!| tr.source |\!)) \wedge \\ \quad (CONNECTOR \notin \phi(\!| tr.target |\!) \vee \\ \quad \{CONNECTOR\} = \phi(\!| tr.target |\!))\end{array}$$

Definition 6.8.1.

$$\begin{array}{|l}transitionCONTINUATION\ \_ : \mathbb{F}_1(TRANSITION) \\ \hline \forall\, tr : \Upsilon \bullet \\ \quad transitionCONTINUATION(tr) \Leftrightarrow \phi(\!| tr.source |\!) = \{CONNECTOR\}\end{array}$$

Definition 6.8.2. *toTRANSITION$_C$ is defined similarly to toTRANSITION$_D$ (Def. 6.1.50 on p. 145),*

$$\begin{array}{|l}toTRANSITION_C : TSet \twoheadrightarrow TRANSITION \\ \hline \forall\, tset : \mathbb{F}_1\, \Upsilon \mid tsetVALID(tset) \bullet \\ \quad toTRANSITION_C(tset) = \\ \qquad \langle\!| \ source == \bigcup\{t : tset \mid \neg\, transitionDEFAULT(t) \wedge \\ \qquad\quad \neg\, transitionCONTINUATION(t) \bullet t.source\}, \\ \qquad target == \bigcup\{tr_s : tset \mid \\ \qquad\quad \neg\, (\exists\, tr_c : tset \bullet continuationDEF(tr_s, tr_c)) \bullet tr_s.target\}, \\ \qquad label == andALL(tset) \ |\!\rangle\end{array}$$

defaultEntranceComplete and other functions have to be modified to take transitions with connectors into account. Some problems with the design for test for the considered connectors are described in Sect. 4.1.3 on p. 70.

Note that loops in the system of transitions comprising a compound transition are impossible due to conjunctional semantics of statecharts (Sect. 1.4.10 on p. 24) within a step[4].

[4]a loop does not make sense for this type of semantics where transitions in a set of them can be executed within a step in any order and will produce the same result.

Given a sketch of how the theory could be extended to handle transitions involving connectors, we will not elaborate on it and leave it for future work.

Chapter 7

Tool support

A method cannot find its acceptance in industry unless a toolset supporting it has been built. Consequently, an implementation of the test set generation method is necessary to have it adopted and used. In this section we describe the tool support built as a part of the ESPRESS project, with particular attention paid to the test set generation tool for statecharts, called *TestGen*, which was developed by the author. All ESPRESS tools are prototypes; if accepted, they will be rebuilt by a firm which would afterwards support them.

7.1 $\mu\mathcal{SZ}$ tool support

Statemate tool is used for statechart editing and simulation. Transition labels are designed using Z separately from statecharts. The editor runs under *xemacs* and provides an easy way of entering schemas. The representation for Z is LaTeX-like. Users can either use an 'insert symbol' menu or type control sequences; after being typed they are visualised as an appropriate Z symbol. A type-checker for Z is implemented. The environment permits code generation from explicit Z schemas and integration with Statemate's simulation facilities. Usage of the SMV model-checker and the HOL-Z/Isabelle theorem-prover is also possible.

7.2 TestGen tool

The TestGen tool is supposed to generate test cases and convert them to test inputs for testing statechart designs against implementations. It implements the main elements of the testing method described in Chap. 3 on p. 46 and Chap. 4 on p. 65. Designs can be loaded and saved within the ESPRESS framework the tool is supposed to run in. They can be *annotated* by a user to make certain refinement assumptions (for AND and OR-states) and to ignore specific transitions at certain stages in the test case generation.

7.2.1 Requirements for the tool

Inputs for the tool

The ZIRP format is used by the ESPRESS toolset (also referred to as `ET`) to communicate data between tools. For example, a Z design can be passed to a theorem-prover to derive disjunctive normal form for Z schemas of it.

Statechart designs in ZIRP and XMI format can be directly loaded into the tool; Statemate databank files and workarea ones have to be converted to the ZIRP format first. XMI [XMI99] is an exchange format developed for UML CASE tools which recently (Mar 99) became a standard. It is based on XML [XML], a widely accepted format for the interchange of information.

Inputs which are needed for test data generation are extracted by TestGen from statecharts with labels adhering to the simple 'single event/action' format; they can also be explicitly added by a user using the tool.

Designs are expected to consist of Units which in turn consist of process classes, containing statecharts. Semantically, a class is a part of larger design, connected to other classes through ports, drawn on a data flow diagram. Behaviour of such a class is given by a statechart[1], refer to Sect. 1.4.11 on p. 24 for more details.

Outputs of the tool

Sets of test cases and test inputs are displayed by TestGen on the screen in a popup window (TestGen graphical interface, also referred to as *GUI*), in the terminal window (command-line version) or saved in a file. If errors are reported, GUI also allows navigation to an erroneous part of a design.

7.2.2 TestGen graphical interface

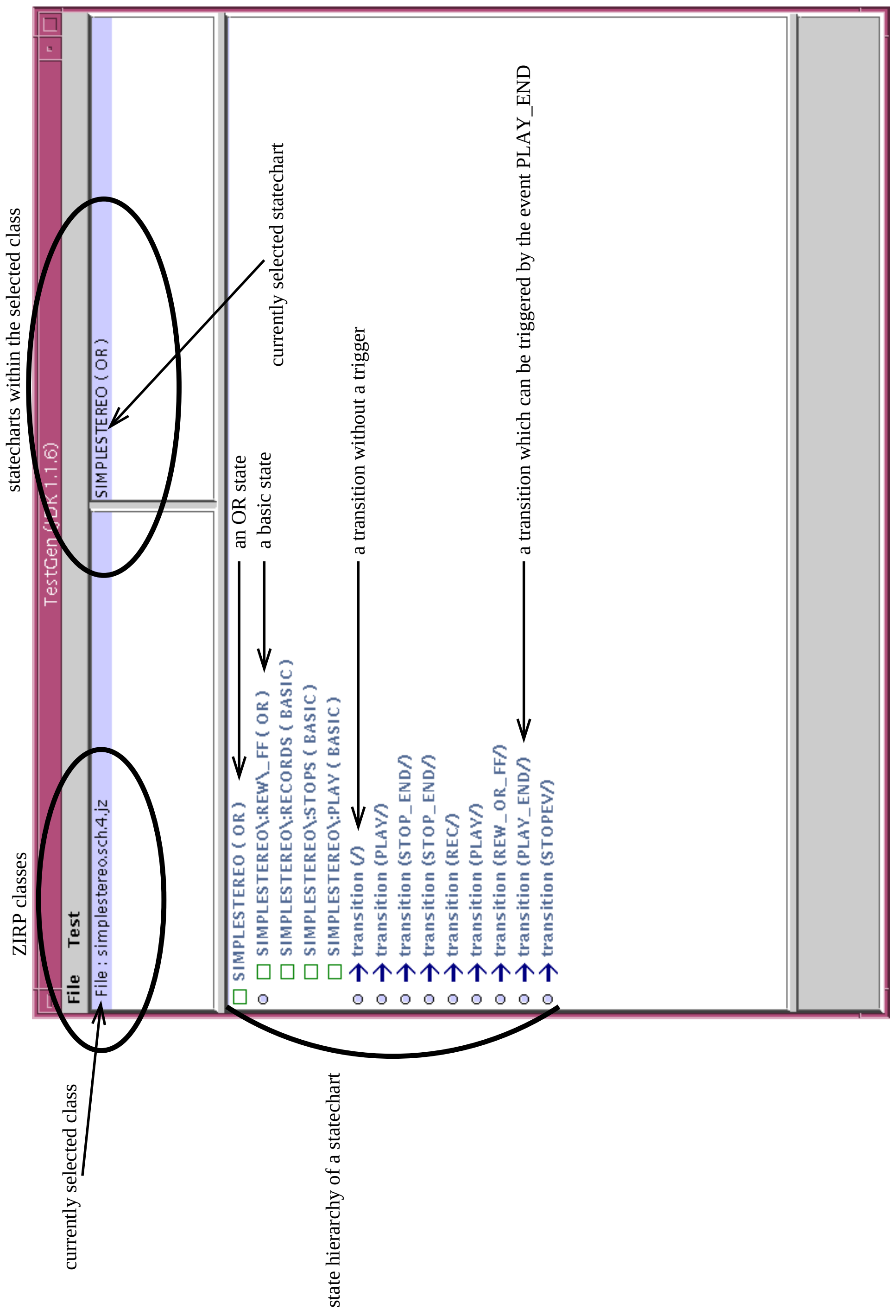
The Main screen

The main screen of the TestGen tool is given in Fig. 7.1. In the top-left of the screen loaded process classes are displayed, in the top-right part of it — statecharts of the currently selected class. Selected classes and statecharts are given in blue. In the main portion of the window the state hierarchy with transitions is shown, depicting our tape recorder in Fig. 1.6[2]. The bottom portion of the main window is for error messages, refer to p. 217 for details.

In the state hierarchy, states are displayed with their corresponding names and types (BASIC, AND, OR). For transitions, their labels in the

[1]Multiple statecharts for a class suggest that they have to be combined to form the whole one. Since TestGen operates on individual statecharts only, the number of them in a class does not matter to the tool.

[2]The state *RECORD* was renamed to *RECORDS* to avoid a conflict with a reserved word.


TestGen (JDK 1.1.6)
File
Test
ZIRP classes
statecharts within the selected class
currently selected statechart
currently selected class
an OR state
a basic state
a transition without a trigger
a transition which can be triggered by the event PLAY_END
state hierarchy of a statechart

File : simplestereo.sch.4.jz

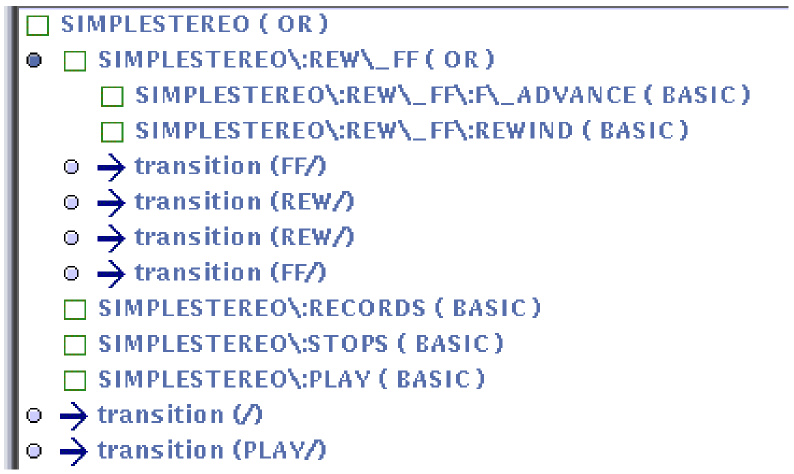
SIMPLESTEREO (OR)

```
SIMPLESTEREO ( OR )
  SIMPLESTEREO\:REW\_FF ( OR )
  SIMPLESTEREO\:RECORDS ( BASIC )
  SIMPLESTEREO\:STOPS ( BASIC )
  SIMPLESTEREO\:PLAY ( BASIC )
  transition (/)
  transition (PLAY/)
  transition (STOP_END/)
  transition (STOP_END/)
  transition (REC/)
  transition (PLAY/)
  transition (REW_OR_FF/)
  transition (PLAY_END/)
  transition (STOPEV/)
```

Figure 7.1: The main screen of the TestGen tool

form of trigger/action pairs are shown in braces. The blobs to the left of states and transitions indicate that these items can be expanded to reveal more information about them. For states we can see their substates and for transitions — their source and target states.

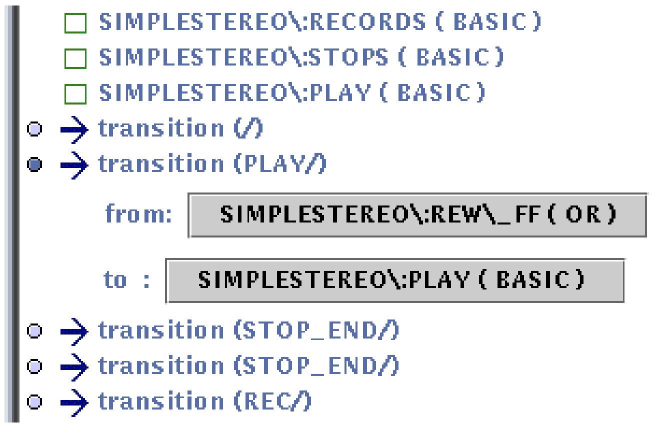
If we consider a more complex statechart which contains some OR-states, we might like to look at their substates as shown in Fig. 7.2. An example



```
☐ SIMPLESTEREO ( OR )
● ☐ SIMPLESTEREO\:REW\_FF ( OR )
      ☐ SIMPLESTEREO\:REW\_FF\:F\_ADVANCE ( BASIC )
      ☐ SIMPLESTEREO\:REW\_FF\:REWIND ( BASIC )
   ○ → transition (FF/)
   ○ → transition (REW/)
   ○ → transition (REW/)
   ○ → transition (FF/)
   ☐ SIMPLESTEREO\:RECORDS ( BASIC )
   ☐ SIMPLESTEREO\:STOPS ( BASIC )
   ☐ SIMPLESTEREO\:PLAY ( BASIC )
○ → transition (/)
○ → transition (PLAY/)
```

Figure 7.2: The expanded REW_FF state

of an expanded transition is given in Fig. 7.3. It is possible to click on



```
   ☐ SIMPLESTEREO\:RECORDS ( BASIC )
   ☐ SIMPLESTEREO\:STOPS ( BASIC )
   ☐ SIMPLESTEREO\:PLAY ( BASIC )
○ → transition (/)
● → transition (PLAY/)

      from: [ SIMPLESTEREO\:REW\_FF ( OR ) ]

      to  : [ SIMPLESTEREO\:PLAY ( BASIC ) ]

○ → transition (STOP_END/)
○ → transition (STOP_END/)
○ → transition (REC/)
```

Figure 7.3: Expanded transition

‘buttoned’ source and target states of a transition to navigate to them.

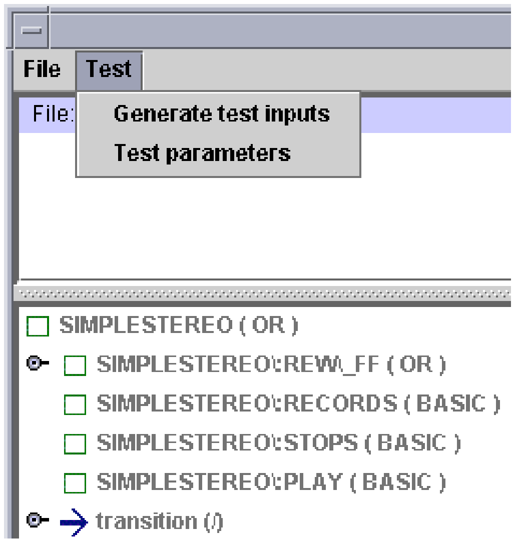

File
Test
File:
Generate test inputs
Test parameters
SIMPLESTEREO (OR)
SIMPLESTEREO\:REW_FF (OR)
SIMPLESTEREO\:RECORDS (BASIC)
SIMPLESTEREO\:STOPS (BASIC)
SIMPLESTEREO\:PLAY (BASIC)
transition (/)

Figure 7.4: The ‘Test’ menu

Test case and test set generation

Test input generation is performed by selecting ‘Test Inputs’ either from the ‘Test’ menu (Fig. 7.4) or from the popup menu of the top-level state of a statechart (the most top-left state on the state hierarchy diagram), with the latter shown in Fig. 7.5. The popup menu can be obtained via a click on a state with the right mouse button. After that, the state becomes selected with the black border around it and the menu appears.

One can choose ‘Test Cases’ in order to generate test cases for a state. Although it is possible to do that for any of them, it makes most sense to generate it for the top-level state of a statechart.

The sample result of test input generation is shown in Fig. 7.6. The caption of the window represents the kind of the set (‘Test Inputs’ in our case), the name of the state for which the operation has been done is shown under it (‘SIMPLESTEREO’) with the test set or a set of test cases following. Every line corresponds to a single test sequence. For a test input set, inputs to be taken in a single step are given in curvy brackets ‘{’, ‘}’.

For test cases, transitions’ triggers are shown in brackets (Fig. 7.7) and those which should be invoked in the same step are separated by dashes. A transition with an empty trigger ’{ }’ is a default one. Test cases are produced simply by multiplication and thus infeasible paths at the end of sequences are not removed. Their removal during test data generation results in a reduction of the set of test sequences, note the different number of the test cases in Fig. 7.7 and test inputs in Fig. 7.6.

When generating sets of test cases or inputs, we need to make an assump-

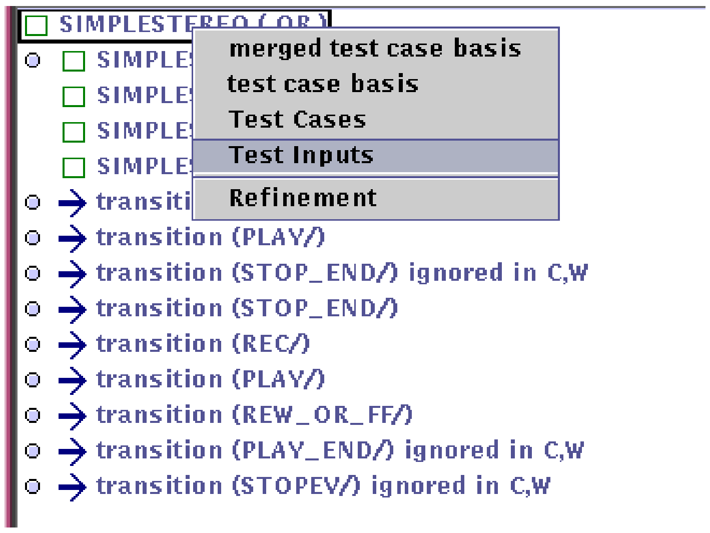


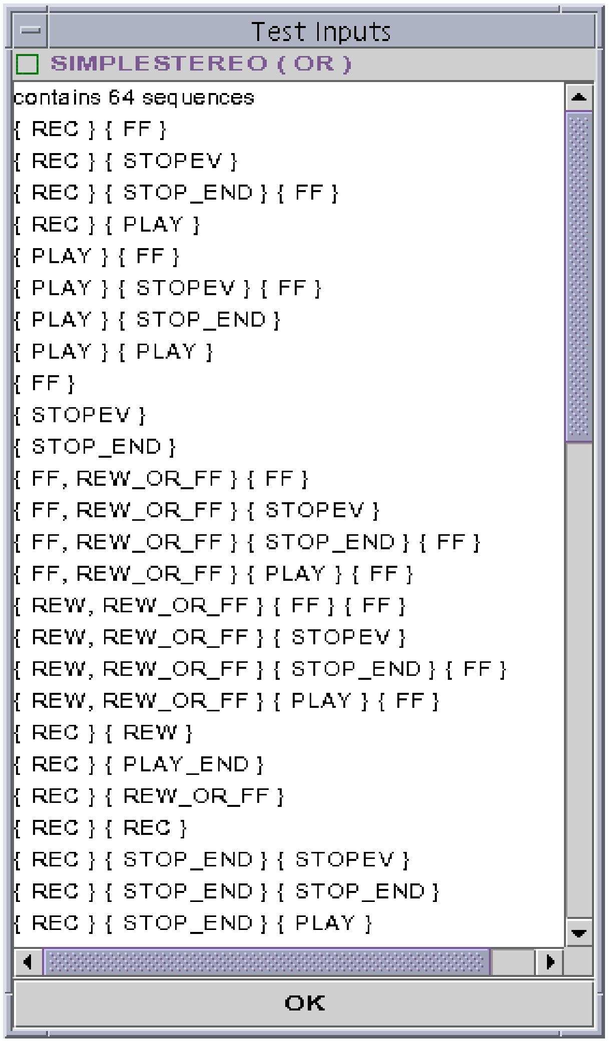
Figure 7.5: Selection of 'Test Inputs' from the popup menu

tion about how many extra states our implementation could have, compared to the number of states in the design (the size of the test set grows depending on the number of 'loops' in a statechart (Sect. 4.2.3 on p. 74), but no faster than the size of the set of test cases). Selection of a difference $m - n$ can be either done through the 'Test' menu or by annotating the top-level state of a statechart with refinement.

Refinement

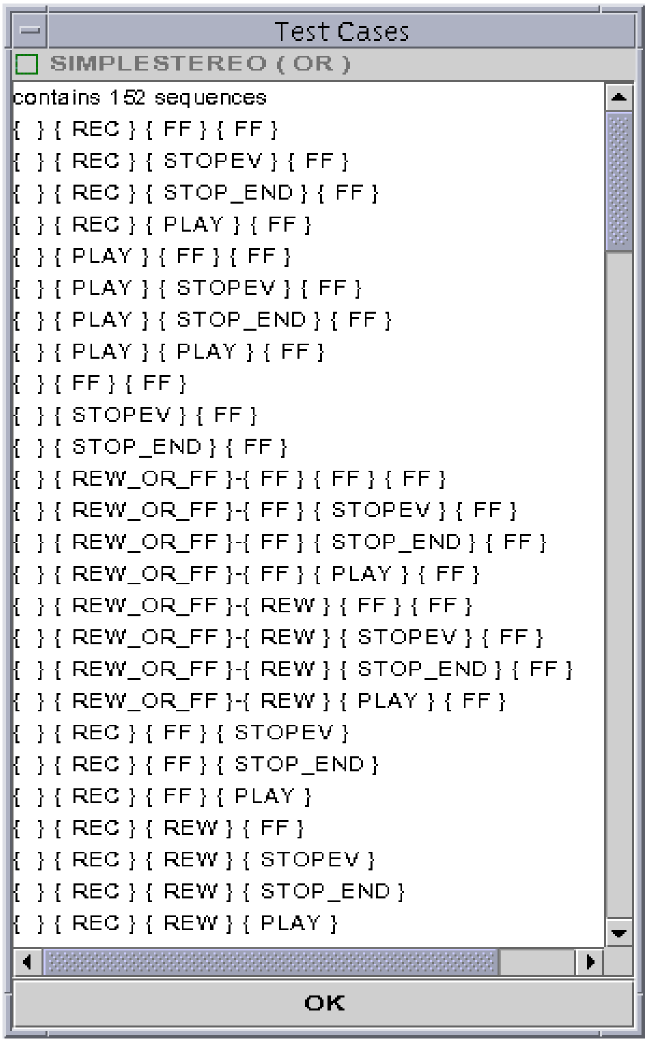
An OR-state being refined means that the statechart within that state will be tested separately from the one in the enclosing state. This allows us to reduce the size of the set of test cases. Testing using refinement is possible if a design and implementation were developed in parallel. It means that at some stage we could have a statechart for the enclosing state and an implementation of it. After that, we add a new statechart to the state we wish to refine and make an appropriate change in the implementation. More detail is given in Sect. 3.2.3 on p. 52.

Refinement annotations are added by clicking with the right mouse button on a state and selecting 'Refinement' from the popup menu. Then, we select the relevant box from a dialog (Fig. 7.8). When having a substate statechart tested separately, we could have a different assumption of the number of extra states for it, compared with that for the enclosing state. 'default refinement' means the tool will take the value from the parent state or any one above it with the assumed maximal number of extra states explicitly assigned. If no such state is found, the information provided in the



```
contains 64 sequences
{ REC } { FF }
{ REC } { STOPEV }
{ REC } { STOP_END } { FF }
{ REC } { PLAY }
{ PLAY } { FF }
{ PLAY } { STOPEV } { FF }
{ PLAY } { STOP_END }
{ PLAY } { PLAY }
{ FF }
{ STOPEV }
{ STOP_END }
{ FF, REW_OR_FF } { FF }
{ FF, REW_OR_FF } { STOPEV }
{ FF, REW_OR_FF } { STOP_END } { FF }
{ FF, REW_OR_FF } { PLAY } { FF }
{ REW, REW_OR_FF } { FF } { FF }
{ REW, REW_OR_FF } { STOPEV }
{ REW, REW_OR_FF } { STOP_END } { FF }
{ REW, REW_OR_FF } { PLAY } { FF }
{ REC } { REW }
{ REC } { PLAY_END }
{ REC } { REW_OR_FF }
{ REC } { REC }
{ REC } { STOP_END } { STOPEV }
{ REC } { STOP_END } { STOP_END }
{ REC } { STOP_END } { PLAY }
```

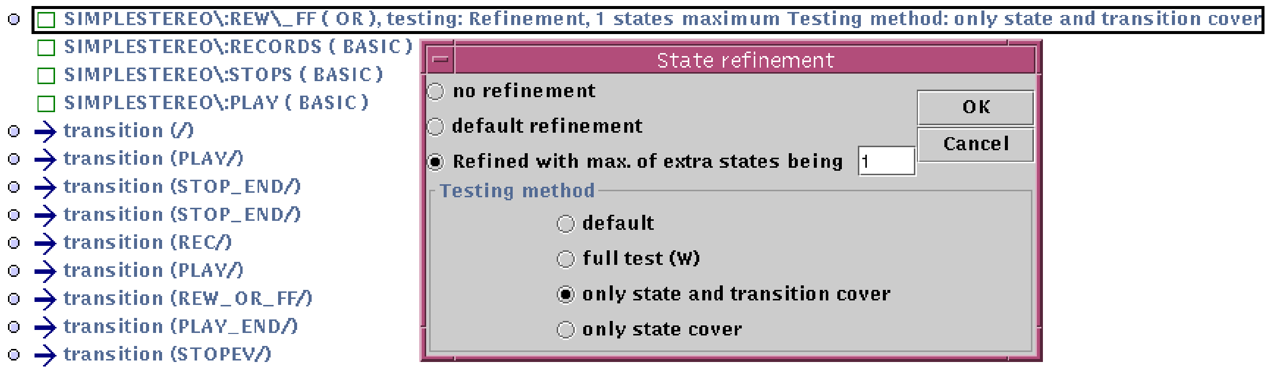
Figure 7.6: Sample sequences of test inputs



```
contains 152 sequences
{ } { REC } { FF } { FF }
{ } { REC } { STOPEV } { FF }
{ } { REC } { STOP_END } { FF }
{ } { REC } { PLAY } { FF }
{ } { PLAY } { FF } { FF }
{ } { PLAY } { STOPEV } { FF }
{ } { PLAY } { STOP_END } { FF }
{ } { PLAY } { PLAY } { FF }
{ } { FF } { FF }
{ } { STOPEV } { FF }
{ } { STOP_END } { FF }
{ } { REW_OR_FF }-{ FF } { FF } { FF }
{ } { REW_OR_FF }-{ FF } { STOPEV } { FF }
{ } { REW_OR_FF }-{ FF } { STOP_END } { FF }
{ } { REW_OR_FF }-{ FF } { PLAY } { FF }
{ } { REW_OR_FF }-{ REW } { FF } { FF }
{ } { REW_OR_FF }-{ REW } { STOPEV } { FF }
{ } { REW_OR_FF }-{ REW } { STOP_END } { FF }
{ } { REW_OR_FF }-{ REW } { PLAY } { FF }
{ } { REC } { FF } { STOPEV }
{ } { REC } { FF } { STOP_END }
{ } { REC } { FF } { PLAY }
{ } { REC } { REW } { FF }
{ } { REC } { REW } { STOPEV }
{ } { REC } { REW } { STOP_END }
{ } { REC } { REW } { PLAY }
```

Figure 7.7: Sample test cases

‘testing parameters’ dialog is used.



SIMPLESTEREO\:REW_FF (OR), testing: Refinement, 1 states maximum Testing method: only state and transition cover
SIMPLESTEREO\:RECORDS (BASIC)
SIMPLESTEREO\:STOPS (BASIC)
SIMPLESTEREO\:PLAY (BASIC)
transition (/)
transition (PLAY/)
transition (STOP_END/)
transition (STOP_END/)
transition (REC/)
transition (PLAY/)
transition (REW_OR_FF/)
transition (PLAY_END/)
transition (STOPEV/)

State refinement

no refinement
default refinement
[✓ Refined with max. of extra states being] 1

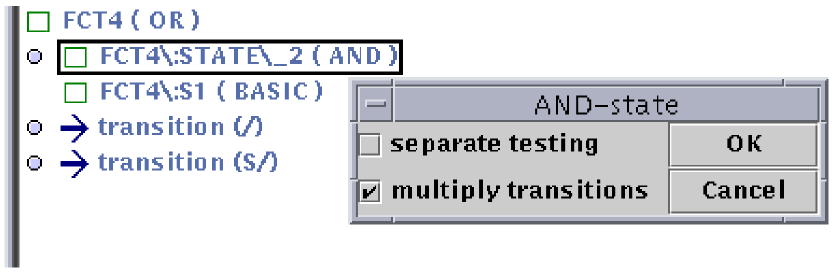
Testing method
default
full test (W)
[✓ only state and transition cover]
only state cover

Figure 7.8: Refinement of an OR-state

The ‘Testing method’ element of the dialog allows us to select the testing method to be utilised to test the statechart within the state. ‘full test’ is the test method described in the thesis; only state and transition cover means that we do not verify entered states, using the Eqn. 2.2 on p. 33 without W at the end and ‘state cover’ only visits all states. The last two selections could be used to reduce the size of a test set, but no claim of correct implementation can be made if they are used. Selection of the ‘default’ testing approach means usage of the method selected for the parent state or the nearest state up in the state hierarchy, annotated with a particular type of testing. If there is none, the one specified in the testing preferences dialog (Fig. 7.13) is used.

The approach to testing AND-states is testing their expansion via state and transition multiplication (Sect. 3.3.1 on p. 54). This results in a big number of test cases; we could reduce it by making some assumptions. The two of them, described in Sect. 3.3.3 on p. 57 and Sect. 3.3.4 on p. 59, are supported by the tool: usage of union of transitions instead of a multiplication and separate testing of substates of an AND-state. The dialog to select the appropriate assumption, shown in Fig. 7.9, is invoked similarly to the one for refinement of OR-states, by selecting the ‘Multiplication parameters’ from the popup menu. Separate testing means strong refinement, if it is not ticked, either no refinement can be selected (shown in the figure) or the weak one. The latter can be chosen by removing the tick from the ‘multiply’ box.

There are some restrictions on the usage of annotations for AND-states and refinement annotations for immediate substates of AND-states. Specifically, if the contents of an AND-state are tested as a multiplication of states (i.e. weak or no refinement), refinement annotations of its OR substates do not make sense. Consequently, the tool will not allow us to annotate substates of an AND-state which is not marked for separate testing and if

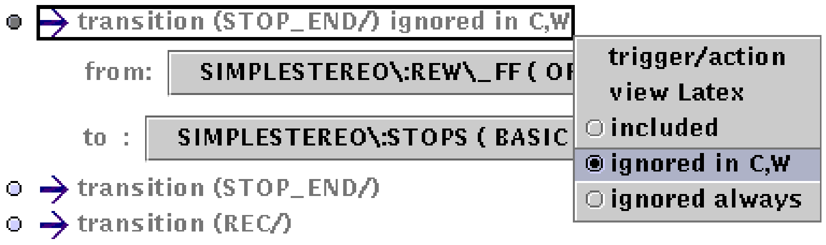

FCT4 (OR)
FCT4\:STATE_2 (AND)
FCT4\:S1 (BASIC)
transition (/)
transition (S/)

AND-state
☐ separate testing
☑ multiply transitions

Figure 7.9: Refinement of an AND-state

such marking is removed, substates' annotations will be removed too.

Some transitions should not participate in the C and W construction or in testing at all, supporting Req. 1d (in which case they have to be assumed correct). The tool allows us to mark these transitions to be ignored in the two cases described. Annotation of transitions for that is performed via the right mouse click on a transition (the transition will become selected and get a black border drawn around it) and selection of a relevant element from the popup menu. It is depicted in Fig. 7.10. If a transition is annotated in such a way, all transitions with the same trigger and output will be ignored in the same way (not at all, only for C and W construction or always). Introduction of a user-defined trigger or output of a transition allows one to define an input and/or an output to be used for testing as described in Sect. 7.2.2 on p. 219.

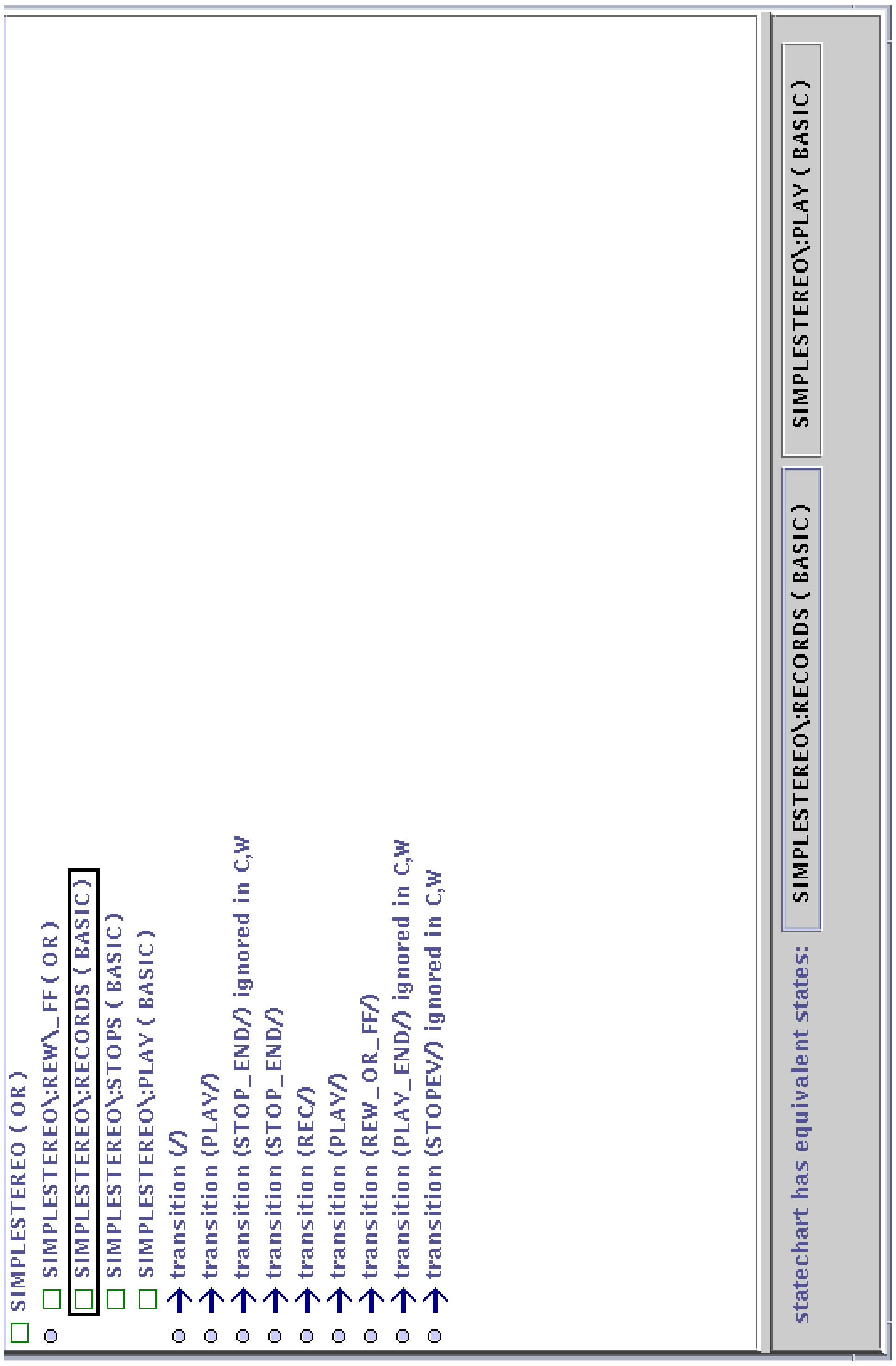

transition (STOP_END/) ignored in C,W
from: SIMPLESTEREO\:REW_FF (OI
to : SIMPLESTEREO\:STOPS (BASIC
transition (STOP_END/)
transition (REC/)

○ included
● ignored in C,W
○ ignored always

Figure 7.10: An annotation of a transition

Error messages

Sometimes, a set of test cases or a test set cannot be generated. In such a case, an error message pops up and the description of an error is added to the bottom part of the main window. As an illustration, the error message which occurs if some states are equivalent, is shown in Fig. 7.11. The two buttons with state names on them can be clicked which will result in relevant states being highlighted in the state tree.



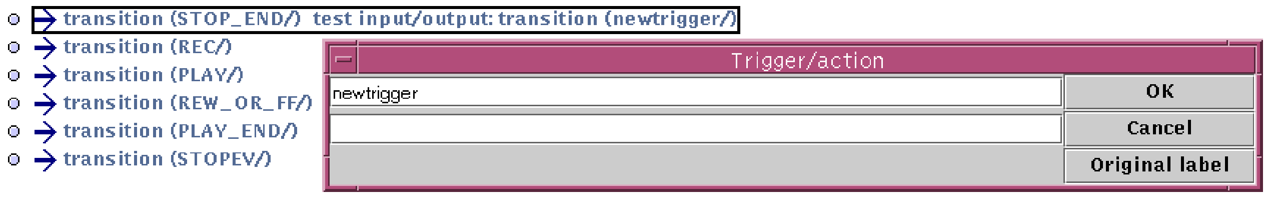
```
SIMPLESTEREO ( OR )
    SIMPLESTEREO\:REW\_FF ( OR )
    SIMPLESTEREO\:RECORDS ( BASIC )
    SIMPLESTEREO\:STOPS ( BASIC )
    SIMPLESTEREO\:PLAY ( BASIC )
    transition (/)
    transition (PLAY/)
    transition (STOP_END/) ignored in C,W
    transition (STOP_END/)
    transition (REC/)
    transition (PLAY/)
    transition (REW_OR_FF/)
    transition (PLAY_END/) ignored in C,W
    transition (STOPEV/) ignored in C,W

statechart has equivalent states:  SIMPLESTEREO\:RECORDS ( BASIC )  SIMPLESTEREO\:PLAY ( BASIC )
```

Figure 7.11: ‘some states are equivalent’ error message

The error above was produced by annotating transitions *STOP_END*, *PLAY_END* and *STOPEV* to be ignored and selecting ‘Test Cases’ for the top-level state. Such an annotation results in all transitions labelled *STOP_END*, *PLAY_END* and *STOPEV* to be ignored, even if some of them are not explicitly marked for that.

Transition triggers

In order to construct a test set, TestGen needs to know what input will trigger which transition. Simple triggers can be extracted from a statechart, if not, a user has to add them. Selection of ‘trigger/action’ from a popup menu of a transition allows that. The dialog is given in Fig. 7.12 with the top line for a trigger and the bottom one — an action. An annotation will be added to a transition containing user-selected trigger and an action; it is afterwards displayed showing the original trigger/action and the one added by a user. It is always possible to remove the pair added by clicking the ‘Original label’ button.


transition (STOP_END/) test input/output: transition (newtrigger/)
transition (REC/)
transition (PLAY/)
transition (REW_OR_FF/)
transition (PLAY_END/)
transition (STOPEV/)

Trigger/action

newtrigger

Figure 7.12: Selection of a trigger and an action for a transition

Main menu

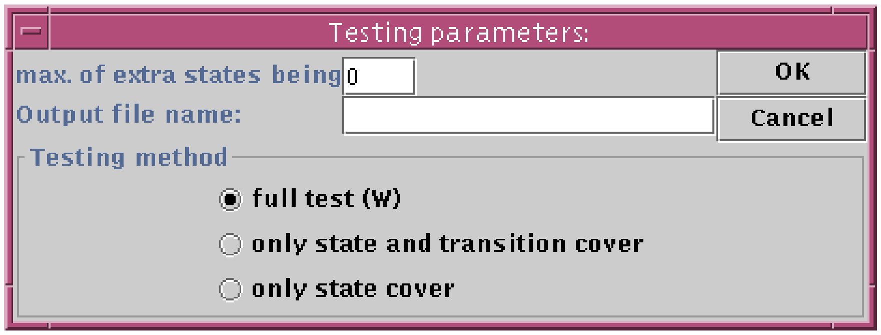
In the top-left corner of the main window, just below the title bar, there are a ‘File’ and ‘Test’ menus. We describe them in turn.

The ‘File’ menu allows us to save the ZIRP unit (corresponding to the currently selected class (units consist of a number of process classes, possibly containing multiple statecharts each), close one (after prompting a user to save) or exit the tool. Names of units are not shown by the tool. Classes corresponding to a modified unit are marked with ‘*’ to the left of their names. For XML statecharts, a ZIRP unit containing one class with a single statechart in it is saved.

The ‘Test’ menu contains two items, ‘Generate test inputs’ and ‘Test parameters’ (Fig. 7.4). The former allows us to generate test inputs. It behaves the same way as the ‘Test inputs’ selection in the popup menu for the root state of a statechart.

The number of extra states and the preferred testing method can be defined by selecting ‘Test parameters’ from the ‘Test’ menu. The dialog is shown in Fig. 7.13 and is very similar to the refinement one (Sect. 7.2.2

on p. 213). Annotations in a statechart take precedence over the values


Testing parameters:

max. of extra states being 0

Output file name:

Testing method

(● full test (W)) ○ only state and transition cover ○ only state cover

Figure 7.13: Selection of parameters for testing

specified. An output file name can be provided using the dialog. In this case, results of test set/test case generation will be recorded in the file rather than displayed on the screen.

Upon exit or closure of a selected class, if there are any unsaved changes, a user is prompted to save them. One can prevent an exit by clicking 'Cancel'; an error during saving is followed by the same behaviour. When a statechart is saved, a backup copy is created.

7.2.3 Design

This section describes the high-level design of the TestGen tool. We provide the simplified class diagram for it and give brief descriptions of each of the classes. Complete description is not included due to the size (the implementation contains 69 source files with 105 classes with more that 12000 lines of Java) which would overwhelm the reader without providing a useful insight into the structure of the tool. Full description also contains the Z design of TestGen and algorithms used in test case generation (42 pages overall) and documentation built from comments in the code (approximately 30% of the overall number of lines are comments).

Development of the Z design of TestGen proved to be very useful. It allowed us to take a high-level view and check its consistency with a type-checker, which eliminated a number of would-be problems later on.

The simplified class hierarchy of the TestGen tool and class usage are depicted in Fig. 7.14. Solid lines represent inheritance, dashed — usage and dotted — implicit usage. Boxes drawn in bold represent classes and non-bold ones — interfaces or abstract classes. 'A statechart in ET' is an abstraction of a number of classes, not shown on the diagram. Classes shown above others and connected with them, provide an appropriate service, for example, `xmachineTXT` is derived from `XMachine` and uses both `textTriggerAction` and `textMatrix`. Some classes have a few of their methods or an outline of behaviour given next to them.

There are five main groups of classes, comprising the tool:

- General-purpose, such as error reporting (not shown on the diagram).
- Representation for statecharts and X-machines.
- Test case and set generation and representation.

  From the diagram, it is clear that the classes comprising these two groups of classes, can be divided into two groups: those dealing with the textual representation of X-machines and statechart-related. The latter use a more complex representation for events in consideration for possible future incorporation of actual test data generation for statecharts.

  When the author's Ph.D. work commenced, the tool *stm_get* was built, implementing the X-machine testing method; a Windows user interface has later been added to it. Stm_get was also translated into Java and formed the basis of the testing method implemented in TestGen. The textual part of the tool is essentially transformed stm_get; this part was useful during the initial testing of the described tool.
- User interface (not shown on the diagram).
- Support for statecharts encoded in XML (not shown on the diagram).

In the following we use the `typewriter` font to indicate class or interface names. The 'abstract' word in front of them means that either the class is abstract, i.e. has some of its functionality not implemented, or is an interface[3]. Either way, nothing abstract can be instantiated, i.e. an object belonging to the abstract class or interface cannot be directly created.

General-purpose classes used by all parts of TestGen

General-purpose classes are classes which are used throughout the tool, such as error reporting classes and the one used to facilitate observability of the behaviour of the tool.

`ET` allows almost every element of it such as elements of statecharts to be annotated. This is used in TestGen to mark transitions which should not be included in C and W or not used at all. In addition, states which are refined are marked as such with related information.

[3]An interface in Java is a set of methods which a class conforming to the interface, must provide functionality for. Such a class is said to *implement* the interface.

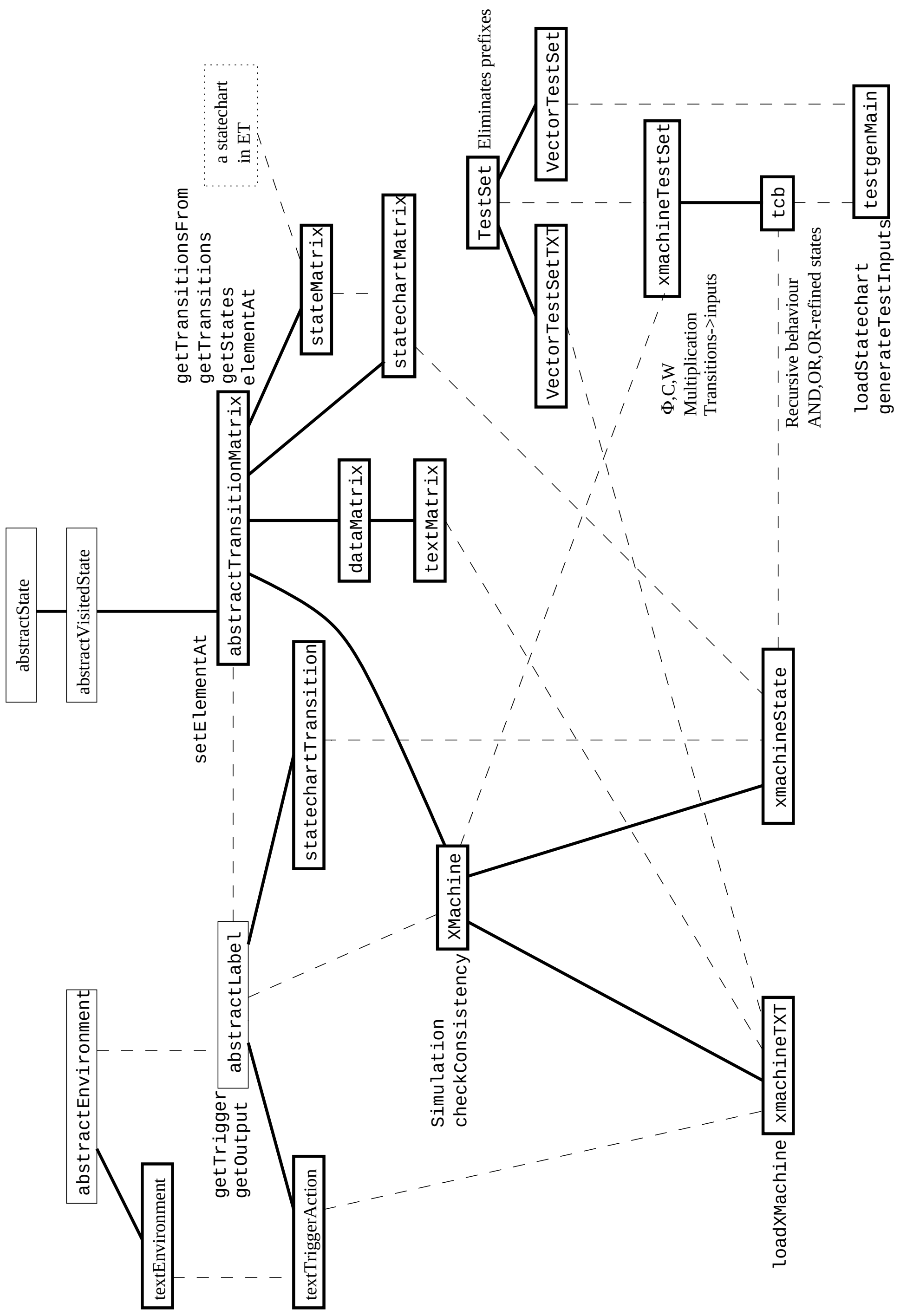


Figure 7.14: The simplified class structure of the TestGen tool

Classes representing a statechart or an X-machine

This group of interfaces and classes contains classes representing fundamental data structures and methods used in the representation of statecharts, X-machines and test set generation. We now focus on the main ones, depicted in Fig. 7.14.

`abstractEnvironment` The `abstractEnvironment` interface is an abstraction of the data of a statechart or memory of an X-machine as well as input and output changes (Sect. 6.1.2 on p. 118). It is possible to say that it reflects *SPACE*.

Although the tool does not operate with data, the design contains provisions to accommodate it. This makes simulation of complete statecharts possible with the purpose of generation of test outputs.

`abstractState` This interface encapsulates the notion of a state. It is a generalisation of all possible states and connectors of statecharts and X-machines. Every state has to have a name which could be empty or `null` (no name assigned).

`abstractVisitedState` This is a state which can be visited, by invoking the `setVisited` method of it and queried with the `getVisited` one. All state-related classes used in statecharts and X-machines derive from it because its functionality is used in test case construction.

`abstractLabel` The `abstractLabel` class encapsulates the concept of a label as used by the test case generation method. In terms of statecharts, these are compound transitions constructed from parts separated by connectors (`ET` removes OR-type connectors such as **C** but not AND ones such as fork or joint). Default connectors are important for test case generation and are not removed.

The `Transition` abstract class is derived from `abstractLabel` and contains the following methods:

`getTrigger` takes *DATA* and returns a trigger which is a *CHANGE* necessary to make to *DATA* to trigger the transition.

`getOutput` takes $data : DATA$, a trigger (possibly returned by `getTrigger`) and returns an output corresponding to the trigger, possibly updating the *data*. In terms of Chap. 6, it acts as

$$data' = modify(andLABEL(modify(trigger, data)), data)$$

This relies on the fact that we assume that every transition has a predefined test input and output, thus discarding of events which are not generated (via *event_discard*) was not implemented in

TestGen; additionally, as we do not trigger conflicting transitions (Prop. 6.1.64 on p. 154), there is no need to consider priorities during simulation, consequently *andALL* rather than *execLABEL* was used in the construction of *andLABEL*.

`getSource,getTarget` return sets of source and target states of a transition.

This class can be subclassed to implement specific types of transitions, such as the following:

transitions of X-machines This class, called `textTriggerAction`, describes transitions with a single source and target states, a textual trigger and an output.

individual transitions of statecharts The `stateTransition` class makes possible usage of multiple source and target states, and a reference to the representation of a trigger and an action in `ET`. Such transitions are used in the `stateMatrix` class described below. They are constructed from `stateTransitions` by removing all connectors but default ones.

interlevel transitions of statecharts are represented by the `interlevelTransition` class.

`abstractTransitionMatrix` This one is used to describe a transition matrix of a state of a statechart, or that of an X-machine. The interface allows user to retrieve the following information:

`getTransitions` returns a set of transitions in the matrix,

`getTransitionsFrom` returns a sequence of pairs (*abstractState*, *abstractTransition*) for every transition which emanate from a given state,

`getTransitionsTo` returns a sequence of pairs (*abstractState*, *abstractTransition*) for every transition entering a given state.

The transition matrix is an abstraction of a transition structure. Main classes implementing it are the following:

`stateMatrix` reflects the transition diagram of a statechart as obtained from the `ET` framework.

`statechartMatrix` is the transition diagram of a statechart with all connectors but default ones removed.

`dataMatrix` holds a transition diagram of an X-machine; it is subclassed to `textMatrix` such that the diagram can be loaded from a text file.

The file uses stm_get file format where every line specifies a transition in the form

source_state_name target_state_name transition_name

Initial state is taken to be the first encountered one.

`XMachine` The `XMachine` class contains operations on the transition matrix needed by an X-machine (represented by an instance of `abstractTransitionMatrix`) and methods to verify its validity.

The purpose of this class is to give an X-machine appearance to a transition matrix and aid with consistency checking and test case and set generation. When an instance of `XMachine` is constructed, it is supplied with a transition matrix and an initial state.

This class is subclassed to implement an X-machine with data being read from a text file as well as a substate statechart of a state read from `ET` framework. In the former case we have the `xmachineTXT` class and in the latter — the `xmachineState` one.

Classes related to test case and set generation

Now we describe classes representing a set of test cases or a test set as well as those used by TestGen to generate these sets.

`TestSet` is an interface. A class implementing it can represent a test set or a set of test cases. Either of them can be viewed as a set of sequences. The `addTestSequence` method provides a way to add a new sequence. When it is invoked, the sequence will be added if it is not a prefix of any one already in the set; if there is already a sequence being a prefix of the one to be added, the existing sequence is replaced with the new one.

The `VectorTestSet` class is an implementation of the `TestSet` interface which has a vector as an underlying storage mechanism for sequences. `VectorTestSetTXT` class is an extension of `VectorTestSet` to support representation of test data in the form of strings of text.

`xmachineTestSet` This class takes an `XMachine` and generates sets Φ, C and W for it. It also provides a multiplication function and constructs a set of test cases from a test case basis. If used with textual representation of X-machines (class `xmachineTXT`), `xmachineTestSet` can also convert a sequence of transition labels to inputs for any test case.

We subclass this class to implement the merging functionality for a test case basis; such a class is called `tcb`. It allows construction of a set of test cases for **OR**- and **AND**-states, including refined ones. Tracking the number of extra states w.r.t. user selection (Fig. 7.8) is also the responsibility of `tcb`.

Finally, the `xmachineTestSet` class implements an exclusion of user-selected transitions from testing (Fig. 7.10).

User interface

Statecharts loaded are drawn in a tree-like form for a user to inspect them and possibly annotate states with refinements. Transitions can be annotated with triggers and outputs, making them compliant with design for test; marking some of them to be ignored during testing (Req. 1d) is possible too (this could be done for shared transitions, refer to p. 104 for details). The comprehensive description of the interface is given in Sect. 7.2.2 on p. 209.

Swing user interface classes were chosen mainly for convenience of implementing the tree-like view of data since standard Java classes do not include this functionality.

The main class of TestGen's browser is called `browse`. The browser allows us to view a test case basis for a state (without consideration of its substates), merged TCB and resulting sets of test cases and test inputs.

Command-line access to the functionality of the TestGen tool is provided by the `testgenMain` class. It is useful for batch processing or testing of the tool as well as when one does not wish to wait for the interface to load.

XML support

In Spring 1999 it became clear that the `ET` framework cannot be relied upon to supply TestGen with statecharts since support for them in `ET` has been effectively dropped at that time. For this reason, the author has developed an approach to load XML-encoded statecharts into the tool. This representation was chosen because XML is a portable way to express information, currently gaining acceptance for data exchange between tools and an ability to import statecharts created by a number of them is considered important.

A standard representation of statecharts in XML is necessary to facilitate the possibilities of importing them and only one representation, which is backed by industry, is available. It is called XMI and focuses at UML. For this reason, a subset of it was chosen (description of which is not included in the thesis) and implemented using IBM's `xml4j` XML parser. As noted above (Sect. 3.8.2 on p. 64), UML statecharts have a different semantics from Statemate ones; TestGen simply treats them as if they were Statemate statecharts.

7.2.4 Implementation

The tool is implemented in Java with Pizza extensions using Swing classes for the interface. Java is the operating system- and hardware- independent programming language developed by Sun Microsystems. Pizza extensions were developed in the University of South Australia and add a number of features of C++ and functional programming languages. These features appealed to the developers of the ESPRESS tool set because Pizza allows easy handling of the syntax of Z in Java. The author of the tool had to use these extensions not only because it was necessary to access data exposed by the toolset but also due to parametric polymorphism (the feature similar to template classes in C++). This allows us to use parameterised collections which only take instances of a specific class, eliminating a number of possible implementation errors. Swing is a collection of user interface classes developed by Sun Microsystems. The collection makes it possible to construct sophisticated user interfaces and in terms of features seems to surpass many existing frameworks for a similar purpose. TestGen contains 12K lines of code in 69 files; 30% of code are comments, much of which can be assembled together using the `pizzadoc` documentation program, to form the description of all classes and methods of them.

7.2.5 Future improvements of the tool

The main limitations of the present version are that transition outputs are not supported because the tool does not generate test outputs; every transition has to have a triggering input, being a single event. Test case and test set generation take considerable time. Possible directions for improvement of the tool are the following:

- At present, when changes have been made to a statechart with the Statemate tool, all annotations have to be added anew. In future, we might add a way to save them separately from a statechart and re-apply after the statechart changes.

- Statemate allows us to associate attribute-value pairs with elements of statecharts. Instead of adding refinement annotations in a separate tool, we could use Statemate for that. Unfortunately, values of such testing attributes may have non-trivial structure and it will be difficult for a user to write them in a textual format, required by Statemate. Integration of the interface of the TestGen tool with Statemate could be done in future but is hindered by a serious lack of extensibility of Statemate.

- At the moment, TestGen is supposed to retrieve statecharts from the `ET` toolset and support for statecharts in the `ET` has been effectively dropped from the beginning of 1999. It would then be a good idea to make the tool read Statemate statecharts independently from the framework. This also includes an extension of XMI support with OR-connectors (only default and AND-type connectors are implemented at the moment) as well as ensuring compatibility with existing implementation of XMI, such as the one by IBM, which works with the Rational Rose tool.

- The test tool could be extended with the possibility for a user to view a list of ports and select those to be used for testing.
- Improvement of the support for default transitions. Here the following limitations and faults to be corrected:
  - The **DE** element of the test case basis is not implemented at present.
  - The present version of TestGen does not support default transitions which are interlevel.
  - The present version of TestGen adds default connectors to transitions entering states, during test data generation rather than gathering them together as described on p. 50. This approach is possible if a label is not used on more than one transition entering non-basic states.
  - Only one default transition in a statechart can have an empty trigger.
- Support for history, deep history and diagram connectors.
- Optimisation of the size of the set of test cases. This includes construction of C with every sequence in it of minimal length (Sect. 2.4.1 on p. 42). W could also be optimised as described in Sect. 2.4.2 on p. 43.
- Enhancement of the simulation capabilities of the tool. Here two possibilities exist: usage of the Statemate tool for simulation, and removal of limitations of the present simulator built as a part of TestGen (p. 223).
- Performance improvements, which may include rewriting parts of the tool in C++.
- Extension of the tool with:
  - addition of the transition tour method (Sect. 2.3.1 on p. 35) which generates the smallest test set compared with other testing methods.
  - generation of test sequences for an expanded statechart where transitions are split into a number of parts, using results of the category-partition method (Sect. 2.3.2 on p. 38).
  - incorporation of handling of a different semantics of statecharts, such as UML and Matlab/Simulink/Stateflow (the latter being a subject of the continuation project between the University of Sheffield and DaimlerChrysler).

Chapter 8

Case studies

In this chapter we show how the testing method developed can be applied to three case studies. Each of them demonstrate certain properties of the method, as outlined below

Speed This is a car control system where the speed of a vehicle is kept to the value set by the driver unless the distance between his car and the one in front falls below a certain threshold, at which moment the speed is reduced.

The statechart is rather simple and 'design for test' properties are satisfiable by an addition of a few testing inputs and outputs.

Air This is a part of an aircraft autopilot system.

The statechart is rather simple but the implementation contains a different set of transitions. Since the two sets of transitions are closely related, the testing method tests the system to the extent it was specified. The case study demonstrates testing of systems somewhat outside those allowed by testing requirements.

HI-FI This is a part of a hi-fi stereo system. It covers the overall control and describes the tape subsystem in detail. The Z description of transitions (40 pages) is not included in the thesis and design for test conditions related to labels are not considered.

Statecharts of the model contain many states and transitions violating testing requirements such as transitions with timeouts in them (App. B.3 on p. 273). The case study also demonstrates how a choice of refinement assumptions can reduce the size of a set of test cases by many orders of magnitude.

The first two case studies are rather confidential and for this reason the presentation of them was modified as to bear little resemblance to originals, at the same time preserving all problems and leading to the same conclusions.

8.1 Speed

8.1.1 Introduction

The Speed[1] system helps a driver to maintain a safe distance from a car in front. When the distance falls to less than a predefined constant, it slows the vehicle down; on a clear road the speed of movement can be set by a driver. The statechart of the control part of Speed is presented in Fig. 8.1. *init_speed* and *init_on* labels do not have a trigger.

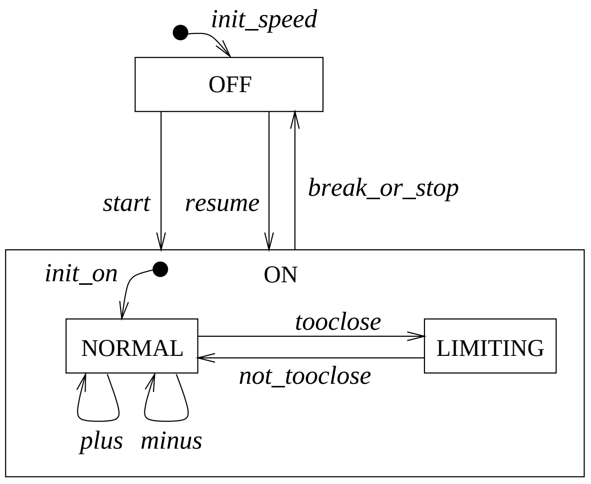


Figure 8.1: The Speed case study

The system can be activated (*start* or *resume*) and becomes inactive upon driver command or his usage of a *break* pedal or the *off* switch. When system is active, driver's preferred speed of movement is set by transitions *plus* and *minus*. If a car comes too close to the one in front of it, the *LIMITING* state is entered, which limits the speed of travelling to that of the vehicle in front. The difference between *start* and *resume* transitions is that *start* sets the desired velocity to the current one of the car and *resume* retains the value of it selected previously.

8.1.2 Test set generation

The case study has to be made compliant with requirements for testing; only t_completeness and output-distinguishability are not satisfied. In order to make Speed comply with them, a testing trigger and generation of a testing event were added to every transition (in the thesis we do not provide details as to why this was needed). Default ones were augmented with an output rather than left without any due to our decision to augment

[1]Essentially the same system is described in [Sad98] under the name of Automatic Cruise Control and in [GHD98], where it is called Cruise Control.

individual compound transitions of the full compound ones (Sect. 4.1.3 on p. 70). The result of augmentation is shown in Tab. 8.1.

Transition	**Trigger**	**Output**
init_speed	none	$\underline{Output_{Init}}$
start	$\underline{Trigger_1}$	$\underline{Output_1}$
resume	$\underline{Trigger_2}$	$\underline{Output_1}$
brake_or_stop	$\underline{Brake}$	$\underline{Output_1}$
init_on	none	$\underline{Output_1}$
plus	$\underline{Plus\_button}$	$\underline{Output_1}$
minus	$\underline{Minus\_button}$	$\underline{Output_1}$
tooclose	$\underline{Trigger_1}$	$\underline{Output_{proximity}}$
not_tooclose	$\underline{Trigger_1}$	$\underline{Output_{no\_proximity}}$

Table 8.1: Augmentation of the Speed system

The port with test input and output events is given below.

Speed

PORT *TESTPORT*

$Trigger_1$, $Trigger_2$: signal

$Output_{Init}$, $Output_{Init\_on}$, $Output_1$,
$Output_{proximity}$, $Output_{no\_proximity}$: signal

INPUT $Trigger_1$, $Trigger_2$

The test set consists of 27 sequences under assumption of no extra states in an implementation. Consideration of the structure in the *ON* state as an OR-state refinement (Sect. 3.2.3 on p. 52) allows us to reduce this to 14.

8.1.3 Conclusion

The Speed case study is rather simple and was used in Oct 96 as the first practical application of the X-machine testing method to a statechart. It demonstrated that satisfiability of the design for test conditions is not hard to achieve. OR-state refinement was shown to significantly reduce the size of the test set.

8.2 Air

8.2.1 Introduction

The subject of this case study is an application of the testing method to a part of an autopilot system using a high-level rather than detailed design of it. In addition, its behaviour is real-time which adds some complications to testing.

The part of the autopilot considered supports the following commands,

climb This command makes an airplane climb (increase the height of flight) by a certain number of meters. Execution of this command causes the plane to face slightly up, continue this way until it almost reaches the desired height and then level (change its position into horizontal again).

descent This command is similar to the *climb* one but causes a descent.

flaps on Deflects flaps in order to increase the lifting force on a plane.

flaps off Levels flaps such that they essentially do not affect the flight of the plane.

terminate Stops the command in progress. This is only applicable to the *climb* and *descent* commands and stops the execution by levelling the plane at the current height.

abort Stops the command currently executing without making any further changes to the controls. For the *climb* and *descent* commands it means that if, for example, the plane was climbing, it would continue doing that and it would be up to a pilot to level it. In case one of the flaps-related commands were active, flaps will be left in the intermediate position.

The operation currently being executed by the autopilot is indicated on the cockpit instrument panel.

8.2.2 Design and implementation

The statecharts of the design and the corresponding implementation are given in Fig. 8.2 and Fig. 8.3. The implementation statechart has been reverse-engineered from the implementation code the author was supplied with. The *OFF* state represents an idle autopilot; in the *OPERATING* one it performs a given task. Two different ways to cancel an execution of a given command, are provided by transitions *terminate* and *abort*, with the latter having precedence over the former. The control of the plane over time is split into multiple phases, represented by the *do_operating*, *do_terminating* and *do_aborting* transitions. Phases are changed by the remaining ones.

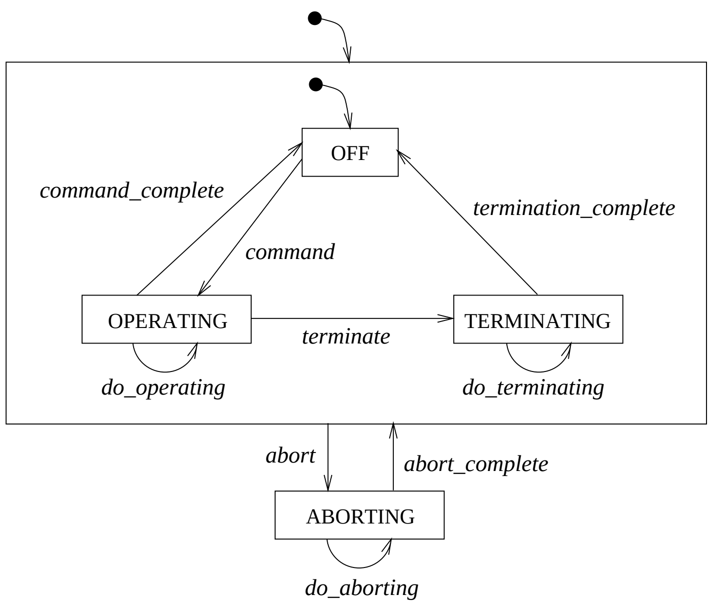


Figure 8.2: The design of the autopilot

From figures 8.2 and 8.3 it is clear that the implementation has different labels for *command*, *command_complete* and *terminate*. Specifically, the first two have been split into three, *climb_descent*, *flaps_on* and *flaps_off* and the corresponding completion transitions. The behaviour of the *terminate* transition is such that it can only be taken if climbing or descent is in progress. For this reason, it seems to be a good choice for it to be implemented as it was from the *CLIMB_DESCENT* state.

8.2.3 Testing requirements and test data generation

Statecharts describing the behaviour of the autopilot are sufficiently simple to satisfy most of the requirements of the design for test condition. The two we have to focus at are t_completeness and output-distinguishability.

Since the set of labels of transitions implemented is different from the designed one and some transitions are implemented with a different domain than designed, making the system t_complete and output-distinguishable assumes the following:

1. finding a correspondence between labels of the design and that of the implementation. This is not an easy task in general because an implementation may combine labels of transitions, making it not amenable to a direct application of the statechart testing method being described. In such cases we could reverse-engineer an implementation and model-check the behaviour of it w.r.t. a design or, simply, perform a partition-based testing of it. This area, though, is not covered

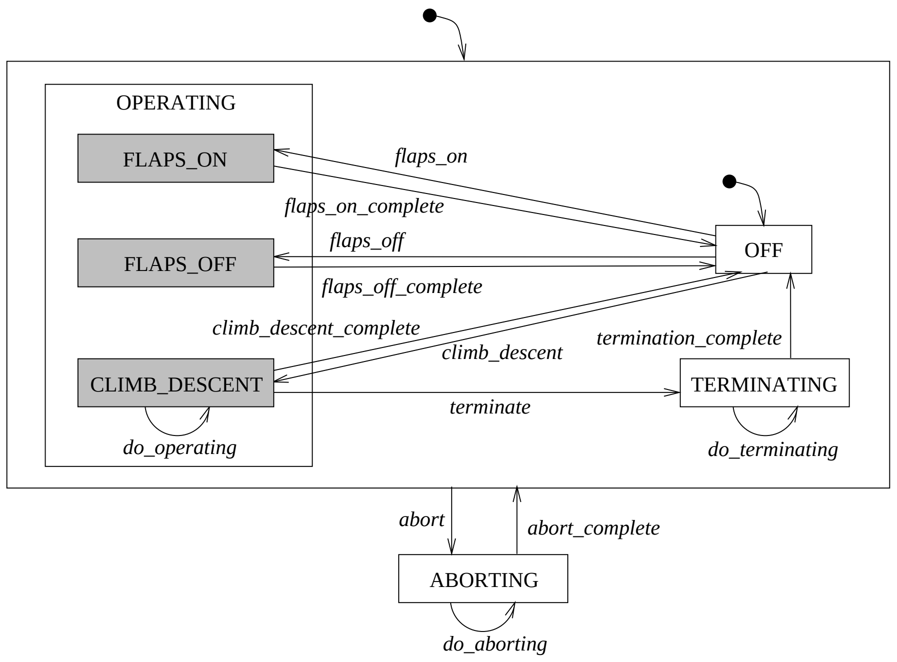


Figure 8.3: The implementation of the autopilot

in the thesis.

2. augmenting those labels, using the reverse-engineered implementation. Reverse-engineering is unnecessary for systems build with testing in mind. Unfortunately, in many cases implementations to test do not implement transitions directly; instead, labels are optimised for the intended behaviour. In such cases, we do need the code to determine triggers and/or outputs of transitions.

During test case generation, we get sequences of labels to trigger. Generally, we do not have to consider them during design for test since we can then always find appropriate test data. Often it means that for every transition we select the triggering input and an expected output in advance. In practice, however, minimal amount of augmentation is preferred and thus selection of test data could be attempted in such a way as to minimise the number of testing events, considering every specific sequence of labels used in the set of test cases. For instance, in this case study (Fig.8.2) we often have to follow the sequence *command terminate*. The definition of the latter is such that it will only occur if the command is *climb* or *descent*. For this reason, it is better to use *climb* as a trigger for *command* as this would allow usage of *terminate* as otherwise the *terminate* transition would have to be augmented with the testing input in order for us to take it after *command*. Note that it is only possible because *command* is the only transition which sets the internal data on which the trigger of *terminate* depends.

With the proposed trigger for *command*, we have that implementation is only tested to the extent it was designed, i.e. the *FLAPS_ON* and *FLAPS_OFF* states are not considered. This is not a limitation of the described testing method but failure of the implementation to follow requirements for the testing method. The method can nonetheless be applied but the scope of the implementation it covers is limited. In principle, selection of different triggers for different sequences of labels or by execution of every sequence with a different test data may allow us to test systems where implementation splits transitions according to input received. For example, we could try to convert test cases to test data using all of the $\underline{\textit{flaps on}}$, $\underline{\textit{flaps off}}$, $\underline{\textit{climb}}$ and $\underline{\textit{descent}}$ as triggers for *command*. While increasing the size of the test set, this would provide a more thorough testing for non-compliant implementations. In case something is known about transitions which can be split, we could try to utilise the method described in Sect. 2.3.2 on p. 40. Here we designate a part of a split transition as a main one (like *climb_descent* above) and test most of the transition structure using it. After that, the rest gets tested. Unfortunately, the autopilot implementation is such that some of the states (*FLAPS_ON FLAPS_OFF*) are unreachable when limiting ourselves to main transitions. It could be possible to derive some conditions allowing complete testing for such implementations; this work is left for future.

All completion labels (*command_complete*, *termination_complete* and *abort_complete*) are designed to occur after a specific time has elapsed. We can either wait long enough such that if any of them may occur, it will definitely do that or augment the *tm* function of the system, used to describe timeouts, in order to have a direct control over those labels (described in App. B.3 on p. 273 and referred to from Sect. 5.2.8 on p. 97). Unless *tm* is augmented, when triggering *do_command*, *do_termination* or *do_abort*, we have to do that fast such that no completion transitions become triggered.

An alternative way to distinguish states can be based on the observation of dashboard lights because the current operation of the autopilot is displayed there. Such observations do not require us to take any transitions, for example, we can always tell if we are in the *OPERATION* state, and using this $\underline{\textit{OPERATION\_LIGHT}}$ output, we can construct an optimised characterisation set. Usage of status is further described in App. C.4 on p. 282.

The considered system is operating in real time. For this reason, observations of input's effect on control surfaces of an aircraft could be used to distinguish states. For example, levelling and climbing require vastly different movements of appropriate surfaces, helping us to tell the *OPERATING* and *TERMINATING* apart. Here we have to observe behaviour over time, since levelling due to termination is likely to be done faster than that in normal operation, i.e. when climbing or decent is complete. This can be done by periodic sampling of the position of an appropriate surface and identifying the trend in its movement. Usage of Fourier transformation could help in this case; such state identification could be considered for future work.

The *abort* transition was implemented such that it will only occur when any of the surfaces is deflected. This makes sense but is not, strictly speaking, correct. Derivation of its real behaviour had to involve reverse-engineering of the implementation.

While we did not have to add testing inputs to all transitions, test outputs had to be added since the system exhibits continuous operation and reaction of it to test inputs has to identify transitions immediately while transitions changing phases of behaviour make no output and those performing deflection of surfaces do that rather slowly.

8.2.4 Conclusion

The test method developed for statecharts appeared to be applicable to a reasonably realistic system. The implementation of the given one appeared to be similar to the design, but sets of transition labels were different. These differences could be divided into two groups, labels being split into a number of using their inputs and labels implemented with different domains. The former prevented testing of the whole of the transition structure of the implementation but most of it was tested; for the latter it was impossible to derive test data from the design itself and implementation had to be used for that. In practical development of safety-critical systems, this would not be a major problem as deviations from a design would likely to be documented and these details combined with the design are expected to permit derivation of test data.

The case study has highlighted alternative approaches to state identification. Specifically, usage of indicators to verify states without taking any transitions was a new idea and inspired extensions of the testing method, see App. C on p. 276 for details. In addition, capture of real-time movement of control surfaces appeared to allow state distinguishing.

8.3 The model of the hi-fi stereo

Here the simplified model of a hi-fi tape deck will be presented which can be viewed as a vastly extended version of that in Fig. 1.6. It will be used later to show how its design can be used to generate a set of test cases, how it can be modified to comply with testing requirements, and the effect different refinement assumptions make on the size of the set of test cases.

The description is limited to the statecharts of the model since a Z design of transitions is only necessary in order to verify the design for test condition. Since applicability of this condition was illustrated by the case studies given above, no work was done on this for the hi-fi. We also slightly simplified the model by excluding static reactions.

8.3.1 Introduction

The prototype for the model is the author's AIWA(TM)NSX-V50 hi-fi stereo. The original can play compact discs, tapes, has an FM/MW/LW radio and a sound input which could be used to amplify a TV, video or any other external sound source. The stereo contains two tape decks both of which are auto-reverse; the second of them (also called deck B) can be used for recording. It has a smarter auto-reverse than the deck A. Specifically, it is possible to select if it will change a side, stop or select deck A for a playback when a tape has stopped. When listening to a tape, it is possible to skip a song in both directions.

The control panel of the model is presented in Fig. 8.4. The main features of the model are described below:

- Only tape decks and an auxiliary input are modelled. Tape decks include almost all features of the original and the auxiliary input generates a saw-shaped signal. In the future we may consider expanding the model with extra functionality.
- Karaoke button and microphone input as well as headphone output are not included (simulating headphones will require us to switch speakers off when the plug is inserted as well as filter out all low and high frequencies of audio signal).

The model can be represented by a top-level data flow diagram shown in Fig. 8.5 with the following elements:

`CIRCUIT_TAPE_PLAY`, `CIRCUIT_TAPE_RECORD` represent the two tape decks of the model.

`CIRCUIT_AUX` represents an auxiliary input. In the model, it just produces a saw-shaped signal.

STEREO_MAIN is the controller which manages the behaviour of the stereo. This is essentially the core of the model. It receives commands from a user, from different parts of the model and manages the tape.

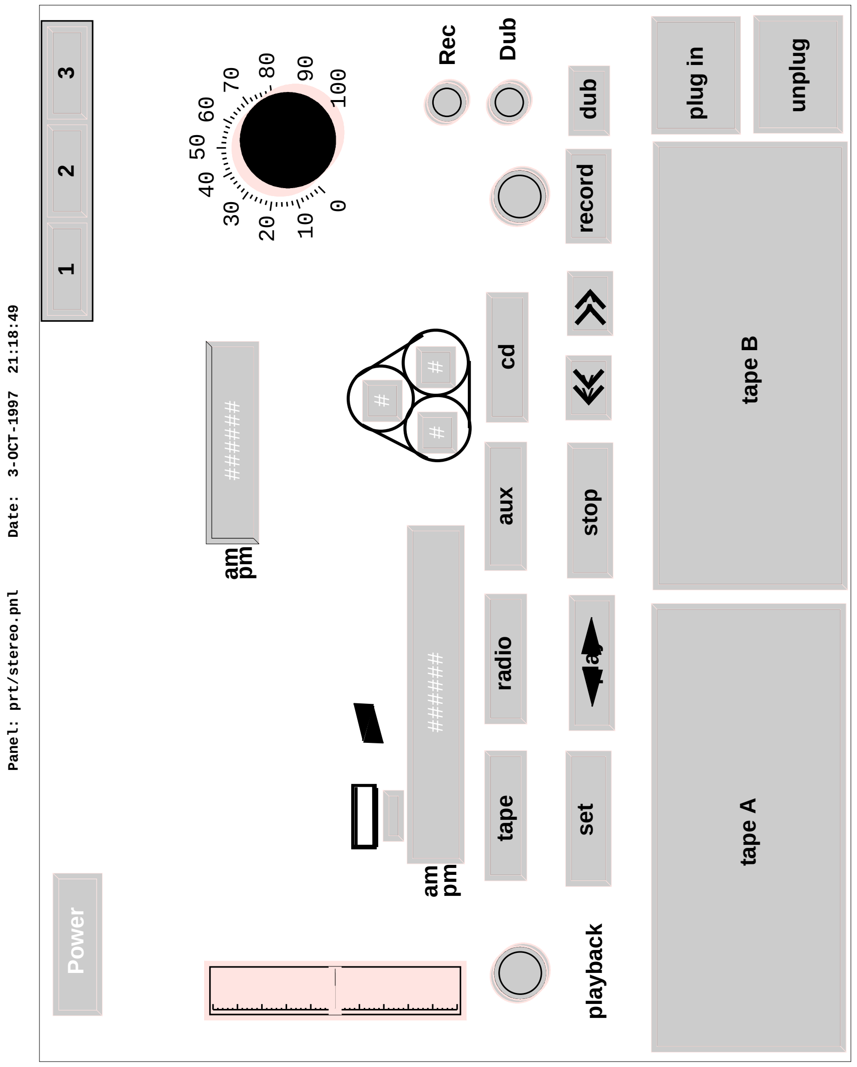


Figure 8.4: The control panel of the hi-fi

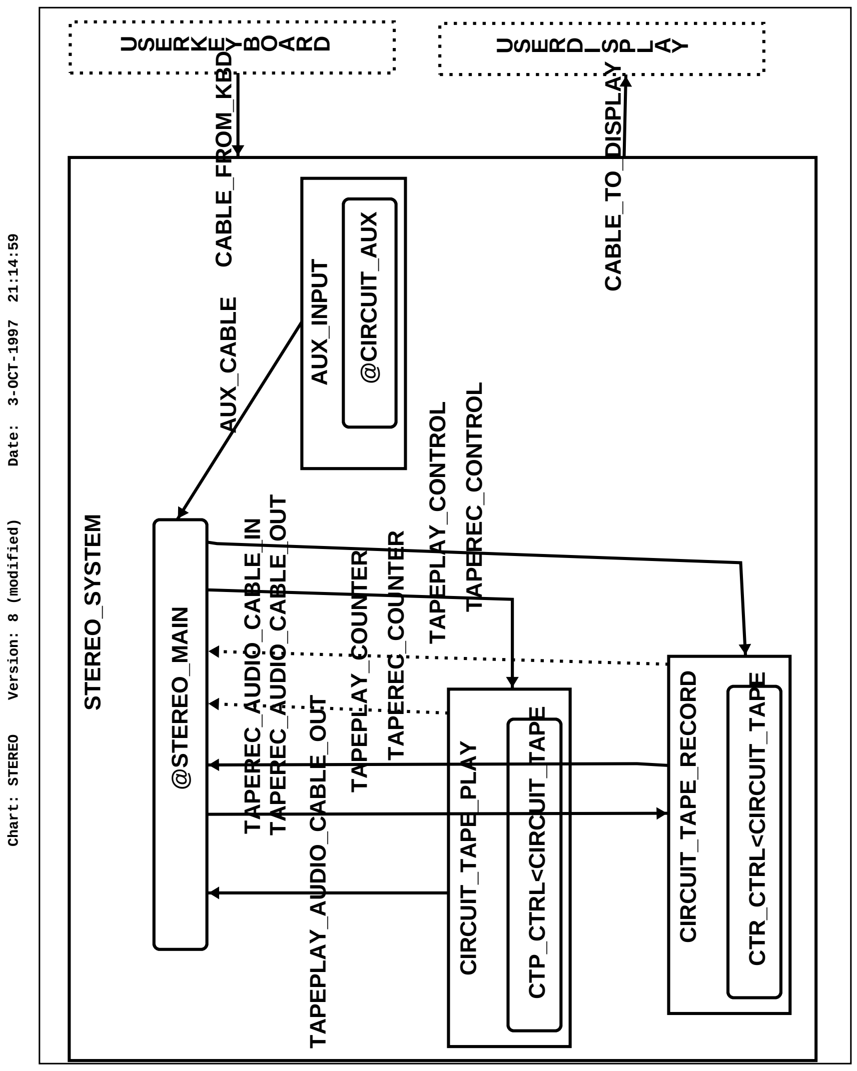


Figure 8.5: The data-flow diagram of the model

The tape deck (`CIRCUIT_TAPE`)

The behaviour of the tape deck is represented by the statechart shown in Fig. 8.6. `Circuit_tape` is a generic statechart. We instantiate it twice to represent a playback-only (`circuit_tape_play`) and recording-capable (`CIRCUIT_TAPE_RECORD`) decks. The physical tape is modelled as a big array, to hold sides A and B in mono. It is hard to model a continuous tape, thus we discreticise it with the sampling rate of `AUDIOSAMPLINGFREQUENCY`. The higher the `AUDIOSAMPLINGFREQUENCY`, the more accurate the model is, but it also takes more time to do the simulation.

The tape deck can be given the following commands:

`tape_direction` — which direction to play, `tapedirection_ff` or `tapedirection_rew`.

`tape_command` — either of the following four commands: `tapecommand_move`, `tapecommand_play`, `tapecommand_record`, `tapecommand_stop`.

`tape_headposition` — which side to play, `tapeside_a`, `tapeside_b`.

`tape_inserted` — whether a tape is inserted or not; we assume that only one tape cartridge may be used.

The following outputs are generated:

`tape_counter` — counter pulses.

`tape_signal` — the sound. It is used as an output during playback and as an input during recording.

The notation used for transitions in the rest of this chapter is [*trigger*]/*action* rather than the one used elsewhere in the thesis. The difference is due to the model being compatible with Statemate notation (refer to Sect. 8.3.5 on p. 260). For this reason all triggers are surrounded with square brackets.

An auxiliary input (`CIRCUIT_AUX`)

The statechart is shown in Fig. 8.7. The only transition with a label generates an appropriate signal. The guard of it is the $preAUX\_ACTION$ schema and AUX_ACTION is the action. We use a similar notation for many other transitions of the model.

The controller ($STEREO\_MAIN$)

The main statechart is the $STEREO\_MAIN$ statechart, shown in Fig. 8.8. It contains a number of off-page statecharts, referred to by using '@' in front of their names. For instance, $STEREO\_TAPE$, contained in the appropriate state, is written as $@STEREO\_TAPE$. Further, we would occasionally refer

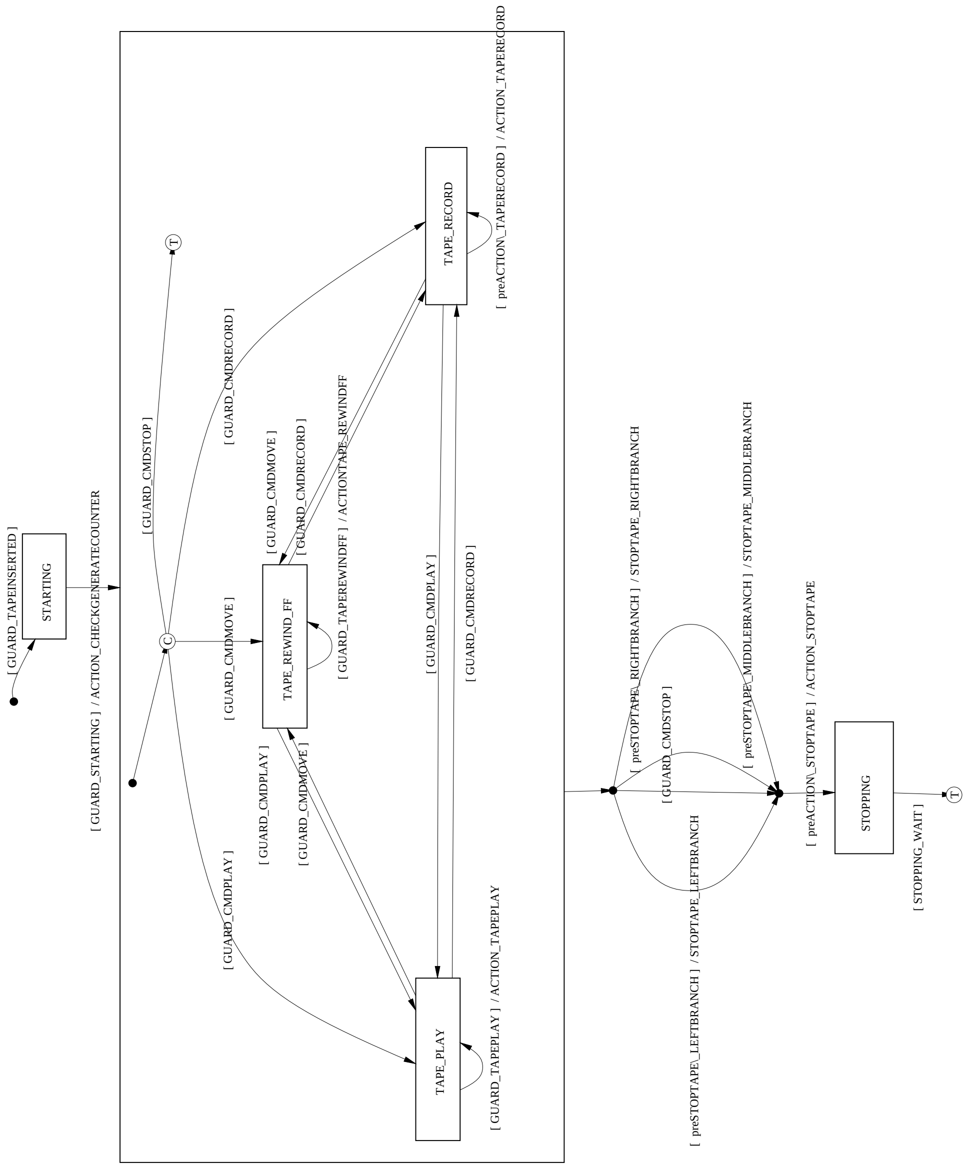

[GUARD_TAPEINSERTED]
STARTING
[GUARD_STARTING] / ACTION_CHECKGENERATECOUNTER
C
T
[GUARD_CMDSTOP]
[GUARD_CMDRECORD]
[GUARD_CMDMOVE]
[GUARD_CMDPLAY]
[GUARD_CMDMOVE]
[GUARD_CMDRECORD]
[GUARD_TAPEREWINDFF] / ACTIONTAPE_REWINDFF
TAPE_REWIND_FF
[GUARD_CMDPLAY]
[GUARD_CMDMOVE]
[GUARD_CMDPLAY]
[GUARD_CMDPLAY]
[GUARD_CMDRECORD]
TAPE_RECORD
[preACTION_TAPERECORD] / ACTION_TAPERECORD
TAPE_PLAY
[GUARD_TAPEPLAY] / ACTION_TAPEPLAY
[preSTOPTAPE_RIGHTBRANCH] / STOPTAPE_RIGHTBRANCH
[GUARD_CMDSTOP]
[preSTOPTAPE_LEFTBRANCH] / STOPTAPE_LEFTBRANCH
[preSTOPTAPE_MIDDLEBRANCH] / STOPTAPE_MIDDLEBRANCH
[preACTION_STOPTAPE] / ACTION_STOPTAPE
STOPPING
[STOPPING_WAIT]
T

Figure 8.6: The statechart of the tape deck (CIRCUIT_TAPE)

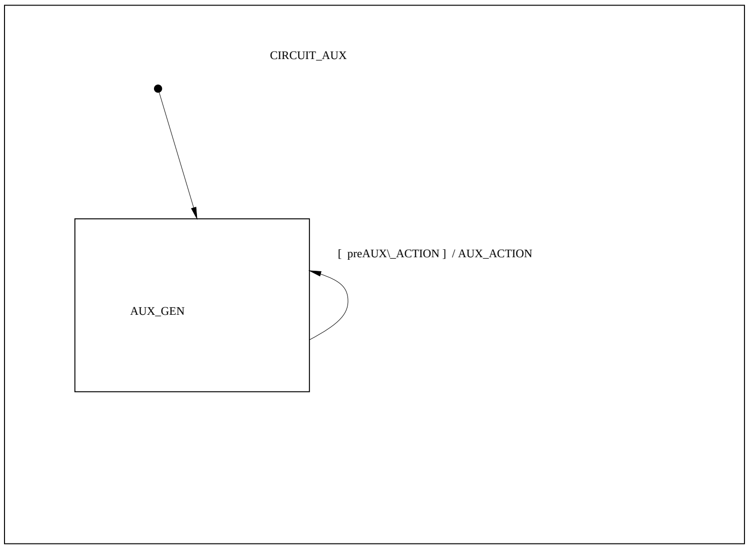


Figure 8.7: An auxiliary input

to such statecharts as states. Some of the statecharts mentioned in Fig. 8.8 are not present in the model. They are shown to describe where they would go if the complete model of the hi-fi stereo were built.

The state hierarchy tree is provided below with the help of indentation; states mentioned are described later.

> *STEREO_CD*. This statechart is supposed to deal with playing a compact disc but the functionality of it is absent.
>
> *STEREO_AUX* (Fig. 8.11). Auxiliary input amplifier.
>
> *STEREO_TAPE* (Fig. 8.15). The statechart responsible for most tape deck operations.
>
>> *PLAYING_TAPES* (Fig. 8.16). Controls tape playback and auto reverse.
>>
>> *MUSIC_SENSOR_OPERATION* (Fig. 8.17). Provides music sensor functionality. It allows one to rewind/fwd. advance a tape to the next song.
>
> *STEREO_RADIO*. A radio receiver is absent in the model.
>
> *CD_CHANGE*. This part of the model is expected to deal with changing a disc, i.e. opening/closing the tray and rotating it. It is absent as well.
>
> *STEREO_TIMER* (Fig. 8.12). Advances a clock and allows to set it.

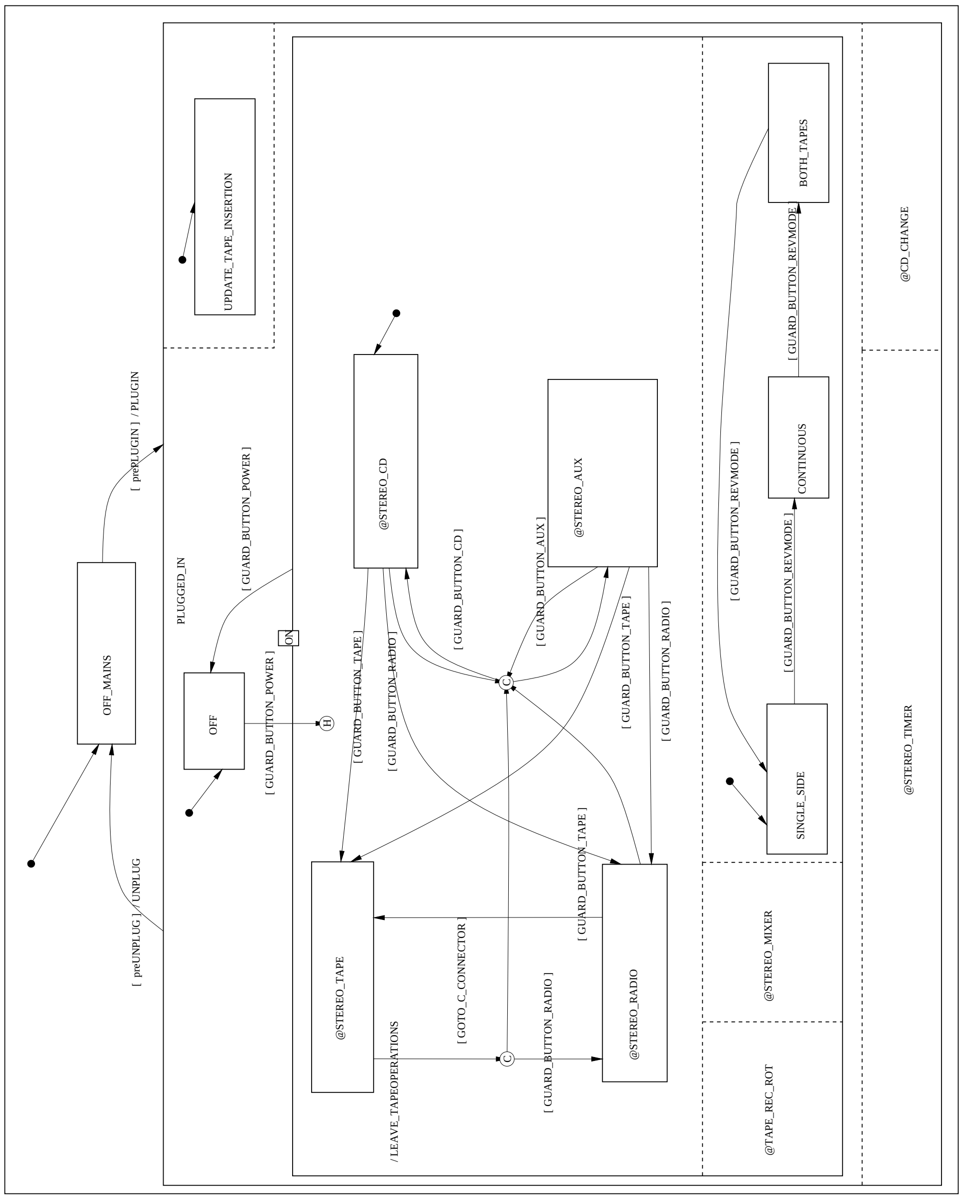


Figure 8.8: The main statechart of the model (*STEREO_MAIN*)

CLOCK_SETTING_STATE(Fig. 8.13). Responsible for clock setting.

DISPLAY_TIME(Fig. 8.14). It is an auxiliary statechart used to display time with possible blinking of an hour or a minute.

STEREO_MIXER(Fig. 8.10). Performs volume control. In a complete model it would also allow selection of a sound source and support an equaliser.

TAPE_REC_ROT(Fig. 8.9). Deals with tape rewinding/f. advancing, dubbing and recording from an auxiliary input.

The *STEREO_MAIN* state contains states responsible for the model being plugged into the mains (the *PLUGGED_IN* state) and being switched on (the *ON* state). While stereo is in the *ON* state, it can either be ready to play tapes (*STEREO_TAPE*), discs (*STEREO_CD*), radio (*STEREO_RADIO*) or just amplify a signal from an external source (*STEREO_AUX*). Each of them is represented by a separate statechart. Additionally, the model handles tape insertion/removal (state *UPDATE_TAPE_INSERTION*). While listening to radio, external signal or playing a compact disc, it is possible to record the sound on a tape. Apart from recording, one can also perform a forward advance, a rewind or dubbing from tape to tape. Due to tape operations being possible in *STEREO_RADIO*, *STEREO_AUX* and *STEREO_CD* states, the *TAPE_REC_ROT* statechart (Fig. 8.9) is concurrent to others in the *ON* state.

The tape reversal mode is selectable independently. It governs what happens if a tape in deck B has reached the end of it during playback. Hi-fi can either stop the playback, continue from a different side or switch to another deck.

Discs can also be changed at any time thus the *CD_CHANGE* statechart is concurrent to others in the *PLUGGED_IN* state too.

STEREO_MIXER (Fig. 8.10) is supposed to update the information on the equaliser display and perform the modifications of the audio signal with respect to user equaliser settings as well as those for the volume and surround. It is presently limited to volume control.

STEREO_TIMER deals with updating the system time and allowing it and the alarm to be set; the statechart for it is shown in Fig. 8.12.

STEREO_MIXER and *STEREO_AUX* statecharts

We shall first consider the most simple of them, the *AUDIO_MIXER*, shown in Fig. 8.10. It contains two states: *MUTE* and *PLAYING*. When the model is switched on, it enters the *MUTE* state and after a short while — the *PLAYING* one. The delay is used on the real system to prevent a click in loudspeakers when the device is being turned on and is initially unstable. In

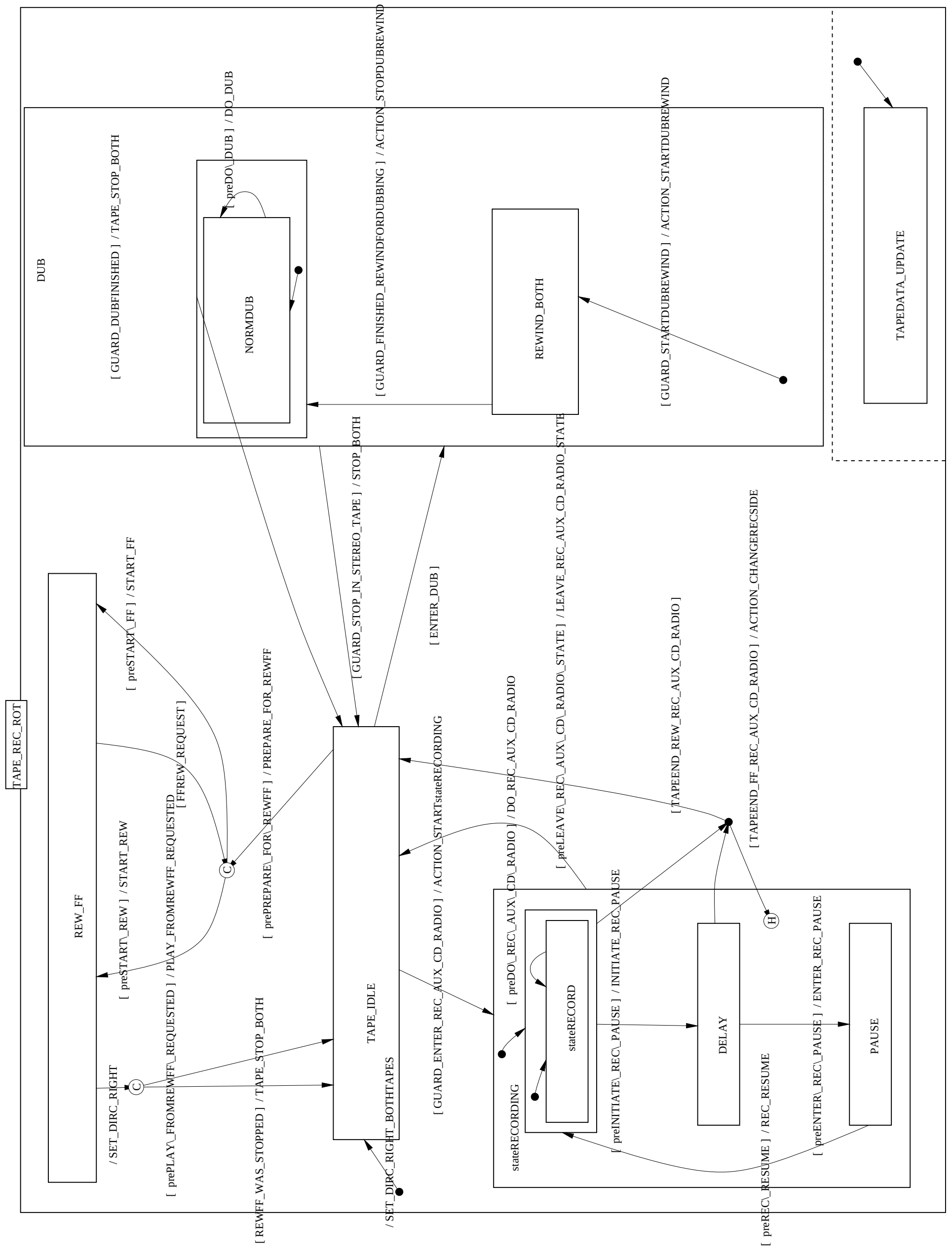

TAPE_REC_ROT
DUB
[GUARD_DUBFINISHED] / TAPE_STOP_BOTH
[preDO_DUB] / DO_DUB
NORMDUB
[GUARD_FINISHED_REWINDFORDUBBING] / ACTION_STOPDUBREWIND
REWIND_BOTH
[GUARD_STARTDUBREWIND] / ACTION_STARTDUBREWIND
TAPEDATA_UPDATE
REW_FF
[preSTART_FF] / START_FF
[FFREW_REQUEST]
[prePREPARE_FOR_REWFF] / PREPARE_FOR_REWFF
[GUARD_STOP_IN_STEREO_TAPE] / STOP_BOTH
[ENTER_DUB]
[preSTART_REW] / START_REW
PLAY_FROMREWFF_REQUESTED
[prePLAY_FROMREWFF_REQUESTED] / PLAY_FROMREWFF_REQUESTED
/ SET_DIRC_RIGHT
[REWFF_WAS_STOPPED] / TAPE_STOP_BOTH
/ SET_DIRC_RIGHT_BOTHTAPES
TAPE_IDLE
[GUARD_ENTER_REC_AUX_CD_RADIO] / ACTION_STARTstateRECORDING
[preDO_REC_AUX_CD_RADIO] / DO_REC_AUX_CD_RADIO
[preLEAVE_REC_AUX_CD_RADIO_STATE] / LEAVE_REC_AUX_CD_RADIO_STATE
[TAPEEND_REW_REC_AUX_CD_RADIO]
[TAPEEND_FF_REC_AUX_CD_RADIO] / ACTION_CHANGERECSIDE
stateRECORDING
stateRECORD
[preINITIATE_REC_PAUSE] / INITIATE_REC_PAUSE
DELAY
[preENTER_REC_PAUSE] / ENTER_REC_PAUSE
PAUSE
[preREC_RESUME] / REC_RESUME
C
C
H

Figure 8.9: The statechart describing operations on a tape which could be done concurrently with listening to radio, CD or auxiliary input

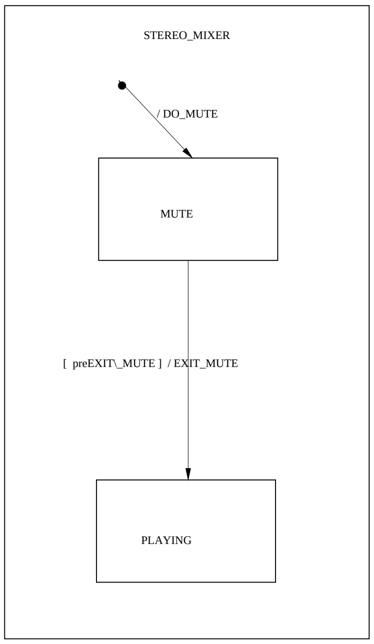


Figure 8.10: The mixer

the *PLAYING* state the statechart takes the **audio_signal** and generates **audio_output** depending on the volume. This behaviour is expressed by a static reaction not shown on the figure.

The statechart *STEREO_AUX* (Fig. 8.11), which is responsible for amplifying an external signal, actually passes the input a saw-shaped signal from **CIRCUIT_AUX** to the model, which can be recorded on a tape as well as listened to.

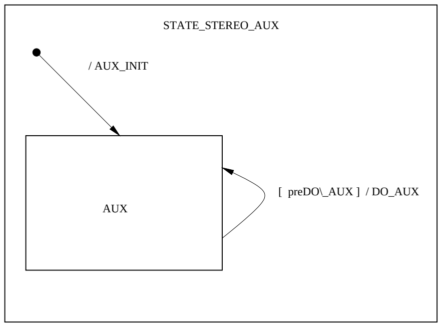


Figure 8.11: An auxiliary input part of the stereo

STEREO_TIMER

The statechart is shown in Fig. 8.12. It is responsible for clock increment,

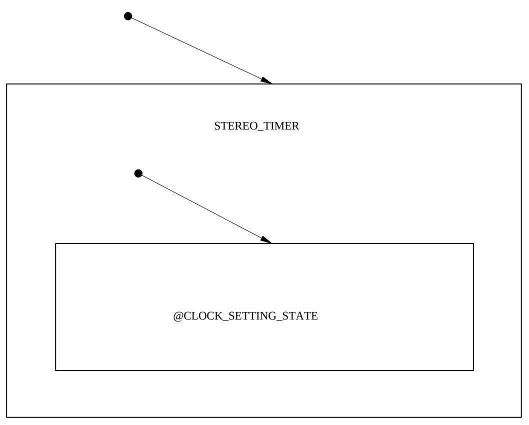


Figure 8.12: The timer

performed within a static reaction. The *CLOCK_SETTING_STATE* state displays and allows to modify the time. It is shown in Fig. 8.13.

We have to consider displaying time when the stereo is off (in the *OFF* state) and when it is on. When off, time should be shown with reduced brightness (state *DISPLAY_TIME_WHEN_OFF*); when on, it should usually not be displayed (state *DISPLAY_NOTHING*). Upon request (the special *clock* button on a remote control) time should appear (the *DISPLAY_TIME* state).

While time is displayed, clock alteration mode is entered by a depress of the *set* button and then time changed with << and >> to set an hour followed by *set* and similarly for a minute. The *stop* button selects 12/24 hour clock.

In clock alteration mode, an hour or a minute blinks. This and time indication are encapsulated in the *DISPLAY_TIME* generic statechart, shown in Fig. 8.14. This statechart contains a single state with a static reaction performing the task.

Tape operations (*STEREO_TAPE*)

This is the most elaborate and complicated part of the model. The tape controller is modelled to a rather high level of detail including the music sensor operation. Its main functionality is located in the following two statecharts, *STEREO_TAPE*, shown in Fig. 8.15 and *PLAYING_TAPES* — Fig. 8.16 (diagram connectors are numbered and drawn as ovals). The former contains the *IDLE* state and those corresponding to manual tape dubbing (*TAPE_DUB*) and erasing (*REC*). The tape erasing state is called *REC* because in the real stereo it deals with recording from microphone. As one is absent from the hi-fi model, a tape is just erased. Automatic tape dubbing is supported in the *TAPE_REC_ROT* statechart, Fig. 8.9, in the *DUB* state.

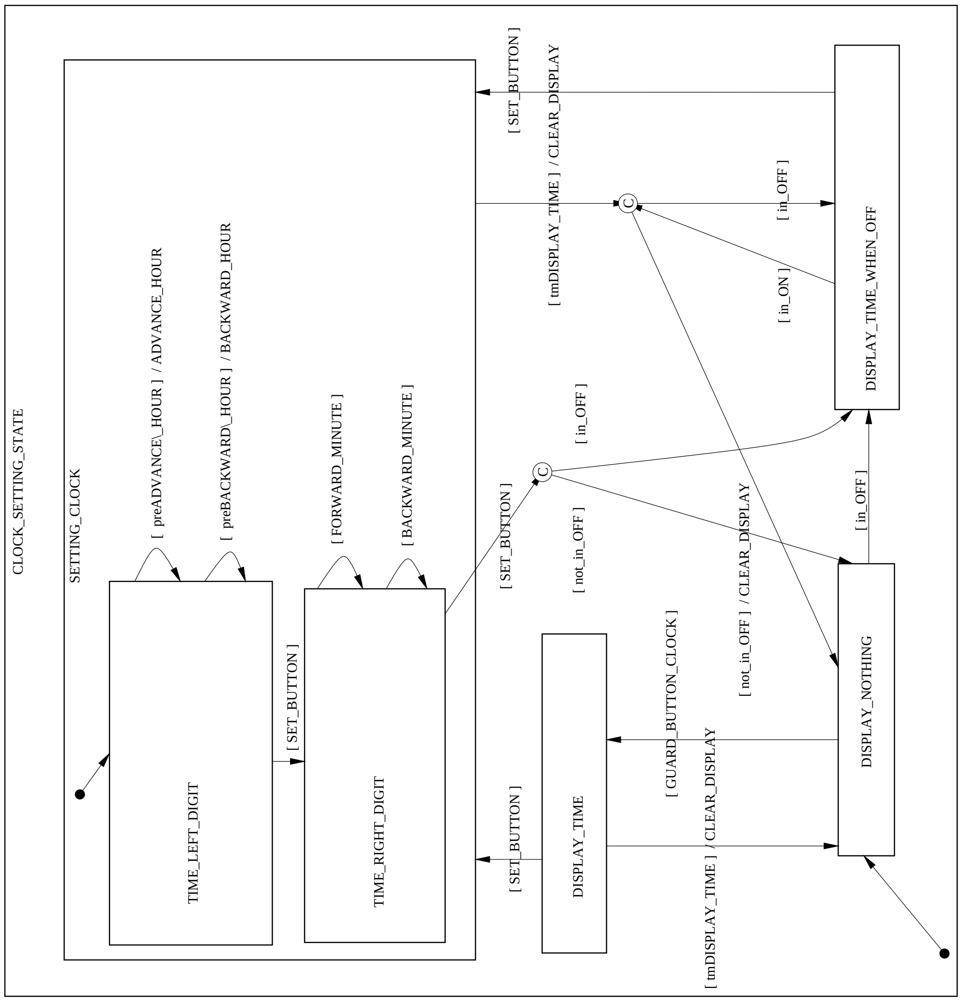


Figure 8.13: The statechart allowing to display and modify the current time

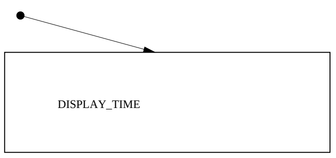


Figure 8.14: The generic statechart which displays time

The *PLAYING_STATES* statechart is responsible for playing tapes. It contains two states *PLAY_1* and *PLAY_2*, representing a tape being played in the first and second decks respectively. When a tape has stopped not due to user action, it can decide to reverse it or switch to another deck. The state without a name is an intermediate one. It deals with the fact that switching to another tape requires a change in `tape_select` variable (which holds the number of the currently active tape) and initialisation of the new tape drive, to side A forward direction. The process of switching is thus two-step. We first change the value of `tape_select` and then supply new parameters to the current deck. Since within every step values of variables are as at the beginning of a step and we wish to modify `tape_select` and perform operations based on its new value, a new step has to be started and the unnamed state is just for that.

Following [Har87] we use history connectors to express loopback transitions (i.e. those leaving and then entering the same state), present in every substate of a state. It means we have them leaving every substate (such as *PLAY_1* and *PLAY_2*), do something (say, change a side) and then return to the state left via a history connector.

The state *DUB* of *TAPE_REC_ROT* (Fig. 8.9 on p. 245) controls the automatic dubbing of the whole tape. It starts with rewinding both tapes to the beginning (state *REWIND_BOTH*) and then does the recording (state *NORMDUB*). If the model of a tape drive were supporting high-speed playback/recording, *HIDUB* state would be included. The *NORMDUB* state is placed as a substate. This is necessary to express that stopping of a tape is of a higher priority than recording.

The *REW_FF* state deals with tape rewind/fwd. advance. As stated before, its behaviour can be executed in parallel to *STEREO_RADIO*, *STEREO_CD*, *STEREO_AUX* and *STEREO_TAPE*. We could start rewinding and switch to listen to something other than a tape or we may try to rewind in order to skip the current song. In the latter case the musical sensor mechanism comes into play.

The music sensor is described in the *MUSIC_SENSOR_OPERATION* statechart, shown in Fig. 8.17. If invoked when there is no music, the pause is skipped first. When music begins, we wait for it to finish. Such waiting is accomplished via timeouts: if we are in the *WP_NOSIGNAL* state for a while, it means that there is no signal for that period of time. Thus the time-

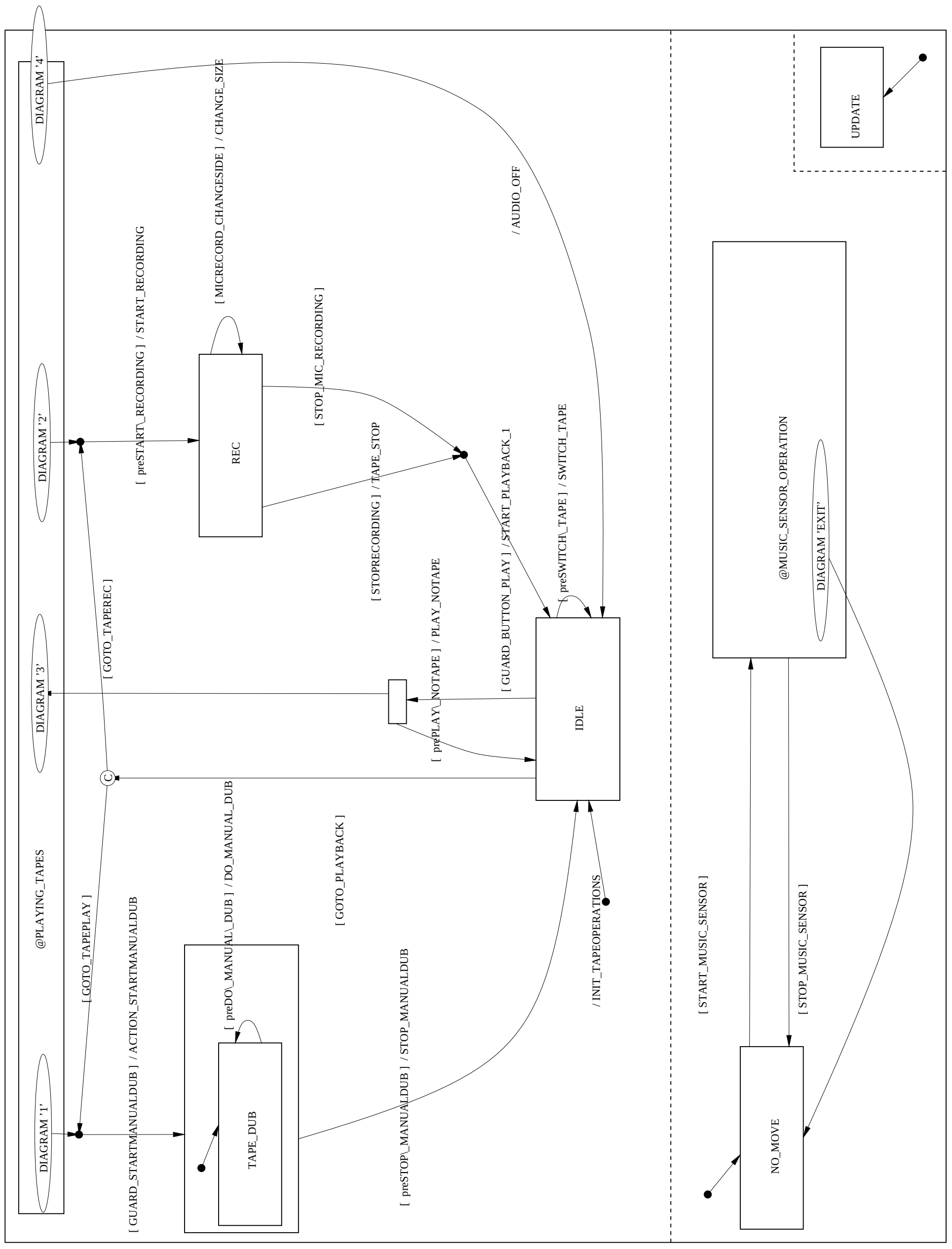


DIAGRAM '4'
DIAGRAM '2'
DIAGRAM '3'
@PLAYING_TAPES
DIAGRAM '1'
[GOTO_TAPEREC]
C
[GOTO_TAPEPLAY]
[preSTART_RECORDING] / START_RECORDING
[MICRECORD_CHANGESIDE] / CHANGE_SIZE
REC
[STOP_MIC_RECORDING]
[STOPRECORDING] / TAPE_STOP
[prePLAY_NOTAPE] / PLAY_NOTAPE
[GUARD_BUTTON_PLAY] / START_PLAYBACK_1
[preSWITCH_TAPE] / SWITCH_TAPE
/ AUDIO_OFF
IDLE
[GUARD_STARTMANUALDUB] / ACTION_STARTMANUALDUB
[preDO_MANUAL_DUB] / DO_MANUAL_DUB
[GOTO_PLAYBACK]
TAPE_DUB
[preSTOP_MANUALDUB] / STOP_MANUALDUB
/ INIT_TAPEOPERATIONS
[START_MUSIC_SENSOR]
[STOP_MUSIC_SENSOR]
@MUSIC_SENSOR_OPERATION
DIAGRAM 'EXIT'
NO_MOVE
UPDATE

Figure 8.15: The *STEREO_TAPE* statechart

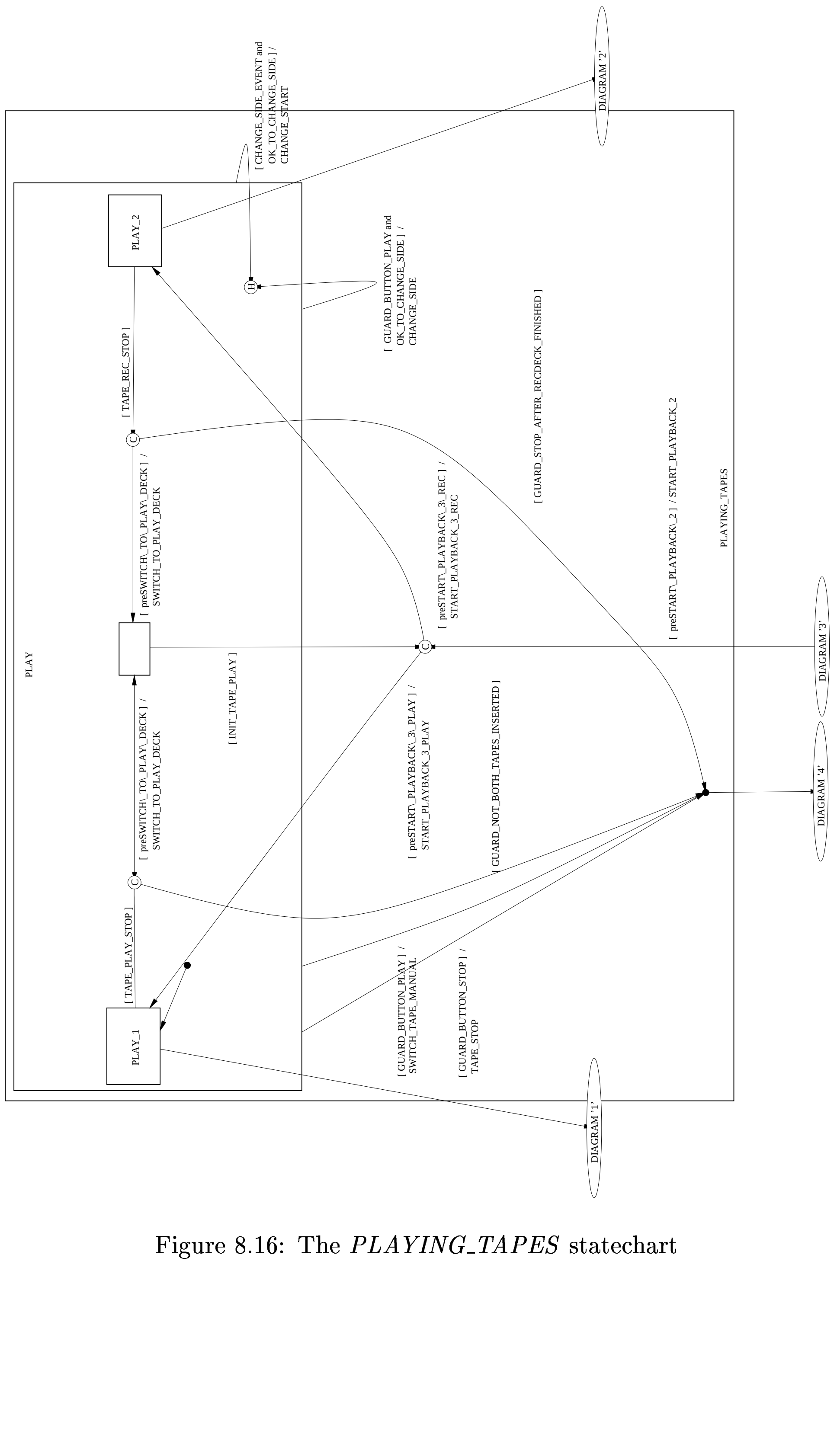

PLAYING_TAPES
PLAY
PLAY_2
PLAY_1
H
C
C
C
DIAGRAM '1'
DIAGRAM '2'
DIAGRAM '3'
DIAGRAM '4'
[CHANGE_SIDE_EVENT and OK_TO_CHANGE_SIDE] / CHANGE_START
[GUARD_BUTTON_PLAY and OK_TO_CHANGE_SIDE] / CHANGE_SIDE
[GUARD_STOP_AFTER_RECDECK_FINISHED]
[preSTART_PLAYBACK_2] / START_PLAYBACK_2
[TAPE_REC_STOP]
[preSWITCH_TO_PLAY_DECK] / SWITCH_TO_PLAY_DECK
[preSTART_PLAYBACK_3_REC] / START_PLAYBACK_3_REC
[INIT_TAPE_PLAY]
[preSWITCH_TO_PLAY_DECK] / SWITCH_TO_PLAY_DECK
[preSTART_PLAYBACK_3_PLAY] / START_PLAYBACK_3_PLAY
[GUARD_NOT_BOTH_TAPES_INSERTED]
[TAPE_PLAY_STOP]
[GUARD_BUTTON_PLAY] / SWITCH_TAPE_MANUAL
[GUARD_BUTTON_STOP] / TAPE_STOP

Figure 8.16: The *PLAYING_TAPES* statechart

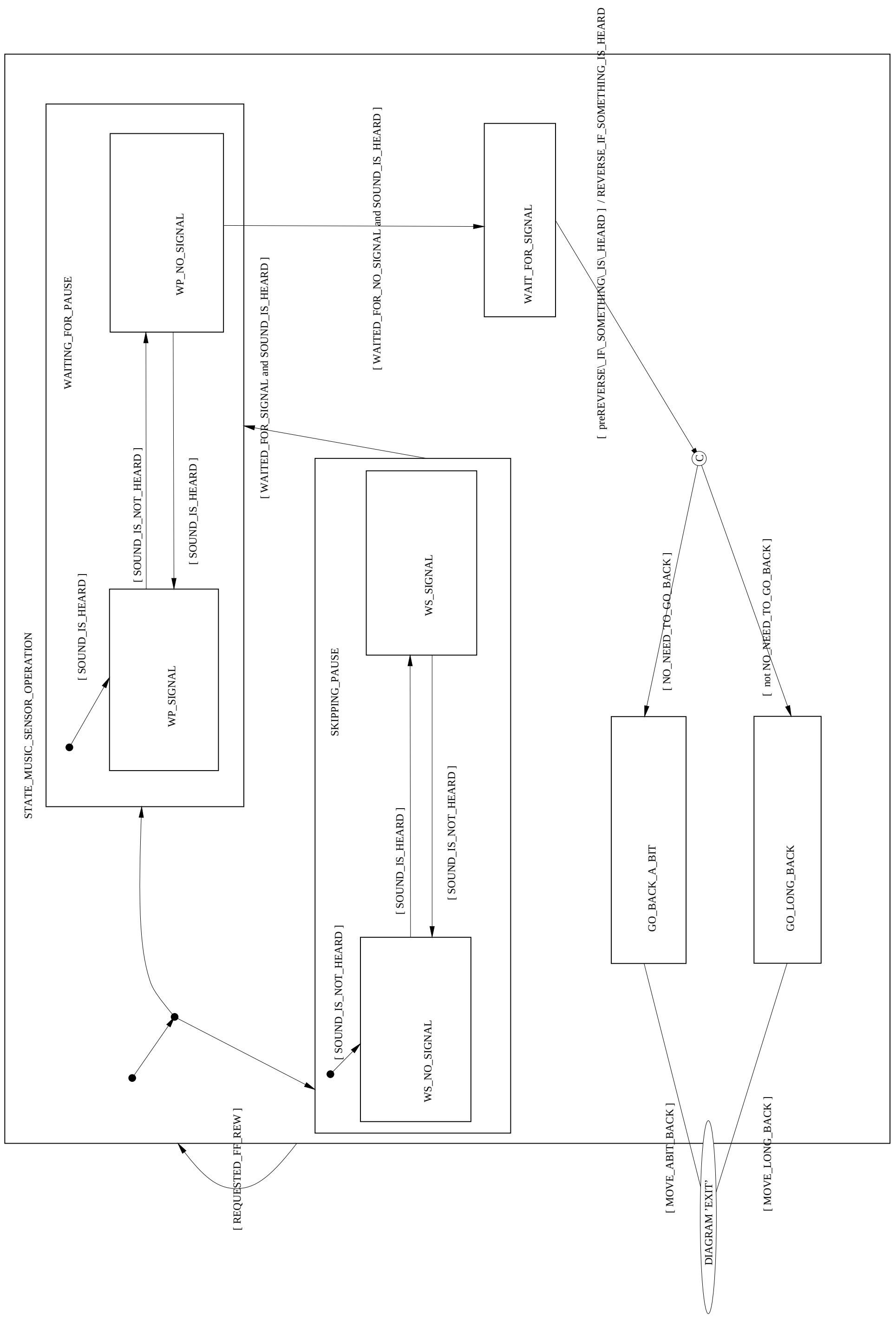


Figure 8.17: The music sensor

out occurs and we enter the *WAIT_FOR_SIGNAL* state where we are waiting for another song to begin. After it happens, it may be necessary to rewind back, as expressed by states *GO_BACK_A_BIT* and *GO_LONG_BACK*. The appropriate one is chosen based on the current side and direction of music searching.

8.3.2 Testing requirements

Since different components of the overall system (Fig. 8.5) are independent, they can be tested separately. The case study focuses at the `stereo controller` since it contains almost all statecharts the hi-fi consists of, at least those with a nontrivial behaviour.

In this section we consider structural requirements the model has to satisfy. Many of them appeared not to be satisfied by the hi-fi. The revealed inconsistencies between its design and testing requirements were picked by the TestGen tool.

The following transformations of the model were performed to make it follow the testing requirements:

- All **C** connectors in the model were removed and equivalent compound transitions constructed.

- States *STEREO_CD* and *STEREO_AUX* appeared to be behaviourally-equivalent since we only observe output from transitions rather than some indicator telling us which state we are it. On a real device, a display "tape radio aux cd" in Fig. 8.4 will indicate the major mode. Such indication we call *status* and consider using it for state identification in App. C.4 on p. 282.

  For the model, the transition between *STEREO_CD* and *STEREO_TAPE* was removed and a status loop transition added to *STEREO_TAPE* to prevent it becoming equivalent to *STEREO_CD*. Extension of the tool to support status could be considered for future work.

- Loopback transitions were added to states *SINGLE_SIDE* and *CONTINUOUS* to distinguish between them. Status would be useful in this case too since the state is displayed on the panel as well.

- State *ON* in Fig. 8.8 is unreachable without taking an interlevel transition from *OFF* because the *[GUARD_BUTTON_POWER]* transition enters a history connector. This connector was removed and transition expanded into four separate transitions, each entering the respective state.

  Similar problem occurs in Fig. 8.16 where the *PLAY* state is unreachable without usage of interlevel transitions. Here the cause is that this state was introduced to conceptually group states *PLAY_1* and

PLAY_2, rather than to express a hierarchy (this is called 'abbreviation' in [Sim00]). To remedy this problem, a transition from the unnamed state at the top of the *IDLE* one in Fig. 8.15, to *PLAY_1* was introduced.

- In states shown in Fig. 8.9 and Fig. 8.16, a history connector was replaced by a number of transitions (this connector did not introduce unreachable states, though). No testing of history connectors, described in Sect. 3.5 on p. 60, was done.

- Labels of loopback transitions leaving states *NORMDUB*, *stateRECORD* (Fig. 8.9) and *TAPE_DUB* (Fig. 8.8) correspond to the tape recorder recording data; these transitions are taken once in a while and with each invocation a fixed chunk of data gets recorded. The loopback transitions appeared to be shared because timeouts on all of them were the same and were for this reason changed to become different. When testing these transitions, we assume that we can trigger timeouts as if they were ordinary events, utilising the approach described in App. B.3.2 on p. 275.

  An alternative approach would be to augment these transitions such that during testing they are triggered by a different input depending on a state they are used in. This approach is described in Sect. 5.2.4 on p. 89.

  Similar problem with shared transitions occurred in the statechart in Fig. 8.17, where the output from a tape is compared with zero in *SOUND_IS_HEARD* and *SOUND_IS_NOT_HEARD* transitions. Consequently, multiple different 'zero' constants were introduced. The new zero variables could be chosen sufficiently close to zero such as not to affect the behaviour not under test. This approach to augmentation is outlined in App. B.2 on p. 273.

- In the statechart shown in Fig. 8.13, transitions labelled *button_set* are shared. For this reason, the one from *SETTING_CLOCK* state was renamed to *button_set1*.

- Substates *TIME_LEFT_DIGIT* and *TIME_RIGHT_DIGIT* are instantiations of generic statecharts and in accordance with Sect. 3.6 on p. 61 are tested as an OR-state refinement.

- In the statechart in Fig. 8.17, transitions to enter *WAITIING_FOR_PAUSE* and *SKIPPING_PAUSE* states have no trigger; the one to fire is determined by triggers on default transitions in the corresponding states. The method considers such a statechart as nondeterministic since the only thing visible without looking inside the two states is the default connector with two transitions leaving it. For this reason, the inner transitions have been moved to start from the default connector of state *STATE_MUSIC_SENSOR_OPERATION*.

Since the above change caused the newly-constructed default transitions to be defined the same way as *SOUND_IS_HEARD* and *SOUND_IS_NOT_HEARD*, more different 'zero' constants were introduced.

- In the statechart in Fig. 8.17, many interlevel transitions are used. This causes the state *WAIT_FOR_SIGNAL* to become unreachable and states *GO_BACK_A_BIT* and *GO_LONG_BACK* to be indistinguishable using non-interlevel transitions. In the former case, transition was modified to start at the border of the *WAITING_FOR_PAUSE* state. In the latter one, a new state, called *RETURNED_BACK*, was introduced, to which the considered *GO_* transitions were leading and from that state — a new transition to the *Diagram'EXIT'* diagram connector.

- Numerous shared transitions not considered above were eliminated by prefixing transition labels with their scope state.

  Some shared transitions were diagnosed by the tool incorrectly, such as default transitions without labels. This problem with TestGen is expected to be corrected in future (Sect. 7.2.5 on p. 227).

- The statechart in Fig. 8.6 has multiple termination connectors in use. These connectors could potentially be treated as the same state. Then the transition from the default connector of the main operational state (entered by *[GUARD_STARTING]/ ACTION_CHECKGENERATECOUNTER*) to the termination one leaves its enclosing state. Such behaviour of default transitions is prohibited by testing requirements. In our case, the best solution is to remove this transition since it does not make much sense because the considered statechart can be always terminated by a transition to the termination state, such as *[GUARD_CMDSTOP]*, with a priority over any transition inside the main operational one.

With the described changes, the hi-fi system complies with structural requirements of the testing method. Had it been built without history connectors, the work would have been easier. In addition, usage of OR-states in order to group related functionality rather than to introduce a hierarchy was causing problems in statecharts depicted in Fig. 8.16 and Fig. 8.17. Such OR-states can be flattened. As indicated above, usage of status information would help us distinguish states without introducing changes to the model. Finally, transitions with timeouts on them should be built to behave differently during testing but almost identically after it is finished (this also means that they should not be i_same — Sect. 5.3 on p. 102). During the design stage, one might also consider not introducing shared transitions.

8.3.3 Test case generation

Here we focus at test case basis generation for the `stereo controller` and estimation of the size of the set of test cases. The statechart itself contains 105 states and 136 transitions. The number for states was arrived at by counting all OR-, AND-, basic states and default connectors; all compound transitions were counted.

We note that usage of **DE** is almost irrelevant for the case study since it is only necessary if there is more than one transition with the same label entering any state or more than one default transition in any state. In our case **DE** is not a singleton in only one state, *STATE_MUSIC_SENSOR_OPERATION*, and while taking it into account would introduce extra complexity, the effect on the size of the set of test cases is negligible (there are not many transitions entering the state *STATE_MUSIC_SENSOR_OPERATION* compared with the overall number of transitions).

The Tab. 8.2, provided for reference, gives the state hierarchy of the considered statechart (`stereo controller`), to the extent it was defined. In our case the tree looks rather like a sequence.

state	type
top-level state	OR
PLUGGED_IN	AND
MAIN	OR
ON	AND
MODE	OR
STEREO_TAPE	AND

Table 8.2: The state hierarchy of the controller

It is possible to make different assumptions about refinements of AND states. Results are shown in Tab.8.3. They were computed from Φ, C, W provided by the tool for every state since TestGen took too much time computing multiplications necessary in test case basis construction. This deficiency of it is expected to be addressed in future.

The table shows the estimate size of the set of test cases, for three different types of testing of all concurrent states; all OR-states are tested without an expectation of refinement.

Type of testing for all AND-states	test case set size
Multiplication of states and transitions	329000M
Multiplication of states and Union of transitions	42M
Separate testing of every concurrent state	3.18K

Table 8.3: Hi-fi estimate test case set sizes for different refinement assumptions

The following abbreviations are used:

M means 'million', i.e. 329000M means $3.29 * 10^{11}$.

K is 'thousand', i.e. 3.18K is 3180.

Mult means that we generate test cases using multiplication of states and transitions (Sect. 3.3.1 on p. 54).

Union multiplication of states and union of transitions is used (weak refinement, Sect. 3.3.3 on p. 57).

Separate we test every substate of the given state separately from the rest of the statechart (strong refinement, Sect. 3.2.3 on p. 52).

It is clear that testing without usage of refinement assumptions can easily create an unmanageable test set, just compare 329000M with 41M which uses a weak refinement assumption. Separate testing, on the other hand, may require too many assumptions to be made; at the same time the amount of testing is negligible.

Consider the state *STEREO_TAPE* separately (i.e., forgetting about all states not underneath it). The Tab. 8.4 reflects how the size of the set of

type of refinement	test case size
Mult	326K
Union	35K
Separate	2.1K

Table 8.4: Test case sizes for different types of refinement of state *STEREO_TAPE*

test cases for the state changes with an assumption of weak and strong refinement. Since the state has much less states and transitions than the whole statechart, the reduction due to refinement is a lot less than that for the whole controller (in both absolute and relative terms).

In Tab. 8.5 we provide estimate sizes of the set of test cases for testing of the considered statechart using separate testing of only specific states.

UniSep means that we consider a given AND-state separately (as in OR-

state	refinement type	test case number
No refinement		42M
top-level	Separate	39M
PLUGGED_IN	UniSep	39M
PLUGGED_IN	Separate	5.7M
MAIN	Separate	5.7M
ON	UniSep	5.1M
ON	Separate	58K
MODE	Separate	63K
STEREO_TAPE	UniSep	1M
STEREO_TAPE	Separate	1M

Table 8.5: Separate testing of different states (all AND-states are tested using weak refinement)

state refinement testing) but test it using multiplication of states and union of transitions. This can only be applied to AND-states.

state	type	test case size
top-level	OR	42M
PLUGGED_IN	AND	39M
MAIN	OR	5.7M
ON	AND	5.1M
MODE	OR	57K
STEREO_TAPE	AND	35K

Table 8.6: State hierarchy and the number of test cases depending on the level in it

From formulas giving test case sizes, it is clear that the size of the set of test cases monotonically grows when numbers of states and transitions grow. Merging rules are such that those numbers never decrease as a result of merging (unless we consider prefix removal), which gives a monotonic growth of the size of the set of test cases, on the number of levels of state hierarchy it spans. Tab. 8.5 shows that introduction of separate testing at a high level does not eliminate the high size of the set since there are many levels below it; the same occurs at the bottom, where when we test *STEREO_TAPE* separately, the reduction is insignificant. It also appears that there is a point in the middle of the hierarchy, such that separate testing used there allows to cut the size down by 3 orders of magnitude in our case.

Let us compare the Tab. 8.5 with Tab. 8.6 which provides the size of the set of test cases at each level in the state hierarchy (AND-states are tested with weak refinement). From Tab. 8.6 we can see that the *ON* state causes a growth of almost 2 orders of magnitude while in all other cases it is 1 order or less. This correspond to the point described above, where the introduction of separate testing allows us to reduce the size of the set of test cases dramatically.

From these observations it is possible therefore to think that using the separate testing for states which contribute most to the size of test set seems to be the most appropriate strategy. Such states can be determined by comparing sizes of the set of test cases for different AND-states, as we did above. Such refinement could be put into practice by splitting the functionality of the system under development into a number of approximately equally complex parts, developing and testing them separately. Each of such parts could be split in turn.

Having removed a lot of complexity associated with testing state *ON* via state multiplication, we could consider further reduction in the size of it. Consider, for instance, the *STEREO_TAPE* state contributing 35K of test cases alone v.s. 58K for testing only the *ON* state separately. By testing both states separately, we get the number of test cases down to 4.6K which is rather close to that for completely separate testing (3.2K). This is a result of splitting the system once in the *ON* state and then one of the parts once again in the *STERE_TAPE* state.

Another observation from Tab. 8.5 is that for states where one substate contributes much to the size of the set of test cases and others — hardly anything, separate testing of such a substate does not change the size of the set of test cases noticeably. Examples of such states include the top-level state of the controller statechart.

8.3.4 Conclusion

The described case study provided an application of the testing method to the model which was developed in an attempt to produce a realistic example of a design, with many features used as well as many states and transitions. This was supposed to be the scalability test for the method under development since the other considered case studies lack anywhere complex state-transition diagrams. The results were quite encouraging: although the size of the set of test cases is rather big without refinement assumptions, making use of refinement can reduce it by many orders of magnitude. This also enables the method to be used in environments with different requirements for reliability of software. On one hand, we can assume refinements at almost any level and make a small test set, on another one, little number of assumptions gives rise to a huge test set; by choosing right ones, we can balance testing complexity and the amount of assumptions.

The case study has exhibited non-conformance with a variety of testing requirements and had to be modified to make the testing method applicable to it. This allowed us to illustrate various approaches to making designs comply with the requirements, which were described theoretically in Chap. 5 on p. 79.

The TestGen test tool appeared to be extremely slow even at generation of the test case basis for the case study. For example, it took 84 minutes on a SUN Ultra-2 workstation to compute the merged test case basis for the *ON* state under assumption of usage of multiplication of states and union of transitions; it was expected to take many hours to do that for the whole model. In future the performance of the tool will be improved.

8.3.5 Tools developed to facilitate the work

When the case study was under development, the `ET` toolkit supporting execution of a subset of Z, was not available. In order to validate the model against the author's perception of what it should do as well as the hi-fi user manual and the actual device, the model was built and simulated in pure Statemate statecharts. Transitions of it were later converted to Z. In order to provide for a possibility of changes of the model and eliminate the need to re-convert the whole thing due to changes, the tool *tomsz* was built. It takes Statemate statecharts and a file listing Statemate definitions of transitions with corresponding $\mu\mathcal{SZ}$ equivalents and does the substitution. This is the way statecharts presented in this chapter were generated.

Usually, visualisation of statecharts can be done using the Statemate tool, but since the author had to run it in a different university department and the license was expiring periodically with lengthy delays on renewal (if at all), an alternative solution was developed. This program is called *hello.java* and displays and randomly simulates a subset of statecharts. The drawing part of it was ported to the Perl language and statecharts of the hi-fi case study were converted from Statemate files using it.

By the time the case study was done, the ESPRESS framework still lacked a version of the tool capable of converting statecharts of the hi-fi to the ZIRP format understood by the tool, consequently, the author implemented an extension to TestGen capable of understanding a subset of statecharts encoded in XML (for more details, refer to Sect. 7.2.3 on p. 226). The hi-fi model was then converted to the considered format and results presented below were developed using the converted design.

Chapter 9

Conclusion and future work

9.1 The summary of the work done

In the thesis, the automated testing method for statechart notation has been developed. Due to the graphical nature of statecharts and their usage in industry, the testing method could be useful in improvement of the quality of safety-critical software. Chapters 2-9 describe the original work done by the author, except where stated otherwise, such as Sect. 2.3 on p. 33.

The application of the black-box testing method to statechart designs is justifiable by the need to show conformance of an implementation to a design, lacked by white-box testing methods. At the same time,

- usage of some white-box information, such as internal data may allow us to verify states (App. C.4 on p. 282),
- details of the process used to develop the system under test, can make is possible to show absence of specific faults and thus in some cases (Sect. 8.3.3 on p. 256) reduce the amount of testing dramatically.

Consequently, the developed testing method can be viewed as a greybox one, combining benefits of both black- and white-box approaches to testing.

With some restrictions on possible statecharts we deal with (Chap. 5 on p. 79), the method is shown to guarantee that all faults in the implementation of a statechart design are detected. While the requirements could be thought to be rather restrictive, case studies provided in Chap. 8 on p. 229 have shown otherwise. It appeared that with probably little effort from designers developing testable systems, it is possible to test realistic implementations (Sect. 8.1 on p. 230, Sect. 8.2 on p. 232). With the underlying X-machine testing method exhibiting good results at fault detection even if some of the testing assumptions are not satisfied (Sect. 5.2.8 on p. 97), we can assume similar behaviour for our statechart testing method, although in such cases complete fault coverage is not guaranteed.

The tool supporting the method has been built (Sect. 7.2 on p. 208). TestGen allows a designer to verify some of the testing requirements and then generate a set of test cases for his system. The tool does not really provide automated derivation of test data. While it can do so for very simple statecharts, for real applications a test engineer has to do that manually. If statecharts have been built with future testing in mind, this task is expected to be easy to automate.

The author also believes that manual derivation of test data from sequences of test cases may help diagnose problems with a design. Indeed, the derivation of those sequences does not take any behaviour expected from the system into account; for this reason, test sequences may expose undesired behaviour, which has to be particularly avoided for safety-critical applications.

Refinement assumptions consider the process used to build the system under test. Exploiting specific properties of the process, we can demonstrate absence of specific types of faults. With this in mind, the size of a test set can be reduced substantially. An experiment with the hi-fi case study has shown (Sect. 8.3.3 on p. 256) that by development of non-monolithic systems with complexity spread evenly between their components, the test set size may be reduced by many orders of magnitude.

The tool support favourably compares to that available in many other research projects. The Autofocus toolset [BS99] only considers transition tour (Sect. 2.3.1 on p. 35) as an approach to testing; Sacres project [Sac98] replaces testing by a proof of correct compilation, without consideration of the actual environment the system will operate in. The author admits, though, that these tools do not specifically focus at testing. [TPvB96] considers only FSMs and [OA99] does not rely on any formal method.

Another difference of the tool developed from others is the usage of the formal specification language Z to develop a high-level design of TestGen, which has been subsequently type checked. This allowed to eliminate a number of problems with tool development which might be rather more difficult to correct afterwards.

9.2 Main problems solved in the course of the research

Statechart notation contains significantly more features than X-machines. For this reason, one of the main tasks of the work was to identify the subset of it for which the method could be developed and state the conditions under which proofs could be constructed. This objective has been successfully completed, with requirements provided in Chap. 5 on p. 79 and proofs — in Chap. 6 on p. 106, with the main result shown in Th. 6.4.26.

Much of the complexity in the application of the X-machine test method to statecharts can be attributed to the consideration of default transitions containing triggers. Indeed, if there are none of them, the test case basis shrinks from 4 elements to 3 and merging rules simplify considerably. The complexity is related to the fact that default transitions can be viewed to have a somewhat different semantics from the rest of statecharts. With incorporation of them in the developed method, one could consider extending it for different semantics without imposing severe limitations of the second clause of t_completeness requirement (p. 63).

9.3 Advantages and disadvantages of the method developed

As shown by the case studies, the fact that the method proves correctness can be seen as both an advantage and a disadvantage.

- Having proofs of the method allows us not only to do complete testing but also makes it possible to introduce changes to it, such as relaxing some testing requirements and have a firm base to reason about completeness of testing. For instance, usage of refinements based on the process used to develop a particular system, is such a modification.
- The disadvantage is a big test set; if the purpose of testing is to demonstrate a specific relationship between a design and an implementation, rather than behavioural equivalence, methods proposed in [PS96b, FJJV96] can be used. Note that these methods can also be helpful for determination of behavioural equivalence between a design and an implementation which cannot be made to satisfy requirements for the testing method (refer to Sect. 8.2.3 on p. 233 for a discussion).

The testing method for statecharts seems to be best suited for testing an implementation against a middle-level design. For designs which are too high-level, we may have difficulties ensuring usage of the same set of transitions as indicated by the Air case study. Too low-level designs might be hard to make compliant with t_completeness and output-distinguishability requirements. Consider, for instance, programs consisting of commands executed in a sequence (i.e. having no branches). Making them triggerable and output-distinguishable could require us to write a lot more code than that of the original implementation.

There are many applications which are centered around data transformations; they are best tested with appropriate methods such as DNF (Sect. 2.3.2 on p. 38) rather than with state-machine- based ones. Unfortunately, most of the behaviour of the TestGen tool falls into this class of applications which is why an application of it to itself would be difficult.

9.4 Future work

In this section we provide a list of directions for possible future work. The following could be done:

- Enhancement of the TestGen tool, described in Sect. 7.2.5 on p. 227. The referred to section describes modification of the tool to handle statecharts to the extent described in the thesis; we could also improve it based on future research, directions for which are provided below.
- Improvement of the approaches to generation of test data (App. B.1 on p. 269, App. B.2 on p. 273). This also includes usage of theorem-proving and/or model-checking tools[1].
- Reduction of the test set size or length of test sequences:
  - research into different assumptions which could be made, other than requirements and refinement assumptions described in the thesis,
  - estimation of the size of the set of test cases/test inputs based on the complexity of statecharts and the recommendation of some possibilities for a user to change a design,
  - optimisations described in Sect. 2.4.2 on p. 44,
  - research into usage of status (App. C.4 on p. 282), and the behaviour of a system over time (Sect. 8.2.3 on p. 235) for state identification together with the Wp method (App. C.3 on p. 281).
- Test result analysis and fault location by testing; estimation of reliability of the testing method based on probabilities of testing assumptions to fail in some way. More details are given in Sect. 4.4 on p. 78.
- Testing step semantics on the basis of the prior successful testing with our method (Sect. 5.2.8 on p. 98).
- An adaptation of the testing method for UML and Pnueli and Shalev's statecharts (Sect. 3.8.2 on p. 63).
- Development of an automated approach to verification of the requirements of the testing method (Sect. 2.5 on p. 44).
- Extension of the testing method to deal with statecharts not satisfying testing requirements, such as statecharts which
  - do not have full compound transitions satisfy (Req. 1g), but can be assumed to contain correct compound transitions,

[1]such as expressing t_completeness and output-distinguishability properties in the form of CTL, for instance, by using a graph (Sect. B.1 on p. 269).

  - have multiple initial configurations (Req. 1g),
  - execute chains of transitions following those we trigger (Req. 3c),
  - are nondeterministic (Req. 1b),
  - contain **C** connectors and others, such as history ones, which are presently considered to be absent (Sect. 3.1 on p. 46); this is outlined in Sect. 6.8 on p. 206 and Sect. 3.5 on p. 60,
  - consideration of designs which are not fully observable or deterministic [LvBP94] or operating in context [PYvBD96].

- We could consider relaxing some of the design for test conditions, such as t_completeness and output-distinguishability. This is possible based on our usage of only a subset of specific data for testing, as described in the note under Def. 6.6.2.

- The growth of the size of test set w.r.t $m - n$ could be looked into (described in Sect. 4.2.3 on p. 74).

- Test set generation acceleration via parallel computation. Using a network of workstations, we can speed up test set generation and application by using them in parallel. Algorithms for testing of statecharts seem to make this easy.

9.5 Work in progress, not included in the thesis

For the duration of the author's Ph.D research, a number of ideas were proposed, not all of them directly related to the testing method. Consequently, such ideas, some of which were only partially elaborated, were not included in the thesis. In this section we provide an outline of them as follows:

Chart objects Statecharts can be considered as objects; systems — constructed from them by combining those objects together using transitions between them. The approach covers the structure and testing of such systems.

Independence theory Statemate statecharts employ two types of semantics, based on conjunction (between transitions taken in the same step) and composition of behaviour of steps. Independence theory tries to unify the two and give a more general semantics for statecharts. Refer to Sect. 1.4.10 on p. 24 for more details.

Usage of language recognisers for testing Since an FSM language recogniser can potentially be reversed to generate the language, we could build an X-machine recogniser to recognise partitions of an input set and reverse it to generate tests falling into them.

The idea can be also adapted to recognise specific classes of X-machines (such as those with a characterisation set of a specific length or simply containing less than certain number of states) and after reversal generate machines falling into them. Test data could be computed from each of those machines [ABM98] as a sequence of inputs to distinguish it from the given one. Unlike referenced paper, our approach is more general in terms of mutant machine generation.

Partition testing and input refinement For a given X-machine, we could partition its input sequences into a number of sets. After that, derivative X-machines could be built from the original ones, each operating on sequences of inputs corresponding to its partition. These machines could be tested using different testing methods, such as the X-machine one with different numbers of expected extra states as well as its relaxations, such as by using UIO sequences (Sect. 2.3.1 on p. 36) instead of a W set. The approach provides a way to combine partition testing with X-machine testing method, where every partition is tested according to its importance. A different combination of the two is provided in Sect. 2.3.2 on p. 40.

Coverage The present testing method separates testing of the behaviour of transitions from that of a transition diagram. An alternative separation is to split all transitions using the DNF approach and assume one of the partitions of every transition correct. The transition diagram consisting of such correct transitions could then be tested, followed by testing of the rest of transitions (Sect. 2.3.2 on p. 40). The coverage idea generalises on coverage of a design combining different testing methods.

Dataflow testing In principle, we could try to use the X-machine testing method for testing data flow diagrams. Processes depicted in a data flow diagram could be considered as states, data flow arrows — as transitions. A test set can then be generated; an application of it involves generation of some events in one process and verifying that they were received in another one. This approach could potentially be applied to communication of concurrent statecharts (Sect. 3.3.2 on p. 57).

Appendix A

Communication of a tester and a system under test

In the thesis we consider a tester communicating with a statechart under test via externally accessible variables, possibly using a port introduced to facilitate testing. There is an alternative: we could embed a tester in a statechart, giving it access to all internal data of the statechart and a capability to observe and influence its behaviour before a superstep or even a step is over. Such an approach allows us to weaken the requirements for statecharts, specifically, we no longer need transitions not to mask each other. The summary of different kinds of embedding is given in Tab. A.1; more detail is given in Sect. 5.2.7 on p. 93.

Here we also assume absence of communication between concurrent components of a statechart during testing (described in Sect. 3.3.2 on p. 57).

	embedding			not embedding
aspect	individual connector-to-connector transitions (‘nanostep’ semantics)	individual CTs (‘microstep’ semantics)	individual FCTs (synchronous step semantics)	sequences of transitions (asynchronous step semantics)
what is visible to a tester	we can see every change made by a transition	we can see every change made by a compound transition. Req. 3b prevents them from masking each other.	we can see every change made by a full compound transition. Req. 3b prevents them from masking each other.	we can see cumulative changes made by a sequence of full compound transitions only.
t_completeness	we can trigger every individual transition and disable those which we do not wish to take.	we can trigger every compound transition and disable those which we do not wish to take.	we can trigger every full compound transition and disable those which we do not wish to take.	we decide which set of full compound transitions we take from a test sequence and compute triggers. It is a hard task since transitions can trigger others and it could be impossible to select initial triggers such that the expected sequence executes.
order of execution	an order of transition execution is clear but due to conjunctional semantics and absence of masking its determination is not needed.	an order of compound transition execution is clear but due to conjunctional semantics and absence of masking its determination is not needed.	an order of full compound transition execution is clear	an order of execution of transitions is hard to derive from outputs.
usage of the embedding	can be used to analyse changes after individual transitions are taken	can be used to analyse changes after compound transitions are taken	can be used to analyse changes after full compound transitions are taken	can be used if the behaviour is essentially synchronous (Req. 3c) or embedding is not possible technically

Table A.1: Comparison between embedding and not embedding approaches

Appendix B

Augmentation approaches to make statecharts comply with t_completeness and output-distinguishability

B.1 The tree approach for transitions

The following approach to augmentation applies to both types of augmentation of transitions, full compound as a whole, CTs they consist of or individual connector-to-connector transitions. It is proposed as a heuristical method to facilitate augmentation and/or test data generation. The applicability study is left for future work.

B.1.1 Disjunctive approach

Every transition can be considered as a disjunction (not necessarily non-intersecting; quantified expressions are considered atomic). Triggering a transition is then equivalent to satisfying any part of its disjunction. We call such parts *disjuncts.*

For a given sequence of transitions, we can construct a directed acyclic graph called an *augmentation graph* and shown in Fig. B.1. Every horizontal dashed line represents a transition with graph nodes (drawn as blobs) being its disjuncts. A node is usually connected to all those in the dashed line below it, with weight functions on edges depending on the complexity (we call it *difficulty*) of triggering the target disjunct of an edge. The sequence of test inputs is given by a sequence of disjuncts triggered, i.e. a path through the graph from top to bottom; bold lines represent a sample one. The difficulty of it can be determined by adding together individual edge difficulties, each of which may depend on the path from the top to the beginning of an

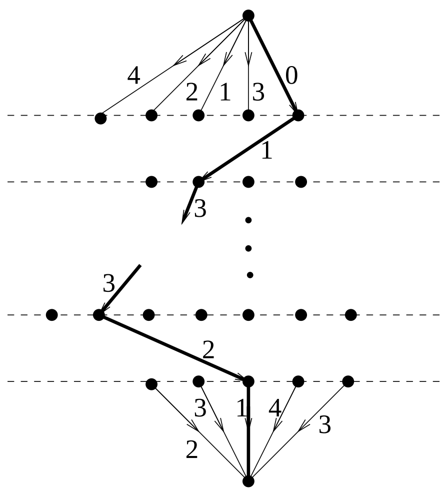


Figure B.1: An augmentation graph

edge. The graph may be used for:

- augmentation of transitions of a statechart; for that they are considered in some order,
- generation of test data from test cases.

Note that some edges would not exist in the augmentation graph, such as those corresponding to contradicting disjuncts. We can say that they possess a very big difficulty.

B.1.2 Syntax tree approach

Syntax trees for labels allow a similar method of augmentation to usage of disjuncts. An example of such a tree for a transition

$$tr_1 : (a \wedge \neg\, b \vee c) \wedge \neg\, d$$

is shown in Fig. B.2, where a branch leading to a leaf is a trigger for a transition. It can be represented in a more compact BDD form [And94].

This approach separates common parts of disjuncts instead of treating every one of them separately. This could lead to improved performance. Syntax trees can be combined together with a path of a statechart to form a graph as shown in Fig. B.3, using which test data generation can be done. Blobs in the figure represent partitions of $DATA \times conf$. Similar to ordering of variables in BDDs, this method suffers from a need to rearrange syntax of labels in order to get best results. The described way of test data generation bears some similarity with the CFTT approach (Sect. 2.3.5 on p. 42). The

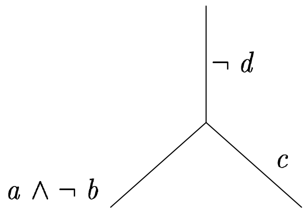


Figure B.2: An example of a syntax tree

differences is that here we take a specific path and try to find a sequence[1] of test data to follow it rather than to build an automaton such that every path in it is feasible. Fig. B.3 can also seem to be a little like Fig. 2.6 on p. 40, but instead of doing complete testing, we are only interested in determining a sequence of inputs to get from the root of the graph to any of its leaves.

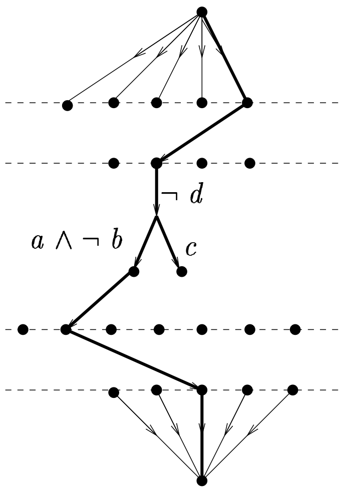


Figure B.3: A test data derivation graph composed using syntax trees

B.1.3 Estimation of 'difficulty'

A recursive function *pathcompl* may be defined to return a minimal 'difficulty' of augmenting a transition. Such a function can be defined as

$$pathcompl(path, node) = \begin{cases} min_{child \in children(node)}(\\ \quad pathcompl(path \frown edge(node, child), child)) \\ \quad + difficulty(path, node, child), \; \textbf{if } children(node) \neq \varnothing \\ 0, \; \textbf{if } children(node) = \varnothing \end{cases}$$

[1]or more than one sequence, see Air case study.

where *difficulty* is a function returning the complexity of following the edge to the node provided the path *path* has been taken. *path* may can be a path in either augmentation or transition syntax graphs.

The *difficulty* function is defined on atomic expressions and has to take the following into consideration:

- Satisfiability of the expression. This also includes considerations for contradictions with previously augmented transitions.
- The number of events we shall have to generate to trigger a transition. For example, for $a \wedge b$, two atomic predicates a and b have to be satisfied while for $a \wedge b \wedge c$ — three.
- Whether new variables or events have to be made accessible to an external tester in order to achieve t_completeness or output-distinguishability.

B.1.4 Output-distinguishability

Now we show how the output-distinguishability is evaluated. In general, we cannot do this but the proposed solution will probably suffice for most transitions. The remaining ones will have to be augmented by hand.

For a disjunct, we begin with identification of atomic outputs where the value of the visible output is clear. This includes most explicit assignments like $\mathsf{df}\ ev'$, $\mathsf{df}\ ev' \wedge \mathsf{vl}\ ev' = 5$, or $var = 3$. For a vector of variables visible to a tester, we fill in the expected values and put question marks for those whose value we cannot determine for certain.

For every transition, we can specify what output makes it distinguishable, in other words where its vector is different from that of others with the same trigger.

Traversing a triggering graph (Fig. B.1), we can keep a set of vectors of expected pure outputs of transitions encountered so far. Every new transition's output can be compared with this union and an augmentation decision reached which input to use (i.e., which edge to follow).

In a more general way, we could keep predicates in an output vector. Consider a statechart with labels $a > 5 \wedge ev_1$, $ev_2/a = 8 + \sqrt{a}$ and the initial value $a = 6$. The first of them will not be enabled when $a \leq 5$, but in practice it will always be. Having the expression $8 + \sqrt{a}$ in the output vector of the second of them could allow us to reason about $a > 5$ (assuming a is a real number). Although applicable to a wider range of designs, usage of expressions is rather complicated and is left for future work.

B.2 Augmentation of transitions which use real numbers

In this subsection we describe a mechanism for augmentation of transitions to stop them from being shared between states, such that no extra testing inputs need to be introduced. It is based on the assumption that every system uses numbers of a specific accuracy.

Consider the following two transitions:

$$\begin{aligned} tr_1 : &\quad a > 2.0/ \\ tr_2 : &\quad a > 2.0/ \end{aligned}$$

Since triggers of these transitions use the number 2.0, we could assume that functionality of our system will not get outside allowable if we change tr_1 to be $a > 2.0001/$. The main idea of this kind of augmentation is that we slightly modify real numbers used in triggers and/or actions such as to make transitions different at the same time making sure that modified transitions do not cause an undesired behaviour. An application of such an augmentation is described in Sect. 8.3.2 on p. 254. The same idea can potentially be applied to timeouts (App. B.3 on p. 273).

Two aspects of this method have to be stressed:

- If the communication channel between tester and the system under test is noisy, we have to select modification values in such a way as to prevent noise from making augmented transitions behave the same way as original ones[2] (for the noise level of 0.5, the augmentation chosen above is ineffective). Things become more interesting if the amount of noise is comparable with augmentation of variables, such as 0.00003. In this case, the two transitions tr_1 and tr_2 can be distinguished statistically (under the assumption of a deterministic system).

- The correctness of behaviour of this type of a system is often defined up to some accuracy. This implies, for instance, that an incorrect implementation which nevertheless produces the result within the expected bounds has to be considered correct.

The area of testing of such systems (which can also be expected to operate in real time, Sect. 8.2.3 on p. 235), can be considered in the future.

B.3 Delays as triggers for transitions

Real-time systems may have delays specified explicitly in them. This section describes usage of delays as inputs for triggering transitions whose triggers

[2]We would prefer not to add an extra level of accuracy to a system by using error-correction codes simply to utilise some augmentation method when other approaches are available.

involve timeouts. Two approaches are presented.

B.3.1 Generation of delays

Consider the following transition (similar to the one on p. 14):

$$\begin{aligned} stop: \quad & tm(\underline{play}, 4320)/ \\ & \mathsf{df}\ \underline{operation}' \wedge \mathsf{vl}\ \underline{operation}' = stop \end{aligned}$$

In order to trigger it, we can wait for 4320 seconds. While it is possible to treat a delay as a regular input, it is harder to make sure we do not trigger an undesired transition.

Consider the case when there is a number of transitions from a state with trigger involving timeouts:

$$\begin{aligned} tm(\underline{tape\_end}, 60) \quad &\wedge \quad \underline{operation} = play \\ tm(\underline{tape\_end}, 5) \quad &\wedge \quad \underline{operation} = play \\ tm(\underline{tape\_end}, 4) \quad &\wedge \quad \underline{operation} = rec \\ \neg\ tm(\underline{tape\_end}, 0.00001) \quad &\wedge \quad \underline{operation} = stop \\ tm(\underline{tape\_end}, 100000) \quad &\wedge \quad \underline{operation} = rew \end{aligned}$$

In the above, the top two transitions have identical triggers except for the timeout. Since in the first case we wait for 60 seconds and 5 will expire before that, a test input has to be added to one of these transitions. The third transition has its time-independent part ($\underline{operation}$=rec) different from all other transitions. We can trigger it without difficulty. The triggerability of the fourth transition depends on the relative performance of the system and the program under test. In the case when the program cannot be supplied with inputs fast enough, the timeout may expire before we generate $\underline{operation} = stop$. In such cases the transition cannot be triggered without an addition of a testing input. The last transition is hard to trigger due to a very long time we have to wait for it. It is best to augment it too.

Below the result of augmentation of the considered five labels is shown.

$$\begin{aligned} tm(\underline{tape\_end}, 60) \vee & \\ \underline{input_{test}} \quad &\wedge \quad operation = play \\ tm(\underline{tape\_end}, 5) \quad &\wedge \quad operation = play \\ tm(\underline{tape\_end}, 4) \quad &\wedge \quad operation = rec \\ \neg\ tm(\underline{tape\_end}, 0.00001) \wedge & \\ TEST\_IN\_PROGRESS = FALSE \vee & \\ \underline{input_{test}} \quad &\wedge \quad operation = stop \\ tm(\underline{tape\_end}, 100000) \vee & \\ \underline{input_{test}} \quad &\wedge \quad operation = rew \end{aligned}$$

We could augment the transition with timeout of 5 rather than the one with that of 60; our decision allows us to reduce the time necessary for test application. $TEST\_IN\_PROGRESS = FALSE$ is used to disable transitions with small timouts.

The *tm* event occurs after the specified time period and expires one step after. In case we have more complex constructs like *before*(*time*) or *after*(*time*), augmentation can be done similarly.

B.3.2 Augmentation of the *tm* function

Timeouts in statechart designs are often described using the *tm* function. For testing, we could augment it such that for every value of the argument the outcome will be the one we need. For instance, in the above example with five transitions, *tm* is supplied with a different delay in every transition; we could augment it such that it would produce the timeout event depending on a single testing input $input_{time}$ such that

$$\begin{array}{l} \forall\, ev : event;\ delay : \mathcal{R} \bullet \\ \quad tm_{augmented}(ev, delay) \Leftrightarrow \\ \qquad input_{time} = delay \land TEST\_IN\_PROGRESS = TRUE \lor \\ \qquad tm(ev, delay) \land TEST\_IN\_PROGRESS = FALSE \end{array}$$

This allows us to invoke transitions which use delays without having to supply inputs with appropriate delays which could be hard for short delays and time-consuming for long ones.

Appendix C

Wp method

Usage of the whole set W is not always necessary. In a number of cases we could construct sets W_s for some states such that the set W_s allows us to distinguish state s from all other states. For example, if variable inspection makes it possible to reason reliably about entered states, we might like to use this for state identification. Most importantly, the Wp method can be applied which allows us to reduce the size of a test set while preserving the behavioural equivalence result of testing not revealing faults.

In this chapter we describe how sets W_s for every state s of a statechart could be constructed, merging rules for them and the Wp method test case generation approach. The proof of correctness is outlined too.

C.1 Basic definitions

Let us denote $dis_{i,j}$ the sequence which distinguishes between states i and j in the same flat statechart. We can turn it into a set with the following definition

$$w_{i,j} = \begin{cases} \{dis_{i,j}\}, & \text{if } i \neq j \\ \varnothing, & \text{otherwise} \end{cases}$$

From this, the definition of characterisation set W can be expressed in terms of $w_{i,j}$ as

$$W_{st} = \bigcup_{i,j \in \rho(st)} w_{i,j}$$

where st is the considered statechart.

For a state $i \in \rho(st)$ we define an *identification set* w_i as

$$w_i = \bigcup_{j \in \rho(st)} w_{i,j}$$

in order to have sets allowing us to distinguish between a particular state in a statechart and all other states in the same statechart (but generally not on a different level of hierarchy). Note the difference between W_s and w_s. The former is the characterisation set for a statechart s while the latter is the distinguishing set for the state s in some statechart. For our tape recorder we could have

$$\begin{aligned} w_{STOP} &= \{stop\} \\ w_{PLAY} &= \{direction\} \\ w_{RECORD} &= \{stop, play\} \\ w_{REW\_FF} &= \{stop, play\} \\ w_{REWIND} &= \{ff\} \\ w_{F\_ADVANCE} &= \{rew\} \end{aligned}$$

Characterisation set W can be rewritten as

$$W_{st} = \bigcup_{i \in \rho(st)} w_i \tag{C.1}$$

C.2 Merging rules for Wp method

Rules to construct a characterisation set W for a statechart involve merging $W_{MAIN\,STATECHART}$ of the main statechart and $W_{SUBSTATE\,STATECHART}$ constructed for every non-basic state in it. We can use the same approach to merging identification sets. A state st in a main statechart can be identified with w_{st}. We write this as $w_{st}^{MAIN\,STATECHART} = w_{st}$. In order to get the identification set $w_s^{MAIN\,STATECHART}$ for a state s contained in the state st of the main statechart (we assume it to be an OR state), we need to identify that we are in st and s. Similar to merging of W,

$$w_s^{MAIN\,STATECHART} = w_{st}^{MAIN\,STATECHART} \cup w_s^{st} \tag{C.2}$$

For our example (Fig. 1.6),

$$\begin{aligned} w_{rew}^{MAIN\,STATECHART} &= \{ff, stop, play\}, \\ w_{ff}^{MAIN\,STATECHART} &= \{rew, stop, play\}. \end{aligned}$$

In case the above sets seem to be unreasonably big, skip to Prop. C.2.5 on p. 280 which describes how to reduce them.

The general rule for merging of w can be put down as follows:

Definition C.2.1.

$$\begin{aligned}
&\forall\, st : \Sigma \bullet w_{st}^{st} = \varnothing \\
&\quad (\forall\, i : \rho(st) \mid \phi(st) = stateOR \bullet w_i^{st} = w_i) \land \\
&\quad (\forall\, i : \rho(st) \mid \phi(st) = stateAND \bullet w_i^{st} = \varnothing) \land \\
&\quad (\forall\, i : \rho^{+}(st) \bullet (\exists_1\, s : \rho(st) \mid i \in \rho^{*}(s) \bullet w_i^{st} = w_s^{st} \cup w_i^{s}))
\end{aligned}$$

The second rule states that w_i identifies a state within it enclosing state; in order to identify it within levels of hierarchy, we need to unite identification sets for all states on a route (refer to Def. 6.1.12 for details) from st to s. This can also be written as $w_i^{st} = w_{s_1}^{st} \cup w_{s_2}^{s_1} \cup \ldots \cup w_i^{s_n}$ where $s_1 \ldots s_n$ are such that $s_1 \in \rho(st), s_2 \in \rho(s_1), \ldots, s_n \in \rho(s_{n-1}), i \in \rho(s_n)$. The last line of the above definition has $\exists_1$ in it due to Prop. 6.1.5.

Proposition C.2.2 (Merging rule for w). *w_i^{st} identifies a state i in a statechart st.*

Proof. The proof is by induction.

1. Consider an OR- state st such that $i \in \rho(s)$, then from Th. 6.2.14 and Th. 6.2.15 w_i^{s} identifies i within s. For $st = parent(s)$, w_s^{st} identifies s in st and thus $w_i^{st} = w_s^{st} \cup w_i^{s}$ will identify i in st.

2. For AND-states s, $w_i^{s} = \varnothing$ as we do not need to identify components of concurrent states because all of them have to be entered by Def. 6.1.3 on p. 111.

□

The above proposition deals with states of an original statechart; those of the flattened one, i.e. configurations of the original, will now be considered.

Definition C.2.3. *The identifying set w_{conf}^{st} for a configuration conf is*

$$\begin{aligned}
&\forall\, conf : \mathbb{F}_1\, \Sigma \mid configuration(root, conf) \bullet \\
&\quad w_{conf}^{st} = \bigcup\{s : conf \bullet w_s^{st}\}
\end{aligned}$$

In a similar way, we can define wf_S^{root}:

$$\forall\, S : \Sigma_f \bullet wf_S^{root} = w_{toCONFIGURATION(S)}^{root}$$

From Def. C.2.1 and Def. C.2.3, we get that

$$\forall S : \Sigma_f \bullet wf_S^{root} = \bigcup\{s : toCONFIGURATION(S) \bullet \\ \bigcup\{i : route(root, s) \setminus \{root\} \bullet w_i^{parent(i)}\}\} = \\ = \bigcup\{s : route(\!|\{root\} \times toCONFIGURATION(S)|\!) \setminus \{root\} \bullet w_s^{parent(s)}\}$$

Proposition C.2.4. *For any $S : \Sigma_f$, wf_S^{root} allows to identify configurations.*

Proof. For some $S : \Sigma_f$, consider a configuration $conf = toCONFIGURATION(S)$ and any other configuration $conf_2$. We show thatwf_S^{root} distinguishes between them.

Let state s be one of the lowest-level OR-states which belong to these two configurations[1]. We then have that $s \in conf_1 \cap conf_2$ and $\rho(s) \cap conf_1 \cap conf_2 = \varnothing$ (the first one follows from $root \in conf_1 \cap conf_2$. If there is no such s such that the second one is true, we get from Def. 6.1.3 that $conf = conf_2$).

From the second statement, we have that

$$\exists\, s_1, s_2 : \rho(s) \bullet s_1 \neq s_2 \land s_1 \in conf_1 \land s_2 \in conf_2$$

Since $w_{s_1}^s \in wf_S^{root}$ by construction of wf_S^{root}, $conf$ and $conf_2$ can be distinguished using wf_S^{root}. □

An extension of the definition of W (Eqn. C.1) for the case of merged w is consistent with merging rules for W (p. 48) as follows:

$$\begin{aligned} W_{st}^{merged} &= \bigcup\{conf : \mathbb{F}_1\,\Sigma \mid configuration(st, conf) \bullet w_{conf}^{st}\} \\ &= \bigcup\{conf : \mathbb{F}_1\,\Sigma \mid configuration(st, conf) \bullet \\ &\qquad \bigcup\{s : route(\!|\{st\} \times conf|\!) \mid s \neq st \bullet w_s^{parent(s)}\}\} \\ &= \bigcup_{s:\rho(st)} \{conf : \mathbb{F}_1\,\Sigma \mid configuration(st, conf) \bullet \\ &\qquad w_s^{st} \cup \bigcup\{i : route(\!|\{s\} \times conf|\!) \mid i \neq s \bullet w_s^{parent(i)}\}\} \\ &= \bigcup_{s:\rho(st)} (w_s^{st} \cup W_s^{merged}) \\ &= \bigcup_{s:\rho(st)} w_s^{st} \cup \bigcup_{s:\rho(st)} W_s^{merged} \end{aligned}$$

If $\phi(st) = stateOR$, $\bigcup_{s:\rho(st)} w_s^{st} = W_s$ and we get

$$W_{st}^{merged} = W_s \cup \bigcup_{s:\rho(st)} W_s^{merged}$$

[1]This part of the proof is similar to that of Prop. 6.4.24.

while if $\phi(st) = stateAND$, $w_s^{st} = \varnothing$ and

$$W_{st}^{merged} = \bigcup_{s:\rho(st)} W_s^{merged}$$

For the tape recorder in Fig. 1.6 we get from above

$$\begin{aligned}
W_{MAINSTATECHART}^{merged} &= w_{STOP}^{MAINSTATECHART} \cup w_{PLAY}^{MAINSTATECHART} \cup w_{RECORD}^{MAINSTATECHART} \cup \\
&\quad w_{REWIND}^{MAINSTATECHART} \cup w_{F\_ADVANCE}^{MAINSTATECHART} \\
&= w_{STOP} \cup w_{PLAY} \cup w_{RECORD} \cup \\
&\quad ((w_{REW\_FF} \cup w_{REWIND}) \cup (w_{REW\_FF} \cup w_{F\_ADVANCE})) \\
&= \{play, stop, rec, ff, rew\}
\end{aligned}$$

Compared with Eqn. 3.4, this $W_{MAINSTATECHART}^{merged}$ contains more sequences of transitions than necessary. The difference is that when building w, we try to identify states and this causes an increase of the size of W constructed from them. This suggests that should construct and merge W separately from w since the two have opposite goals (individual state identification and distinguishing all states respectively, refer to Sect. 2.4.2 on p. 42 for details).

Above, we merged w_{REWIND} and $w_{F\_ADVANCE}$ with w_{REW_FF} even though it is not necessary as w_{REWIND} and $w_{F\_ADVANCE}$ already identify states in the expanded statechart. Note that if we used $w_{F\_ADVANCE} = \{ff\}$ there would be no way to tell $STOP$ and $F\_ADVANCE$ apart in the merged statechart because *ff* transition exists from neither of them. These considerations give rise to the following proposition:

Proposition C.2.5. *Merging rules (Def. C.2.1) can be simplified for an OR-state main statechart to become* $w_s^{\text{MAIN STATECHART}} = w_s^{parent(s)}$ *(i.e.* $w_s^{\text{MAIN STATECHART}} = w_s$*) if the following holds:*

- *labels used in a substate statechart are not used anywhere else*

  $$(\forall A, B : \Sigma \mid A \neq B \bullet T^{ni}(A) \cap T^{ni}(B) = \varnothing)$$

- *for all states s of all substate statecharts st, w_s only contains transitions which exist from s*

  $$\forall s : \Sigma \bullet (\forall tr : w_s \bullet clpathEXISTS(s, tr))$$

Proof. By construction, w_s distinguishes between s and all other states in st, i.e.

$$\forall p : w_s \bullet clpathEXISTS(s, p) \wedge (\forall ss : \rho(st) \setminus \{s\} \bullet \neg\, clpathEXISTS(ss, p)$$

Since transitions of w_s are not used in any other statechart, this can be generalised to $\forall\, p : w_s;\ ss : \Sigma \setminus \{s\} \bullet \neg\ \mathit{clpathEXISTS}(p, ss)$. According to Th. 6.2.14 and Th. 6.2.15, w_s identifies the state corresponding to s in the flattened statechart. □

The first condition for this theorem holds due to requirements for the test method, the second one can be made true by construction of w_s.

Using the proposition, we get for our tape recorder

$$\begin{aligned} w^{MAINSTATECHART}_{REWIND} &= \{\mathit{ff}\}, \\ w^{MAINSTATECHART}_{F\_ADVANCE} &= \{\mathit{rew}\}. \end{aligned}$$

C.3 Description of the Wp method

The Wp method is [FvBK+91] an improvement of a W one, targeted at reduction of a number of test sequences. We describe here an adaptation of it to statecharts.

Definition C.3.1. *Sets for the Wp method can be defined as follows:* Φ, C *and* W *are* Φ^{merged}_{final}, C^{merged} *and* W^{merged}_{final}, *i.e. the same as those for the W method (Def. 6.4.16, Def. 6.4.11).*

$$\forall\, S : \Sigma_f \bullet w^{root,final}_S = \mathit{defaultComplete}(wf^{root}_S)$$

The main idea of the method is usage of identification sets instead of a full W set. As shown above, each of these sets can be smaller than W which leads to a smaller test set. Unfortunately, in a faulty implementation small sets wf^{root}_S may fail to identify states correctly. To cope with this, the two-phase approach is taken.

1. Checking whether states defined in a design exist in an implementation, using full W as in the W-method. Each state k is also checked whether it could be identified by the smaller set w_k. The set of test cases can be written as

   $$T_1 = C^{merged} * (\{1\} \cup \Phi^{merged}_{final} \cup \Phi^{merged^2}_{final} \cup \ldots \cup \Phi^{merged^{m-n}}_{final}) * W^{merged}_{final}$$

   The full W is used instead of appropriate w_j as in case of faulty implementations when individual w_j sets may lose their power of identification. This differs from the full test set in that the highest power of Φ^{merged}_{final} is $m - n$ rather than $m - n + 1$.

2. Testing all transitions left out in the first step. Here we can use small w_k sets to identify states and therefore create less test cases as compared with the W method while still providing the same level of confidence in results of testing.

   From schema *stINIT* (p. 175), we have an initial configuration *conf* of a considered statechart which we shall denote $conf_{init}$ here. With this, the set of test cases for the second phase of the Wp method is the following:

$$\begin{aligned} T_2 \;=\; & \bigcup\{lseqset : C^{merged} * \Phi_{final}^{merged} * (\{1\} \cup \Phi_{final}^{merged} \cup {\Phi_{final}^{merged}}^2 \cup \ldots \\ & \cup {\Phi_{final}^{merged}}^{m-n}) \bullet \{lseqset\} * w_{lfollowPATH(lseqset, conf_{init})}^{root}\} \end{aligned} \tag{C.3}$$

Theorem C.3.2 (Wp method). *Consider a statechart design satisfying the design requirements of Chap. 5 on p. 79, and an implementation of the considered system satisfying the implementation-related requirements of the same chapter. Then if the implementation delivers the expected output for $t(T)$ where t is the fundamental test function (Th. 6.7.3) and T the set of test cases as follows*

$$\begin{aligned} T \;=\; & C^{merged} * (\{1\} \cup \Phi_{final}^{merged} \cup {\Phi_{final}^{merged}}^2 \cup \ldots \\ & \cup {\Phi_{final}^{merged}}^{m-n}) * W_{final}^{merged} \\ & \cup \bigcup\{lseqset : C^{merged} * \Phi_{final}^{merged} * (\{1\} \cup \Phi_{final}^{merged} \cup {\Phi_{final}^{merged}}^2 \cup \ldots \\ & \cup {\Phi_{final}^{merged}}^{m-n}) \bullet \{lseqset\} * w_{lfollowPATH(lseqset, conf_{init})}^{root}\} \end{aligned}$$

then we have the behavioural equivalence between the considered design and implementation.

Proof. The proof of Th. 6.7.3 as well as [FvBK$^+$91] allows us to restrict our attention to proofs of properties of sets used in the method. Those for wf_S^{root} follow from Prop. 6.4.24 (i.e. Th. 6.2.14 and Th. 6.2.15), Prop. 6.4.20 and Prop. C.2.4; properties of C^{merged}, Φ_{final}^{merged} and W_{final}^{merged} are proven in Th. 6.4.26. □

C.4 Usage of status information

In some statecharts, we can detect a state we are in by observing a value of some variable (such as a portion of memory of an X-machine), without taking any transitions or by using a static reaction (which will not cause any state to be exited or entered). This could help us reduce the size of a test set considerably. Information given by such variables is further called 'status'.

We consider status information to be correctly implemented and its usage restricted in the same way as transitions (Req. 1e): if some variable is used in a group of states to identify individual states in it, all those states should be in the same statechart.

The above considerations for distinguishing sets w_i could be applied to the case where we can use status. Status variables allow us to identify groups of states without taking any transitions and thus when constructing w_s for a state s, we have to distinguish it only from members of the group it belongs to.

More precisely, when constructing w_s, we identify groups gr of transitions we can detect using status variables $status_{gr}$ and then construct $w_{s,gr}$ for states within each group. Then $w_s = w_{s,gr} \cup status_{gr}$. For example, in our tape recorder we can see whether a tape is moving or not. This allows us to have the *status_moving* boolean status variable and $w_{STOP} = \{\neg\ status\_moving\}$ ($\neg$ means that the output should be negative from the status. The negation sign is not actually used in the set of test cases but is shown here for clarification of the expected output);
for $s \in \{PLAY, RECORD, REW\_FF\}$, $w_s = \{play, stop\} \cup \{status\_moving\} = \{play, stop, status\_moving\}$. We have to use both *stop* and *play* to tell $PLAY$, $RECORD$ and REW_FF apart. Usage of *status_moving* is enough to distinguish the $STOP$ state from others, we do not have to try *stop* or *play* from it.

There could be more than a single status variable; for example, if we can identify that a tape is moving really fast, the REW_FF state can be assumed and $w_{F\_ADVANCE}^{MAINSTATECHART}$ would be $\{status\_fastmoving, rew\}$. We could also write $w_{F\_ADVANCE}^{MAINSTATECHART} = \{status\_moving, status\_fastmoving, rew\}$. For all states, we have

$$\begin{aligned}
w_{REWIND}^{MAINSTATECHART} &= \{status\_fastmoving, ff\}\\
w_{F\_ADVANCE}^{MAINSTATECHART} &= \{status\_fastmoving, rew\}\\
w_{STOP}^{MAINSTATECHART} &= \{\neg\ status\_moving\}\\
w_{PLAY}^{MAINSTATECHART} &= \{status\_moving, direction\}\\
w_{RECORD}^{MAINSTATECHART} &= \{status\_moving, direction\}
\end{aligned}$$

In the above we are using the *direction* transition to distinguish states $PLAY$ and $STOP$ in the group $status\_moving \wedge \neg\ status\_fastmoving$. Note that we can identify every state with a singe transition.

Similar to the construction of w sets, we can use status to build a characterisation set. The difference is that instead of trying to identify individual states, we try to distinguish between them (details in Sect. 2.4.2 on p. 42).

During the construction of $status_{gr}$, status variables can be combined using $\underline{*}$ but static reactions cannot since two static reactions in the same state cannot be taken concurrently. Such static reactions can be combined sequentially with $*$.

Having described the construction of $status_{gr}$ and the corresponding $w_{s,gr}$ state identification set, we need to integrate them. While it is possible to use $\cup$ to merge the two as shown above, this leads to an unnecessary increase of the size of test set and for this reason combining them using multiplication operators is preferred. During test set application we observe output after taking transitions rather than before. Consequently, $*_1$ sign cannot be used when combining $w_{s,gr}$ with status gr since status has to be checked before taking transitions rather than after. For example, $w_{F\_ADVANCE}^{MAINSTATECHART}$ can be $\{status\_moving\text{-}status\_fastmoving\ rew\}$.

The modification of the Wp method to take advantage of status information is rather easy. Below we present the new transition verification part since state verification is unaffected.

For a sequence *lseqset* in the Eqn. C.3, entering configuration $conf = lfollowPATH(lseqset, conf_{init})$, we can identify groups gr_s to which every state $s : conf \setminus \{root\}$ belongs and produce the test case

$$\{lseqset\} * status_{gr_{s_1}} *_1 status_{gr_{s_2}} *_1 \ldots *_1 status_{gr_{s_k}} * \\ * w_{s,gr_{s_1}\ldots gr_{s_k}}^{root}$$

Note that without status the above test case is equivalent to that of Eqn. C.3 due to Def. C.2.1.

Certain aspects of consistency of statecharts can be verified by testing. For example, we can ascertain that if a state is entered, its parent state is entered too. For the tape recorder, we could verify that $F\_ADVANCE \in \rho(REW\_FF)$ by using $w_{F\_ADVANCE} = \{status\_moving\text{-}status\_fastmoving\ rew\}$, since $w_{REW\_FF} = \{status\_moving\text{-}status\_fastmoving\}$ and using Prop. C.2.5, the optimal $w_{F\_ADVANCE} = \{rew\}$. Status information provides a 'free' way to do this type of checking.

Index

Bibliography

[ABM98] P. Ammann, P. E. Black, and William Majurski. Using model checking to generate tests from specifications. In *Proceedings of the 2nd IEEE International Conference on Formal Engineering Methods (ICFEM '98)*, pages 46–54, Brisbane, Australia, December 1998.

[And94] H. R. Andersen. An introduction to binary decision diagrams. `http://www.daimi.aau.dk/BRICKS/vip/users/btools/bdd-note.ps`, December 1994.

[BCCZ98] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. Obtained privately, 1998.

[BCM$^+$92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98:142–170, 1992.

[BDAR97] C. Bourhfir, R. Dssouli, E. Aboulhamid, and N. Rico. Automatic executable test case generation for extended finite state machine protocols. Technical Report 1060, Déparment d'informatique et de recherche opérationnelle, Universidé of Montréal, Universidé of Montréal, Montréal, Canada, March 1997.

[Ber94] P. J. Bernhard. A reduced test suite for protocol conformance testing. *ACM Trans. on Software Engineering and Methodology*, 3(3):201–220, July 1994.

[BFH$^+$97] K. Bogdanov, M. Fairtlough, M. Holcombe, F. Ipate, and C. Jordan. X-machine specification and refinement of digital devices. Technical Report CS-97-16, Department of Computer Science, The University of Sheffield, 1997.

[BG98] R. Buessow and W. Grieskamp. The Z of ZETA. distributed with ZETA,
`http://uebb.cs.tu-berlin.de/zeta/`, December 1998.

[BGG98] T. Bălănescu, H. Georgescu, and M. Gheorghe. Stream X-machines with underlying distributed grammars. Obtained privately; the paper submitted to Informatica, 1998.

[BGG$^+$99] T. Bălănescu, H. Georgescu, M. Georghe, M. Holcombe, and C. Vertan. A new appproach to communicating X-machines suystems. Obtained through private correspondence, 1999.

[BGGK97] R. Büssow, R. Geisler, W. Grieskamp, and M. Klar. The μSZ notation version 1.0. Technical Report 97–26, Technische Universitat Berlin, Fachbereich Informatik, December 1997.

[BGK98] R. Buessow, R. Geisler, and M. Klar. Specifying safety-critical embedded systems with statecharts and Z: A case study. *Lecture Notes in Computer Science*, 1382:71–87, 1998.

[BH97] K. Bogdanov and M. Holcombe. The mapping between $\mu\mathcal{SZ}$ statecharts and X-machines. ESPRESS Workshop, Berlin, Germany, January 1997.

[BH98] K. Bogdanov and M. Holcombe. September 98 report. internal report for Daimler-Benz, October 1998.

[BHS98a] K. Bogdanov, M. Holcombe, and H. Singh. Test generation from statemate specifications. Testing Workshop, York, UK, September 1998.

[BHS98b] K. Bogdanov, M. Holcombe, and H. Singh. Testing statemate models. Notes presented at the X-Machines Day, Department of Computer Science, University of Sheffield, UK, July 1998.

[BHS99] K. Bogdanov, M. Holcombe, and H. Singh. Automated test set generation for statecharts. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *Applied Formal Methods - FM-Trends 98*, volume 1641 of *Lecture Notes in Computer Science*, pages 107–121. Springer Verlag, 1999.

[BKN] M. Benini, S. Kalvala, and D. Nowotka. Program abstraction in a higher-level logic framework. `http://www.dcs.warwick.ac.uk/holly/papers/AbstractionAndLogic.ps.gz`.

[Bog96] K. Bogdanov. Six-month report. Technical report, Department of Computer Science, The University of Sheffield, October 1996.

[Bog97] K. Bogdanov. Basics of mapping from statecharts to X-machines, the first year report. Technical report, Department of Computer Science, The University of Sheffield, April 1997.

[Bog98] K. Bogdanov. The second year report: a road to Ph.D. Technical report, Department of Computer Science, The University of Sheffield, May 1998.

[BS96] A. Bertolino and L. Strigini. On the use of testability measures for dependability assessment. *IEEE transactions on Software Engineering*, 22, NO. 2:97–108, February 1996.

[BS99] M. Broy and O. Slotosch. Enriching the software development process by formal methods. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *Applied Formal Methods - FM-Trends 98*, volume 1641 of *Lecture Notes in Computer Science*, pages 44–61. Springer Verlag, 1999.

[Bur98] S. Burton. 1st year qualifying dissertation, 6 July 1998. Obtained privately.

[Bur99] S. Burton. Towards automated unit testing of statechart implementations. Technical Report YCS 319, Department of Computer Science, University of York, UK, 1999.

[BW98] U. Brockmeyer and G. Wittich. Tamagotchis need not die — verification of STATEMATE designs. *Lecture Notes in Computer Science*, 1384:217–231, 1998.

[CAB$^+$98] W. Chan, R. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J. Reese. Model Checking Large Software Specifications. *IEEE Transactions on Software Engineering*, 24(7):498–520, 1998.

[CGH94] E. Clarke, O. Grumberg, and K. Hamaguchi. Another look at LTL model checking. *Lecture Notes in Computer Science*, 818:415–427, 1994.

[Cho78] T. S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, SE-4(3):178–187, 1978.

[CJ95] E. M. Clarke and S. Jha. Symmetry and induction in model checking. *Lecture Notes in Computer Science*, 1000:455–470, 1995.

[CK96] S. C. Cheung and J. Kramer. Checking subsystem safety properties in compositional reachability analysis. In *18th International Conference on Software Engineering*, pages 144–154, Berlin - Heidelberg - New York, March 1996. Springer.

[Cor96] J. C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering*, 22, NO. 3:161–180, March 1996.

[CT98] Cadd and Todd. The final year project, 1998.

[CW96] E. M. Clarke and J. M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, December 1996.

[Day93] N. Day. A model checker for statecharts. Technical Report TR-93-35, UBC, October 1993. `ftp://ftp.cs.ubc.ca/ftp/local/techreports/1993/TR-93-35.ps`.

[DB94] TESSY - yet another computer-aided software testing tool? In *EuroSTAR*. Daimler-Benz AG, 1994.

[DB98] J. Derrick and E. A. Boiten. Testing refinements by refining tests. In J. P. Bowen, A. Fett, and M. G. Hinchey, editors, *ZUM'98: The Z Formal Specification Notation*, volume 1493 of *Lecture Notes in Computer Science*, pages 265–283. Springer-Verlag, September 1998. http://www.cs.ukc.ac.uk/pubs/1998/609.

[DF93] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In J.C.P. Woodcock and P.G. Larsen, editors, *FME '93: Industrial Strength Formal Methods*, volume 670 of *Lecture Notes in Computer Science*, pages 268–284. Formal Methods Europe, Springer Verlag, April 1993.

[DN84] J. W. Duran and S. C. Ntafos. An evaluation of random testing. *IEEE Transactions on Software Engineering*, SE-10(4):438–443, July 1984.

[Fau99] D. R. Faught. comp.software.testing.faq. `http://www.rstcorp.com/c.s.t.faq.html`, 1999.

[Fel98] C. Feldman. False alarm on jet causes 2 near-crashes. `http://www.cnn.com/US/9801/15/near.collision/index.html`, January 1998.

[FHI+95] M. Fairtlough, M. Holcombe, F. Ipate, C. Jordan, G. Laycock, and Z. Duan. Using an X-machine to model a video cassette recorder. *Current issues in electronic modelling*, 3:141–161, 1995.

[FJJV96] J.-C. Fernandez, C. Jard, T. Jeron, and G. Viho. Using on-the-fly verification techniques for the generation of test suites. *Lecture Notes in Computer Science*, 1102:348–359, 1996.

[FJW97] V. Friesen, S. Jähnichen, and M. Weber. Specification of software controlling a discrete-continuous environment. In *Proceedings of the 19th International Conference on Software Engineering (ICSE '97)*, pages 315–325, Berlin - Heidelberg - New York, May 1997. Springer.

[FS97] F. Fummi and D. Sciuto. a complete test strategy based on interacting and hierarchical FSMs. In *IEEE International symposium on Circuits and Systems*, pages 2709–2712, June 1997.

[FvBK$^+$91] S. Fujiwara, G. von Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17(6):591 – 603, June 1991.

[GH99] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. In *proceedings of ESEC/FSE'99*, Tolouse, France, September 1999.

[GHD98] W. Grieskamp, M. Heisel, and H. Dörr. Specifying embedded systems with statecharts and Z: An agenda for cyclic software components. *Lecture Notes in Computer Science*, 1382:88–106, 1998.

[GK96] R. Geisler and M. Klar. Towards a formal semantics for statemate statecharts. `http://(espress/Arbeitsgruppen/Spezifikation/dokumente/semantik.ps)`, August 1996.

[Har87] D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[Har97] D. Harel. Some thoughts on statecharts, 13 years later. *Lecture Notes in Computer Science*, 1254:226–231, 1997.

[HB97] M. Holcombe and K. Bogdanov. The third step towards correct software. British Colloquium for Theoretical Computer Science (BCTCS) 13, Sheffield, UK, March 1997.

[HCB92] F. Hayes, D. Coleman, and S. Bear. Introducing objectcharts or how to use statecharts in object oriented design. *IEEE Transactions on Software Engineering*, 18(1):9–18, January 1992.

[HG94] D. Harel and E. Gery. Executable object modeling with statecharts.
`http://www.wisdom.weizmann.ac.il/Papers/trs/CS94-20/abstract.html`, 1994.

[HG96] D. Harel and E. Gery. Executable object modeling with statecharts. In *18th International Conference on Software Engineering*, pages 246–257, Berlin - Heidelberg - New York, March 1996. Springer-Verlag.

[HGdR88] C. Huizing, R. Gerth, and W. P. de Roever. Modelling statecharts behaviour in a fully abstract way. *Lecture Notes in Computer Science*, 299:271–294, 1988.

[HHH$^+$99] M. Harman, R. Hierons, M. Holcombe, B. Jones, S. Reid, M. Roper, and M. Woodward. Towards a maturity model for empirical studies of software testing. Obtained privately, July 1999.

[HHS86] J. He, C. A. R. Hoare, and J. W. Sanders. Data refinement refined (resumé). In ESOP'86, volume 213 of *Lecture Notes in Computer Science*, pages 187–196. Springer Verlag, 1986.

[HI98] M. Holcombe and F. Ipate. *Correct Systems: building a business process solution.* Springer-Verlag Berlin and Heidelberg GmbH & Co. KG, September 1998.

[Hie96] R. M. Hierons. Extending test sequence overlap by invertibility. *COMPJ: The Computer Journal*, 39(4):325–330, 1996.
`http://www.oup.co.uk/jnls/list/comjnl/hdb/Volume_39/Issue_04/390325.sgm.abs.html`.

[Hie97a] R. M. Hierons. Controlling testing and failure location using a finite state machine. Obtained through private correspondence, 1997.

[Hie97b] R. M. Hierons. Testing from a finite state machine: extending invertibility to sequences. *COMPJ: The computer journal*, 40(4), 1997.

[Hie97c] R. M. Hierons. Testing from a Z specification. *Journal of software testing, verification and reliability*, 7(1):19–33, 1997.

[Hie97d] R. M. Hierons. Testing from semi-independent communicating finite state machines with a slow environment. *IEE Proceedings on Software Engineering*, 144(5–6):291–295, 1997.

[HL96] M. E. Heimdahl and N. G. Leveson. Completeness and consistency in hierarchical state-based requirements. *IEEE Transactions on Software Engineering*, 22(6):363–377, June 1996.

[HLN+90] D. Harel, H. Lachover, A. Nammad, 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, April 1990.

[HMLS98] G. Holzmann, E. Mikk, Y. Lakhnech, and M. Siegel. Verifying statecharts with Spin. *Proc. Workshop on Industrial-strength Formal specification Techniques*, October 1998.

[HN96] D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.

[HNS] S. Helke, T. Neustupny, and T. Santen. Automating test case generation from Z specifications with isabelle. `http://(espress/Arbeitsgruppen/V&V/papers)`.

[HPSS87] D. Harel, A. Pnueli, J. P. Schmidt, and R. Sherman. On the formal semantics of statecharts (extended abstract). In *Symposium on Logic in Computer Science (LICS '87)*, pages 54–64, Washington, D.C., USA, June 1987. IEEE Computer Society Press.

[HRdR92] J. J. M. Hooman, S. Ramesh, and W.P. de Roever. A compositional axiomatization of statecharts. *Theoretical Computer Science*, 101:289–335, 1992.

[HT90] D. Hamlet and R. Taylor. Partition testing does not inspire confidence. *IEEE Transactions on Software Engineering*, 16(12):1402–1411, December 1990.

[IH96] F. Ipate and M. Holcombe. Another look at computability. *Informatica*, 20:359–372, 1996.

[IH97] F. Ipate and M. Holcombe. An integration testing method that is proved to find all faults. *International Journal on Computer Mathematics*, 63:159–178, 1997.

[IH98a] F. Ipate and M. Holcombe. A method for refining and testing generalised machine specifications. *International Journal on Computer Mathematics*, 68:197–219, 1998.

[IH98b] F. Ipate and M. Holcombe. Specification and testing using generalized machines:a presentation and a case study. *Software testing, verification and reliability*, 8, 1998.

[IH99] F. Ipate and M. Holcombe. Generating test sequences from non-deterministic X-machines. Obtained from Prof. M. Holcombe, 1999.

[Ilo95a] Ilogix, inc. *Statemate Analyser reference manual, v6.0*, May 1995.

[Ilo95b] Ilogix, inc. *Statemate User reference manual, Volume 1, v6.0*, May 1995.

[Ipa95] F. E. Ipate. *Theory of X-machines and Applications in Specification and Testing.* PhD thesis, University of Sheffield, July 1995.

[JM94] F. Jahanian and A. K. Mok. Modechart: A specification language for real-time systems. *IEEE Transactions on Software Engineering*, 20(12):933–947, December 1994.

[Kan99] C. Kaner.
`http://www.nmsvr.com/~kaner/`, 1999.

[Kar92] G. Karjoth. Implementing LOTOS specifications by communicating state machines. In W. R. Cleaveland, editor, *CONCUR '92: Third International Conference on Concurrency Theory*, volume 630 of *Lecture Notes in Computer Science*, pages 386–400, Stony Brook, New York, August 1992. Springer-Verlag.

[KE98] Khan and East. The final year project, 1998.

[KHC$^+$99] Y. G. Kim, H. S. Hong, S. M. Cho, D. H. Bae, and S. D. Cha. Test cases generation from UML state diagrams. *IEE Proceedings - Software*, 146(4):187–192, August 1999.

[KP92] Y. Kesten and A. Pnueli. Timed and hybrid statecharts and their textual representation. In J. Vytopil, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, number 571 in Lecture Notes in Computer Science, pages 591–620, Berlin, 1992. Springer-Verlag.

[KP98] T. Koomen and M. Pol. Improvement of the test process using TPI.
`http://www.iquip.nl/images/tpi_summary.doc`, 1998.

[Lay92] G. T. Laycock. *The Theory and Practice of Specification Based Software Testing.* PhD thesis, University of Sheffield, September 1992.

[LC96] K. R. P. H. Leung and D. K. C. Chan. Extending statecharts with duration.
`ftp://ftp.cs.utwente.nl/pub/doc/OM/chan/96.08_P_Statecharts.ps.gz`, 1996.

[LFHH] L. Logrippo, M. Faci, and M. Haj-Hussein. An introduction to LOTOS: Learning by examples.
`ftp://lotos.csi.uottawa.ca/pub/Lotos/Papers/tutorial.ps.Z`.

[LHHR94] N. G. Leveson, M. P. E. Heimdahl, H. Hildreth, and J. D. Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, 20(9):684–707, September 1994.

[LvBP94] G. Luo, G. von Bochmann, and A. Petrenko. Test selection based on communicating nondeterministic finite-state machines using a generalized WP-method. *IEEE Transactions on Software Engineering*, 20(2):149–162, February 1994.

[LY94] D. Lee and M. Yannakakis. Testing finite state machines: State identification and verification. *IEEE Transactions on Software Engineering*, 43(3):306–320, March 1994.

[Mar92] F. Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In W. R. Cleaveland, editor, *CONCUR '92: Third International Conference on Concurrency Theory*, volume 630 of *Lecture Notes in Computer Science*, pages 550–564, Stony Brook, New York, August 1992. Springer-Verlag.

[Meu98] C. Meudec. *Aytomatic Generation of Software Test Cases from Formal Specifications.* PhD thesis, The Queen's University of Belfast, May 1998.
`http://www.geocities.com/CollegePark/Square/4148/research/thesis/thesis.zip`.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[MLPS97] E. Mikk, Y. Lakhnech, C. Petersohn, and M. Siegel. On formal semantics of statecharts as supported by statemate. In *BCS-FACS Nothern Formal Methods Workshop*, pages 0–13, Craiglands Hotel, Ilkley, West Yorkshire, U.K., July 1997.

[MLS97] E. Mikk, Y. Lakhnech, and M. Siegel. Hierarchical automata as model for statecharts. *Lecture Notes in Computer Science*, 1345:181–196, 1997.
`http://www.informatik.uni-kiel.de/~erm/FILES/asian97.ps`.

[MSP96] A. Maggiolo-Schettini and A. Peron. A graph rewriting framework for statecharts semantics. *Lecture Notes in Computer Science*, 1073:107–121, 1996.

[MSPT96] A. Maggiolo-Schettini, A. Peron, and S. Tini. Equivalences of statecharts. In Ugo Montanari and Vladimiro Sassone, editors, *CONCUR '96: Concurrency Theory, 7th International Conference*, volume 1119 of *Lecture Notes in Computer Science*, pages 687–702, Pisa, Italy, August 1996. Springer-Verlag.

[MST97] A. Maggiolo-Schettini and S. Tini. Projectable semantics for statecharts.
`http://www.daimi.aau.dk/~bra8130/LOMAPS_archive/DIPISA-37.ps.Z`, 1997.

[Mye79] G. J. Myers. *The art of software testing.* John Wiley and Sons, 1979.

[NH95] A. Naamad and D. Harel. The statemate semantics of statecharts. *Technical Report, Weizmann Institute of Science*, 1995.

[NRS96] D. Nazareth, F. Regensburger, and P. Scholz. Mini-statecharts: A lean version of statecharts.
`http://www4.informatik.tu-muenchen.de/reports/TUM-I9610.ps.gz`, February 1996.

[OA99] J. Offutt and A. Abdurazik. Generating tests from UML specifications. In *Second International Conference on the Unified Modeling Language (UML99), Fort Collins, CO, October 1999*, October 1999.

[Ove94] J. Overback. Testing generic classes. *EuroSTAR 1994*, pages 41/1–41/11, 1994.

[PCCW93] M. Paulk, B. Curtis, M. Chrissis, and C. Weber. Capability maturity model for software, version 1.1. Technical Report CMU/SEI-93-TR-024,ESC-TR-93-177, SEI, February 1993.

[Pel96] J. Peleska. Test automation for safety-critical systems: Industrial application and future developments. In M. C. Gaudel and J. Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, pages 39–59. Springer, 1996.

[Per95] A. Peron. Statecharts, transition structures and transformations. *Lecture Notes in Computer Science*, 915:454–468, 1995.

[PF90] D. H. Pitt and D. Freestone. The derivation of conformabnce tests from LOTOS specifications. *IEEE Transactions on Software Engineering*, 16(12):1337–1343, December 1990.

[PM94] A. Peron and A. Maggiolo-Schettini. Transitions as interrupts: A new semantics for timed statecharts. *Lecture Notes in Computer Science*, 789:806–821, 1994.

[Pre94] R. S. Pressman. *Software Engineering, a practitioner's approach.* London, McGraw-Hill, third edition, 1994.

[PS91] A. Pnueli and M. Shalev. What is in a step: On the semantics of statecharts. In T. Ito and A. Meyer, editors, *Int. Conf. TACS'91: Theoretical aspects of Computer Software*, volume 526, pages 244–264. Springer-Verlag, September 1991.

[PS96a] J. Peleska and M. Siegel. From testing theory to test driver implementation. In M. C. Gaudel and J. Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, pages 538–556. Springer, 1996.

[PS96b] J. Peleska and M. Siegel. Test automation of safety-critical reactive systems.
`http://www.informatik.uni-bremen.de:80/~jp/papers/sacj97.ps.gz`, August 1996.

[PS97a] J. Philipps and P. Scholz. Compositional specification of embedded systems with statecharts. In Michel Bidoit and Max Dauchet, editors, *TAPSOFT'97:Theory and Practice of Software Development*, volume 1214 of *Lecture Notes in Computer Science*, pages 637–651, Lille, France, April 1997. Springer-Verlag.
`http://hpbroy3.informatik.tu-muenchen.de/~scholzp/Postscript/tapsoft.ps`.

[PS97b] J. Philipps and P. Scholz. Formal verification of statecharts with instantaneous chain reactions. In *TACAS'97, Twente*, 1997.
`http://hpbroy3.informatik.tu-muenchen.de/~scholzp/Postscript/tacas.ps`.

[PSM96] C. Puchol, D. Stuart, and A. K. Mok. An operational semantics and a compiler for modechart specificiations. Technical Report CS-TR-95-37, University of Texas, Austin, July 1996.
`ftp://ftp.cs.utexas.edu/pub/techreports/tr95-37.ps.Z`.

[PSS98] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. *Lecture Notes in Computer Science*, 1384:151–166, 1998.

[PYvBD96] A. Petrenko, N. Yevtushenko, G. von Bochmann, and R. Dssouli. Testing in context: framework and test derivation. *Computer Communications*, 19:1236–1249, 1996.

[Rat97] Rational Corp. UML 1.1 semantics - behavioral elements package: State machines. `http://www.rational.com/uml/resources/documentation/semantics/semant11a.jtmpl`, September 1997.

[RDT95a] T. Ramalingam, A. Das, and K. Thulasiraman. Fault detection and diagnosis capabilioties of test sequence selection methods based on the fsm model. *Computer Communications*, 18(2):113–122, February 1995.

[RDT95b] T. Ramalingam, A. Das, and K. Thulasiraman. On testing and diagnosis of communication protocols based on the fsm model. *Computer communications*, 18(5):329–337, May 1995.

[Rop94] M. Roper. *Software Testing.* McGraw-Hill, 1994.

[RR93] M. Roper and R. Rahim. Software testing using analysis and design based techniques. *Software Testing, Verification and Reliability*, 3:165–179, 1993.

[Sac98] Sacres project. `http://www.tni.fr/sacres/indexpres.html`, 1998.

[Sad98] S. Sadeghipour. *Testing Cyclic Software Componenets of Reactive Systems on the Basis of Formal Specifications.* Verlag Dr. Kovač, 1998.

[Sch96] P. Scholz. An extended version of mini-statecharts. `http://www4.informatik.tu-muenchen.de/reports/TUM-I9628.ps.gz`, June 1996.

[SCW98] S. Stepney, D. Cooper, and J. C. P. Woodcock. More powerful Z data refinement: Pushing the state of the art in industrial refinement. *Lecture Notes in Computer Science*, 1493:284–307, 1998.

[Shi95] F. Shi. *Automatic coding from ZedCharts to SPARC Ada for Safety-Critical Systems.* PhD thesis, University of York, UK, October 1995.

[Sim00] A. J. H. Simons. On the compositional properties of UML statechart diagrams. To appear in proceedings of the 3rd Rigorous Object-Oriented Methods workshop (ROOM3), University of York, UK, January 2000.

[SL89] D. Sidhu and T. Leung. Formal methods for protocol testing: A detailed study. *IEEE Transactions on Software Engineering*, 15(4):413 – 426, 1989.

[SLD90] Y.-N. Shen, F. Lombardi, and A. T. Dahbura. Protocol conformance testing using multiple uio sequences. In E. Brinksma, G. Scollo, and C. A. Vissers, editors, *Protocol Specification, Testing and Veification IX*, pages 131–143. Elsevier Schence Publishers B.V. (North-Holland), 1990.

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall International Series in Computer Science, 2nd edition, 1992.
`http://www.blackwell.co.uk/cgi-bin/bb_item?0139785299`.

[SS98] S. Sadeghipour and H. Singh. A tool environmenet for testing on the basis of formal specifications. Obtained privately, 1998.

[SVG] A. A. Saloña, J. Q. Vives, and S. P. Gómez. An introduction to LOTOS.
`ftp://ftp.dit.upm.es/pub/lotos/papers/tutorial/lotos_language_tutorial.ps`.

[Toy98] I. Toyn. Innovations in the notation of standard Z. *Lecture Notes in Computer Science*, 1493:193–213, 1998.

[TPvB96] Q. M. Tan, A. Petrenko, and G. v. Bochmann. A test generation tool for specifications in the form of state machines. In *International Communications Conference (ICC) 96, session on advanced tools and technologies for developing high integrity software systems, Dallas, Texas*, pages 225–229, 23–27June 1996.

[TPvB97] Q. M. Tan, A. Petrenko, and G. von Bochmann. Deriving test with fault coverage for specificatios in the form of labeled transition systems. Technical Report 1073, Universiteé de Montreéal, June 1997.
`http://www.iro.umontreal.ca/labs/teleinfo/TRs/P1073.ps.gz`.

[TS95] P. Tripathy and B. Sarikaya. Analysis and representation of test cases generated from LOTOS. *Computer Communications*, 18(7):493–506, July 1995.

[Ura92] H. Ural. Formal methods for test sequence generation. *Computer Communications*, 15(5):311–325, 1992.

[US94] A. C. Uselton and S. A. Smolka. A compositional semantics for statecharts using labeled transition systems. In Bengt Jonsson and Joachim Parrow, editors, *CONCUR '94: Concurrency Theory*, volume 836 of *Lecture Notes in Computer Science*, pages 2–17, Berlin, 1994. Springer Verlag.

[vdB93] M. von der Beeck. Integration of structured analysis and timed statecharts for real-time and concurrency specification. *Lecture Notes in Computer Science*, 717:313–328, 1993.

[vdB94] M. von der Beeck. A comparison of statecharts variants. *Lecture Notes in Computer Science*, 863:128–148, 1994.

[vdSU95] H. van der Schoot and H. Ural. Data flow oriented test selection for LOTOS. *Computer Networks and ISDN Systems*, 27(7):1111–1136, 1995.

[WD96] J. Woodcock and J. Davies. *Using Z Specification, Refinement, and Proof.* London, Prentice Hall, 1996.

[Web96] M. Weber. Combining statecharts and Z for the design of safety-critical systems. In M. C. Gaudel and J. Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, volume 1051, pages 307–326. Springer, 1996.

[Wez90] C. Wezeman. The CO-OP method for compositional derivation of conformance testers. In E. Brinksma, G. Scollo, and C. A. Vissers, editors, *Protocol Specification, Testing and Veification IX*, pages 145–158, 1990.

[WTF94] H. Waeselynck and P. Thévenod-Fosse. An experimentation with statistical testing. *EuroSTAR 1994*, 1994.

[WVF95] J. M. Wing and M. Vaziri-Farahani. Model Checking Software Systems: A Case Study. In *Proceedings of SIGSOFT'95 Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 128–139, October 1995.

[XMI99] XML metadata interchange. `http://www.software.ibm.com/ad/standards/xmi.html`, 1999.

[XML] Extensible markup language. `http://www.w3.org/XML/`.