

# The Design and Construction of a State Machine System that Handles Nondeterminism

Graham G. Thomason

Submitted for the Degree of  
Doctor of Philosophy  
from the  
University of Surrey



Department of Computing  
School of Electronics and Physical Sciences  
University of Surrey  
Guildford, Surrey GU2 7XH, UK

July 2004

© Graham G. Thomason 2003-2004

## Summary

We describe a language system (called STATECRUNCHER) which implements statecharts, handling nondeterminism in a novel way. Statecharts specified in the style of UML dynamic models can generally easily be expressed in STATECRUNCHER. STATECRUNCHER is intended as a test oracle, working in conjunction with a test generator and a test harness connected to an implementation. Such a tool chain *tests an implementation* for conformance against a specification (compare *model checking*, which checks properties of a specification without the need for an implementation). Nondeterminism is becoming an increasingly important issue, especially in integration testing, where internal behaviour may be subject to some freedom, and where control over subsystems is limited, so that alternatives in behaviour are acceptable. We cover the language, its implementation, and experience with it in a tool chain automatically generating and executing tests on embedded software at the sponsoring company, Philips Electronics N.V.

## Acknowledgements

Thanks are due to my supervisors:

- Timotheo Trew, *ars cui summa est, studium doctrinae pudorque. quem magni artifices semper dicunt magistrum. doctior hoc nemo est; potest quem vincere nemo programmata qui noverit probare.*
- Aan Prof. Paul Krause. Hij bezit de grootste vaardigheid, enthousiasme voor het vak, en bescheidenheid. Grote waarschijnlijkheidsleerdeskundigen noemen hem steeds meester. Niemand is geleerder dan hij; niemand kan hem overtreffen die kan redeneren onder onzekerheid.
- To Dr. David Pitt, who has the greatest skill, enthusiasm for his subject, and modesty. Great academics always call him the expert. No-one is more scholarly than he; no-one who knows how to formally specify a system can surpass him.

If STATECRUNCHER is found to have syntax and semantics that map well to (models of) a variety of industrial systems, —that are intuitive to system architects and testing practitioners, —that are powerful enough to satisfy the intellectually adventurous, —then this is thanks to the expert guidance of Tim Trew.

If STATECRUNCHER is found to be of interest to the scholarly world, and if the works of the scholarly world are found to be amenable to engagement with STATECRUNCHER, then thanks for this are due to Prof. Paul Krause who first proposed that the present author submit the work in an academic context.

If the present thesis bridges the proverbial industrial-academic gap, then thanks are due to Dr David Pitt, without whose assistance much of the academic side would have remained an unknown quantity.

Thanks are also due to Nitin Koppalkar at Philips Research India Bangalore for his competent integration skills in a cross-continental co-operation to see the successful integration of STATECRUNCHER in the TorX tool chain, and subsequent testing of various embedded software components.

Thanks are also due to many others at Philips, including my internal customer Ing. Wil Hoogenstraaten at Consumer Electronics who was contract research project owner for this project, and Bob Barnes at Philips Digital Systems Laboratories Redhill, whose support for the undertaking was invaluable.

# Contents

1.	Introduction .....	1
1.1	Context of the work .....	1
1.2	The problem to be considered.....	1
1.3	A peek at the result of the work .....	2
1.4	What STATECRUNCHER is not .....	2
1.5	The structure of this thesis .....	4
2.	Software testing in context .....	5
3.	State-based testing and STATECRUNCHER overview .....	9
3.1	States and events.....	9
3.2	Deterministic state-based testing .....	12
3.3	Nondeterministic testing .....	25
3.4	STATECRUNCHER and the TorX tool chain.....	26
3.5	Alternative modelling techniques to state-based modelling .....	26
3.6	Summary of this section .....	28
4.	Nondeterminism .....	29
4.1	Review of nondeterministic testing.....	29
4.2	Fork nondeterminism .....	30
4.3	Race nondeterminism .....	33
4.4	Set transit nondeterminism .....	35
4.5	Fired-event and multiple nondeterminism .....	36
4.6	Set-action nondeterminism .....	38
4.7	Set-meta-event nondeterminism .....	39
4.8	Effects of nondeterminism .....	41
4.9	Worlds .....	42
4.10	Containment of combinatorial explosion .....	44
4.11	Test generation under nondeterminism .....	63
4.12	Summary of this section .....	64
5.	STATECRUNCHER as a language .....	66
5.1	General syntax .....	66
5.2	STATECRUNCHER statements.....	68
5.3	Basic syntax of statechart / cluster / set and (leaf-)states in a hierarchy .....	69
5.4	More about hierarchical states .....	70
5.5	Declarations and scoping .....	75
5.6	Expressions, operators and functions .....	80
5.7	Review of items parsed as expressions .....	99
5.8	Transition block .....	101
6.	Algorithmic sequencing.....	118
6.1	Cycling .....	119

6.2	Maintaining machine integrity .....	120
6.3	An <i>in-flight</i> approach .....	124
6.4	An <i>after-landing</i> approach .....	131
6.5	Client-server composition and PCOs .....	133
6.6	Conclusions on the sequencing in the transition algorithm .....	135
7.	The transition algorithm .....	136
7.1	The formal statechart and the nondeterministic transition function .....	136
7.2	Statechart properties .....	139
7.3	Transition selection.....	142
7.4	Discussion of hierarchical fork nondeterminism.....	145
7.5	Transition course .....	150
7.6	Task processing .....	165
8.	The STATECRUNCHER command language .....	190
9.	Using STATECRUNCHER.....	195
9.1	Data flow .....	195
9.2	Running STATECRUNCHER.....	196
9.3	Testing of STATECRUNCHER .....	197
9.4	The dining philosophers.....	198
10.	Experience with STATECRUNCHER and conclusions .....	218
10.1	Experience at Philips .....	218
10.2	PROLOG as the implementation language .....	226
10.3	Future directions .....	229
10.4	Final conclusion.....	233
11.	Glossary and abbreviations etc. ....	234
11.1	Greek letters.....	234
11.2	Glossary and abbreviations .....	234
12.	References .....	239

### ***Appendices***

Appendix 1	[StCrContext]	Software Testing in Context
Appendix 2	[StCrSemComp]	A Semantic Comparison of STATECRUNCHER and Process Algebras
Appendix 3	[StCrOutput]	A Quick Reference of STATECRUNCHER's Output Format
Appendix 4	[StCrDistArb]	Distributed Arbiter Modelling in CCS and STATECRUNCHER - A Comparison
Appendix 5	[StCrNim]	The Game of Nim in Z and STATECRUNCHER

Appendix 6      [StCrBiblRef]      Bibliography and References

***Related reports***

Related report 1	[StCrPrimer]	STATECRUNCHER-to-Primer Protocol
Related report 2	[StCrManual]	STATECRUNCHER User Manual
Related report 3	[StCrGP4]	GP4 - The Generic Prolog Parsing and Prototyping Package ( <i>underlies the STATECRUNCHER compiler</i> )
Related report 4	[StCrParsing]	STATECRUNCHER Parsing
Related report 5	[StCrTest]	STATECRUNCHER Test Models
Related report 6	[StCrFunMod]	State-based Modelling of Functions and Pump Engines

# 1. Introduction

## 1.1 Context of the work

We are concerned with testing embedded and distributed software systems. They are difficult to test, yet it is vital that they *are* properly tested, as consumers expect reliable products. The introduction of *component technology* has facilitated the design and construction of such systems, but the issue of integration testing remains – indeed the lack of knowledge of component internals may increase the potential for integration faults, c.f. [Trew 01]. Lack of implementation knowledge may translate itself into a nondeterministic view of a component, where several behaviours are acceptable. This, too, increases the complexity of testing. Furthermore, system composability leads to large state spaces from which, for an effective test-suite, an intelligently selected subspace must be distilled – as a separate problem in its own right.

We discuss software testing in more detail in the next main section, and in more detail still in the appendix [StCrContext], where we consider various approaches to automating test execution and test generation. In the present introduction, we focus on the approach that is our main subject matter: state-based testing.

## 1.2 The problem to be considered

One of the most successful approaches taken to software testing is *state-based-testing*. Tests (and their ‘oracle’) can be automatically generated from a model based on a description of state behaviour. The *statechart* concept of [Harel] has made this approach much more manageable than it was previously, with large, flat state machines, and statecharts are now part of standard [UML] dynamic modelling. In this area, Philips Research has in the past helped deploy State Relation Tables [Yule] and Concurrent Hierarchical State Machines [CHSM]. These tools are powerful but they have limitations. Neither can deal with nondeterminism, a factor that is becoming increasingly important. Although Philips Research can demonstrate many techniques to address these issues, they use special, often academic, products such as the Concurrency Workbench [CWB], or LOTOS or PROMELA based tools, such as [SPIN], that would not be suitable for direct use by most testing practitioners. An aim of the present research programme as a whole at Philips is to provide an integrated toolset that is sufficiently easy to deploy for use on development sites. UML is well-known to many software professionals, and the UML dynamic model – the *statechart* – is the basic model from which we wish to derive tests. The broad problem considered is: ***how best to test (composed) systems based on a nondeterministic UML dynamic model.*** We tackle a specific aspect of this problem, ***the design and construction of a UML-statechart based nondeterministic test oracle***, since existing tools for the remainder of a testing tool chain are conveniently already in place, thanks to e.g. the TorX tool chain delivered by the Côte de

Resyste project [CdR]. While constructing our nondeterministic oracle program, we investigate the usefulness of PROLOG as the implementation language for compilation and as a runtime ‘machine engine’.

### 1.3 A peek at the result of the work

The work underlying this thesis has resulted in a state based test oracle program called STATECRUNCHER. *Its main novel and distinguishing feature is its handling of nondeterminism.* In STATECRUNCHER, provision has also been made for component composition at a language level by its scoping operators.

At the time of writing, STATECRUNCHER is being used with the TorX tool chain (which is part of the Côte de Resyste [CdR] project) to derive tests from formal specifications. Philips Research India - Bangalore (PRI-B) is testing software components using this tool chain, illustrated in the following figure:

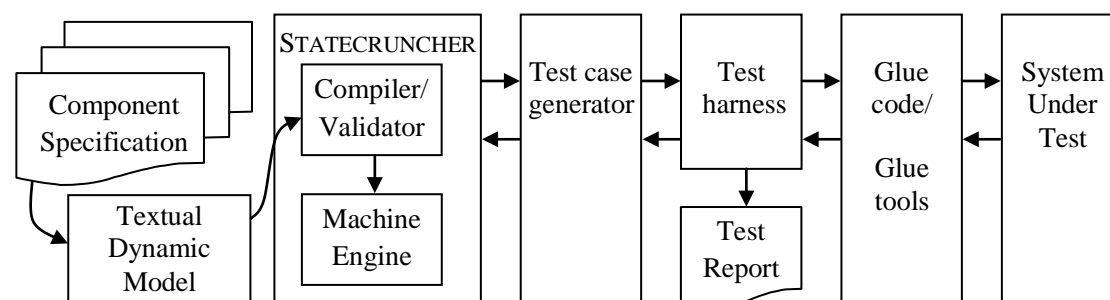


Figure 1. STATECRUNCHER in a tool chain

Experience with this tool chain is described in the concluding part of this thesis (chapter 10).

### 1.4 What STATECRUNCHER is not

Remembering that STATECRUNCHER is a test oracle program, we discuss the issue of what STATECRUNCHER is not, for clarification with respect to related disciplines.

#### *STATECRUNCHER is not a property checker*

We distinguish two kinds of tool: *model checkers* and *simulators/test oracles*. The corresponding activities may be called *property checking* and *testing* respectively. A software system needs a *design* and an *implementation*, and both need a separate kind of tool and activity to ensure the quality of the final system<sup>1</sup>.

<sup>1</sup> *Property checking* is often called software *verification* [Bérard], but others, e.g. [Callahan], effectively equate *validation* with *property checking*, and *verification* with *testing*. Neither [IEEE-610.12.1990] nor [CMMI] makes a clear distinction of the V&V terms along these lines. They should always be looked at in context.



The distinction is as follows:

- The design must guarantee certain properties, e.g. safety, liveness, fairness, freedom from deadlock. Given a formal design, such as a statechart with properties attached to states, and a formulation of the properties required in a system, a model checker can attempt to prove them. Two possible limitations are: the expressiveness of the property language (typically a temporal logic), and the size of the state space (though some techniques allow for vast numbers of states).
- Given a design, the system must be implemented, and the implementation tested. Televisions, mobile phones etc. are a combination of hardware and software. The concept of *being in a state* means much more to a real system than to a simulator: mobile phone transmitters may be switched on, threads may be waiting for semaphores, buffers should have certain content, such as a teletext page. Testing involves making sure that these things that should happen really do happen. The state model tells us what it is that should happen.

A slogan popular in Philips in the 1990s was: *Doing the right thing and doing things right*. This is like saying: checking the properties of the design, and testing that the implementation conforms to the design. Both are extremely important, but distinct.

Despite the above, model checking tools necessarily have state exploration capabilities, whether by exhaustive search or algebraic manipulation, and some tools offer verification and simulation facilities, e.g. [SPIN]. For an interesting combination of tools, using a property checking tool to generate state-based test inputs, see [Jagadeesan].

STATECRUNCHER was designed as a *test oracle*, and the thrust of the main thesis is that its design will help in *testing*. Nevertheless it could be used to verify properties, given the aid of an additional tool communicating with it, because it offers facilities which will help in exploring state spaces. However, STATECRUNCHER is probably not a very efficient tool for this purpose.

The appendix with a bibliography [StCrBiblRef] includes many references to property checking because it is a closely related field to testing.

### ***STATECRUNCHER is not a test generator***

There are two concerns in state based testing that can usefully be separated out: (1) determining *what test* to perform and (2) obtaining an *expected result* (an *oracle*) to that test. A tool for the first is a *test generator*; a tool for the second is a *simulator* or *oracle program*. STATECRUNCHER belongs to the latter category. We mention test generation techniques in section 3.2.4 and include many annotated references in our appendix [StCrBiblRef], since it is an important related subject.

## 1.5 The structure of this thesis

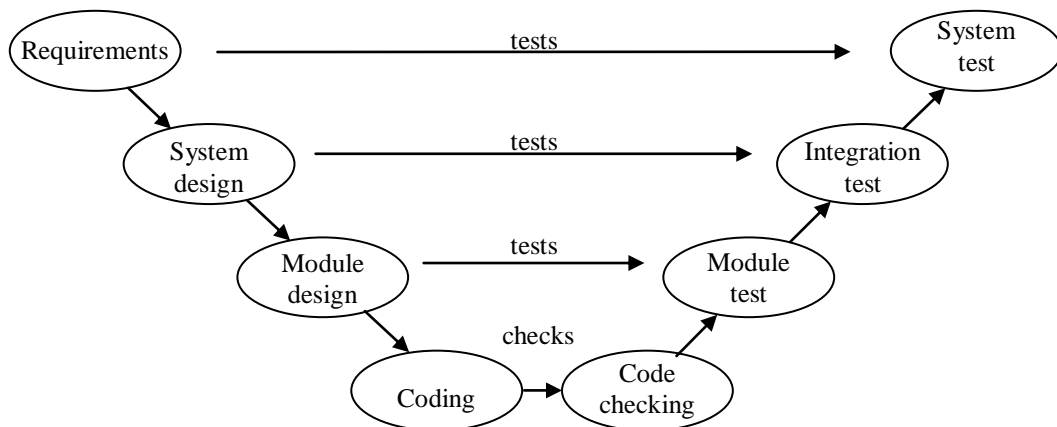
We first put software testing in context. Then we introduce the concepts of state behaviour and state-based testing, with an introduction to STATECRUNCHER's role in this. *Special attention is given to handling of nondeterminism, as this is the main novel feature in the system.* The subsequent section covers the syntax of STATECRUNCHER in more detail. This is followed by a discussion of approaches to detailed transition semantics, and the chosen transition algorithm is described in depth. Since STATECRUNCHER is intended to work with other tools, its command-level interface is explained. Finally, the deployment of STATECRUNCHER at Philips is discussed, and the PROLOG-based implementation technology is reviewed. There are various appendices to this thesis to support many of the discussions in more detail, including a comparison of STATECRUNCHER's semantics with those of some process algebras. There are also many “related reports”, based on Philips reports produced in connection with the work. These are listed under the references.

STATECRUNCHER has been implemented in PROLOG, but the ordinary user need not be aware of this, because STATECRUNCHER has its own syntax which is independent of PROLOG. Nevertheless, the author feels that some samples of PROLOG code, for some key algorithms, are valuable for the record, and they have been included in this thesis.

## 2. Software testing in context

In this section we describe the various kinds of software testing activities and what the aim is in each case. This will give a context to our main theme of state-based testing. For a more detailed discussion of what kind of testing is applicable under what circumstances, the reader is referred to appendix [StCrContext].

The V-model for the software development life-cycle is well-known from standard works on software engineering. The testing phases of this model are shown in Figure 2.



**Figure 2. V-model and testing**

The V-model identifies various kinds of testing activity, and each has its own emphasis. We consider the aims of and techniques for each form of testing, starting at the bottom of the V-model and working up the right-hand side:

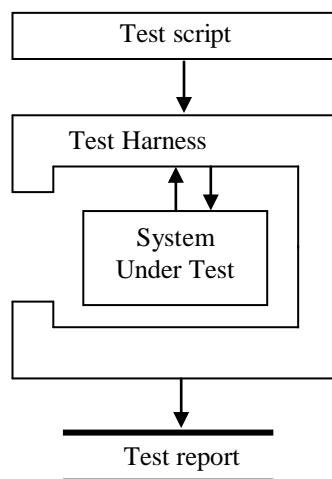
- **Code checking in general:** Static analysis can reveal bad coding style and possible pitfalls. Dynamic techniques can check for memory leaks and can provide code coverage, such as statement coverage, described in more detail in [StCrContext].
- **Module testing:** The question to be answered is: Does the implementation correspond to the design? Modules are usually single functions, or a small number of tightly coupled functions designed against a single specification. Exercise code statements and branches. Use code instrumentation to check for coverage of these. Also include a memory leak check in the tests. Module testing is typically *white-box testing* - we have a knowledge of the code structure and use it to guide us in designing test cases, and we have detailed *controllability* and *observability* of the module.

- **Integration testing:** The question to be answered is: Is the design internally consistent? Exercise interfaces between modules. Measure call-pair coverage (i.e. every call and every return from it). Integration testing is typically **black-box testing** - some modules may even be only available as object code, and the only way we can test the integrated system is via the published interfaces.
- **System testing.** The question to be answered is: Does the system satisfy the project requirements? This will typically be a **black-box testing** activity, since the requirements do not normally specify internal controllability and observability, but rather the operations and their outputs to which the end-user has access. For some kinds of system, a part of system testing will be **volume testing**. For example, a set-top box will need to be tested with large quantities of MPEG streams, and a Global Positioning System will need to be tested with large quantities of sampled radio-front-end (intermediate frequency) satellite data.

All the above testing phases are suitable for at least some automation. There are two levels of test automation: automated test *execution* and automated test *generation*.

### **Automated test execution**

The first level of automation is to be able to run tests automatically and have a test report produced. Tests are preferably called in a uniform way, and each test should provide its own pass/fail criterion. The test report should produce a uniform description of whether each test passed or failed. A tool providing facilities for doing this is called a *test harness*. We can picture automatic execution of tests as follows:

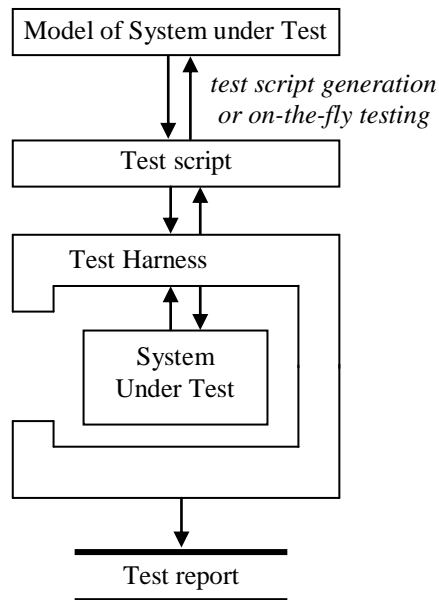


**Figure 3. Automated test execution**

There are good commercial and public domain test harnesses. A Unix-based public-domain test harness with which Philips Research has considerable experience is **Deja Gnu** [DejaGnu]. A commercial tool for GUI-driven testing under Windows NT with which Philips Research is also familiar is **WinRunner** [WinRun]. A Philips Research tool to give an

(embedded) multi-threaded application a GUI so that it can be tested using WinRunner is **GFET** [GFET].

A second level of automation is *automated test generation*. In this case we have some formal specification or model of the system to be tested. From that we derive tests, either as a batch or dynamically during testing.



**Figure 4. Automated test generation**

The kinds of model that are most used for automated test generation are:

- A state behaviour model, or *statechart*, such as the UML dynamic model
- A cause-effect graph (or a decision table, which is a simple form of cause effect graph)
- A grammar of a language or protocol for syntax testing
- Orthogonal arrays for parameter/property interaction testing.

The next section of this thesis focusses on state behaviour and modelling. The other techniques are described briefly at the end of that section. More detail on them is given in the appendix [StCrContext].

In addition to being aware of *model-based* testing techniques, the tester should be aware of other technical considerations in ensuring adequate testing, such as a static and dynamic analysis of *code properties*. We have mentioned measuring the degree of statement and branch coverage exercised in a test suite, (preferably using code instrumentation techniques); this gives guidance on how to design more tests to cover unexecuted statements and branches. Similarly *data flow* analysis techniques examine the declaration, write-usage, read-usage, and destruction of variables, signalling any anomalies. These and related techniques are well described in [Beizer] and [BCS-SIGIST].

### ***Summary of this section***

We have seen that different forms of testing are applicable in different phases of the V-model. Code can be statically analysed and instrumented for dynamic checking and coverage measurement. Testing is more efficient when automatically *executed*, and for this we use a test harness, and define all tests in a uniform way, where each test defines its own pass/fail criterion. Results are logged to a test report. Further gains are made when we automatically *generate* tests, using a model of the system under test. We mentioned state behaviour models, cause-effect graphs, grammars, and the use of orthogonal arrays. These will be described in the next section, with a heavy emphasis on state behaviour models, since that is the area we focus on in this thesis.

## 3. State-based testing and STATECRUNCHER overview

In this section we consider what is meant by a system state and an event, both from the perspective of a mental model of a system, and from the microscopic computer hardware perspective. We show how a model of state behaviour can be used in test generation and execution. The question of how to represent the model is addressed, leading to the concept of a statechart. Then we introduce STATECRUNCHER as a statechart system, restricting ourselves to deterministic situations while we introduce the fundamental features. White box and black box testing issues are addressed. Nondeterministic testing is mentioned, but details are reserved for the next section, as this is a major topic. STATECRUNCHER cannot perform testing on its own, and we mention how it can fit into the TorX tool chain as an example of a complete testing tool chain. We conclude the section with a brief look at alternative testing approaches to state-based testing.

### 3.1 States and events

Many systems can be modelled according to their state behaviour – that is their *state* and how the state changes as a result of some stimulus or signal, which we call an *event*.

Under this modelling technique, if a system is “in a particular state”, it will remain so indefinitely until an event occurs. In other words, the notion of a state entails *durability* - the state exists over a period of time. Even if a system enters a particular state *s1* and there is an event ready and waiting to cause a change of state (say to state *s2*), we still regard the moment when the system is in state *s1* as a point at which the system has become stable in terms of its state behaviour. At such a point, the state of the system (in a wide sense) will map to a state in our model of the system.

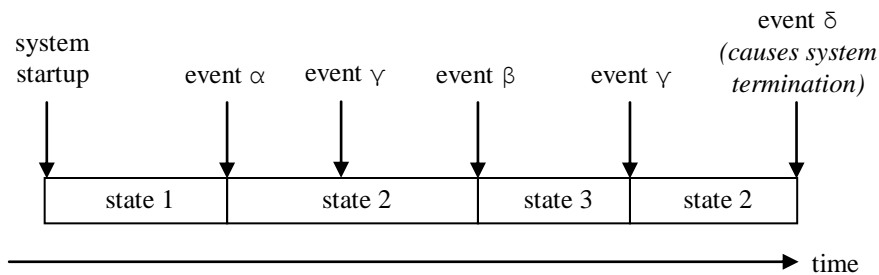
*Events* are modelled as instantaneous signals which have no duration. They are able to trigger some processing in the system which may or may not result in a new state. In some states, events may effectively be ignored by the system without any further processing at all, so leaving the system in its previous state.

While the system is processing an event, at a modelling level we do not talk about its state, while still recognising that the system will assume ‘states’ at a detailed level which we do not model. At a modelling level we regard processing an event ideally as an infinitely fast and atomic activity, whilst recognising that real-world implementations require time to process events.

If an event would appear to require duration, the situation should probably be modelled by two events (*start x* and *stop x* events) and an intermediate state (*doing x*).

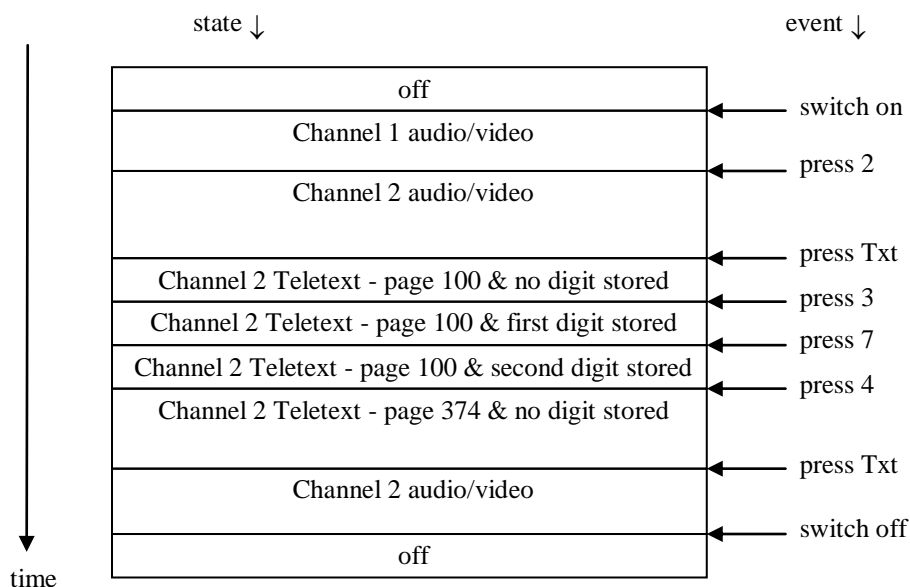
A system may be of the kind that theoretically runs indefinitely, such as an operating system or real-time kernel, or it may have a clear lifecycle. But even operating systems can generally be closed down in a controlled way.

A simple picture of a system state lifecycle under a specific set of events, (so *not* a state transition diagram, which will be introduced later) is as follows:



**Figure 5. Specific system-life-cycle – abstract example**

As a concrete example, we take using a television (in a simplified way - for example tuning and teletext page acquisition are regarded as instantaneous). Here we place the time axis vertically



**Figure 6. Specific system-life-cycle – concrete example: Television**

We have a concept of a state as something durable until an event is presented and processed. Systems characterized by this kind of behaviour are called *reactive systems*, since they do



nothing until they react to an event. For a computer system, this suggests that the system is actually idle (as regards machine processing cycles) when it has settled into a state. However, this need not be the case. For example, a multi-threaded application might be modelled with states which represent the fact that low priority threads are running - such a system would still be able to react to events which interrupt at a higher priority. It may even be necessary to represent cpu-bound tasks as states, perhaps using several states so as to model events as having been recorded but unable to be processed until the task completes.

Input data to a program can also often conveniently be thought of as a sequence of events. In this case, the program will normally have instant access to the “next event” (apart perhaps from an occasional disc-access), and so will be cpu-bound, but this does not detract from the state model. An example of such a kind of program is a compiler where the input tokens can be regarded as events; the state is some record of completed successful parsing of ‘terms’ in production rules.

We can ask the question: what does it mean to say a computer system is in a particular state? The system modeller may distinguish states according to a mental model of the system, or according to situations (such as use-case situations) from the requirements or specification documentation.

It should be possible to distinguish in the system implementation between states which the modeller has defined *somehow* - either by direct observation of the system, or by examining the system behaviour as further events are presented and processed. If two states show *identical* responses to any sequence of events that is processed from a system in such a state, then they are indistinguishable and are best modelled as one state, so as to avoid redundancy in the model.

Conversely, if a particular state has been defined in a model, that state must show *identical* behaviour as regards its response to further events, irrespective of how that state was arrived at by preceding sequences of events.

As an initial modelling technique, we consider a system as being in just *one* state at any one time. This will be extended later.

We can also describe the state at a microscopic level. A computer application, based on binary memory and registers, at the finest level of detail, has as many states as bit patterns in its memory and registers (e.g. program counter, accumulator, working registers, overflow and carry indicators, interrupt registers, device registers, system clock) - as far as these can impinge on the application - in other words  $2^N$ , where N is the number of bits in all this memory and registers. The *macroscopic* states that a system modeller defines are equivalence classes of the *microscopic* states.

## 3.2 Deterministic state-based testing

Deterministic systems always process an event from a given state in the same way. Nondeterministic systems show alternative permissible outcomes. This is usually due to working at a level of abstraction at which detailed system information is lacking, or because of limited control and observation of the IUT (Implementation Under Test). We first consider the deterministic case.

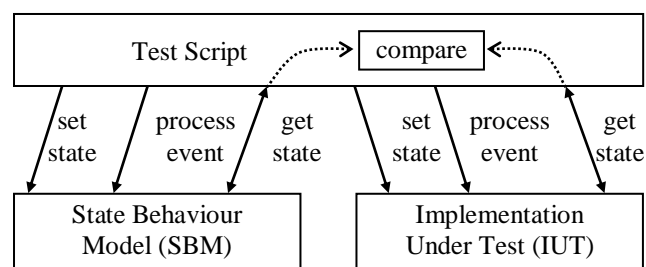
When states are controllable (i.e. we can directly set any state in the IUT), and are observable, we have the *white-box* situation. If states can only be set by driving the system through a transition sequence to reach them, and if states must be deduced from system output produced on transitions (traces), then this is a *black box* situation. We consider these in turn.

### 3.2.1 White box testing

We wish to exercise all events under all state configurations. For a state machine consisting of three parallel machines, we wish to execute the following pseudo-code:

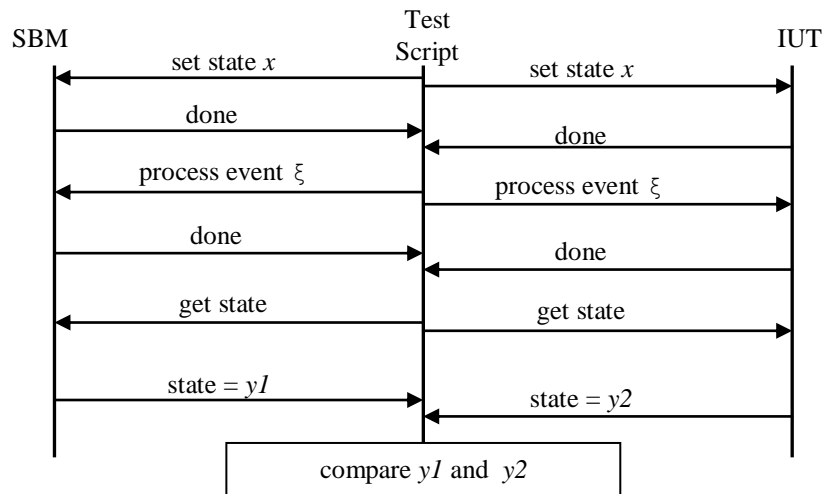
```
For each state i in parallel machine 1
  For each state j in parallel machine 2
    For each state k in parallel machine 3
      For each event
        {
          Put machine 1 in state i
          Put machine 2 in state j
          Put machine 3 in state k
          Process event
          Check IUT is in correct state
        }
      }
```

The oracle comes from some executable state behaviour model (SBM). The process of sending instructions to the SBM and IUT is illustrated in the following figure:



**Figure 7. White-box state-based testing**

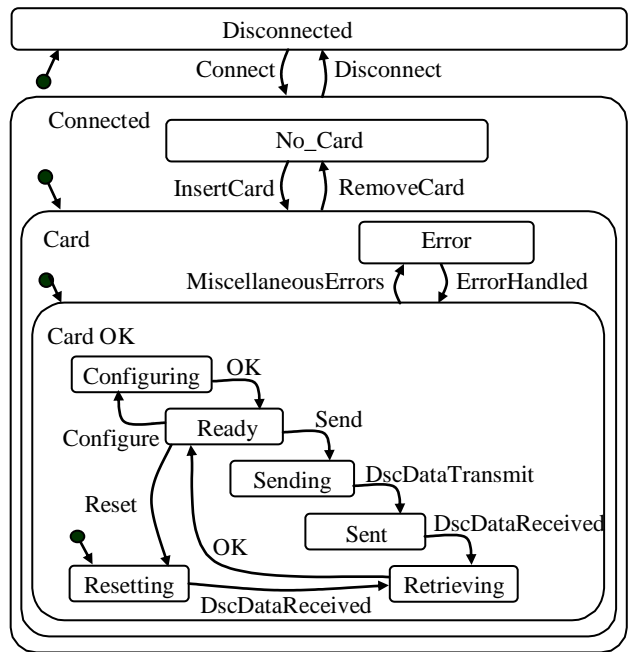
A typical message-sequence diagram of the testing process is as follows:



**Figure 8. Message Sequence Diagram for White-Box State Based Testing**

There is an issue as to whether the “For all events” loop should refer to all events that are *transitionable* (i.e. they will trigger some transition) from the state as set in the outer loop, or to all events in the model *absolutely*. A possible problem with the latter is that some tests may be hard to run, or be unrunnable. This might be because a certain event cannot be offered to the implementation for processing in certain states. For example, one cannot press a button on a GUI (graphical user interface) if that button is not present in some context (though one can verify that the button really is absent). As another example, one cannot call a function on a certain thread if that thread is currently executing another function. So certain tests may have to be excluded. A caveat to the tester is that when a designer or developer says “the program logic precludes the situation where this event is offered to this state”, the tester should verify this before accepting it, by some form of testing and/or by a code review.

How should the state behaviour be represented? Early work used a *state-relation table*, or SRT, in which entries in the first columns define initial states, a middle column contains the event, and the latter columns define final states, i.e. states after processing the event. The use of wildcards can help keep the table size reasonable. An SRT has affinities with a decision table. At Philips, a program by David Yule has been used to obtain an oracle to state behaviour this way, to test *inter alia* a DVD player and a set-top box. As an example (without parallelism), the figure below shows a dynamic model of a smart card reader, followed by part of an SRT representing it.



**Figure 9. Dynamic model of smart card reader**

The state-relation table below represents part of the above model, using the notation “?” for a wildcard, and “#0” for *as in the first column*.

Start State	Event	Result State
Disconnected	<i>Connect</i>	No_Card
Disconnected	?	#0
?	<i>Disconnect</i>	Disconnected
No_Card	<i>InsertCard</i>	Resetting
No_Card	?	#0
?	<i>RemoveCard</i>	No_Card
Error	<i>ErrorHandled</i>	Resetting

**Table 1. Partial state-relation table for a smart card reader**

A disadvantage to state-relation tables is that they are hard to maintain (“write only”). What is needed is something that users can more easily relate to the *diagram* of a statechart.

### 3.2.2 Statechart systems

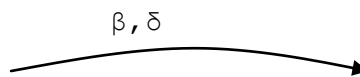
A diagram showing states and transitions is called a *state-transition diagram*. Statecharts extend the basic notion with hierarchical structure, to be described in detail later, but evident in Figure 9, which is a statechart. Such a representation provides a compact and intuitive means to express all the relationships between states, events, and new states after processing

the event. Statecharts were first proposed and used by David Harel [Harel]. We now consider the primitives of a statechart in more detail.

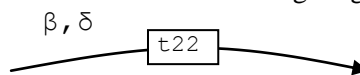
A *transition* is what maps a source state and *event* to a new state (the target or destination state). We say the event *triggers* the transition.

States are conventionally denoted by circles or rounded boxes, and transitions by arcs with an arrowhead. Transition arcs are normally annotated with the *events that trigger the transition* (not with transition names). The present author frequently adopts for compactness the convention of [CHSM] in using Roman-letter names for states and Greek-letter names for events in an abstract model. Transitions are often not named – they are normally referred to as “the transition on event *some event*”, qualified by the source state if necessary to avoid ambiguity.

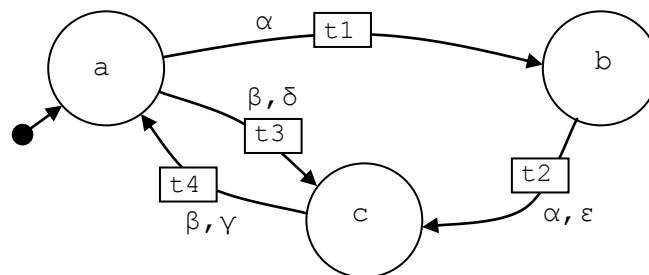
A transition triggered by events  $\beta$  or  $\delta$  is drawn as follows:



To explicitly name a transition, we will use the following diagrammatic convention:



We now give an elementary example of a state-transition diagram.



**Figure 10. Elementary state-transition diagram**

The above diagram models a system as having:

- three states: a, b and c. The initial state is a (symbol ).
- five events:  $\alpha$ ,  $\beta$ ,  $\gamma$ ,  $\delta$  and  $\epsilon$ .
- four transitions: t1, t2, t3 and t4.

At any one time, a system modelled by the above state-transition diagram will be in one and only one state. That state is called the *occupied* (or active) state. The others are *vacant* (or inactive).

Transitions whose source states are vacant (at the time an event occurs) do not cause any state transitioning to take place – they are *inapplicable* (or invalid) in the current state.

If an event occurs which is the trigger to a transition whose source state is occupied, then (apart from exceptional situations<sup>1</sup> to be considered later) the transition takes place. The source state is vacated and the target state is occupied.

In the above example, when the system is in state a, it will react to event  $\alpha$  by executing transition  $t_1$ , i.e. by transitioning from state a to state b. If the system is not in state a, then transition  $t_1$  is not applicable because the system is not in  $t_1$ 's source state. Only one transition takes place as a result of one occurrence of this event, so transition  $t_2$  does not take place as well, unless (and, in this case, until) another event ( $\alpha$  or  $\epsilon$ ) occurs. Notice that:

- there can be several transitions emanating from any state (for example  $t_1$  and  $t_3$  from state a).
- an event can be a trigger to more than one transition (for example  $\alpha$  triggers  $t_1$  and  $t_2$ ), but, (until we consider nondeterminism), we do not expect to find two transitions triggered by the same event from the same source state.
- a transition can be triggered by more than one event, in which case *any one* of the events will trigger the transition. For example, transition  $t_3$  is triggered by event  $\beta$  or  $\delta$ .

If an event occurs which does not trigger a transition, (for example if in state b event  $\beta$  occurs), then the event is disregarded and no state change occurs. This is not an indication of an error. Indeed, if such an event does represent an error in a system, then the state-transition diagram should model the error-handling, for example with a transition to a new state 'error'. There is then nothing special about a state called 'error' except its interpretation.

The way in which the state transition diagram of Figure 10 is represented in the STATECRUNCHER language is:

```
statechart sc(s)
  event alpha,beta,gamma,delta;
  cluster s(a,b,c)
    state a {alpha->b;beta,delta->c;}
    state b {alpha,epsilon->c;}
    state c {beta,gamma->a;}
```

The syntax will be fully explained later. For the moment, observe that the state transition diagram is declared as a “statechart”, which consists of a cluster *s*, which consists of three (leaf-) states a, b, and c. A cluster indicates a grouping in which no more than one member state can be occupied (the XOR-state of [Harel]). Events are declared and are used in transitions, which are denoted by

---

<sup>1</sup> e.g. hierarchical prioritisation, where an inner transition masks an outer one or vice versa

*events -> target state;*

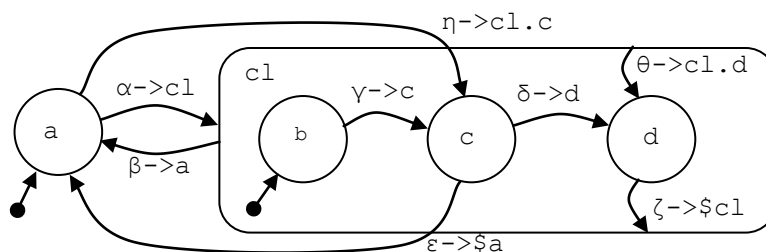
State behaviour modelling is part of the UML (Unified Modelling Language) dynamic view. It is not particularly onerous to prepare STATECRUNCHER models using a text editor. But an alternative way might be to use CASE (Computer Aided Software Engineering) tools to draw the diagram, and use them to export the state machine view in textual form. Utilities could then be written as necessary to convert exported descriptions to STATECRUNCHER code.

A good public domain tool that relates well to statechart diagrams, supporting hierarchy and concurrency, is CHSM by Paul J Lucas [CHSM]. It generates a C++ class having the same behaviour as the statechart. As such, the class behaves consistently, even if the statechart is nondeterministic. CHSM has been used for testing at Philips, and it provided the inspiration and a basis for the extended system, which is the subject of this thesis. The main extension to be discussed is alternative outcomes under *nondeterminism*, but we begin with some more basic concepts.

The hierarchical structures supported by statecharts are hierarchy and parallelism, which lead to the concept of a *cluster* and *set*. If a cluster is occupied, then exactly one of its member states must be occupied. If a set is occupied, all its member states must be occupied (the AND-state of [Harel]). The members may be leaf-states, or clusters or sets themselves.

### Clusters

The following figure illustrates a *cluster*, with the source code in STATECRUNCHER (which is similar to that of CHSM).



**Figure 11. Cluster and transition target notation [model t4160]**

Source code:

```

statechart sc(sys)
event alpha, beta, gamma, delta;
event epsilon, zeta, eta, theta;
cluster sys(a,cl)
  state a      {alpha->cl; eta->cl.c;}
  cluster cl(b,c,d){beta->a;  theta->cl.d;}
    state b    {gamma->c;}
    state c    {delta->d;  epsilon->$a;}
    state d    {zeta->$cl;}
  
```

The syntax of STATECRUNCHER is such that target states are by default a sibling of the source state. Non-sibling target states need more precise specification than just their name, giving their *scope*. Parent scope is specified using the operator "\$". A grandparent scope would be designated by "\$\$". Descent into child states is achieved using the operator ".". Grandchildren would be designated using this operator twice, e.g. `cl.d.grch`. Note that on event `theta` a transition will take place from anywhere in cluster `cl` to member state `d`.

On loading this model, STATECRUNCHER will enter the default state and give the following output:

```

2  statechart sc
2      cluster sys [sc] = OCC []  **
2          leafstate a [sys, sc] = OCC []  **
2          cluster cl [sys, sc] = VAC []
2              leafstate b [cl, sys, sc] = VAC []
2              leafstate c [cl, sys, sc] = VAC []
2              leafstate d [cl, sys, sc] = VAC []
2  TRACE =[]
2  TREV [[alpha, [sc]], 0, [], []]
2  TREV [[eta, [sc]], 0, [], []]

```

States are indicated with their position in the hierarchy and their occupancy. Occupied states are emphasized by a double asterisk. The output also shows TRansitionable EVent information, i.e. what events can be responded to, (with some other details not discussed right now). On processing event `eta`, the following output is obtained:

```

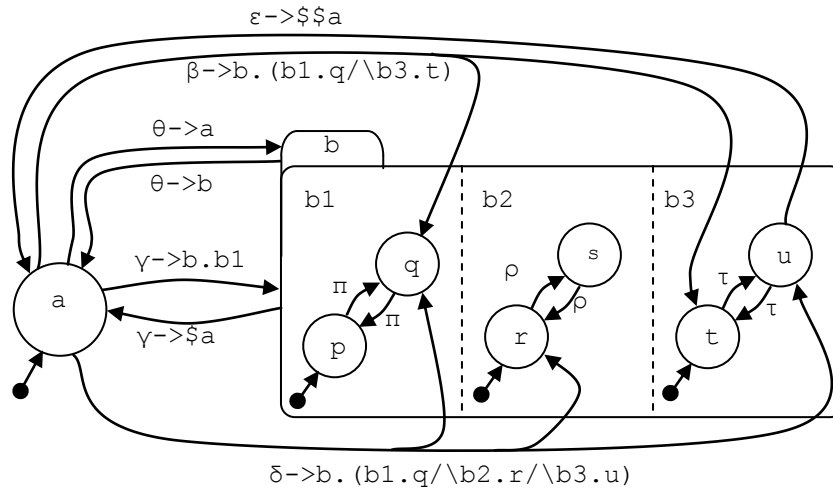
3  statechart sc
3      cluster sys [sc] = OCC []  **
3          leafstate a [sys, sc] = VAC []
3          cluster cl [sys, sc] = OCC []  **
3              leafstate b [cl, sys, sc] = VAC []
3              leafstate c [cl, sys, sc] = OCC []  **
3              leafstate d [cl, sys, sc] = VAC []
3  TRACE =[]
3  TREV [[delta, [sc]], 0, [], []]
3  TREV [[epsilon, [sc]], 0, [], []]
3  TREV [[beta, [sc]], 0, [], []]
3  TREV [[theta, [sc]], 0, [], []]

```

### **Sets**

A *set* is illustrated in the figure below, with STATECRUNCHER source code following:





**Figure 12. Set and transition target notation [model t4170]**

Source code:

```
statechart sc(sys)
event beta, gamma, delta, epsilon, theta;
event pi, rho, tau;
cluster sys(a,b)
  state a      {theta->b; gamma->b.b1;          \
                beta-> b.(b1.q/\b3.t);        \
                delta->b.(b1.q/\b2.r/\b3.u);}
  set b(b1,b2,b3) {theta->a;}
  cluster b1(p,q) {gamma->$$a;}
    state p      {pi->q;}
    state q      {pi->p;}
  cluster b2(r,s)
    state r      {rho->s;}
    state s      {rho->r;}
  cluster b3(t,u)
    state t      {tau->u;}
    state u      {tau->t; epsilon->$$a;}
```

When members of sets are clusters (as they often are), the rounded rectangle for the cluster can be omitted. In defining transitions, strictly one should distinguish targeting the set as a whole (as is done by a transition on *theta*), and targeting a single member, as is done by a transition on *gamma*). But in practice there is no difference, because targeting the whole set entails entering the default state in each member, and targeting just one member entails entering that member and, (in order to maintain integrity of the set occupation rule) the remaining members too.

When targeting sets, individual states in different members can be specified, using the split operator, `"/\`". The transition on *beta* does this, though it does not specify a target in *all* members. Where no explicit target is specified, the default is taken.

On entering the initial state, the STATECRUNCHER output is:

```
2 statechart sc
2   cluster sys [sc] = OCC [] **
2     leafstate a [sys, sc] = OCC [] **
2     set b [sys, sc] = VAC []
2       cluster b1 [b, sys, sc] = VAC []
2         leafstate p [b1, b, sys, sc] = VAC []
2         leafstate q [b1, b, sys, sc] = VAC []
2       cluster b2 [b, sys, sc] = VAC []
2         leafstate r [b2, b, sys, sc] = VAC []
2         leafstate s [b2, b, sys, sc] = VAC []
2       cluster b3 [b, sys, sc] = VAC []
2         leafstate t [b3, b, sys, sc] = VAC []
2         leafstate u [b3, b, sys, sc] = VAC []
2 TRACE =[]
2 TREV [[theta, [sc]], 0, [], []]
2 TREV [[gamma, [sc]], 0, [], []]
2 TREV [[beta, [sc]], 0, [], []]
2 TREV [[delta, [sc]], 0, [], []]
```

On processing event beta, the output is:

```
3 statechart sc
3   cluster sys [sc] = OCC [] **
3     leafstate a [sys, sc] = VAC []
3     set b [sys, sc] = OCC [] **
3       cluster b1 [b, sys, sc] = OCC [] **
3         leafstate p [b1, b, sys, sc] = VAC []
3         leafstate q [b1, b, sys, sc] = OCC [] **
3       cluster b2 [b, sys, sc] = OCC [] **
3         leafstate r [b2, b, sys, sc] = OCC [] **
3         leafstate s [b2, b, sys, sc] = VAC []
3       cluster b3 [b, sys, sc] = OCC [] **
3         leafstate t [b3, b, sys, sc] = OCC [] **
3         leafstate u [b3, b, sys, sc] = VAC []
3 TRACE =[]
3 TREV [[pi, [sc]], 0, [], []]
3 TREV [[rho, [sc]], 0, [], []]
3 TREV [[tau, [sc]], 0, [], []]
3 TREV [[gamma, [sc]], 0, [], []]
3 TREV [[theta, [sc]], 0, [], []]
```

The rule for set occupancy is seen, with each member cluster (b1, b2 and b3) being occupied.

### 3.2.3 Additional (deterministic) features

A summary of additional enhancements to the basic idea of a statechart is now given. These are illustrated in STATECRUNCHER syntax, but the features are not unique to STATECRUNCHER. It should be borne in mind that these are introduced for user convenience (as with the cluster and set structures). Any finite model can be “flattened” to an equivalent leafstate-only model, but for any sizeable statechart, the flattened model is totally unwieldy.

*Internal events* are generated when any state is entered or exited. So it is possible to have a transition as follows, where  $\$x.y$  is some parallel state (addressed relative to the parent of state b).

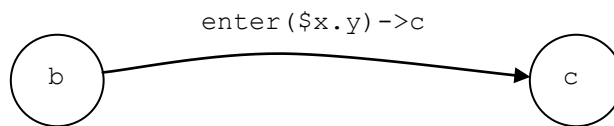


Figure 13. Internal event

*Variables* can be defined and assigned to expressions on state entry or exit (the triangles pointing in or out of a state make for a compact notation, but UML uses keywords *entry/* and *exit/* inside the state). Assignments can also be on transitions. STATECRUNCHER allows for integer ranges and enumerated types, booleans, and strings.

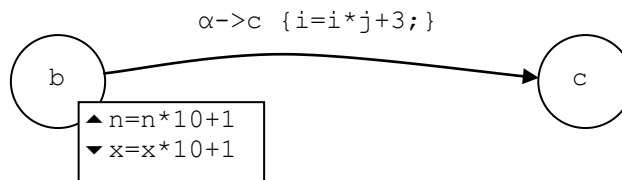


Figure 14. Variable assignment

Variables and events can also be declared locally to a part of the hierarchy and be addressed with scoping operators. The operators have high precedences and can be used in arithmetic expressions, e.g.  $n=i+\$j+s.t.k$ . If this assignment is found on a transition,  $n$  and  $i$  are in the scope of the source state of the transition,  $j$  is in the scope of the parent of the source state, and  $k$  is in the scope of child  $t$  of sibling  $s$  of the source state. A library of functions (such as *maximum*) is also provided.

Transitions can be *conditional*. The conditional expression in square brackets will evaluate to a Boolean value (but as in the ‘C’ language, 0 is taken as false and nonzero is interpreted as true). The expression may refer to the occupancy of another parallel state, using the  $in()$  function, as in the example below. This gives the equivalent of multiple source states of a transition.

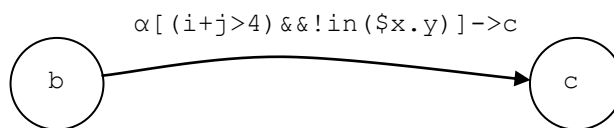
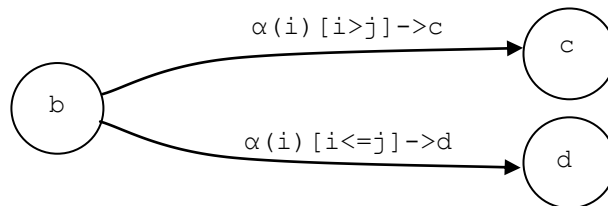


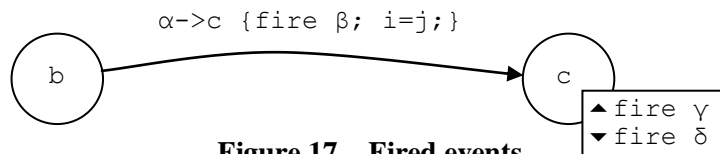
Figure 15. Conditional transition

Events can be *parameterized*. The destinations for the parameters are listed in round brackets. A parameter may be used in the condition of the transition triggered by the parameterized event. In this example, care has been taken that there should be no nondeterminism.



**Figure 16. Parameterized event**

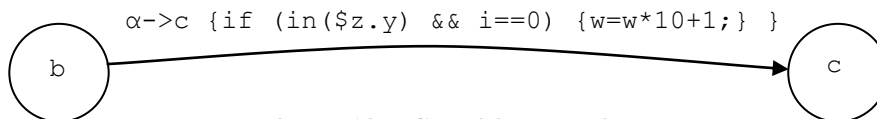
Events can be *fired* on state entry or exit, or on transitions. Fired events and variable assignments are examples of what STATECRUNCHER calls *actions*. Some parallel part of the statechart will respond to the fired events if that is applicable.



**Figure 17. Fired events**

There can of course be several actions on a transition or on entry or exit. An assignment has been included in the above figure to show this. The exact ordering of actions is a semantic issue, discussed later, with good arguments being made for various alternatives.

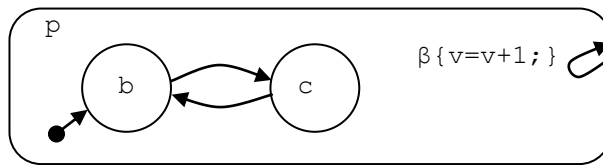
Actions can be *conditional*. This is a separate matter to transitions being conditional. In the figure below, the transition is unconditional, but the action is conditional.



**Figure 18. Conditional action**

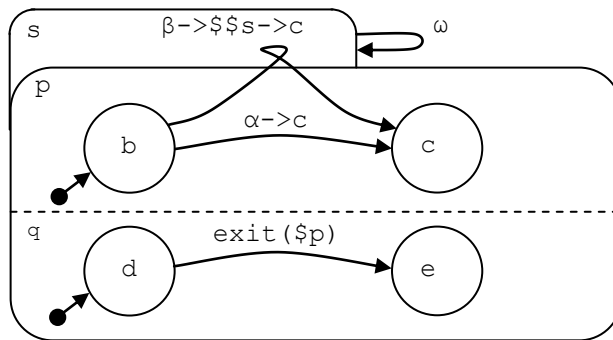
Conditional actions can also have an else part, and the if-actions and else-actions can themselves be conditional (not illustrated).

Transitions can be *internal*. This means that there will be no state change, but any actions on the transition will be executed. In Figure 19, on event beta the internal transition will take place provided cluster p is in the occupied state.



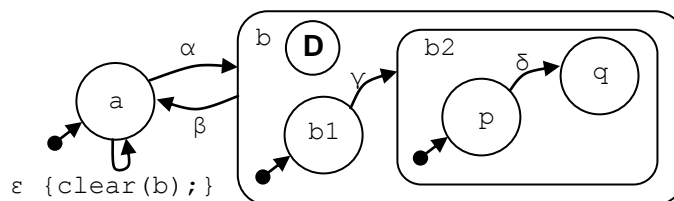
**Figure 19. Internal transition**

Transitions can have an *orbital* trajectory. In the figure below, the transition on event  $\beta$  causes cluster  $p$  to be exited and re-entered, whereas the transition on event  $\alpha$  does not. This is reflected in the resultant occupied member of cluster  $q$ . Orbits can be to any height in the hierarchy, and are specified as *event*->*orbital-state*->*target-state*. In the diagram, the loop in the transition arc emphasizes the orbit.



**Figure 20. Orbital transition**

When a cluster is exited, the member that was occupied is stored as the *historical* state. UML uses pseudo-states to indicate entering clusters either recursively (deep history) or just at the top cluster level. History can be (deep-) cleared. STATECRUNCHER currently marks a cluster with a (deep-) history marker (as in CHSM), indicating how the cluster is to be entered if a transition targets it. A deep-history cluster can be shallow-history-entered by deep-clearing its child history, or default-entered by deep-clearing its own history. UML's pseudo-states may be implemented in the future, where transitions can individually specify whether a (deep-) historical state is to be entered or not. In Figure 21, the transition on  $\alpha$  will cause the historical states of cluster  $b$  to be entered. Initially this is state  $b1$ , but if the last occupied states were  $b2$  and  $q$ , then these would now be entered. However, event  $\varepsilon$  clears  $b$ 's history, and if this has happened since exiting  $b$ , then the transition on  $\alpha$  will target member  $b1$ .



**Figure 21. (Deep-) History**

### 3.2.4 Black box testing

With *white-box* testing, we assume the state and variable values in the IUT (Implementation Under Test) are observable. In the *black box* case, this is not so, and only sequences of outputs, called *traces*, are observable. The basic testing paradigm is as shown in the figure below (compare with Figure 7).

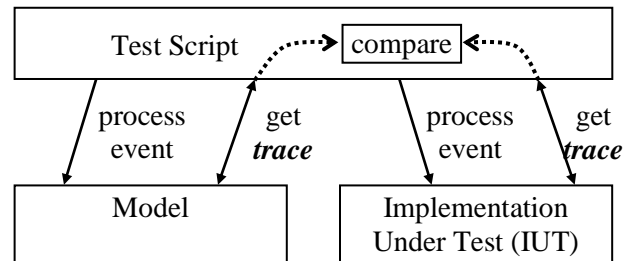


Figure 22. Black box testing - compare traces

Trace elements can be produced wherever an action is allowed: on transitions, on state entry and on state exit. Some transitions may not produce any output, or produce the same output that other transitions produce. For this reason, a transition tour, (where all transitions are taken, and output from the tour is verified, but where that is all), is not a strong test. This is also known as the *Chinese postman* tour, after a publication by [Kwan] in 1962. Stronger testing can become quite difficult, involving transfer sequences to each state, with further event sequences to be executed in order to verify that the system is in the expected state. States can be checked in various ways; for deterministic systems, the best-known methods are:

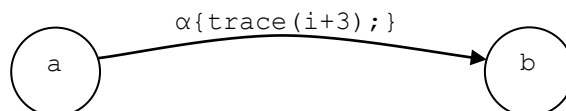
- the D-method, or *distinguishing method*, where a sequence of events is sought such that the output produced distinguishes all states. A distinguishing sequence might not exist. The concept was known to [Hennie] in 1964.
- the W-method, also known as the *characterizing set method* [Chow], where a set of event sequences are sought which collectively identify the state. A disadvantage is that in general the state under investigation must be regenerated many times so that each member of the characterizing set can be applied to it.
- the U-method, or *unique I/O sequence method* [Sabnani], where an event sequence is sought for the expected state, which distinguishes this state from any other state, without necessarily identifying the actual state in the case of mismatch.

Some of the methods are often considered impracticable, due to exponential calculation time with the size of the machine, or the sequence may not exist, (D and U methods). There are many optimizations to the basic algorithms in the literature, sometimes making extra assumptions about the state machine. For an overview of test sequence generation, see [Lee], [Dahbura], and the Philips report by [Koymans].

Although most theoretical articles describe a finite state machine in terms of a machine without hierarchy or parallelism, a concurrent hierarchical statechart can be *flattened* (or *unfolded*), since any configuration of state occupancies, variable values and historical states

can be regarded as a single flattened state. So the theoretical results are fully applicable to statecharts.

The figure below shows a trace of an expression value on a transition.



**Figure 23. Trace on a transition**

All traces recorded in this way are part of the output STATECRUNCHER produces per world when given a command to do so, e.g.

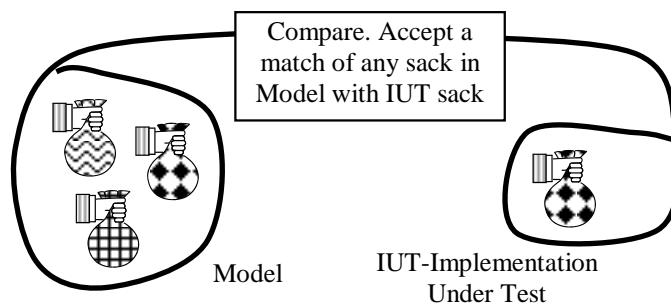
```
3 TRACE =[44]
```

### 3.2.5 Points of control and observation

When testing distributed systems, or systems with restricted observability and controllability, it is useful to categorize events (and traces) according to their PCO –Point of Control and Observation. PCOs are defined in [ISO 9646-1]. When STATECRUNCHER lists transitionable (and other) events, it includes their PCO. Traces are under user control and can contain an indication of the PCO that produced them.

## 3.3 Nondeterministic testing

The distinguishing feature of STATECRUNCHER is its handling of nondeterminism. The basic principle that is applied is that, where alternative outcomes of processing an event are possible, each one is produced in a “world” of its own. In general, there will be several worlds in existence, and when an event is processed, it is processed in all of them. Identical worlds are merged (i.e. redundant worlds are eliminated). For worlds to be identical, their state occupancy and history and all data (variable values) and their traces must be identical. When testing, a comparison must be made between actual output and a match in any of the extant worlds. In the figure below, the sacks on the model side represent worlds.



**Figure 24. Testing with a nondeterministic oracle**

It is a major issue to discuss the ways in which the different worlds can come about. The subject is addressed in section 4, where we meet fork nondeterminism, race nondeterminism and other forms of nondeterminism.

### 3.4 STATECRUNCHER and the TorX tool chain

TorX is a tool chain delivered by the Côte de Resyste Project [CdR]. It separates out areas of concern in testing into distinct processes. Different test generation algorithms can be plugged in at the Primer level. STATECRUNCHER, which is an *Explorer* in TorX terminology, provides a command language to this end, described in detail in [StCrPrimer], but summarized in section 8. The test harness is incorporated into the Driver.

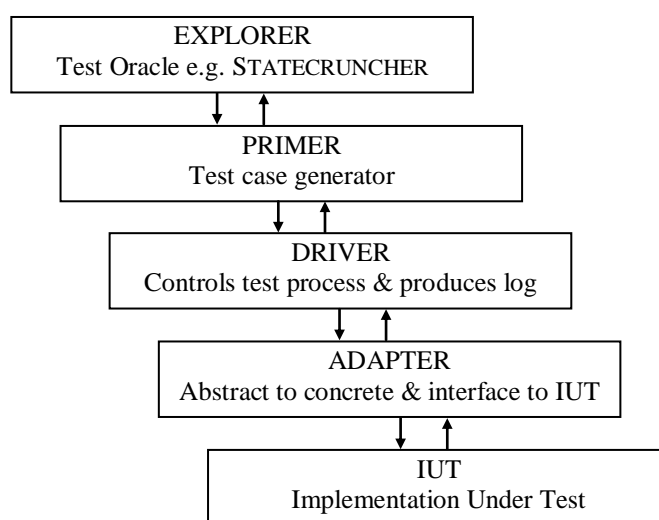


Figure 25. TorX tool chain

STATECRUNCHER has been experimentally integrated into this tool chain by Philips Research India - Bangalore. We show screenshots of this in chapter 10.

### 3.5 Alternative modelling techniques to state-based modelling

Experience has shown that a common category of system defects is a fault in their state behaviour. However, state behaviour is not always the dominant characteristic of a system, and it is worth mentioning alternative approaches and discussing when each approach is particularly relevant.

State-based modelling is appropriate where the *memory* aspect of a system is prominent: the system reacts one way or another way to the same event depending on something that has happened in the past.



### Decision tables and cause-effect graphs

Systems which simply show *feed-forward* logical behaviour are often better modelled by Decision Tables or Cause-Effect Graphing, described in [Myers, p.56]. The idea here is to model the relationships between logical (binary) inputs and outputs in terms of logical functions (and-gates, or-gates, not-gates) and constraint relationships between them and their derivatives (exclusive, requires, masks etc). The figure below shows how outputs Y and Z are related to inputs A B C F G H J K P Q R S and T. It also how the inputs are constrained amongst themselves in that one and only one of B and F can and must be true, and G requires H, i.e. for G to be true, H must be true.

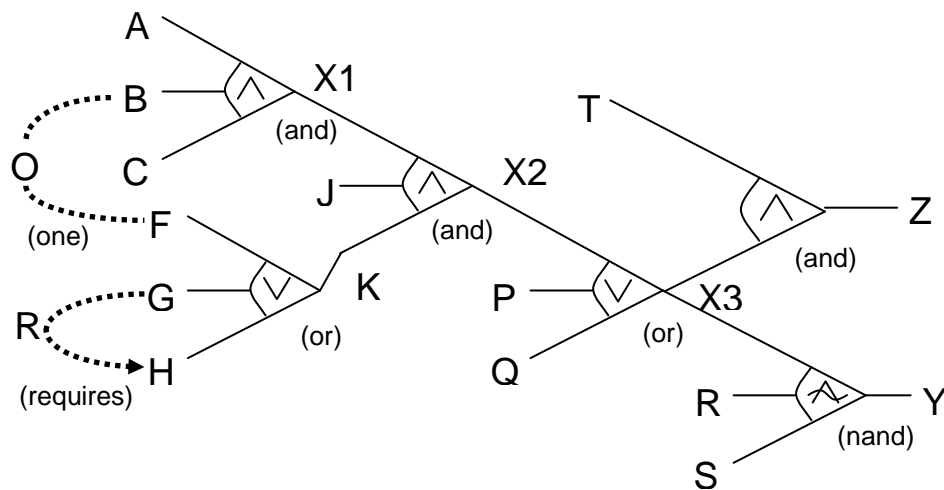


Figure 26. A cause-effect graph

State behaviour can be imitated to some extent using cause-effect graphs – some of the inputs could represent states, and others events, and the outputs might represent new states. But this is clearly not as elegant as a state machine model. Moreover, it has its limitations, since we cannot obtain a transition tour directly from this format.

### Syntax testing

Another modelling technique is to describe the *syntax*, not only of input data and input commands, but of the conventions and protocols of inter-process communication – perhaps even of inter-module communication. This is related to state modelling (mention has already been made of regarding input tokens to a compiler as events), but there is a difference in perspective. In addition to basic coverage of legal syntax, there will probably be a strong emphasis on checking the behaviour of the system when invalid input is processed. Reference: [Beizer, Ch. 9]

### Orthogonal arrays

A testing technique to test pairwise (or any subset-wise) every combination of parameters is to use *orthogonal arrays*. The technique is applicable to interacting subsystems as well as

parameters. For a popular article, see [Phadke]; for a library of orthogonal arrays, see [Sloane]. Suppose a routine needs testing with 4 parameters, (A, B, C, and D), each of which can take 3 values (1, 2, and 3). Exhaustive testing would require running  $3^4=81$  tests. Now suppose we find it adequate that all pairwise parameter value combinations are taken. A table can be found satisfying this with 9 entries of values of the 4 parameters as follows:

ABCD
1111
1223
1332
2122
2231
2313
3133
3212
3321

For *pairwise* coverage we speak of orthogonal arrays of *strength 2*. If we had required that all *triples* of parameters should be covered for all combinations of values, the strength would be 3 and so on. The above array is equivalent to the one published by Sloane at <http://www.research.att.com/~njas/oadir/oa.9.4.3.2.txt>. There is opportunity to combine orthogonal array techniques with state-based testing where there are parameterized events.

### 3.6 Summary of this section

This section discussed the concepts involved in state modelling and state-based testing, and introduced STATECRUNCHER, but reserved its handling of nondeterminism for the next section. We concluded with a quick look at alternatives to state-based testing: cause-effect graphs, syntax testing, and the use of orthogonal arrays.

For a more detailed discussion of testing in relation to the software development lifecycle, see the appendix [StCrContext].

## 4. Nondeterminism

This section gives an informal treatment of nondeterminism in state behaviour; for a precise definition, see section 7 (The transition algorithm). Although the concepts of forks, races and interleavings are well-known in the literature, we believe that our implementation of a UML-compatible language to handle these concepts in a concurrent, hierarchical statechart exhibits many novel features. Since nondeterminism is a major source of combinatorial explosion, we consider ways of containing state space issues in this section.

### 4.1 Review of nondeterministic testing

In the previous section, we saw that nondeterminism is represented by different *worlds*, and that in testing an implementation, we accept its behaviour provided that it is in accordance with one world generated by the model:

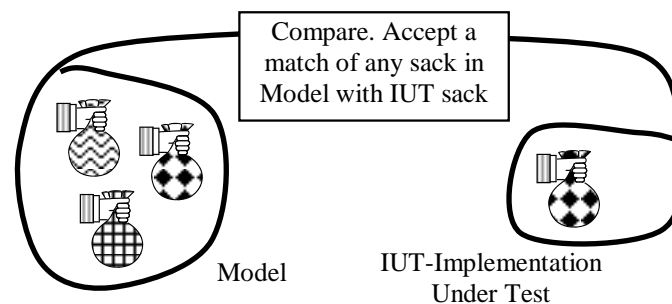
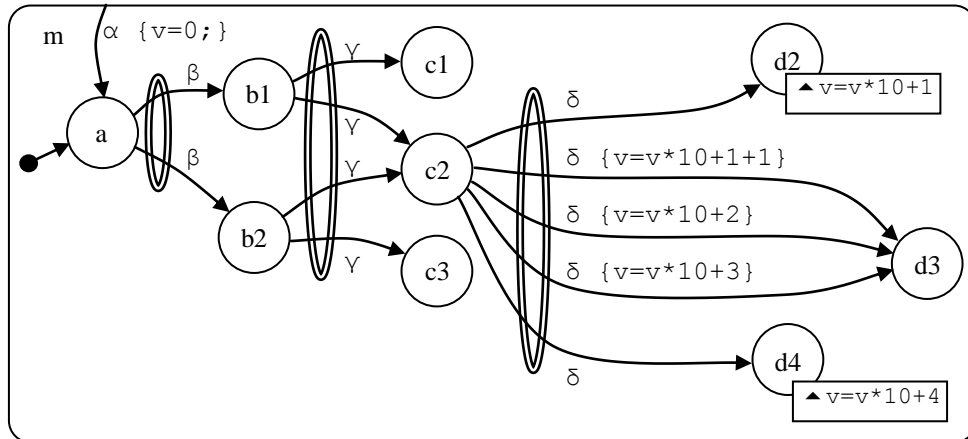


Figure 27. Review of testing with a nondeterministic oracle

We now consider various forms of structured nondeterminism as supported by STATECRUNCHER. *This is the main novel area of the present work.* The novelty with respect to existing systems is that we provide a broadly UML-compliant statechart language supporting structured nondeterminism, i.e. nondeterminism relating to the concurrent and hierarchical elements of statecharts. Existing experiments in nondeterministic testing, such as the Côte de Resyste project [CdR], use the languages LOTOS and PROMELA. Whilst these experiments have been very successful, are well-suited to the telecommunications industry, and have provided great inspiration, we feel that UML-aligned modelling is more accessible to most software practitioners. Within Philips, evaluations are currently (2003) taking place with STATECRUNCHER in the TorX tool chain as delivered by the Côte de Resyste project, and the results are encouraging (discussed in section 10).

## 4.2 Fork nondeterminism

*Fork nondeterminism* occurs where there are several transitions on the same event from the same source state. The figure below illustrates fork nondeterminism on events  $\beta$ ,  $\gamma$  and  $\delta$ .



**Figure 28. Fork nondeterminism [model u5420]**

The forks are emphasized by the double ellipses. The first fork is on event  $\beta$ , where the fork leads to two different target states. Then on event  $\gamma$  there is another fork, but with two transitions from different source states ( $b1$  and  $b2$ ) converging on the same target state. A duplicate world will be discarded, and there will be 3 resultant worlds. On event  $\delta$ , two worlds do not respond (those in states  $c1$  and  $c3$ ); these will be left intact. Departing from the world where  $c2$  is occupied, there are 5 transitions, but they only lead to 4 new worlds, because the transitions marked  $\delta \{v=v*10+1+1\}$  and  $\delta \{v=v*10+2\}$  lead to an identical world. They target the same state and set an identical value of the only variable  $v$ , whilst history and traces do not come into play. In all there are 6 worlds after event  $\delta$ . The model can effectively be reset by event  $\alpha$ , which will be processed in all worlds, but will take them to the same configuration, and duplicates will be removed, leaving one world.

After processing event  $\beta$ , the configuration as given by STATECRUNCHER is as follows.

```

3 statechart sc
3   cluster m [sc] = OCC [] **
3     leafstate a [m, sc] = VAC []
3     leafstate b1 [m, sc] = VAC []
3     leafstate b2 [m, sc] = OCC [] **
3     leafstate c1 [m, sc] = VAC []
3     leafstate c2 [m, sc] = VAC []
3     leafstate c3 [m, sc] = VAC []
3     leafstate d2 [m, sc] = VAC []
3     leafstate d3 [m, sc] = VAC []
3     leafstate d4 [m, sc] = VAC []
3 VAR INTEGER v [sc] =0
3 TRACE =[]
3 TREV [[gamma, [sc]], 0, [], []]

```

```

3   TREV [[alpha, [sc]], 0, [], []]
4
4   statechart sc
4       cluster m [sc] = OCC [] **
4           leafstate a [m, sc] = VAC []
4           leafstate b1 [m, sc] = OCC [] **
4           leafstate b2 [m, sc] = VAC []
4           leafstate c1 [m, sc] = VAC []
4           leafstate c2 [m, sc] = VAC []
4           leafstate c3 [m, sc] = VAC []
4           leafstate d2 [m, sc] = VAC []
4           leafstate d3 [m, sc] = VAC []
4           leafstate d4 [m, sc] = VAC []
4   VAR   INTEGER v [sc] =0
4   TRACE =[]
4   TREV [[gamma, [sc]], 0, [], []]
4   TREV [[alpha, [sc]], 0, [], []]

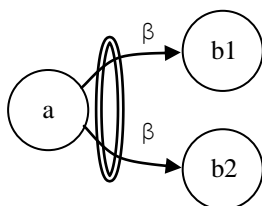
```

STATECRUNCHER has produced 2 worlds. Space does not permit us to reproduce the output on processing events  $\gamma$  and  $\delta$ .

In practice, fork nondeterminism is used to model cases in which there is uncertainty about what will happen, e.g. because of limited control over the IUT's environment.

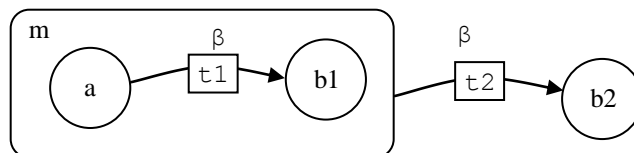
### *An issue in fork nondeterminism*

We have seen fork nondeterminism where the transitions have the identical source state:



**Figure 29. Fork nondeterminism with same source state**

But how is the following situation to be handled? The transitions are named  $t1$  and  $t2$ .



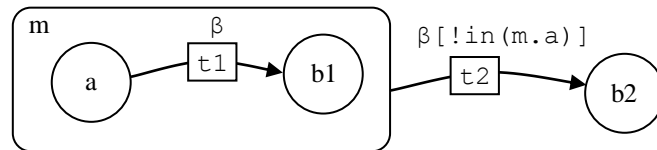
**Figure 30. Hierarchical issue**

There are three ways this could be handled:

- (1) We could say it is fork nondeterminism, with one world ending up in state  $m.b1$  and the other in state  $b2$ .

- (2) We could say that we prioritize and override by specialisation, saying that  $t_1$  takes precedence, because its *source* state is deeper in the hierarchy, and it masks  $t_2$ . In this case, the model is deterministic. This is the approach taken by UML, and is in line with overriding member methods in C++ derived classes.
- (3) We could say that we prioritize and override by the more external transition, saying that  $t_2$  takes precedence and masks  $t_1$ . In this case, the model is again deterministic. This approach has the advantage that an external transition cannot be affected by (perhaps poorly understood) internals of a deeply embedded machine. This is the approach taken by [CHSM].

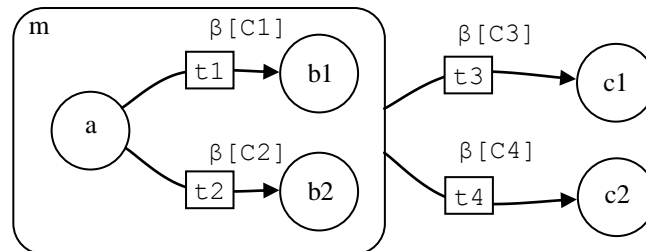
As pointed out by Lucas in [CHSM], under this scheme we can alter the precedence as follows:



**Figure 31. Forced prioritisation reversal giving specialisation**

STATECRUNCHER implements option (2) and conforms with UML, since that is the standard with which many designs comply.

A more general situation occurs when there are different levels of forks, and where the transitions are conditional:



**Figure 32. Forks in a hierarchy with conditional transitions**

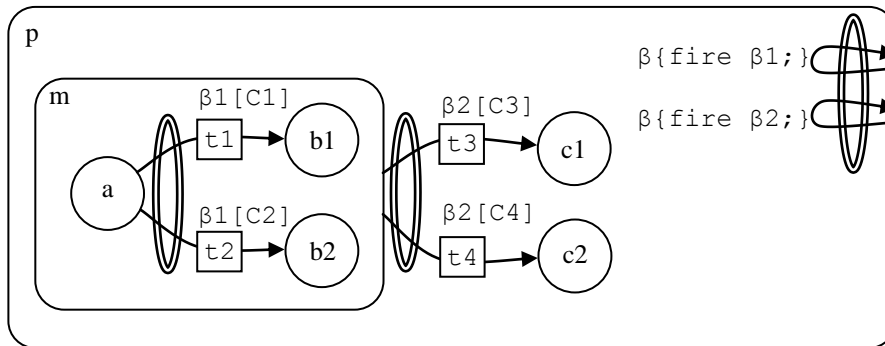
The *hierarchical prioritization* scheme means that transitions  $t_1$  and  $t_2$  form a fork, and  $t_3$  and  $t_4$  are masked by this and are not triggered by event  $\beta$ . If  $t_1$  has a false condition, then only  $t_2$  is taken and there is no nondeterminism. If  $t_1$  and  $t_2$  have false conditions, then  $t_3$  and  $t_4$  come into view and form a fork.

STATECRUNCHER proceeds as follows:

- Under an event, collect all possible transitions on it in the entire statechart hierarchy
- Evaluate all their conditions

- Find all innermost layers of the hierarchy that have at least one transition attached with a satisfied condition
- Take all satisfied transitions from these layers.

To obtain behaviour equivalent to *hierarchical impartiality* on event  $\beta$  in the above figure under the *hierarchical prioritization* scheme, a self-transition fork can be introduced as follows:



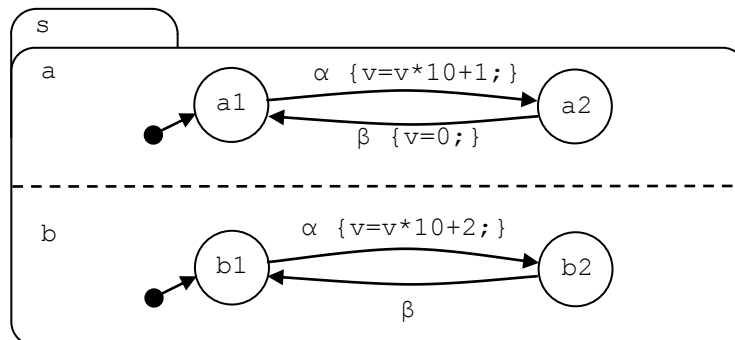
**Figure 33. Equivalent for hierarchical nondeterminism**

The original transitions on  $\beta$  are renamed  $\beta_1$  and  $\beta_2$ . Two internal self-transitions are introduced as a fork on  $\beta$ . One fires  $\beta_1$  and the other  $\beta_2$ . STATECRUNCHER will generate separate new worlds for each.

### 4.3 Race nondeterminism

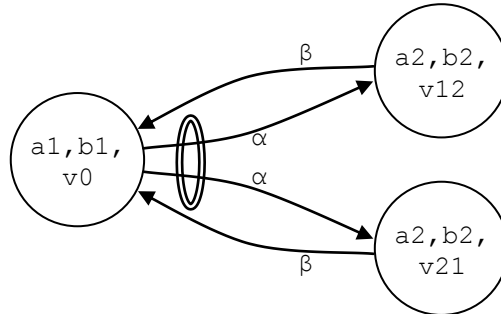
Race nondeterminism occurs where there are transitions on the same event in parallel components of the model (i.e. in different set members). The winner of the race may be distinguished by state occupancy or a variable value or a trace value or by cluster history.

In the figure below, there is a race between the transitions on  $\alpha$ . They are distinguished by the resultant value of variable  $v$ , which, given an initial value of 0, is 12 in one world and 21 in the other. The resultant state occupancy is identical in these worlds.



**Figure 34. Race - winner determined by variable value**

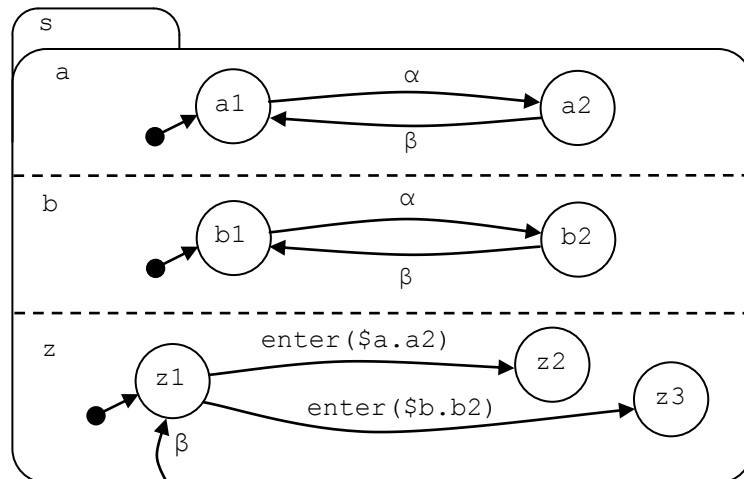
Race nondeterminism is a convenient way of expressing what would be fork nondeterminism in a *flattened* (or *unfolded*) model. The above model is equivalent to the following one:



**Figure 35. Flattened race model**

In the above model, the states are marked so as to indicate the corresponding states and variable value in the statechart of Figure 34. All structured nondeterminism is equivalent to fork nondeterminism in a flattened model. Although the flattened model in this case is very small, that is not normally the case, and a flattened representation is often not practicable.

The next example shows a similar race, but the winner is distinguished by the transition that takes place in set member z. Only one can take place, and as soon as it has taken place, the internal event on the other one will have no effect, since the source state of that transition, z1, is no longer occupied.



**Figure 36. Race - winner determined by state**

As with fork nondeterminism, the distinguishing aspect of the worlds generated, (so in a race, revealing the race winner), could also be *trace values* or *cluster history*.



## 4.4 Set transit nondeterminism

When a set is entered, all its members are entered. The order in which the members are entered may be significant, because of *upon enter* actions. STATECRUNCHER offers the facility to generate different orderings of entering the members. Similarly when a set is exited.

Consider the following model:

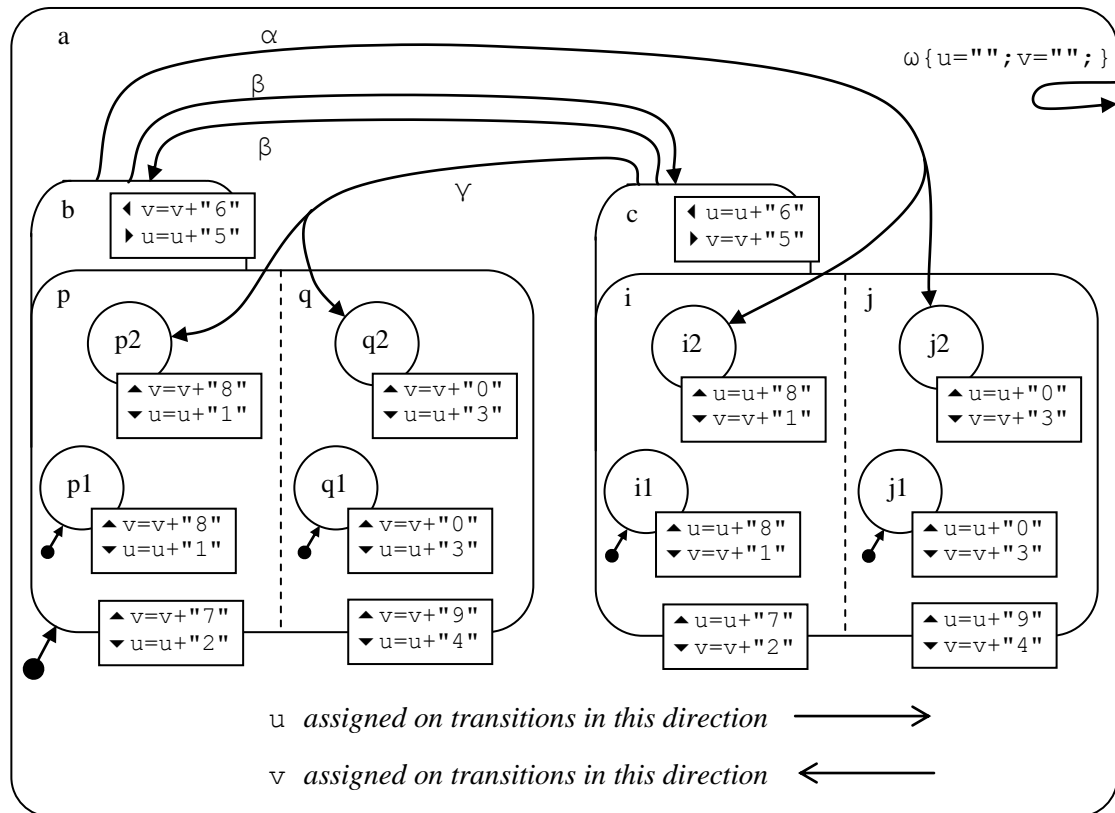


Figure 37. Set transit nondeterminism [model u5410]

We use strings rather than integers in the actions, because the integers could become very large. On processing event alpha, set b is exited in two orderings, then for each of those orderings, set c is entered in two different orderings. There are 4 different orderings of the set transit, and the values of u will register them:

```
exit: (p2,p), (q2,q),b;   enter: c, (i,i2), (j,j2); u=1234567890
exit: (p2,p), (q2,q),b;   enter: c, (j,j2), (i,i2); u=1234569078
exit: (q2,q), (p2,p),b;   enter: c, (i,i2), (j,j2); u=3412567890
exit: (q2,q), (p2,p),b;   enter: c, (j,j2), (i,i2); u=3412569078
```

These orderings are produced in different worlds. The output lines showing the value of u in each world are:

```
22  VAR  STRING  u [sc] = [49, ...] =1234569078
23  VAR  STRING  u [sc] = [51, ...] =3412569078
```

```

32  VAR  STRING  u [sc] =[49, ...] =1234567890
33  VAR  STRING  u [sc] =[51, ...] =3412567890

```

If we transition back to set *a* with event *gamma*, say, then variable *v* will track another 4 orderings. And these will be done in the 4 existing worlds. That will produce 16 worlds. The last lines of output are:

```

157 VAR  STRING  u [sc] =[49, ...] =1234569078
157 VAR  STRING  v [sc] =[51, ...] =3412567890
157 TRACE =[]
157 TREV [[omega, [sc]], 0, [], []]
157 TREV [[beta, [sc]], 0, [], []]
157 TREV [[alpha, [sc]], 0, [], []]

```

```

outworlds=[53, 54, 63, ... 156, 157]
number of outworlds=16

```

The order of transit in this last world was:

```

exit (j2,j), (i2,i), c; enter: b, (p,p2), (q,q2).

```

Note that when a set member is exited, we exit the leafstate then always immediately follow this by the set member, before moving on to the other member. So we never have an ordering such as `exit j2, exit i2, exit j, exit i`. This would be too fine an interleaving, and would exacerbate combinatorial explosion. We have bracketed tied orderings such as `(j2,j)` in the above descriptions.

If event *beta* is now given, then there will be 64 worlds. If then we process event *omega*, the variables are reset, and the number of worlds goes down from 64 to 1.

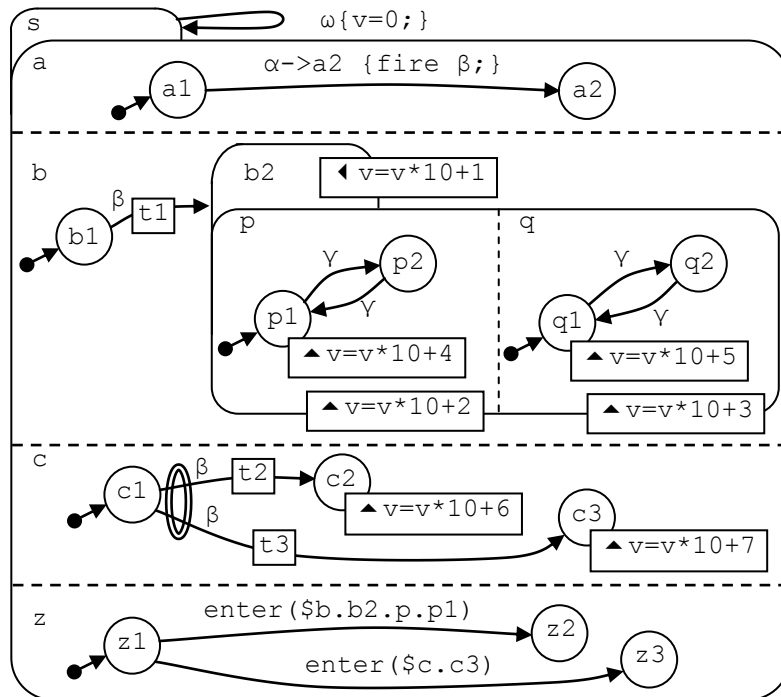
Although our model does not show it, set transit nondeterminism is applied at several levels in the hierarchy if there are several sets at different hierarchical levels. Test model `t6311` illustrates this, for which see [StCrTest].

## 4.5 Fired-event and multiple nondeterminism

Fired event nondeterminism is an indirect form of nondeterminism that occurs when an action associated with a transition causes another event to be fired, and that other event itself gives rise to some form of nondeterminism.

The following figure shows a model exhibiting fork, race, set-transit and fired-event nondeterminism in concert. The *action* of the transition on event  $\alpha$  is to fire event  $\beta$ . Event  $\beta$  triggers three transitions, which are explicitly named  $t_1$ ,  $t_2$  and  $t_3$ . These give rise to a fork and race. The set of sequences produced is:  $\{ \langle t_1, t_2 \rangle, \langle t_2, t_1 \rangle, \langle t_1, t_3 \rangle, \langle t_3, t_1 \rangle \}$ . Transition  $t_1$  gives rise to set-transit nondeterminism on entering set  $b_2$ . In one set of worlds states  $p$  and  $p_1$  will be entered before states  $q$  and  $q_1$ , and in another set of worlds this will be the other way around. The net result of processing event  $\alpha$  is therefore to generate 8

worlds. The order in which transitions and set-member entry is done is recorded in the variable  $v$ , since each assignment to this variable adds a unique digit to the end of the current value.



**Figure 38. Four kinds of nondeterminism in concert [model t5480]**

An example world generated on event  $\alpha$  is:

```

66 statechart sc
66   set s [sc] = OCC [] **
66   cluster a [s, sc] = OCC [] **
66     leafstate a1 [a, s, sc] = VAC []
66     leafstate a2 [a, s, sc] = OCC [] **
66   cluster b [s, sc] = OCC [] **
66     leafstate b1 [b, s, sc] = VAC []
66     set b2 [b, s, sc] = OCC [] **
66       cluster p [b2, b, s, sc] = OCC [] **
66         leafstate p1 [p, b2, b, s, sc] = OCC [] **
66         leafstate p2 [p, b2, b, s, sc] = VAC []
66       cluster q [b2, b, s, sc] = OCC [] **
66         leafstate q1 [q, b2, b, s, sc] = OCC [] **
66         leafstate q2 [q, b2, b, s, sc] = VAC []
66   cluster c [s, sc] = OCC [] **
66     leafstate c1 [c, s, sc] = VAC []
66     leafstate c2 [c, s, sc] = OCC [] **
66     leafstate c3 [c, s, sc] = VAC []
66   cluster z [s, sc] = OCC [] **
66     leafstate z1 [z, s, sc] = VAC []

```

```

66         leafstate z2 [z, s, sc] = OCC [] **
66         leafstate z3 [z, s, sc] = VAC []
66     VAR INTEGER v [sc] =612435
66     TRACE =[]
66     TREV [[gamma, [sc]], 0, [], []]
66     TREV [[omega, [sc]], 0, [], []]

```

The value of  $v$  (=612435) shows that transition  $t_2$  was chosen from the fork of  $t_2$  and  $t_3$ , and that it was executed before  $t_1$  in the race. This is corroborated by the occupancies of  $c_2$  and  $z_2$ . The value of  $v$  shows that order of entering set  $b_2$  and its members is:  $b_2$ ,  $p$ ,  $p_1$ ,  $q$ ,  $q_1$ . The other seven worlds have values of  $v$  of 613524, 135246, 124356, 712435, 713524, 135247, and 124357, with the corresponding state occupancies of  $c_2$ ,  $c_3$ ,  $z_2$ , and  $z_3$ .

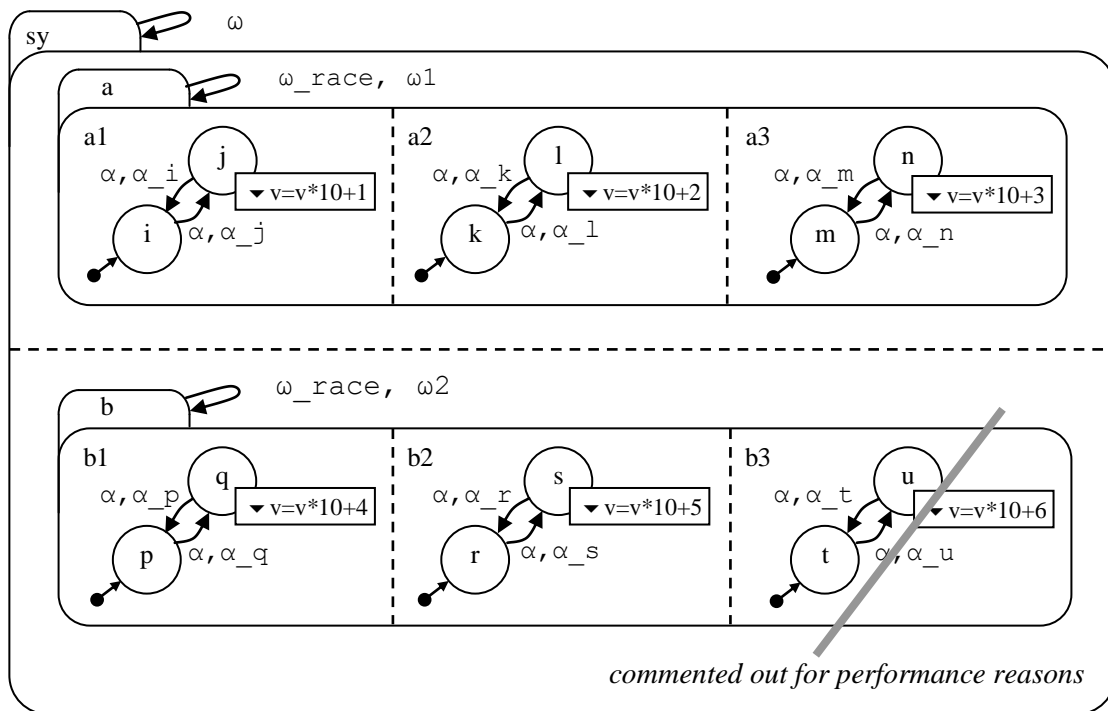
Permutations give rise to factorials, which are soon large numbers. In STATECRUNCHER, the following options for limiting the number of permutations are offered:

- the basic sequence without permutation (1 sequence)
- forwards and backwards only (2 sequences)
- all cyclic and anticyclic permutations (2n sequences)
- all permutations (n! sequences)

Separate control of race and set permuting is offered.

## 4.6 Set-action nondeterminism

Processing a single transition may lead to actions taking place in several set members, even though no set member may be entered or exited. This could be seen as a special case of set-transit nondeterminism, but we consider it separately. The actions will be hierarchically grouped (or bracketed) and permuted as for set-transit nondeterminism. The example below contains a set of sets, and suffers from the beginnings of poor performance due to the many permutations involved. For that reason, part of the model has been commented out.



**Figure 39. Set action nondeterminism [model t5412]**

When event  $\alpha$  is given, all the set members undergo a local transition. (There is actually a race between them, but there is no difference in outcome whatever the race order, and we ignore the race).

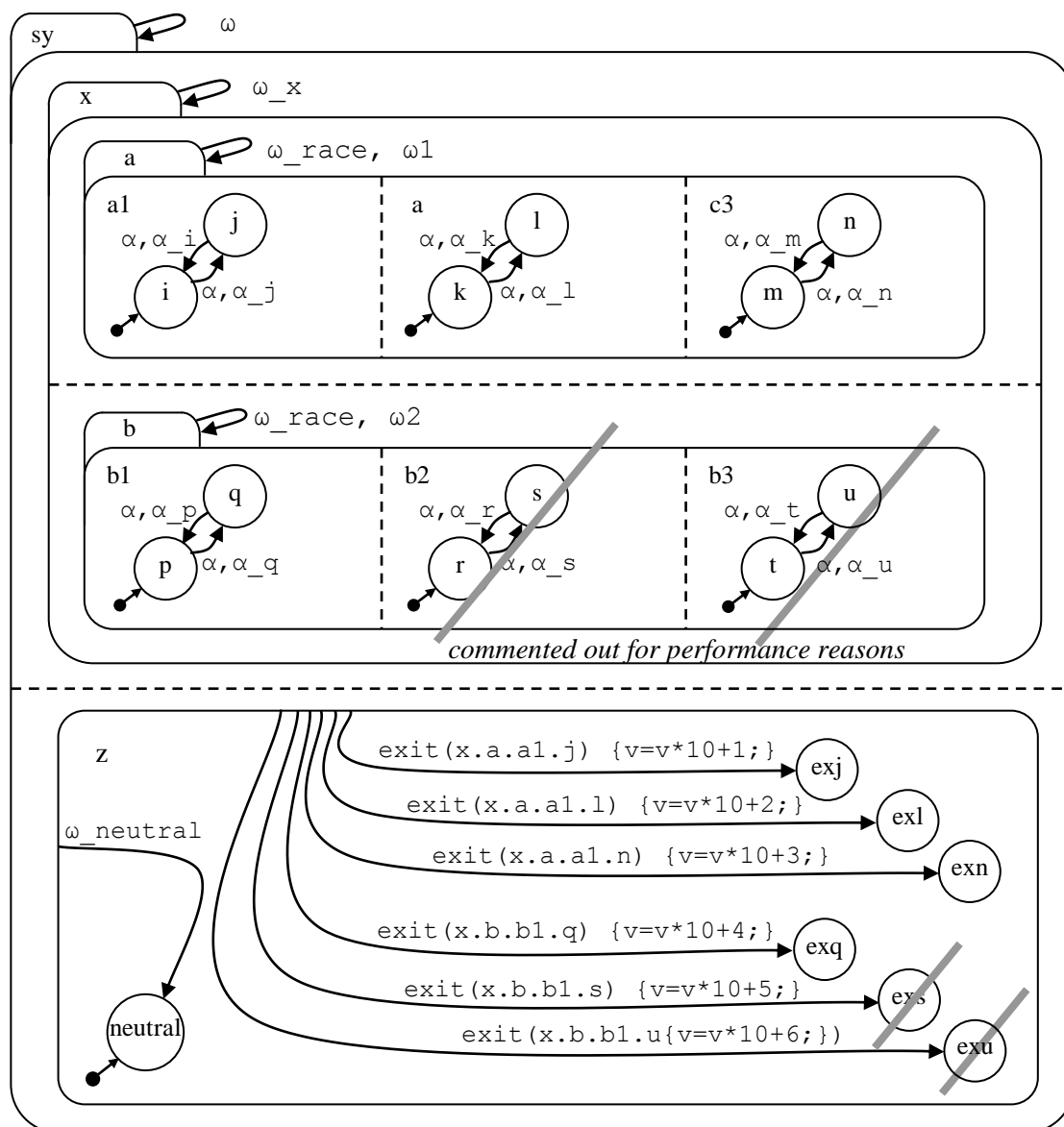
We could make all these set members transition back with another request to process event  $\alpha$ . As the set members transition back, they generate values of  $v$  that record the order in which it happened. Each order generates a different value of  $v$ . There are  $5! = 120$  orderings.

Now event  $\omega$  will do a similar thing in principle, although it is only attached to one transition. But there is one difference in what happens: orderings will be hierarchically generated as follows: the  $3! = 6$  orderings within set  $a$  will be generated, and the  $2! = 2$  orderings within set  $b$  will be generated. Then these 6 and 2 orderings will be regarded as single entities and ordered in  $2! = 2$  different ways. So the total number of orderings will be  $3!.2!.2! = 24$ . We call this set-action nondeterminism.

#### 4.7 Set-meta-event nondeterminism

This is similar to set-action nondeterminism. In our example below, we have a set containing a set containing two more sets, and we are not surprised to see poor performance, which is why part of the model has been commented out. Processing a single transition may lead to broadcast meta-events taking place relating to several set members, even though no set

member may be entered or exited. The meta-events will be hierarchically grouped (or bracketed) and permuted as for set-transit nondeterminism. Example:



**Figure 40. Set meta-event nondeterminism [model t5414]**

After event  $\alpha$ , any of events  $\omega$ ,  $\omega_x$ , or  $\omega_{\text{race}}$  will cause exiting of states, generating exit meta-events, triggering transitions in cluster  $z$ . Note that the transitions on the meta events respond from *any* state in cluster  $z$ , not just *neutral*. So *all* exit meta-events under consideration are recorded, in order. Events  $\omega$  and  $\omega_x$  cause hierarchically grouped orderings as with set action nondeterminism, producing in this case  $3! \cdot 1! \cdot 2! = 12$  orderings. Event  $\omega_{\text{race}}$  will generate 12 worlds by a different mechanism: the transitions on this event are sequenced in two orderings by race nondeterminism, and one of the transitions produces 6 orderings by set nondeterminism. As it happens,  $\omega_{\text{race}}$  is faster to process than

$\omega_x$ . If after event  $\alpha$ , we again process event  $\alpha$ , a similar reset to the initial states in sets a and b occurs, but now all transitions race each other, and  $4!=24$  worlds are produced (when all permutations are enabled, i.e. under *high race*), or under *medium race*, 8 worlds.

## 4.8 Effects of nondeterminism

We have seen six causes of nondeterminism (fork, race, set-transit, fired event, set-action and set-meta-event). We now discuss the effects of nondeterminism, i.e. the ways in which it may manifest itself.

### 4.8.1 State-occupancy nondeterminism

This is the most obvious form of nondeterminism, where different states are occupied after the different transitions, and is naturally associated with fork nondeterminism.

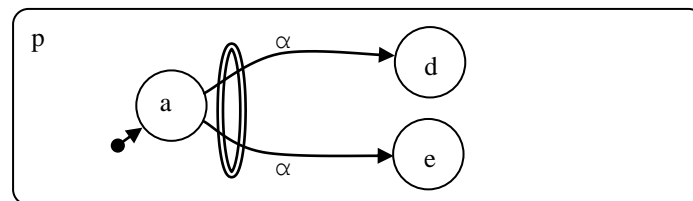


Figure 41. State occupancy nondeterminism

### 4.8.2 Variable-value nondeterminism

If two statecharts have the same state occupancy, but with different variable values, the result is that the worlds generated are distinct. The following example illustrates fork nondeterminism resulting in different variable values.

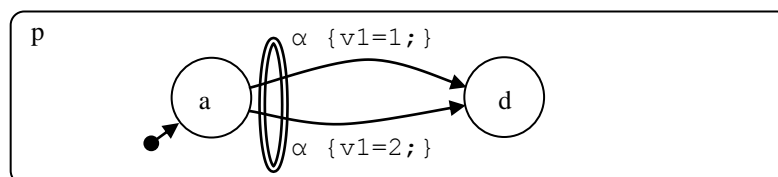


Figure 42. Variable-value nondeterminism

### 4.8.3 Trace-value nondeterminism

Traces are by definition *observable*. They are written in by the `trace()` function. The following example illustrates fork nondeterminism resulting in different trace values.

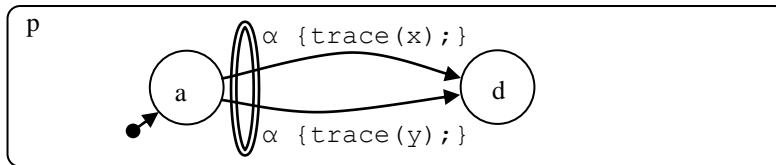


Figure 43. Trace value nondeterminism

#### 4.8.4 History nondeterminism

Just as variables can be the distinguishing factors in nondeterministic target states, so can history. In the following example, a transition from state  $q_b$  under event  $\alpha$  will lead to the same target state,  $c$ , as regards state occupancy, but history data distinguishes worlds and two worlds would be generated. A return transition on  $\beta$  will return to state  $q_b$  if history data is present, or to state  $q_a$  if history data has been cleared.

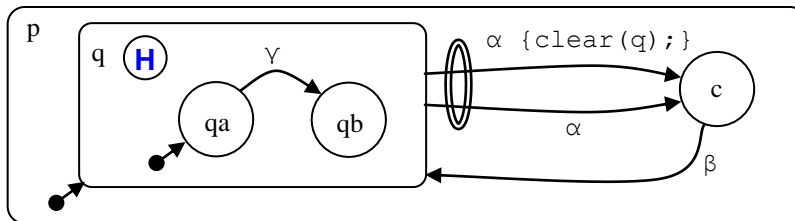


Figure 44. History nondeterminism

### 4.9 Worlds

As has been seen, under nondeterminism, STATECRUNCHER maintains several worlds. We will look at this in a little more detail. Consider the following model:

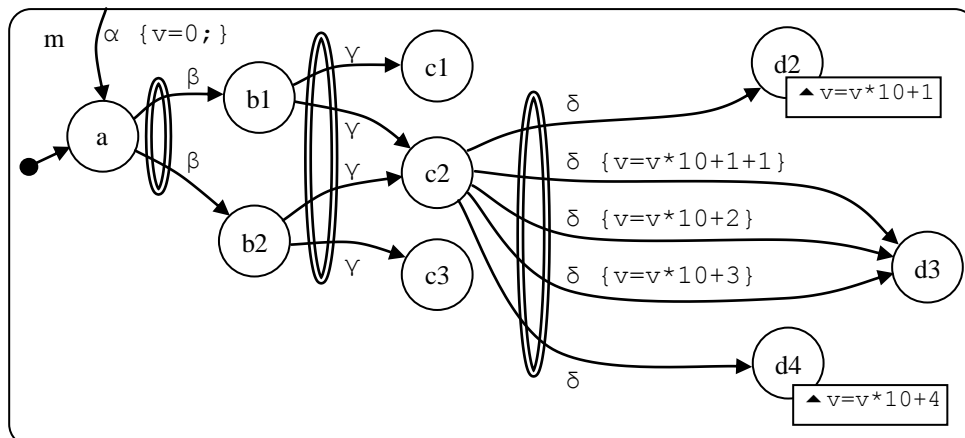


Figure 45. Fork nondeterminism [model u5420], for description of worlds

The forks are emphasized as usual by the double ellipses. The first fork is on event  $\beta$ , where the fork leads to two different target states. Then on event  $\gamma$  there is another fork, but with



two transitions from different source states ( $b_1$  and  $b_2$ ) converging on the same target state. A duplicate world will be discarded, and there will be 3 resultant worlds. On event  $\delta$ , two worlds do not respond (those in states  $c_1$  and  $c_3$ ); these will be left intact. Departing from the world where  $c_2$  is occupied, there are 5 transitions, but they only lead to 4 new worlds, because two transitions lead to an identical world. In all, there are 6 worlds after event delta. The model can effectively be reset by event  $\alpha$ , which will be processed in all worlds, but will take them to the same configuration, and duplicates will be removed, leaving one world.

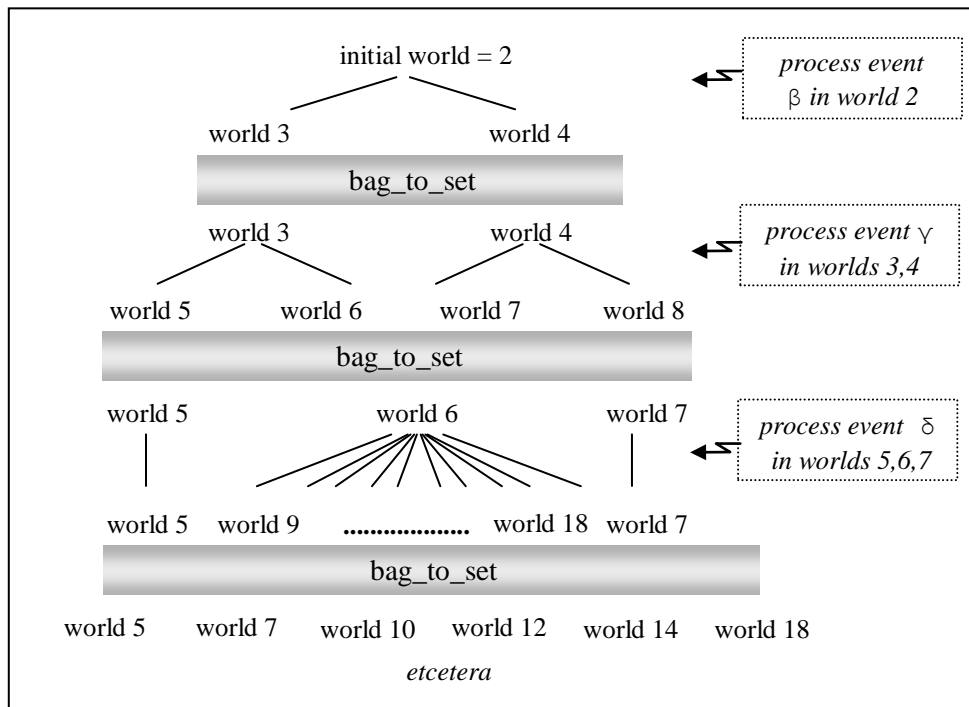
World numbers are arbitrary. Internally, the numbers are allocated sequentially as more and more events, transitions and actions are processed, but some world numbers may never be seen by the user as they are only used temporarily during processing (particularly when *actions* are involved). Worlds are not necessarily presented in numerical order, and the order is not significant.

After any events, internally, before the user sees them, the worlds produced are a *bag*. If any worlds in the bag are identical, the bag is reduced to a *set*, as here; then they are presented to the user. Merging is just a *bag\_to\_set* operation. As has been mentioned, for the worlds to be identical, their state occupancy and history and all data (variable values) and their traces must be identical.

World number 1 is special as it contains the initial data. This is kept as a *save area* to enable a reset to be done. The initial action when a model is run is to clone world 1 into world 2 and set that up as a starting point for further processing. On processing every event, new world-numbers are created for every derivative world. So we might have the following situation:

- Process event  $\beta$ : worldbag immediately after processing this event = [3,4]
- Worldbag after reducing to a set = [3,4]
- Process event  $\gamma$ : worlds are [5,6,7,8]
- Worldbag after reducing to a set = [5,6,7].
- Process event  $\delta$ : worlds are [6,7,9,10,11,12,13,14,15,16,17,18]
- Worldbag after reducing to a set = [6,7,10,12,14,18].

This may seem to be uneconomical use of world numbers – the first derivative world could sometimes use its ancestor's number – but this scheme is simpler algorithmically and facilitates debugging and tracing the progress of event processing. A log with a unique reference to each world can be made. The figure below shows how worlds are generated as events are processed.



**Figure 46. World generation and merging**

## 4.10 Containment of combinatorial explosion

Statechart systems are subject to combinatorial explosion of state spaces, and when nondeterminism is introduced, the problem is exacerbated. We address the combinatorial explosion problem in this section.

There are various levels at which some form of state explosion can occur:

- **Representation explosion.** The representation of the state space may require explicitly defining a large number of states. The use of statechart hierarchy and parallelism often mitigates this problem. If that is not the case, such a situation would suggest that the application being modelled is intrinsically complex or extensive.
- **Effective state space explosion.** Although there may be a compact representation of a model, using statechart facilities, there may still be a vast number of distinct effective states in the model. These would be explicit in a *flattened* (or *unfolded*) model.
- **Coverage explosion.** The testing technique may require visiting a large number of states, or executing a vast number of transitions, in order to achieve certain coverage requirements.
- **World explosion.** The number of nondeterministic worlds may become large.

The first of these, *representation explosion* appears to be an application-specific issue. We briefly consider *effective state space explosion* and *coverage explosion*. The fourth level,

*world explosion* is very pertinent to STATECRUNCHER and we will describe in some detail the ways in which the design of STATECRUNCHER addresses the issue.

#### 4.10.1 Effective state space explosion

The effective number of states may be very large, even though the statechart representation is compact. This is especially the case because the hierarchical structure allows for parallel state machines, where the number of states in a flattened state space is the product of the number of states in each parallel machine. This is only a problem if every state in the whole machine needs to be *visited*, and if this really is the case the approach is to do it as efficiently as possible. Techniques for compact storage of many states are *binary decision diagrams* (see e.g. [Bérard, pp.51-58]), used in [SPIN]), and hashing algorithms (to record whether a state has been visited). Minimizing the dynamic number of states generated is achieved by *on-the-fly* (or *adaptive*) testing, as opposed to *batch* (or *preset*) testing. With adaptive testing, shorter sequences of events can be used, because the feedback from the system under test to the test generator enables it to apply some intelligence and prune search spaces.

Variables and state history adversely affect the number of states in the flattened state space, since we must take the Cartesian product of states and variable values and state history values. The modeller should take care to do equivalence partitioning (maybe using enumerated values), rather than declaring an integer as, say,  $\{0, \dots, 10000\}$ , as it explodes the flattened state space.

#### 4.10.2 Coverage explosion

Some test coverage criteria are:

- Reach every state of the flattened machine
- Take every transition arc in a flattened machine

Even though a state space is large, it may be acceptable to traverse it in a limited way. A possibility is:

- Use Projected State Machine Coverage [Friedman, Farchi]. In this technique, states are grouped into equivalence classes. Each equivalence class is a single state in the projected machine.

Specific forms of projected state machine coverage would be to:

- Reach and vacate every leaf-state of the hierarchical statechart, i.e. to ensure that every leafstate is occupied in some test, and that every leafstate is vacant in some test, whilst remaining indifferent to which leafstates are occupied in combination with which other ones (and to variable/history values).
- Take every transition arc in the hierarchical statechart, but again showing indifference to the circumstances (occupancy of states in parallel parts of the machine etc.).
- To regard all leafstates in clusters as equivalent.

More research is needed to ascertain whether these forms of limited coverage are useful in practical testing. Useful ones will find (almost) as many faults as would have been found with more exhaustive testing.

### 4.10.3 World explosion

In STATECRUNCHER, it is not a model that specifies its nondeterminism, but the transition semantics that apply in principle to all STATECRUNCHER models. The number of dynamic states *per world* is similar to what it would be without nondeterminism.

World explosion arises from large number of worlds that can be generated, as follows:

- There will be  $f$  worlds for a fork with  $f$  prongs
- There will be  $r!$  worlds for a race between  $r$  transitions
- There will be  $s!$  worlds for a set operation involving  $s$  set members.

It is the factorials that are especially troublesome; we consider ways of containing them. In any case, race and set-transit (with its derivatives) nondeterminism are separately controllable in STATECRUNCHER and can be switched off.

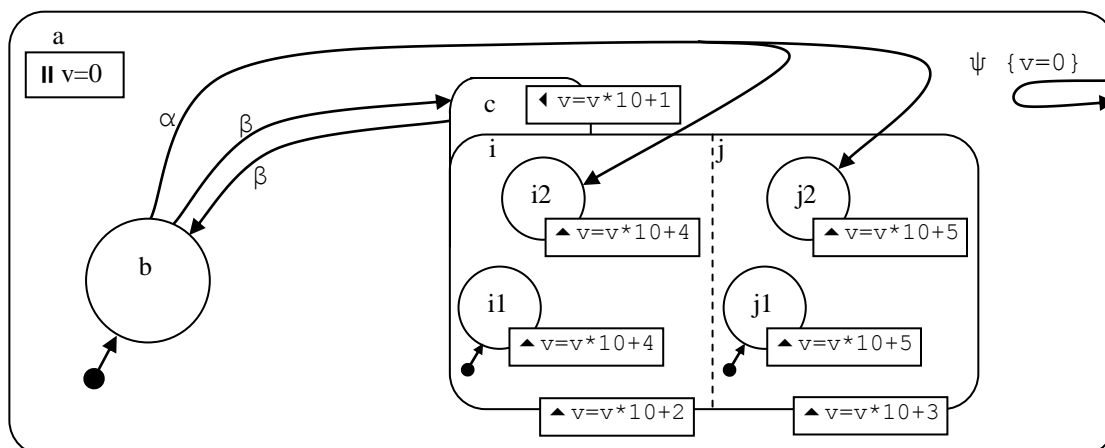
STATECRUNCHER offers the following containment features:

- A reasonable, not-too-fine granularity of interleavings in set nondeterminism, avoiding micro-orderings of state entry/exit.
- Separate control of how many permutations are generated under race and set nondeterminism.
- The ability to kill unwanted worlds, either as an explicit command, or in mid event processing, by specifying the expected trace (i.e. what the implementation has already given), so that worlds with a mismatching trace are pruned away quickly, nipping them in the bud.

We explain these more detail below.

#### 4.10.3.1 The granularity of set-transit nondeterminism

The number of worlds STATECRUNCHER generates on set-transit nondeterminism has been kept within reasonable bounds by avoiding excessive orderings of transition steps. This is illustrated using the figure below:



**Figure 47. Set-transit detail**

On transition  $\alpha$ , two interleavings of the on-entry actions are set up (assume  $v=0$ ):

- enter c, enter i, enter i2, enter j, enter j2. Variable  $v$  will be set to 12435.
- enter c, enter j, enter j2, enter i, enter i2. Variable  $v$  will be set to 13524.

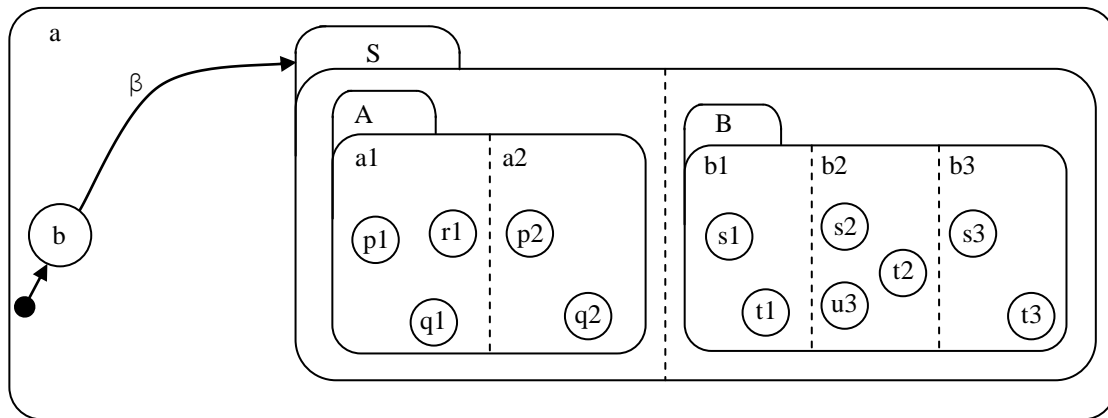
We do not generate the following interleavings (or any others):

- enter c, enter i, enter j, enter i2, enter j2. Variable  $v$  would be set to 12345.
- enter c, enter i, enter j, enter j2, enter i2. Variable  $v$  would be set to 12354.
- enter c, enter j, enter i, enter i2, enter j2. Variable  $v$  would be set to 13245.
- enter c, enter j, enter i, enter j2, enter i2. Variable  $v$  would be set to 13254.

The above interleavings show what is lost by the restrictions imposed. The interleavings retained are analogous to a depth-first set-entering algorithm, and the ones discarded are analogous to a breadth-first set-entering algorithm. Depth-first algorithms are much more natural in most situations and in most programming languages, leading to the notion of a call tree. This explains our choice.

In order to model a system which was capable of exhibiting the “breadth-first” orders of execution, it would probably be best to switch set-transit nondeterminism off (which can be done as an action in a model, or as an external command), and to manually supply separate transitions with actions representing each possible ordering the system could generate. These would then be processed as an explicit fork.

When a transition enters several sets, the permutations take place at each hierarchical level, illustrated from Figure 48 below.



**Figure 48. Hierarchical set-transit (in outline)**

The outer set members are A (with members a1 and a2) and B (with members b1, b2 and b3). The following permutations will be generated:

- within S:  $\langle A, B \rangle$  and  $\langle B, A \rangle$
- within A:  $\langle a1, a2 \rangle$  and  $\langle a2, a1 \rangle$
- within B:  $\langle b1, b2, b3 \rangle$ ,  $\langle b2, b3, b1 \rangle$ ,  $\langle b3, b1, b2 \rangle$ ,  $\langle b3, b2, b1 \rangle$ ,  $\langle b2, b1, b3 \rangle$ ,  $\langle b1, b3, b2 \rangle$

The net orderings on leafstate entry are therefore:

- $\langle a1, a2, b1, b2, b3 \rangle$ ,  $\langle a1, a2, b2, b3, b1 \rangle$ ,  $\langle a1, a2, b3, b1, b2 \rangle$ ,  $\langle a1, a2, b3, b2, b1 \rangle$ ,  
 $\langle a1, a2, b2, b1, b3 \rangle$ ,  $\langle a1, a2, b1, b3, b2 \rangle$
- $\langle a2, a1, b1, b2, b3 \rangle$ ,  $\langle a2, a1, b2, b3, b1 \rangle$ ,  $\langle a2, a1, b3, b1, b2 \rangle$ ,  $\langle a2, a1, b3, b2, b1 \rangle$ ,  
 $\langle a2, a1, b2, b1, b3 \rangle$ ,  $\langle a2, a1, b1, b3, b2 \rangle$
- $\langle b1, b2, b3, a1, a2 \rangle$ ,  $\langle b2, b3, b1, a1, a2 \rangle$ ,  $\langle b3, b1, b2, a1, a2 \rangle$ ,  $\langle b3, b2, b1, a1, a2 \rangle$ ,  
 $\langle b2, b1, b3, a1, a2 \rangle$ ,  $\langle b1, b3, b2, a1, a2 \rangle$
- $\langle b1, b2, b3, a2, a1 \rangle$ ,  $\langle b2, b3, b1, a2, a1 \rangle$ ,  $\langle b3, b1, b2, a2, a1 \rangle$ ,  $\langle b3, b2, b1, a2, a1 \rangle$ ,  
 $\langle b2, b1, b3, a2, a1 \rangle$ ,  $\langle b1, b3, b2, a2, a1 \rangle$

This hierarchical permutation technique generates fewer permutations (here,  $2! \cdot 2! \cdot 3! = 24$ ) than flat member permutation (here,  $5! = 120$ ), since a subset of flat member permutations is always taken. This too is a form of containment of world explosion (assuming, as always, that we have not excluded a mode of behaviour that the system under test might actually exhibit).

The orderings that are lost are ones such as  $\langle a1, b3, b1, a2, b2 \rangle$ . If they are required, they can be simulated (as in the Figure 47 situation) by switching set-transit nondeterminism off and manually supplying separate transitions with actions representing each possible ordering the system could generate.

#### 4.10.3.2 Limited permutation generation

Race condition nondeterminism and set-transit nondeterminism require, in principle, the generation of all permutations of a set of transitions. Different orderings of transitions can

lead to different resultant states or different values of variables. A sequence of assignments, (each of which could be attached to separate transitions) such as

$$v=v*10+1; \quad v=v*10+2; \quad v=v*10+3;$$

gives a different result for each order of execution of the three assignments.

The number of permutations of a sequence of length  $n$  is  $n!$ . For performance reasons, this restricts the applicability of exhaustive permutation generation to low values of  $n$ . If several cascaded permutations are involved, then the number of permutation sequences may be of the order of  $(n!)^2$  or  $(n!)^3$ . The world-merging algorithm is not particularly efficient, and experience shows that it is necessary to keep the number of worlds below about 100 in practice, although this number will increase a little over time with the increasing power of computers. The number of variables and states in the statechart is an additional factor in this processing. The following table shows some powers of factorials:

<b>n</b>	<b>n!</b>	<b>(n!)<sup>2</sup></b>	<b>(n!)<sup>3</sup></b>
4	24	576	13,824
5	120	14,400	1,728,000
7	5,040	25,401,600	1.2802 x 10 <sup>11</sup>
10	3,628,800	1.3168 x 10 <sup>13</sup>	4.7784 x 10 <sup>19</sup>

**Table 2. Factorial growth**

We would like to find a weaker alternative to generating all permutations of elements of the sets involved, but still retain some useful properties concerning the relative orderings of *some* of the elements of a set. In particular, a subset of all the permutations which covered *all* relative orderings of, say, *any* 3 elements of the set, would be useful.

**Example:** given a set of 4 elements {a,b,c,d}, there are 24 sequences representing all permutations. However, if we only require that *all* relative orderings of *any* 3 elements are represented in a subset of the permutations, then just the following 6 permutations will suffice:<sup>1</sup>

$$\langle a,b,c,d \rangle, \langle a,d,c,b \rangle, \langle b,d,c,a \rangle, \langle c,b,a,d \rangle, \langle c,d,a,b \rangle, \langle d,b,a,c \rangle$$

The reader can verify that whatever subset of 3 elements of {a,b,c,d} is taken, e.g. {a,c,d}, and whatever permutation of this subset is taken, e.g.  $\langle d,a,c \rangle$ , then the relative ordering of these 3 elements will be found in at least one of the above 6 permutations of the original set. For our example,  $\langle d,a,c \rangle$ , the last-mentioned permutation meets the requirement:  $\langle d,b,a,c \rangle$ .

### ***The [n,k] problem***

The **[n,k]** problem is to find a (small) subset  $G$  of the permutations of a set  $S$  of  $n$  elements, such that *all* permutations of *any*  $k$  elements of  $S$  are found with their relative ordering in *some* element of  $G$ .

---

<sup>1</sup> This set was found by Alistair Willis.

What we have shown above is a solution to the problem of selecting a subset  $G$  of the permutation of a set of **4** elements, such that *all* permutations of **3** elements of the set retain their relative ordering in *some* element of  $G$ . We call this a solution to the **[4,3]** problem.

We will now define some terminology, including the notion of *embedding*. Then we will address the  $[n,2]$  problem (which is very simple) and the  $[n,3]$  problem.

### Some terminology and context

#### *Sets*

All sets in the discussions that follow are assumed to be finite.

#### *Power set*

We denote the power set of a set  $S$  by  $P(S)$ .

#### *Sequences*

Sequences contain elements in a particular order. In this discussion, sequences are assumed to be finite and with distinct elements. We represent sequences using angle brackets to enclose the elements. The **head** of the sequence is the first element; the **tail** of the sequence is the sequence remaining after removing the head. Example:

$Q = \langle a, b, c, d \rangle$ . Its head is 'a' and its tail is  $\langle b, c, d \rangle$ .

#### *Precedence*

For a sequence

$$A = \langle a_i \rangle_{i=1}^k \quad (k \geq 2) = \langle a_1, a_2, a_3, \dots, a_k \rangle$$

$a_i$  precedes  $a_j$  (in  $A$ ) if  $i < j$ .

#### *Embedding*

In our example of a useful subset of permutations, we introduced the concept of the relative ordering of elements in one sequence being maintained in another sequence. This is the concept of one sequence *embedding* into another sequence.

To be more precise, for any sequences  $A$  and  $B$

$$A = \langle a_i \rangle_{i=1}^k = \langle a_1, a_2, a_3, \dots, a_k \rangle$$

$$B = \langle b_i \rangle_{i=1}^n = \langle b_1, b_2, b_3, \dots, b_n \rangle$$

Sequence  $\langle a_i \rangle$  embeds into  $\langle b_i \rangle$  if

there is a strictly increasing function  $f: [1..k] \rightarrow [1..n]$  such that

$$\forall r \in [1..k] \quad a_r = b_{f(r)}$$

Clearly  $|A| \leq |B|$ ; one sequence cannot embed into a smaller sequence.



We use the notation  $A \text{ F } B$  to denote that sequence A embeds *into* sequence B.

***Example of embedding***

$\langle \mathbf{b,c,e} \rangle \text{ F } \langle \mathbf{a,b,c,d,e,f} \rangle$

***The permutation function Perm***

For any set X, we define  $\text{Perm}(X)$  to be the *set of sequences* representing all permuted orderings of the elements of X.

***Example of the permutation function***

$\text{Perm}(\{a,b,c\}) = \{\langle a,b,c \rangle, \langle b,c,a \rangle, \langle c,a,b \rangle, \langle c,b,a \rangle, \langle b,a,c \rangle, \langle a,c,b \rangle\}$

***Useful subsets of a permutation set***

Given a set S of  $n$  elements, it is desirable to have a subset of  $\text{Perm}(S)$ , which we call G, i.e.  $G \circ \text{Perm}(S)$ , such that our embedding property holds for all sequences derived from all subsets of S of a certain size.

We first define the set  $S^k$ , which is the set of subsets of S of size k,

$$S^k = \{s : P(S) \mid |s| = k \times s\}$$

*i.e. the set of elements s of the power set of S such that the size of s is k.*

The embedding property that must hold is:

$$\forall s : S^k \times (\forall p : \text{Perm}(s) \times (\exists g : G \times p \text{ F } g))$$

*i.e. for all s in  $S^k$  it is the case that [ for all p in  $\text{Perm}(s)$  it is the case that there exists a g in G for which  $p \text{ F } g$  ].*

That is, every permutation-sequence of every size-k-subset of S embeds into at least one element of G.

For convenience, we denote the elements of  $S^k$ , which are *sets*, by a single subscript:  $S^k_i$

We denote the permutation *sequences* of  $S^k_i$ ,  $\text{Perm}(S^k_i)$ , using a second subscript:  $S^k_{i,j}$

The subscripts  $i$  and  $j$  simply enumerate the elements.

***Example of useful subsets of a permutation set***

$S = \{a, b, c, d\}$

$P = \text{Perm}(S) = \{$

$\langle \mathbf{a,b,c,d} \rangle, \langle a, b, d, c \rangle, \langle a, c, b, d \rangle, \langle a, c, d, b \rangle, \langle a, d, b, c \rangle, \langle \mathbf{a,d,c,b} \rangle,$   
 $\langle b, a, c, d \rangle, \langle b, a, d, c \rangle, \langle b, c, a, d \rangle, \langle b, c, d, a \rangle, \langle b, d, a, c \rangle, \langle \mathbf{b,d,c,a} \rangle,$   
 $\langle c, a, b, d \rangle, \langle c, a, d, b \rangle, \langle \mathbf{c,b,a,d} \rangle, \langle c, b, d, a \rangle, \langle \mathbf{c,d,a,b} \rangle, \langle c, d, b, a \rangle,$

$\langle d, a, b, c \rangle, \langle d, a, c, b \rangle, \langle \mathbf{d, b, a, c} \rangle, \langle d, b, c, a \rangle, \langle d, c, a, b \rangle, \langle d, c, b, a \rangle\}$

Subset G consists of the bold sequences above:  $G = \{G_1, G_2, G_3, G_4, G_5, G_6\}$ , where

$G_1 = \langle \mathbf{a, b, c, d} \rangle, G_2 = \langle \mathbf{a, d, c, b} \rangle, G_3 = \langle \mathbf{b, d, c, a} \rangle,$   
 $G_4 = \langle \mathbf{c, b, a, d} \rangle, G_5 = \langle \mathbf{c, d, a, b} \rangle, G_6 = \langle \mathbf{d, b, a, c} \rangle$

The set  $S^3$ , (i.e. the set of subsets of S of size 3), is

$S^3 = \{\{a, b, c\}, \{a, b, d\}, \{a, c, d\}, \{b, c, d\}\}$

The individual elements of  $S^3$  are:

$S^3_1 = \{a, b, c\} \quad S^3_2 = \{a, b, d\} \quad S^3_3 = \{a, c, d\} \quad S^3_4 = \{b, c, d\}$

The permutations of these subsets together with the element of G into which they embed are:

$S^3_{1,1} = \langle a, b, c \rangle FG_1 \quad S^3_{1,2} = \langle a, c, b \rangle FG_2$   
 $S^3_{1,3} = \langle b, a, c \rangle FG_6 \quad S^3_{1,4} = \langle b, c, a \rangle FG_3$   
 $S^3_{1,5} = \langle c, a, b \rangle FG_5 \quad S^3_{1,6} = \langle c, b, a \rangle FG_4$

$S^3_{2,1} = \langle a, b, d \rangle FG_1 \quad S^3_{2,2} = \langle a, d, b \rangle FG_2$   
 $S^3_{2,3} = \langle b, a, d \rangle FG_4 \quad S^3_{2,4} = \langle b, d, a \rangle FG_3$   
 $S^3_{2,5} = \langle d, a, b \rangle FG_5 \quad S^3_{2,6} = \langle d, b, a \rangle FG_6$

$S^3_{3,1} = \langle a, c, d \rangle FG_1 \quad S^3_{3,2} = \langle a, d, c \rangle FG_2$   
 $S^3_{3,3} = \langle c, a, d \rangle FG_4 \quad S^3_{3,4} = \langle c, d, a \rangle FG_5$   
 $S^3_{3,5} = \langle d, a, c \rangle FG_6 \quad S^3_{3,6} = \langle d, c, a \rangle FG_3$

$S^3_{4,1} = \langle b, c, d \rangle FG_1 \quad S^3_{4,2} = \langle b, d, c \rangle FG_3$   
 $S^3_{4,3} = \langle c, b, d \rangle FG_4 \quad S^3_{4,4} = \langle c, d, b \rangle FG_5$   
 $S^3_{4,5} = \langle d, b, c \rangle FG_6 \quad S^3_{4,6} = \langle d, c, b \rangle FG_2$

The above set G is optimal, i.e. there is no smaller set with the desired property. This can easily be seen because G contains the same number of elements as **Perm** of an  $S^3_i$  set. G can never have fewer elements, as no two permutations of elements the same  $S^3_i$  for any  $i$  can embed into the same element of G.

### Optimal solution to the [n,2] problem

Given a set S

$S = \{s_i\}_{i=1}^n = \{s_1, s_2, s_3, \dots, s_n\}$

We select G to be the set of two sequences of the elements: in one order and in its reverse:

$G = \{G_1, G_2\} = \{\langle s_i \rangle_{i=1}^n, \langle s_j \rangle_{j=n}^1\} = \{\langle s_1, s_2, s_3, \dots, s_n \rangle, \langle s_n, s_{n-1}, s_{n-2}, \dots, s_1 \rangle\}$

**Theorem**

For any set S, |S| ≥ 2, and set G as defined above:

$$\forall s : S^2 \times [ \forall p : \text{Perm}(s) \times ( \exists g : G \times pFg ) ]$$

This is the previously mentioned embedding property that must hold, for k=2.

**Proof**

All sequences  $S^2_{u,v}$  derived from S contain two distinct elements of S. For an arbitrary u and v,

$$S^2_{u,v} = \langle s_p, s_q \rangle$$

Case 1: p < q. It is seen from the definition of  $G_1$  that  $\langle s_p, s_q \rangle \in FG_1$

Case 2: p > q. It is seen from the definition of  $G_2$  that  $\langle s_p, s_q \rangle \in FG_2$

□

Example:

$$S = \{a, b, c, d, e\}$$

$$G_1 = \langle a, b, c, d, e \rangle, \quad G_2 = \langle e, d, c, b, a \rangle$$

$$S^2_{u,v} = \langle d, b \rangle, \text{ which embeds into } G_2.$$

**Sub-optimal solution to the [n,3] problem (n ≥ 3)**

Although this solution is not optimal, it is linear with n, (|G|=2n), so it can be considered to be fairly good.

Given a set S

$$S = \{s_i\}_{i=1}^n = \{s_1, s_2, s_3, \dots, s_n\}$$

and the permutation set  $P = \text{Perm}(S)$ :

We select G to be the set of the following 2n sequences of the elements:

$$G_i = \begin{cases} \langle s_i \rangle_{i=1}^n & \text{when } i=1 & \text{(cyclic)} \\ \langle s_j \rangle_{j=i}^n, j=1^{i-1} & \text{when } i > 1, i \leq n & \text{(cyclic)} \\ \langle s_i \rangle_{i=n}^1 & \text{when } i=n+1 & \text{(anticyclic)} \\ \langle s_j \rangle_{j=2n+1-i}^1, j=n^{2n+2-i} & \text{when } i > n+1, i \leq 2n & \text{(anticyclic)} \end{cases}$$

**Examples of G**

$$S = \{s_1, s_2, s_3, s_4, s_5\}$$

$$G_1 = \langle s_1, s_2, s_3, s_4, s_5 \rangle \quad G_2 = \langle s_2, s_3, s_4, s_5, s_1 \rangle$$

$$G_3 = \langle s_3, s_4, s_5, s_1, s_2 \rangle \quad G_4 = \langle s_4, s_5, s_1, s_2, s_3 \rangle$$

$$G_5 = \langle s_5, s_1, s_2, s_4, s_3 \rangle$$

$$G_6 = \langle s_5, s_4, s_3, s_2, s_1 \rangle \quad G_7 = \langle s_4, s_3, s_2, s_1, s_5 \rangle$$

$$G_8 = \langle s_3, s_2, s_1, s_5, s_4 \rangle \quad G_9 = \langle s_2, s_1, s_5, s_4, s_3 \rangle$$

$$G_{10} = \langle s_1, s_5, s_4, s_3, s_2 \rangle$$

$S=\{a,b,c,d,e\}$

$G_1=\langle a,b,c,d,e \rangle, G_2=\langle b,c,d,e,a \rangle, G_3=\langle c,d,e,a,b \rangle, G_4=\langle d,e,a,b,c \rangle, G_5=\langle e,a,b,c,d \rangle$

$G_6=\langle e,d,c,b,a \rangle, G_7=\langle d,c,b,a,e \rangle, G_8=\langle c,b,a,e,d \rangle, G_9=\langle b,a,e,d,c \rangle, G_{10}=\langle a,e,d,c,b \rangle$

We call  $G_1\dots G_n$  the *cyclic* elements of  $G$ , and  $G_{n+1}\dots G_{2n}$  the *anticyclic* elements.

**Theorem**

For any set  $S$ ,  $|S| \geq 3$ , and  $G$  as defined generically above:

$$\forall s : S^3 \times [ \forall p : \text{Perm}(s) \times ( \exists g : G \times pFg ) ]$$

This is the previously mentioned embedding property that must hold, for  $k=3$ .

**Proof**

All sequences  $S^3_{u,v}$  derived from  $S$  contain three distinct elements of  $S$ . For an arbitrary  $u$  and  $v$ ,

$$S^3_{u,v} = \langle s_p, s_q, s_r \rangle$$

Without loss of generality, we can see an element of  $G$  into which this will embed by writing the elements of  $G$  in a form emphasizing the position of  $s_p$ . We will use a form of arithmetic *modulo n with an offset of 1* such that

if  $p=n$ , then  $p+1 \equiv 1$

if  $p=1$ , then  $p-1 \equiv n$

It is not possible for *both* the above modulo adjustments to need to be made for the same  $p$  (since  $p=1, p=n, n \geq 3$  is false).

All **cyclic** elements of  $G$  are of the form

$$\langle s_p, s_{p+1}, \dots, s_{p-1} \rangle \quad (\text{prior to explicit modulo adjustment})$$

Three cases come into view after making modulo adjustments:

$$G_{c-1}: \langle s_p, s_{p+1}, \dots, s_n, s_1, \dots, s_{p-1} \rangle \quad (p \neq n, p \neq 1)$$

$$G_{c-2}: \langle s_n, s_1, \dots, s_{n-1} \rangle \quad (p=n)$$

$$G_{c-3}: \langle s_1, s_2, \dots, s_n \rangle \quad (p=1)$$

All **anticyclic** elements of  $G$  are of the form

$$\langle s_p, s_{p-1}, \dots, s_{p+1} \rangle \quad (\text{prior to explicit modulo adjustment})$$

Again, three cases come into view after making modulo adjustments:

$$G_{a-1}: \langle s_p, s_{p-1}, \dots, s_1, s_n, \dots, s_{p+1} \rangle \quad (p \neq n, p \neq 1)$$

$$G_{a-2}: \langle s_n, s_{n-1}, \dots, s_1 \rangle \quad (p=n)$$

$$G_{a-3}: \langle s_1, s_n, \dots, s_2 \rangle \quad (p=1)$$

There are 6 main cases of  $\langle s_p, s_q, s_r \rangle$  to consider:

Case 1:  $p < q, q < r, r > p$ .  $\langle s_p, s_q, s_r \rangle$  embeds into the cyclic case  $G_{c-3}$

Case 2:  $p < q, q > r, r < p$ .  $\langle s_p, s_q, s_r \rangle$  embeds into the cyclic case  $G_{c-1}$

Case 3:  $p > q, q < r, r < p$ .  $\langle s_p, s_q, s_r \rangle$  embeds into the cyclic case  $G_{c-1}$  ( $p \neq n$ ) or  $G_{c-2}$  ( $p=n$ )

Case 4:  $p > q, q > r, r < p$ .  $\langle s_p, s_q, s_r \rangle$  embeds into the anticyclic case  $G_{a-2}$

- Case 5:  $p > q, q < r, r > p$ .  $\langle s_p, s_q, s_r \rangle$  embeds into the anticyclic case  $G_{a-1}$   
Case 6:  $p < q, q > r, r > p$ .  $\langle s_p, s_q, s_r \rangle$  embeds into the anticyclic case  $G_{a-1}$  ( $p \neq 1$ ) or  $G_{a-3}$  ( $p = 1$ )  
□

**Examples** of values of  $p, q, r$  for these cases representing typical permutations of three elements of a set of, say, 40 elements:  $\{s_1 \dots s_{40}\}$ :

- Case 1:  $p < q, q < r, r > p$ .  $p=10, q=20, r=30$   
Case 2:  $p < q, q > r, r < p$ .  $p=20, q=30, r=10$   
Case 3:  $p > q, q < r, r < p$ .  $p=30, q=10, r=20$   
Case 4:  $p > q, q > r, r < p$ .  $p=30, q=20, r=10$   
Case 5:  $p > q, q < r, r > p$ .  $p=20, q=10, r=30$   
Case 6:  $p < q, q > r, r > p$ .  $p=10, q=30, r=20$

We can also see the above case selection as working as follows. There are two sequences which start with any  $s_p$  –a cyclic one and an anticyclic one. A sequence  $\langle s_p, s_q, s_r \rangle$  is a candidate to embed into one of these. The tails of the two such sequences contain the remaining  $s_q, s_r$  elements in *opposite orders*. So one or the other will always satisfy the relative precedence requirement of  $s_q$  and  $s_r$ .

### **Application in STATECRUNCHER**

STATECRUNCHER gives separate control over *race* and *set* nondeterminism, both from within a model and as an external command.

For control of *race nondeterminism*:

<i>function</i>	<i>external command</i>	<i>effect</i>
no_race	nr	Only one ordering taken (forwards)
low_race	lr	Two orderings taken (forwards/reverse)
med_race	mr	$2n$ orderings taken (all cyclic, all anticyclic)
high_race	hr	All $n!$ orderings of the permutation taken

**Table 3. Control of race nondeterminism**

For control of *set nondeterminism*:

<i>function</i>	<i>external command</i>	<i>effect</i>
no_set_tran	nst	Only one ordering taken (forwards)
low_set_tran	lst	Two orderings taken (forwards/reverse)
med_set_tran	mst	$2n$ orderings taken (all cyclic, all anticyclic)
high_set_tran	hst	All $n!$ orderings of the permutation taken

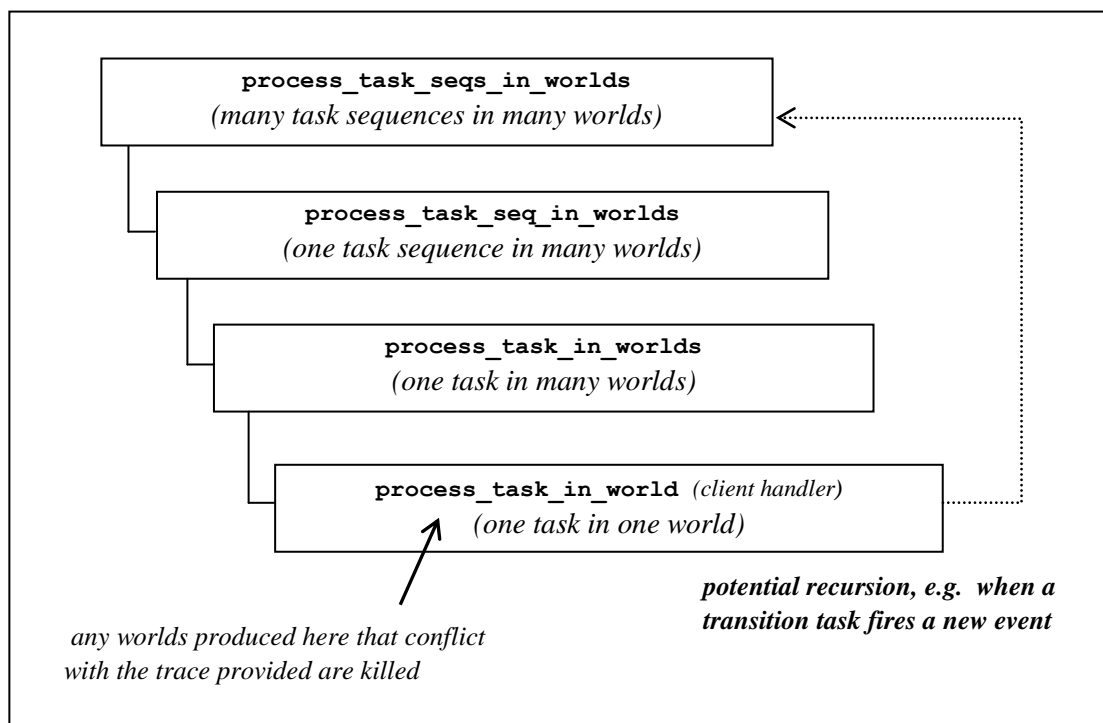
**Table 4. Control of set nondeterminism**

Set nondeterminism consists of set-transit nondeterminism, set-action nondeterminism and set-meta-event nondeterminism; they are all controlled by the same setting (the suffix `_tran` is a little misleading in this respect).

#### 4.10.3.3 Pruning worlds based on traces

The commands to STATECRUNCHER include one to kill worlds, and this enables world pruning to be done by the Primer/Driver, when it is seen that some worlds do not match the IUT (Implementation Under Test) behaviour. The idea of optimising this process within STATECRUNCHER processing was first put forward by Tim Trew. The idea is for the IUT to produce its traces first, and for STATECRUNCHER to be given these and be asked to *verify* them, pruning worlds whenever it can *en route*.

Non-matching worlds will be killed by STATECRUNCHER after processing any event, but also in the routine that processes a transition in one world. This routine is called after a series of reductions from the routine to process a set of transition sequences in many worlds, as explained in section 7.6. It is a good point in mid-algorithm, just after new worlds have been produced, to prune them, “nipping them in the bud”.



**Figure 49. Reduction of task processing**

*Implementation in STATECRUNCHER Release 1.05*

Abbrev.	Command
Command	<i>showing typical example and/or typical output</i>

<b>pe ...</b>	<b>process event EVENT ?p=PARAMETERS ?t=EXPECTEDTRACE</b>
	pe gamma p=[4,xy] (statechart scope assumed)
	pe [alpha, [sc]] p=1 t=[2,4]
	pe [alpha, [sc]]
	<i>Parameters can also be supplied in STATECRUNCHER internal form, e.g.</i>
	p=[[ex_co, int, 4], [ex_str, [120, 121]]]

**Table 5. STATECRUNCHER command to prune worlds given a trace**

The idea is to process an event giving STATECRUNCHER a trace to expect. This would typically be what a SUT has already revealed. Supplying the expected trace to STATECRUNCHER serves two purposes:

- It may save the primer having to kill worlds
- It enables optimisations in STATECRUNCHER, because mismatching worlds can be nipped in the bud.

Some traces are plain mismatches. But what should be done when STATECRUNCHER produces too little trace (undertrace), or too much trace (overtrace) while not being in flagrant violation of the expected trace? Examples (trace lists are read from right to left):

- undertrace: Expected-trace= [cd, ab], STATECRUNCHER-trace= [ab]
- overtrace: Expected-trace= [cd, ab], STATECRUNCHER-trace= [ef, cd, ab]

Which of these should be permitted?

Clearly, in mid-algorithm we must allow undertrace, as the rest of the algorithm may produce the required remaining trace.

For the total algorithm, the requirement is not clear, and it depends on modelling philosophy.

An argument for allowing *overtrace* is that the SUT may “spontaneously” produce the missing trace (e.g. by unsolicited notifications which have not been modelled as being initiated by an event). But is this a good approach to modelling?

There is no clear argument for allowing *undertrace*. However, there may be ways of modelling in which it is required.

STATECRUNCHER currently applies a very lenient strategy of allowing everything except a flagrant trace violation. This can be changed if required.

Test models `t5550`, `t5555`, `t5560`, `t5565` (q.v. in [StCrTest]) can be used for experimentation.

#### *Application in test strategies*

When black-box testing, worlds produced by STATECRUNCHER will be killed if their traces do not match the IUT's traces. This can either be done using the above mentioned pruning technique, or by explicit `kill` commands.

If after a test STATECRUNCHER has been left with no worlds, the test has given a failure. The problem arises how to continue. It may be acceptable to fix the problem manually before continuing with testing; if not, automatic recovery will involve either recreating a previous set of worlds (which can be done by feeding world output back to STATECRUNCHER), from which subsequent tests can continue, or by a reset to the initial world from which an independent part of the test suite can run.

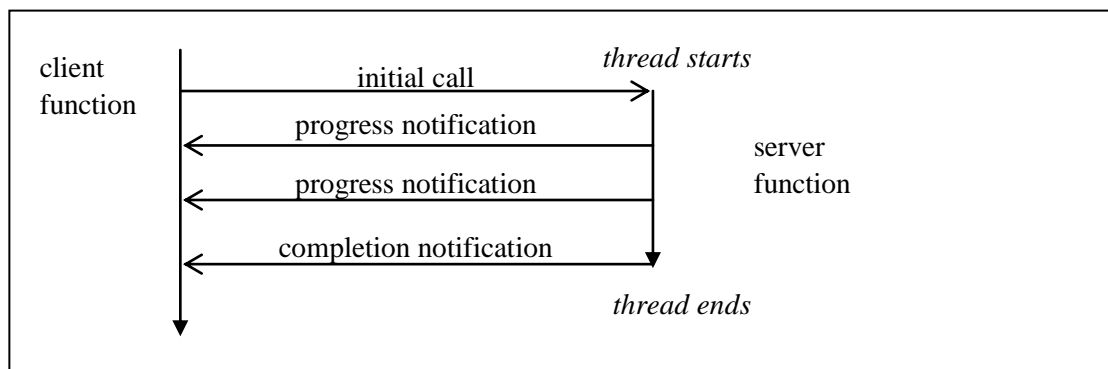
If after a test STATECRUNCHER has been left with one world, then the tests are running as efficiently as possible.

#### **4.10.4 The notification example - and containment approaches**

A practical example shows that more is needed than the devices we have discussed so far.

The *notification* problem as discussed here was identified by Tim Trew [Trew 03], who also proposed the basic technique of pruning worlds based on a supplied trace.

A notification is a message between asynchronous processes, e.g. after one function (a client function) has called another (a server function) on a different thread. After the call, both functions can proceed on their own thread. The server function can communicate with the client function by sending messages to indicate progress, and ultimately, completion. (The client may also communicate with the server, of course).

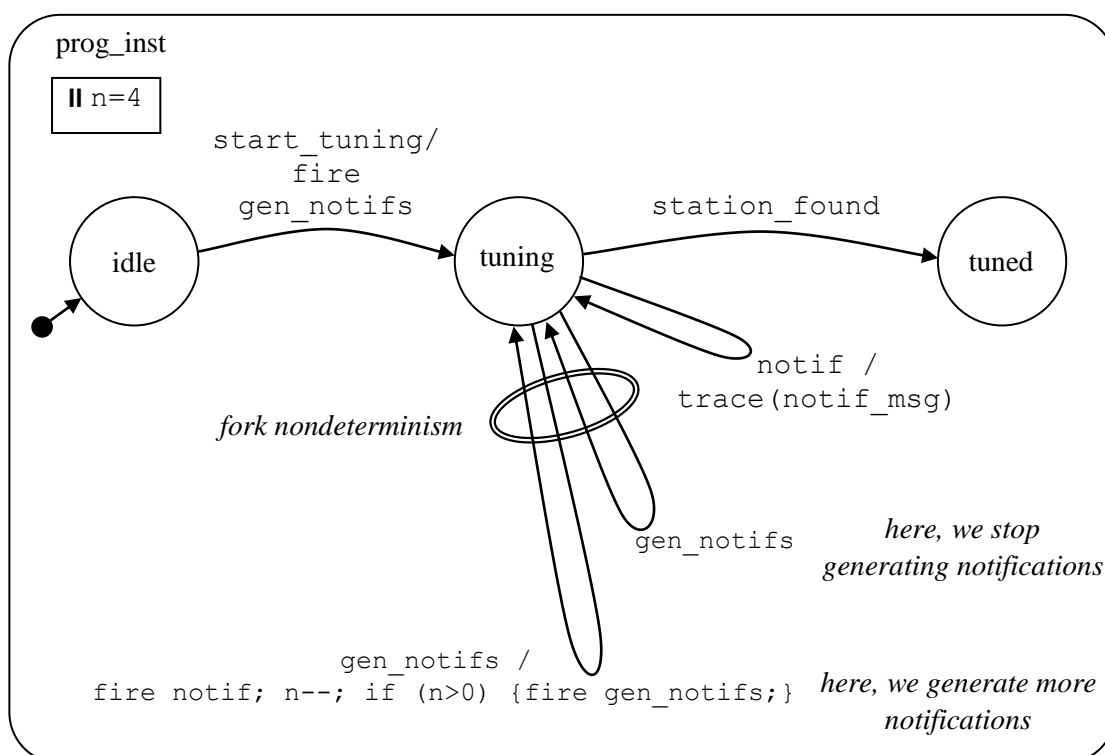


**Figure 50. Notifications**



A problem involving notifications is a good example of parallelism, two threads being active in parallel. It has the added difficulty that notifications are events that are *generated by the system under test*, rather than being events that are *offered to the system under test*. The result is that the system behaves nondeterministically - the number of notification events that will be generated is not known a priori by the state model. This is not a problem until the potential number of notifications becomes large, which is exactly what happens in the example we investigate: a TV program installation example. Program installation for one channel is a process of searching for a station with the tuner, reporting with notifications that that the search is in process. If a station is found, it will be registered. If no station can be found, the TV remains untuned. The program installation process can be stopped at any time.

The state behaviour is (in part) represented by the following figure.



**Figure 51. Notification example [model t4152]**

On the `start_tuning` event, the TV searches for a station by tuning. During the search, notifications are generated, representing “search in progress”. These notifications can be used to fill a progress bar. From the above model STATECRUNCHER generates worlds containing various numbers of notifications. When a station is found, the `station_found` event is generated. A fuller model would allow for stopping the program installation, and for failure to find a station.

In a composite system of program installation and tuner as above, the `start_tuning` event is under the tester's control, but the `notif` and `station_found` events are generated internally to the IUT (Implementation Under Test), ultimately by hardware. The problem arises that a large number of notifications could be generated. The above model caters for up to 4 notification messages by using fork nondeterminism on event `gen_notifs` to generate:

- a world with no notifications
- a world with 1 notification
- a world with 2 notifications
- a world with 3 notifications
- a world with 4 notifications

The STATECRUNCHER traces corresponding to this are:

```
4   TRACE =[]
9   TRACE =[notif_msg]
14  TRACE =[notif_msg, notif_msg]
19  TRACE =[notif_msg, notif_msg, notif_msg]
23  TRACE =[notif_msg, notif_msg, notif_msg, notif_msg]
```

In practice, over 800 notifications can be generated. This number of worlds is rather excessive for STATECRUNCHER. What solutions can be found? One is to change our model of testing. Up to now we have been treating the model and the IUT symmetrically (Figure 7, Figure 22), giving them the same input and comparing their output. With white-box testing we can set and observe states, but with black box testing, we are restricted to processing events and observing trace output.

The following improvements in efficiency are possible:

- Allowing for repetitions
- Conversion of traces to pseudo-events.

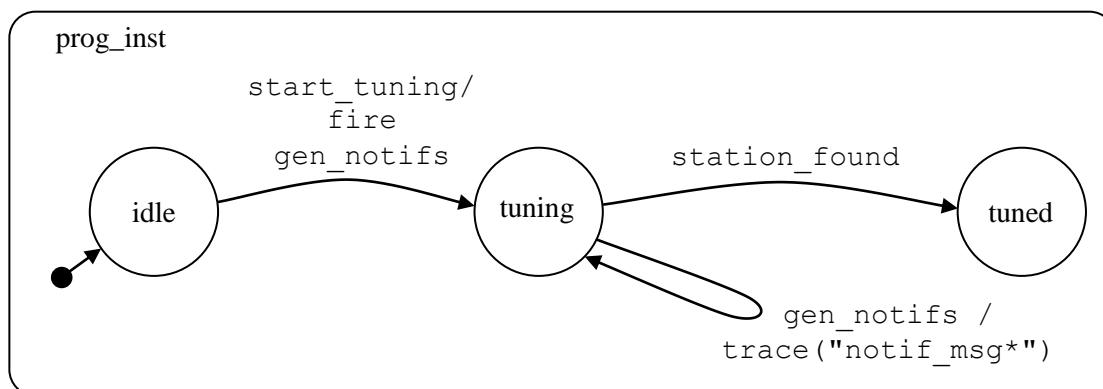
They involve some degree of asymmetry between model and IUT. The first still allows for simultaneous processing in IUT and model, but requires special interpretation of certain model outputs. The second has the IUT precede the model in execution, and interprets IUT output in determining how best to verify against the model.

A third technique is:

- Pruning worlds based on traces (section 4.10.3.3).

### ***Repetitions***

We allow the IUT time to produce several outputs before comparing them with the model's output. We use an asterisk convention that the comparator should allow any number of `notif_msg` traces from the IUT against a `notif_msg*` trace from the model.



**Figure 52. Repetition convention [model t4153]**

After processing event `start_tuning`, we have a trace of

```
5 TRACE =[notif_msg*]
```

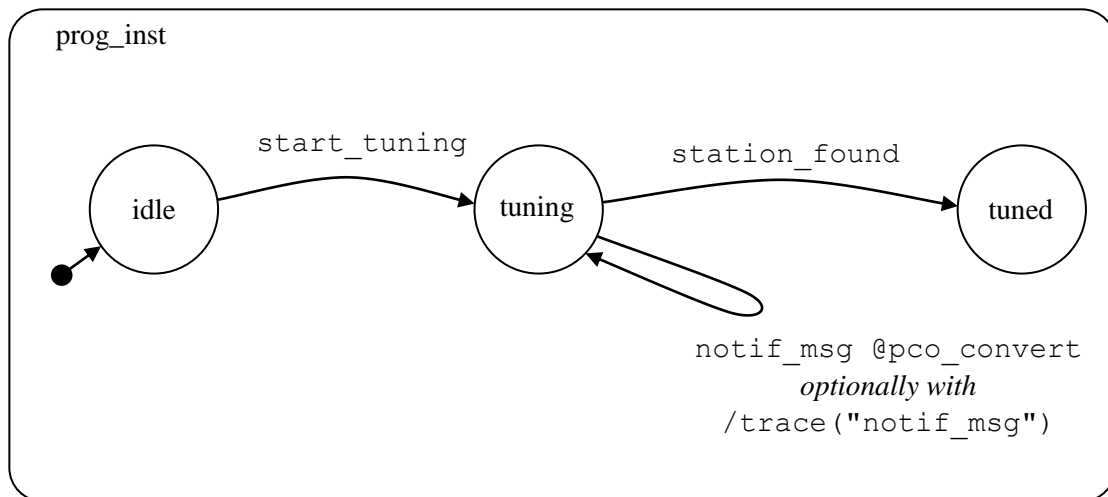
However, a problem arises if there can be several separate arbitrarily-interleaved notifications from different servers. Although a convention could be elaborated to cater for this, allowing for expressions with *and* and *or* operators, it would be rather cumbersome.

#### ***Conversion of IUT traces to model events***

With this technique, we have a very simple model, as in Figure 53. In state `tuning`, we wait to see what the output the IUT produces before processing an external event. (The driver may have instructions to wait a certain time when it sees a transitionable conversion-type event, which can be identified by its PCO). Every time the IUT produces an *output* of `notif_msg`, we see whether the model allows a transitionable *event* named `notif_msg`, on a special PCO (Point of Control and Observation), indicating that the output can be converted to an event. We use the PCO name `pco_convert`. If this is the case, we have two approaches:

- allow the output without further ado, i.e. without processing any event in the model
- convert the output to an event and feed that event back into STATECRUNCHER and check that the actual trace produced matches the IUT output. This is shown by the *optionally with* `/trace("notif_msg")` action in Figure 53.

The former of these may be adequate in many cases and is very efficient; the latter may give extra flexibility, e.g. where a notification is parameterised, or where it causes a state change itself, or where the number of notifications must be counted in the model.



**Figure 53. Conversion of traces to events**

The number of worlds generated at any one time is kept to a minimum, because the notifications are processed one by one, and they do not in themselves entail nondeterminism. However, with the second option only, performance may be affected if the model is called a very large number of times.

#### 4.10.5 Summary of containment techniques

The following summarises the ways described for containing combinatorial explosion.

##### *Compact representation of a large number of states and transitions*

- The use of hierarchy and concurrency: STATECRUNCHER's clusters and sets
- Binary decision diagrams are efficient, and are used in SPIN.

##### *Minimising the number of states*

- Equivalence partitioning of numerical ranges; use an enumerated value per partition
- On-the-fly (adaptive) testing prunes away states that would have to be generated in batch (preset) testing.

##### *Limited state machine coverage in testing*

- Projection coverage.

##### *Nondeterministic restriction of world explosion*

- Fork nondeterminism: not controllable except by excision of forks in model source code
- Race nondeterminism: A race with  $n$  competitors can be set to
  - no race ( $1$  interleaving)
  - low race ( $2$  interleavings)
  - medium race ( $2n$  interleavings)
  - high race ( $n!$  interleavings)

- Set nondeterminism. Where there are  $n$  set member operations, the nondeterminism can be set to
    - no set tran ( $1$  interleaving)
    - low set tran ( $2$  interleavings)
    - medium set tran ( $2n$  interleavings)
    - high set tran ( $n!$  interleavings)
- Also
- The transition semantics avoid micro-orderings of set entry/exit
  - The hierarchical permutation technique, applied to nested sets, reduces the number of interleavings.

### **World pruning**

- Kill invalid worlds after every test
- Mid-algorithm world pruning based on *expected trace*
- A special technique when testing against a deterministic IUT [Hierons 98].

### **Handling notifications**

- Allow for repetitions of a notification in one pseudo-trace
- Conversion of traces to pseudo-events

## **4.11 Test generation under nondeterminism**

Whilst it is not STATECRUNCHER's responsibility to generate test sequences, (but that of its neighbour in the tool chain, the *primer*), we give some informal descriptions of some of the issues and approaches involved. For precise descriptions, see [Hierons 98] and the other publications referred to.

In section 3.2.4, we described some methods used in generating tests for deterministic finite state machines. When the specification is nondeterministic, we wish to show that everything the implementation can do is allowed by the specification. We do not need to show equivalence between specification and implementation, because the specification may *allow* certain aspects of behaviour whilst *not insisting* on them.

Various assumptions about the NFSM (Nondeterministic Finite State Machine) are generally necessary, including the fact that it is *observable*, an ONFSM, i.e. that a unique target state on a transition can be deduced from the output generated by the transition. A non-observable NFSM can be converted to an equivalent observable NFSM, (though, of course, knowing the state of the ONFSM does not imply uniquely knowing the state of the NFSM).

One definition of conformance of an implementation NFSM  $M_I$  to a specification NFSM  $M$ , is as follows. Define a *language* of an NFSM  $M$  with the *symbols* in its *alphabet* being input-event/output-trace pairs. The language of an NFSM  $M$ ,  $L(M)$ , is the set of such symbol sequences that can be produced by it.  $M_I$  conforms to  $M$  if  $L(M_I) \subseteq L(M)$ .

Tretmans, in a presentation on Côte de Resyste [CdR], (where inputs and outputs are both events, and traces are sequences of processed events, as in CSP) defines conformance of an implementation  $i$  to a specification  $\mathbf{s}$  as:

$$i \text{ ioco } \mathbf{s} \stackrel{\text{def}}{=} \forall \sigma \in \text{Straces}(\mathbf{s}) : \text{out}(i \text{ after } \sigma) \subseteq \text{out}(\mathbf{s} \text{ after } \sigma)$$

Tretmans explains this as:  $i$  **ioco**-conforms to  $\mathbf{s}$  iff

- if  $i$  produces output  $x$  after trace  $\sigma$ , then  $\mathbf{s}$  can produce  $x$  after  $\sigma$
- if  $i$  cannot produce any output after trace  $\sigma$ , then  $\mathbf{s}$  cannot produce any output after  $\sigma$ , (quiescence).

A test suite  $\mathbb{T}$  is *sound* if  $i \text{ ioco } \mathbf{s} \Rightarrow i \text{ passes } \mathbb{T}$ .

A test suite  $\mathbb{T}$  is *exhaustive* if  $i \text{ passes } \mathbb{T} \Rightarrow i \text{ ioco } \mathbf{s}$ .

Test sequence derivation algorithms for NFSMs are given by [Petrenko], who introduces the concept of *r-distinguishing* sequences to distinguish states in an observable NFSM. [Hierons 98] addresses the issue of testing an implementation that is known to be deterministic against a nondeterministic specification, introducing *d-distinguishing* sequences, that distinguish states on this assumption. The paper also shows how adaptive testing is more efficient than preset testing. [Hierons 03] addressing the same issue shows how a *candidate* can be used, a deterministic FSM that is generated from the nondeterministic specification and the implementation. It has the property that if the implementation conforms to the candidate, the implementation conforms to the specification. Tests can then be derived from the candidate, using test generation algorithms for deterministic FSMs. The references given cover more issues and cite additional authors on this subject.

Although there are algorithms for the generation of very strong test suites, we note that random testing is also very effective, and was used in the Côte de Resyste experiments [CdR], [Du Bousquet].

## 4.12 Summary of this section

We have seen how STATECRUNCHER supports the following forms of nondeterminism in a UML-like statechart: fork, race, set-transit, set-action, set-meta-event and fired-event nondeterminism. Combinations of these forms of nondeterminism can be present at the same time. For each outcome, STATECRUNCHER generates a *world*, and events are processed in all worlds. Reference has been made to some approaches to test generation when a specification is nondeterministic. We have considered how to contain combinatorial explosion of worlds.

STATECRUNCHER may be able to play a role in adaptive, on-the-fly testing, but this is a subject for further consideration and research. STATECRUNCHER can certainly flatten UML-style state spaces, and may be useful as a test oracle in adaptive testing too. For example, if after a test STATECRUNCHER has been left with more than one world (all with the same trace, but differing in internal state), and *if it is known that the implementation is deterministic*, then there may be very efficient disambiguating sequences of events (*d-distinguishing* sequences, [Hierons 98]) which could be applied to the IUT and STATECRUNCHER, after

which STATECRUNCHER would be pruned to the matching world only. However, this does not prune the underlying model, only the data it has produced. A future very advanced possibility would be for STATECRUNCHER to allow for adaptation of its model, whereby states and transitions can be created and destroyed.

Precise details of the language syntax, of design considerations, of the transition algorithm, and of the implementation strategy have not yet been given. These are the subjects of the ensuing chapters.

## 5. STATECRUNCHER as a language

In this section we describe STATECRUNCHER primarily from a syntactic point of view. The aspects of syntax and parsing fall into three main areas: declarations, expressions/operators and the transition block.

### 5.1 General syntax

STATECRUNCHER syntax is an extension to that described in [CHSM] and [ECHSM]. The distinguishing feature of STATECRUNCHER is primarily its semantics, with its handling of nondeterminism, rather than its syntax.

Before the detailed syntax of states, clusters, sets and statecharts is described, some introductory syntax descriptions and conventions are needed. Then we use the ‘railroad’ diagramming technique to describe the main syntax. The diagrams contain ‘reverse-flow’ arrows to represent repetitions; the syntax is actually implemented in PROLOG Definite Clause Grammars (DCG’s) – which requires a ‘forward-flow’ only description, using recursion to obtain arbitrary repetition. For parsing details, including a forward-flow description of the grammar, see [StCrGP4] and [StCrParsing].

#### 5.1.1 General syntax conventions

This subsection covers aspects of syntax that could be applicable to any statement.

1. Statements currently must be written on a line of their own, and only on one line, except that a continuation character, the backslash, "\", may be used at the end of a line to denote continuation onto the next line. Use of the backslash may be repeated over many lines. Avoid having *anything* (e.g. spaces, comments) following the continuation character on the same line; it must be the last character of the line.
2. STATECRUNCHER syntax is case sensitive throughout. Language keywords must be specified in the correct case. User-defined names (identifiers) must be consistent with respect to case.
3. Identifiers are user-defined names of states, events, variables etc. The rules are:
  - Identifiers must not be a language keyword, transition label or function name.Language keywords are:

bool	clear	cluster	deep	deep_clear
else	enter	enum	event	exit
false	fire	history	if	in
is	PCO	set	state	statechart



true	upon			
------	------	--	--	--

Keywords reserved for transition labels are:

cost	name	time	utility	
lk_cost	lk_name	lk_time	lk_utility	

Function names are:

abs	cast	format	get_nworlds
high_race	high_set_tran	length	lower_case
low_race	low_set_tran	maximum	med_race
med_set_tran	minimum	no_race	no_set_tran
upper_case			

- Identifiers must begin with a letter (upper case or lower case) or an underscore. This is optionally followed by a sequence containing upper or lowercase letters, decimal digits and underscores.
4. Numbers are in accordance with their representation in C. Real numbers are not currently supported in any STATECRUNCHER statement.

Examples of integer constants<sup>1</sup>:

0                      -0                      123                      -123  
013 (*octal*)              0X12f (*hex*)              0x12F (*hex*)

Examples of character constants:

'C'                      'x'                      '\n'                      '\36' (*decimal*)  
'\057' (*octal*)              '\0x2F' (*hex*)

No distinction is made in STATECRUNCHER in practice between characters and integers.

5. White space, used to separate syntactic items, consists of a sequence containing the following characters (with their decimal ASCII code)<sup>2</sup>

space (32)	alert (7)	backspace (8)	horizontal tab (9)
line feed (10)	vertical tab (11)	form feed (12)	

<sup>1</sup> Additionally, the suffixes for long and unsigned or both may be appended, e.g. 123l(*long*) 123L(*long*) 123u(*unsigned*) 123U(*unsigned*) 123ul(*unsigned long*) 123UL(*unsigned long*). 123Lu(*unsigned long*) However, these do not alter the internal representation.

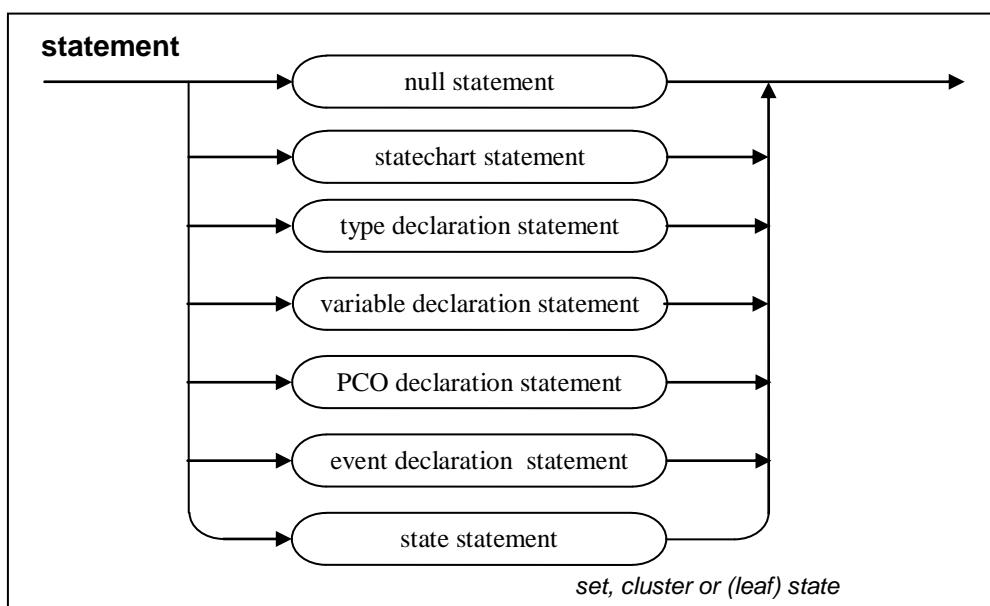
<sup>2</sup> For normal use the white space characters are **space** and **horizontal tab**. An embedded backspace does not remove the preceding character. Line feed and/or carriage return may not be possible as embedded characters as they may be absorbed in the line read process. DOS and Unix have different end-of-line conventions. The user need not normally be concerned about this. Some text editors may not allow embedding of some of these characters in a file.

Comments (see below) also count as white space. White space can be omitted where that does not lead to an erroneous tokenization or parse. For example, if there is no white space between the keyword **cluster** and the identifier **volume**, a new identifier **clustervolume** is formed, so white space is required. But after brackets, commas, operators, semicolons etc., no white space is required.

6. Comments in STATECRUNCHER source can be in either of the following styles, or a mixture of both:
  - the 'C' and PROLOG convention: `/* . . . . .*/` The comment must be closed in the statement which opened it.
  - the 'C++' convention: `// . . .` (running to the end of the line)
 The continuation line character, backslash, "\", retains its continuation function after 'C++' style comments, and does not terminate a // comment.

## 5.2 STATECRUNCHER statements

A STATECRUNCHER model consists of **statements**. The figure below shows this top level of the STATECRUNCHER grammar.



**Figure 54. STATECRUNCHER statements**

In the sections following, these statements are considered individually.

### 5.3 Basic syntax of statechart / cluster / set and (leaf-)states in a hierarchy

We now show how to define hierarchical states in a STATECRUNCHER model. The grammar is shown with reverse-flow for compactness; for the feed-forward transformation (which is not difficult for this part), see [StCrParsing].

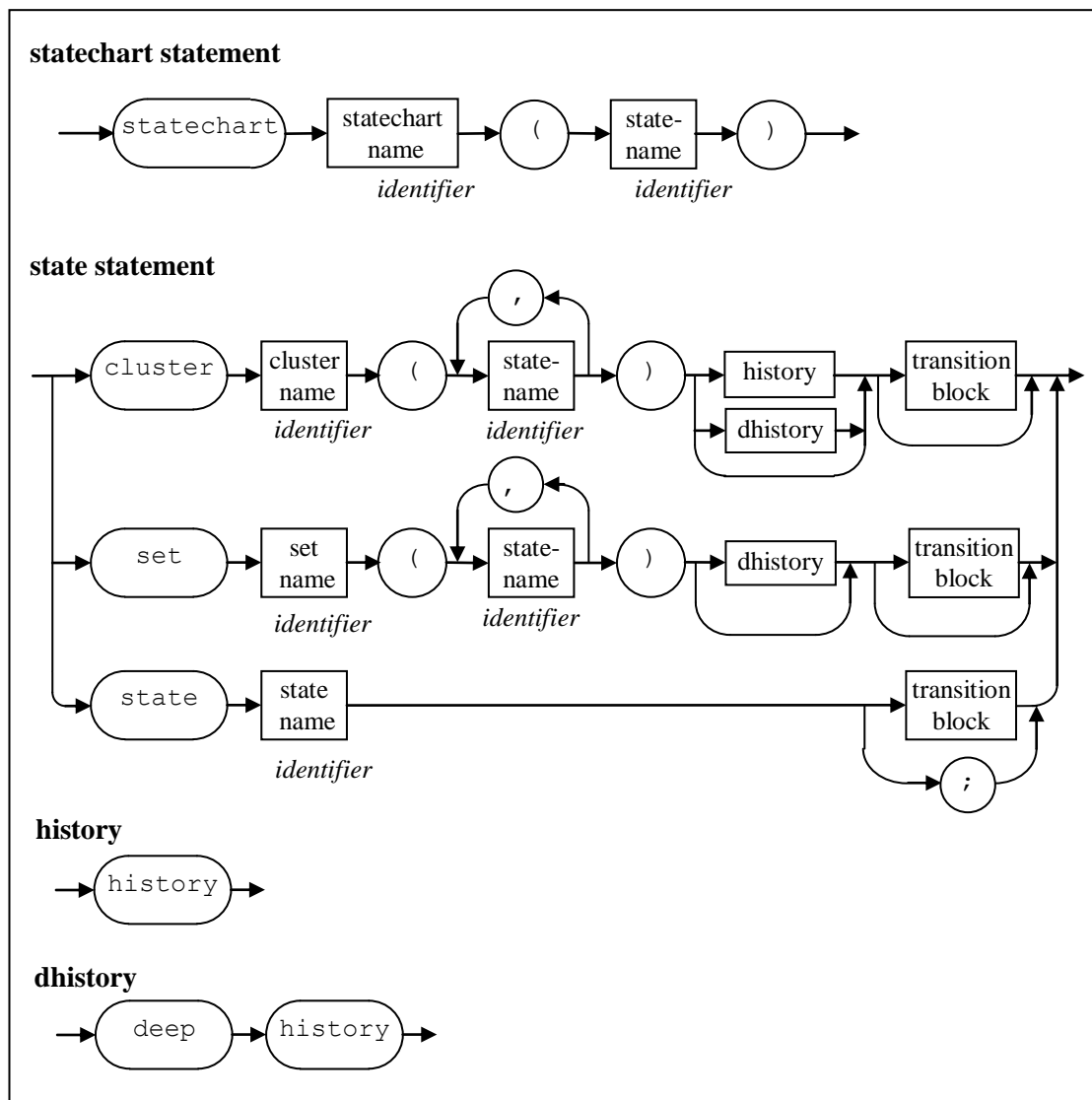


Figure 55. Basic syntax of statechart / cluster / set and (leaf-)states

The **statenames** block contains the names of the member states of the cluster or set. The statements defining these member states must occur immediately after their parent. This gives the entire hierarchy a depth-first structure, as will also be seen in the example that follows.

If there is an error in defining the member states (because the child states *announced* in a parent state do not actually occur, or do not occur in the right place), this is flagged as a **machine path error**. The *machine path* is the hierarchical path from the statechart level down the hierarchy to a state at some place in the hierarchy.

**History** and **deep history** are described in more detail here; their effect on the ‘transition course’ is considered in detail in section 7.5.

The **transition block** is considered in section 5.8.

## 5.4 More about hierarchical states


### 5.4.1 Statecharts

A STATECRUNCHER model is wrapped in the highest (outermost) hierarchical level by a ‘statechart’. This formality does not offer any additional functionality, except to provide a clear marker as to where one or more ‘statecharts’ starts in a source file (but currently only one is supported).

### 5.4.2 Clusters

A cluster is a group of states (members of the cluster) such that at most one member state can be occupied. If one member is occupied, the cluster is regarded as occupied. If all members are vacant, the cluster is vacant. The members of a cluster can be other clusters, sets (to be introduced) or leafstates.

The diagrammatic notation for a cluster is a rounded rectangle with its name at the top left.

One member of the cluster is designated the default member (symbol ). This state is entered:

- if the cluster is entered when the statechart is initially entered
- if the cluster is the target state of a transition (to be discussed in detail later), unless other (history-related) factors come into play.

Transitions can have a cluster as their source state. They can also have a cluster as a target state – details of this will be discussed later. This gives a compact way to express what would otherwise be multiple transitions.

The following diagrams show by example how a cluster is equivalent to a flat state machine, i.e. one without hierarchy.

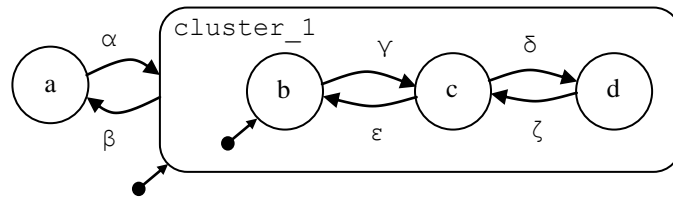


Figure 56. Cluster with transitions

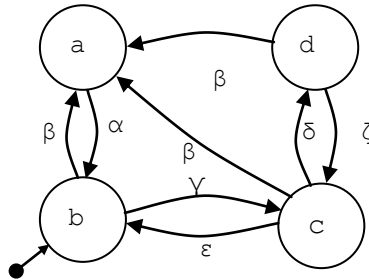


Figure 57. Cluster - equivalent flattened state machine

### 5.4.3 History and deep history

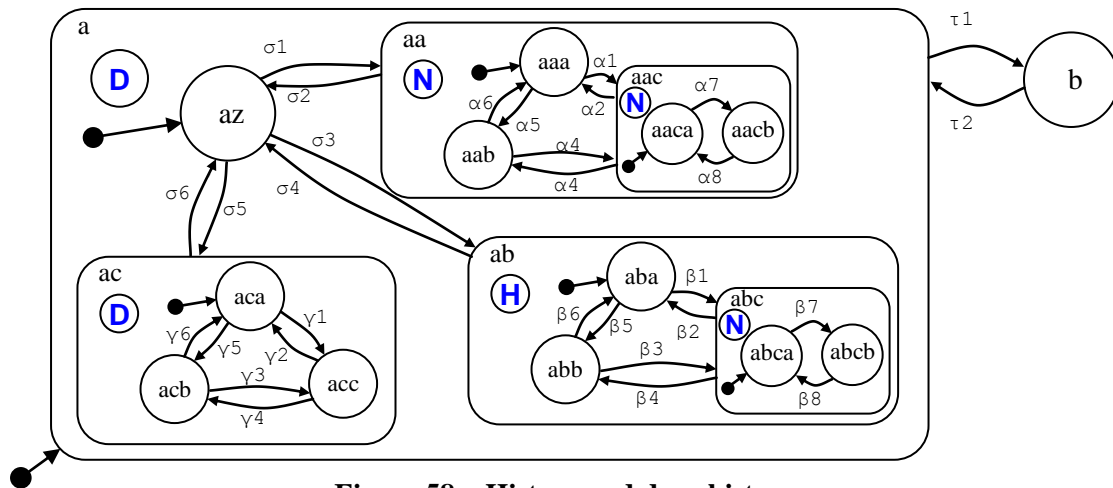
A cluster can be marked with a history or deep history *marker*. The history *data* records the member that was occupied when the cluster was last occupied.

On our diagrams, history is marked according to the following legend:

N = no history (default)     
 H = (shallow) history     
 D = deep history

A cluster with a history marker, when it is targeted without a specific member being specified, will enter the historical state. This assumes the history data is available – otherwise the default state will be taken. Deep history indicates that historical data is to be used (assuming it is available) on re-entering the cluster and *all descendant clusters* below the marked cluster. The descendant clusters are entered under a *deep history obligation* – whether or not they have a history marker. The deep history obligation is *not* applicable simply because a particular cluster is below another one with a deep history marker. It must be the case that the cluster with the deep history marker is *actually entered* in the course of the transition for the deep history obligation to apply. ‘Low flying’ transitions will not ‘see’ the deep history marker.

In practice, history data is saved whenever a cluster is exited, and decisions are taken on whether to *use* the data on cluster entry. The following statechart shows the basic use of history.



**Figure 58. History and deep history**

We consider some transitions:

- The transition on  $\tau_1$  causes cluster *a* to be exited. The transition on  $\tau_2$  causes it to be re-entered, and as cluster *a* has a deep history marker, it and all descendants will assume the previous occupancy (for example, states *ab*, *abc* and *abcb*, showing the applicability of history in a cluster without a history marker).
- The transition on  $\sigma_2$  causes cluster *aa* to be exited. The transition on  $\sigma_1$  causes it to be re-entered. The deep history marker in cluster *a* is *not* effective, as cluster *a* is not being re-entered on this transition. Since cluster *aa* does not have a history marker, the default member state is taken: this is state *aaa*.
- The transition on  $\sigma_4$  causes cluster *ab* to be exited. The transition on  $\sigma_3$  causes it to be re-entered. The history marker in cluster *ab* indicates that the historical member is to be entered. Suppose this is *abc*. Cluster *abc* is duly entered, followed by its default member: state *abca*.

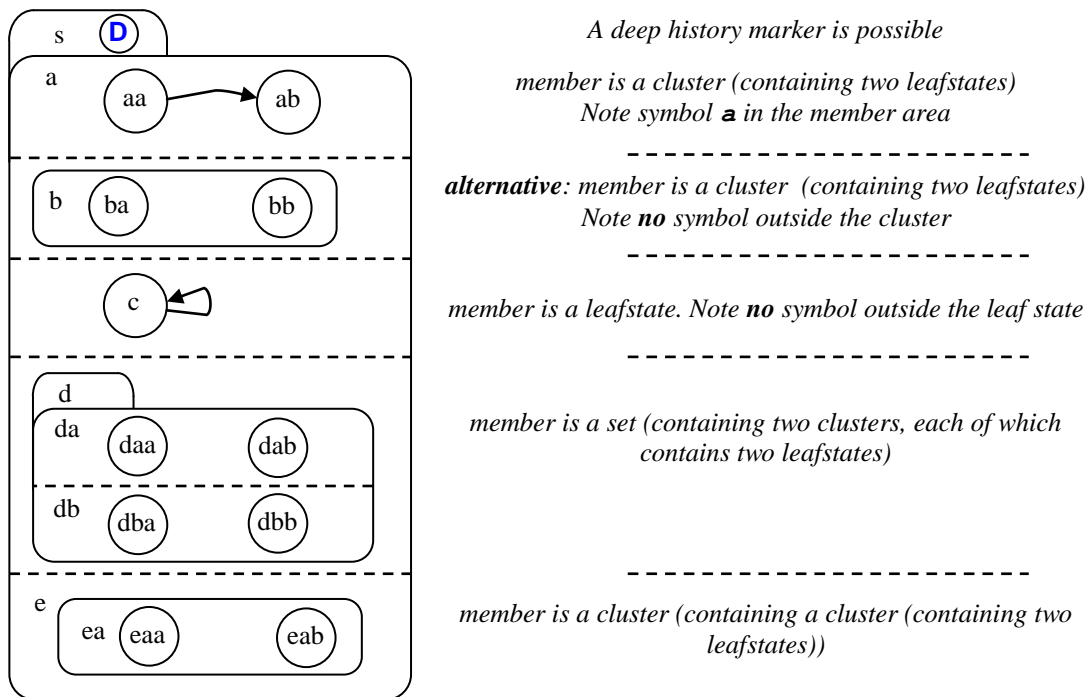
Notes:

- History *data* can be cleared (as an action - described later) using the functions `clear(state-expr)` and `deep_clear(state-expr)`.
- A *set* (to be described) cannot be marked with a history marker, but it can be marked with a deep history marker.
- History also impinges on the ‘transition course’ under more complex circumstances – such as transitions targeting a parent state of the source state – to be described later.
- STATECRUNCHER may be changed in the future to handle UML pseudo-states, where it is the transition, not the cluster, that specifies how history is to be handled. But STATECRUNCHER can simulate these, since all clusters can be marked with deep history, and history can be cleared beforehand when the historical states are not required.

### 5.4.4 Sets

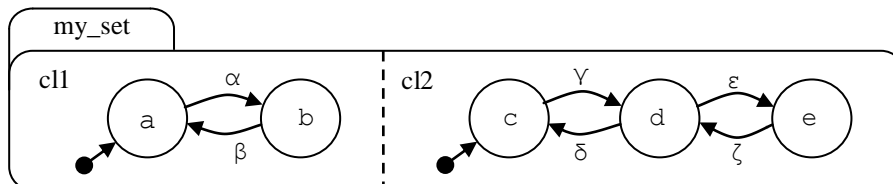
A set is another way to group states hierarchically. If a set is occupied, *all* its members must be occupied. If the set is vacant, all its members must be vacant. The members of a set can be clusters, sets or leafstates. A set normally has at least two members, though it may have just one (but, in STATECRUNCHER, not zero). This gives the statechart concurrency (i.e. parallelism): several states can be occupied in parallel.

The notation for a set is a rounded rectangle with a tab. Members are separated by a dotted line. If the member of a set is a cluster, no separate enclosing rectangle around the cluster is required; the symbol in the member area indicates a cluster. The following figure shows how members of sets can be designated.

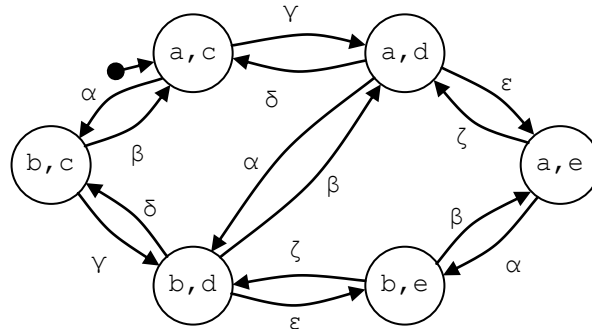


**Figure 59. Notation for members of sets**

The following diagrams show how a set is equivalent to a flattened state machine:

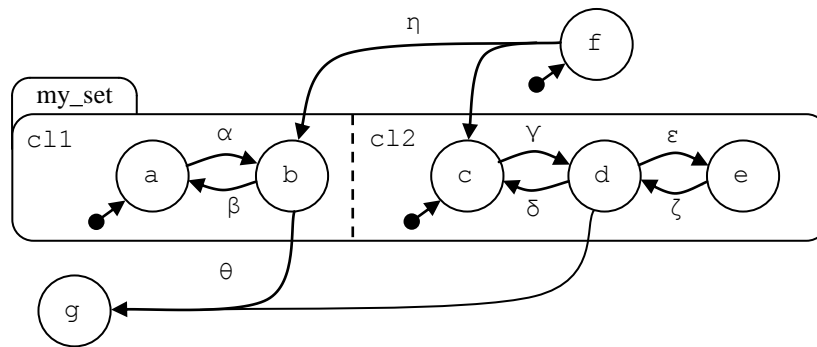


**Figure 60. Set with transitions**



**Figure 61. Set - equivalent flattened state machine**

Transitions can have multiple targets so as to specify which states within set members are entered. They can also effectively have multiple source states, indicating that the transition requires all the source states to be occupied, but this must be modelled in STATECRUNCHER as a transition from one of the source states with a condition attached, testing for occupancy of the others. Conditions are described later.



**Figure 62. Transitions with multiple source and multiple target states**

### Sets and history

A set cannot be marked with a history marker, since there is no choice as to which member to enter – if the set is entered, all its members are entered. A set can be marked with a deep history marker. This means that on entry into the set and then into the set members, a *deep history obligation* will be passed on to all members of the set. Any clusters below the set in the hierarchy will then be entered in their historical state, in the same way as was described under cluster deep history.

### 5.4.5 Example of hierarchical states

In the figure below, default states are marked in **bold** font. The source code is shown alongside.



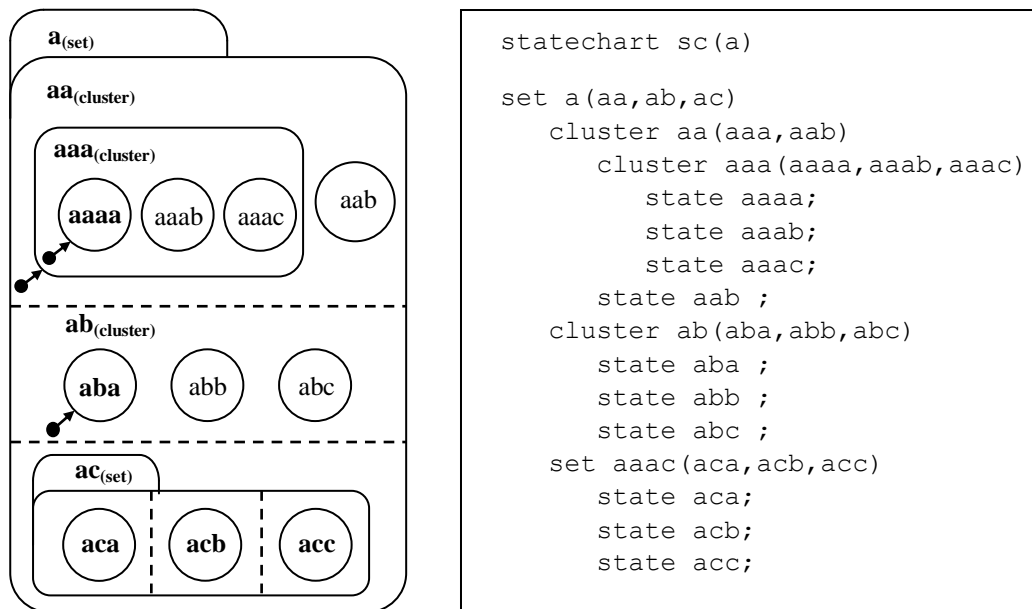


Figure 63. Example of hierarchical states [from model t6205, without aa prefixed]

## 5.5 Declarations and scoping

STATECRUNCHER supports the following declared items

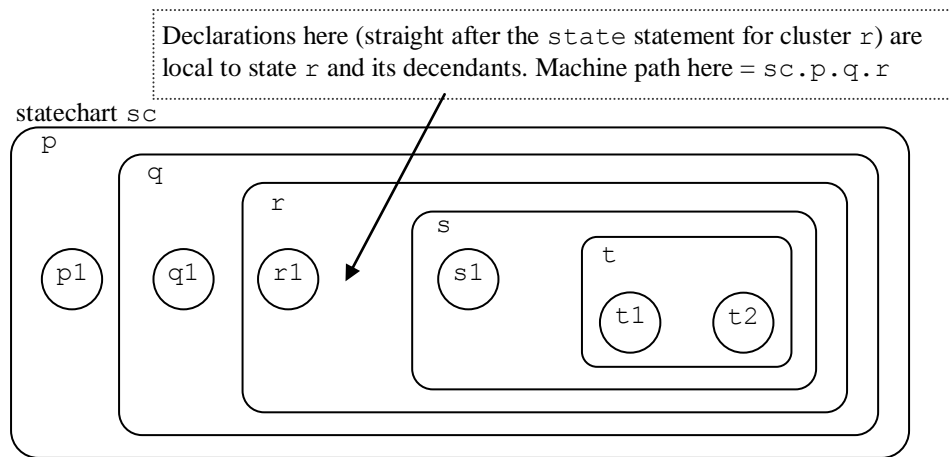
- States
- PCOs: Points of Control and Observation
- Events
- Types
- Variables

In STATECRUNCHER it is not necessary for all items (states, PCOs, events and variables) to have unique names. There can be *global* and *local* definitions of an item using the same name; the items are then quite distinct. This is roughly equivalent to global and local variables in ‘C’. STATECRUNCHER uses scoping operators to ensure that all items are accessible everywhere, if required.

The scope of an item is given by a *machine path*. This is a sequence of hierarchical states starting at the statechart level and descending as far as some particular state. We denote the sequence using a dot to link the states in the path, e.g. `sc.p.q.r`, or the internal representation, a PROLOG list in reverse order, also used in STATECRUNCHER output: `[r,q,p,sc]`.

The way states are declared has already been seen. Other items (PCOs, events and variables) can be declared straight after the `statechart` statement, in which case they are, *in the absence of scoping operators*, global to the statechart, or they may be declared after any

state statement in the source code, in which case, *in the absence of scoping operators*, they are local to some part of the statechart.



**Figure 64. Scope of declarations - default**

Scoping expressions allow a declaration or a reference to be made to a non-default scope, which could be higher in the hierarchy, lower in the hierarchy, or across the hierarchy (e.g. in a cousin relationship). Example operators are the `$`, which backs out one level in the hierarchy, and the *dot*, which deepens the machine path by the operand following it. There are more (described in section 5.6.2.2). These operators will probably only rarely be employed directly by the user. However, statechart composition utilities may make copious use of them.

The use of scoping expressions means that, in the syntax which follows shortly, an expression will be seen where just an identifier might have been expected. For example, an event can be declared as

```
event alpha;
```

but where `alpha` stands, an *event-expression* is allowed, modifying the scope of the defined event. So we might see

```
event $$alpha; // scope is more global than current machine path
```

or

```
event a.b.alpha; // scope is more local than current machine path
```

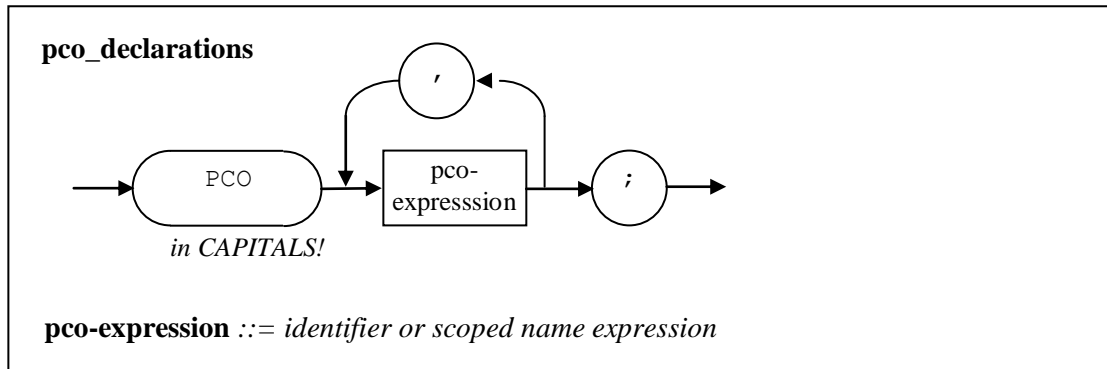
The syntactic items *PCO-expression*, *event-expression*, *tag-expression*, *var-expression* are **scoped-name expressions**. When evaluated, they return a *name* and a scope for that name. The syntactic item *expression* is a more conventional expression, using arithmetic operators, though scoping operators are *also* allowed. An *expression* evaluates to a *value*, not a name.

### 5.5.1 State declarations

States are declared and defined in the hierarchical way by the statements described in section 5.4. The transition part of `state` statements is described in section 5.8.

### 5.5.2 PCOs and events

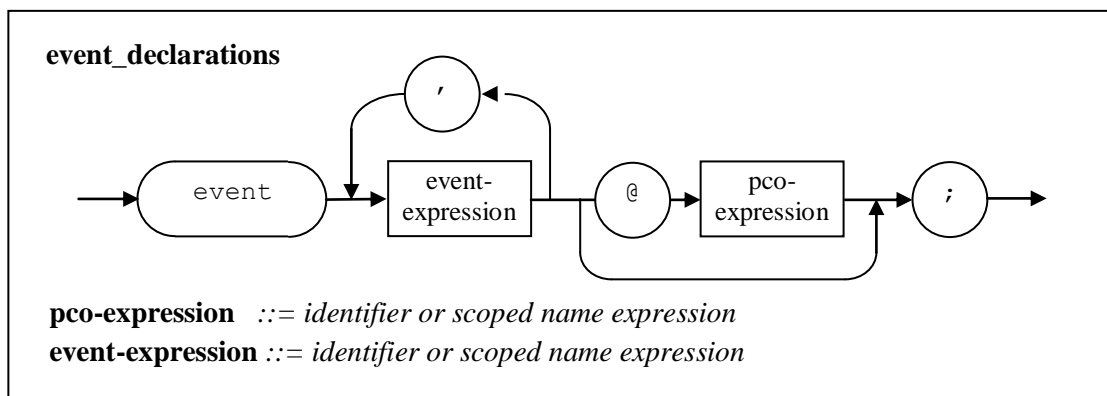
**PCOs** (Points of Control and Observation) must be declared in order to be used (though they need not be declared in a source line preceding their use). **Events** must similarly be declared in order to be used. PCOs serve to classify events according to whether (and where) they are externally controllable and observable or not – but use of them is a *Primer* (test generator) affair, and all STATECRUNCHER does with them is to provide information on them in its output. There can be several PCO and event declaration statements in a STATECRUNCHER model.



**Figure 65. PCO declarations**

Examples:

```
PCO pco1;
PCO alf,bert,$$bert,charlie; // two berts (in different scopes)
```



**Figure 66. Event declaration**

Examples

```
event alpha;
event beta,$$gamma,delta@pco1;
event $$epsilon,zeta@$$$$pco2;
```

Events are not declared with parameters, but, as will be seen, *transitions* can be labelled with events *and their parameters*.

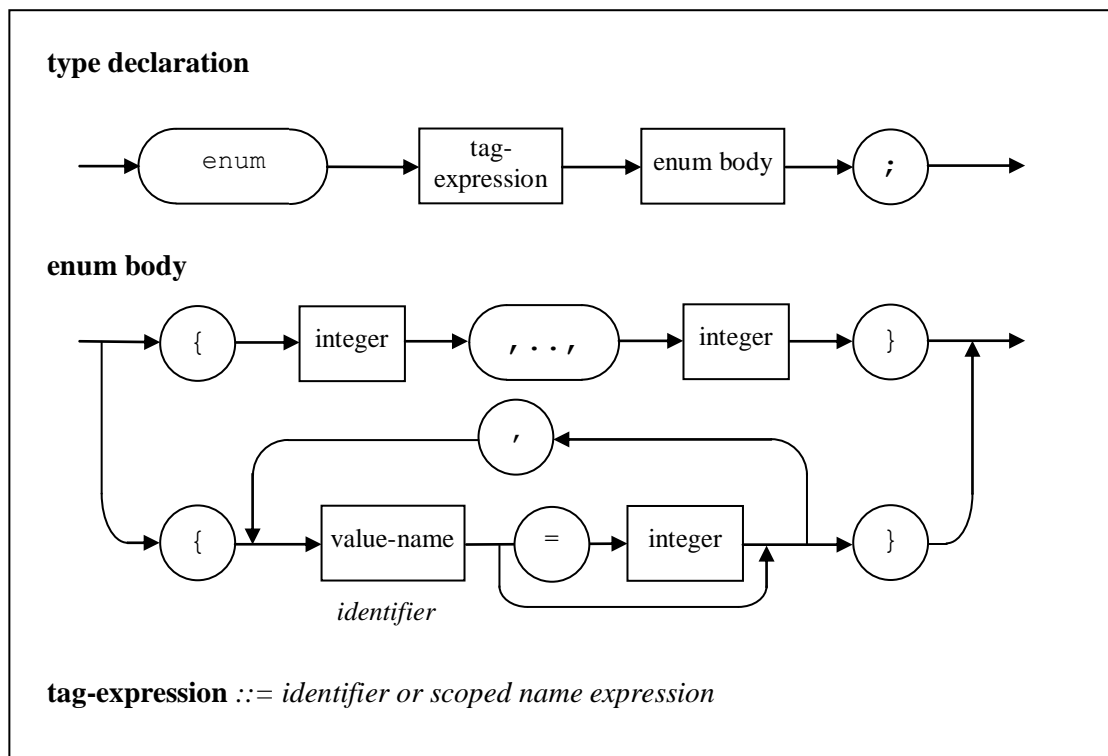
### 5.5.3 Types and variables

Variables in STATECRUNCHER are typed. The types are

- bool (boolean) – this is a built-in type
- user-typed using an integer range
- user-typed using integer enumeration by means of tagnames
- strings

Reals are not supported. They would make a finite state space infinite, (theoretically; in practice, just very large), and the user when modelling a system should always partition reals into equivalence classes and model these with integers.

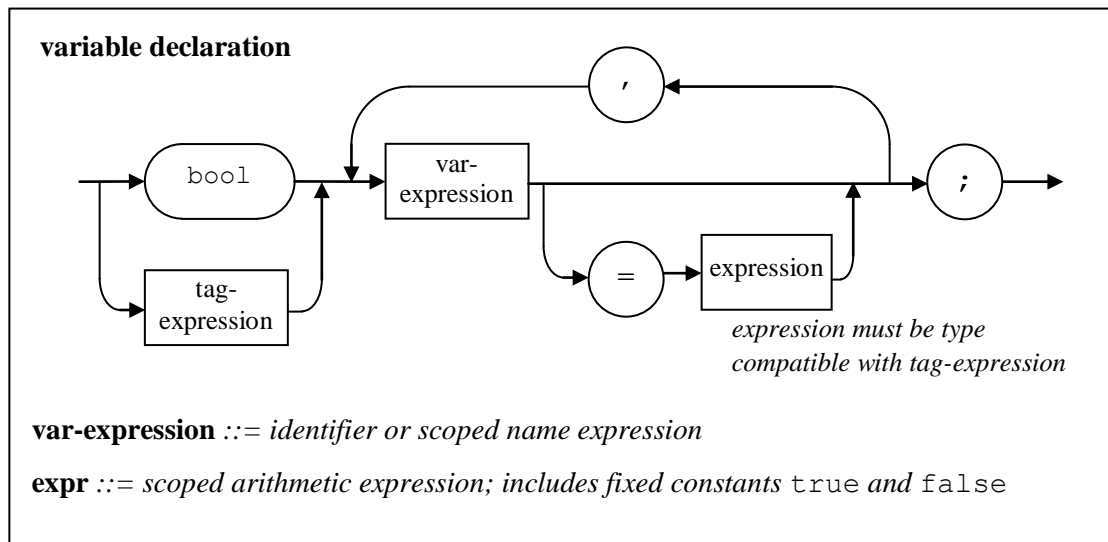
Type declarations and variable declarations are separate statements.



**Figure 67. Type declaration**

Examples of type declarations

```
enum channel {14, ..., 18};
enum colour {red=6, blue, green=9};
enum $$channels {90, ..., 99};
enum $$colour {white, red=6, blue, green=9};
```



**Figure 68. Variable declaration**

Examples of variable declarations

```
bool b1;
bool b1,b2=true,b3=false,b4=b2 && !b3;
bool $$b1=false;
```

```
channel    fav_channel=15,your_channel=fav_channel+2;
$channel   $favourite_channel=91;
```

```
colour     tie_col, sock_col=maximum(red,green,blue);
colour     $$tie_col, $$sock_col=$$red;
$$$colour  $my_tie_col = $$colour_of_the_day;
```

### 5.5.4 PCOs, events and variables in diagrams

Since PCOs, events and variables can also have the same name in different scopes, it may be desirable to show where they are declared. We do that with the  $\textcircled{P}$ ,  $\textcircled{N}$  and  $\textcircled{L}$  symbols. In the absence of any symbol, the names can be considered unique and in scope, though it is not specified whether they are global or local.

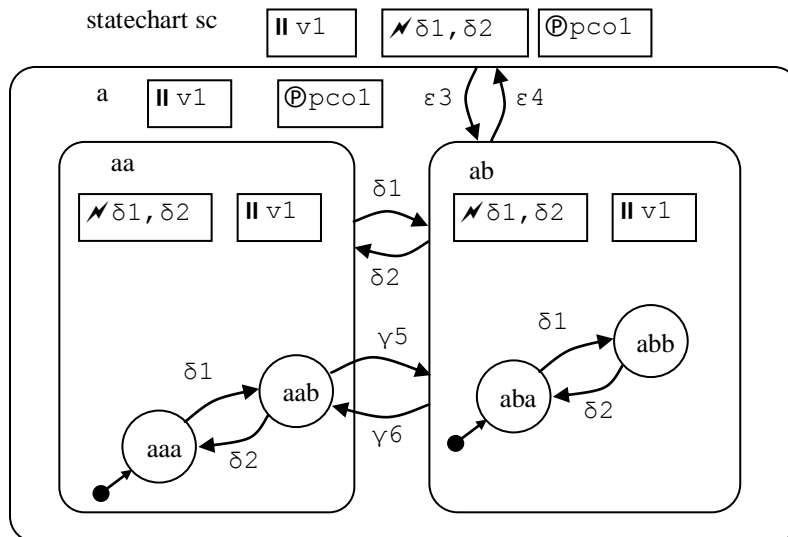


Figure 69. PCOs events and variables in diagrams

In the above figure, there are

- PCO declarations in scopes  $sc$  and  $sc.a$
- event declarations in scopes  $sc$ ,  $sc.a.aa$ , and  $sc.a.ab$ .
- variable declarations in scopes  $sc$ ,  $sc.a$ ,  $sc.a.aa$ , and  $sc.a.ab$ .

The effect of the event declarations is that the  $\delta 1$ ,  $\delta 2$  labels on transitions refer to different events according to the scope of the transition source state. Similarly, there are two PCOs called  $pcol$ , which must be distinguished. Similarly again, any expressions using variables (not shown on the diagram) would address the appropriate variable  $v1$ .

## 5.6 Expressions, operators and functions

### 5.6.1 Expression parsing

Expression grammars can be represented in a *feed-forward* form and so that parsers can be implemented using PROLOG Definite Clause Grammars (DCGs). For an early paper illustrating the principle, with two operator precedences, (but dating from before the PROLOG “ $\rightarrow$ ” DCG notation), see [Warren].

Expressions in different contexts can be allocated different operator sets, and parsed using the GP4 parser – details of this are given in [StCrGP4]. We give a summary and a flavour of that here, by showing a left-recursive grammar and its transformation into a feed-forward grammar for expressions. The grammar terminals are tokens from a lexical pass performed by GP4, which include constants, identifiers and strings, but not operators, which are identified at expression parsing time. Expressions and terms are parameterized according to their

precedence level, i.e. the level of operator precedence that is being parsed, with higher precedence *expressions* forming *terms* at the lower level concerned. A few features that are not pure syntax were introduced:

- Expression grammar rules are parameterized with a precedence level, which is the precedence level of the operators used to combine terms in the grammar rule for the expression at that level.
- Term sequences are also parameterized with an associativity parameter.
- Some small non-grammar operations are performed, indicated by  $\rightarrow$  with a double slash through it.

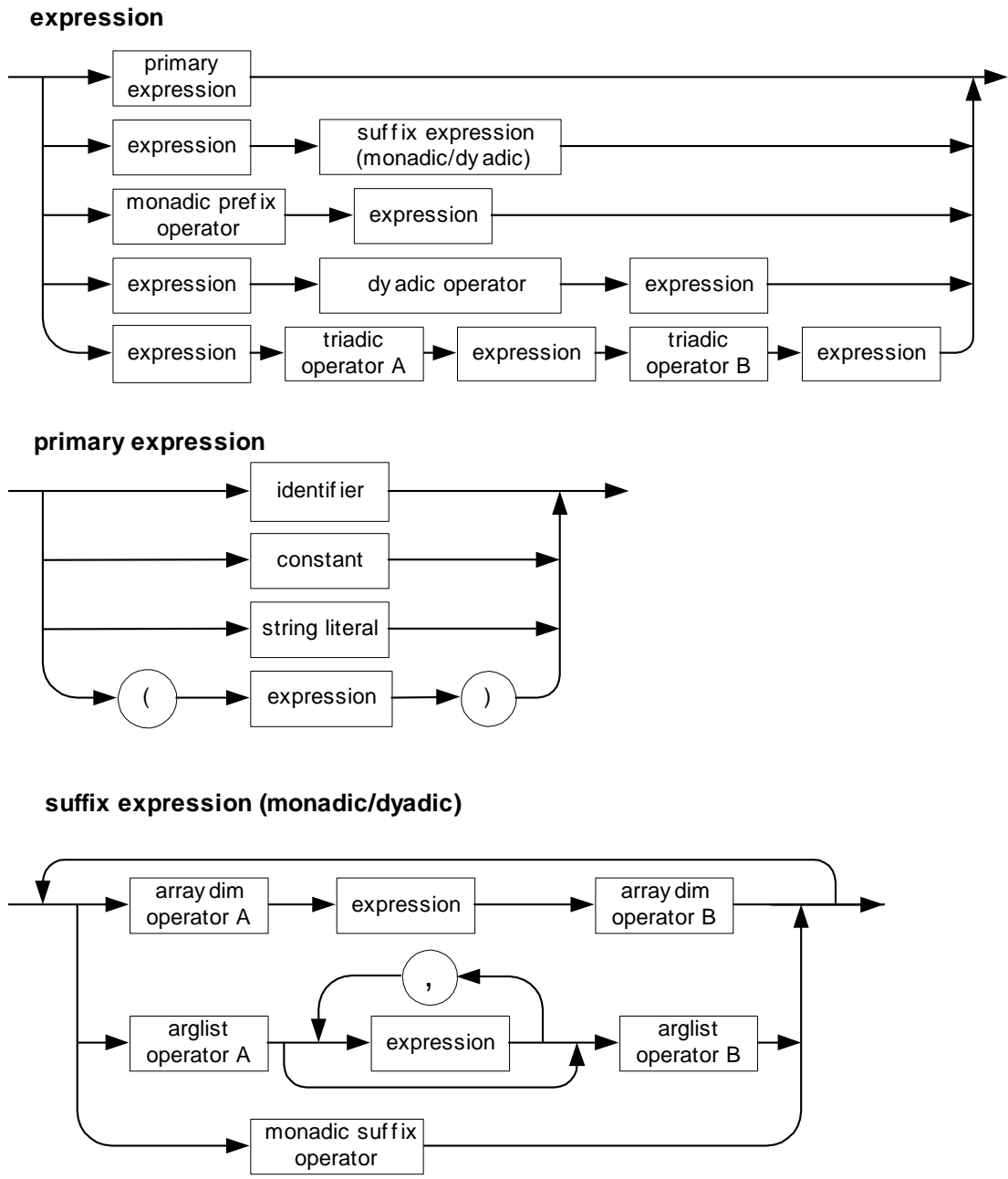
Examples:

- to *left associate*, which basically transforms  $[a+b-c+d]$  into  $[[a+b]-c]+d]$
- to test for a property, or assign a parameter (such as  $ASSOC=yfx$ ).

**Arity, position, and associativity** are defined as follows in the grammar (analogously to a PROLOG convention):

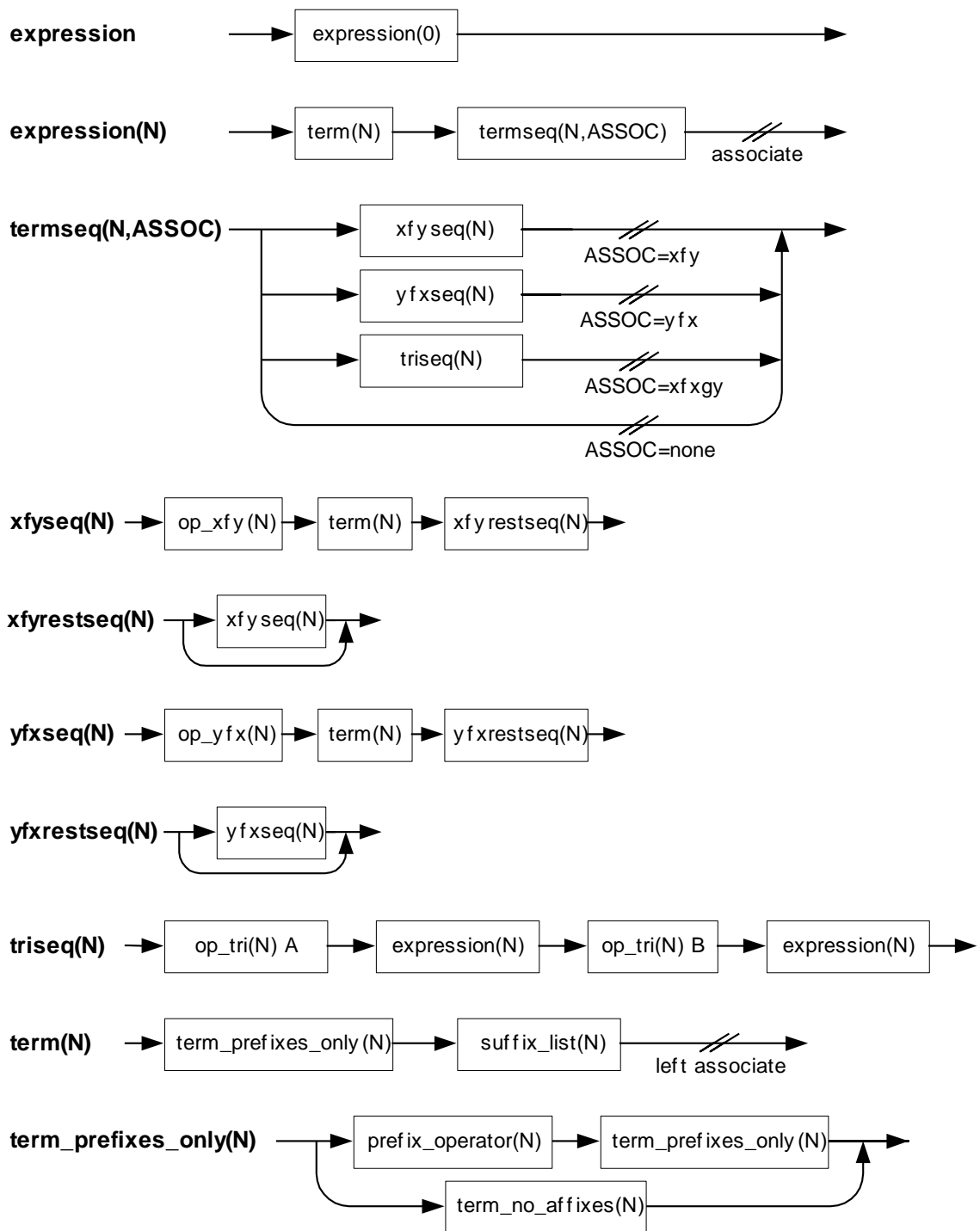
- $fx$  monadic, prefix, non-associative
- $fy$  monadic, prefix, right-associative
- $xf$  monadic, postfix, non-associative
- $yf$  monadic, postfix, left-associative
- $xfy$  dyadic, infix, right-associative
- $yfx$  dyadic, infix, left-associative

The diagrams below are not claimed to be an original exhibition of a general expression grammar, but Figure 71, Figure 72 and Figure 73 were constructed from first principles by the author from the left-recursive grammar of Figure 70, which is a variation of the expression grammar for 'C' given in [Darnell]. (A moderate amount of searching and enquiry amongst compiler colleagues failed to come up with anything explicitly similar, apart from the early example of [Warren], though it could be argued that many parsers, though outwardly not similar, *effectively* implement what is shown here). For that reason, the approach may have some original aspects of some interest to others in a related field.

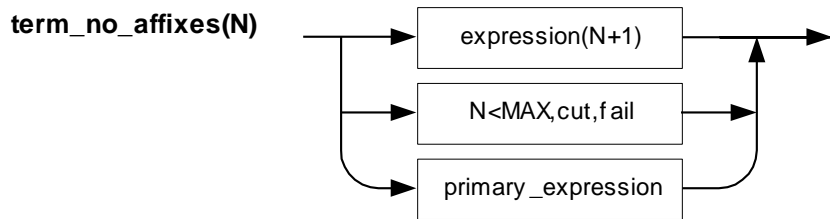


**Figure 70. Left recursive grammar (requiring transformation)**





**Figure 71. GP4 expressions - feed-forward grammar (1)**



The **cut,fail** combination is reached if the input stream cannot be parsed as `expression(N+1)`.

If  $N=MAX$ , we ignore the  **$N<MAX,cut,fail$**  route and proceed to look for a primary expression in the input stream

If  $N<MAX$ , we execute the **cut,fail** combination. This means that the syntactic item `term_no_affixes(N)` is considered to have failed to parse and no further options for it are to be examined.

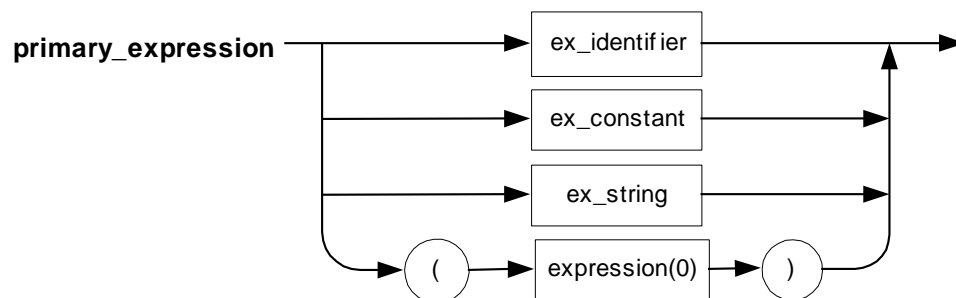
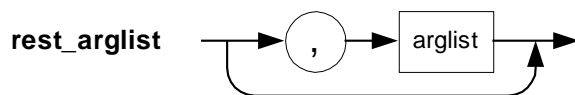
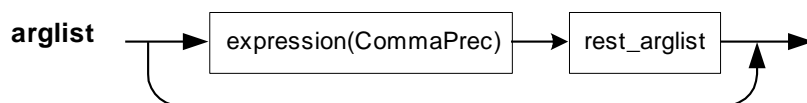
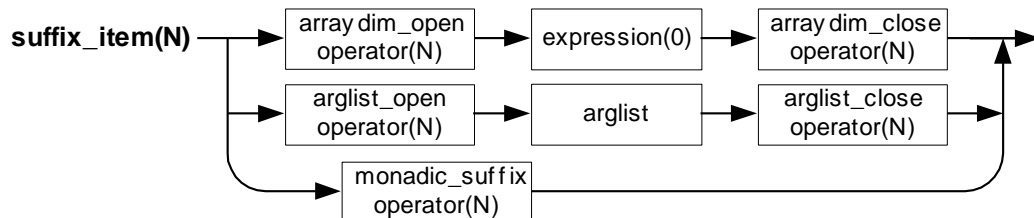
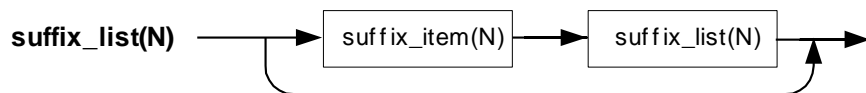


Figure 72. GP4 expressions - feed-forward grammar (2)

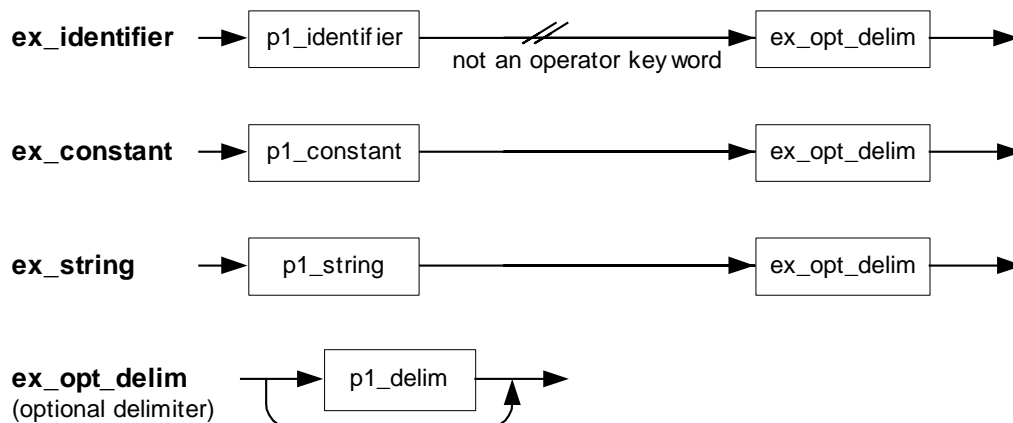


Figure 73. GP4 expressions - feed-forward grammar - (3)

### 5.6.2 Operators

Operators are used to construct expressions – including the initialisation expressions in variable declarations. The expression parser is supplied with a set of operators as a parameter per expression, so that it can parse the various kinds of expressions required according to their individual operator set.

STATECRUNCHER operators fall into two categories:

- Arithmetic operators, which return a *value*
- Scoping operators, whose action depends on the kind of expression in which they are applied:
  - In an *arithmetic expression* (sometimes just called an *expression*), they cause an evaluation to be performed under a modified scope, and the expression ultimately returns a *value*.
  - In a *state-expression*, *pco-expression*, *event-expression*, *tag-expression*, or *var-expression*, they return a *name*.

Most functions require that their parameters, (which are expressions) are evaluated to *values*. Currently, all functions return a *value*. There are also some functions (e.g. ‘in’), described later, which have special handlers, whereby the parameter is evaluated to a *name*.

These operators and functions can be mixed seamlessly in expressions.

Operators have the following attributes:

- A symbol, e.g. +, &&
- A name, used internally, which distinguishes between operators of like symbol, e.g. mplus (monadic plus), dplus (dyadic plus).

- A precedence (also called *priority*). Higher precedence operators bind their arguments before lower precedence ones. Note that this does **not** mean that they will necessarily be **evaluated** sooner, although this is sometimes performed the case. Example  $a-b+c*d+e = a-b+(c*d)+e$ , since multiplication has a higher precedence than addition and subtraction.
- A position. This can be
  - **prefix** (as in  $-x$ )
  - **postfix** (as in  $i++$ )
  - **dyadic infix** (as in  $a+b$ )
  - **post-circumfix** (as in the brackets of function call operator, e.g.  $\text{maximum}(a, b)$ ). Note how these operators come in two parts.
  - **triadic infix** (as in  $a?b:c$ ) –but this is not currently supported.
- An associativity. This can be
  - left associative:  $a+b+c+d$  is equivalent to  $((a+b)+c)+d$
  - right associative:  $a=b=c=d$  is equivalent to  $a=(b=(c=d))$
- An arity. This gives the number of arguments to the operator. It can be
  - monadic:  $-a$
  - dyadic:  $a+b$
  - triadic:  $a?b:c$  –but not currently supported.
- Some semantics. The STATECRUNCHER arithmetic and logical operators are commonly known, being mainly compatible with ‘C’.

The tables below define the STATECRUNCHER operators. For their definition in GP4 format, see [StCrParsing].

### 5.6.2.1 Arithmetic operators

The following operators are supported:

<u>Operation</u>	<u>Symbol</u>	<u>Arity</u>	<u>Precedence</u>	<u>Associativity</u>	<u>Position</u>
<b>Primary Suffixes</b>					
Array indexing	[ ]	dyadic	18	none	circumfix
Function call	( )	dyadic	17	none	circumfix
<b>Various monadic</b>					
plus	+	monadic	16	right	prefix
minus	-	monadic	16	right	prefix
logical not	!	monadic	16	right	prefix
post increment	++	monadic	16	left	postfix
post decrement	--	monadic	16	left	postfix
pre increment	++	monadic	16	left	postfix
pre decrement	--	monadic	16	left	postfix

<b>Multiplicative</b>					
multiplication	*	dyadic	15	left	infix
division	/	dyadic	15	left	infix
modulo	%	dyadic	15	left	infix
<b>Additive</b>					
addition	+	dyadic	14	left	infix
subtraction	-	dyadic	14	left	infix
<b>Relational</b>					
less than or equal	<=	dyadic	12	left	infix
greater than or equal	>=	dyadic	12	left	infix
less than	<	dyadic	12	left	infix
greater than	>	dyadic	12	left	infix
equal	==	dyadic	12	left	infix
not equal	!=	dyadic	12	left	infix
<b>Logical</b>					
short-circuit and	&&	dyadic	7	left	infix
xor	^^	dyadic	6	left	infix
equivalence	!^^	dyadic	6	left	infix
short-circuit or		dyadic	5	left	infix
<b>Assignment</b>					
assign	=	dyadic	2	right	infix
multiply-assign	*=	dyadic	2	right	infix
divide-assign	/=	dyadic	2	right	infix
modulo-assign	%=	dyadic	2	right	infix
add-assign	+=	dyadic	2	right	infix
subtract-assign	-=	dyadic	2	right	infix

**Table 6. Arithmetic operators**

Notes:

- The logical operators work with a tri-valued logic, including the value *unknown*.
- The difference between *logical equivalence* (!^^) and *arithmetic equality* (==) is evident from an example with variables a and b, say, with values 1 and 2. The expression a==b is false, but a!^^b is true, since, as in "C", any nonzero value is counted as true.

### 5.6.2.2 Scoping operators

The motivation for scoping operators is that they will be needed when composing models so as to have a model of a system made by composing formal software components. The scoping operators allow local items (events, variables etc) to remain local, but for global ones to be made accessible to many components by renaming them with a scoping expression.

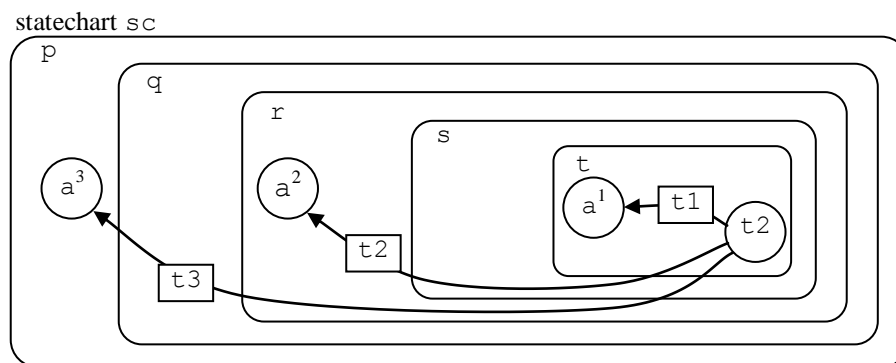
Scoping operators have been introduced summarily (section 5.5), mainly in the context of *declarations*. They are also used to *reference* items (states, PCOs, events, tagnames and variables) in other scopes than the current one, which can be regarded as a default scope. Remember that a scope corresponds to a state in the hierarchy, and that it is represented by a *machine path*. The scope in which an expression is evaluated (and so the default scope, i.e. the scope of a plain identifier) is as follows:

- when referencing PCOs, events, tagnames and variables, it is the machine path of current state.
- when referencing other states, it is the parent of the current state. This gives the most natural representation of states.

The following figure illustrates how scoping operators are used to specify states by referring to their precise position in the hierarchy. The operators in use here are:

- `§` (back out one level and enter state named by right-hand argument)
- `.` (starting from scope of left-hand argument, descend into state named by right-hand argument)

Two examples showing state referencing follow.



**Figure 74. Scoping example - states (1)**

In the above example, there are various states called ‘a’. The superscript serves to distinguish them in this description – it is not part of the name.

How are the targets of the three transitions specified in STATECRUNCHER? They cannot all be specified by

`event -> a`

as that does not distinguish the different targets.

The transitions are specified as part of the `state b` statement. They are specified by:

```
(for t1): event -> a           // references a sibling of state b
(for t2): event -> $$a        // backs out two levels in the hierarchy
(for t3): event -> $$$a      // backs out four levels in the hierarchy
```

Where a target state is not masked by a more local target of the same name, the back-out operator `$` can be omitted. STATECRUNCHER will find the state by an *outbound search* from the precise state specified. So if `t1` and `t2` were not present, `t3` could be specified by just

```
event -> a
```

The target will be found by looking for it in states `t,s,r,q,p` in that order.

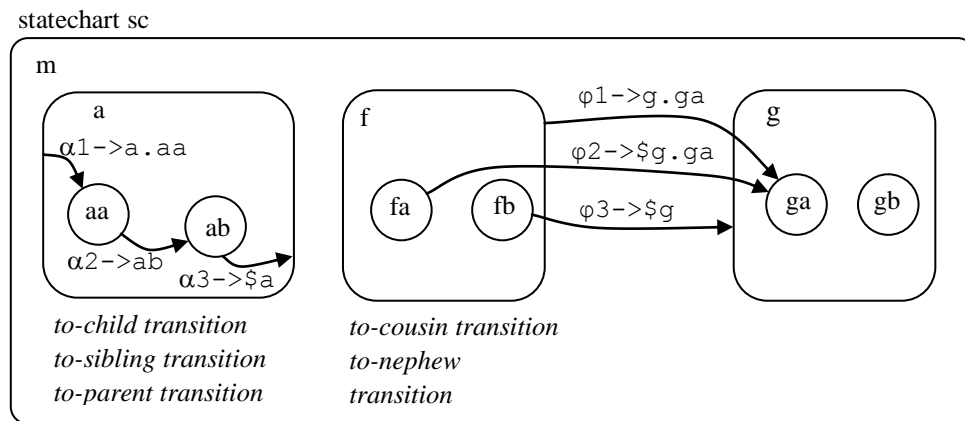
With all three transitions present, transition `t2` could be specified by just

```
event -> $a
```

since that specifies 'a' in the scope of state `s`, and `a2` is the nearest state of that name in state `s`. Similarly transition `t2` could be specified by just

```
event -> $$$a
```

We now show how states in some other common relationships to a transition source state are referenced:



**Figure 75. Scoping example - basic specifications of states (2)**

The statechart level is the outermost named level, and *global* PCOs, events, tagnames and variables are declared in this scope by putting their declarations between the `statechart` statement and the first `state` statement. More *local* PCOs, events, tagnames and variables are declared either by putting their declarations immediately after the `state` statement of the required scope, or by placing the declarations elsewhere, but applying scoping operators to specify their effective scope.

Every state/PCO/event/tagname/variable that is declared is in scope to its descendants, unless a descendant defines a new item of the same name, in which case the *most local* item is in scope by default. Looking at this from the perspective of an item being referenced: the item will be found by an *outbound search*, starting at the current scoping level, and, if the item is not found to have been declared there, backing out one hierarchical level at a time until the item is found. This means that scoping operators are not needed to address the most local name.

When items need to be referenced which are more local than the current scope, scoping operators *must* be used to ‘descend’ into the required scope to address the item.

We now discuss the scoping operators themselves, and then the application of them is reviewed.

### *Design of scoping operators*

There are four scoping operators:

- back-out one level and then evaluate the argument in this scope
- back-out to a named parent and then evaluate the argument in this scope
- back out to the outermost level and then evaluate the argument in this scope
- enter one named level and then evaluate the argument in this scope

These operators are composable into a scoping *expression*, and are compatible with arithmetic operators. This is achieved by an appropriate selection of

- operator symbols
- operator precedence
- operator associativity

The operators are defined as follows:

<u>Operation</u>	<u>Symbol</u>	<u>Arity</u>	<u>Precedence</u>	<u>Associativity</u>	<u>Position</u>
parent scope	§	monadic	19	right	prefix
statechart scope	::	monadic	19	right	prefix
named child scope ( <i>evaluate arg2 in child arg1 scope</i> ).	.	dyadic	20	right	infix
named ancestor scope ( <i>evaluate arg2 in ancestor arg1 scope, backing out one level anyway, and then as far as the first occurrence of arg1</i> ).	%%	dyadic	20	right	infix

**Table 7. Scoping operators**



It is good to realise that there is a major difference in the way scoping operators work compared with arithmetic operators. Arithmetic operators apply their own operation *after* evaluating their arguments (which they do by a recursive call to the evaluator). For example, a simplified<sup>1</sup> PROLOG predicate to evaluate the monadic minus operation on a parameter P1 might be:

```

ev_expr(MPATH, [[ex_monadic, mminus], P1], V) :-
    ev_expr(MPATH, P1, VV),      /* evaluate argument */
    V is -VV,                   /* operator's own action */
    !.

```

P1 is evaluated by a recursive call before the negation takes place (V is -VV).

Similarly for dyadic operations (simplified):

```

ev_expr(MPATH, [[ex_dyadic, dminus], P1, P2], V) :-
    ev_expr(MPATH, P1, VV1),    /* evaluate P1 */
    ev_expr(MPATH, P2, VV2),    /* evaluate P2 */
    V is VV1-VV2,              /* operator's own action */
    !.

```

In these predicates, MPATH is the machine path (i.e. scope) in which the evaluation takes place. Termination of the recursion takes place at a terminal item, such as an identifier (whose value is then obtained from a 'database').

Now when it comes to scoping operators, they must perform their own operation – i.e. changing the scope – *before* evaluating their arguments. It will be seen that this has implications for the choice of precedence and associativity. Here is what the back-out operator does:

```

ev_expr([HMPATH|TMPATH], [[ex_monadic, mback], P1], V) :-
    ev_expr(TMPATH, P1, V),     /* remove head of machine path */
    !.

```

The predicate first modifies the supplied machine path. It effectively removes the head of a list describing the machine path [HMPATH|TMPATH], the head HMPATH being the most local part of the path. Then it performs the recursive call to have its parameter, P1, evaluated in the new scope.

---

<sup>1</sup> Various factors ignored here: error conditions, details of type and wrapping of data, and overloading of the operator (i.e. different actions on different types of data).

Similarly for a dyadic scoping operator. The following operator evaluates its first argument (P1), as a required addition to the machine path, so as to make the scope more local. The second argument (P2) is then evaluated in the new scope.

```

ev_com_expr(MPATH, [[ex_dyadic, descend], P1, P2], V) :-
    ev_com_expr(MPATH, P1, V1),
    V1=[ID, _],
    MPATH2=[ID|MPATH],
    ev_com_expr(MPATH2, P2, V),
    !.

```

The "." (descend) and "%%" (dparent) operators are right associative. This means that an expression such as

aa.bb.cc.dd

is equivalent to

aa.(bb.(cc.dd))

At first sight, this might seem wrong. It appears that the term (cc.dd) will act first and add element cc to the machine path first, whereas we want to add element aa to the machine path first. But bearing in mind the reasoning about scoping operators performing their operation *before* evaluating their arguments, the above expression will add element cc to the machine path *last*, and behave as follows:

- add aa to the machine path, making it one level deeper than the caller's level
- add bb to the machine path, making it one level deeper than as above
- add cc to the machine path, making it one level deeper still
- evaluate dd in this new scope

Similarly

aa%%bb%%cc%%dd

will evaluate dd in the scope that backs out to the first occurrence of aa (cutting blindly through bb's and cc's if they occur), then backs out further to the next occurrence of bb (cutting blindly through cc's if they occur), then backs out further to the first occurrence of cc, and finally evaluates dd in this scope.

Similarly, the "::" (mscope) and monadic "\$" (mback) operators are right associative. This means that expressions consisting of multiple monadic operators can be composed simply:

\$\$\$aa

which is equivalent to

\$((\$aa))

backs out three levels then evaluates aa.

The expression

::\$aa

backs out to the outermost shell, then backs out one more level, which in STATECRUNCHER is admissible, as the ":" operator backs out to the `statechart` level, from which it is possible to back out once more to the absolute level.

The expression

```
$ : aa
```

would normally be pointless, as it backs out one level before performing a global back-out operation.

These monadic and dyadic operators combine with dyadic operations binding tighter, so that

```
$$aa.bb.cc
```

which is equivalent to

```
$( $(aa.(bb.cc)) )
```

means back out two levels, then enter `aa` then enter `bb` then enter `cc`. The rule is emerging that **the expression is to be interpreted as a sequence of actions in left-to-right reading order**.

One consideration is that dyadic operators have a higher precedence than monadic ones, which is fine for expressions such as

```
$$aa.bb.cc
```

but it means that brackets are needed for *adjacent dyadic-monadic accumulations*, e.g.

```
cc%%($$dd.var2)
```

which is to be read as: back-out to parent `cc`, then back out twice more, then descend into `dd`, then evaluate `var2` in this scope.

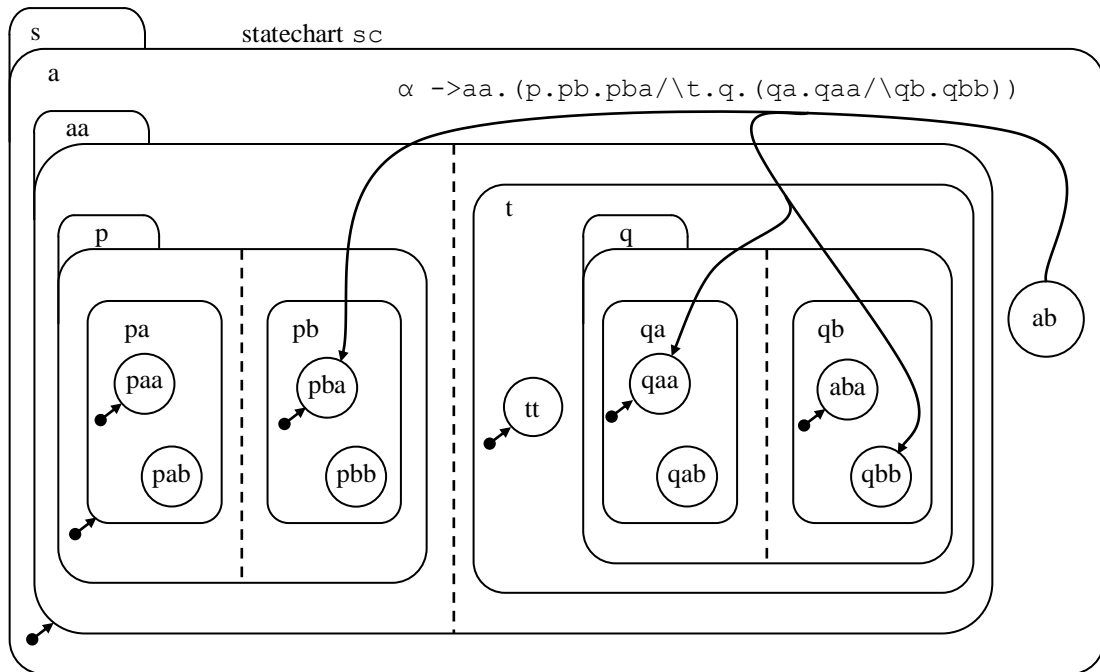
Scoping operators have a higher precedence than non-scoping ones. An example of a combined expression, extending the above example, is:

```
var1 + cc%%($$dd.var2)
```

which is to be read as: evaluate `var1`, back-out to parent `cc`, then back out twice more, then descend into `dd`, then evaluate `var2` in this new scope, then finally add together with the evaluation of `var1`.

### 5.6.2.3 The split operator

This operator is used to define multiple target states of transitions. STATECRUNCHER allows transitions to specify targets in more than one member of a set. This can take place at various hierarchical levels, so requiring a *target state tree*. This is illustrated in the figure below.



**Figure 76. Multiple target states**

Note that the target state tree need not specify all targets in a set – defaults (or historical states) will be taken where no specific target is specified.

The target state tree is specified using the *split* operator denoting "and co-member", represented above by the symbol  $\wedge$ . The operator is available to target state expressions but is not available in other state expressions.

The operator is specified (in the same notation as used for scoping operators) as follows

<u>Operation</u>	<u>Symbol</u>	<u>Arity</u>	<u>Precedence</u>	<u>Associativity</u>	<u>Position</u>
split	$\wedge$	dyadic	14	left	infix

**Table 8. Split operator**

This gives a lower binding precedence than the scoping operators ( $::$   $\% \%$   $\$$   $.$ ). It is a left associative operator, (such as the  $+$  operator), so that

$$a \wedge b \wedge c \wedge d = ((a \wedge b) \wedge c) \wedge d.$$

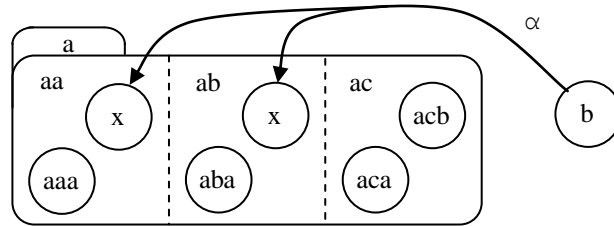
***A restriction***

The left hand side of the  $.$  and  $\% \%$  operators should not be a term which has already been split, (although such a thing does make sense), since such a construction is unusual and the evaluator does not currently support it. So, in the figure below, it would not be permissible to write

$\alpha \rightarrow a . ( (aa / \backslash ab) . x )$

Instead, the following should be used:

$\alpha \rightarrow a . ( aa . x / \backslash ab . x )$



**Figure 77. Restriction in use of the *split* operator**

### *Evaluation of the split operator*

The *evaluator* for terms combined with this operator produces a list of lists representing the target tree. Expressions are evaluated in an *evaluation scope* representing a state in hierarchy. Typical evaluations are as follows:

	<b>Evaluation Scope</b>	<b>Expression</b>	<b>Evaluation</b>
1	[bb, aa]	dd / \ ee	[[dd, bb, aa], [ee, bb, aa]]
2	[bb, aa]	pp . dd / \ ee	[[dd, pp, bb, aa], [ee, bb, aa]]
3	[bb, aa]	(pp . dd) / \ ee	[[dd, pp, bb, aa], [ee, bb, aa]]
4	[bb, aa]	pp . (dd / \ ee)	[[dd, pp, bb, aa], [ee, pp, bb, aa]]
5	[bb, aa]	pp . (dd / \ (ee / \ ff . gg))	[[dd, pp, bb, aa], [ee, pp, bb, aa], [gg, ff, pp, bb, aa]]
6	[bb, aa]	pp . ((dd / \ ee) / \ ff . gg)	[[dd, pp, bb, aa], [ee, pp, bb, aa], [gg, ff, pp, bb, aa]]
7	[bb, aa]	pp . ((dd / \ \$ee) / \ ff . gg)	[[dd, pp, bb, aa], [ee, bb, aa], [gg, ff, pp, bb, aa]]
8	[bb, aa]	pp . ((dd / \ ee) / \ (ff . f2 / \ gg . hh))	[[dd, pp, bb, aa], [ee, pp, bb, aa], [f2, ff, pp, bb, aa], [hh, gg, pp, bb, aa]]
9	[bb, aa]	pp . (dd / \ ee) / \ (ff . f2 / \ gg . hh)	[[dd, pp, bb, aa], [ee, pp, bb, aa], [f2, ff, bb, aa], [hh, gg, bb, aa]]
10	[cc, bb, aa]	\$\$pp . (dd . ee . ff / \ \$gg . hh . ii)	[[ff, ee, dd, pp, aa], [ii, hh, gg, aa]]
11	[cc, bb, aa, sc]	::pp . (dd . ee . ff / \ \$gg . hh . ii)	[[ff, ee, dd, pp, sc], [ii, hh, gg, sc]]
12	[cc, bb, aa, sc]	::\$pp / \ \$\$dd	[[pp], [dd, aa, sc]]

13	[cc,bb,x1,x2,aa,sc]	aa%%pp/\\$dd	[[pp,aa,sc],[ddx1,x2,aa,sc]]
14	[cc,bb,aa]	(pp/\qq).rr // violates the restriction mentioned above.	unknown

**Table 9. Evaluation of the split operator**

The target of transition  $\alpha$  in Figure 76 is represented by

aa.(p.pb.pba/\t.q.(qa.qaa/\qb.qbb))

in evaluation scope

[a,s,sc]

evaluating to

[[pba,pb,p,aa,a,s,sc],  
[qaa,qa,q,t,aa,a,s,sc],  
[qbb,qb,q,t,aa,a,s,sc]]

### 5.6.3 Functions

#### 5.6.3.1 Arithmetic functions

Arguments are a comma-separated list of expressions. P1, P2 refer to the first and second parameter respectively. The return value is an integer (which may represent a boolean), or string value. The value may be ignored. The functions are as follows:

<b>Basic arithmetic</b>	
abs(P1)	absolute value of a number
maximum(list)	maximum of several numbers, e.g. i=maximum(v1,v2+1,v3)
minimum(list)	minimum of several numbers, e.g. i=minimum(v1,v2+1,v3)
<b>String related</b>	
format(P1,P2)	Format integer expression P1 as text. P2 is the field width: -ve for left justify, 0 for just fit, +ve for right justify.
length(P1)	length of string
lower_case(P1)	convert string to lower case
upper_case(P1)	convert string to upper case
<b>Casting</b>	
cast(P1)	i=cast(j) allows an assignment that would otherwise be a type mismatch
<b>Tracing</b>	
trace(list)	add parameter(s) to the trace list
trace_clear()	clear the trace list
<b>System information</b>	

get_nworlds(P1)	get_nworlds() or get_nworlds(1) gets the number of worlds at the start of event processing. get_nworlds(2) gets the dynamic number of worlds.
<b><i>Nondeterminism control</i></b>	
no_race()	turn race nondeterminism off
low_race()	allows only two race permutations, forwards and backwards.
med_race()	allows 2N race permutations. Allows distinction of all triplet orderings
high_race()	allows all N! race permutations
<b><i>Special functions taking a state-expression argument</i></b>	
in(P1)	returns true (=1) if the state specified is occupied, else false (=0)
clear(P1)	clear history of the state specified
deep_clear(P1)	clear history of the state specified and its descendants

**Table 10. Functions**

### 5.6.3.2 Special functions

The evaluation of most functions proceeds as follows:

- evaluate the arguments (which can contain arithmetic and scoping operators) as values
- pass the evaluated parameters to the function
- return a value from the function

Certain functions are exceptions to this in that their parameters are evaluated to a *name*. These functions are described in this section.

#### ***in***

The function

```
in(state-expression)
```

returns a boolean value: `true` if the specified state is occupied, `false` if it is not.

#### ***clear and deep\_clear***

The function

```
clear(state-expression)
```

removes history *data* from the specified state.

The function

```
deep_clear(state-expression)
```

removes history *data* from the specified state and all its child states recursively down the hierarchy.

### ***trace***

The function

```
trace(expression)
```

writes the evaluation of its argument to a special location called the *trace list*. Traces model black-box outputs of the Implementation Under Test. The trace list, along with state occupancies, variable values and other information, is provided by STATECRUNCHER after processing an event.

## **5.6.4 Type compatibility in expressions**

A rigorously typed language would require exact type matching of terms in expressions, and in left and right hand sides of assignments. It is felt that in STATECRUNCHER more freedom should be allowed: certainly, a range-type variable should be compatible with raw integers.

Note that there is a type incompatibility if two types have the same name but due to scoping considerations they refer to type definitions at different scoping levels.

Example

```
$$$colour $mycolour = $yourcolour;
```

There are two references to a type definition named `colour`.

1. the one found by an outward search starting from `$$$<current machine path>`
2. the one found by an outward search starting from `$<current machine path>`, to find the definition of `yourcolour`, and the scope of its *declared* type, followed by another outward search to find the scope of its *actual* type.

If these yield the same definition, the expression is type compatible, otherwise it is not.

In the current version of STATECRUNCHER, raw integers are compatible with *all* enum types.

## **5.6.5 Type compatibility in functions**

STATECRUNCHER supports functions according to the GP4 implementation paradigm. For simplicity in the current version (1.05) of STATECRUNCHER, functions are typeless. All functions accept any type in their parameters and the return parameter will match any type. This means that an identity function could act as a cast – such a function exists, and it is called `cast`.



## 5.7 Review of items parsed as expressions

Items (states/PCOs/events/tagnames/variables) in STATECRUNCHER occur once in their **declaration**, and any number of times when **used**, (i.e. when referenced, whether read-accessed or write-accessed).

As can be seen from the syntax diagrams, the following items are scoped expressions:

- States in **usage** (State scope on “declaration” is determined by the statement position in the machine hierarchy)
- PCOs in **declaration / usage**
- Events in **declaration / usage**
- Tagnames in **declaration** (enum statement) / **usage** (variable declaration)
- Variables in **declaration / usage** (e.g. initialisation, condition, action, label)

This means that there is opportunity to access, and even declare, items in a scope other than the current scope, whether more globally, more locally or in a different relation to the current scope.

States, PCOs, events, tagnames, variables defined in a more global scope than the current scope are implicitly in scope, unless masked by a more local homonym.

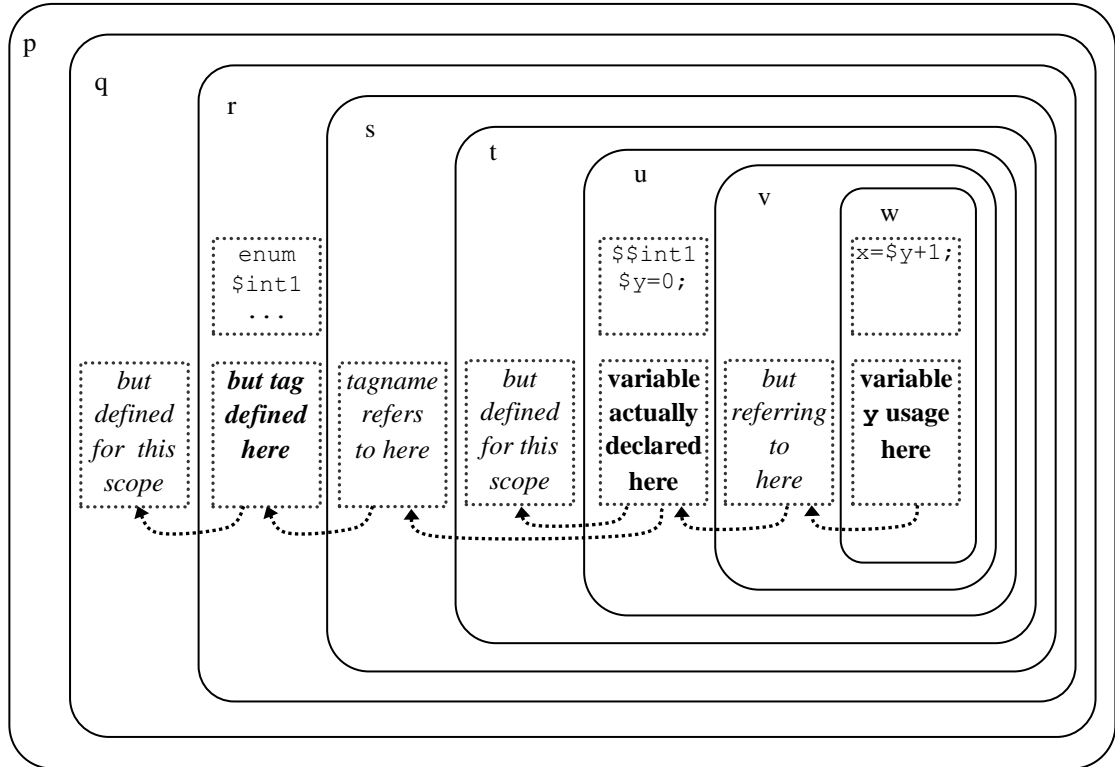
It is recommended that non-local scoping should be used sparingly, especially non-local declarations. In any case exceptional scoping should not be used gratuitously (for readability reasons), but only when composition of subsystem models requires it.

However, in compositions of components, scoping operators should be used. A useful construction is to define a wrapper *set* for the composition (called, say, `Composition`) with set members for the comprising components. An individual component model declares its own inter-component events *inside* the confines of its source code as regards where the statement is positioned, but *outside* its confines as regards its effective scope, specifying `Composition` scope e.g. as follows:

```
event Composition%%ReturnDropRequestAccepted;
```

The following (rather concocted) example shows the potential complexity of scoping operators and the *outbound search* mechanism to find the nearest variable and its type in scope.

statechart sc

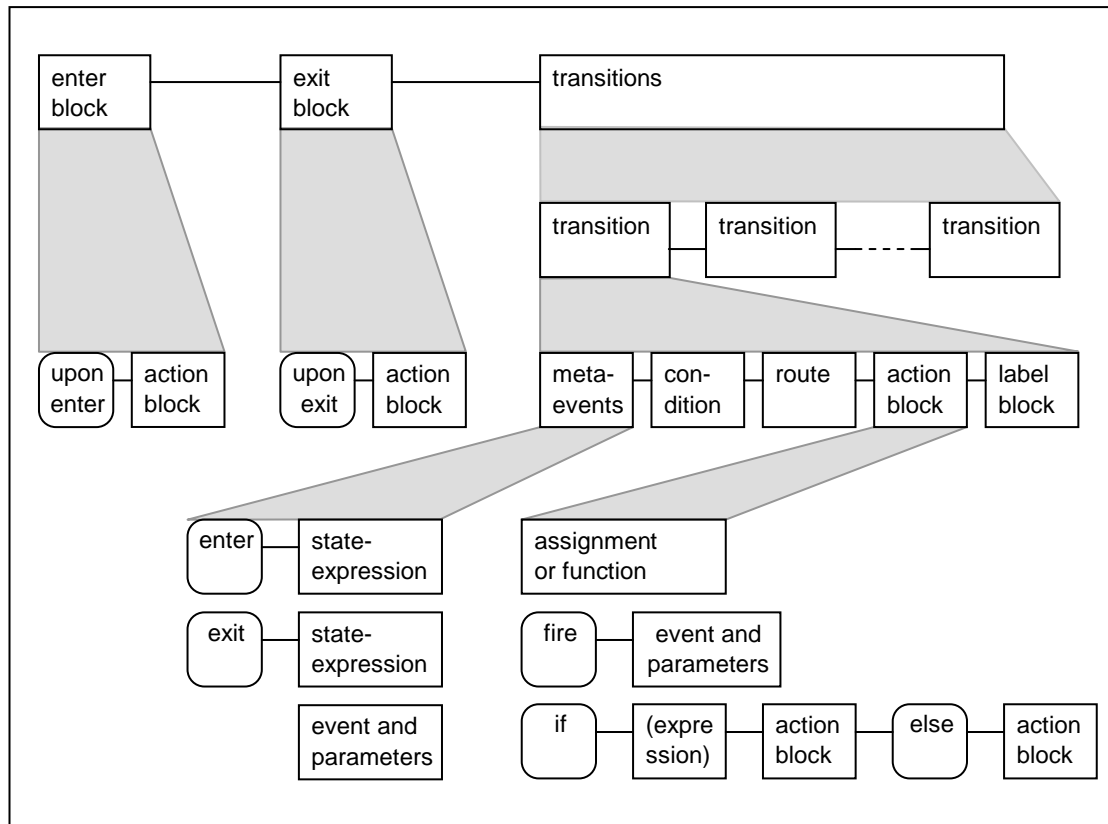


**Figure 78. Complex tagname/variable scoping**

## 5.8 Transition block

Transition blocks are part of `state` statements.

### 5.8.1 Transition block overview



**Figure 79. Overview of transition block**

## 5.8.2 Transition block syntax

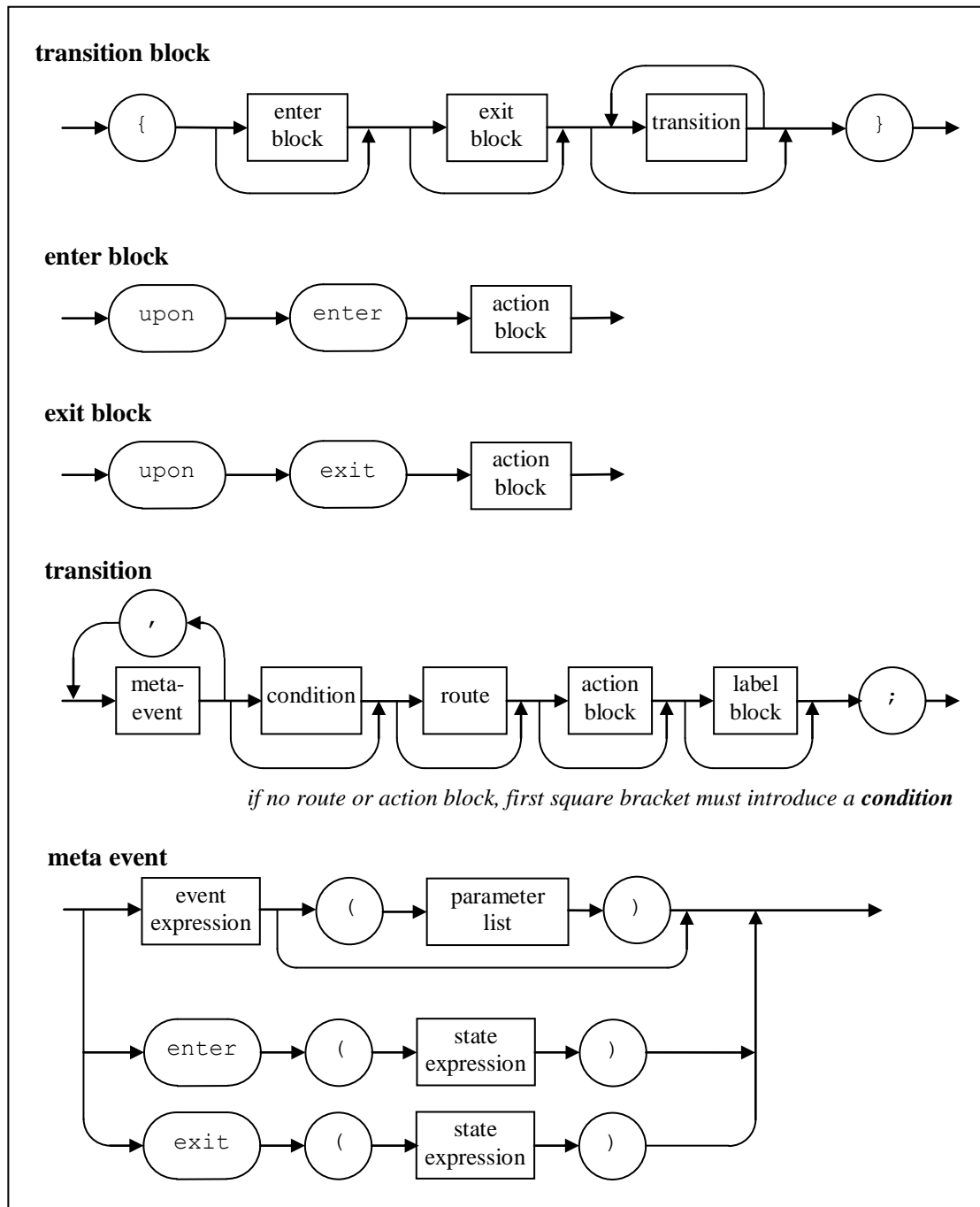


Figure 80. Transition block syntax (1)

Transition block syntax continued:

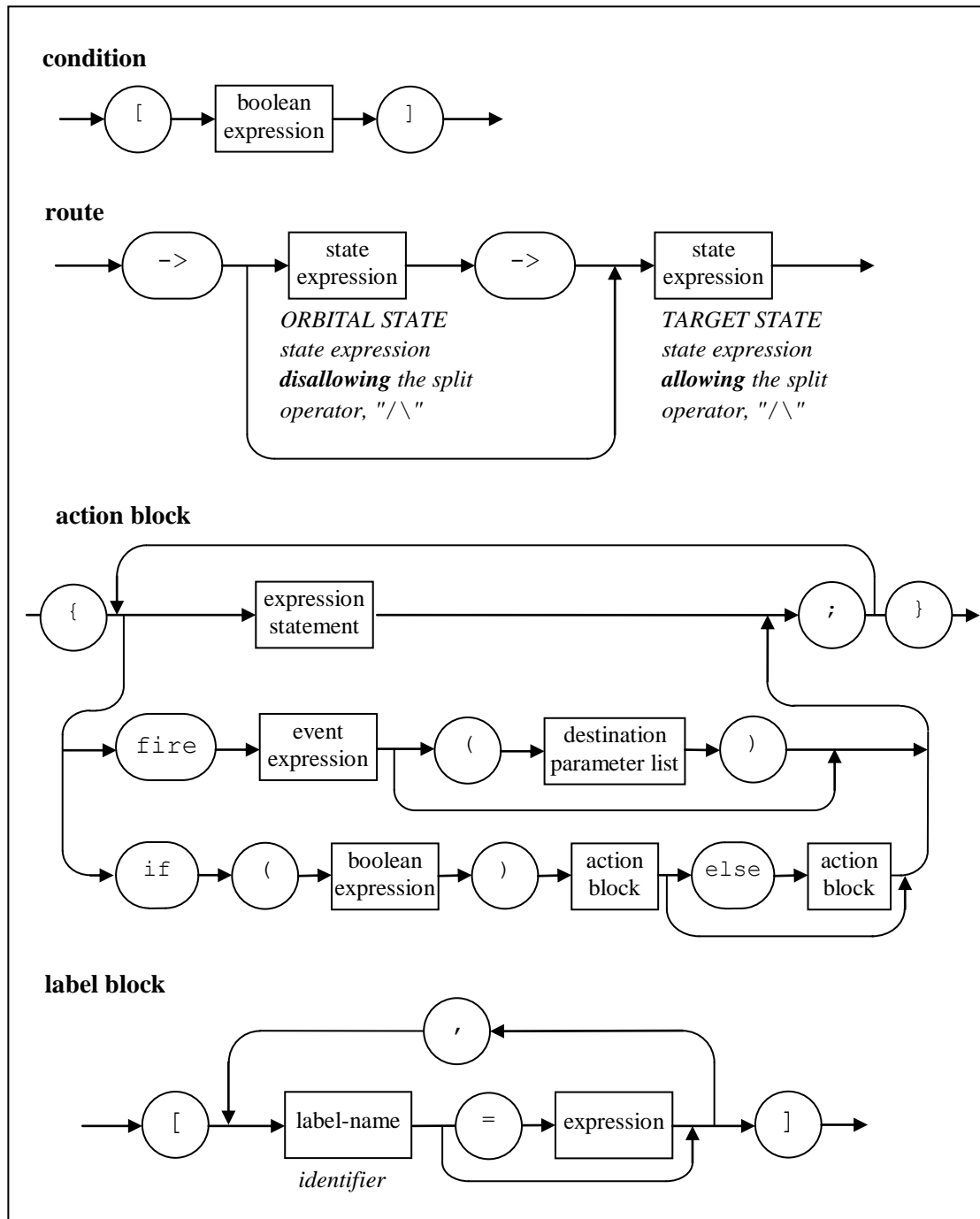


Figure 81. Transition block syntax (2)

### 5.8.3 Detailed examples of transition block functionality

#### *Remark*

In the state diagrams that follow, for compactness the transition labelling may not be the full STATECRUNCHER syntax. We may exclude braces, destination states, and semicolons. So we may have, e.g.  $\beta 1 \{ \$v1 += 2 \}$  rather than  $\{ \beta 1 \rightarrow bb \{ \$v1 += 2 ; ; \} \}$ . To compensate for this, we provide the full model source code of some examples in this section.

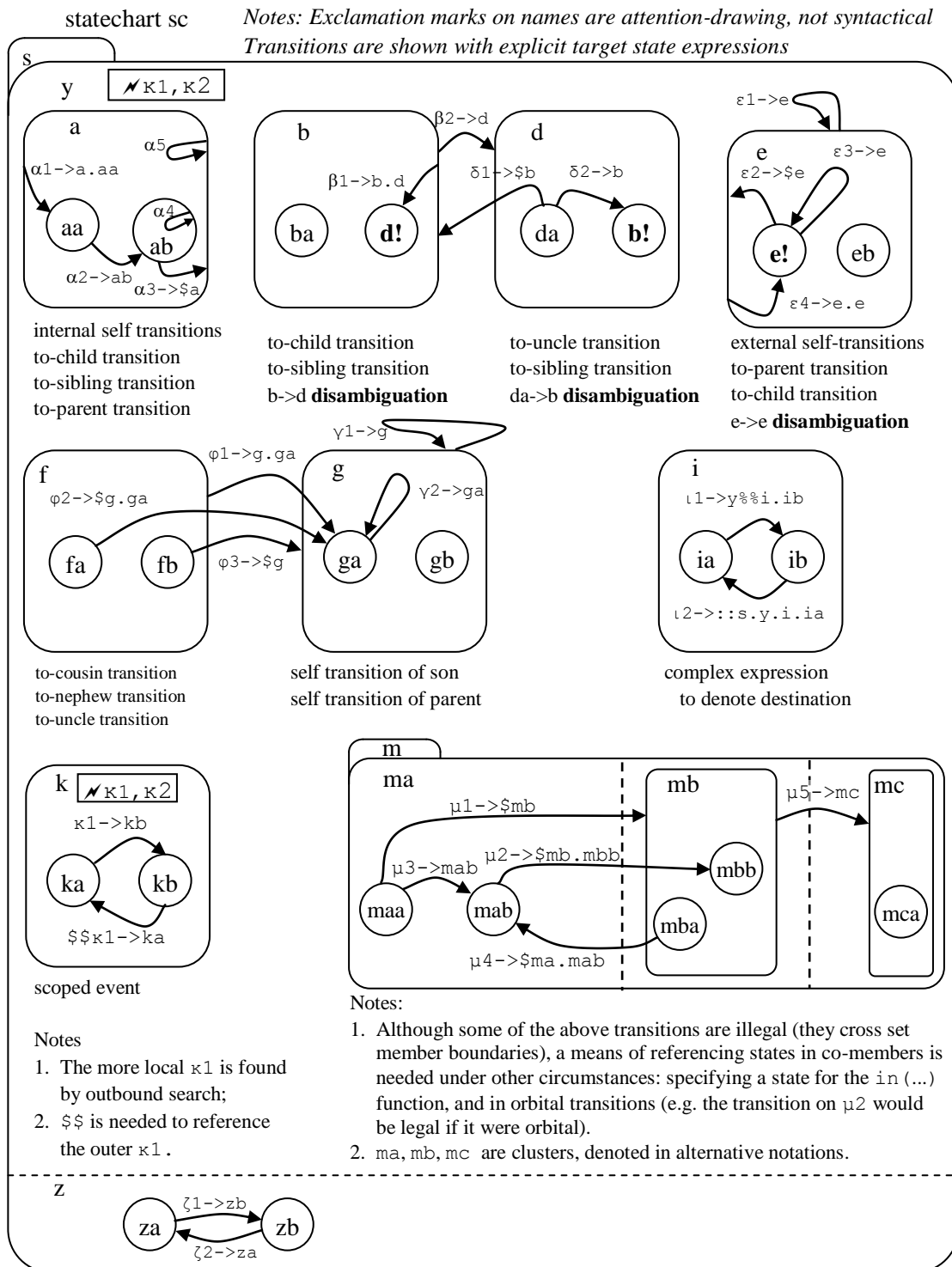
#### 5.8.3.1 Specification of states (as transition targets) - further examples

##### *Reminder*

The scope in which an expression is evaluated is as follows:

- when referencing PCOs, events, tagnames and variables, it is the machine path of current state.
- when referencing other states, it is the parent of the current state. This gives the most natural representation of states.

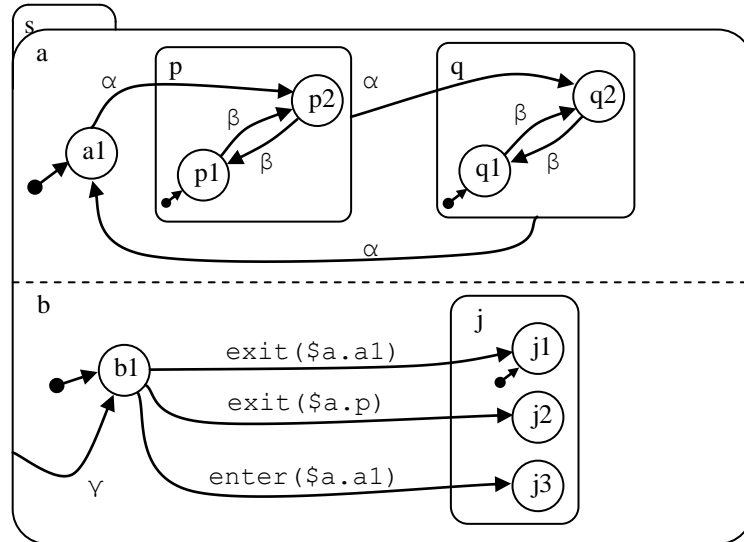
The following figure shows some common examples of transitions. Self-transitions are explained later in this section.



**Figure 82. Specification of states**

### 5.8.3.2 A model illustrating internal events

Internal events were introduced in Figure 13. Meta events include ordinary events and internal events. In the figure below, the transitions on  $\alpha$  cause various states (leafstates and hierarchical states) to be exited / entered. Some of the corresponding enter and exit meta events are used to trigger transitions in a parallel part of the statechart, in cluster b.



**Figure 83. Meta event (state entry/exit) [model u5180]**

Source code of the model

```
statechart sc(s)
event alpha,beta,gamma;
set s(a,b)
  cluster a(a1,p,q)
    state a1          {alpha->p.p2;}

    cluster p(p1,p2)  {alpha->q.q2;}
      state p1 {beta->p2;}
      state p2 {beta->p1;}

    cluster q(q1,q2)  {alpha->a1;}
      state q1 {beta->q2;}
      state q2 {beta->q1;}

  cluster b(b1,j) {gamma->b.b1;}
    state b1      {exit ($a.a1)-> j.j1; \
                  exit ($a.p) -> j.j2; \
                  enter ($a.a1)-> j.j3; }

  cluster j(j1,j2,j3)
    state j1;
    state j2;
    state j3;
```



### 5.8.3.3 Conditional transitions and conditional actions

In Figure 15 we saw a conditional transition, and in Figure 18 a conditional action. A complete model illustrating some detail of this is given below. An action (conditional or otherwise) can be triggered by an event without transitioning between states by using an internal transition, such as the one on event `setv` in the diagram below (to be discussed in more detail later).

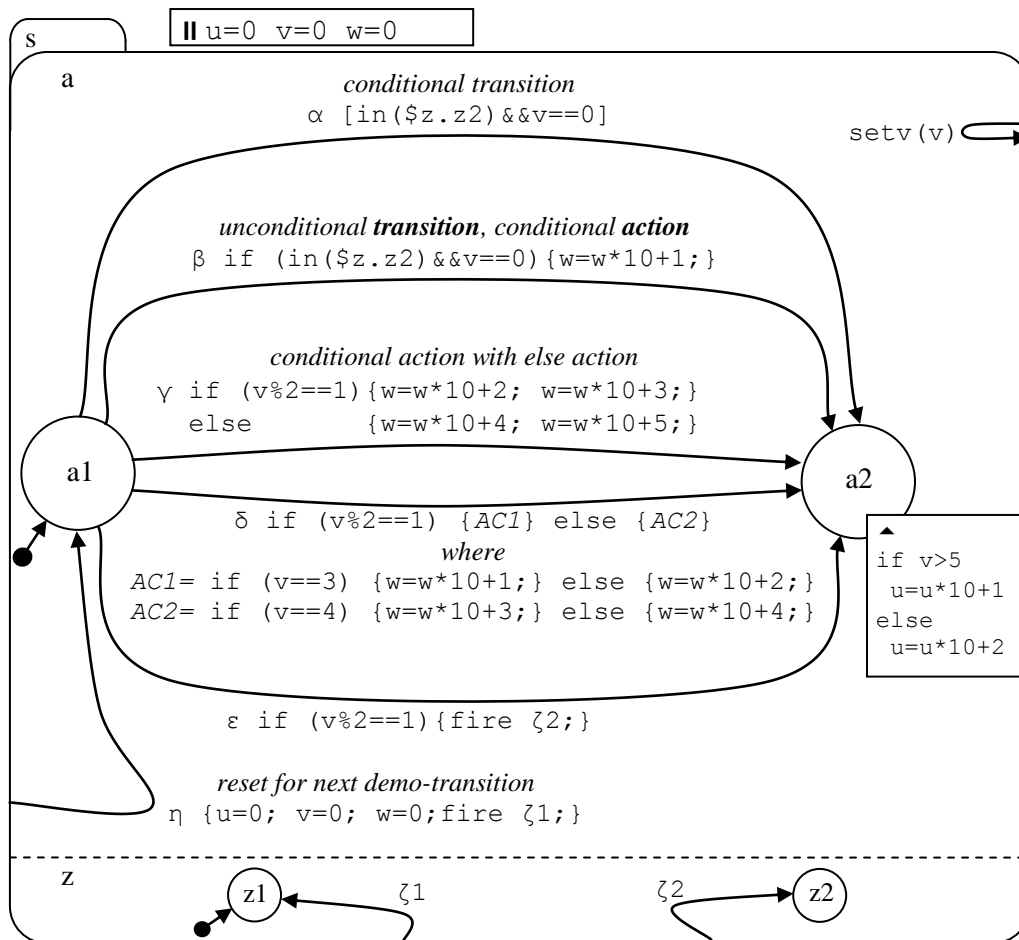


Figure 84. Conditional transitions and actions, and the `in()` function [model u5190]

#### Points to note

- There is a conditional transition on  $\alpha$ .
- There is a conditional action on the transition on  $\beta$ , and also on entering state `a2`.
- The transition on  $\gamma$  has an *else* part.
- The transition on  $\delta$  has nested conditional actions.
- The conditional action of the transition on  $\epsilon$  fires an event, putting cluster `z` in state `z2`.
- We can set the value of `v` (used in the conditions) using the `setv` event.
- We can reset variables and states using the  $\eta$  event.

Source code of the above model:

```

statechart sc(s)
event alpha,beta,gamma,delta,epsilon,eta;
event setv;
event zeta1,zeta2;

enum int1 {0,..,10000};
int1 u=0,v=0,w=0;

set s(a,z)
  cluster a(a1,a2) {setv(v); eta->a.a1 {u=v=w=0; fire zeta1;}; }

  state a1
    {alpha [in($z.xxx.z2) && (v==0)]->a2;
    beta-> a2 {if (in($z.z2) && (v==0)) {w=w*10+1;}; }
    gamma-> a2 {if (v%2==1) {w=w*10+2;w=w*10+3;}
               else {w=w*10+4;w=w*10+5;}; }
    delta-> a2 {if (v%2==1)
                {if (v==3) {w=w*10+1;} else {w=w*10+2;}}
               else
                {if (v==4) {w=w*10+3;} else {w=w*10+4;}} };
    epsilon->a2 {if (v%2==1) {fire zeta2;}};

  state a2 {upon enter { if(v>5) {u=u*10+1;} else {u=u*10+2;}} }

  cluster z(z1,z2) {zeta2->z.z2; zeta1->z.z1;}
  state z1;
  state z2;

```

### 5.8.3.4 Route; orbit; internal and external self-transitions

The transition *route* describes the target state(s) of the transition, and also which states must be exited and entered en-route. The highest state in the route is called the *orbit*. The orbit is optional – if omitted, no more states than necessary will be exited and entered en-route. The whole route is also optional – if omitted, the transition is an *internal self-transition*. *External self transitions* are transitions with the same source and target state. They may nevertheless cause a transition between states. We illustrate these things in the next figure.

*Internal self-transitions* are drawn on the inside of the state and never cause transitions between states. As with other transitions, they are valid for processing if the state to which they are attached is occupied; if not, they are totally discounted.

- There is no difference between leafstate and non-leafstate internal self-transitions. If they are valid and there is an action attached to them, the action is performed (see transitions on  $\zeta_1$  and  $\varepsilon_1$  below).
- Internal transitions cannot be orbital (the transition on  $\zeta_2$  is unspecifiable).

**External self-transitions** are drawn outside the state.

- If they are on a nonleaf state, they can cause transitions to default states, (but not in clusters with history, because the current state is counted as the historical state). This applies to the self-transition on  $\varepsilon_3$  when state  $p_2$  is occupied below.
- If they are on a leafstate, nothing is exited or entered (unless the self-transition is orbital), but actions are executed, and they behave like internal transitions (see transitions on  $\zeta_1$  and  $\zeta_3$ ).
- External self transitions can be orbital (to any height of orbit). In this case they always cause exiting and entering to the height of the orbit (transitions on  $\zeta_4$  and  $\varepsilon_4$ ).
- How is the transition on  $\varepsilon_2$  to be interpreted? As an internal orbital transition it is undefined and unspecifiable in STATECRUNCHER. It can, however, be regarded as an external transition, a shorthand for what might otherwise be drawn as the transition on  $\varepsilon_5$ . This is specifiable in STATECRUNCHER and the meaning is to exit from whatever deeper states are occupied as far as the orbit, and to re-enter states according to the transition course algorithm as described in section 7.5.

Internal orbital self-transitions (as on  $\zeta_2$ , and as on  $\varepsilon_2$  if it were to be regarded as internal) are currently unspecifiable. However, they could be given a syntax such as

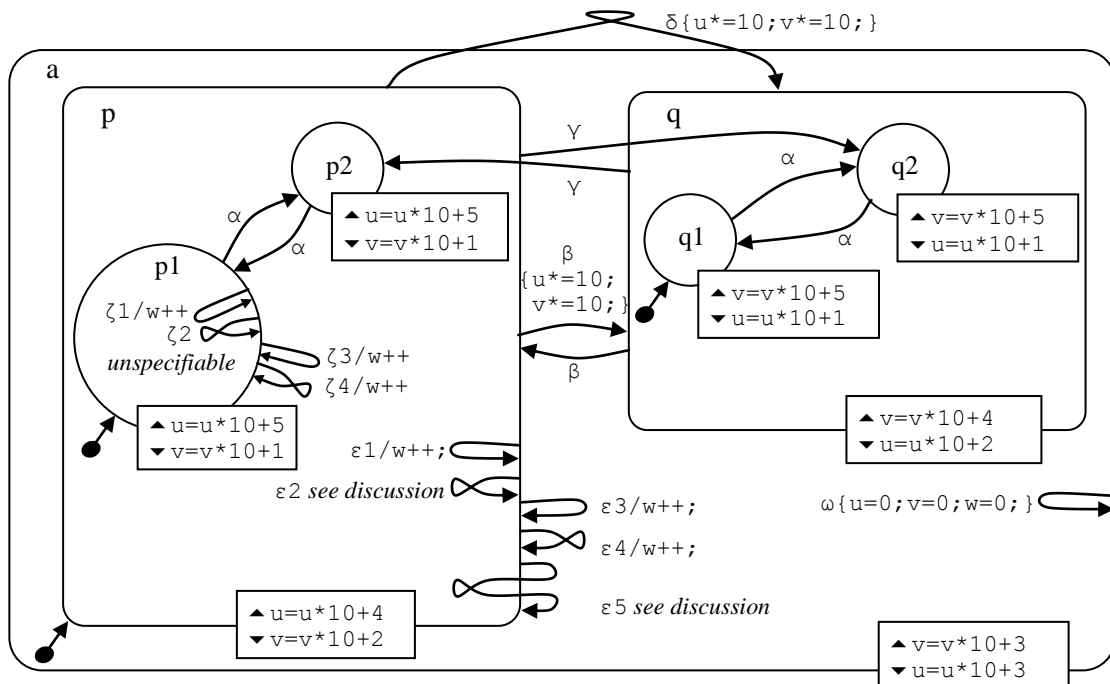
```

ε2 ->@shallow_internal
ε2 ->@deep_internal

```

and some semantics: execute the exit and entry actions on the current member state, either at the current hierarchical level only, or at all occupied states in the hierarchy.

Self transitions can be parameterized, but we do not illustrate that in our example below.



**Figure 85. Orbits and self-transitions, [model u5170b]**

## Source of this model

```

statechart sc(a)

event alpha,beta,gamma,delta;
event epsilon1,epsilon2,epsilon3,epsilon4;
event zeta1,zeta3,zeta4;
event omega;

enum int {0,..,10000};
int u=0,v=0,w=0;

cluster a(p,q)      {upon enter{u=u*10+3;}      upon exit{v=v*10+3;} \
                    omega{u=0;v=0;w=0;};      }

cluster p(p1,p2) {upon enter{u=u*10+4;}      upon exit{v=v*10+2;} \
                    delta->$$sc->q{u*=10;v*=10;}; \
                    beta->q{u*=10;v*=10;};      gamma->q.q2; \
                    epsilon1{w++;};            epsilon2->p->p{w++;}; \
                    epsilon3->p{w++;};            epsilon4->$a->p{w++;};}

state p1           {upon enter{u=u*10+5;}      upon exit{v=v*10+1;} \
                    zeta1{w++;};                zeta3->p1{w++;}; \
                    zeta4->$p->p1{w++;};        alpha->p2;      }

state p2           {upon enter{u=u*10+5;}      upon exit{v=v*10+1;} \
                    alpha->p1;                    }

cluster q(q1,q2) {upon enter{v=v*10+4;}      upon exit{u=u*10+2;} \
                    beta->p;                        gamma->p.p2;      }

state q1           {upon enter{v=v*10+5;}      upon exit{u=u*10+1;} \
                    alpha->q2;                    }

state q2           {upon enter{v=v*10+5;}      upon exit{u=u*10+1;} \
                    alpha->q1;                    }

```

### Points to note

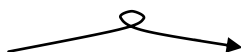
- Variable  $v$  tracks a transition from  $p$  to  $q$ . Variable  $u$  tracks a transition from  $q$  to  $p$ . The *on-transition actions* simply add digit 0 to  $u$  and  $v$  by multiplying by 10. This gives us a complete record of the order of the actions that take place during a transition. The variables can be reset without any transitioning by executing event  $\omega$ .
- If there are *upon enter* actions and *upon exit* actions, the *upon enter* actions must be specified first.
- An example of orbital notation is  $\text{delta} \rightarrow \$\$sc \rightarrow q$ . More detail is given later in this section.

So far, we have been precise about the orbital state. Where states have unique names, the operators can be omitted and the correct state will be found by the outbound search for the nearest state in scope. So we can also specify the example as simply `delta->sc->q`.

### ***More on orbital transitions***

The feature of orbital transitions is that they exit and enter superstates up to a higher level than a direct (non-orbital) transition. In so doing they generate additional enter and exit meta-events, and can cause re-entered states with no history to revert to default occupancies.

We draw orbital transitions with a loop in the orbital state of the transition arc:



The transition notation specifies this with an extra arrow:

```
event -> orbital_state -> target_state
```

Note that an orbital transition is *not* achieved by specifying the target state in any particular way: a transition on event  $\alpha 1$  in Figure 86 below might be specified as any of the following:

```
 $\alpha 1$  -> aab
 $\alpha 1$  -> $aa.aab
 $\alpha 1$  -> $$a.aa.aab
 $\alpha 1$  -> ::s.y.a.aa.aab
```

It is a *state*, not an *operator sequence* (such as \$\$), that is specified as the *orbital state*. The evaluation scope for the expressions for the orbital state and target state is (as for target state expressions) that of the *parent of the source state*.

Referring to Figure 86, note that it is possible to have an orbital from-superstate transition (transition on event  $\varepsilon 1$ ).

It is possible to define orbital states that make little or no sense:

- because they are lower in the hierarchy than the highest point of the equivalent non-orbital transition.
- because they specify a state that is not an ancestor of source or target.

Such orbital data is ignored by STATECRUNCHER.

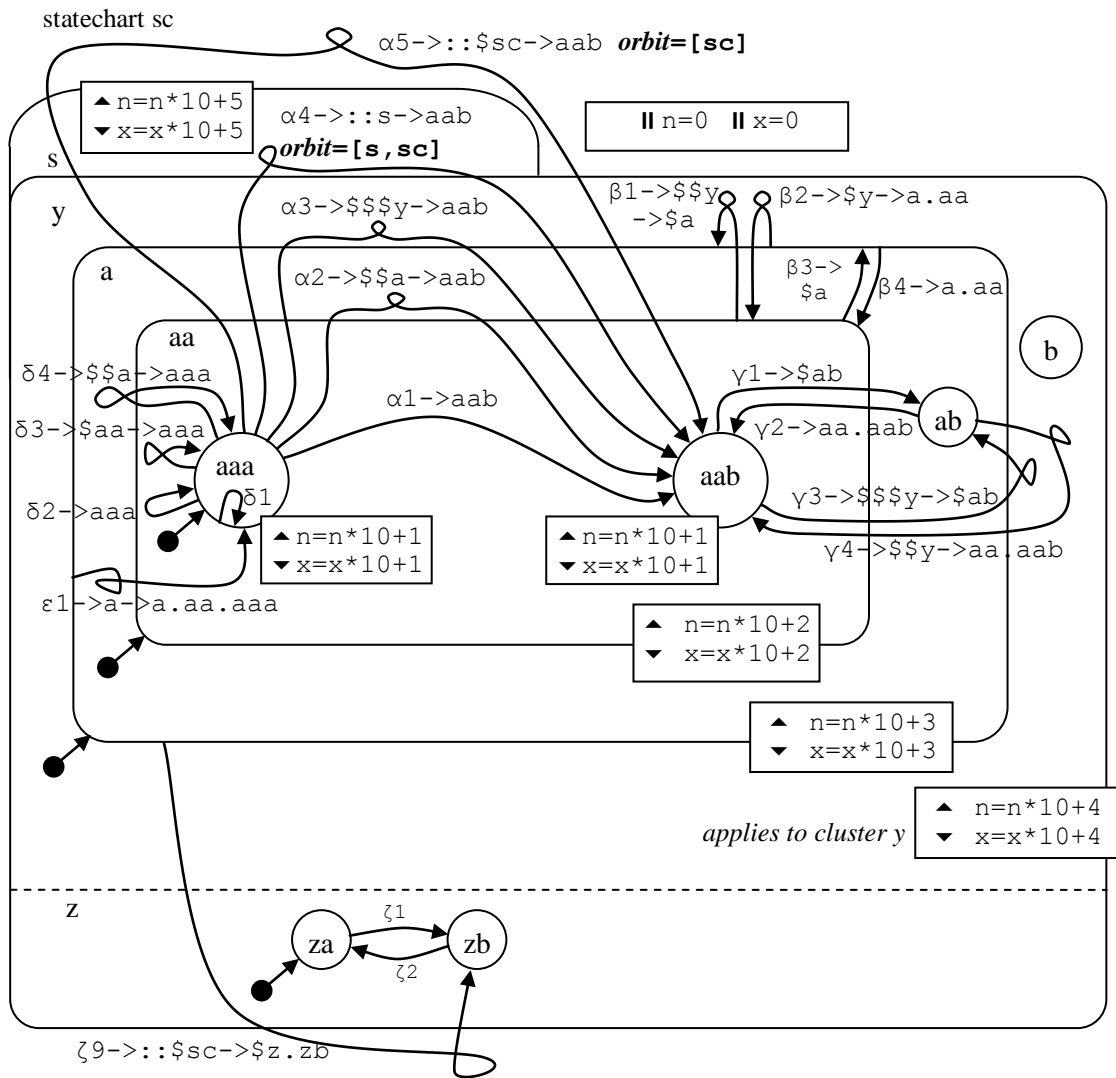


Figure 86. Orbital transitions [model t6260]

Useful rules on orbital states

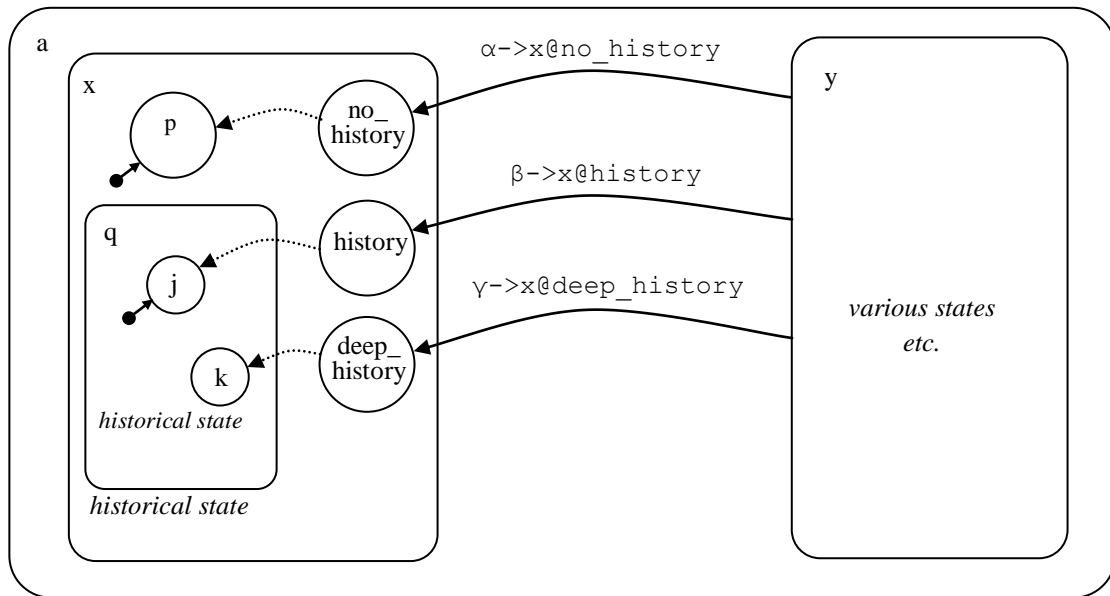
- If the transition arc to an *orbital* state crosses n hierarchical layers, use (n+1) \$ characters in specifying it.
- If the transition arc to a *target* state crosses n hierarchical layers, use (n) \$ characters in specifying it.
- The hierarchical layers can be counted by counting the number of boxes crossed (but *not* set member boundaries, i.e. the dotted line). Note, however, that a cluster member of a set can be specified without drawing a box round it, so when counting boxes exited, allow for an ‘invisible’ box in this case.

**Notes**

- Two examples of *evaluated orbits* are shown, for the transitions on  $\alpha$ 4 and  $\alpha$ 5. Evaluated orbits are machine paths, here in PROLOG list notation, to be read from right to left when descending in the hierarchy.
- To specify the very highest orbital level, the state expression `::$sc` is used. The reason for this is that `::arg` evaluates *arg* in the statechart level (i.e. machine path *sc*), not at an absolute root level (machine path []). This convention is convenient for statechart-global declarations such as `::alpha`, `::var1`. But to specify an orbital state at statechart level it is admittedly not so convenient. Since `::` must take an argument, it will be the statechart name, and the evaluation scope must be further back still, which is effected by the `$`.

**5.8.4 UML pseudo-states**

In a future release, we hope to introduce UML pseudo states `no_history`, `history` and `deep_history` which will give the user more flexible control over the issue. Figure 87 shows how transitions would be made to pseudo states and what the effective target state is (by means of the dotted arrow). Multi-target transitions to a mixture of pseudo and real states would have to be supported (not illustrated).



**Figure 87. Pseudo-states (option for possible future implementation)**

### 5.8.5 Illegal transitions

The following figure illustrates some examples of illegal transitions

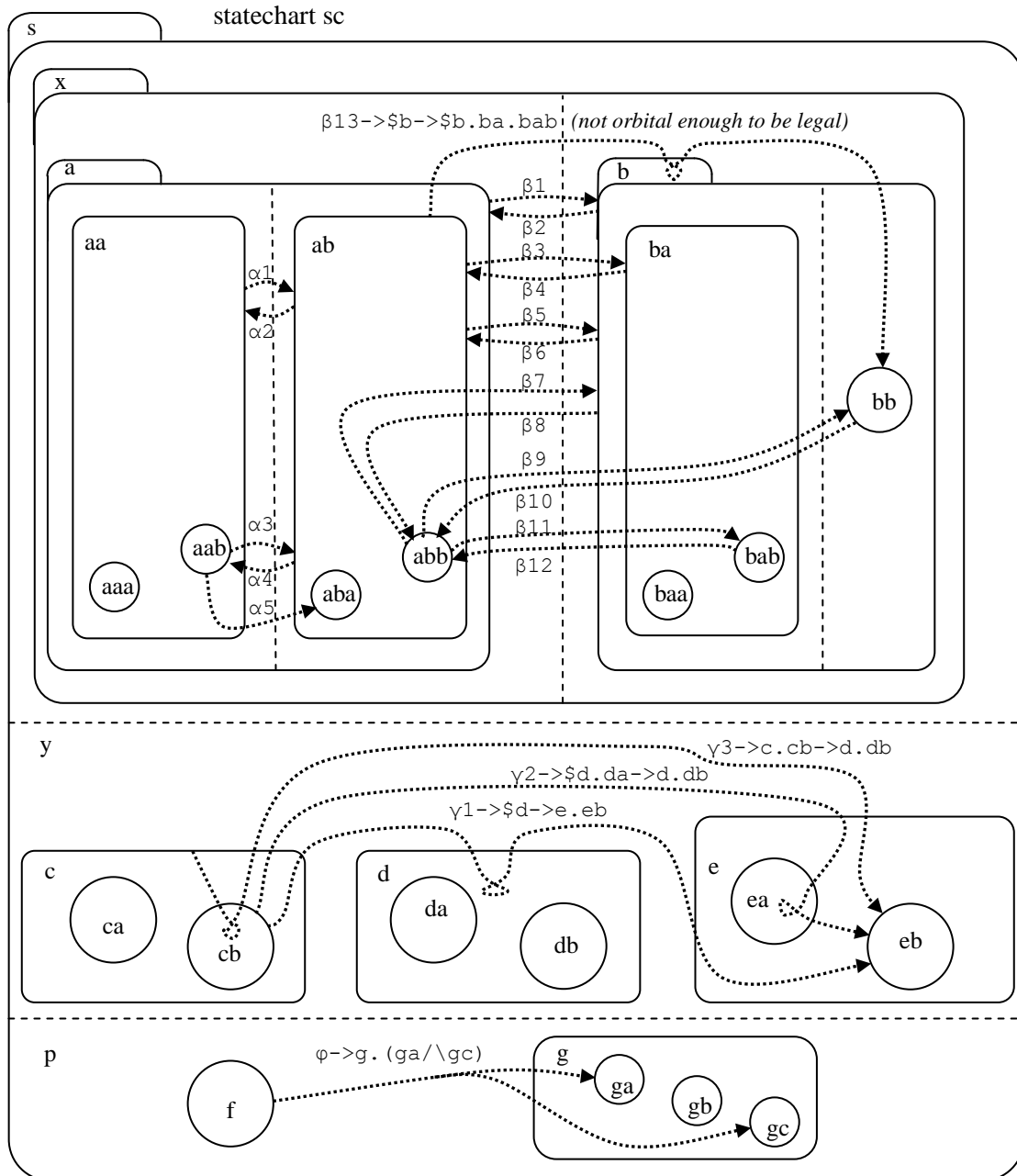


Figure 88. Illegal transitions



### Categories of (potentially) illegal transitions

- Set member to co-member: the transitions on  $\beta 1$  etc. Such transitions can be legalized by raising the orbit.
- Illegal route: the transitions on  $\gamma 1$ ,  $\gamma 2$ ,  $\gamma 3$  do not have a straight-out straight-in route.
- Multiple target states include cluster co-members: the transition on  $\phi$ .

### Detection of illegal transitions

It is possible to detect before executing a transition whether it is legal or not, at least for cases where the transition is *always* illegal. The STATECRUNCHER validator could do this; it is an option for an extension. Hong provides rules for how this could be done [Hong] (though these do not allow for orbital transitions).

Assuming that the worst thing that can happen with an illegal transition is that the state machine is left in an illegal state, there is a simpler way to check for illegal transitions. It is to execute the transition anyway, and examine the resulting state for integrity. Integrity means that

- the statechart machine as a whole is in an occupied state
- exactly one member of every **occupied cluster** is occupied; the rest are vacant
- all members of every **occupied set** are occupied.
- all members of a **vacant set or cluster** are vacant

Integrity checking is used in the test suite for STATECRUNCHER, but it is slow, and has not been included in normal use of the product. The user bears responsibility not to specify illegal transitions.

### 5.8.6 Actions

Actions occur in upon-enter and upon-exit blocks and in transition blocks. The kinds of action have already been seen, and are as follows

- expressions
- firing of events
- conditional actions containing any of these three kinds of action in the *if* and optional *else* part.

Expressions can contain function calls, and might only consist of a function call, and need not return a value.

Under the current semantics (discussed in section 6), actions in one action block take place sequentially.

### Knock-on effects of fired event actions

Anticipating the discussion on semantics, we show in the following model that actions as currently implemented can have knock-on effects. Use will be made of this in composing models (section 6.5).

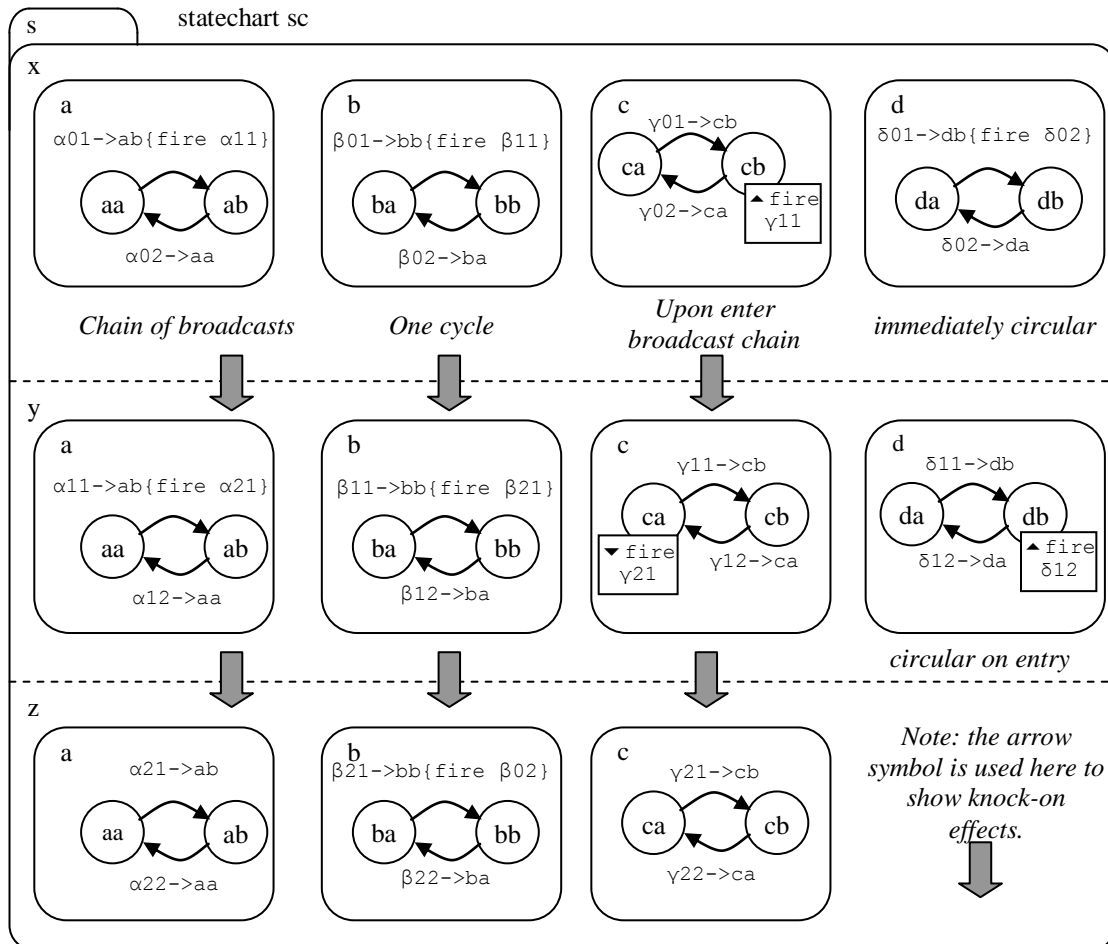


Figure 89. Knock-on effects of fired event actions

### 5.8.7 Labels

Labels can be used to provide extra information about transitions. Specific labels have not currently been finalized, but candidates are:

- the execution **time** taken in performing a transition. It could be based on an actual measurement. This enables transition tour algorithms to optimize test cases against execution time.
- the **probability** of a transition in the case of fork nondeterminism. This could make some optimizations in testing strategy possible; see [Zhang].

- the **cost** of a transition, if there are factors other than execution time that make a transition expensive (or cheap). Any transition requiring manual intervention or observation would probably be classed as very expensive.
- a **name** for the transition
- a **usefulness** factor indicating how important it is felt that such a transition should be taken in a test suite.

If it turns out that there is a need to provide a selection from *various options of distinct transition semantics* for some transitions, a label could be used to identify the semantics required in each case.

## 6. Algorithmic sequencing

There are many different approaches that can be taken as to how a transition algorithm should be designed, with the decisions taken affecting the possible features and semantics of the statechart system as a whole. The characteristics of various state machine systems in the literature, including that of [Harel], have been compared in a paper by [von der Beeck]. In [StCrBibIRef], where we annotate that reference, we characterize STATECRUNCHER according to von der Beeck's criteria.

Here, we first consider how steps in the algorithm can be sequenced, this being a key area for exploration and evaluation of alternatives. Then, having motivated and taken the main decisions, we describe the transition algorithm in detail (chapter 7).

The relationship between aspects of the transition algorithm and *process algebras* (or *process calculusses*) such as CCS and CSP is rather complex, and we approach our transition algorithm design from an algorithmic rather than an algebraic perspective. However, having arrived at a satisfactory transition algorithm, accommodating composition and interaction of statecharts, we are able to make a comparison with the CCS and CSP approaches. For that, we refer the reader to our appendices [StCrSemComp], [StCrDistArb] and to the dining philosophers problem discussed in section 9.4. In addition, we have taken an example Z specification, for the game of Nim, and implemented it in STATECRUNCHER, showing the relationship between the two formalisms.

This section addresses the (potentially conflicting) requirements of:

- Allowing repeated cycling through a sequence of transitions - though Lucas and von der Beeck consider this undesirable [CHSM, section 1.4.2.2].
- Ensuring machine integrity (i.e. ensuring that the rules for occupancy of states according to their kinds and their parent-child relationships are not violated).

Sequencing issues concern:

- When conditions on transitions are evaluated.
- The use of an original or current value of a variable.
- The ordering of processing of on-transition actions.
- The ordering of processing of upon-exit actions.
- The ordering of processing of upon-enter actions.
- The ordering of generation and processing internal meta-events.

The design of algorithms to meet the requirements is a matter of identifying the micro-steps of the transition algorithm and sequencing them in the right order. Some algorithms introduce extra restrictions on transitions, e.g. blocking them when other transitions are in certain phases of execution, but our final choice of algorithm does not require any special restrictions.

## 6.1 Cycling

Consider the transitions of the figure below:

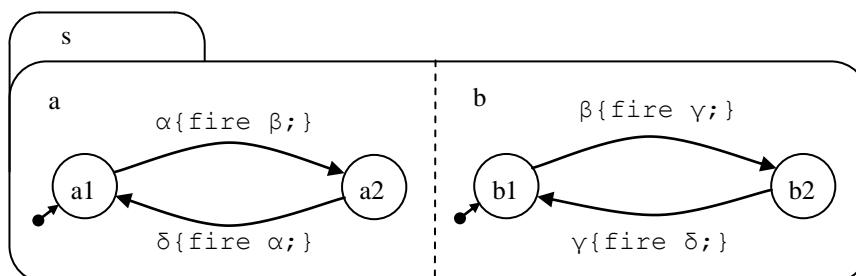


Figure 90. Cycling

Starting with the transition on  $\alpha$ , a cycle is seen: effectively (via the transitions and their actions)  $\alpha$  fires  $\beta$ ,  $\beta$  fires  $\gamma$ ,  $\gamma$  fires  $\delta$  and  $\delta$  fires  $\alpha$  again. Clearly, this machine as it stands is unsuitable, at least for testing purposes. However, if there were extra conditions and actions on the transitions, the cycling might be terminated at some point, as follows:

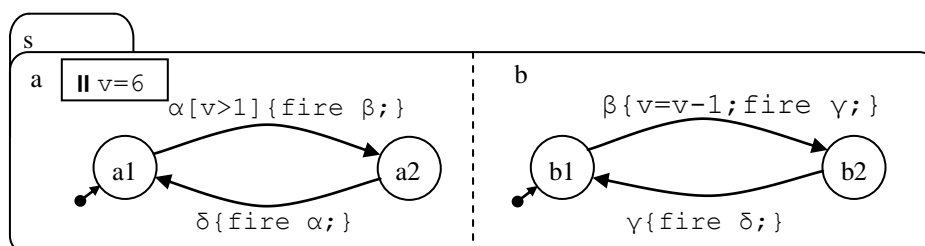


Figure 91. Cycling with termination

Here, a variable  $v$  is initially set to a value of 6. The start of the cycle has a guard on it,  $v > 1$ . The cycle decrements  $v$  on the transition on  $\beta$ , so the loop will terminate. It is possible that certain systems should be modeled this way. For example, if  $v$  is the volume of a television, it might be that a client module needs to reduce the volume step by step to the minimum volume, but that the actual decrementing is done in a separate server module. Another application of cycling to generate interleavings of system-under-test-internal events (over which the environment has no control, such as *notifications*), with user-generated events. This might be done with self-transitions cycling a number of times, generating the required events, using nondeterminism to generate different interleavings. The problem is addressed in [Trew 03].

We note that CHSM prevents cycling by ‘marking’ transitions [CHSM]. CHSM processes fired (‘broadcast’) events after exiting all states on the initiating transition, but before entering any states. Every time a transition is ‘taken’, it is excluded from further participation in the processing ensuing from the initiating transition.

An alternative way to prevent cycling is to block *states* involved in the initiating and subsequent transitions as they are taken. This is considered below in the context of maintaining machine integrity. But, in the STATECRUNCHER system, we ultimately opt for an algorithm that allows cycling and does not require marking transitions as taken or blocking states.

### **Prevention of infinite cycling**

If no protection is built into a system to prevent infinite cycling, then the system will probably crash on a heap or stack overflow condition, though it is conceivable that some kinds of infinite loops will run indefinitely without consuming memory. Given that we do not mark transitions as taken, or block states, infinite cycling could be prevented by recognising that a configuration of state occupancies, state histories, variable values and traces has been seen before in the cycle. However, this is computationally expensive, as it involves comparing the configuration of a machine (which may be quite extensive) with a number of recorded configurations (which may be quite high). A weakened version of this is to evaluate a hash function of the full state, and to store and compare against that instead. If the co-domain of the hash function is effectively a set of say  $2^{64}$  ( $\approx 10^{20}$ ) pseudo-random numbers, then the probability of a false positive match compares favourably with the probability of the user being struck by lightning in a year ( $\approx 10^{-8.5}$ ). A weaker method still is to count transitions executed within the compass of an initiating transition, and to put a maximum, say 100, on the number of ensuing transitions. The initial version of STATECRUNCHER for simplicity will not contain protection against cycling, thus leaving the responsibility with the user (as with looping in conventional programming languages).

## **6.2 Maintaining machine integrity**

During a transition, there are **five** sources of new events (which can, of course, entail new transitions). A major design issue in the transition algorithm is when to perform them. We first review them:

- ***exit meta-events***

These are meta-events that are generated when a state is exited. Other transitions may be triggered by this event.

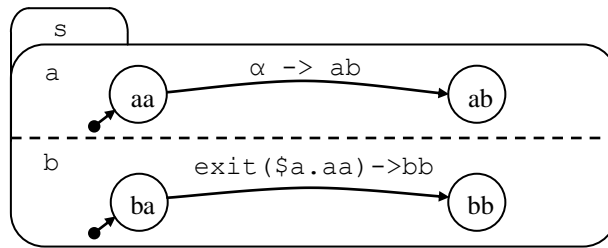


Figure 92. Review of exit meta-events

- *upon exit actions*

These are specified as part of a state's transition block, but they belong rather to the state than any one transition. They are the actions that are executed when the state is exited, and can contain events to be fired.

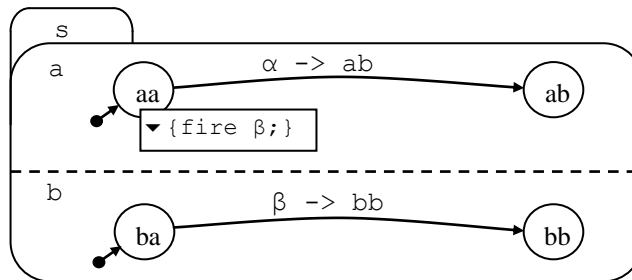


Figure 93. Review of upon-exit actions

- *transition actions, which may consist of firing new events*

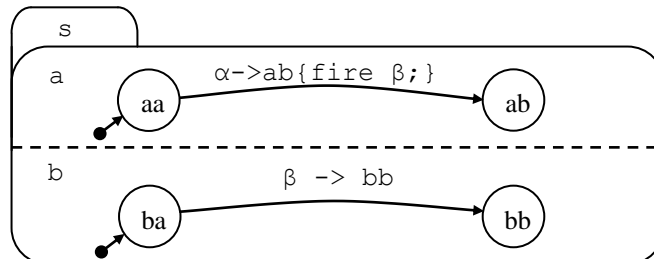


Figure 94. Review of transition actions

- *upon enter actions, analogous to upon exit actions*

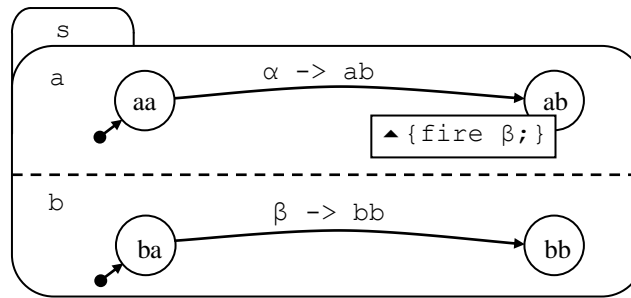


Figure 95. Review of *upon enter* actions

- *enter meta-events, analogous to exit meta events*

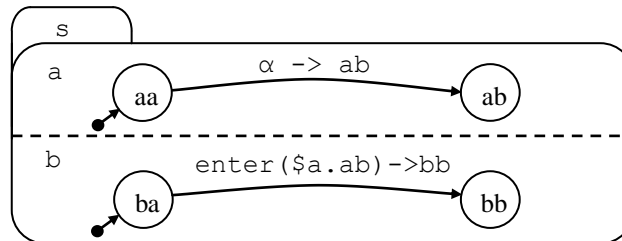


Figure 96. Review of *enter* meta-events

A major algorithm design issue is when to process these processing steps. As will become apparent, it is not a good idea to execute any of these actions as they occur. Instead, it is better to *collect* the actions first, and *execute* at some other time. This gives us various possibilities as to exactly when to execute them, and what other precautions need to be taken.

What we do *not* do is to regard differing execution strategies as differing nondeterministic interpretations that must be catered for. This would lead to excessive generation of ‘worlds’ as combinatorial explosion took place. Instead, these processing steps must follow a prescribed sequence. The modeller should be aware of this sequence, and if, exceptionally, alternative orderings are required, they should be modelled manually using existing STATECRUNCHER constructs.

In addition to the ordering of transition actions and meta-events, two more issues arise. They concern:

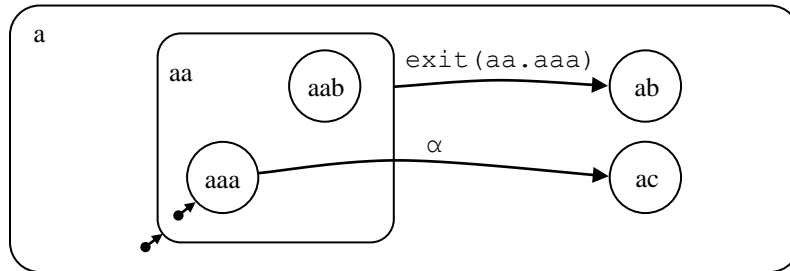
- When conditions on transitions are evaluated.
- The use of an original or current value of a variable.

We first acquaint ourselves with situations leading to potential breakdown of machine integrity.

Figure 97 shows one way in which, unless precautions are taken, performing transition actions too early can lead to breakdown of the statechart integrity. Suppose state *aaa* is



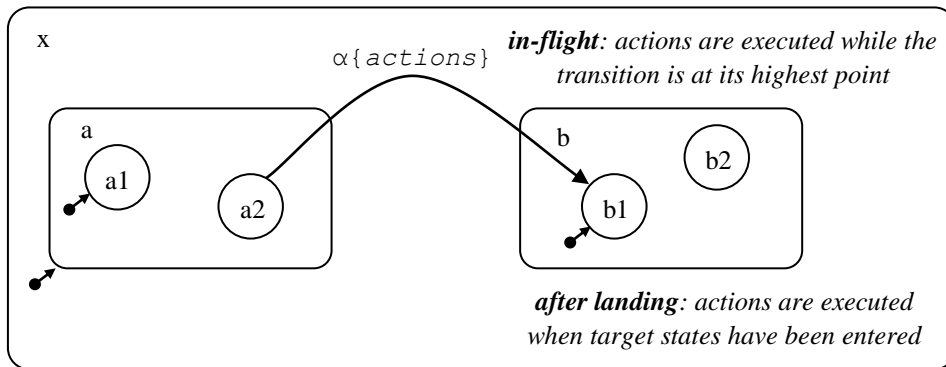
occupied. On event  $\alpha$ , state *aaa* is exited. If we immediately process the `exit(aa.aaa)` meta-event (and so exit state *aa* and enter state *ab*), and then return to the transition on  $\alpha$ , we also end up in state *ac*, and so break the cluster rule that only one member can be occupied.



**Figure 97. Integrity threat (1)**

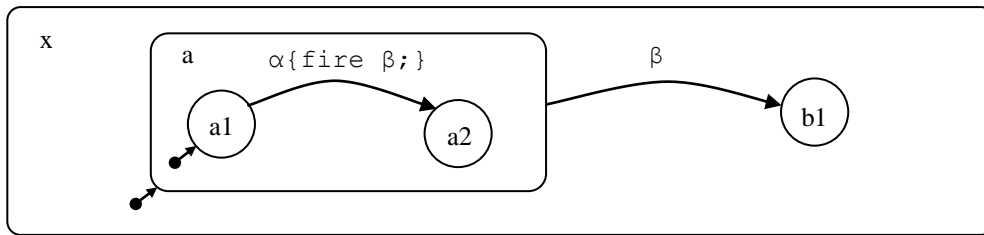
One option in avoiding integrity breakdown would be to cancel state *ab* as an occupied state when entering state *ac*. However, this leads to other problems: what if there were actions on `enter(ab)`? It would be most inelegant to have to undo them.

Other solutions are in two basic categories, depending on whether the transition actions are performed *in-flight* or *after-landing* of the transition. *In-flight* means that the actions are performed after the transition has performed all its state exit duties, but before its state entry duties, and with some precautions in place. *After-landing* means that the transition actions are executed after the target states have been entered.



**Figure 98. In flight and after landing**

A simple test for whether a statechart system uses an in-flight or after-landing approach, is as follows:



**Figure 99. Distinguishing *in-flight* and *after-landing***

If the system uses an in-flight transition algorithm, then the event  $\beta$  will have no effect (unless the algorithm is adapted in some way). If the after-landing approach is taken, then fired event  $\beta$  will trigger a knock-on transition.

## 6.3 An *in-flight* approach

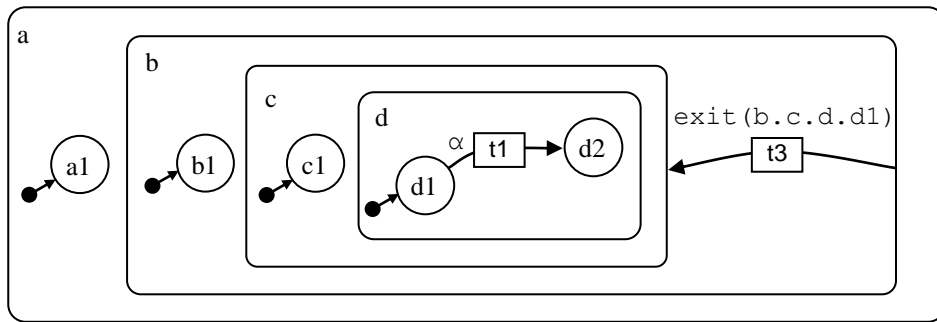
### 6.3.1 In-flight state blocking

Although the *in-flight* approach will be laid aside in favour of the *after-landing* approach, we consider it in detail since it is an intuitive approach, is applicable for some purposes, and (with many variations possible) is present in the literature: see [von der Beeck]. Many issues that are raised in the in-flight descriptions that follow are also applicable to the after-landing approach.

Consider Figure 97 again. We postpone consideration of execution of `exit(aaa)` until the transition on  $\alpha$  has reached its outermost point, and block the exited states from further participation in the transition algorithm. By the time we consider `exit(aaa)`, state `aa` is in a blocked state, which we will call *shadow-vacant*. The `exit(aaa)` meta-event becomes inapplicable and integrity is preserved.

We introduce the concept of *shadow-exiting* and *shadow-entering* a state. The states that *will be* exited and entered are first *collected* (or *acquired*) on traversing a transition route, so as to acquire `exit(...)` and `enter(...)` broadcast events and upon `exit` and upon `enter` actions. As they are collected, these states are set to a state which is neither occupied nor vacant: *shadow occupied* or *shadow vacant*. These shadow states are temporary internal states that can be regarded as *blocked* states, since they block further transitioning on them. If the source state or *any* target state of a transition is blocked, the whole transition is inapplicable. Shadow states are set to a real *vacant* and *occupied* state towards the end of the algorithm.

However, a little more is needed. Consider the following situation:

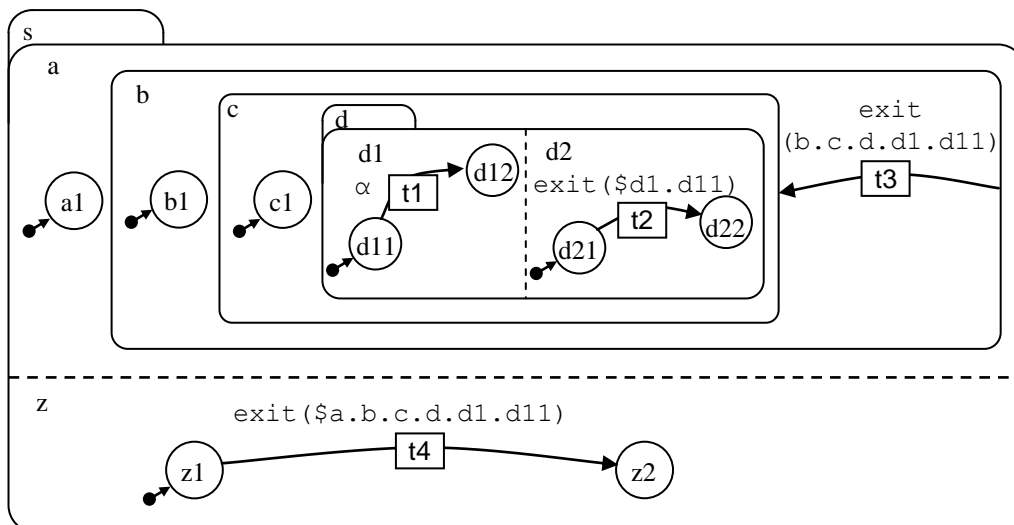


**Figure 100. Integrity threat (2)**

Suppose in the above machine, the transition  $t_1$  on  $\alpha$  takes place. When state  $d_1$  is exited, transition  $t_3$  from  $b$  to  $c$  will *potentially* be triggered. Although this transition *could* be executed after processing the original transition on  $\alpha$  (which would take us to state  $c_1$ ), we would opt to block it. It ‘interferes’ with the *incomplete* originating transition  $t_1$  on  $\alpha$  in the sense that the transition is robbed of its target state. A way we could prevent this kind of transition is by blocking all non-shadow-exited or shadow-entered *ancestral* states up the hierarchy from  $d_1$  as far as the statechart level, (so only leaving *non-ancestral set co-members* unblocked). If there are no sets, then all states will be blocked. If there are multiple target states, we apply the blocking technique to the relevant ancestors of *all* these target states.

States which need blocking but are not shadow-exited or shadow-entered are given a simple blocked state until the end of the transition, when they are necessarily restored to *occupied* (since if they are not shadow exited, they must remain occupied).

The example below shows an elaboration of the previous example where some set-co-members remain unblocked.



**Figure 101. Integrity threat (3)**

On processing transition  $t_1$ , the whole of member  $a$  of set  $s$  will be blocked except set member  $d_2$ . There is no blocking of member  $z$ , so the transition there (transition  $t_4$ , on exiting  $d_{11}$ ) can take place. Nor does it affect state  $d_2$ , as it is not an ancestor of the source or target state of our original transition  $t_1$  on  $\alpha$ . So the transition  $t_2$  from  $d_{21}$  to  $d_{22}$  can in principle be triggered. Transition  $t_3$  is invalid in this situation.

### Orbital transitions

An orbital transition is blocked if its orbital level takes it to a blocked state. In the figure below, as transition  $t_1$  takes place, transition  $t_2$  becomes blocked, because state  $d$  becomes blocked, so it cannot be exited or entered.

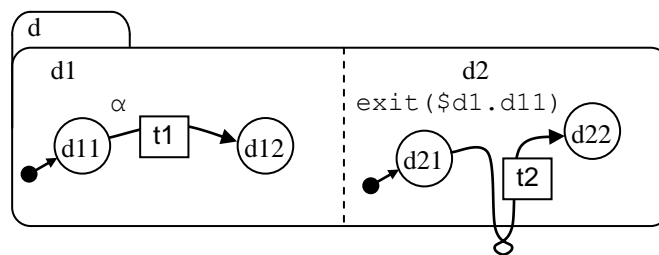


Figure 102. Blocking of orbital transitions

### Unblocking of states

Each transition causes its own set of states to be blocked. In the example below, processing the transition on  $\alpha$  will block states  $d_{11}$ ,  $d_{12}$ ,  $d_1$  and  $d$ ; the transition on  $\beta$  will block  $d_{21}$ ,  $d_{22}$ ,  $d_2$  and  $d$ . As the processing of  $\beta$  completes, the states that were blocked by processing of  $\beta$  *only* will be unblocked, i.e.  $d_{21}$ ,  $d_{22}$  and  $d_2$ .

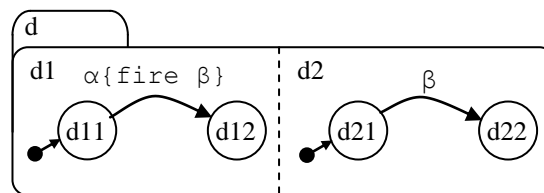


Figure 103. Unblocking example

### History

The history setting is only relevant on entering a vacant cluster. Since, under an in-flight approach, a vacated cluster cannot be re-entered as a consequence of the one initiating event, it is not critical when history is set. History can conveniently be set when a state is *really* vacated.

The record of a historically occupied child can conveniently always be set whether or not the History/Deep History markers indicate that it is required. The issue of whether to *make use* of

this data is resolved on cluster entry. This policy is robust in the event of changes to the algorithm.

### 6.3.2 When should the *conditions* associated with transitions/actions be evaluated?

Under ‘the conditions’ we understand

- the requirement that a source state is occupied
- the requirement that the boolean condition expression, (or guard), evaluates to true.

The options are:

- at collection time only
- at execution time only
- on both occasions

The choice will depend on either what is *necessary* to ensure machine integrity, or what is *expedient*, in giving the most desirable behaviour. We consider a number of typical situations, and the consequences of each strategy in each case.

The issues revolve around race-nondeterministic situations. The key question is: if two or more transitions on the same event are eligible at collection time, can the consequences of starting or completing one invalidate the other?

In the following figure, at transition collection time, two transitions on  $\alpha$  are valid. This gives rise to race nondeterminism, so that transition sequences  $\langle t1, t2 \rangle$  and  $\langle t2, t1 \rangle$  will be prepared. If the condition  $[v==0]$  is re-evaluated at execution time, then in the world which processes  $\langle t1, t2 \rangle$ , transition  $t2$  will not take place.

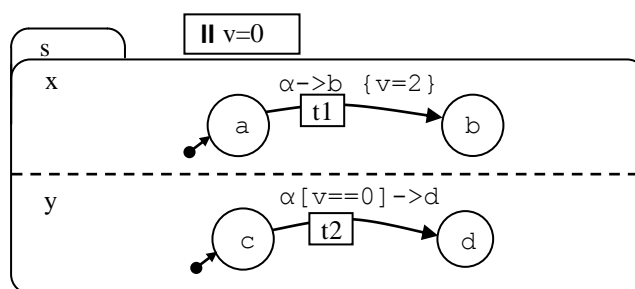
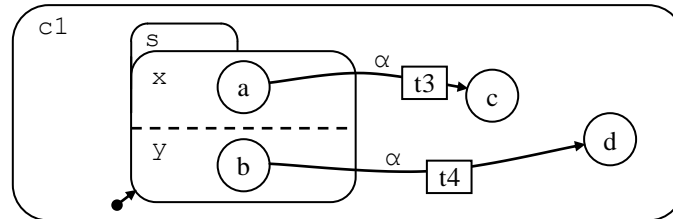


Figure 104. Race with arithmetic transition condition

The following figure shows that there is a need for condition re-evaluation, at least as regards the occupancy requirement. There is race nondeterminism. However, owing to statechart integrity considerations, one transition *must* invalidate the other. This can be achieved by in-flight blocking or execution time re-evaluation of the conditions (including the source state occupancy). Under nondeterministic processing, a world will be generated in which c is the

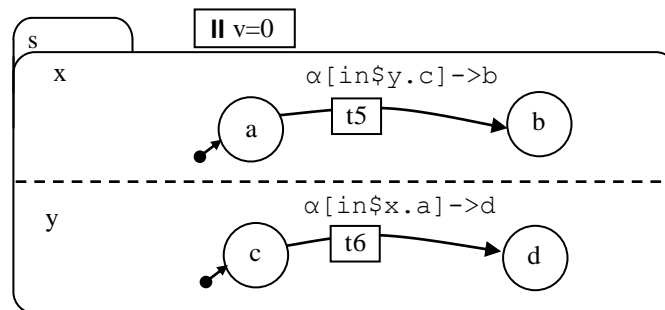
final occupied state and a world will be generated in which d is final occupied state. It is simply not possible to proceed on the basis that (perhaps just in some world) both transitions must take place.



**Figure 105. Race with occupancy requirement**

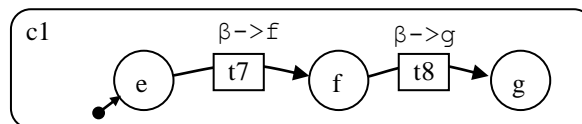
In the Figure 106, it might be argued that (whatever the nondeterministic world being considered), both transitions should take place. However, it can also be argued that one transition does invalidate the other, as in the previous figure.

It will furthermore be argued that if a transition sequence, as produced by nondeterministic processing, such as  $\langle t5, t6 \rangle$  is to be processed *as a sequence*, then the second transition in the sequence must take into consideration the effects of the first.



**Figure 106. Race with in(..) transition condition**

The following figure shows that collection of transitions is a one-off process. Suppose event  $\beta$  occurs when state e is occupied. Although transition  $t8$  becomes eligible for processing, it must not be processed on the same occurrence of event  $\beta$  that triggered  $t7$ , because when  $\beta$  occurs, state f is vacant.



**Figure 107. Collection is a one-off process**

### Conclusion on condition evaluation

In view of the threats to machine integrity in race condition situations, we opt for condition evaluation at *collection and execution time*.

### 6.3.3 Mutual order of actions and meta-events

We consider the best order in which to process transition actions and meta-events relative to each other, the categories being:

- *exit* meta-events
- upon exit actions
- transition actions
- *enter* meta-events
- upon enter actions

The order is relevant, because

- Variables are always referenced in the *latest* context – not, say, the context just prior to the transition. So if a variable is modified by one collected action, a subsequent collected action will see the modified value.
- States are also referenced in the latest context, and may become occupied or vacant through a certain action, so that subsequent fired or generated events do not trigger a transition which they would otherwise have triggered.

Within the context of one transition, each action, however ordered, will be completed before the next one is executed, so it will never be the case that one action causes new blocked states to come into effect and be ‘seen’ by subsequent actions in the list of collected actions. Note, however, that knock-on actions, (actions associated with transitions triggered by events that were fired as an action of an original transition) will typically see more blocked states.

It is clear that *upon exit* actions should precede *upon enter* actions and that these should take place in the order in which they were generated. Similarly *upon exit* meta-events should precede *upon enter* meta-events.

Where in the sequence should the transition actions be executed? Candidate orderings are:

- (1) 1<sup>st</sup> transition actions, 2<sup>nd</sup> *exit* and *upon exit* actions, 3<sup>rd</sup> *enter* and *upon enter* actions
- (2) 1<sup>st</sup> *exit* and *upon exit* actions, 2<sup>nd</sup> transition actions, 3<sup>rd</sup> *enter* and *upon enter* actions
- (3) 1<sup>st</sup> *exit* and *upon exit* actions, 2<sup>nd</sup> *enter* and *upon enter* actions, 3<sup>rd</sup> transition actions

Option (2) has an intuitive feel to it. Note that *actually entering* the target state can never be invalidated by earlier actions, because it has already shadow-taken-place. One disadvantage is that transition actions cannot override on-enter actions. This would be useful, as on-entry actions are generic to many transitions. So if an on-entry action is  $v=v\%3$ , (the modulo function) but for a specific transition we would like  $v$  to be set to 5, we cannot do it this way. A work-around is to cancel the on-entry actions and re-write all relevant transition actions to include the appropriate assignment to  $v$ . This argument lends support to option 3.

However, we feel that user-intuitiveness is important, and provisionally choose option (2).

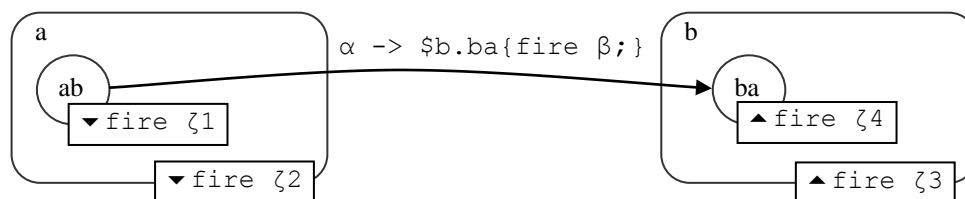
As to the question of the order of *exit meta-events* versus *upon exit actions*, we choose to do the *upon exit actions* first. Similarly concerning the order of *enter meta-events* versus *upon enter actions*, we choose to do the *upon enter actions* first.

Where a hierarchy is exited, the *exit meta-event* and *upon exit actions* for one level are performed before those of the next level up. Similarly where a hierarchy is entered the *enter meta-event* and *upon enter actions* for one level are performed before those of the next level down.

The ordering of all aspects of transition processing for the in-flight approach is therefore:

- (1) shadow exit (all relevant states), collecting exit meta-events and upon exit actions
- (2) shadow enter (all relevant states) collecting enter meta-events and upon enter actions
- (3) block ancestors
- (4) execute upon exit actions (loop with next step)
- (5) execute exit meta-events (inner loop to previous step for each hierarchical level)
- (6) execute transition actions
- (7) execute upon enter actions(loop with next step)
- (8) execute enter meta-event (inner loop to previous step for each hierarchical level)
- (9) unblock ancestors
- (10) execute real exit (loop with next step)
- (11) set history (inner loop to previous step for each hierarchical level)
- (12) execute real enter (all relevant states)

We illustrate this with an example:



**Figure 108. Order of actions on hierarchical entry/exit**

The ordering, with bracketed reference to the above numbering, will be:

1. (1a) shadow exit ab
2. (1b) shadow exit a
3. (2a) shadow enter b
4. (2b) shadow enter ba
5. (3a) execute upon exit (ab) action (fire ζ1)
6. (4a) execute meta-event exit (ab)
7. (3b) execute upon exit (a) action (fire ζ2)
8. (4b) execute meta-event exit (a)



9. (6) execute on-transition action (`fire β`)
10. (7a) execute upon enter (b) action (`fire ζ3`)
11. (8a) execute meta-event enter (b)
12. (7b) execute upon enter (ba) action (`fire ζ4`)
13. (8b) execute meta-event enter (ba)
14. (10a) real exit ab
15. (10b) real exit a
16. (11b) set history of a
17. (12a) real enter b
18. (12b) real enter ba

### Major disadvantages of the in-flight approach

The problems with the in-flight approach are that by blocking states:

- it *prevents cycling*. The fact that this is so can be seen by reference to Figure 90. The transition on  $\alpha$  fires  $\beta$ , which triggers a transition involving states which will not be blocked, so that transition can take place, However, the transition on  $\gamma$  will not take place because its source and target states are blocked. Cycling has been found to be useful in generating a number of interleaved traces.
- it may prevent *knock-on* transitions as in Figure 99. In-flight approaches that do not prevent (all) knock-on effects may be possible. It will be seen that knock-on transitions are essential to composition of models (section 6.5)

Given that the ability to cycle under well-constructed circumstances is desirable, and the relative complexity of blocking and unblocking states, we examine an alternative approach (in the next sub-section), which we will adopt.

## 6.4 An *after-landing* approach

### 6.4.1 *After landing* ordering

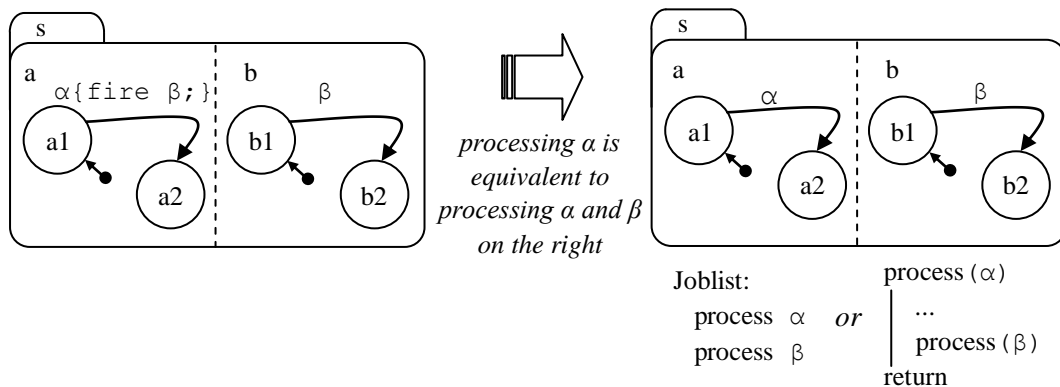
In this approach, the transition actions are executed after the initiating transition has actually entered the target states. Fired events (and other actions, and meta-events) are processed after completion of exit and enter processing of the transition that fired them. Processing them may be done by an in-line call at the end of processing the original transition, or by placing the new event as a job in a buffer, which we could call a *joblist*, for a read-execute loop.

*The actual implementation in STATECRUNCHER is an in-line call, elaborated on in Figure 140, (p.170).*

The net effect in either case is that, referring to Figure 108 again, we have a new ordering such as the following:

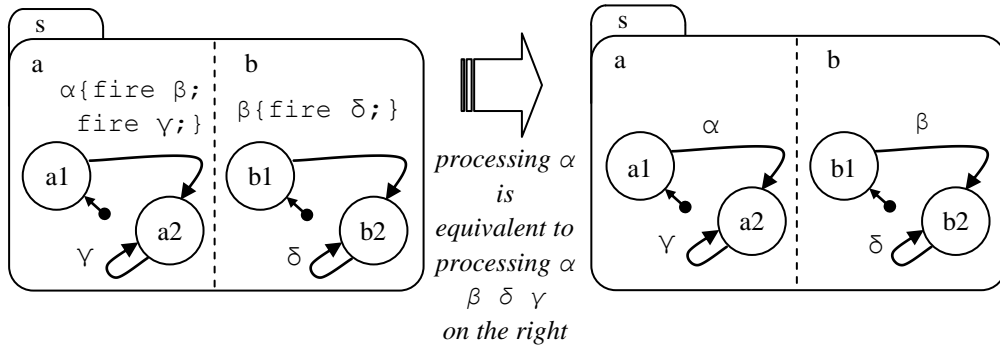
1. real exit ab
2. real exit a
3. set history of a
4. real enter b
5. real enter ba
6. execute upon exit (ab) action (fire  $\zeta_1$ )
7. execute meta-event exit (ab)
8. execute upon exit (a) action (fire  $\zeta_2$ )
9. execute meta-event exit (a)
10. execute on-transition action (fire  $\beta$ )
11. execute upon enter (b) action (fire  $\zeta_3$ )
12. execute meta-event enter (b)
13. execute upon enter (ba) action (fire  $\zeta_4$ )
14. execute meta-event enter (ba)

In Figure 109, the transition on  $\alpha$  will be processed to completion, while its action (fire  $\beta$ ) will be collected and executed afterwards.



**Figure 109. After-landing equivalence**

A more complex example shows how multiple fired events and their consequent actions are sequenced:



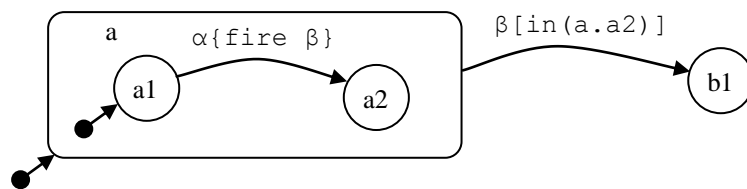
**Figure 110. Multiple fired events**

Note that the list of events to be processed is built up by depth-first traversal of the nested fired events, but that all are processed at a top-level after completion of the previous one – there is never anything to be re-visited for a previous event. For the processing order to actually make a difference in our example, there would have to be more detail in the model, such as variable assignments on the self-transitions, but we keep the example simple.

### 6.4.2 Condition evaluation

The discussions under the in-flight approach on race conditions apply equally well to the after-landing approach, as they are not concerned with fired events. The conclusion there, that conditions on transitions processed in transition sequences must be re-evaluated at execution time, applies to the after-landing approach too. An example is given illustrating the time reference of the `in()` function below.

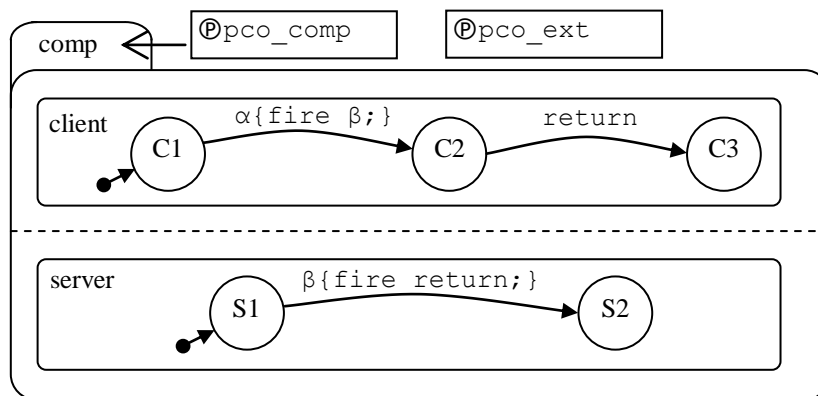
The after-landing approach views transitions such as the one on  $\beta$  in Figure 111 from the point in time of completion of the transition on  $\alpha$ . The transition on  $\beta$  will be accepted. This may or may not correspond to the user's instinctive idea of when conditions are evaluated.



**Figure 111. Time reference of the `in()` function**

## 6.5 Client-server composition and PCOs

In this section, we see how the after-landing approach enables us to model one software component or function calling another using fired events.



**Figure 112. Component composition**

**Points to note**

- STATECRUNCHER's composition paradigm is closely analogous to the function call and return of imperative languages such as 'C'.
  - The *making* of the function call is modeled by a fired event
  - The *response* to this is modeled by a transition on the event that was fired
  - The *return statement* is modeled by fired return event
  - The *response* to this is modeled by a transition on the return event that was fired.

If there are many such calling sequences in a model, return names can be made unique to a server function by affixing the function name to the event (e.g. `return_max`) or by putting the return event in a sufficiently local scope (using STATECRUNCHER's scoping capabilities).
- The client can be seen as an independent state machine, which can be driven through its cycle with events  $\alpha$  and `return`. It does not care who it is that responds to its firing of  $\beta$ , nor who it is that provides the `return` event. A different server to the one shown might be connected to the client, e.g. with more states and transitions between its initial and final states (S1 and S2). Similarly, the server is independent of its client, except for the agreed interface of  $\beta$  and `return`.
- Event  $\alpha$  is supplied externally to the client and server. Events  $\beta$  and `return` are part of the agreed interface between the client and server. We indicate this by putting the events on different PCOs. STATECRUNCHER's output will reveal the PCOs so that a test generator program can distinguish, and if required, restrict itself to certain PCOs only. We put  $\alpha$  on `pco_ext` (for *external*) and  $\beta$  on `pco_cmp` (for *composition*). If we had more events local to the server only, say, we could put them on `pco_serv` and so on, but we have kept this model to the basics.
- The scheme would not work with the in-flight approach, because the return event would not be eligible when needed.

For a discussion of these semantics in relation to the process algebras CSP and CCS, see [StCrSemComp].

## **6.6 Conclusions on the sequencing in the transition algorithm**

Given a requirement to allow cycling and composition between parallel machines, the conclusions for the best approach to the transition algorithm are:

- An after-landing approach
- Condition re-evaluation for transitions at the time they are executed.

# 7. The transition algorithm

## 7.1 The formal statechart and the nondeterministic transition function

Finite state machines (FSMs) are often formally described without reference to the hierarchical structures of a Harel or UML or STATECRUNCHER statechart (Harel's AND- and XOR-states; in UML's concurrent and non-concurrent composite states; STATECRUNCHER's sets and clusters). This is because the hierarchical structure is just a convenient way of expressing a mathematically equivalent flattened state space. When the hierarchy is introduced, the terminology changes from FSMs to *statecharts*, but the two are equivalent. A state in the flattened state space is an element of the Cartesian product of parallel states in the statechart. Only statechart leafstates need be considered, because the occupancy of their ancestors is a derivative of that of the leafstates. If the statechart contains history, variables and traces, then these must also present as terms in the Cartesian product in defining flattened states.

Just as the hierarchical states of a statechart offer convenience in representing the state space, so the *structured forms of nondeterminism* offer convenience in representing what is equivalent to *FSM nondeterminism* in the flattened state space. STATECRUNCHER simply structures the nondeterminism into various categories that are easy to visualize in a statechart. As has been seen, STATECRUNCHER supports the following forms of structured nondeterminism, all equivalent to fork nondeterminism in the flattened state space.

- fork
- race
- set-transit
- set action
- set meta-event
- fired event (*or* broadcast event) nondeterminism.

We gave an example of flattened race nondeterminism in Figure 35.

After processing an event STATECRUNCHER produces a *world* per distinct state configuration, which, in flattened state space terms, is equivalent to a world for every possible resultant flattened state.

We develop the notion of a world more formally, working from the definition of a NFSM (Nondeterministic Finite State Machine) given by [Hierons 98]:

An NFSM  $M$  is defined by a tuple  $(S, s_l, h, X, Y)$  in which

- $S$  is a set of states
- $s_l$  is the initial state
- $h$  is the state transition function
- $X$  is the input alphabet
- $Y$  is the output alphabet

Given an NFSM  $M$ ,  $S_M$  shall denote the state set of  $M$ . When  $M$  receives an input value  $x \in X$ , while in state  $s \in S$ , a *transition* is executed producing an output value  $y \in Y$  and moving  $M$  to some state  $s' \in S$ . The function  $h$  gives the possible transitions and has the type  $S \times X \rightarrow P(S \times Y)$  where  $P$  denotes the power set operator. ... An NFSM  $M$  is *completely specified* if, for each  $s \in S$  and  $x \in X$ ,  $|h(s,x)| \leq 1$ .  $M$  is *deterministic* if for each  $s \in S$  and  $x \in X$ ,  $|h(s,x)| \leq 1$ .

What in Hierons' description is the notion of  $M$  being *in state  $s$* , is to STATECRUNCHER *having an occupancy configuration  $s$ , and other dynamic properties*, where an occupancy configuration gives the occupancy (occupied or vacant) of every state. Several states can be occupied, due to parallelism (modelled by a STATECRUNCHER *set*), and hierarchy (the fact that a parent of an occupied state is also an occupied state). Remark: the occupancy of non-leaf states can be derived from that of their child states (by the *set* and *cluster* rules), so, given the hierarchical structure, the occupancy configuration need only explicitly comprise the set of occupied leaf states.

The '*other dynamic properties*' which  $s$  must comprise are cluster history and variable values.

In our definitions below, we define  $G(A \times B) \circ P(A \times B)$  to be the set of all *functions* from  $A$  to  $B$ .

A STATECRUNCHER statechart is therefore  $(C, V, P, s_l, v_l, p_l, X, Y, h)$  where

- $C$  is a hierarchy of states (sets, clusters and leafstates), from which we can easily derive
  - $S$ , the set of all states
  - $P$ , the set of all clusters,  $P \subseteq S$
- $V$  is a set of variables. We assume the range of values is finite - it is determined by practical limitations.
- $s_l$  is the initial state
- $v_l$  is a function giving the initial variable values,  $V \rightarrow Z$ , where  $Z$  is the set of integers
- $p_l$  is a function giving the initial history values per cluster,  $P \rightarrow S$
- $X$  is the input alphabet (a set of events in STATECRUNCHER)
- $Y$  is the output alphabet (a set of trace elements in STATECRUNCHER)
- $h$  is the state transition function

$$h : [S \times G(V \times Z) \times G(P \times S)] \times X \rightarrow P([S \times G(V \times Z) \times G(P \times S)] \times Y), \text{ where}$$

- the  $G(V \times Z)$  term represents all the variables with their values
- the  $G(P \times S)$  term represents all the clusters with their histories
- the [...] bracketing on the LHS and RHS is introduced because of the commonality of these terms; they are the STATECRUNCHER worlds, which we can denote by  $W$ . There may be no worlds in existence.

We could add to this definition

- $Q$  the set of PCOs
- $A$  the set of actions

and a way of attaching them to other components of the statechart, but PCOs are effectively a simple attribute to events, and actions can be absorbed into the transition function, since they occur on transitions and influence the final configurations.

The domain and range of  $h$  can be represented as

$$\begin{aligned} \text{domain}(h) &: [S \times G(V \times Z) \times G(P \times S)] \times Y = W \times X \\ \text{range}(h) &: P([S \times G(V \times Z) \times G(P \times S)] \times Y) = P(W \times Y) \end{aligned}$$

When an event is processed in many worlds, a new set of worlds is produced.

To represent this, we define a multi-input-world transition function:

$$\begin{aligned} H &: P(W \times X) \rightarrow P(W \times Y) \\ H(A) &= \zeta_{B^3 A} h(B) \end{aligned}$$

In a practical situation, the elements of the domain of  $H$  will all contain the *same* event in all the Cartesian product terms.

Remark: in the actual STATECRUNCHER implementation, *traces* also distinguish worlds, so we should strictly say that the dynamic configuration  $d$  of a statechart is of type

$$S \times G(V \times Z) \times G(P \times S) \times Y^*$$

where  $Y^*$  is the set of strings consisting of elements of  $Y$ , (including the empty sequence). So this could be considered to be the actual type of the range of the transition function  $h$ . However, the most efficient mode of operation is to clear traces and merge worlds between processing events; if this is not done, old and new traces are concatenated. Traces do not impinge on the transition algorithm. With this understanding, we discount the traces in a dynamic state; in this way we more closely map to the description given by Hierons.

Unfortunately, the term *state* is overloaded, since it can mean either of

- a part of a statechart: a set, cluster or leafstate. We may also call this a *state-machine* or just a *machine*.
- an occupancy configuration of a state-machine.

However, the word *state* is so much more natural than, say, *machine* and *occupancy* that it is often retained, with clarification where needed.



## 7.2 Statechart properties

The following definitions are available in expressing various properties of a statechart:

source(t): the source state of a transition t  
orbit(t): the orbital state of a transition t  
targets(t): the set of target states of a transition t  
cond(t): the condition on transition t, (dynamically true or false)  
actions(t): the sequence of actions attached to transition t

sources(T): the *set* of source states of a *set* of transitions T  
$$\text{sources}(T) = \{ \text{source}(t) \mid t \in T \}$$

parent(s): the set of parent states of state s (or  $\emptyset$  for a top level state)  
ancestors(s): the set of ancestor states (superstates) of state s (or  $\emptyset$  for top level states)  
children(s): the set of child states of state s (or  $\emptyset$  for leafstates)  
descendants(s): the set of descendant states (substates) of state s  
enter\_actions(s): the set of on-enter actions attached to state s  
exit\_actions(s): the set of on-exit actions attached to state s

Machine states S are partitioned into state-types { **clusters**, **sets**, **leafstates** }.

We also define

$$\text{nonleafs} = \text{clusters} \cup \text{sets} = S \setminus \text{leafstates}$$

For convenience, we write “s is a cluster” to mean “s  $\in$  **clusters**” etc.

Furthermore, the arrangement of states is a tree-like hierarchy:

- The set of top-level states is the set of states which are no state's descendant:

$$\text{toplevels} = \{ s \in S \mid \text{parents}(s) = \emptyset \}$$

- There is only one top-level state<sup>1</sup>.

$$|\text{toplevels}| = 1$$

- Clusters and sets must have at least one member (=child)<sup>2</sup>:

$$\forall s \in \text{nonleafs} \bullet |\text{children}(s)| \geq 1$$

---

<sup>1</sup> One could imagine allowing more than one top-level state, e.g. so as to have two totally independent machines in one source or object file. However, this has little value, and would complicate the descriptions.

<sup>2</sup> One could imagine allowing sets and clusters that contain no children. This would introduce a partition of cluster and sets into { **empties**, **nonempties** }. However, an empty set or cluster has little benefit (a leafstate will serve as a replacement). To allow empty sets and clusters would only complicate the properties of a statechart.

- Leafstates do not have children  
 $\forall s \in \text{leafstates} \bullet |\text{children}(s)| = 0$
- States have at most one parent  
 $\forall s \in S \bullet |\text{parent}(s)| \leq 1$
- If a state has children, then the parent of those children is the original state  
 $\forall s \in \text{children}(p) \bullet \text{parent}(s) = p$
- Ancestors are parents, or parents of ancestors; to express this nonrecursively:  
 $a \in \text{ancestor}(s)$  iff  $\exists$  some sequence  $(p_1, p_2, \dots, p_n)$  where  $p_1 = a$ ,  $p_n = s$   
*such that*  $\forall i \in [1, n-1] \bullet p_i = \text{parent}(p_{i+1})$
- Descendants are child states or descendants of child states:  
 $d \in \text{descendant}(s)$  iff  $\exists$  some sequence  $(p_1, p_2, \dots, p_n)$  where  $p_1 = d$ ,  $p_n = s$   
*such that*  $\forall i \in [1, n-1] \bullet p_i \in \text{children}(p_{i+1})$

### 7.2.1 Dynamic aspects of a statechart

Each state has occupancy; it can be *occupied* or *vacant*. States also have a history indication, although it is only relevant to clusters.

States have a history attribute, but for sets and leafstates it is *none*. For clusters it is either *none* or the child state that was last occupied.

A third dynamic aspect of a statechart is the value of the variables. We assume the range of values is finite – it is determined by practical limitations.

A full configuration of a statechart contains the occupancies of all states, all state history, and all variable values. An occupancy configuration  $F$  comprises a tuple

$\{\text{occs}, \text{vacs}\}^1$

where **occs** is the set of state that are occupied, and **vacs** is the set of states that are vacant.

---

<sup>1</sup> For an in-flight algorithm, this would be  $\{\text{occs}, \text{vacs}, \text{shadow\_occs}, \text{shadow\_vacs}, \text{blockeds}\}$

The function **H** maps a machine state in a configuration to its history.

**H**:  $F \times S \rightarrow \{none\} \cup S$   
 if for any  $\mathbf{H}(f,s) = x \in S$   
 then  
     s is a cluster  
 and  
      $x \in \text{children}(s)$

The function **V** maps a variable in a configuration to its (integral) value.  $Z$  is the set of integers (within some practical limits)<sup>1</sup>

**V**:  $F \times V \rightarrow Z$

### Configuration sets<sup>2</sup>

As discussed in section 4.9, nondeterminism is handled by creating *worlds* to represent the various alternative outcomes when an event is processed. Worlds contain the dynamic data associated with a statechart (state occupancy, state history and variable values). In other words, each world corresponds to a configuration.

A **configuration set** is a set of worlds  $W$  containing state data of a particular statechart. At specific intermediate phases of the transition algorithm, the configuration-set-to-be will in general be a *bag* of worlds rather than a set, though this will be converted to a set on completion of a transition.

### Properties of a valid configuration of a statechart

1. The statechart as a whole is occupied. This means that all top-level states (although we only allow one) are occupied:

$$\text{toplevels} \subseteq \text{occs}$$

2. Every state is occupied or vacant but not both  
 $\{\text{occs}, \text{vaccs}\}$  becomes a partition

3. For every *occupied* cluster, the number of occupied children is 1

$$\forall s \in \text{clusters} \cap \text{occs} \bullet |\text{children}(s) \cap \text{occs}| = 1$$

4. For every *vacant* cluster, no children are occupied

$$\forall s \in \text{clusters} \cap \text{vaccs} \bullet \text{children}(s) \subseteq \text{vaccs}$$

---

<sup>1</sup> Later additions are strings and arrays. Array elements can be counted as scalar variables, and the concatenated ASCII values in strings can be considered as integers.

<sup>2</sup> Another term that was considered to express this, but which is too imprecise, is *state vectors*.

5. For every *occupied* set, all children are occupied

$$\forall s \in \mathbf{sets} \cap \mathbf{occs} \bullet \mathbf{children}(s) \subseteq \mathbf{occs}$$

6. For every *vacant* set, no children are occupied

$$\forall s \in \mathbf{sets} \cap \mathbf{vacs} \bullet \mathbf{children}(s) \subseteq \mathbf{vacs}$$

### 7.3 Transition selection

We consider a statechart in configuration  $f$  under some event  $\alpha$

$T_\alpha$  is the set of all transitions on event  $\alpha$ , (whatever their condition and whatever the configuration-state of the statechart).

$T_{f,\alpha,true}$  is the set of all transitions where the associated source/orbit/target pre-requisites and transition conditions are true. The default condition is true. The source pre-requisite is that the source state is occupied. No orbit or target pre-requisite is needed in *after-landing* semantics. (Otherwise, these states must not be blocked in any way).

$$T_{f,\alpha,true} = \{ t : t \in T_\alpha, \\ \text{cond}(t)=\text{true} \\ \wedge \text{source}(t) \in \mathbf{occs} \\ \wedge \text{targets}(t) \subseteq \mathbf{occs} \cup \mathbf{vacs} \\ \wedge (\text{orbit}(t) \in \mathbf{occs} \cup \mathbf{vacs} \vee \text{orbit}(t) = \emptyset) \}$$

$T_{f,\alpha,false}$  is the set of all transitions where the associated pre-requisites and conditions are false.

$$T_{f,\alpha,false} = T_\alpha \setminus T_{f,\alpha,true}$$

$S_{f,\alpha,true}$  is the set of *source states* of transitions on the event under consideration for which at least one associated transition condition is true:

$$S_{f,\alpha,true} = \text{sources}(T_{f,\alpha,true})$$

$T_{f,\alpha,true}^s$  is the set of transitions from source state  $s$  where the associated pre-requisites and conditions are true:

$$T_{f,\alpha,true}^s = \{ t \in T_{f,\alpha,true} \mid \text{source}(t)=s \}$$

$S_{f,\alpha,qual}$  is the set of source states of transitions on the event under consideration for which at least one associated transition condition is true, and for which the source state qualifies under the hierarchy prioritisation algorithm. A state qualifies if there is no transition with a true condition (on the same event) having a source state hierarchically below<sup>1</sup> it.

---

<sup>1</sup> In an alternative prioritisation: *above*

$$S_{f,\alpha,qual} = \{ s \in S_{f,\alpha,true} \mid \text{descendants}^1(s) \cap S_{f,\alpha,true} = \emptyset \}$$

$T_{f,\alpha,qual}$  is the set of all transitions where the associated pre-requisites and conditions are true and which qualify under the hierarchy prioritisation algorithm

$$T_{f,\alpha,qual} = \{ t \in T_{f,\alpha,true} \mid \text{source}(t) \in S_{f,\alpha,qual} \}$$

$T_{f,\alpha,disq}$  is the set of all transitions where the associated pre-requisites and conditions are true but which are disqualified by the hierarchy prioritisation algorithm

$$T_{f,\alpha,disq} = T_{f,\alpha,true} \setminus T_{f,\alpha,qual}$$

Qualifying transitions come from the outermost statechart layer(s) containing true transitions. This could be regarded as an exercise to

- Find the innermost<sup>2</sup> layer of the hierarchy that has at least one true transition
- All true transitions from this layer are qualifying
- All true transitions above<sup>3</sup> this layer are disqualified

$T_{f,\alpha,qual}^s$  is the set of all transitions *from source state*  $s$  where the associated pre-requisites and conditions are true and which qualify under the hierarchy prioritisation algorithm

$$T_{f,\alpha,qual}^s = \{ t \in T_{f,\alpha,qual} \mid \text{source}(t) = s \}$$

$T_{f,\alpha,qual}^*$  is the set of sets  $T_{f,\alpha,qual}^s$ , for all states  $s$  in  $S_{f,\alpha,qual}$ . Each member set contains all qualifying transitions from the same qualifying source state.

$$T_{f,\alpha,qual}^* = \{ T_{f,\alpha,qual}^s \mid s \in S_{f,\alpha,qual} \}$$

Since different elements of  $T_{f,\alpha,qual}^*$  contain transitions from different source states, they are disjoint:

$$\tilde{\cap} T_1, T_2 \in T_{f,\alpha,qual}^* \times T_1 \not\approx T_2 = \emptyset$$

$T_{f,\alpha,qual}^\times$  is the set of sets where each element of  $T_{f,\alpha,qual}^\times$  is formed by taking one element from each element of  $T_{f,\alpha,qual}^*$ . (It is rather like a distributed cartesian product, but it is a set of sets, not a set of tuples). Each element of  $T_{f,\alpha,qual}^\times$  contains a qualifying transition from each qualifying source-state. There is as yet no notion of orderings of transitions. These elements represent *fork* nondeterminism.

$$T_{f,\alpha,qual}^\times = \{ T \in \text{PT}_{f,\alpha,qual} \mid (\tilde{\cap} T_1 \in T, \tilde{\cap} T_2 \in T_{f,\alpha,qual}^* \times \#(T_1 \not\approx T_2) = 1) \}$$

Here, # is used to denote the size of a set.

$T_{f,\alpha,exec}$  is the set of sequences formed by replacing each set in  $T_{f,\alpha,qual}^\times$  by sequences covering every *permutation* (i.e. ordering) of the replaced set. So each sequence

<sup>1</sup> In the alternative prioritisation: *ancestors*

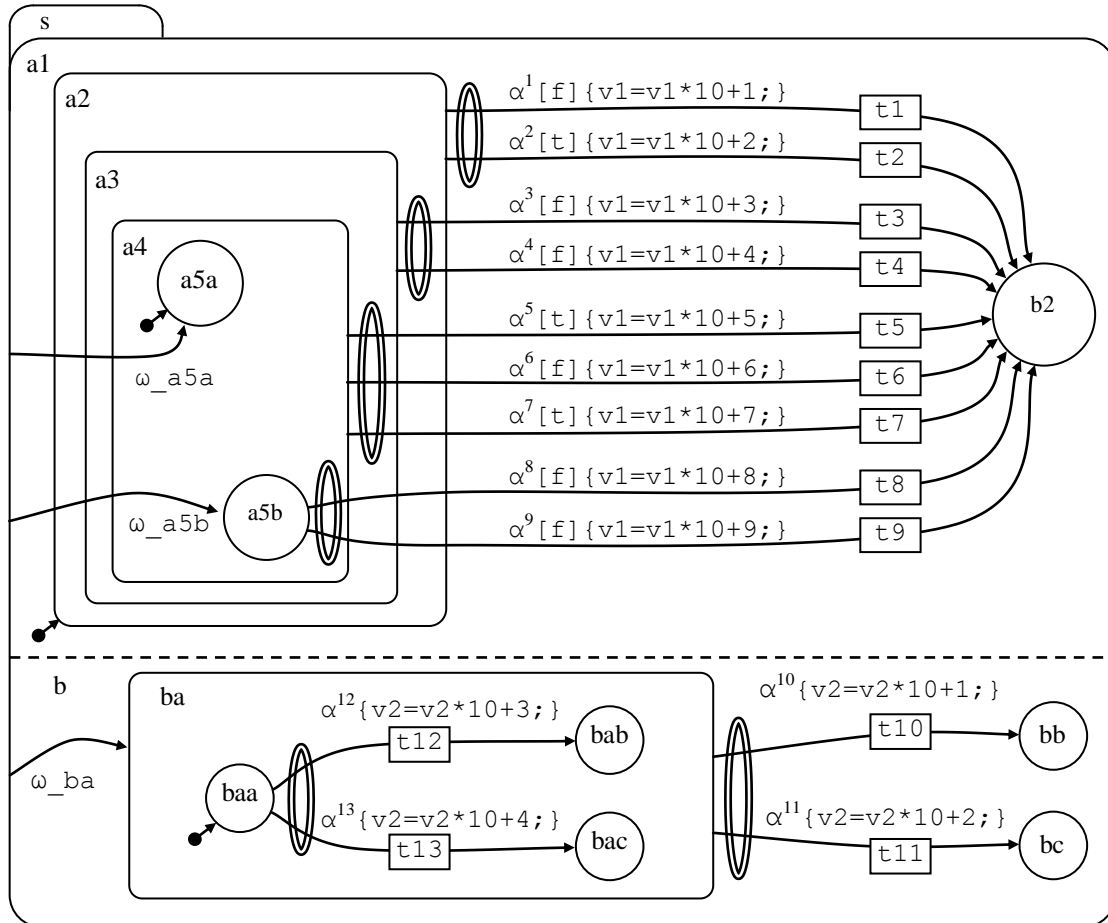
<sup>2</sup> In the alternative prioritisation: *outermost*

<sup>3</sup> In the alternative prioritisation: *below*

contains an ordering of a qualifying transition from each qualifying source-state. These sequences represent *fork and race* nondeterminism.

$$T_{f,a,exec} = \{ seq_i \in Perm(tup_j) \mid tup_j \in T_{f,a,qual}^x \}$$

Set transit nondeterminism is not part of transition *selection*; it is handled within the transition *processing* algorithm.



**Figure 113. Transition derivatives example (similar to test model  $\tau 6240$ )**

Notes:

1. There is just one *event*  $\alpha$  - the superscript identifies *transitions* on  $\alpha$ .
2.  $[t]$  stands for a true condition,  $[f]$  for a false one.
3. To illustrate the alternative prioritisation scheme, we would have  $\alpha^9[t], \alpha^2[f]$ .

As an example, given the statechart in Figure 113, assuming leafstates *a5b* and *baa* are occupied, event  $\alpha$  leads to the following quantities:

$T_{\alpha}$	$\{t1, t2, t3, t4, t5, t6, t7, t8, t9, t10, t11, t12, t13\}$
$T_{f,\alpha,true}$	$\{t2, t5, t7, t10, t11, t12, t13\}$
$T_{f,\alpha,false}$	$\{t1, t3, t4, t6, t8, t9\}$
$S_{f,\alpha,true}$	$\{a2, a4, ba, baa\}$
$T_{f,\alpha,true}^{a2}$	$\{t2\}$
$T_{f,\alpha,true}^{a4}$	$\{t5, t7\}$
$T_{f,\alpha,true}^{ba}$	$\{t10, t11\}$
$T_{f,\alpha,true}^{baa}$	$\{t12, t13\}$
$S_{f,\alpha,qual}$	$\{a4, baa\}$
$T_{f,\alpha,qual}$	$\{t5, t7, t12, t13\}$
$T_{f,\alpha,disq}$	$\{t2, t10, t11\}$
$T_{f,\alpha,qual}^{a4}$	$\{t5, t7\}$
$T_{f,\alpha,qual}^{baa}$	$\{t12, t13\}$
$T_{f,\alpha,qual}^*$	$\{ \{t5, t7\}, \{t12, t13\} \}$
$T_{f,\alpha,qual}^x$	$\{ \{t5, t12\}, \{t5, t13\}, \{t7, t12\}, \{t7, t13\} \}$
$T_{f,\alpha,exec}$	$\langle \{t5, t12\}, \{t12, t5\}, \{t5, t13\}, \{t13, t5\}, \langle t7, t12 \rangle, \langle t12, t7 \rangle, \langle t7, t13 \rangle, \langle t13, t7 \rangle \rangle$

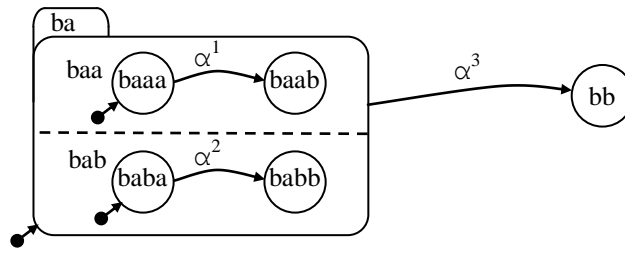
*STATECRUNCHER's transition selection algorithm* is to select all transition sequences in  $T_{f,\alpha,exec}$ .

For each sequence, a new *world* (or more than one) can result after execution of the sequence. As will be seen, worlds are not created in advance of processing each sequence, but rather are created deeper in the algorithm where each individual transition is processed when it needs to change the configuration.

## 7.4 Discussion of hierarchical fork nondeterminism

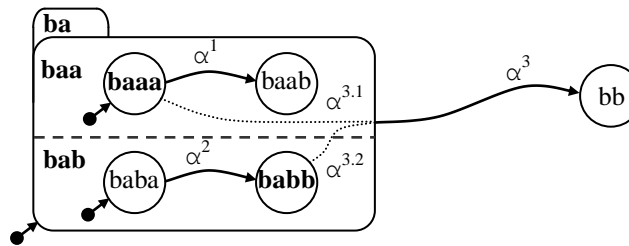
As mentioned, when there are transitions on the same event at different hierarchical levels, STATECRUNCHER applied the UML-conformant policy of specialisation, whereby inner transitions take precedence over outer ones. We consider here what procedure would best be followed if hierarchical prioritisation is replaced by fork nondeterminism across different hierarchical levels, which we call *hierarchical fork nondeterminism*.

It must first be decided what is meant by event  $\alpha$  in the figure below.



**Figure 114. Hierarchical ambiguity (1)**

The nondeterminism lies in choosing either ( $\alpha^1$  and  $\alpha^2$ ) or just  $\alpha^3$  (with set-transit consequences). We do *not* allow combinations such as  $\alpha^1$  and  $\alpha^{3.2}$  in Figure 115 below (where  $\alpha^{3.2}$  is considered a logical component of  $\alpha^3$ ).



*bold font in a leafstate name indicates an occupied state*

**Figure 115. Hierarchical ambiguity (2)**

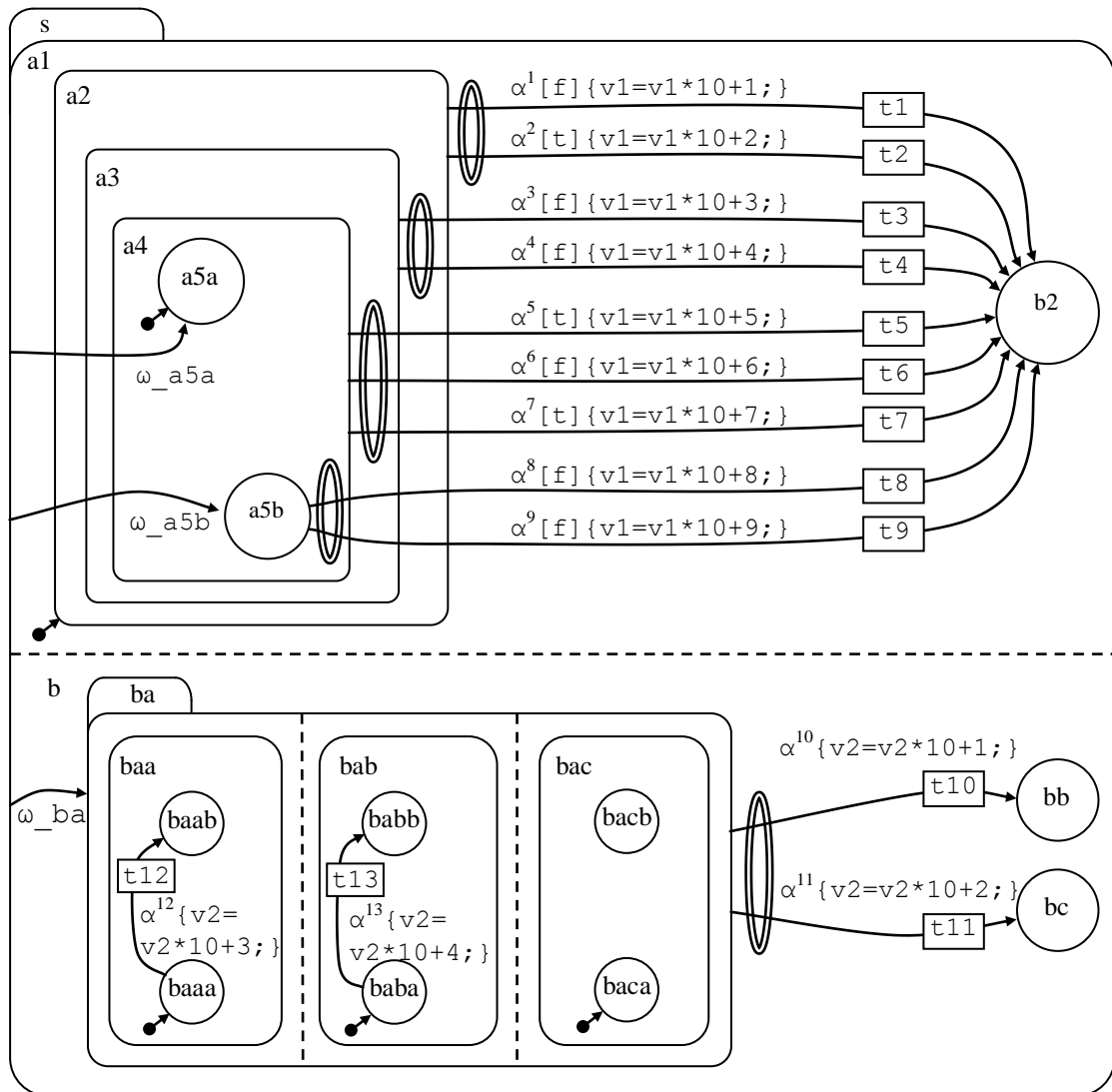
For more general state structures containing many nested clusters and sets, we organize transitions into groups originating from source states which are *members of the same cluster or set* and groups which stand in *hierarchical* relationship to one another. We wrap and mark sibling source states in a set with a **from-each** tag, indicating that a transition must be taken from each source state. We wrap and mark source states in a hierarchical relationship with a **from-one** tag. Sometimes there will only be one state in a **from-each** or **from-one** package, making the issue irrelevant, but in the examples, we show the tag anyway.

We create a quantity  $S_{f,\alpha,true}$  to represent this; for Figure 114 above, this would be

```
[ from-one,
  [ from-each, baaa, baba ],
  [ from-one, ba ] ] .
```

As a more extensive example, we take the following model, which is similar to the previous one, but with more depth of hierarchy.





**Figure 116. Model to illustrate hierarchical fork nondeterminism**

Hierarchical fork nondeterminism, as a variation on hierarchical prioritisation, will generate a new  $S_{f,\alpha,qual}$  set, from which a new  $T_{f,\alpha,qual}$  and  $T_{f,\alpha,exec}$  set can be constructed analogously to the previous algorithm.

$S_{f,\alpha,true} =$

```
[ from-each,
  [ from-one, a2, a4 ],
  [ from-one,
    [ from-one, ba ],
    [ from-each, baaa, baba ] ] ]
```

Next, per source state, we substitute all transitions from it, with a *from-one* tag, giving

$T_{f,\alpha,true}^*$  =

```
[from-each,
  [from-one, [from-one, $\alpha^2$ ], [from-one,  $\alpha^5$ ,  $\alpha^7$ ] ],
  [from-one,
    [from-one, $\alpha^{10}$ , $\alpha^{11}$ ],
    [from-each, [from-one, $\alpha^{12}$ ], [from-one, $\alpha^{13}$ ] ] ] ]
```

This tree can be walked according to the tagged instruction, with example PROLOG code shown following.

### *PROLOG code for an each-one walker*

```
/* Each/One-walker data */
eodata1(X):-
  X= [from_each,
      [from_one,
        [from_one,a2],
        [from_one,a5,a7]],
      [from_one,
        [from_one,a10,a11],
        [from_each,
          [from_one,a12],
          [from_one,a13]]]].

/* Each/One Walker */
eowalk(X,X):-
  atom(X).

eowalk([from_one|T],X):-
  gn_member(M,T),
  eowalk(M,X).

eowalk([from_each|T],X):-
  eowalks(T,X).

eowalks([],[]).
eowalks([H|T],[LH|LT]):-
  eowalk(H,LH),
  eowalks(T,LT).

/* Walk the example data */
go_eo:-
  eodata1(X),
  eowalk(X,Y),
  gn_flatten(Y,W),
  write(W),nl,
  fail.
```

The output of running `go_εo` is:

```
[a2, a10]
[a2, a11]
[a2, a12, a13]
[a5, a10]
[a5, a11]
[a5, a12, a13]
[a7, a10]
[a7, a11]
[a7, a12, a13]
```

This corresponds to:

$$T_{f,\alpha,\text{qual}}^{\times} =$$

```
{ {a2, a10},
  {a2, a11},
  {a2, a12, a13},
  {a5, a10},
  {a5, a11},
  {a5, a12, a13},
  {a7, a10},
  {a7, a11},
  {a7, a12, a13} }
```

Permuting the transitions, we obtain the sequences we wish to execute:

$$T_{f,\alpha,\text{exec}} =$$

```
{
<t2, t10>, <t10, t2>,
<t2, t11>, <t11, t2>,
<t2, t12, t13>, <t2, t13, t12>, <t12, t2, t13>, <t12, t13, t2>, <t13, t2, t13>, <t13, t12, t2>,
<t5, t10>, <t10, t5>,
<t7, t11>, <t11, t7>,
<t5, t12, t13>, <t5, t13, t12>, <t12, t5, t13>, <t12, t13, t5>, <t13, t5, t13>, <t13, t12, t5>,
<t7, t10>, <t10, t7>,
<t7, t11>, <t11, t7>,
<t7, t12, t13>, <t7, t13, t12>, <t12, t7, t13>, <t12, t13, t7>, <t13, t7, t12>, <t13, t12, t7>
}
```

## 7.5 Transition course

### 7.5.1 Effective transitions

A transition arc (including bifurcations) indicates one source state and one or more target states. In general, the transition arc does not indicate leafstates at either end, and these must be determined by some algorithm. The *transition course* is the actual sequence of states exited and entered, and can be indicated by an *effective* transition arc, which we show by a dotted line in the figures below. A requirement is that, in the absence of orbital transitions, the transition should be as “low flying” as possible, i.e. it should not exit and enter any states unnecessarily.

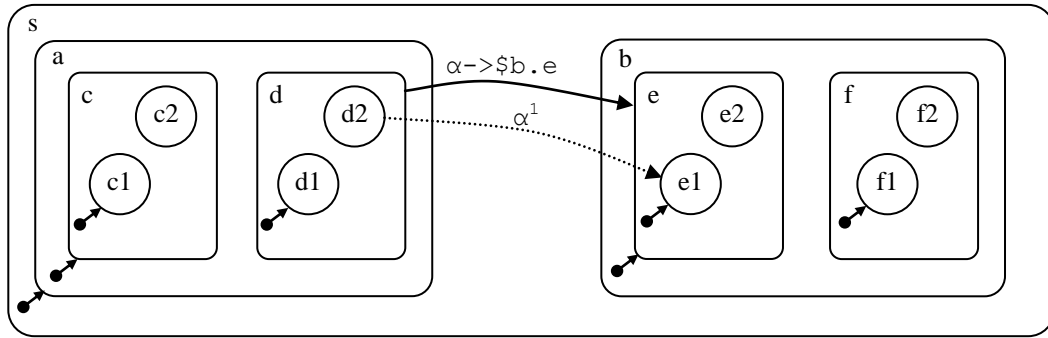
The algorithm to find the effective transition arc basically involves:

- Determining the *enter tree scope* and *exit tree scope*. These are sometimes (but not always), identical, and might be the common ancestor of the source and target states of the transition. The reason these scopes are needed is given below.
- Constructing an *intermediate exit tree* to the exit tree scope level.
- Constructing an *intermediate enter tree* to the enter tree scope level.
- Removing common states between the enter and exit trees, (but not necessarily so when the transition is orbital). The reason for this operation is that effective transitions are as low flying as possible, which means that the exit and enter trees must not take the transition to an unnecessary height. An example of a low-flying transition with higher level intermediate exit and enter trees is the transition on  $\alpha_1$  in Figure 120, (given the occupancy configuration shown in the figure). After common state removal, the highest level remaining is called the *altitude* of the transition. The residual exit and enter trees are called the *definitive exit tree* and the *definitive entry tree*.

The algorithm is explained in more detail in this section.

The scopes are needed, because without them, we would have to exit to statechart level, and we could then be exiting set members that are not involved in the transition. Constructing the enter tree would then be more difficult, because we would have to re-enter states that really never should have been considered for exit, when we want to concentrate on entering states because the transition demands it. Moreover, we would have to ensure that such states are never *actually* exited and re-entered. It would also be inefficient to work with exit and entry trees to statechart level if this is not necessary.

Figure 117 below indicates how a transition on event  $\alpha$  might effectively correspond to the transition marked by  $\alpha^1$ . The tail and tip of the transition arc explicitly give states to be exited and entered.



**Figure 117. Transition arc and effective transition**

### *Deep History*

The effect of **deep history** is to ensure that when a decision must be taken as to which member to enter of a cluster that is under deep history, the *member that was last occupied* is entered. We call this the *historical member*. If no member has ever been entered, or the record of the history has been cleared, the default member is taken. If the cluster is already occupied, the currently occupied member is regarded as the target member.

A cluster member is liable to be “under deep history” if there is an ancestor set or cluster that has a deep history marker. We shall see that there are nevertheless circumstances when we do not regard a transition entry step as being under the dominion of an ancestral deep history marker.

Deep history ensures that after an ‘excursion’ from a cluster, such as the excursion from  $\tau$  to  $c$  and back again in Figure 118 below, defined by the transitions on  $\alpha$  and  $\beta$ , the original cluster is back in its original state.

However, a transition such as the one on event  $\gamma$  below does not ‘see’ the deep history, since no member of the cluster with the deep history marker,  $\tau$ , undergoes an enter operation. This is the behaviour we want; the local behaviour in clusters  $a$ ,  $c$  and  $d$  should not be altered by an outer wrapper such as  $\tau$ .

The orbital transition on  $\delta$ , however, does see deep history, because it actually enters cluster  $\tau$ , which is marked with deep history, since cluster  $\tau$  is *below* its orbital level.

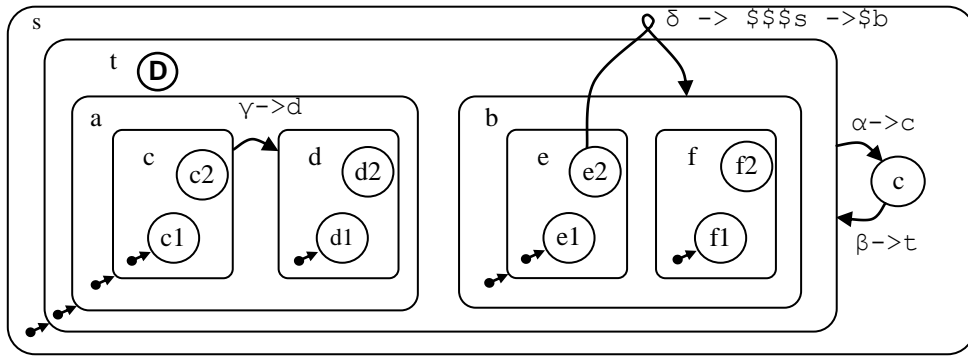
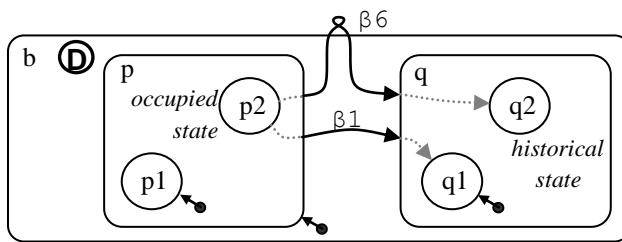
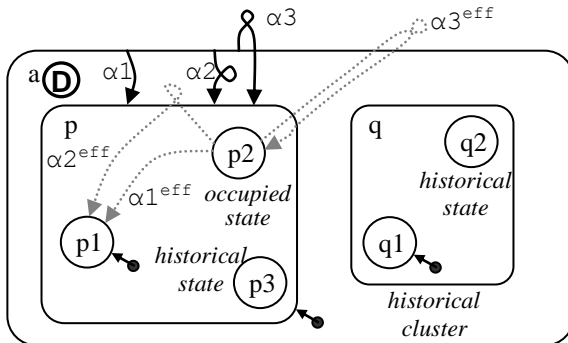


Figure 118. Deep history illustration



Deep history is only "seen" if the effective transition arc actually enters the deep history cluster.

Figure 119. Deep history illustrations - effective transitions (1)



In STATECRUNCHER, history is recorded on cluster exit. So it is still present on subsequent cluster entry.

We therefore take the **current cluster** to act as a more recent equivalent to history than the formal **historical cluster**, when dealing with a target cluster that is already occupied.

Figure 120. Deep history illustrations - effective transitions (2)

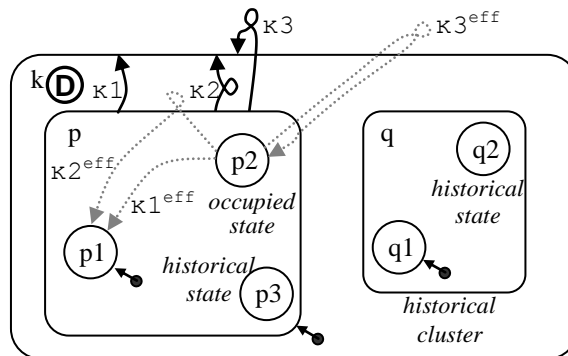


Figure 121. Deep history illustrations - effective transitions (3)

Now the issue in finding the transition course in a statechart with deep history appears at first sight to be a chicken and egg problem:

- to find the *transition course*, we need, amongst other things, the *enter tree*
- constructing the *enter tree* depends on knowing when to apply *deep history*
- knowing when to apply *deep history* depends on knowing whether a particular cluster will *actually* be entered in the effective transition
- knowing whether a particular cluster will *actually* be entered depends on knowing the *transition course*.

Despite the apparent circular reasoning, it is possible to find a satisfactory algorithm.

The algorithm parameters available to control the transition course are as follows:

- *Exit/enter tree scope logic* is based on transition source and transition target states, and orbital state
- *Enter tree construction logic* is based on explicit target states, history markers, target occupancy, and the orbital state.

The intermediate **exit tree** is created by (recursively) exiting the highest level in the exit tree scope then the child of each state exited. When a member of a set is exited, sibling set members are also exited.

The intermediate **entry tree** is more difficult to construct; details follow in Section 7.5.3.

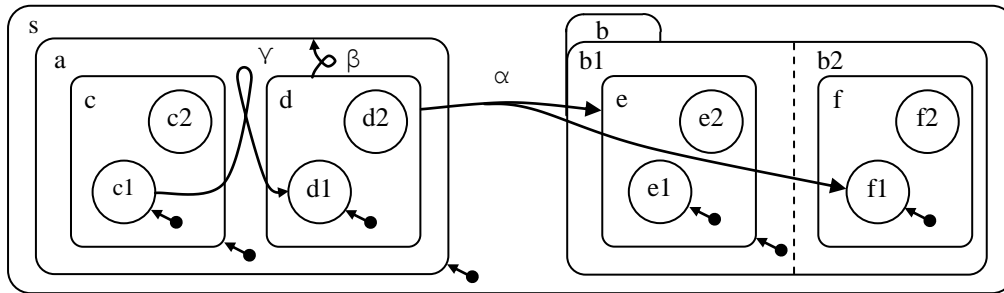
## 7.5.2 Logic for exit and enter tree scopes

### *Terminology for the decision logic*

Given a transition, with its source state, orbit and target states, we may refer to:

- the *transition common ancestor, TnCA* (the common ancestor of the source state and all target states)
- the *target common ancestor, TgCA* (the common ancestor of all target states)
- the source-side child of transition common ancestor
- the target-side child of transition common ancestor
- the source-side child of orbit
- the target-side child of orbit

Not all conceivable algorithms require all these terms. Examples of the terms are given with reference to Figure 122.



**Figure 122. Model for reference to transition common ancestor and related concepts**

For the transition on  $\alpha$ :

- the **transition common ancestor** is the common ancestor of cluster  $d$ , cluster  $e$  and leafstate  $f1$ , which is **cluster  $s$**
- the **target common ancestor** is the common ancestor of cluster  $e$  and leafstate  $f1$ , which is **set  $b$**

For the transition on  $\beta$ :

- the **transition common ancestor** is the common ancestor of cluster  $d$  and cluster  $a$ , which is **cluster  $a$**

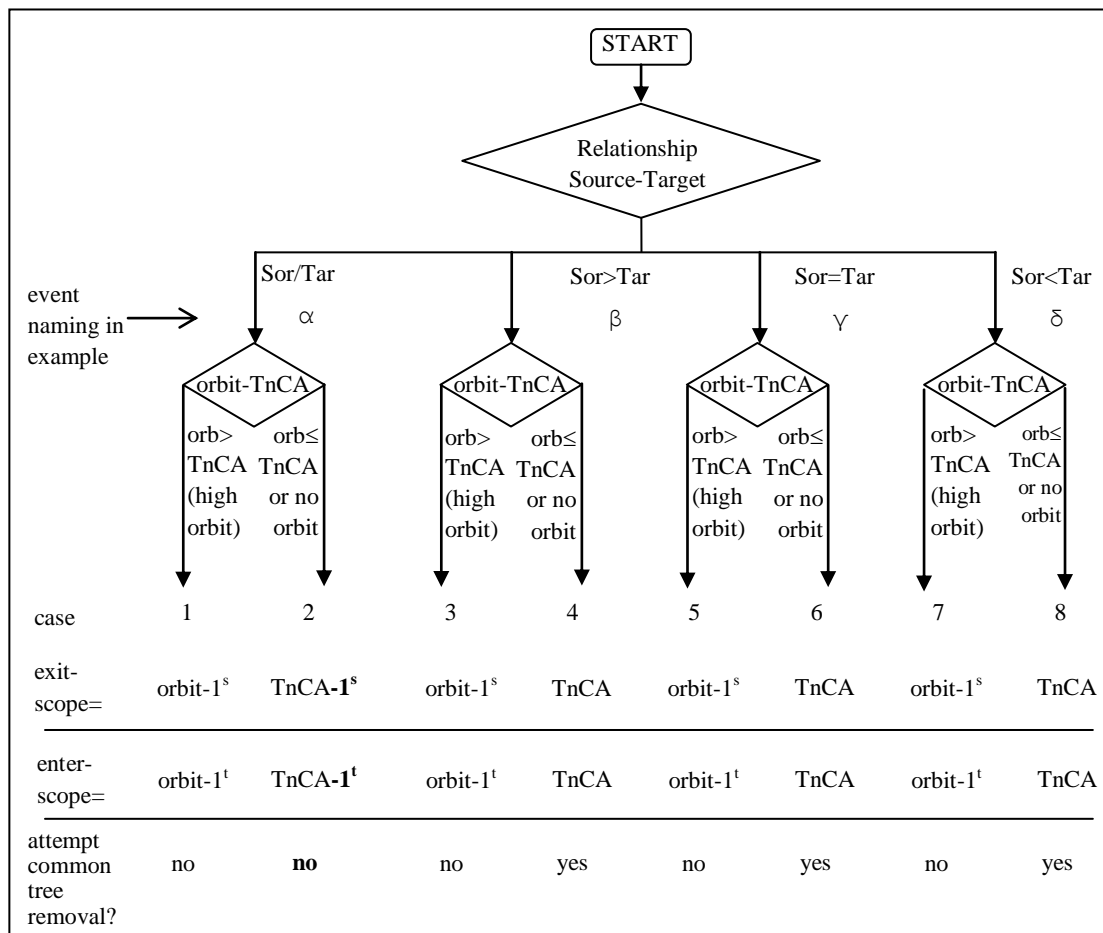
For the transition on  $\gamma$ :

- the **transition common ancestor** is the common ancestor of leafstate  $c1$  and leafstate  $d1$ , which is **cluster  $a$**
- the source side child of transition common ancestor is cluster  $c$
- the target side child of transition common ancestor is cluster  $d$
- the orbit is cluster  $a$
- the source side child of orbit is cluster  $c$
- the target side child of orbit is cluster  $d$

The logic for the scope of the intermediate exit and enter trees is given in Figure 123. Legend for that figure (not all terms necessarily used in the current algorithm):

- Sor = Source state of transition
- Tar = Target state of transition, or common ancestor of target states if there are several
- TnCA = Transition common ancestor
- orb = orbital state
- $A > B$  reads "A is a strict ancestor of B." [A is greater in age, as it were].
- $A < B$  reads "A is a strict descendant of B."
- $A / B$  reads "A and B are not in a direct ancestral line."
- $A-1$  reads "a child of A".
- $A-1^s$  reads "the child of A on the source side of the transition, i.e. the active child of A."
- $A-1^t$  reads "the child of A on the target side".





**Figure 123. Decision logic for scopes**

Notes

1. By target scope, we mean the common ancestor of all targets.
2. The above logic could be exhibited in a more condensed form, but as it stands, it brings out separate cases more explicitly, making it easier to review the logic. In particular, cases 1,3,5,7 (orbital cases) condense, as do cases 4,6,8 (line-of-descent cases).
3. For all orbital cases, the enter and exit trees have exit and enter scope of orbit-1<sup>s</sup> and orbit-1<sup>t</sup> respectively. Where the source and target are in the same line of descent, the enter and exit scope given will necessarily be the same for each.
4. In case 2 we have TnCA-1<sup>s</sup> and TnCA-1<sup>t</sup>, and that we do not attempt to remove any common tree, because we know there is no common tree.
5. Where there is a *low* orbit, it will have the effect of limiting the amount of common tree removal. Where there is *no* orbit, the maximum amount possible of common tree removal will take place.

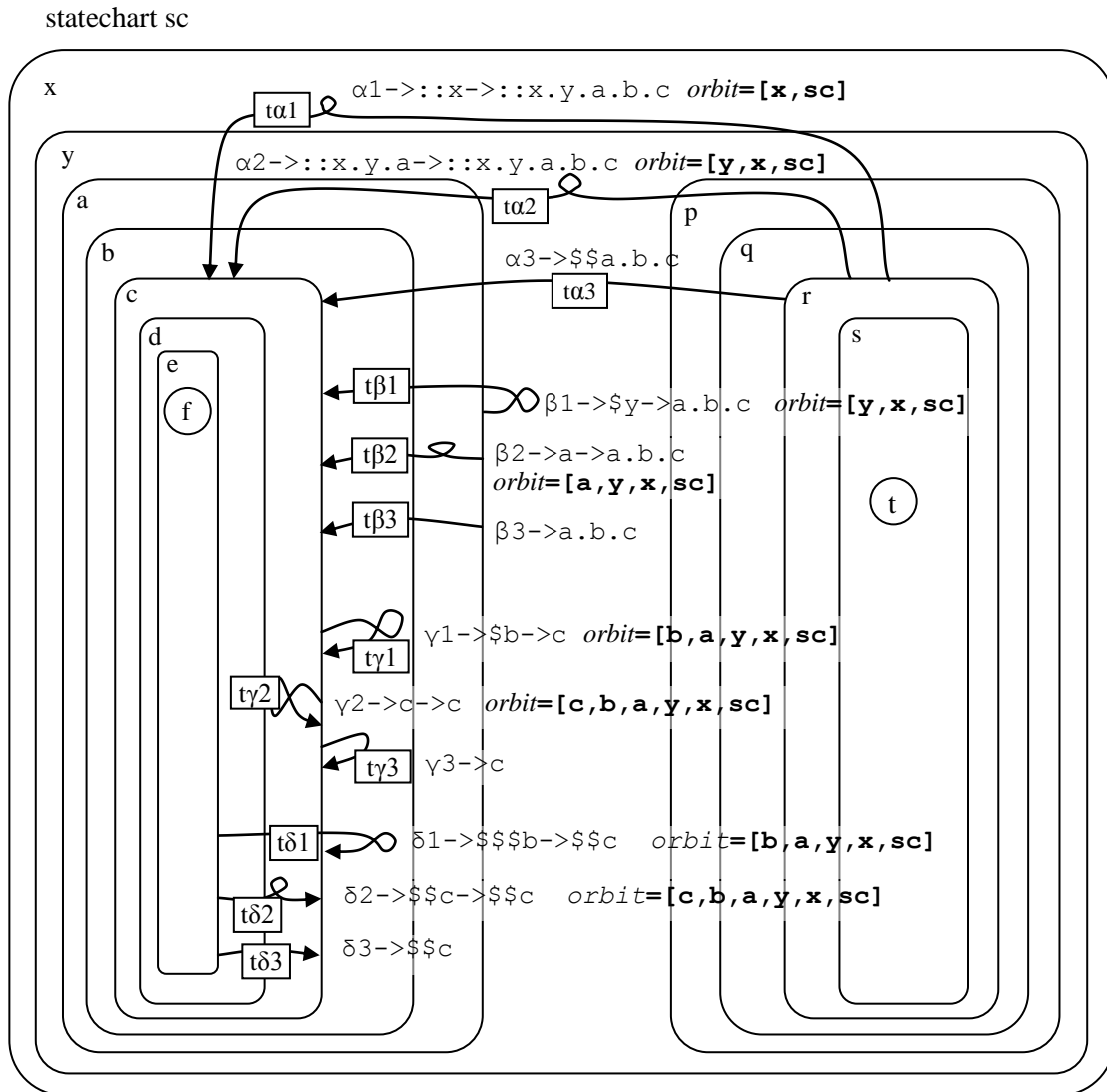


Figure 124. Scope of enter/exit trees

Notes

1. All the above transitions terminate on state c.
2. The  $\alpha 1/\beta 1/\gamma 1/\delta 1$  events are high orbital, the  $\alpha 2/\beta 2/\gamma 2/\delta 2$  events are lower orbital, and the  $\alpha 3/\beta 3/\gamma 3/\delta 3$  events are non-orbital.

Examples from the above figure:

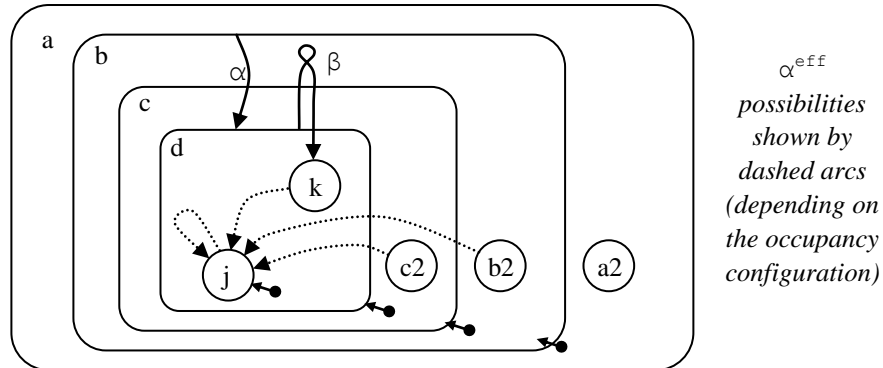
Case	Transition	Orbit	TnCA	Key Properties	Intermed. exit scope	Intermed. enter scope	Attempt common tree removal?	Remarks
1	tα1	x	y	Sor/Tar orbit>TnCA (high orbit)	orbit-1 <sup>s</sup> = x-1 <sup>s</sup> = y	orbit-1 <sup>t</sup> = x-1 <sup>t</sup> = y	No	Orbit <i>above</i> common ancestor Orbit-child scopes No common tree removal.
1	tα2	y	y	Sor/Tar orbit=TnCA (high orbit)	TnCA-1 <sup>s</sup> y-1 <sup>s</sup> =p	TnCA-1 <sup>t</sup> y-1 <sup>t</sup> =a	No	Orbit <i>at</i> common ancestor Common tree removal would fail if attempted
2	tα3	-	y	Sor/Tar no orbit	TnCA-1 <sup>s</sup> y-1 <sup>s</sup> =p	TnCA-1 <sup>t</sup> y-1 <sup>t</sup> =a	No	Common tree removal would fail if attempted
3	tβ1	y	a	Sor/Tar orbit>TnCA (high orbit)	orbit-1 <sup>s</sup> = y-1 <sup>s</sup> = a	orbit-1 <sup>t</sup> = y-1 <sup>t</sup> = a	No	Orbit <i>above</i> common ancestor Orbit-child scopes No common tree removal.
4	tβ2	a	a	Sor>Tar orbit<TnCA (low orbit)	TnCA =a	TnCA =a	Yes, <i>but...</i>	Orbit <i>at</i> common ancestor Transition common ancestor scope <b><i>ORBIT will restrict common tree removal</i></b>
4	tβ3	-	a	Sor>Tar no orbit	TnCA = a	TnCA =a	Yes	Transition common ancestor scope Common tree will remove c (at least) Note that c's history marker will be seen.
5	tγ1	b	c	Sor=Tar orbit>TnCA (high orbit)	orbit-1 <sup>s</sup> = b-1 <sup>s</sup> = c	orbit-1 <sup>t</sup> = b-1 <sup>t</sup> = c	No	Orbit <i>above</i> common ancestor Orbit-child scopes No common tree removal
6	tγ2	c	c	Sor=Tar orbit=TnCA (low orbit)	TnCA = c	TnCA = c	Yes, <i>but...</i>	Orbit <i>at</i> common ancestor Transition common ancestor scope <b><i>ORBIT will restrict common tree removal</i></b>
6	tγ3	-	c	Sor=Tar no orbit	TnCA = c	TnCA = c	Yes	Transition common ancestor scope Common tree will remove c (at least) Note that c's history marker will be seen.
7	tδ1	b	c	Sor<Tar orbit>TnCA high orbit	orbit-1 <sup>s</sup> = b-1 <sup>s</sup> = c	orbit-1 <sup>t</sup> = b-1 <sup>t</sup> = c	No	Orbit <i>above</i> common ancestor Orbit-child scopes No common tree removal
8	tδ2	c	c	Sor<Tar orbit=TnCA low orbit	TnCA =c	TnCA =c	Yes <i>but...</i>	Orbit <i>at</i> common ancestor Transition common ancestor scope <b><i>ORBIT will restrict common tree removal</i></b>
8	tδ3	-	c	Sor<Tar no orbit	TnCA =c	TnCA =c	Yes	Transition common ancestor scope Common tree will remove c (at least) Note that c's history marker will be seen.

**Table 11. Exit and enter tree scope examples**

### 7.5.3 Entry tree construction

#### Entry of a cluster

We first introduce the terminology *guide-mode* and *orbitality*.



**Figure 125. Guide mode and orbitality**

The transition on  $\alpha$  in Figure 125 is specified as coming from cluster b, but this is a non-leafstate; the *effective* transition source could be various leafstates within cluster b: j, k, c2, or b2 (but not a2, which is not within cluster b). The target state is also specified at non-leafstate level, (cluster d), the *effective* target state always being in fact j. The transition arrow is not entirely an explicit guide for determining the effective transition. If the transition actually comes from state b2, (because state b2 is the occupied leafstate) it is clear *from the transition arrow* that clusters c and d must be entered. This is **guided entry**. If this transition comes from state c2, then cluster d will be entered as guided entry. But the final part of determining the actual transition target (leafstate j) is not explicit in the transition arrow, and will be performed as **unguided entry**.

The transition on  $\beta$  illustrates orbitality:

- Cluster b is **at-orbit**
- Cluster a is above the orbit, i.e. **super-orbital**
- Clusters c and d and leafstate k are below the orbit, i.e. **sub-orbital**
- Note that for the transition on  $\alpha$  all states in the hierarchy are qualified as **no-orbit**

The dependency factors for entering a cluster are:

- whether the cluster is entered in guide-mode = guided or unguided
- whether the cluster history-attribute = deep history or history or no history
- whether the cluster history-availability = available or unavailable
- whether the cluster is entered under a dho = deep-history-obligation (*on statechart entry set to false*) or not. This means that the historical member must be (recursively) entered if possible, due to a deep history marker having set this up in a preceding part of the transition course.

- whether the cluster orbitality = suborbital (i.e. at a level at-or-below orbital level) or superorbital or no-orbit.
- whether the *target* state occupancy = occupied or vacant

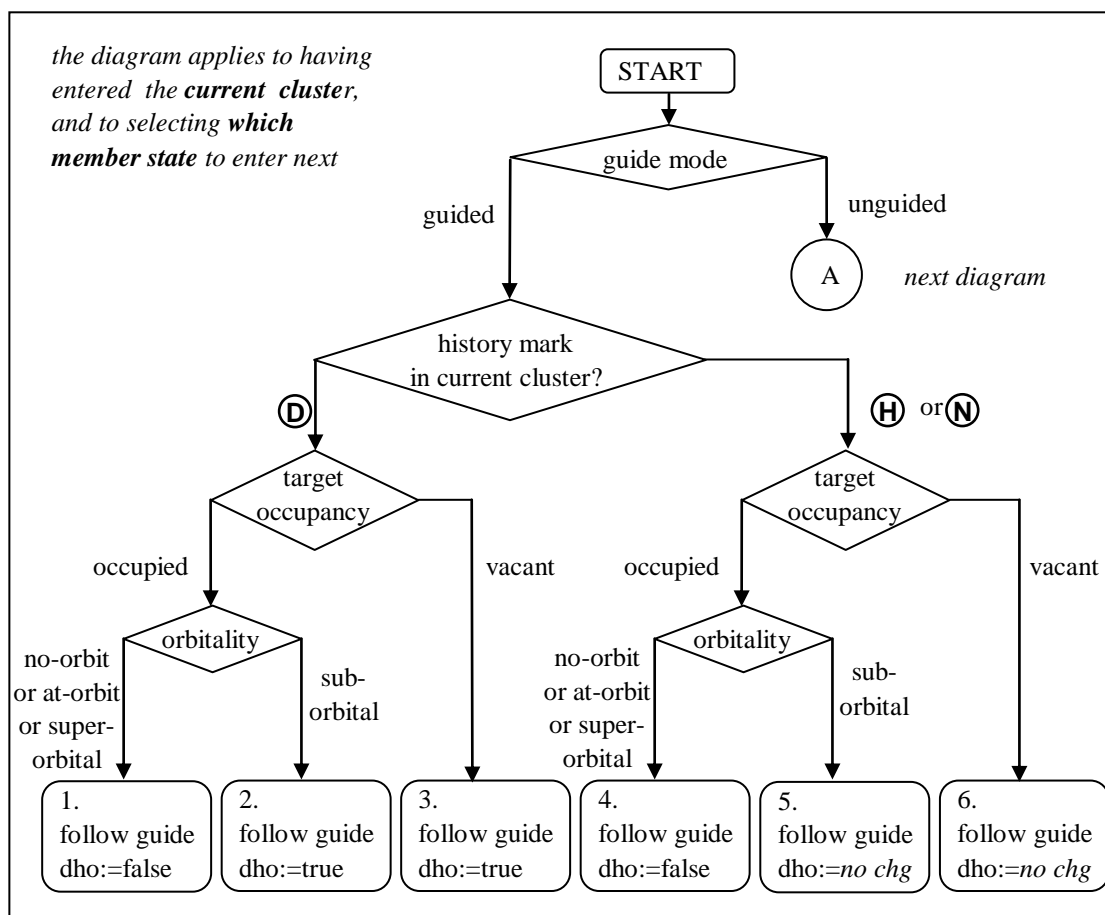
Entry of a set

This is basically as for a cluster, except that

- all members are entered
- there are typically several guides, prescribing entry into various set members.

An illustrative example is given at Figure 128.

The following figures show flow diagrams that specify which member of a cluster is entered using the above factors.



**Figure 126. Entry tree for clusters (1)**

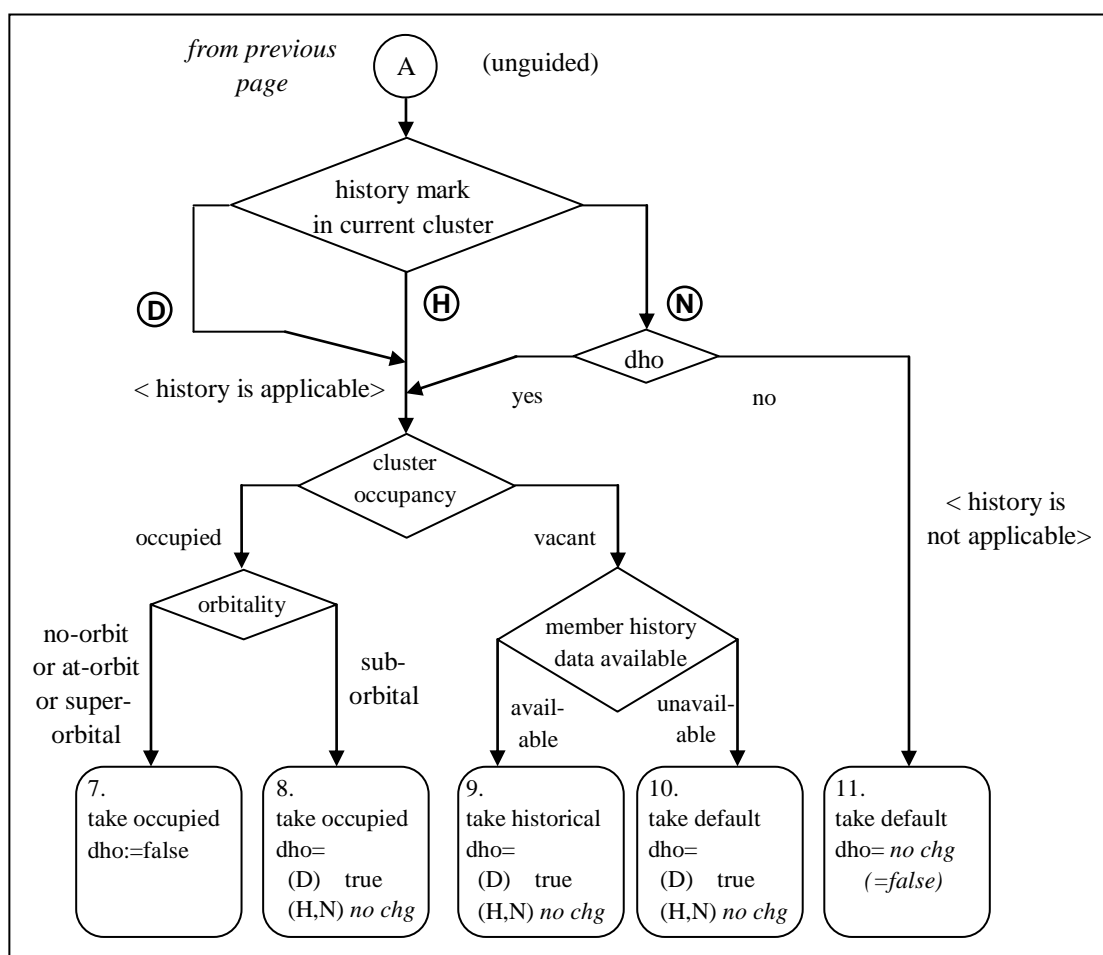
Rationale for the above

As these are all the *guided* entry mode cases, the cluster member entered will certainly be the one specified by the guide. The remaining issue is whether or not to impose a *deep history obligation (dho)* on the member state that is entered, for its (or its descendants’) use when the

guide ceases. The **dho** is imposed when a cluster is entered with a deep history marker, but it can be cancelled. Cancellation takes place (cases 1 and 4 in Figure 126) when a cluster is entered which has the property that *both source and target state belong to it* (the transition being local to the cluster, and the deep history being inapplicable) – providing there is no orbit that takes the transition above the cluster being entered. An example of cancellation of the dho taking place is the transition on  $\alpha 1$  in Figure 120.

If a cluster is entered which is vacant, or which is occupied but sub-orbital, then it is known that this entry step will form part of the effective transition, as the transition cannot be more local. In these cases, a deep history marker will set up a dho, and an existing dho will be imposed on the member cluster.

Unguided entry:



**Figure 127. Entry tree for clusters (2)**

Rationale for the above

The first issue is to determine which member state is to be entered. We first determine whether history is applicable: this is the case if there is a **(D)** or **(H)** history marker, or if a

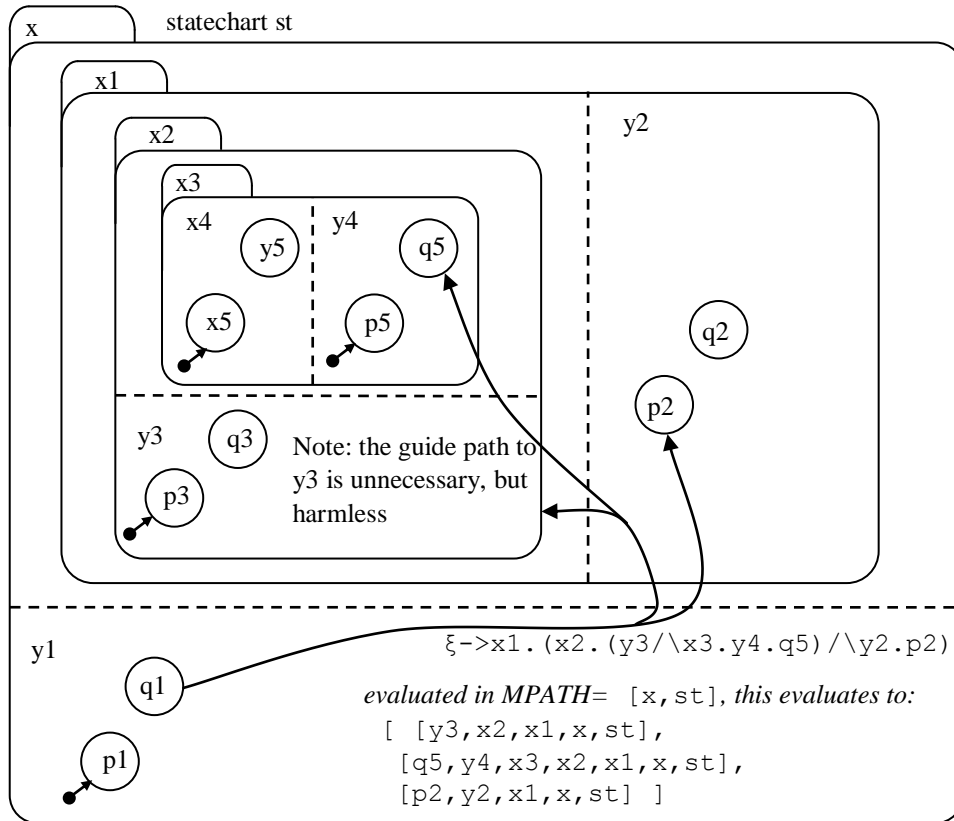
deep history obligation (dho) has been imposed. Having established *applicability* of history, we regard a *currently occupied* member state as the intended target, i.e. the *present* as overriding *history*, or to put it another way, the currently occupied state is the last entered state and so *is* the historical state.

This is the correct choice of member state whether or not the *effective* transition passes through this hierarchical level. If it does, then the entry step will be reflected in the effective transition. If it does not, then the transition is at a local level in the cluster we are entering, so the enter-tree must reflect this; it will be eliminated when the intersection of the entry tree and exit tree is taken to form the common tree.

To apply history to a vacant cluster requires that history data is also *available*. If it is, we take the historical state; if not, we take the default state. If history is not even *applicable*, we ignore historical state information and take the default state (even if another member is currently occupied). In this last situation, there is no dho, and this is maintained that way. The fact that the cluster is vacant implies it will be entered in the *effective* transition too.

The second issue is the deep history obligation (dho) setting. The dho is imposed, maintained or cancelled employing the same considerations as those given under guided entry relating to Figure 126.

**Example for a set**



**Figure 128. Example course for sets**

The rules as given for a cluster apply, but with the following extra provisions:

- If any member of a set is exited, the entire set must be exited. So if the transition altitude would otherwise be a member of a set, the whole set must be exited recursively upwards in the hierarchy until a *non-set* state is found (cluster or statechart).
- There can be multiple target states. Construction of the entry tree involves following all multiple target states (as far as they go), then relying on history and default settings. If any one member of a set is entered, all members must of course be entered, be it guided by a target state or relying on history and default states.

When members of a set (or a member of a cluster) are entered under guided entry, *all* elements of the guide-list that can be consumed, must be consumed as entry takes place. The guide list will be supplied to each member entered, and irrelevant elements in it for each particular member can be discarded. Sometimes (when we are about to commence unguided entry) an entry may be reduced to the empty list; such entries can also be discarded. An example of this is the guide-path as far as y3 in the figure above. The paths of the target of the transition on  $\xi$  above, as the entry progresses, would be

Initial guide-paths, in set x, as we are about to enter members x1 and y1



[ [y3, x2, x1], [q5, y4, x3, x2, x1], [p2, y2, x1] ]

After entering member x1 the guide-paths are:

[ [y3, x2], [q5, y4, x3, x2], [p2, y2] ]

In parallel, member y1 is entered, but the guide paths do not apply, and will be ignored or effaced.

From set x1, members x2 (a set), and y2 (a cluster) must be entered.

After entering set x2, this set retains guide paths as follows (irrelevant ones struck through)

[ [y3], [q5, y4, x3], ~~[p2, y2]~~ ]

From x2 we enter x3 and y3.

In x3, we retain one guide path in the list which is as follows.

[ [q5, y4] ]

In y3, which we must enter anyway, we retain an empty path, which can be effaced from the list

[ [] ]

leaving no guide paths, represented by the empty list:

[ ]

The remaining guide-path in to y4 and q5 is followed through, being consumed as entry steps are taken.

#### 7.5.4 Common tree removal

An exit or enter tree is a nested structure of states, e.g. (simplified) [a, [b, [c]]], with the outermost layer representing the highest part of the statechart hierarchy in the tree. The process of removing common states is to examine the top of each tree for a match and if found, to peel off the outer layer from each structure (giving in our example [b, [c]]) and to repeat the operation until a difference is found. An intermediate enter tree of [a, [b, [c, [d]]]] and an intermediate exit tree of [a, [b, [e, [f]]]] would yield definitive trees of [c, [d]] and [e, [f]] respectively.

If the trees are identical, all states are removed, and the transition is effectively a self-transition.

Some nesting layers may represent sets, giving e.g. [a, [b, [c, d, e]]]. Where the two trees contain several top-level elements, representing set members, the set members are subjected to common tree removal by a recursive call for each member. This can only occur when their parents have just been removed, so guaranteeing that the enter and exit trees contain the *same* set members at this stage. The exit-tree and enter-tree orderings of set

members correspond to enable this (set members are ordered in their declaration order). As soon as the exit and enter trees differ, the removal process is complete and they become the definitive exit and enter trees.

Throughout the process, an orbital level (if present) is used in a check so that the process can terminate prematurely, as it were, if the enter and exit trees have reached the level at which no more common tree removal is permitted.

The enter and exit trees actually contain permutation markers on set members so as to support set nondeterminism (section 7.6.5), but this does not affect the test for commonality or the removal of common states.

## 7.6 Task processing

### 7.6.1 Introduction to event processing and generalisation to tasks

The algorithm presented here for processing an event, taking account of all forms of nondeterminism (as discussed) involves extensive mutual recursion at many processing levels. We take a top-down approach to event processing, generalising to a task, and leading to a highly general top-level call, which effectively abstracts away many details which are best considered at a lower level.

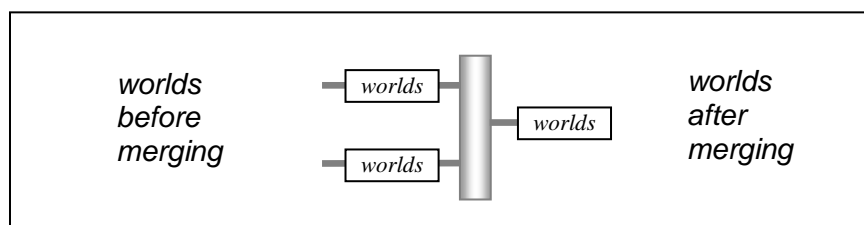
The main function of the machine engine is to process a single event. The *first* event is normally processed on the initial state, which is unique and so is represented by just *one* world. However, apart from special situations, an event is typically processed in *each of several worlds*. This is because the previous event (which may entail broadcast events, i.e. fired events and internally generated meta-events) will in general produce *many* worlds.

When one event has been processed, the resulting worlds will be needed for subsequent event processing. Any worlds representing an *earlier* situation can be destroyed, unless a record of them is required for some reason, in which case they can be retained, but they will not participate in any world merging during event processing.

When an event is processed, the transition *selection* algorithm produces *a set of transition sequences*. This is the input to the transition *execution* algorithm. We generalize a transition into a task. The most general internal STATECRUNCHER call is a call to ***process a set of sequences of tasks in a set of worlds***. We also generalize *events* and *actions* into tasks. The outer layer of our algorithm for processing sets of sequences of tasks in many worlds will be applicable to any kind of task. When we come to process one task in one world, we will identify the task and handle it with a specific handler, (a *client handler* to the more general routines).

#### ***The world merging symbol***

In the diagrams, the following symbol is used to indicate world merging:



**Figure 129. World merging diagram**

### Processing a sequence

We represent sequences of items, and sets of items, as *lists*. As is conventional in PROLOG and elsewhere, we call the first element the *head*, and the remainder, which is a list, the *tail*. The tail may contain many elements, or just one, or none at all (in which case it is a null list).

Typical recursive processing of a given list is as follows:

- Process the head using a different, lower level routine, which knows how to process the one item
- Process the tail by a recursive call to the same routine that is handling the given list
- Combine results of the processed head, and processed tail
- The termination condition of the recursion is to process the null list, and return a null list as the processed output.

The example code for *processing a task sequence in worlds*, given later in this section, illustrates this.

### An issue for any processing routine:

- In typical *parallel world* head/tail processing, where we are not concerned with a specific *sequence*, the processing order (of head and tail) may be reversed. As long as there are no side effects in the two calls, these are equivalent.

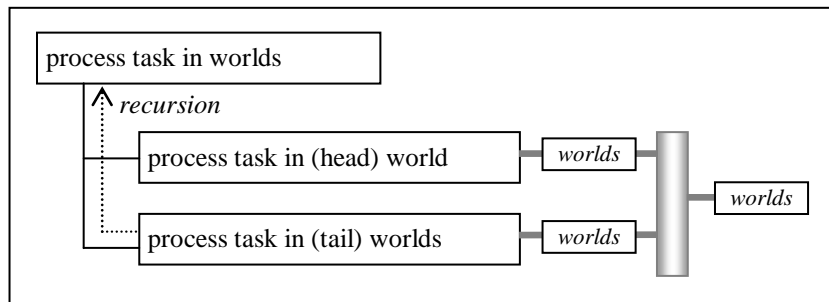


Figure 130. Process task in worlds (i)

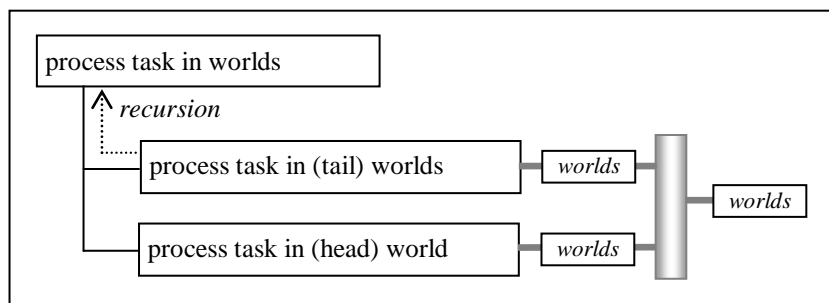
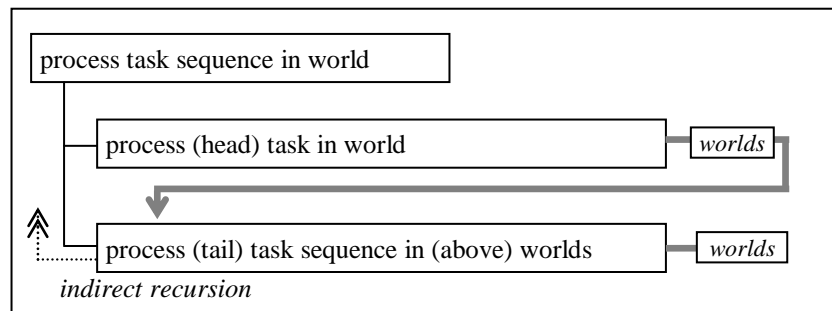


Figure 131. Process task in worlds (ii)

- Totally different is the *serial* (or *feed-forward*) case, used for *sequences*, where the output worlds of head processing feed into the tail processing.<sup>1</sup>

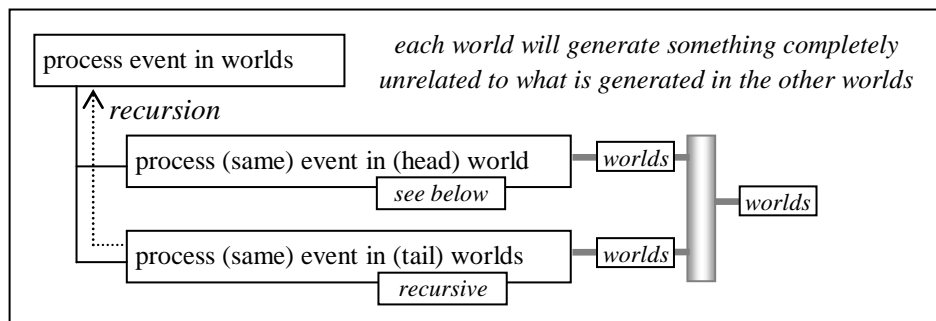


**Figure 132. Process task sequence in world**

### 7.6.2 The specific routines

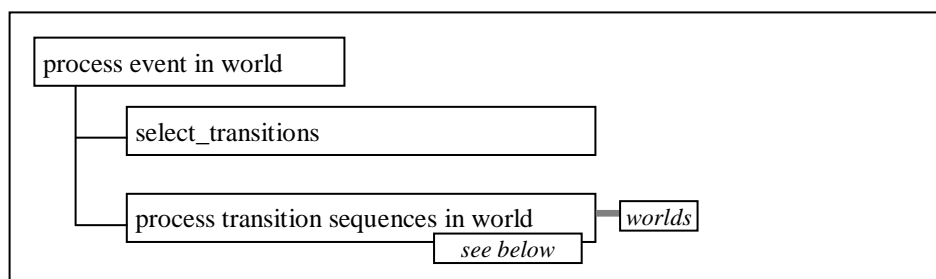
We now consider what routines are needed when an event is to be processed in many worlds. In the section following this one, we will generalize these routines to tasks.

#### Process event in worlds



**Figure 133. Process event in worlds**

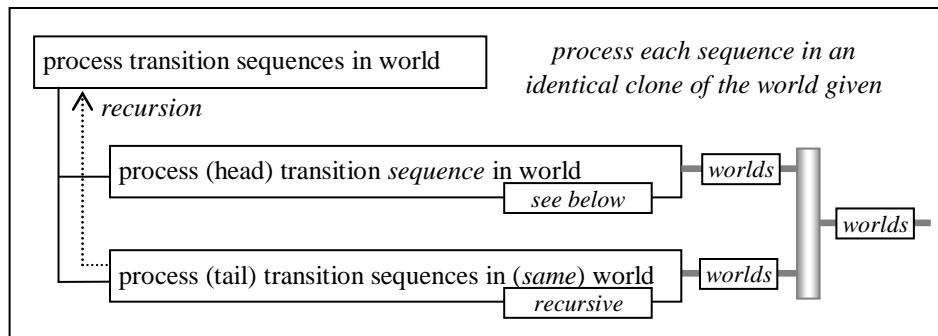
#### Process event in world



**Figure 134. Process event in world**

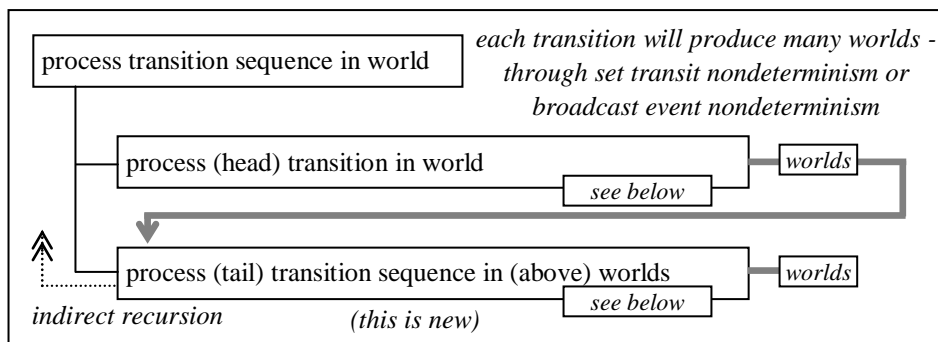
<sup>1</sup> If the sequence ordering convention is reversed, then the output of tail processing feeds into the head processing.

**Process transition sequences in world**



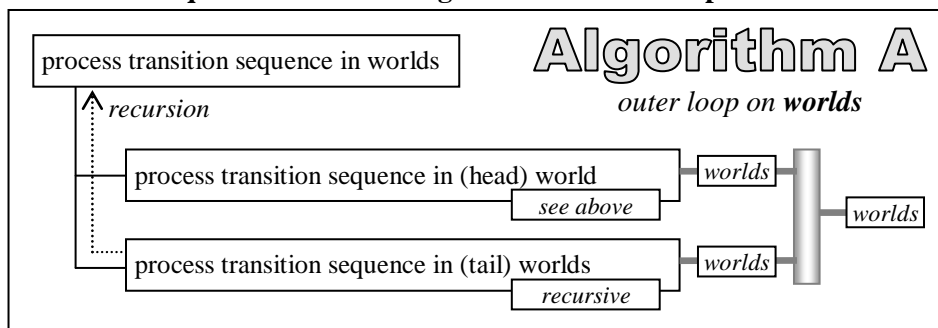
**Figure 135. Process transition sequences in world**

**Process transition sequence in world**



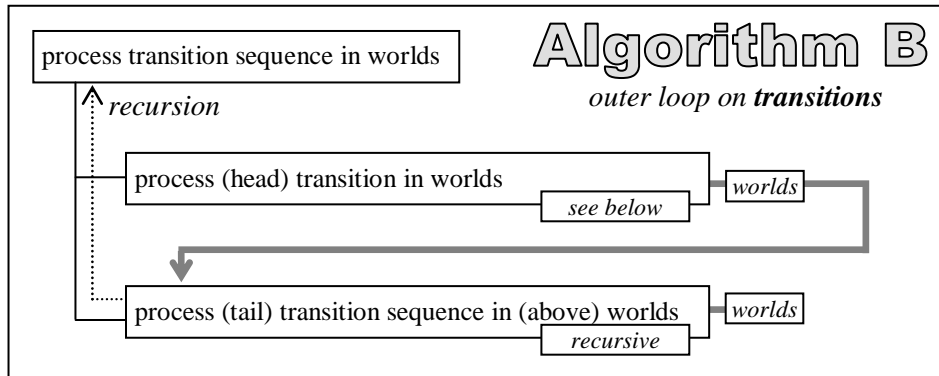
**Figure 136. Process transition sequence in world**

**Process transition sequence in worlds: Algorithm A - outer loop on worlds**



**Figure 137. Process transition sequence in worlds (A)**

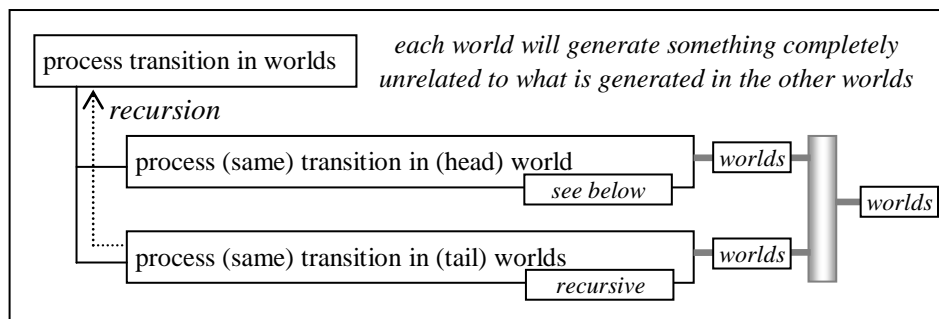
**Process transition sequence in worlds: Algorithm B - outer loop on *transitions***



**Figure 138. Process transition sequence in worlds (B)**

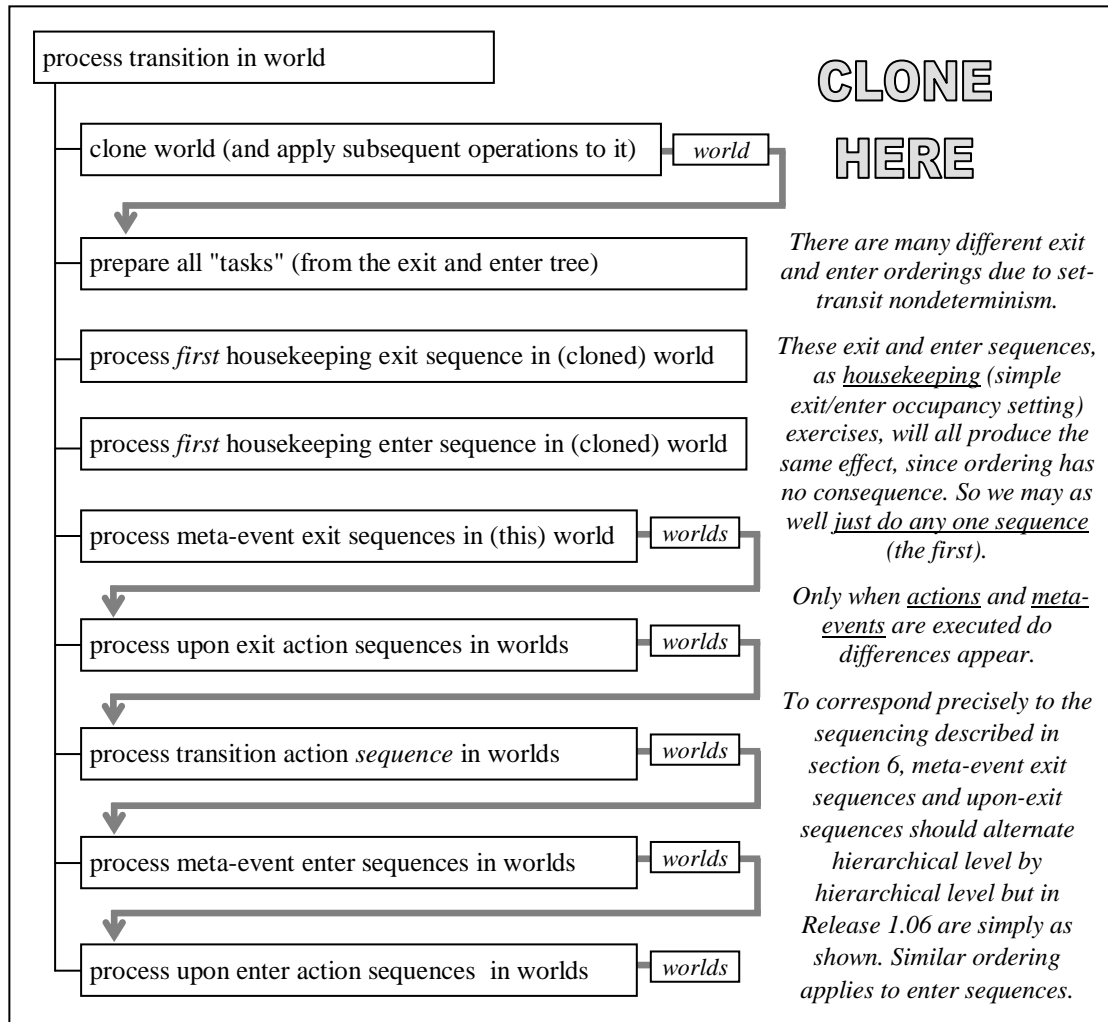
**Process transition in worlds**

This is required by the algorithm B approach to *process transition sequence in worlds*



**Figure 139. Process transition in worlds**

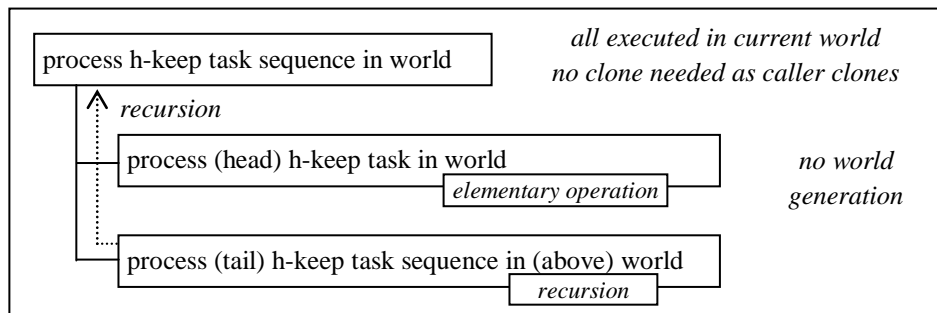
### Process transition in world



**Figure 140. Process transition in world**

### Housekeeping exit and enter tasks

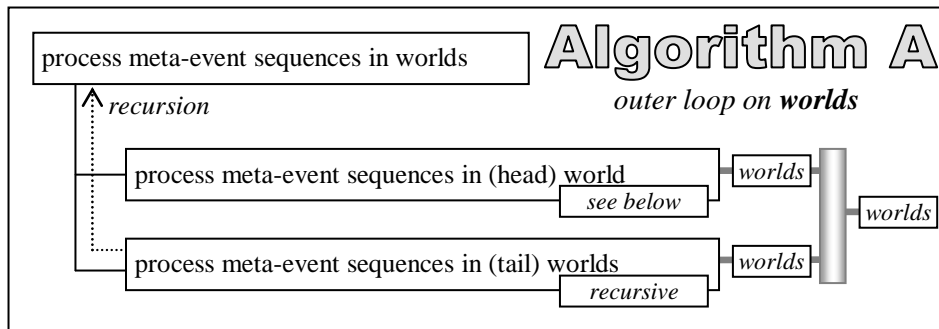
These are the simple state occupancy changes without execution of any actions.



**Figure 141. Process housekeeping task sequence in world**

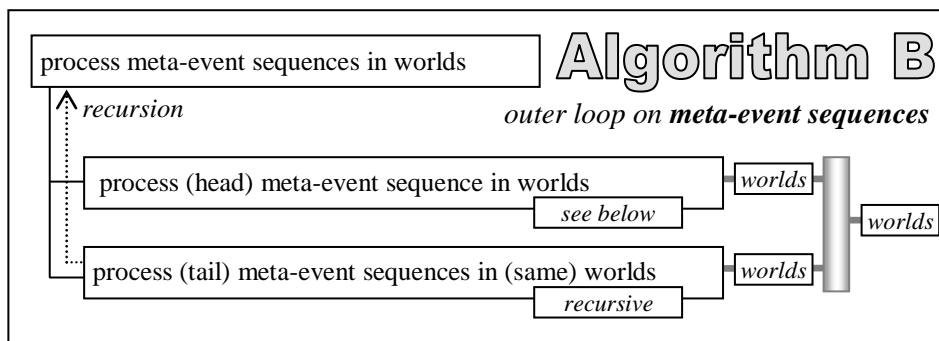


**Process meta-event sequences in worlds: Algorithm A - outer loop on *worlds***



**Figure 142. Process meta-event sequences in worlds (A)**

**Process meta-event sequences in worlds: Algorithm B - outer loop on *meta-event sequences***

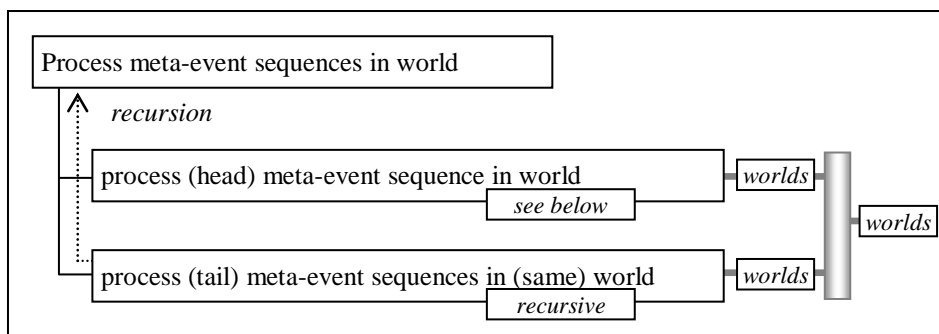


**Figure 143. Process meta-event sequences in worlds (B)**

We opt for algorithm B (see the dependency analysis below in this section).

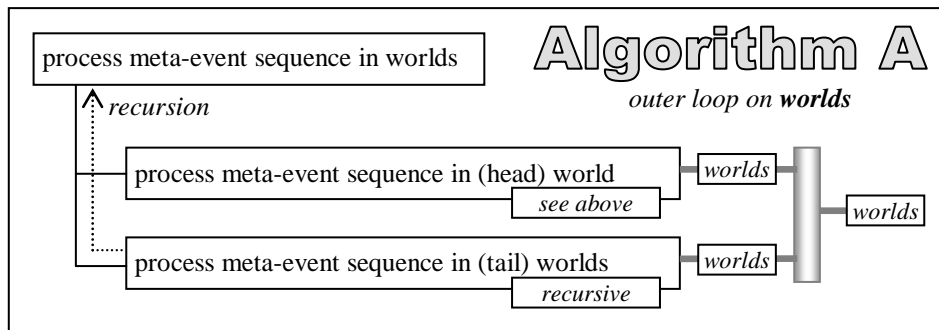
**Process meta-event sequences in world**

[Required by the algorithm A approach to *process meta-event sequences in worlds*]



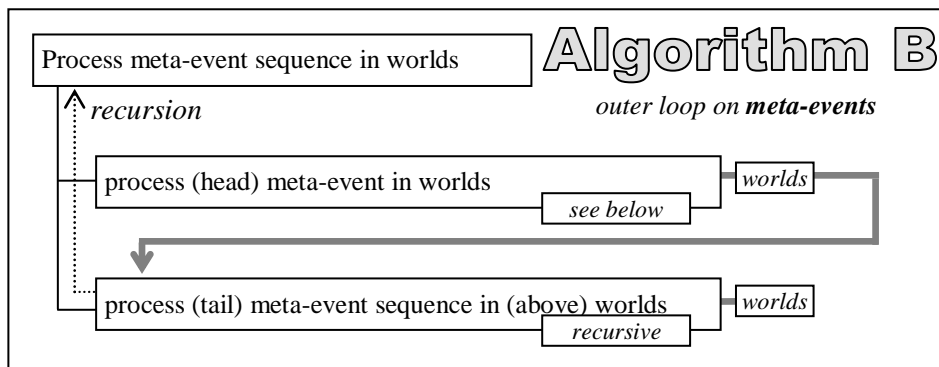
**Figure 144. Process meta-event sequences in world**

**Process meta-event sequence in worlds - Algorithm A - outer loop on worlds**



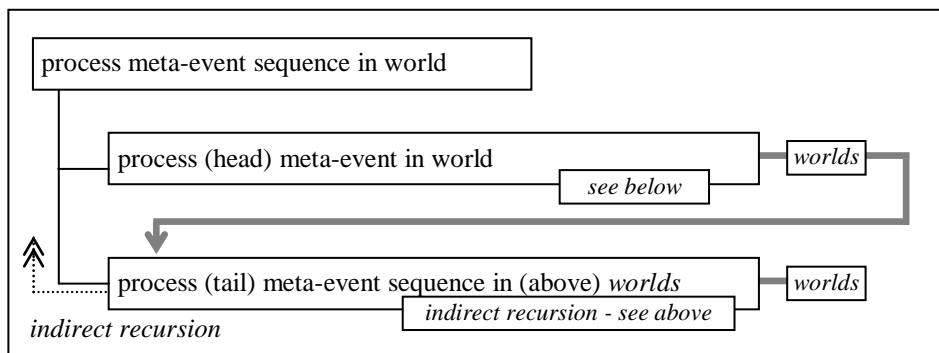
**Figure 145. Process meta-event sequence in worlds (A)**

**Process meta-event sequence in worlds - Algorithm B outer loop on meta-events**



**Figure 146. Process meta-event sequence in worlds (B)**

**Process meta-event sequence in world**



**Figure 147. Process meta-event sequence in world**

### Process action in world (1): Clone-world action type

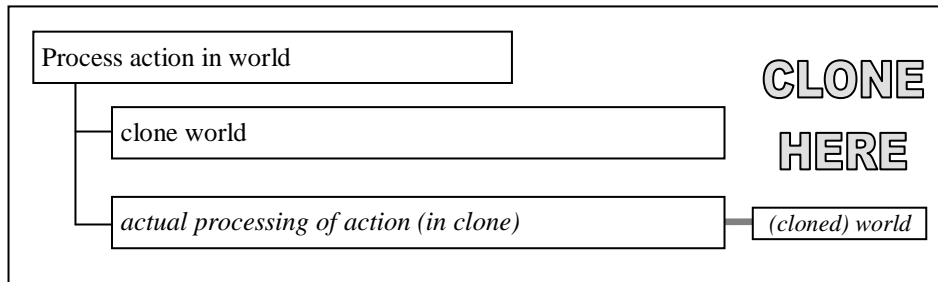


Figure 148. Clone-world action type

The clone world action type applies to assignments (including function calls).

### Process action in world (2): Delegated action types

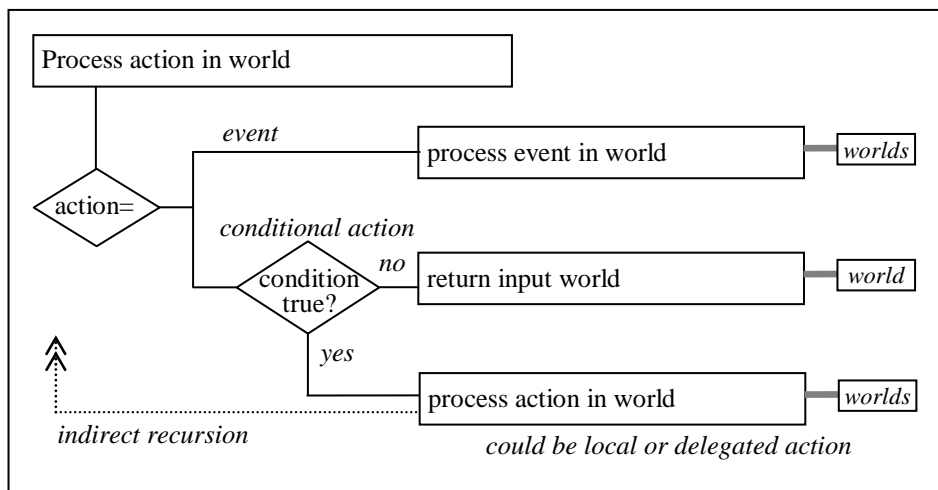


Figure 149. Delegated action types

Delegated action types are:

- fire event
- conditional action

These action types do not clone directly.

### 7.6.3 Task generalisation

Generalisation is possible across different kinds of task as long as such tasks are wrapped up in a similar way, with a tag to identify the actual task when it comes to be processed at a low level.

We have the following routines and their generalisation, with the following classification of *world mode*:

- *serial mode*, as previously explained

- **parallel mode**, as previously explained
- **specific mode**, where one task is to be processed in one given world, i.e. we are at a *client handler* level for processing the task. The task will be identified (as an *event*, *transition*, *action* etc.), and handled accordingly. Responsibility is taken for cloning if any changes are to be made to the world, and the changed world (or indirectly generated worlds) are the output. If cloning responsibility has been taken care of by the caller, the routine is free to make alterations in the world given. This mode is used for making direct state occupancy changes.

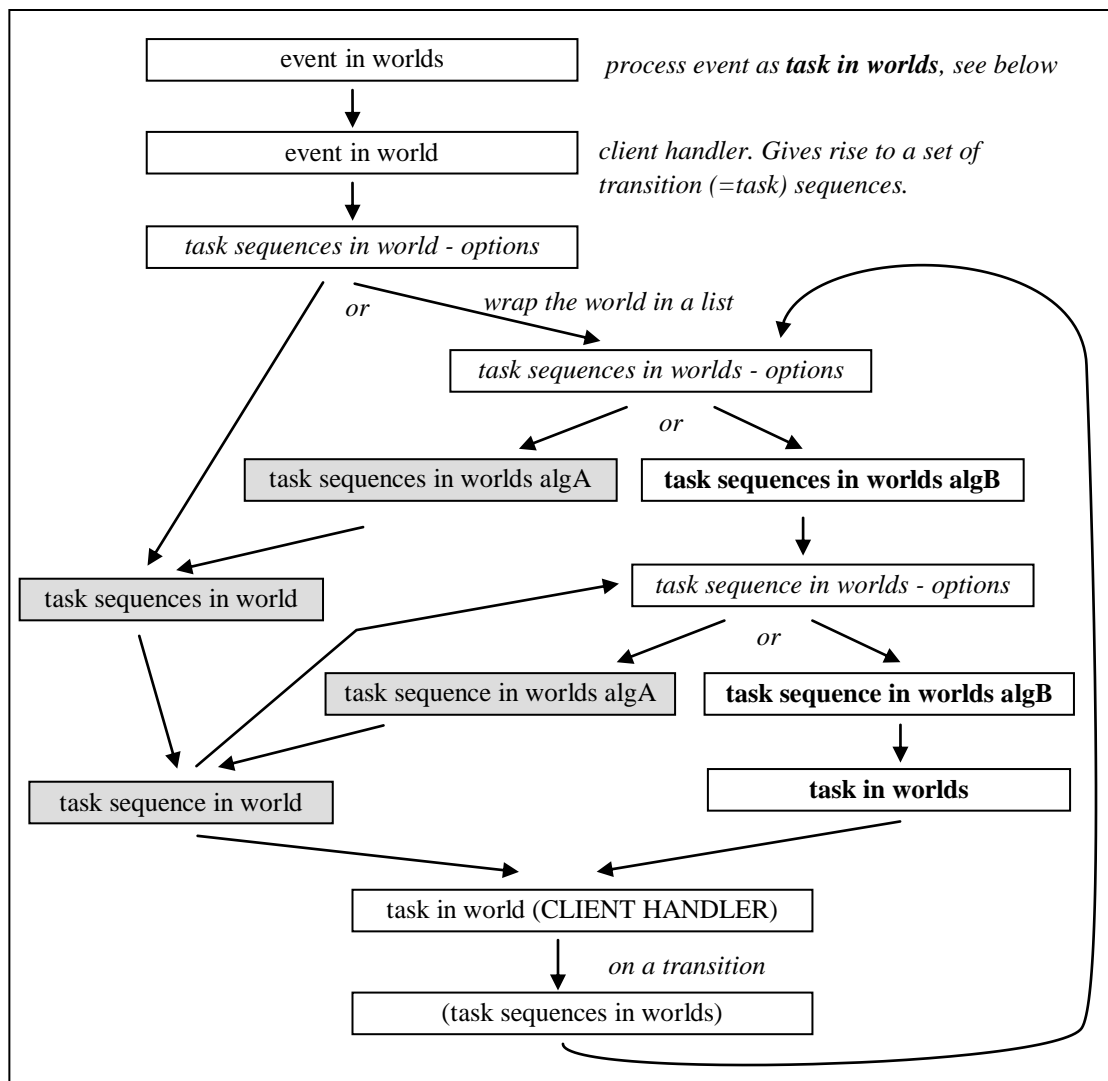
Specific	Generalized	Direct Cloning	World mode
process event in world	specific client handler	no	specific
process event in worlds	process task in <i>worlds</i>	no	parallel
process transition <i>seq</i> in <i>world</i>	process task <i>seq</i> in <i>world</i>	no	serial
process transition <i>seqs</i> in <i>world</i>	process task <i>seqs</i> in <i>world</i>	no	parallel
process transition <i>seq</i> in <i>worlds</i>	process task <i>seq</i> in <i>worlds</i> alg A/B <ul style="list-style-type: none"> <li>• Alg. A - outer loop over <b><i>worlds</i></b></li> <li>• Alg. B - outer loop over <b><i>tasks</i></b></li> </ul>	no no	parallel serial
process transition in world	specific client handler	<b>yes</b>	specific
process hkeep exit <i>seqs</i> in world process hkeep enter <i>seqs</i> in world	process task <i>seqs</i> in <i>world</i>	no	parallel
process hkeep exit task in world process hkeep enter task in world	specific client handler	no	specific
process meta-event <i>seqs</i> in <i>worlds</i>	process task <i>seqs</i> in <i>worlds</i> alg A/B <ul style="list-style-type: none"> <li>• Alg. A - outer loop over <b><i>worlds</i></b></li> <li>• Alg. B - outer loop over <b><i>seqs</i></b></li> </ul>	no no	parallel parallel
process meta-event <i>seqs</i> in <i>world</i>	process task <i>seqs</i> in <i>world</i>	no	parallel
process meta-event <i>seq</i> in <i>worlds</i>	process task <i>seq</i> in <i>worlds</i> alg A/B <ul style="list-style-type: none"> <li>• Alg. A - outer loop over <b><i>worlds</i></b></li> <li>• Alg. B - outer loop over <b><i>tasks</i></b></li> </ul>	no no	parallel serial
process meta-event <i>seq</i> in <i>world</i>	process task <i>seq</i> in <i>world</i>	no	serial
process meta-event in world	specific client handler	no	specific
process action <i>seqs</i> in <i>world</i>	process task <i>seqs</i> in <i>world</i>	no	parallel
process action <i>seq</i> in <i>worlds</i>	process task <i>seqs</i> in <i>worlds</i>	no	parallel
process action <i>seq</i> in <i>world</i>	process task <i>seq</i> in <i>world</i>	no	serial
process action in world	specific client handler	<b>some</b>	specific

**Table 12. Task generalisation**

The *actions* mentioned in the above table could be upon-exit actions, on-transition actions, or upon-enter actions. Some actions clone a world directly (e.g. an assignment); others may cause world generation indirectly (e.g. firing an event).

Having generalized, we regard the general routines as a *task-processing server*, serving *client handlers* that handle single tasks in a single world.

A *dependency analysis* shows that if we select the algorithm-B options, a minimal set of processing routines will suffice. Shaded routines are not required.



**Figure 150. Task processing dependency diagram**

#### 7.6.4 Further descriptions of task processing routines

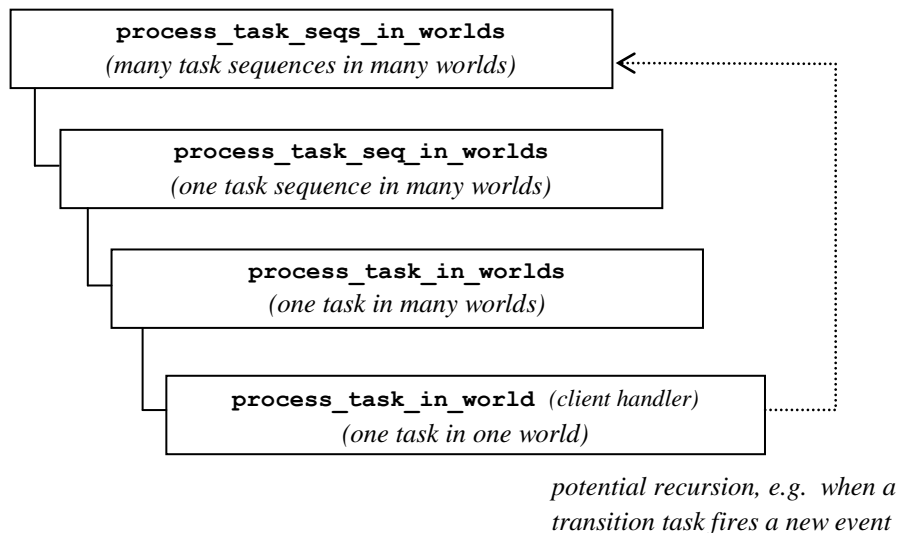
These descriptions complement those of the previous section, including some additional world generation diagrams and actual STATECRUNCHER PROLOG-code (which is remarkably compact for the functionality it gives).

Various equivalent names are used in the descriptions that precede and follow, e.g.

- Process task sequences in worlds                    *a descriptive name*
- process\_task\_seqs\_in\_worlds                    *a pseudo-code name*
- me\_process\_task\_seqs\_in\_worlds\_algB        *a specific actual code example*

The prefix “me\_” is the machine engine module naming prefix.

The hierarchy of routines derived from the dependency diagram (see previous section) can be represented as follows.



**Figure 151. Hierarchy of transition-processing routines**

##### *Process task sequences in worlds*

In Algorithm A we turn the **Process task sequences in worlds** call into **Process task sequences in world** calls. This algorithm was not chosen.

In Algorithm B we turn the **Process task sequences in worlds** call into **Process task sequence in worlds** calls. This algorithm was chosen.

## PROLOG code for process task sequences in worlds (Algorithm B)

### Process task sequences in worlds

```
/*-----*/
/* no sequences, OUTWORLDS:=INWORLDS */
/*-----*/
me_process_task_seqs_in_worlds_algB([],INWORLDS,INWORLDS):-
    me_set_world_and_bag(INWORLDS),
    !.

/*-----*/
/* one sequence, many worlds */
/*-----*/
me_process_task_seqs_in_worlds_algB([TSEQ],INWORLDS,OUTWORLDS):-
    !, /* this must be the ONLY way to handle one sequence, many worlds */
    me_process_task_seq_in_worlds(TSEQ, INWORLDS,OUTWORLDS),
    me_set_world_and_bag(OUTWORLDS),
    !.

/*-----*/
/* many sequences, many worlds */
/*-----*/
me_process_task_seqs_in_worlds_algB([H_TSEQ|T_TSEQS],INWORLDS,OUTWORLDS):-
    me_process_task_seqs_in_worlds_algB(T_TSEQS,INWORLDS,OUTWORLDS1),
    me_process_task_seq_in_worlds(H_TSEQ,INWORLDS,OUTWORLDS2),
    /*care,INWORLDS!*/
    me_merge_worlds(OUTWORLDS1,OUTWORLDS2,OUTWORLDS),
    me_set_world_and_bag(OUTWORLDS),
    !.
```

### Process task sequence in worlds

There are two possible approaches, which we discuss and illustrate in figures following:

1. Algorithm A: Outer loop over *worlds*, inner loop over *tasks*, requiring an intermediate routine **process\_task\_seq\_in\_world**
2. Algorithm B: Outer loop over *tasks*, inner loop over *worlds*, requiring an intermediate routine **process\_task\_in\_worlds**

The second of these options is probably better in general, as it probably involves merging of one small worldbag with one large worldbag. (We use the term *worldbag* for consistency with the STATECRUNCHER code - during processing it is often a bag, but to the user it is always a *set*, because the user is never confronted with intermediate results). The smaller worldbag is the result of processing just one task since the previous world merge.

We have the option of processing the worlds in head first or tail first order, though in Algorithm B this is determined at a lower level, in *process\_task\_in\_worlds*. The diagrams following illustrate world generation:

- according to algorithm A, with head world first
- according to algorithm B, with head world first
- according to algorithm B, with tail worlds first.

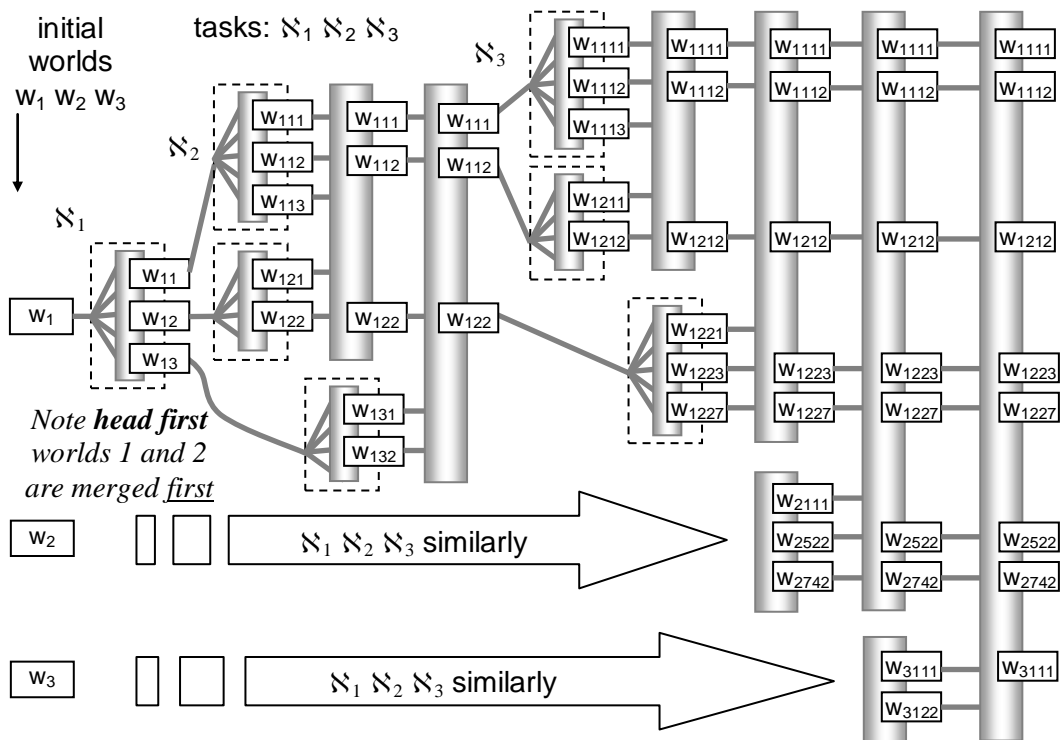


Figure 152. Process task sequence in worlds - Alg. A with head world first

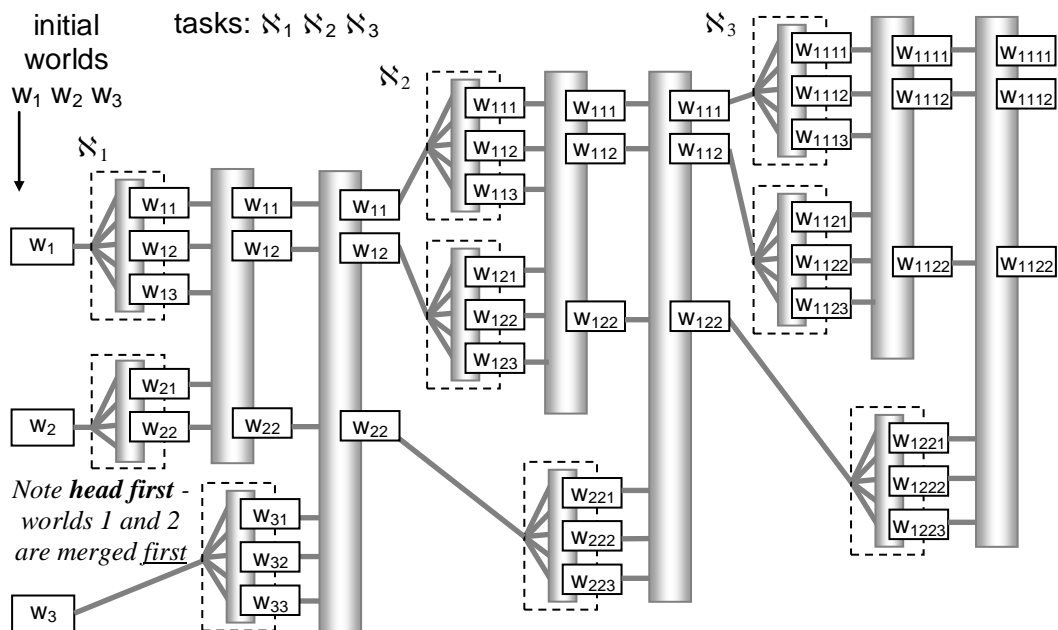
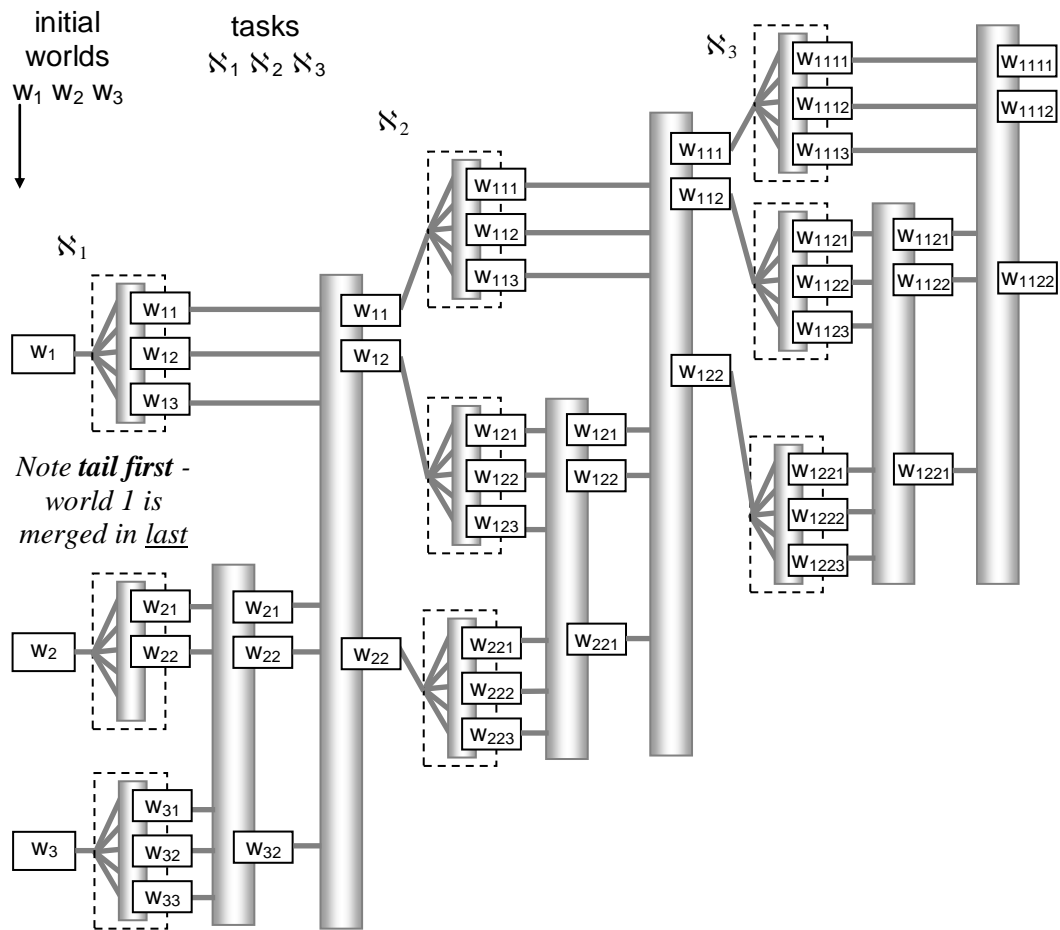


Figure 153. Process task sequence in worlds - Alg. B with head world first





**Figure 154. Process task sequence in worlds - Alg. B with tail worlds first**

N.B. The example diagrams taken do *not* correspond to the *same* state behaviour as this would not be practical in the limited diagram width.

The **Process task sequence in worlds** Algorithm B routine processes the sequence of tasks in each world in the worldbag. The first task is processed in the worldbag which obtains on calling the routine. Subsequent tasks are processed in the subsequent worldbags resulting from processing the previous task.

***PROLOG code for process task sequence in worlds<sup>1</sup>***

```

Process task sequence in worlds

/*-----*/
/* no tasks, OUTWORLDS:=INWORLDS          */
/*-----*/
me_process_task_seq_in_worlds_algB([],INWORLDS,INWORLDS):-
    me_set_world_and_bag(INWORLDS),
    !.

/*-----*/
/* one task, many worlds                  */
/*-----*/
me_process_task_seq_in_worlds_algB([TASK],INWORLDS,OUTWORLDS):-
    !, /* this must be the ONLY way to handle one task, many worlds */
    me_process_task_in_worlds(TASK, INWORLDS,OUTWORLDS),
    me_set_world_and_bag(OUTWORLDS),
    !.

/*-----*/
/* many tasks, many worlds                */
/*-----*/
me_process_task_seq_in_worlds_algB([H_TASK|T_TASKS],INWORLDS,OUTWORLDS):-
    me_process_task_seq_in_worlds_algB(T_TASKS,INWORLDS,OUTWORLDS1),
    me_process_task_in_worlds(H_TASK,OUTWORLDS1,OUTWORLDS2),
    me_merge_worlds(OUTWORLDS2,OUTWORLDS),
    me_set_world_and_bag(OUTWORLDS),
    !.

```

---

<sup>1</sup> By convention in the implementation, the tasks are in tail first order (head of list is the last task).

### ***Process task in worlds***

This routine calls **Process task in world**, which is regarded as a client routine to the task processing service. Client **Process task in world** routines will be written for event processing, transition processing, action processing etc.

It is seen that all client handlers are of signature

```
me_process_task_in_world(TASK,WORLD,OUTWORLDS)
```

and that by conforming to this, routines which really are hardly aware of nondeterminism are embeddable in a scheme for handling highly nondeterministic tasks.

### ***PROLOG code for process task in worlds***

#### **Process task in worlds**

```
/*-----*/
/* no worlds      */
/*-----*/
me_process_task_in_worlds(_, [], []) :-
    !.

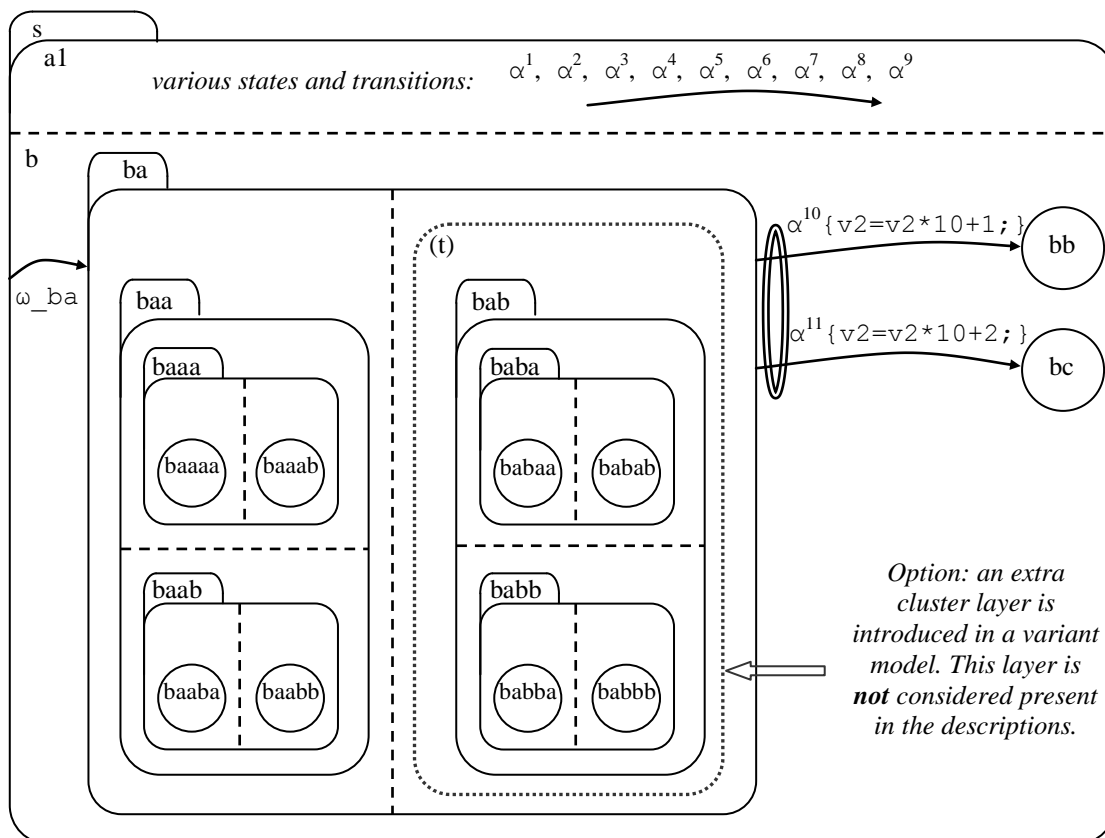
/*-----*/
/* one world      */
/*-----*/
me_process_task_in_worlds(TASK, [WORLD], OUTWORLDS) :-
    !, /* this must be the ONLY way to handle one task, one world */
    da_write_world(WORLD),
    me_process_task_in_world(TASK, WORLD, OUTWORLDS), /* calls client handler */
    me_set_world_and_bag(OUTWORLDS),
    !.

/*-----*/
/* many worlds    */
/*-----*/
me_process_task_in_worlds(TASK, [H_INWORLD|T_INWORLDS], OUTWORLDS) :-
    me_process_task_in_worlds(TASK, T_INWORLDS, OUTWORLDS1),
    da_write_world(H_INWORLD),
    me_process_task_in_world(TASK, H_INWORLD, OUTWORLDS2), /*calls client handler*/
    me_merge_worlds(OUTWORLDS1, OUTWORLDS2, OUTWORLDS),
    me_set_world_and_bag(OUTWORLDS),
    !.
```

### **7.6.5 Set-transit nondeterminism; permutable sequences and trees**

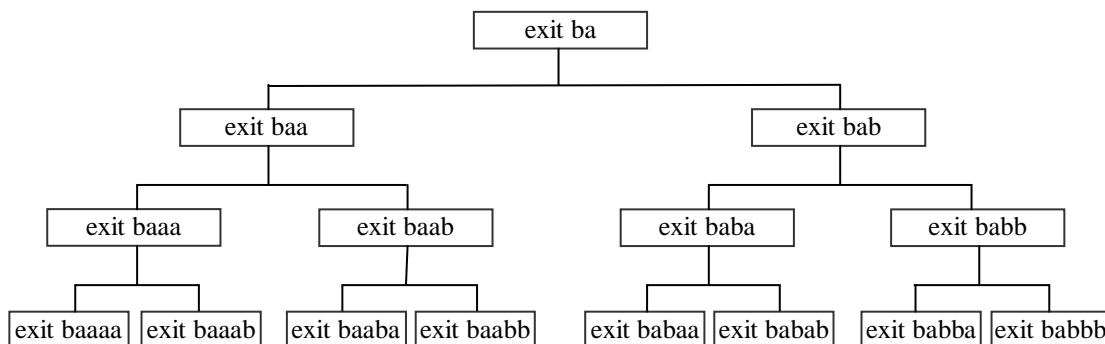
We are nearly ready to review the individual task *client handlers*. But first we illustrate exit and enter trees, and task permutations derived from them, as needed in *Process transition in world*, where we process set nondeterminism. In this section, we will denote sequences using square brackets, for compatibility with illustrative PROLOG examples.

The following model gives rise to set-transit nondeterminism:



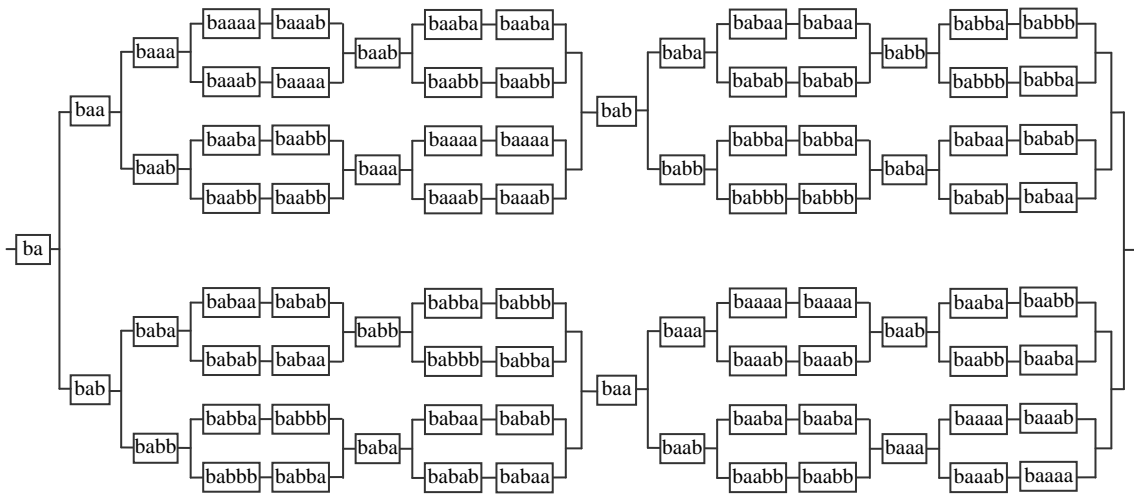
**Figure 155. Effect of set-transit nondeterminism (cf. Figure 113)**

We ignore the *fork* on transitions  $\alpha^{10}$  and  $\alpha^{11}$ , and any *races* with any others ( $\alpha^1 - \alpha^9$ ), since these will have been abstracted away by the time one transition is to be processed in one world. We take the exiting part of  $\alpha^{10}$  as our example. In general an exit tree is produced, in our example as follows:



**Figure 156. Exit tree**

Each node will give rise to a permutation of its branches. The exit sequences are equivalent to the paths (from right to left) through the diagram below.



**Figure 157. Exit sequences**

Notice that we do not permute *all* lowest-level exit tasks in one big permutation. We permute on a level by level basis, retaining orderings imposed by a previous level. So we do not permute the lower level exit tasks from one set member with any exit tasks of a different member. The total number of paths through the above figure is  $2^7 = 128$  (being less than the number of permutations of all 8 leafstates to be exited, which is  $8! = 40320$ ).

The above tree happens to be a binary one, because our sets have just two members, but that is of course not the case in general. If there had been an intervening *cluster* in the exit tree, it would not give rise to any extra permutations, as it would be at a node with one branch, and would not be marked for permutation. Such a cluster is shown in dotted outline in Figure 155.

### Permutation handling

Permutable sequences need to be able to represent parts of the sequence being permuted and parts not. This must apply across different nesting levels. Two elements,

[A, B]

may form a permutable subsequence, so requiring expansion into

[A, B] and [B, A].

One of these elements, say A, may itself be a subsequence, say [a1, a2], that is to be permuted. The other may be a subsequence [b1, b2] that is *not* to be permuted. The required generated subsequences from

[A, B] = [[a1, a2], [b1, b2]]

if flattened are then

[a1, a2, b1, b2], [a2, a1, b1, b2], [b1, b2, a1, a2], [b1, b2, a2, a1].

It may be that [b1, b2] should be treated as a single element, so that we must generate

[a1, a2, [b1, b2]], [a2, a1, [b1, b2]], [[b1, b2], a1, a2], [[b1, b2], a2, a1].

In general, we will need control over what is to be permuted, and what is not, and what is to be flattened and what is not. Certain elements of a sequence are likely to be nested lists in themselves, and as such they must neither be permuted nor flattened.

We can represent all our requirements in a PROLOG-compatible way using the following indicators in a list

- leading element **\$pm\_y** ('permute-yes') means generate all permutations of this list. The **\$** is to avoid clashes with user symbols, and in PROLOG code this needs quoting, '**\$pm\_y**'. Each permutation generated will substitute **\$pm\_d** ('permute-done') for **\$pm\_y**. Also, all sublists will be walked for further permutation indications. The **\$pm\_d**'s can be removed later. A *nonleading* **\$pm\_y** element is not recognized as an indicator.
- for any other leading element, the list will be not be permuted at this level, but it will be walked looking for permutations at lower levels.

When a permuted list is flattened, that all sublists starting with **\$pm\_d** are raised up a level.

If the user wishes to effect a permutation on certain chunks of a list monolithically (but with possible sublist permutations as well), then the user will need to wrap the chunks as sublists. Automatic unwrapping of such chunks can be performed if the user *supplies an extra \$pm\_d* element at the head of such chunks.

The following examples show this in action. They show sequences wrapped as permutations.

Wrapped sequence	Equivalent straight sequences after flattening
[a, [b, c]]	[a, [b, c]] <i>no permutation because no indicator</i>
['\$pm_y', a, b, c]	[a, b, c] [a, c, b] [b, a, c] [b, c, a] [c, a, b] [c, b, a]
[a, [b, c], ['\$pm_y', d, ['\$pm_y', e1, e2]], f] <i>two independent permutations</i>	[a, [b, c], d, e1, e2, f] [a, [b, c], d, e2, e1, f] [a, [b, c], e1, e2, d, f] [a, [b, c], e2, e1, d, f]

```
[ba, ['$pm_y', BAA, BAB]],
where
BAA=['$pm_d', baa, ['$pm_y', BAAA, BAAB]],
BAB='BAB',
BAAA=['$pm_d', baaa, ['$pm_y', baaaa, baaab]],
BAAB=['$pm_d', baab, ['$pm_y', baaba, baabb]].
Note the user of $pm_d in the input.
This is the set-transit example, but simplified by condensing all bab... items into one symbol, BAB.
```

```
[ba,baa,baaa,baaaa,baaab,baab,baaba,baabb,BAB]
[ba,baa,baaa,baaaa,baaab,baab,baabb,baaba,BAB]
[ba,baa,baaa,baaab,baaaa,baab,baaba,baabb,BAB]
[ba,baa,baaa,baaab,baaaa,baab,baabb,baaba,BAB]
[ba,baa,baab,baaba,baabb,baaa,baaaa,baaab,BAB]
[ba,baa,baab,baaba,baabb,baaa,baaab,baaaa,BAB]
[ba,baa,baab,baabb,baaba,baaa,baaaa,baaab,BAB]
[ba,baa,baab,baabb,baaba,baaa,baaab,baaaa,BAB]
[ba,BAB,baa,baaa,baaaa,baaab,baab,baaba,baabb]
[ba,BAB,baa,baaa,baaaa,baaab,baab,baabb,baaba]
[ba,BAB,baa,baaa,baaab,baaaa,baab,baaba,baabb]
[ba,BAB,baa,baaa,baaab,baaaa,baab,baabb,baaba]
[ba,BAB,baa,baab,baaba,baabb,baaa,baaaa,baaab]
[ba,BAB,baa,baab,baaba,baabb,baaa,baaab,baaaa]
[ba,BAB,baa,baab,baabb,baaba,baaa,baaaa,baaab]
[ba,BAB,baa,baab,baabb,baaba,baaa,baaab,baaaa]
```

```
X=[ba, ['$pm_y', BAA, BAB]],
BAA=['$pm_d',baa, ['$pm_y', BAAA, BAAB]],
BAB=['$pm_d',bab, ['$pm_y', BABA, BABB]],
BAAA=['$pm_d',baaa, ['$pm_y',baaaa,baaab]],
BAAB=['$pm_d',baab, ['$pm_y',baaba,baabb]],
BABA=['$pm_d',baba, ['$pm_y',babaa,babab]],
BABB=['$pm_d',babb, ['$pm_y',babba,babbb]].
```

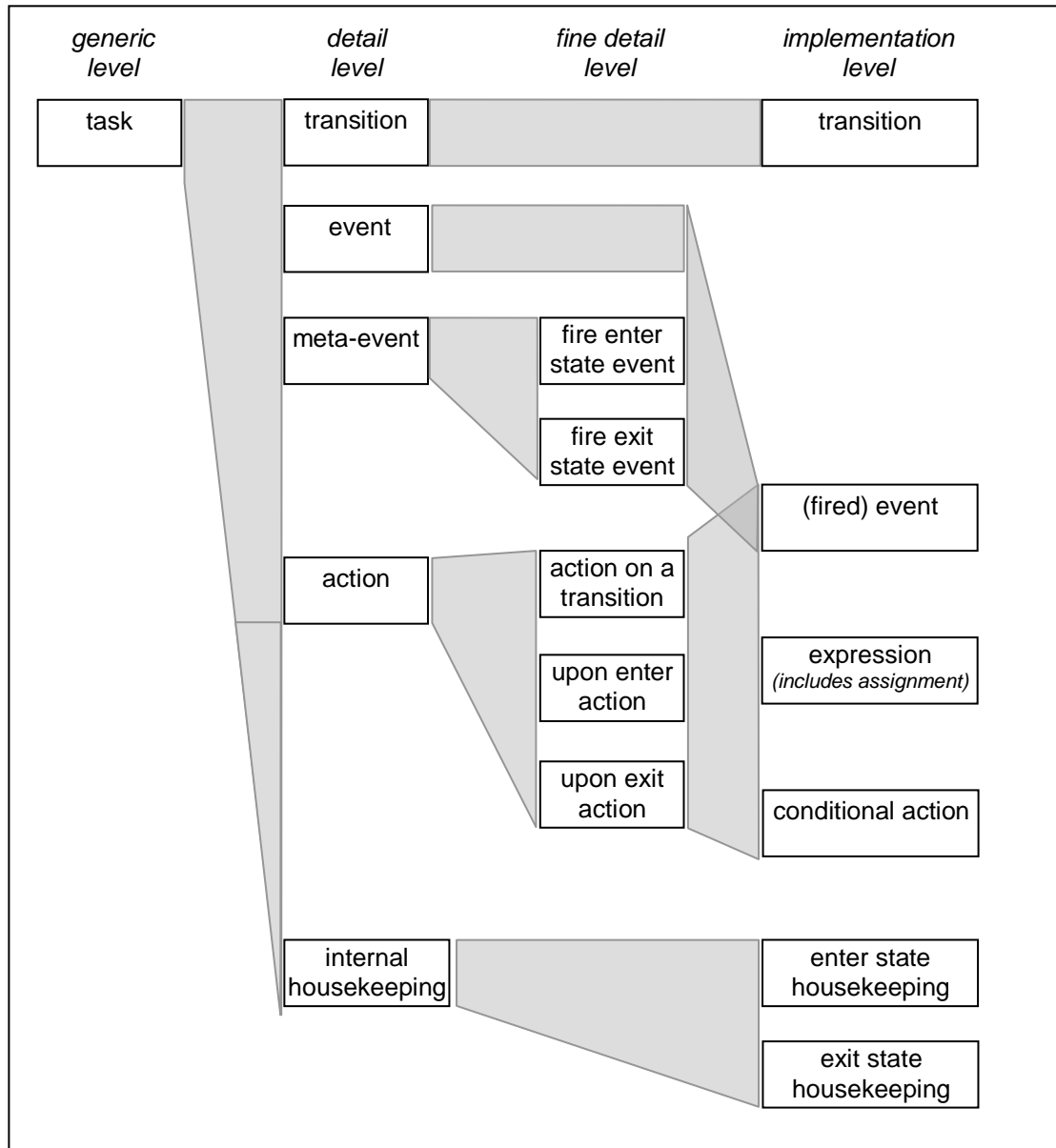
*This is the set-transit example above.*

```
[ba,baa,baaa,baaaa,baaab,baab,baaba,baabb,bab,baba,babaa,babab,babb,babba,babbb]
[ba,baa,baaa,baaaa,baaab,baab,baaba,baabb,bab,baba,babaa,babab,babb,babbb,babba]
[ba,baa,baaa,baaaa,baaab,baab,baaba,baabb,bab,baba,babab,babaa,babb,babba,babbb]
... (128 permutations)
[ba,bab,babb,babbb,babba,baba,babab,babaa,baa,baab,baaba,baabb,baaa,baaab,baaaa]
[ba,bab,babb,babbb,babba,baba,babab,babaa,baa,baab,baabb,baaba,baaa,baaaa,baaab]
[ba,bab,babb,babbb,babba,baba,babab,babaa,baa,baab,baabb,baaba,baaa,baaab,baaaa]
```

**Table 13. Permutation generation**

## 7.6.6 Review of tasks

This section is a review of the various tasks. The following figure shows what tasks exist.



**Figure 158. Breakdown of tasks**

### 7.6.6.1 Process task in world

Here we come to the innermost part of the hierarchy of sets and sequences and worlds, where we must process according to the specific kind of task involved. In C terms, this is just a switch statement to route control to the right lower-level routine. In C++ terms this might be a



question of matching a prototype function according to a parameter type. In PROLOG terms, it is a question of matching a call with a predicate using parameter unification to obtain the right predicate for the task in question.

<b>Process_task_in_world</b>	
Given an input world and kind of task, switch on kind of task	
Case <i>transition</i> :	call process_transition
Case <i>fired event</i> :	call process_event
Case <i>expression( incl. assignment)</i> :	call process_expression
Case <i>conditional</i> :	call process_conditional_action
Case <i>enter-state housekeeping</i> :	call process_enter_state_housekeeping
Case <i>exit-state housekeeping</i> :	call process_exit_state_housekeeping

**Figure 159. Process\_task\_in\_world**

We have seen the general nature of these (Figure 140, Figure 141, Figure 148 and Figure 149). In this section, we consider the tasks in more detail, especially the processing of a transition, where set-nondeterminism is handled.

### 7.6.6.2 Process event

An outline was given in Figure 134. Transitions for the supplied world only are selected according to the algorithm given in section 7.1. This gives rise to a *set of transition sequences*. Note that this is the case whether we opt for hierarchical prioritisation or hierarchical fork nondeterminism, except that the latter case generally produces larger sets and longer sequences. The resulting *set of transition sequences* can be processed by *process\_task\_seqs\_in\_worlds*.

<b>process_event</b>
In world supplied $w_i$
Generate the <i>set of transition sequences</i> on this event, $T_{exec}$
Wrap the world as a list (with this one element)
Process set of transition sequences using <i>process_task_seqs_in_worlds</i>

**Figure 160. Review of process event**

We do not pre-clone for any of these transition sequences. This routine will perform all processing needed departing from a given world (which will be left intact, eventually being cloned at a lower level). We do not need to think about intermediate processing, such as *process a transition sequence*, as all has been taken care of in our hierarchy of task processing routines.

### 7.6.6.3 Process transition

The structure of this task was shown in Figure 140. Transitions are processed according to the ‘after-landing’ principles and sequencing as already discussed.

A transition can alter state occupancies, and so clones the supplied world. Through the multiple calls to this routine as a result of *event* processing, many new worlds will be created (and merged). Even if there is no nondeterminism, one new world will be generated, because this routine does not know how much nondeterminism is involved, if any.

We considered transition processing in more detail, covering especially set nondeterminism. Set nondeterminism involves entering (or exiting) the members of a set in different orderings. Unlike fork nondeterminism and race nondeterminism, set-nondeterminism permutations are generated during transition *processing*, not transition *selection*. From the transition, an enter tree and an exit tree are derived. From these trees, all forms of set nondeterminism are derived (set transit/action and set meta-event nondeterminism).

### 7.6.6.4 Process expression

A clone takes place (Figure 148). Expressions are evaluated in a specified scope by a standard call to the evaluator. An assignment is regarded as an expression including the assignment operator ‘=’.

### 7.6.6.5 Process conditional action

This simply consists of evaluating the condition and recursively calling the relevant nested action (Figure 149).

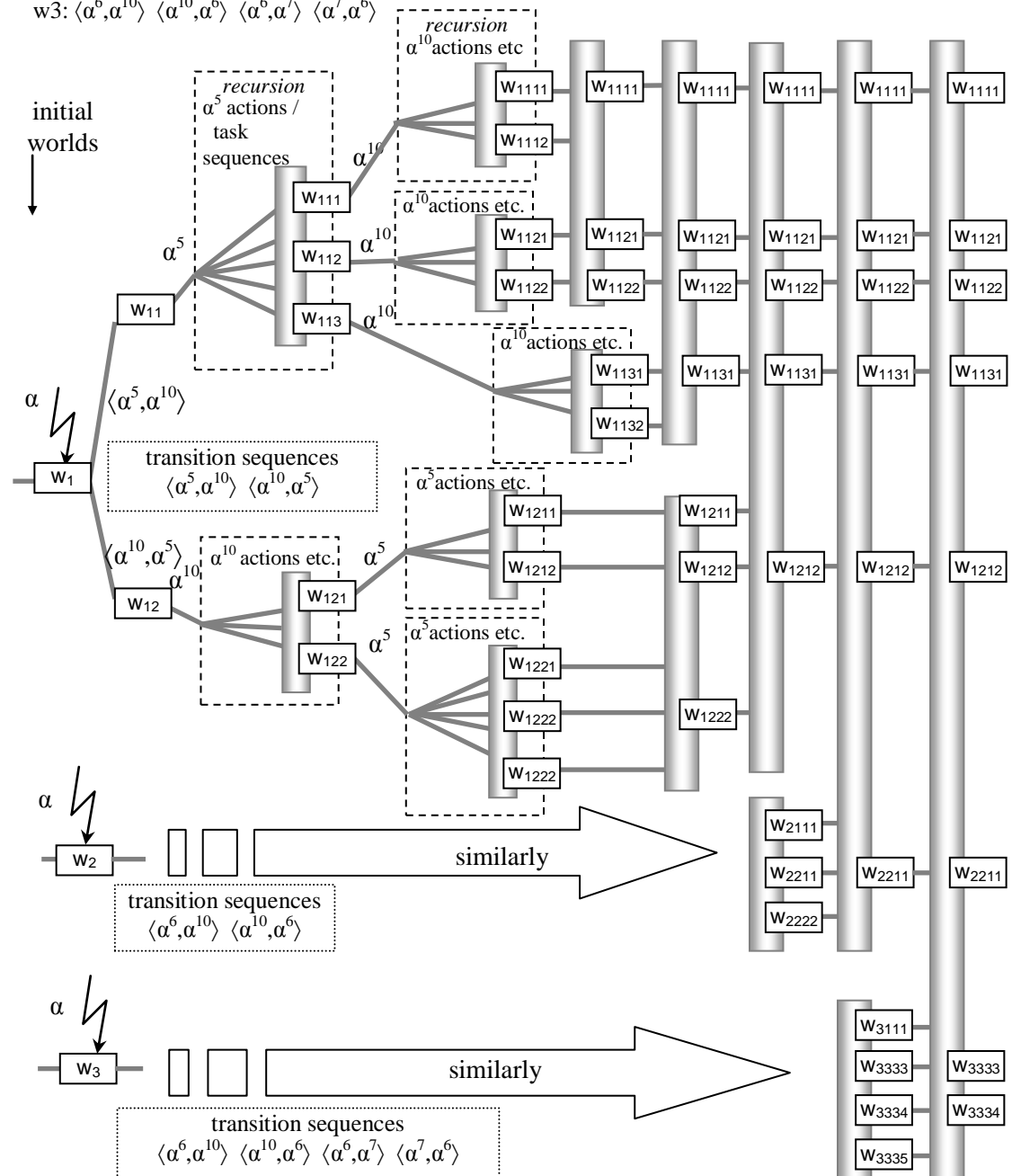
### 7.6.6.6 Process enter state housekeeping and Process exit state housekeeping

An outline was given in Figure 141. The routines changes state occupancies and cluster history settings in the current world. No clone of the world is needed as the transition processing routine takes responsibility for it. Since all *consequences* of state changes have been separated out (on-exit actions etc.), the order in which housekeeping changes are made is irrelevant. Only one of the many orderings generated by permutation of the enter/exit trees is used by the transition processing routine when it calls this routine.

### 7.6.7 Summary by example of event processing

We process event  $\alpha$  in 3 worlds. The transition sequences per world to process are:

- w1:  $\langle \alpha^5, \alpha^{10} \rangle \langle \alpha^{10}, \alpha^5 \rangle$
- w2:  $\langle \alpha^6, \alpha^{10} \rangle \langle \alpha^{10}, \alpha^6 \rangle$
- w3:  $\langle \alpha^6, \alpha^{10} \rangle \langle \alpha^{10}, \alpha^6 \rangle \langle \alpha^6, \alpha^7 \rangle \langle \alpha^7, \alpha^6 \rangle$



**Figure 161. Example of event processing**

## 8. The STATECRUNCHER command language

This topic is discussed in detail in [StCrPrimer]. Here, we give the inventory of all STATECRUNCHER commands. The most important point about the commands is that they enable a *primer* (a neighbouring program) to communicate with STATECRUNCHER in various ways, the combination potentially providing very sophisticated test generation algorithms.

The table below shows abbreviated commands as well as unabbreviated ones. Where abbreviated ones are not available, the arrow (→) refers the reader to the unabbreviated one.

Syntax of the descriptions: An optional argument to a command is preceded by a question mark, (?). Normal *courier* indicates a literal item; *italics* indicate a non-literal or explanation. A choice is indicated by a vertical bar (/).

The important commands are those that allow setting of state occupancies and variables and traces. These make a state-space exploration algorithm possible. These are

- *WORLD STATEKIND STATENAME MPATH = OCCUPANCY HISTORY*
- *WORLD VAR VARKIND VARIABLENAME MPATH = VALUE*
- *WORLD TRACE = TRACE*

These commands are in STATECRUNCHER's own output format.

Abbrev.	Command
Command	<i>showing typical example and/or typical output</i>

### *Main processing: high priority black box testing commands*

<b>pe ...</b>	<b>process event EVENT ?p=PARAMETERS ?t=EXPECTEDTRACE</b> pe gamma p=[4,xy] ( <i>statechart scope assumed</i> ) pe [alpha,[sc]] p=1 t=[2,4] pe [alpha,[sc]] <i>Parameters can also be supplied in STATECRUNCHER internal form, e.g.</i> p=[[ex_co,int,4],[ex_str,[120,121]]] <i>Worlds in direct violation of EXPECTEDTRACE will be killed, but overtrace and undertrace are tolerated.</i>
<b>gt</b>	<b>get trace</b> 7 TRACE =[1,2]

<b>ct</b>	<b>clear trace</b> <i>(this also causes a world merge)</i>
-----------	---

**Main processing: medium priority commands**

<b>gae</b>	<b>get all events</b> <i>(whether transitionable or not; not world-related)</i> EVENT [theta2, [z3,z,s,sc]] [pcol,[z,s,sc]]
<b>gate</b>	<b>get all transitionable events</b> <i>(union from all worlds; no worlds shown)</i> TREV [[delta,[sc]],0,[],[]] TREV [[gamma,[sc]],3, [[r,0,100000],[r,0,100000],[r,0,100000]],[]] TREV [[gamma,[sc]],1,[[r,0,100000]],[]] TREV [[gamma,[sc]],2, [[r,0,100000],[r,0,100000]],[]] TREV [[alpha,[sc]],0,[],[]]
<b>gav</b>	<b>get all variables</b> <i>Gets the value-ranges, not the current value per world</i> VAR INTEGER bool1 [sc] RANGE=[0, 1] VAR INTEGER col1 [sc] ENUM=[0, 7, 8, 4, 8] VAR INTEGER p1 [b2, b, s, sc] RANGE=[0, 9] VAR STRING str [sc]
<b>gaw</b>	<b>get all worlds</b> <i>Gets the current worlds</i> [2,7,8]
<b>gc</b>	<b>get config</b> 2 statechart sc 2 cluster a [s, sc] =OCC [] ** 2 leafstate a1 [a, s, sc] =OCC [] ** 2 cluster a2 [a, s, sc] =VAC [] 2 VAR INTEGER bool1 [sc] =1 2 VAR INTEGER col1 [sc] =8 2 VAR INTEGER p1 [b2, b, s, sc] =unknown 2 VAR STRING p5 [b2, b, s, sc] =unknown 2 VAR STRING str [sc] =[98] =b 2 TRACE =[] 2 TREV [[zeta,[s,sc]], 4,[[r,0,9],[e,0,7,8,4,8],[r,0,1],[<string>]], [pcol,[z3,z,s,sc]]] outworlds=[2,4] number of outworlds=2

<b>gst</b>	<b>get symbol table</b> SYMB delta [sc] eventdecl [] XREF leafstate b1:[b, s, sc] XREF leafstate z1:[z, s, sc]
<b>kill ...</b>	<b>kill WORLD / WORLDS</b> kill 2 kill [2,7,10]
→	<b>WORLD TRACE = TRACE</b> <i>input is as the output of get config this does <b>not</b> cause a world merge (we will probably issue this kind of command several times before requiring a world merge)</i>
→	<b>WORLD STATEKIND STATENAME MPATH = OCCUPANCY HISTORY</b> <i>input is as the output of get config this does <b>not</b> cause a world merge (we will probably change more)</i>
→	<b>WORLD VAR VARKIND VARIABLENAME MPATH = VALUE</b> <i>input is as the output of get config this does <b>not</b> cause a world merge (c.f. WORLD TRACE = TRACE)</i>
<b>cnw</b>	<b>create new world</b> <i>Creates a new world in its default state - needed before writing variable/state/trace values to a new world 34 (the new world number is returned)</i>
<b>mw</b>	<b>merge worlds</b> <i>(useful when all trace/state/variable changes have been made)</i>
<b>gpt</b>	<b>get processing time</b> <i>(timing data is set on processing an event) exec time=00h 00m 00s 210ms</i>
<b>gd</b>	<b>get date</b> <i>(get date and time) DATE: 24 Apr 2003 16:01:40/649</i>

***Containment of combinatorial explosion: low priority commands***

*These commands limit the number of permutations used in set transit  
nondeterminism and race nondeterminism.*

<b>nst</b>	<b>no set tran</b>
<b>lst</b>	<b>low set tran</b>
<b>mst</b>	<b>medium set tran</b>
<b>hst</b>	<b>high set tran</b>
<b>nr</b>	<b>no race</b>
<b>lr</b>	<b>low race</b>
<b>mr</b>	<b>medium race</b>
<b>hr</b>	<b>high race</b>

*Compilation, loading, start-up, and finish: very low priority*

<b>root ...</b>	<b>root</b> <i>ROOTDIRECTORY</i> <i>Sets the root directory to be used with FILENAMEs</i>
<b>mm</b>	<b>mode</b> <i>modelnames</i> <i>Sets compilation etc. to work with model names. The directory structure must be set up correctly.</i>
<b>mf</b>	<b>mode</b> <i>filenames</i> <i>(Default). Sets compilation etc. to work with file names. Use the root command to set the directory (can be null, then give a full path here).</i>
<b>cp ...</b>	<b>compile</b> <i>FILENAME / MODELNAME</i> <i>(also loads machine, and enters it (as of Rel 1.05))</i>
<b>ld ...</b>	<b>load</b> <i>FILENAME / MODELNAME</i> <i>(does not enter machine)</i>
<b>run ...</b>	<b>run</b> <i>FILENAME / MODELNAME</i> <i>=Load and enter machine</i>
<b>nm</b>	<b>enter</b> <i>machine</i> <i>Machine enters default state</i>
<b>xm</b>	<b>exit</b> <i>machine</i> <i>Leaves a pristine machine ready to be entered</i>
<b>um</b>	<b>unload</b> <i>machine</i> <i>Removes data and object code</i>
<b>rm</b>	<b>reset</b> <i>machine</i> <i>=exit and enter</i>
<b>quit</b>	<b>quit</b>

*System/diagnostic: very low priority*

<b>help</b>	<b>help</b>
<b>prolog</b>	<b>prolog</b> <i>Gives a PROLOG prompt; enter a PROLOG goal</i>

**Table 14. STATECRUNCHER commands**

Notes:

- By priority, we mean the priority given through the parse-attempt order, which will affect the response time.
- If anything is to be set in non-existent world, it is created (but a model must have been loaded)

***A typical sequence of commands***

1. mm                *set model mode*
2. run t5110       *load model and enter machine*
3. pe alpha        *process event alpha (in statechart scope)*
4. gc                *get configuration*
5. pe gamma       *process event gamma (in statechart scope)*
6. gc                *get configuration*
7. rm                *reset machine*
8. pe gamma       *process event gamma (in statechart scope)*
9. quit              *quit STATECRUNCHER*

Error and warning messages are shown in the following table.

***Command parsing***

PR-E-020	COMMAND SYNTAX ERROR
----------	----------------------

***Preliminary checks***

PR-E-040	NO MODEL LOADED (compiler-produced part)
PR-E-041	NO MODEL LOADED (validator-produced part)
PR-E-042	MULTIPLE COMPILED FILES LOADED
PR-E-043	MULTIPLE VALIDATED FILES LOADED
PR-E-044	THERE WAS A COMPILATION ERROR
PR-E-045	THERE WAS A VALIDATION ERROR
PR-E-046	VERSION INCOMPATIBILITY

***Command execution***

PR-E-060	COMMAND EXECUTION ERROR
PR-E-061	WORLD IS NEITHER EXTANT NOR EXTINCT

***Internal errors***

PR-E-900	INTERNAL ERROR - NO COMMAND HANDLER
----------	-------------------------------------

**Table 15. Error and warning messages**



## 9. Using STATECRUNCHER

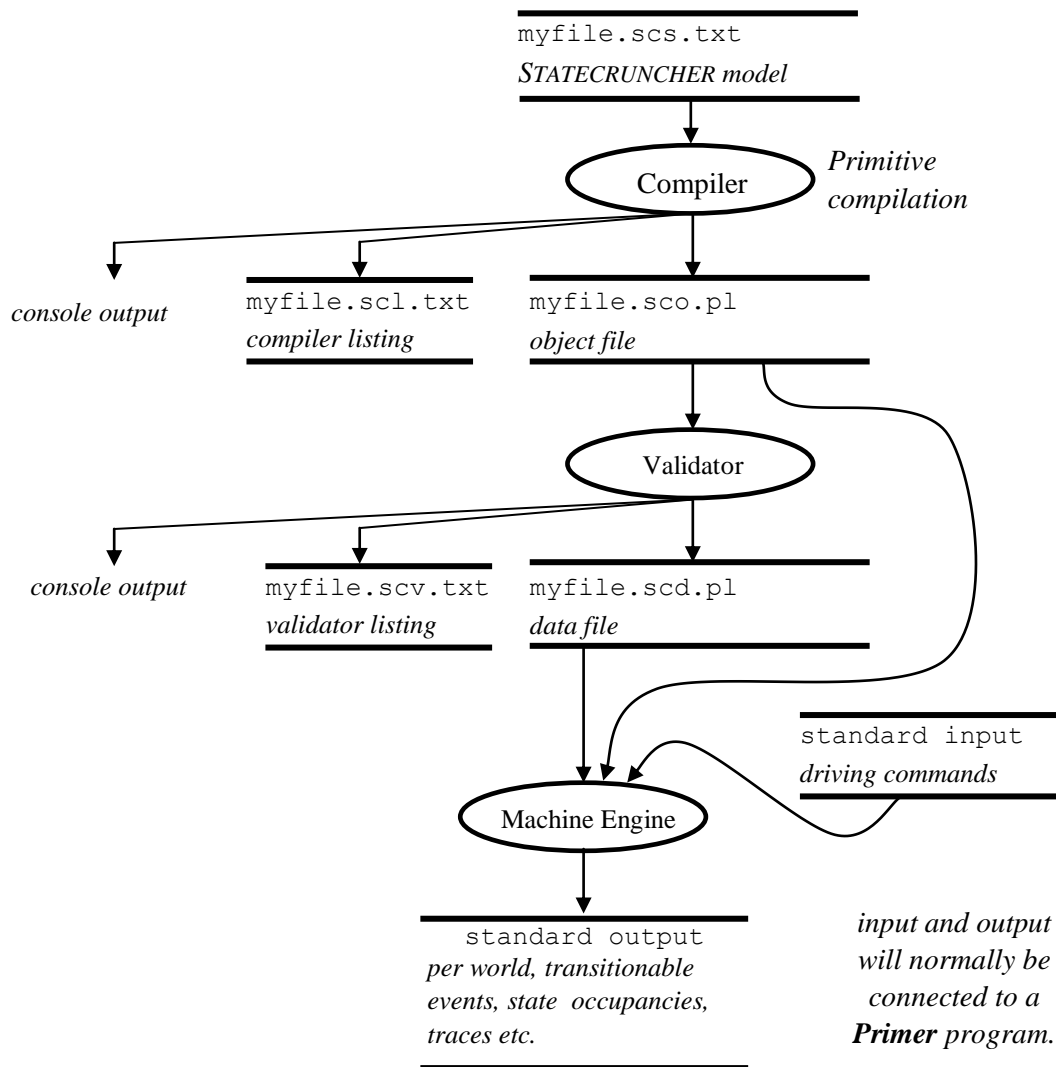
In this section, we briefly describe what STATECRUNCHER does from an input/output perspective, and how a user prepares a model. Full details of operation are given in the user manual, [StCrUser], which is designed also as a training manual.

This section also gives an indication of how STATECRUNCHER was tested.

In order to illustrate STATECRUNCHER in action in a concrete way, models of the well-known “dining philosophers” problem are developed in this section, without and with the use of a semaphore. These models are deterministic (though care must be taken to ensure that); we discuss a nondeterministic model of a television component as developed in Philips Research in section 10.

### 9.1 Data flow

The following figure shows the data flow in model compilation and event processing. *Primitive compilation* and *validation* are regarded as one compilation process by the user, as the `cp` command invokes the validator automatically, (unless the previous phase gives errors).



**Figure 162. Data flow in preparing and running models**

## 9.2 Running STATECRUNCHER

STATECRUNCHER runs under [WinProlog] and [SWI-Prolog], and is also available as an MS-DOS executable (using the WinProlog kernel, but the user need not know that the implementation language is PROLOG). Details of how to install and run STATECRUNCHER are given in [StCrManual].

As an executable, STATECRUNCHER will read commands from standard input and direct its output to standard output. The protocol between STATECRUNCHER and the primer program is the subject of a separate report [StCrPrimer].

The development cycle of a STATECRUNCHER model is basically to:

1. Load or run STATECRUNCHER.
2. Prepare a model using a text editor
3. Compile the model with the `cp` command. This includes validation and loading and entering the initial configuration of the model.
4. If there are no errors, the model is ready to be driven with `pe` (process event) commands. Otherwise, edit and re-compile.

A previously compiled model is loaded and made ready for use with the `run` command.

The user manual [StCrManual] serves as a detailed set of demonstration models, with model source code supplied, and compilation and running instructions given, and output explained.

### 9.3 Testing of STATECRUNCHER

STATECRUNCHER has been tested throughout its course of development with module tests, where test cases are defined by a PROLOG predicate as follows:

```
tc(test_name, description, predicate_under_test, pass_criterion).
```

The *test\_name* is hierarchically defined, e.g. `[sc,sy,decl,evns,2]`, so that any subtree of all tests can be run, e.g. `[sc,sy]`. A test harness, described in [StCrGP4], picks up all test cases specified and runs them, producing a test report. An example of an actual test, testing the parse of a list of event-expressions in an event declaration such as

```
event ev1 , $ev2 , ev3;
```

is:

```
tc([sc,sy,decl,evns,2],syzc(sy_event_names,A,SP,R),SP=E):-
  A=' ev1 , $ev2 , ev3 ',
  E=[g_ok, [eventnames,l_ok,
           [ [ex_evt_expr,[ex_id,ev1]],
             [ex_evt_expr,[ex_monadic,mback],[ex_id,ev2]],
             [ex_evt_expr,[ex_id,ev3]] ] ] ].
```

The `syzc` predicate is a testing auxiliary to apply a parsing predicate under test (here `sy_event_names`) to an ASCII string (the second argument, `A`, which is `' ev1 , $ev2 , ev3 '`) and to produce a status-and-parse (the third argument, `SP`), and a rest-string (fourth argument, `R`). The `SP=E` (`E` for Expected) term tests that the parse is as expected.

There are also, as system tests, 23 models to test the compiler, 31 models to test the validator, 80 models for machine engine tests, 9 models for stress testing, and many models of practical examples. Tests using these models are run using the [StCrGP4] test harness in the same way as the above parsing example. In all there are well over 10,000 tests, covering general utilities (such as permutation generation), parsing, expression evaluation, machine engine operations etc. System testing of STATECRUNCHER is described in detail in [StCrTest], where diagrams of the main models are given. Users report that STATECRUNCHER is reliable.

## 9.4 The dining philosophers

In this subsection, we show how a system taken from the CSP literature can be modelled in STATECRUNCHER. We take a fairly easy example that nevertheless illustrates the essence of CSP and which is discussed in [Hoare] and [Schneider] (and many other books): the *Dining Philosophers*. A first STATECRUNCHER model is shown, with output from a session driving it to deadlock. A refined model shows how a semaphore can be used to prevent deadlock.

### 9.4.1 The dining philosophers in CSP

The description of the exercise is given in [Hoare, p77] :

In ancient times, a wealthy philanthropist endowed a College to accommodate five eminent philosophers. Each philosopher had a room in which he could engage in his professional activity of thinking; there was also a common dining room, furnished with a circular table, surrounded by five chairs, each labelled by the name of the philosopher who was to sit in it. The names of the philosophers were  $PHIL_0$ ,  $PHIL_1$ ,  $PHIL_2$ ,  $PHIL_3$ ,  $PHIL_4$ , and they were disposed in this order anticlockwise round the table. To the left of each philosopher there was laid a golden fork, and in the centre stood a large bowl of spaghetti, which was constantly replenished.

A philosopher was expected to spend most of his time thinking; but when he felt hungry, he went to the dining room, sat down in his own chair, picked up his own fork on his left, and plunged it into the spaghetti. But such is the tangled nature of spaghetti that a second fork is required to carry it to his mouth. The philosopher therefore has also to pick up the fork on his right. When he has finished, he would put down both his forks, get up from his chair, and continue thinking. Of course, a fork can be used by only one philosopher at a time. If another philosopher wants it, he just has to wait until the fork is available again.

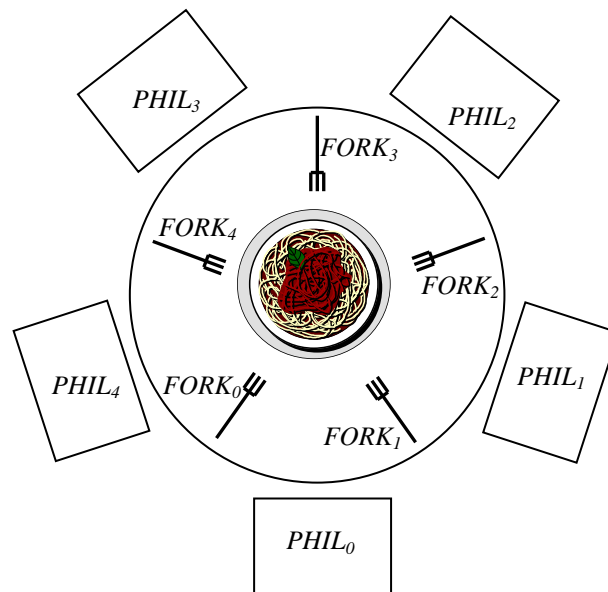


Figure 163. The dining philosophers

Schneider [Schneider, p79] also describes the problem, (but with chopsticks, not forks). Beveridge [Beveridge, p93] describes the problem, and shows a Win32 solution to the deadlock problem using *mutexes*.

The description of the behaviour in CSP is as follows, where the symbol  $p$  means addition modulo 5 and  $q$  means subtraction modulo 5.

```
PHILi=
    (i.SitsDownai.PickUpFork.iai.PicksUpFork.(ip1)ai.PutsDownFork.ia
    i.PutsDownFork.(ip1)ai.GetUpaPHILi)
FORKi=(i.PicksUpFork.iai.PutsDownFork.ia FORKi
    | (iq1).PicksUpFork.ia(iq1).PutsDownFork.ia FORKi)
PHILOS=(PHIL0 + PHIL1 + PHIL2 + PHIL3 + PHIL4)
FORKS=(FORK0 + FORK1 + FORK2 + FORK3 + FORK4)
COLLEGE=(PHILOS || FORKS)
```

## 9.4.2 The dining philosophers in STATECRUNCHER

### 9.4.2.1 The model of the dining philosophers in STATECRUNCHER

Figure 164 shows how the dining philosophers can be modelled in STATECRUNCHER.

Following the figure, a description of the model is given, then a session running the model is reproduced.

The source code of the model given later in this section. It corresponds to the figure in almost every detail.

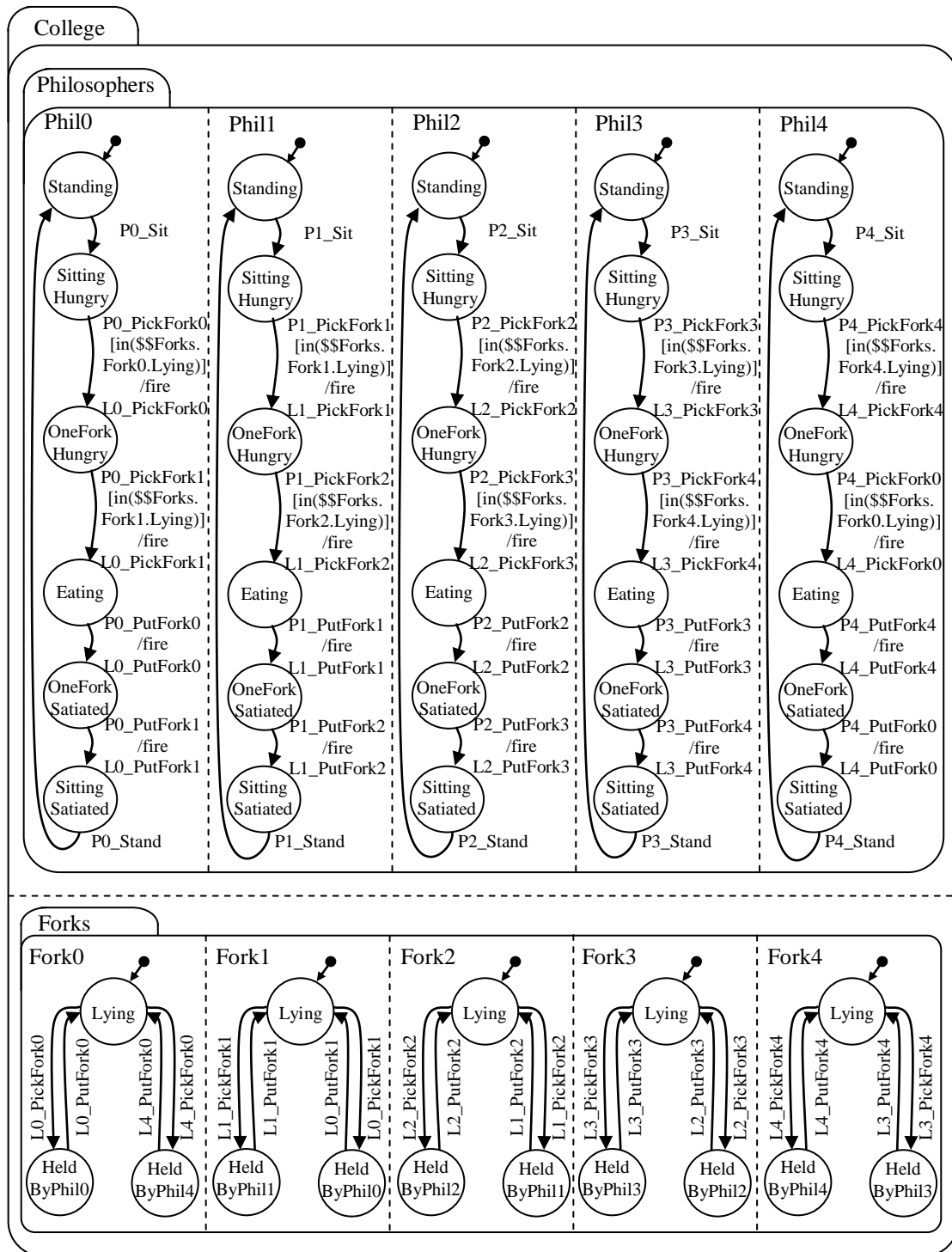
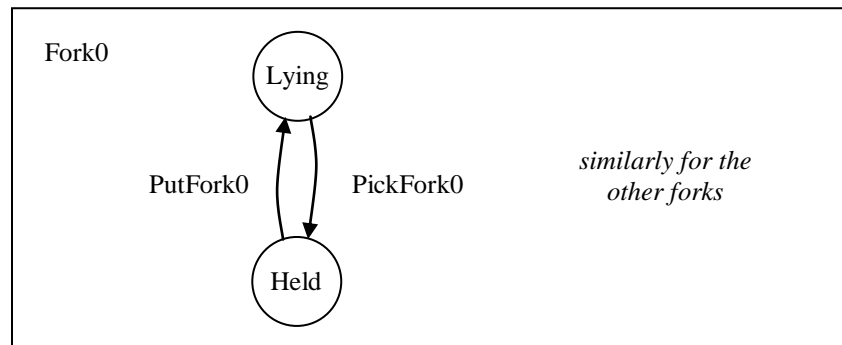


Figure 164. The dining philosophers [model t4330]

## A description of the STATECRUNCHER model, with the relationship to the CSP specification

It would have been sufficient to represent the forks as in Figure 165, but we more closely follow the CSP model as implemented in Figure 164. In the Figure 165 model, if a fork is being held, examination of the philosopher states will reveal who is holding it.



**Figure 165. Simpler Fork Model**

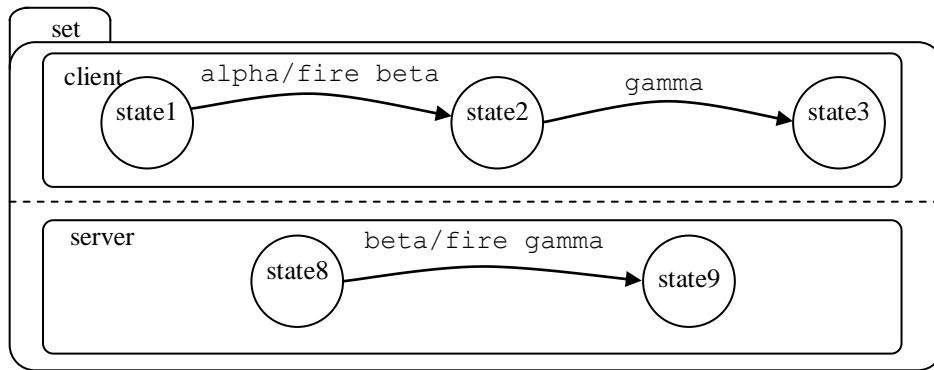
*Now there is a fundamental difference in approach between CSP and STATECRUNCHER, described in the following paragraphs.*

[Hoare p.65-66]

When two processes are brought together, the usual intention is that they will interact with each other. These interactions may be regarded as events that *require simultaneous participation of both the processes involved*.

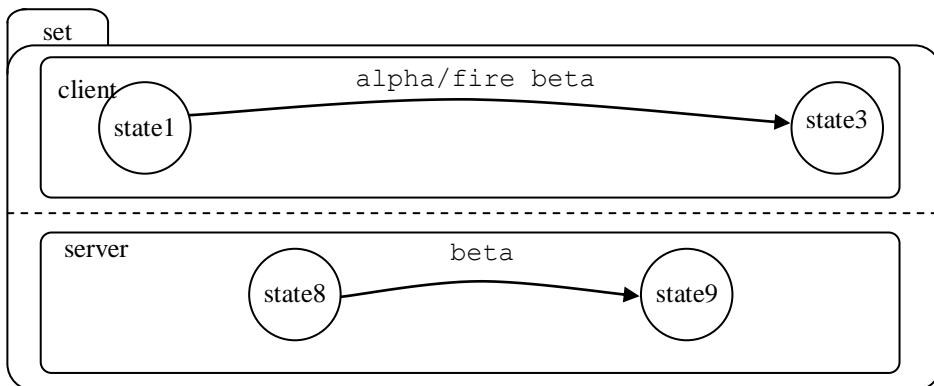
CSP has an AND condition on combined processes: they must both be able to respond to the common event. STATECRUNCHER transitionable events are transitionable if they trigger a transition in ANY (OR) set members.

The CSP model for composition is not applicable in STATECRUNCHER. The standard model for communication in STATECRUNCHER is the fired event, and a returned fired event, with "after landing" semantics, so that in Figure 166, event `alpha` is sufficient to bring client to `state3` and server to `state9`.



**Figure 166. STATECRUNCHER client-server composition (1)**

If the intermediate *state2* is never observed by the user, and the server is regarded as completing instantly, the following simplification can be used:

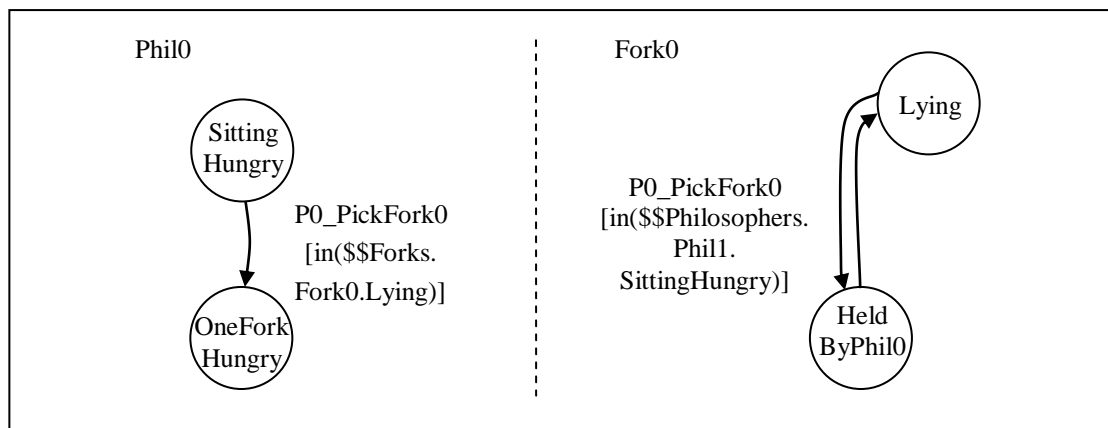


**Figure 167. STATECRUNCHER client-server composition (2)**

***Why we need the STATECRUNCHER composition paradigm***

If we allow server and client to respond to the same event, we get a race problem as follows: (we take the *Phil0* situation, but it applies to all the philosophers).





**Figure 168. Race problem**

We see that we need a condition on `P0_PickFork0`, `[in($$Forks.Fork0.Lying)]`, because without it, Phil0 can pick up a fork that is in use by Phil4.

We also put a condition on `P0_PickFork0`, `[in($$Philosophers.Phil0.SittingHungry)]`, because it causes the event to show as transitionable only when it really is. When finding transitionable events, STATECRUNCHER evaluates the condition on the transition.

We do not want a race between two transitions on

`P0_PickFork0[in($$Philosophers.Phil0.SittingHungry)]`

and

`P0_PickFork0[in($$Forks.Fork0.Lying)]`

This is because the transitions invalidate each other. But the current semantics will not allow both transitions, because the condition is re-evaluated at execution time.

Without a condition on the second transition, two race orderings will be run, and one will not execute the first transition. Two worlds will be produced, one of which is unwanted.

*The solution adopted is the fired event system between client (philosopher) and server (fork), as shown in Figure 164.* That is why the fork transitions are called `L0_PickFork0` etc., where L stands for local, as opposed to the external one initiated by the philosopher.

#### 9.4.2.2 Session with the dining philosophers [model t4330]

We allow all the philosophers to sit down, then we have them each pick a fork. The events to do this are shown in **bold font**.

```
?- cruncher.
SC:|: mm
SC:|: run t4330
...

```

```

SC:|: gc
2   statechart sc
2     set College [sc] = OCC [] **
2     set Philosophers [College,sc] = OCC [] **
2     cluster Phil0 [Philosophers,College,sc] = OCC [] **
2       leafstate Standing [Phil0,Philosophers,College,sc] = OCC [] **
2       leafstate SittingHungry [Phil0,Philosophers,College,sc] = VAC []
2       leafstate OneForkHungry [Phil0,Philosophers,College,sc] = VAC []
2       leafstate Eating [Phil0,Philosophers,College,sc] = VAC []
2       leafstate OneForkSatiated [Phil0,Philosophers,College,sc] = VAC []
2       leafstate SittingSatiated [Phil0,Philosophers,College,sc] = VAC []
2     cluster Phil1 [Philosophers,College,sc] = OCC [] **
2       leafstate Standing [Phil1,Philosophers,College,sc] = OCC [] **
2       leafstate SittingHungry [Phil1,Philosophers,College,sc] = VAC []
2       leafstate OneForkHungry [Phil1,Philosophers,College,sc] = VAC []
2       leafstate Eating [Phil1,Philosophers,College,sc] = VAC []
2       leafstate OneForkSatiated [Phil1,Philosophers,College,sc] = VAC []
2       leafstate SittingSatiated [Phil1,Philosophers,College,sc] = VAC []
2     cluster Phil2 [Philosophers,College,sc] = OCC [] **
2       leafstate Standing [Phil2,Philosophers,College,sc] = OCC [] **
2       leafstate SittingHungry [Phil2,Philosophers,College,sc] = VAC []
2       leafstate OneForkHungry [Phil2,Philosophers,College,sc] = VAC []
2       leafstate Eating [Phil2,Philosophers,College,sc] = VAC []
2       leafstate OneForkSatiated [Phil2,Philosophers,College,sc] = VAC []
2       leafstate SittingSatiated [Phil2,Philosophers,College,sc] = VAC []
2     cluster Phil3 [Philosophers,College,sc] = OCC [] **
2       leafstate Standing [Phil3,Philosophers,College,sc] = OCC [] **
2       leafstate SittingHungry [Phil3,Philosophers,College,sc] = VAC []
2       leafstate OneForkHungry [Phil3,Philosophers,College,sc] = VAC []
2       leafstate Eating [Phil3,Philosophers,College,sc] = VAC []
2       leafstate OneForkSatiated [Phil3,Philosophers,College,sc] = VAC []
2       leafstate SittingSatiated [Phil3,Philosophers,College,sc] = VAC []
2     cluster Phil4 [Philosophers,College,sc] = OCC [] **
2       leafstate Standing [Phil4,Philosophers,College,sc] = OCC [] **
2       leafstate SittingHungry [Phil4,Philosophers,College,sc] = VAC []
2       leafstate OneForkHungry [Phil4,Philosophers,College,sc] = VAC []
2       leafstate Eating [Phil4,Philosophers,College,sc] = VAC []
2       leafstate OneForkSatiated [Phil4,Philosophers,College,sc] = VAC []
2       leafstate SittingSatiated [Phil4,Philosophers,College,sc] = VAC []
2     set Forks [College,sc] = OCC [] **
2     cluster Fork0 [Forks,College,sc] = OCC [] **
2       leafstate Lying [Fork0,Forks,College,sc] = OCC [] **
2       leafstate HeldByPhil0 [Fork0,Forks,College,sc] = VAC []
2       leafstate HeldByPhil4 [Fork0,Forks,College,sc] = VAC []
2     cluster Fork1 [Forks,College,sc] = OCC [] **
2       leafstate Lying [Fork1,Forks,College,sc] = OCC [] **
2       leafstate HeldByPhil1 [Fork1,Forks,College,sc] = VAC []
2       leafstate HeldByPhil0 [Fork1,Forks,College,sc] = VAC []
2     cluster Fork2 [Forks,College,sc] = OCC [] **
2       leafstate Lying [Fork2,Forks,College,sc] = OCC [] **
2       leafstate HeldByPhil2 [Fork2,Forks,College,sc] = VAC []
2       leafstate HeldByPhil1 [Fork2,Forks,College,sc] = VAC []
2     cluster Fork3 [Forks,College,sc] = OCC [] **
2       leafstate Lying [Fork3,Forks,College,sc] = OCC [] **
2       leafstate HeldByPhil3 [Fork3,Forks,College,sc] = VAC []
2       leafstate HeldByPhil2 [Fork3,Forks,College,sc] = VAC []
2     cluster Fork4 [Forks,College,sc] = OCC [] **
2       leafstate Lying [Fork4,Forks,College,sc] = OCC [] **
2       leafstate HeldByPhil4 [Fork4,Forks,College,sc] = VAC []
2       leafstate HeldByPhil3 [Fork4,Forks,College,sc] = VAC []
2
2   TRACE =[]
2   TREV [[P0_Sit,[sc]],0,[],[external,[sc]]]
2   TREV [[P1_Sit,[sc]],0,[],[external,[sc]]]
2   TREV [[P2_Sit,[sc]],0,[],[external,[sc]]]
2   TREV [[P3_Sit,[sc]],0,[],[external,[sc]]]
2   TREV [[P4_Sit,[sc]],0,[],[external,[sc]]]
2   TREV [[L0_PickFork0,[sc]],0,[],[internal,[sc]]]

```

```

2   TREV [[L4_PickFork0,[sc]],0,[],[internal,[sc]]]
2   TREV [[L1_PickFork1,[sc]],0,[],[internal,[sc]]]
2   TREV [[L0_PickFork1,[sc]],0,[],[internal,[sc]]]
2   TREV [[L2_PickFork2,[sc]],0,[],[internal,[sc]]]
2   TREV [[L1_PickFork2,[sc]],0,[],[internal,[sc]]]
2   TREV [[L3_PickFork3,[sc]],0,[],[internal,[sc]]]
2   TREV [[L2_PickFork3,[sc]],0,[],[internal,[sc]]]
2   TREV [[L4_PickFork4,[sc]],0,[],[internal,[sc]]]
2   TREV [[L3_PickFork4,[sc]],0,[],[internal,[sc]]]

```

```

outworlds=[2]
number of outworlds=1

```

```

SC:|: pe P0_Sit
SC:|: pe P1_Sit
SC:|: pe P2_Sit
SC:|: pe P3_Sit
SC:|: pe P4_Sit
SC:|: gc (occupied leaf states and external transitionable events only)
7   leafstate SittingHungry [Phil0,Philosophers,College,sc] = OCC [] **
7   leafstate SittingHungry [Phil1,Philosophers,College,sc] = OCC [] **
7   leafstate SittingHungry [Phil2,Philosophers,College,sc] = OCC [] **
7   leafstate SittingHungry [Phil3,Philosophers,College,sc] = OCC [] **
7   leafstate SittingHungry [Phil4,Philosophers,College,sc] = OCC [] **
7   leafstate Lying [Fork0,Forks,College,sc] = OCC [] **
7   leafstate Lying [Fork1,Forks,College,sc] = OCC [] **
7   leafstate Lying [Fork2,Forks,College,sc] = OCC [] **
7   leafstate Lying [Fork3,Forks,College,sc] = OCC [] **
7   leafstate Lying [Fork4,Forks,College,sc] = OCC [] **
7   TRACE =[]
7   TREV [[P0_PickFork0,[sc]],0,[],[external,[sc]]]
7   TREV [[P1_PickFork1,[sc]],0,[],[external,[sc]]]
7   TREV [[P2_PickFork2,[sc]],0,[],[external,[sc]]]
7   TREV [[P3_PickFork3,[sc]],0,[],[external,[sc]]]
7   TREV [[P4_PickFork4,[sc]],0,[],[external,[sc]]]

```

```

outworlds=[7]
number of outworlds=1

```

```

SC:|: pe P0_PickFork0
SC:|: gc (occupied leaf states and external transitionable events only)
9   leafstate OneForkHungry [Phil0,Philosophers,College,sc] = OCC [] **
9   leafstate SittingHungry [Phil1,Philosophers,College,sc] = OCC [] **
9   leafstate SittingHungry [Phil2,Philosophers,College,sc] = OCC [] **
9   leafstate SittingHungry [Phil3,Philosophers,College,sc] = OCC [] **
9   leafstate SittingHungry [Phil4,Philosophers,College,sc] = OCC [] **
9   leafstate HeldByPhil0 [Fork0,Forks,College,sc] = OCC [] **
9   leafstate Lying [Fork1,Forks,College,sc] = OCC [] **
9   leafstate Lying [Fork2,Forks,College,sc] = OCC [] **
9   leafstate Lying [Fork3,Forks,College,sc] = OCC [] **
9   leafstate Lying [Fork4,Forks,College,sc] = OCC [] **
9   TRACE =[]
9   TREV [[P0_PickFork1,[sc]],0,[],[external,[sc]]]
9   TREV [[P1_PickFork1,[sc]],0,[],[external,[sc]]]
9   TREV [[P2_PickFork2,[sc]],0,[],[external,[sc]]]
9   TREV [[P3_PickFork3,[sc]],0,[],[external,[sc]]]
9   TREV [[P4_PickFork4,[sc]],0,[],[external,[sc]]]

```

```

outworlds=[9]
number of outworlds=1

```

```

SC:|: pe P1_PickFork1
SC:|: pe P2_PickFork2
SC:|: pe P3_PickFork3
SC:|: pe P4_PickFork4
SC:|: gc (unabridged)

```

```

17 statechart sc
17   set College [sc] = OCC [] **
17     set Philosophers [College,sc] = OCC [] **
17       cluster Phil0 [Philosophers,College,sc] = OCC [] **
17         leafstate Standing [Phil0,Philosophers,College,sc] = VAC []
17         leafstate SittingHungry [Phil0,Philosophers,College,sc] = VAC []
17         leafstate OneForkHungry [Phil0,Philosophers,College,sc] = OCC [] **
17         leafstate Eating [Phil0,Philosophers,College,sc] = VAC []
17         leafstate OneForkSatiated [Phil0,Philosophers,College,sc] = VAC []
17         leafstate SittingSatiated [Phil0,Philosophers,College,sc] = VAC []
17       cluster Phil1 [Philosophers,College,sc] = OCC [] **
17         leafstate Standing [Phil1,Philosophers,College,sc] = VAC []
17         leafstate SittingHungry [Phil1,Philosophers,College,sc] = VAC []
17         leafstate OneForkHungry [Phil1,Philosophers,College,sc] = OCC [] **
17         leafstate Eating [Phil1,Philosophers,College,sc] = VAC []
17         leafstate OneForkSatiated [Phil1,Philosophers,College,sc] = VAC []
17         leafstate SittingSatiated [Phil1,Philosophers,College,sc] = VAC []
17       cluster Phil2 [Philosophers,College,sc] = OCC [] **
17         leafstate Standing [Phil2,Philosophers,College,sc] = VAC []
17         leafstate SittingHungry [Phil2,Philosophers,College,sc] = VAC []
17         leafstate OneForkHungry [Phil2,Philosophers,College,sc] = OCC [] **
17         leafstate Eating [Phil2,Philosophers,College,sc] = VAC []
17         leafstate OneForkSatiated [Phil2,Philosophers,College,sc] = VAC []
17         leafstate SittingSatiated [Phil2,Philosophers,College,sc] = VAC []
17       cluster Phil3 [Philosophers,College,sc] = OCC [] **
17         leafstate Standing [Phil3,Philosophers,College,sc] = VAC []
17         leafstate SittingHungry [Phil3,Philosophers,College,sc] = VAC []
17         leafstate OneForkHungry [Phil3,Philosophers,College,sc] = OCC [] **
17         leafstate Eating [Phil3,Philosophers,College,sc] = VAC []
17         leafstate OneForkSatiated [Phil3,Philosophers,College,sc] = VAC []
17         leafstate SittingSatiated [Phil3,Philosophers,College,sc] = VAC []
17       cluster Phil4 [Philosophers,College,sc] = OCC [] **
17         leafstate Standing [Phil4,Philosophers,College,sc] = VAC []
17         leafstate SittingHungry [Phil4,Philosophers,College,sc] = VAC []
17         leafstate OneForkHungry [Phil4,Philosophers,College,sc] = OCC [] **
17         leafstate Eating [Phil4,Philosophers,College,sc] = VAC []
17         leafstate OneForkSatiated [Phil4,Philosophers,College,sc] = VAC []
17         leafstate SittingSatiated [Phil4,Philosophers,College,sc] = VAC []
17     set Forks [College,sc] = OCC [] **
17       cluster Fork0 [Forks,College,sc] = OCC [] **
17         leafstate Lying [Fork0,Forks,College,sc] = VAC []
17         leafstate HeldByPhil0 [Fork0,Forks,College,sc] = OCC [] **
17         leafstate HeldByPhil4 [Fork0,Forks,College,sc] = VAC []
17       cluster Fork1 [Forks,College,sc] = OCC [] **
17         leafstate Lying [Fork1,Forks,College,sc] = VAC []
17         leafstate HeldByPhil1 [Fork1,Forks,College,sc] = OCC [] **
17         leafstate HeldByPhil0 [Fork1,Forks,College,sc] = VAC []
17       cluster Fork2 [Forks,College,sc] = OCC [] **
17         leafstate Lying [Fork2,Forks,College,sc] = VAC []
17         leafstate HeldByPhil2 [Fork2,Forks,College,sc] = OCC [] **
17         leafstate HeldByPhil1 [Fork2,Forks,College,sc] = VAC []
17       cluster Fork3 [Forks,College,sc] = OCC [] **
17         leafstate Lying [Fork3,Forks,College,sc] = VAC []
17         leafstate HeldByPhil3 [Fork3,Forks,College,sc] = OCC [] **
17         leafstate HeldByPhil2 [Fork3,Forks,College,sc] = VAC []
17       cluster Fork4 [Forks,College,sc] = OCC [] **
17         leafstate Lying [Fork4,Forks,College,sc] = VAC []
17         leafstate HeldByPhil4 [Fork4,Forks,College,sc] = OCC [] **
17         leafstate HeldByPhil3 [Fork4,Forks,College,sc] = VAC []
17   TRACE =[]
17   TREV [[L0_PutFork0,[sc]],0,[],[internal,[sc]]]
17   TREV [[L1_PutFork1,[sc]],0,[],[internal,[sc]]]
17   TREV [[L2_PutFork2,[sc]],0,[],[internal,[sc]]]
17   TREV [[L3_PutFork3,[sc]],0,[],[internal,[sc]]]
17   TREV [[L4_PutFork4,[sc]],0,[],[internal,[sc]]]

```

outworlds=[17]

number of outworlds=1  
SC:|:

***There are no transitionable events at the external PCO: deadlock!***

### 9.4.3 Introduction of a semaphore on picking up forks

#### 9.4.3.1 The model with semaphores

*Hoare* discusses the following solutions to the deadlock:

- Agree that one philosopher should always pick up the wrong fork first.
- Buy more forks.
- Employ a footman to restrict the number of seated philosophers to a maximum of 4.

*Schneider* adds

- Allowing a philosopher to release a fork if he holds only one.

Neither considers the use of a *semaphore*, which is the obvious software-technical choice. *Beveridge* shows how to use Win32 *mutexes* (mutual exclusions, which are essentially semaphores with a maximum count of one), to solve the problem. The mutexes enable the philosophers to wait for two forks atomically.

In order to reduce unnecessary elements of the model, we make the following simplifications:

- We eliminate the *Standing* state, and we call the sitting-with-no-forks-held-or-requested the *Thinking* state. The philosophers now do their thinking at the table.
- We restrict fork states to *Lying* and *Held*. The forks respond to events *PickFork0* etc., in which no account is taken of who is interacting with the fork.
- We add STATECRUNCHER traces, which are not the same as CSP traces – they are a record of specific selected outputs, generated by the trace(...) function. They are used in black-box testing, representing observable outputs. We record traces on entering and exiting the *Eating* state: ▲ trace(P4Eat) and ▼ trace(P4Stp).

We also shorten the names of some items for convenience. We also distinguish between various categories of event:

#### ***External events***

- A philosopher has a *Pang* of hunger
- A philosopher has eaten enough and becomes *Full*

#### ***Events for communication with the semaphore***

- *Request*, *Acquire* and *Release* a pair of forks

#### ***Internal events***

- Fork status administration

In the model, the different events have different PCOs (Points of Control and Observation). The model should be driven by external events only.

The pairwise fork operations work broadly as follows. The *Reset* state is for when there is no outstanding request. Whenever in a fork-pair cluster a *Request* for a pair of forks is made, it is either satisfied, broadcasting the *Acquisition*, with no change of state here, or the cluster goes into the *Requested* State. Whenever, elsewhere, one of the participating forks is *Released*, a broadcast event causes a new *Try* in this cluster to be made to satisfy the request. By the same token, when in the present cluster the forks are *Released*, two *Try* events are broadcast so that other clusters can respond to them, each involving one of the forks just released.

The self-transitions on *Try01*, *Try12* etc. are unnecessary, are not present in the implemented model. However, such transitions could be used to trace what has happened, and could be useful in debugging a model.

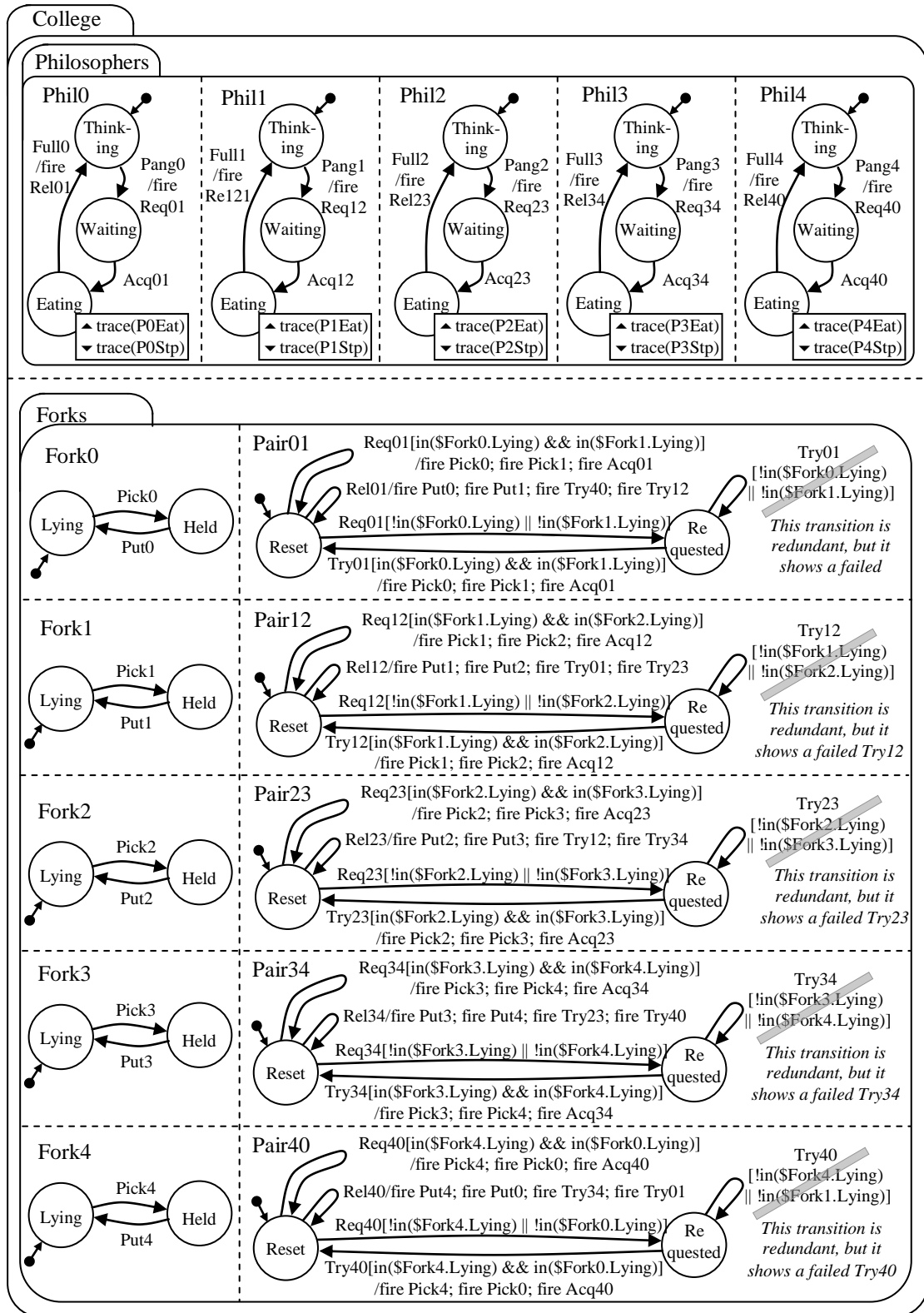


Figure 169. Model with semaphores [model t4335]

### 9.4.3.2 A session with the model with semaphores [model t4335]

```
SC:|: gc
2   statechart sc
2     set College [sc] = OCC [] **
2     set Philosophers [College,sc] = OCC [] **
2       cluster Phil0 [Philosophers,College,sc] = OCC [] **
2         leafstate Thinking [Phil0,Philosophers,College,sc] = OCC [] **
2         leafstate Waiting [Phil0,Philosophers,College,sc] = VAC []
2         leafstate Eating [Phil0,Philosophers,College,sc] = VAC []
2       cluster Phil1 [Philosophers,College,sc] = OCC [] **
2         leafstate Thinking [Phil1,Philosophers,College,sc] = OCC [] **
2         leafstate Waiting [Phil1,Philosophers,College,sc] = VAC []
2         leafstate Eating [Phil1,Philosophers,College,sc] = VAC []
2       cluster Phil2 [Philosophers,College,sc] = OCC [] **
2         leafstate Thinking [Phil2,Philosophers,College,sc] = OCC [] **
2         leafstate Waiting [Phil2,Philosophers,College,sc] = VAC []
2         leafstate Eating [Phil2,Philosophers,College,sc] = VAC []
2       cluster Phil3 [Philosophers,College,sc] = OCC [] **
2         leafstate Thinking [Phil3,Philosophers,College,sc] = OCC [] **
2         leafstate Waiting [Phil3,Philosophers,College,sc] = VAC []
2         leafstate Eating [Phil3,Philosophers,College,sc] = VAC []
2       cluster Phil4 [Philosophers,College,sc] = OCC [] **
2         leafstate Thinking [Phil4,Philosophers,College,sc] = OCC [] **
2         leafstate Waiting [Phil4,Philosophers,College,sc] = VAC []
2         leafstate Eating [Phil4,Philosophers,College,sc] = VAC []
2     set Forks [College,sc] = OCC [] **
2       cluster Fork0 [Forks,College,sc] = OCC [] **
2         leafstate Lying [Fork0,Forks,College,sc] = OCC [] **
2         leafstate Held [Fork0,Forks,College,sc] = VAC []
2       cluster Fork1 [Forks,College,sc] = OCC [] **
2         leafstate Lying [Fork1,Forks,College,sc] = OCC [] **
2         leafstate Held [Fork1,Forks,College,sc] = VAC []
2       cluster Fork2 [Forks,College,sc] = OCC [] **
2         leafstate Lying [Fork2,Forks,College,sc] = OCC [] **
2         leafstate Held [Fork2,Forks,College,sc] = VAC []
2       cluster Fork3 [Forks,College,sc] = OCC [] **
2         leafstate Lying [Fork3,Forks,College,sc] = OCC [] **
2         leafstate Held [Fork3,Forks,College,sc] = VAC []
2       cluster Fork4 [Forks,College,sc] = OCC [] **
2         leafstate Lying [Fork4,Forks,College,sc] = OCC [] **
2         leafstate Held [Fork4,Forks,College,sc] = VAC []
2       cluster Pair01 [Forks,College,sc] = OCC [] **
2         leafstate Reset [Pair01,Forks,College,sc] = OCC [] **
2         leafstate Requested [Pair01,Forks,College,sc] = VAC []
2       cluster Pair12 [Forks,College,sc] = OCC [] **
2         leafstate Reset [Pair12,Forks,College,sc] = OCC [] **
2         leafstate Requested [Pair12,Forks,College,sc] = VAC []
2       cluster Pair23 [Forks,College,sc] = OCC [] **
2         leafstate Reset [Pair23,Forks,College,sc] = OCC [] **
2         leafstate Requested [Pair23,Forks,College,sc] = VAC []
2       cluster Pair34 [Forks,College,sc] = OCC [] **
2         leafstate Reset [Pair34,Forks,College,sc] = OCC [] **
2         leafstate Requested [Pair34,Forks,College,sc] = VAC []
2       cluster Pair40 [Forks,College,sc] = OCC [] **
2         leafstate Reset [Pair40,Forks,College,sc] = OCC [] **
2         leafstate Requested [Pair40,Forks,College,sc] = VAC []
2   TRACE =[]
2   TREV [[Pang0,[sc]],0,[],[external,[sc]]]
2   TREV [[Pang1,[sc]],0,[],[external,[sc]]]
2   TREV [[Pang2,[sc]],0,[],[external,[sc]]]
2   TREV [[Pang3,[sc]],0,[],[external,[sc]]]
2   TREV [[Pang4,[sc]],0,[],[external,[sc]]]
2   TREV [[Pick0,[sc]],0,[],[internal,[sc]]]
```



```

2   TREV [[Pick1,[sc]],0,[],[internal,[sc]]]
2   TREV [[Pick2,[sc]],0,[],[internal,[sc]]]
2   TREV [[Pick3,[sc]],0,[],[internal,[sc]]]
2   TREV [[Pick4,[sc]],0,[],[internal,[sc]]]
2   TREV [[Req01,[sc]],0,[],[composition,[sc]]]
2   TREV [[Rel01,[sc]],0,[],[composition,[sc]]]
2   TREV [[Req12,[sc]],0,[],[composition,[sc]]]
2   TREV [[Rel12,[sc]],0,[],[composition,[sc]]]
2   TREV [[Req23,[sc]],0,[],[composition,[sc]]]
2   TREV [[Rel23,[sc]],0,[],[composition,[sc]]]
2   TREV [[Req34,[sc]],0,[],[composition,[sc]]]
2   TREV [[Rel34,[sc]],0,[],[composition,[sc]]]
2   TREV [[Req40,[sc]],0,[],[composition,[sc]]]
2   TREV [[Rel40,[sc]],0,[],[composition,[sc]]]

outworlds=[2]
number of outworlds=1

SC:|: pe Pang0
SC:|: pe Pang1
SC:|: pe Pang2
SC:|: pe Pang3
SC:|: pe Pang4
SC:|: gt
20  TRACE =[P2Eat,P0Eat]
SC:|: pe Full10
SC:|: gt
30  TRACE =[P4Eat,P0Stp,P2Eat,P0Eat]
SC:|: pe Full14
SC:|: gt
35  TRACE =[P4Stp,P4Eat,P0Stp,P2Eat,P0Eat]
SC:|: pe Full12
SC:|: gt
50  TRACE =[P3Eat,P1Eat,P2Stp,P4Stp,P4Eat,P0Stp,P2Eat,P0Eat]
SC:|: gc
50  statechart sc
50      set College [sc] = OCC [] **
50          set Philosophers [College,sc] = OCC [] **
50              cluster Phil0 [Philosophers,College,sc] = OCC [] **
50                  leafstate Thinking [Phil0,Philosophers,College,sc] = OCC [] **
50                  leafstate Waiting [Phil0,Philosophers,College,sc] = VAC []
50                  leafstate Eating [Phil0,Philosophers,College,sc] = VAC []
50              cluster Phil1 [Philosophers,College,sc] = OCC [] **
50                  leafstate Thinking [Phil1,Philosophers,College,sc] = VAC []
50                  leafstate Waiting [Phil1,Philosophers,College,sc] = VAC []
50                  leafstate Eating [Phil1,Philosophers,College,sc] = OCC [] **
50              cluster Phil2 [Philosophers,College,sc] = OCC [] **
50                  leafstate Thinking [Phil2,Philosophers,College,sc] = OCC [] **
50                  leafstate Waiting [Phil2,Philosophers,College,sc] = VAC []
50                  leafstate Eating [Phil2,Philosophers,College,sc] = VAC []
50              cluster Phil3 [Philosophers,College,sc] = OCC [] **
50                  leafstate Thinking [Phil3,Philosophers,College,sc] = VAC []
50                  leafstate Waiting [Phil3,Philosophers,College,sc] = VAC []
50                  leafstate Eating [Phil3,Philosophers,College,sc] = OCC [] **
50              cluster Phil4 [Philosophers,College,sc] = OCC [] **
50                  leafstate Thinking [Phil4,Philosophers,College,sc] = OCC [] **
50                  leafstate Waiting [Phil4,Philosophers,College,sc] = VAC []
50                  leafstate Eating [Phil4,Philosophers,College,sc] = VAC []
50          set Forks [College,sc] = OCC [] **
50              cluster Fork0 [Forks,College,sc] = OCC [] **
50                  leafstate Lying [Fork0,Forks,College,sc] = OCC [] **
50                  leafstate Held [Fork0,Forks,College,sc] = VAC []
50              cluster Fork1 [Forks,College,sc] = OCC [] **
50                  leafstate Lying [Fork1,Forks,College,sc] = VAC []
50                  leafstate Held [Fork1,Forks,College,sc] = OCC [] **
50              cluster Fork2 [Forks,College,sc] = OCC [] **
50                  leafstate Lying [Fork2,Forks,College,sc] = VAC []

```

```

50         leafstate Held [Fork2,Forks,College,sc] = OCC [] **
50     cluster Fork3 [Forks,College,sc] = OCC [] **
50         leafstate Lying [Fork3,Forks,College,sc] = VAC []
50         leafstate Held [Fork3,Forks,College,sc] = OCC [] **
50     cluster Fork4 [Forks,College,sc] = OCC [] **
50         leafstate Lying [Fork4,Forks,College,sc] = VAC []
50         leafstate Held [Fork4,Forks,College,sc] = OCC [] **
50     cluster Pair01 [Forks,College,sc] = OCC [] **
50         leafstate Reset [Pair01,Forks,College,sc] = OCC [] **
50         leafstate Requested [Pair01,Forks,College,sc] = VAC []
50     cluster Pair12 [Forks,College,sc] = OCC [] **
50         leafstate Reset [Pair12,Forks,College,sc] = OCC [] **
50         leafstate Requested [Pair12,Forks,College,sc] = VAC []
50     cluster Pair23 [Forks,College,sc] = OCC [] **
50         leafstate Reset [Pair23,Forks,College,sc] = OCC [] **
50         leafstate Requested [Pair23,Forks,College,sc] = VAC []
50     cluster Pair34 [Forks,College,sc] = OCC [] **
50         leafstate Reset [Pair34,Forks,College,sc] = OCC [] **
50         leafstate Requested [Pair34,Forks,College,sc] = VAC []
50     cluster Pair40 [Forks,College,sc] = OCC [] **
50         leafstate Reset [Pair40,Forks,College,sc] = OCC [] **
50         leafstate Requested [Pair40,Forks,College,sc] = VAC []
50 TRACE = [P3Eat, P1Eat, P2Stp, P4Stp, P4Eat, P0Stp, P2Eat, P0Eat]
50 TREV [[Pang0, [sc]], 0, [], [external, [sc]]]
50 TREV [[Full1, [sc]], 0, [], [external, [sc]]]
50 TREV [[Pang2, [sc]], 0, [], [external, [sc]]]
50 TREV [[Full3, [sc]], 0, [], [external, [sc]]]
50 TREV [[Pang4, [sc]], 0, [], [external, [sc]]]
50 TREV [[Pick0, [sc]], 0, [], [internal, [sc]]]
50 TREV [[Put1, [sc]], 0, [], [internal, [sc]]]
50 TREV [[Put2, [sc]], 0, [], [internal, [sc]]]
50 TREV [[Put3, [sc]], 0, [], [internal, [sc]]]
50 TREV [[Put4, [sc]], 0, [], [internal, [sc]]]
50 TREV [[Rel01, [sc]], 0, [], [composition, [sc]]]
50 TREV [[Req01, [sc]], 0, [], [composition, [sc]]]
50 TREV [[Rel12, [sc]], 0, [], [composition, [sc]]]
50 TREV [[Req12, [sc]], 0, [], [composition, [sc]]]
50 TREV [[Rel23, [sc]], 0, [], [composition, [sc]]]
50 TREV [[Req23, [sc]], 0, [], [composition, [sc]]]
50 TREV [[Rel34, [sc]], 0, [], [composition, [sc]]]
50 TREV [[Req34, [sc]], 0, [], [composition, [sc]]]
50 TREV [[Rel40, [sc]], 0, [], [composition, [sc]]]
50 TREV [[Req40, [sc]], 0, [], [composition, [sc]]]

```

```

outworlds=[50]
number of outworlds=1
SC:|:

```

### 9.4.3.3 Diagram of the events

- Shading shows the fork in use
- **Bold font** shows the change(s) due to the last event

Event	PHIL0	PHIL1	PHIL2	PHIL3	PHIL4
<i>initial state</i>	Thinking	Thinking	Thinking	Thinking	Thinking
Pang0	<b>Eating</b>	Thinking	Thinking	Thinking	Thinking
Pang1	Eating	<b>Waiting</b>	Thinking	Thinking	Thinking
Pang2	Eating	Waiting	<b>Eating</b>	Thinking	Thinking
Pang3	Eating	Waiting	Eating	<b>Waiting</b>	Thinking
Pang4	Eating	Waiting	Eating	Waiting	<b>Waiting</b>
Full0	<b>Thinking</b>	Waiting	Eating	Waiting	<b>Eating</b>
Full4	Thinking	Waiting	Eating	Waiting	<b>Thinking</b>
Full2	Thinking	<b>Eating</b>	<b>Thinking</b>	<b>Eating</b>	Thinking

Table 16. Diagram of the events

### 9.4.4 Conclusion on the dining philosophers

This section has shown how a typical client-server application is modelled in STATECRUNCHER, providing a direct comparison with a well-known example in the literature. Both STATECRUNCHER and CSP are amenable to the problem, but the emphasis is different: STATECRUNCHER is a state machine engine providing the white box or black box oracle to tests and does not support calculus manipulations; CSP is a *calculus* which is used to prove properties of composed systems.

### 9.4.5 Source listings of models

#### 9.4.5.1 Source listing of the dining philosophers without semaphores [model t4330]

```
//-----
// Module:    Philosophers.scs.txt
// Author:    Graham Thomason, Philips Digital Systems Laboratories, Redhill
// Date:      18 July, 2003
// Purpose:   StateCruncher model: The Dining philosophers [Hoare, p.75]
//
// Copyright (C) 2003 Philips Electronics N.V.
//-----1-----2-----3-----4-----5-----6-----7-----8-----

statechart sc(College)

PCO external;
PCO internal;

event P0_Sit, P0_Stand @external;
event P1_Sit, P1_Stand @external;
event P2_Sit, P2_Stand @external;
event P3_Sit, P3_Stand @external;
event P4_Sit, P4_Stand @external;

event P0_PickFork0, P0_PickFork1, P0_PutFork0, P0_PutFork1 @external;
event P1_PickFork1, P1_PickFork2, P1_PutFork1, P1_PutFork2 @external;
```

```

event P2_PickFork2, P2_PickFork3, P2_PutFork2, P2_PutFork3 @external;
event P3_PickFork3, P3_PickFork4, P3_PutFork3, P3_PutFork4 @external;
event P4_PickFork4, P4_PickFork0, P4_PutFork4, P4_PutFork0 @external;

event L0_PickFork0, L1_PickFork1, L2_PickFork2, L3_PickFork3, L4_PickFork4 @internal;
event L0_PutFork0, L1_PutFork1, L2_PutFork2, L3_PutFork3, L4_PutFork4 @internal;
event L0_PickFork1, L1_PickFork2, L2_PickFork3, L3_PickFork4, L4_PickFork0 @internal;
event L0_PutFork1, L1_PutFork2, L2_PutFork3, L3_PutFork4, L4_PutFork0 @internal;

set College(Philosophers,Forks)

set Philosophers(Phil0,Phil1,Phil2,Phil3,Phil4)

cluster Phil0( \
  Standing,SittingHungry,OneForkHungry,Eating,OneForkSatiated,SittingSatiated)
state Standing {P0_Sit->SittingHungry;}
state SittingHungry {P0_PickFork0[in($$Forks.Fork0.Lying)]->OneForkHungry \
  {fire L0_PickFork0;};}
state OneForkHungry {P0_PickFork1[in($$Forks.Fork1.Lying)]->Eating \
  {fire L0_PickFork1;};}
state Eating {P0_PutFork0->OneForkSatiated \
  {fire L0_PutFork0;};}
state OneForkSatiated {P0_PutFork1->SittingSatiated \
  {fire L0_PutFork1;};}
state SittingSatiated {P0_Stand->Standing;}

cluster Phil1( \
  Standing,SittingHungry,OneForkHungry,Eating,OneForkSatiated,SittingSatiated)
state Standing {P1_Sit->SittingHungry;}
state SittingHungry {P1_PickFork1[in($$Forks.Fork1.Lying)]->OneForkHungry \
  {fire L1_PickFork1;};}
state OneForkHungry {P1_PickFork2[in($$Forks.Fork2.Lying)]->Eating \
  {fire L1_PickFork2;};}
state Eating {P1_PutFork1->OneForkSatiated \
  {fire L1_PutFork1;};}
state OneForkSatiated {P1_PutFork2->SittingSatiated \
  {fire L1_PutFork2;};}
state SittingSatiated {P1_Stand->Standing;}

cluster Phil2( \
  Standing,SittingHungry,OneForkHungry,Eating,OneForkSatiated,SittingSatiated)
state Standing {P2_Sit->SittingHungry;}
state SittingHungry {P2_PickFork2[in($$Forks.Fork2.Lying)]->OneForkHungry \
  {fire L2_PickFork2;};}
state OneForkHungry {P2_PickFork3[in($$Forks.Fork3.Lying)]->Eating \
  {fire L2_PickFork3;};}
state Eating {P2_PutFork2->OneForkSatiated \
  {fire L2_PutFork2;};}
state OneForkSatiated {P2_PutFork3->SittingSatiated \
  {fire L2_PutFork3;};}
state SittingSatiated {P2_Stand->Standing;}

cluster Phil3( \
  Standing,SittingHungry,OneForkHungry,Eating,OneForkSatiated,SittingSatiated)
state Standing {P3_Sit->SittingHungry;}
state SittingHungry {P3_PickFork3[in($$Forks.Fork3.Lying)]->OneForkHungry \
  {fire L3_PickFork3;};}
state OneForkHungry {P3_PickFork4[in($$Forks.Fork4.Lying)]->Eating \
  {fire L3_PickFork4;};}
state Eating {P3_PutFork3->OneForkSatiated \
  {fire L3_PutFork3;};}
state OneForkSatiated {P3_PutFork4->SittingSatiated \
  {fire L3_PutFork4;};}
state SittingSatiated {P3_Stand->Standing;}

cluster Phil4( \
  Standing,SittingHungry,OneForkHungry,Eating,OneForkSatiated,SittingSatiated)
state Standing {P4_Sit->SittingHungry;}
state SittingHungry {P4_PickFork4[in($$Forks.Fork4.Lying)]->OneForkHungry \
  {fire L4_PickFork4;};}
state OneForkHungry {P4_PickFork0[in($$Forks.Fork0.Lying)]->Eating \
  {fire L4_PickFork0;};}
state Eating {P4_PutFork4->OneForkSatiated \

```

```

                {fire L4_PutFork4;};
state OneForkSatiated {P4_PutFork0->SittingSatiated \
                {fire L4_PutFork0;};}
state SittingSatiated {P4_Stand->Standing;}

set Forks(Fork0,Fork1,Fork2,Fork3,Fork4)

cluster Fork0(Lying,HeldByPhil0,HeldByPhil4)
state Lying      {L0_PickFork0->HeldByPhil0;\
                L4_PickFork0->HeldByPhil4;}
state HeldByPhil0 {L0_PutFork0 ->Lying;}
state HeldByPhil4 {L4_PutFork0 ->Lying;}

cluster Fork1(Lying,HeldByPhil1,HeldByPhil0)
state Lying      {L1_PickFork1->HeldByPhil1;\
                L0_PickFork1->HeldByPhil0;}
state HeldByPhil1 {L1_PutFork1 ->Lying;}
state HeldByPhil0 {L0_PutFork1 ->Lying;}

cluster Fork2(Lying,HeldByPhil2,HeldByPhil1)
state Lying      {L2_PickFork2->HeldByPhil2;\
                L1_PickFork2->HeldByPhil1;}
state HeldByPhil2 {L2_PutFork2 ->Lying;}
state HeldByPhil1 {L1_PutFork2 ->Lying;}

cluster Fork3(Lying,HeldByPhil3,HeldByPhil2)
state Lying      {L3_PickFork3->HeldByPhil3;\
                L2_PickFork3->HeldByPhil2;}
state HeldByPhil3 {L3_PutFork3 ->Lying;}
state HeldByPhil2 {L2_PutFork3 ->Lying;}

cluster Fork4(Lying,HeldByPhil4,HeldByPhil3)
state Lying      {L4_PickFork4->HeldByPhil4;\
                L3_PickFork4->HeldByPhil3;}
state HeldByPhil4 {L4_PutFork4 ->Lying;}
state HeldByPhil3 {L3_PutFork4 ->Lying;}

//-----[end of module]-----

```

### 9.4.5.2 Source listing of the dining philosophers with semaphores [model t4335]

```

//-----
// Module:   phil_semaph.scs.txt
// Author:   Graham Thomason, Philips Digital Systems Laboratories, Redhill
// Date:     19 July, 2003
// Purpose:  StateCruncher model: The Dining philosophers with semaphores
//
// Copyright (C) 2003 Philips Electronics N.V.
//-----1-----2-----3-----4-----5-----6-----7-----8-----

statechart sc(College)

PCO external;          // For philosopher actions
PCO composition;      // For communication from semaphote to philosopher
PCO internal;         // Internal events

event Pang0, Pang1, Pang2, Pang3, Pang4 @external;
event Full0, Full1, Full2, Full3, Full4 @external;

event Req01, Req12, Req23, Req34, Req40 @composition;
event Rel01, Rel12, Rel23, Rel34, Rel40 @composition;
event Acq01, Acq12, Acq23, Acq34, Acq40 @composition;

event Try01, Try12, Try23, Try34, Try40 @internal;
event Pick0, Pick1, Pick2, Pick3, Pick4 @internal;
event Put0, Put1, Put2, Put3, Put4 @internal;

```

```

set College(Philosophers,Forks)

set Philosophers(Phil0,Phil1,Phil2,Phil3,Phil4)

cluster Phil0(Thinking,Waiting,Eating)
  state Thinking {Pang0->Waiting {fire Req01;};}
  state Waiting {Acq01->Eating;}
  state Eating {upon enter {trace("P0Eat");} \
                upon exit {trace("P0Stp");} \
                Full0->Thinking {fire Rel01;};}

cluster Phil1(Thinking,Waiting,Eating)
  state Thinking {Pang1->Waiting {fire Req12;};}
  state Waiting {Acq12->Eating;}
  state Eating {upon enter {trace("P1Eat");} \
                upon exit {trace("P1Stp");} \
                Full1->Thinking {fire Rel12;};}

cluster Phil2(Thinking,Waiting,Eating)
  state Thinking {Pang2->Waiting {fire Req23;};}
  state Waiting {Acq23->Eating;}
  state Eating {upon enter {trace("P2Eat");} \
                upon exit {trace("P2Stp");} \
                Full2->Thinking {fire Rel23;};}

cluster Phil3(Thinking,Waiting,Eating)
  state Thinking {Pang3->Waiting {fire Req34;};}
  state Waiting {Acq34->Eating;}
  state Eating {upon enter {trace("P3Eat");} \
                upon exit {trace("P3Stp");} \
                Full3->Thinking {fire Rel34;};}

cluster Phil4(Thinking,Waiting,Eating)
  state Thinking {Pang4->Waiting {fire Req40;};}
  state Waiting {Acq40->Eating;}
  state Eating {upon enter {trace("P4Eat");} \
                upon exit {trace("P4Stp");} \
                Full4->Thinking {fire Rel40;};}

set Forks(Fork0,Fork1,Fork2,Fork3,Fork4, Pair01,Pair12,Pair23,Pair34,Pair40)

cluster Fork0(Lying,Held)
  state Lying {Pick0->Held;}
  state Held {Put0->Lying;}

cluster Fork1(Lying,Held)
  state Lying {Pick1->Held;}
  state Held {Put1->Lying;}

cluster Fork2(Lying,Held)
  state Lying {Pick2->Held;}
  state Held {Put2->Lying;}

cluster Fork3(Lying,Held)
  state Lying {Pick3->Held;}
  state Held {Put3->Lying;}

cluster Fork4(Lying,Held)
  state Lying {Pick4->Held;}
  state Held {Put4->Lying;}

/*--[Fork Pair Control]--*/

cluster Pair01(Reset,Requested)
  state Reset {Req01[in($Fork0.Lying) && in($Fork1.Lying)] \
              {fire Pick0; fire Pick1; fire Acq01;}; \
              Rel01 \
              {fire Put0; fire Put1; fire Try40; fire Try12;}; \
              Req01[ !in($Fork0.Lying) || !in($Fork1.Lying)] \
              ->Requested; \
              }

  state Requested {Try01[in($Fork0.Lying) && in($Fork1.Lying)] \
                  }

```

```

        -> Reset
        {fire Pick0; fire Pick1; fire Acq01;};
    }

cluster Pair12(Reset,Requested)
    state Reset
        {Req12[in($Fork1.Lying) && in($Fork2.Lying)]
         {fire Pick1; fire Pick2; fire Acq12;};
         Rel12
         {fire Put1; fire Put2; fire Try01; fire Try23;};
         Req12[ !in($Fork1.Lying) || !in($Fork2.Lying)]
         ->Requested;
        }

    state Requested {Try12[in($Fork1.Lying) && in($Fork2.Lying)]
        -> Reset
        {fire Pick1; fire Pick2; fire Acq12;};
    }

cluster Pair23(Reset,Requested)
    state Reset
        {Req23[in($Fork2.Lying) && in($Fork3.Lying)]
         {fire Pick2; fire Pick3; fire Acq23;};
         Rel23
         {fire Put2; fire Put3; fire Try12; fire Try34;};
         Req23[ !in($Fork2.Lying) || !in($Fork3.Lying)]
         ->Requested;
        }

    state Requested {Try23[in($Fork2.Lying) && in($Fork3.Lying)]
        -> Reset
        {fire Pick2; fire Pick3; fire Acq23;};
    }

cluster Pair34(Reset,Requested)
    state Reset
        {Req34[in($Fork3.Lying) && in($Fork4.Lying)]
         {fire Pick3; fire Pick4; fire Acq34;};
         Rel34
         {fire Put3; fire Put4; fire Try23; fire Try40;};
         Req34[ !in($Fork3.Lying) || !in($Fork4.Lying)]
         ->Requested;
        }

    state Requested {Try34[in($Fork3.Lying) && in($Fork4.Lying)]
        -> Reset
        {fire Pick3; fire Pick4; fire Acq34;};
    }

cluster Pair40(Reset,Requested)
    state Reset
        {Req40[in($Fork4.Lying) && in($Fork0.Lying)]
         {fire Pick4; fire Pick0; fire Acq40;};
         Rel40
         {fire Put4; fire Put0; fire Try34; fire Try01;};
         Req40[ !in($Fork4.Lying) || !in($Fork0.Lying)]
         ->Requested;
        }

    state Requested {Try40[in($Fork4.Lying) && in($Fork0.Lying)]
        -> Reset
        {fire Pick4; fire Pick0; fire Acq40;};
    }

//-----[end of module]-----

```

# 10. Experience with STATECRUNCHER and conclusions

The project set out with two experimental goals: (1) to investigate whether an approach to automatic generation of state-based tests of nondeterministic systems using a nondeterministic oracle would offer an improved testing technique, and (2) to see whether PROLOG is a feasible implementation language for such a tool, both from an ease-of-coding viewpoint and from a run-time performance perspective. This section reports on how the testing approach is being pursued within Philips. We illustrate how STATECRUNCHER has been successfully transferred to an end-user within Philips Electronics, with a real example of an embedded software component being tested in a tool chain using STATECRUNCHER as the test oracle. We also review the implementation approach taken. Lastly, we draw a final conclusion.

## 10.1 Experience at Philips

Software testing as a Research activity was formally transferred from PRL (Philips Research Laboratories - Redhill) to PRI-B (Philips Research India - Bangalore) at the start of 2002. The development of STATECRUNCHER at Redhill, and support to PRI-B continued in 2002 and part of 2003, carried out in the PDSL-R organisation (Philips Digital Systems Laboratories - Redhill).

Philips Research India - Bangalore (PRI-B) has successfully worked with STATECRUNCHER, having integrated it into the TorX tool chain, testing [Koala] components for television systems.

The following figure is by Nitin Koppalkar at PRI-B, who did the integration.

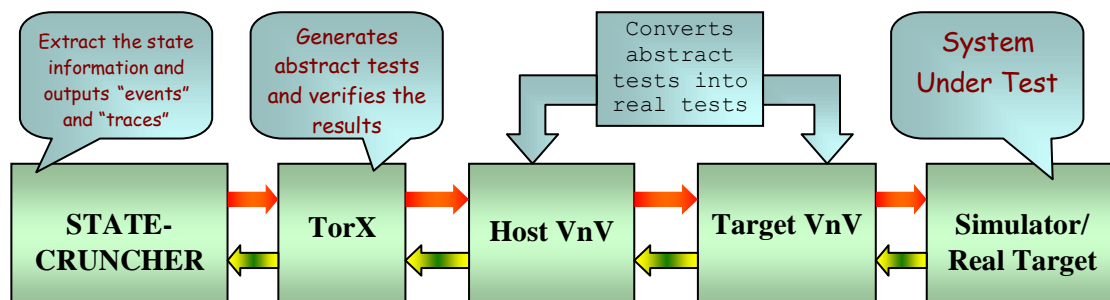


Figure 170. STATECRUNCHER integrated in the TorX tool chain (Nitin Koppalkar)



Two components that have been modelled and tested are *TV Program Installation* and the *Last Status Manager*.

***TV Program Installation (modelled by Tim Trew)***

A STATECRUNCHER model has been produced for a component that installs a program in a TV. The sequence of operations is to:

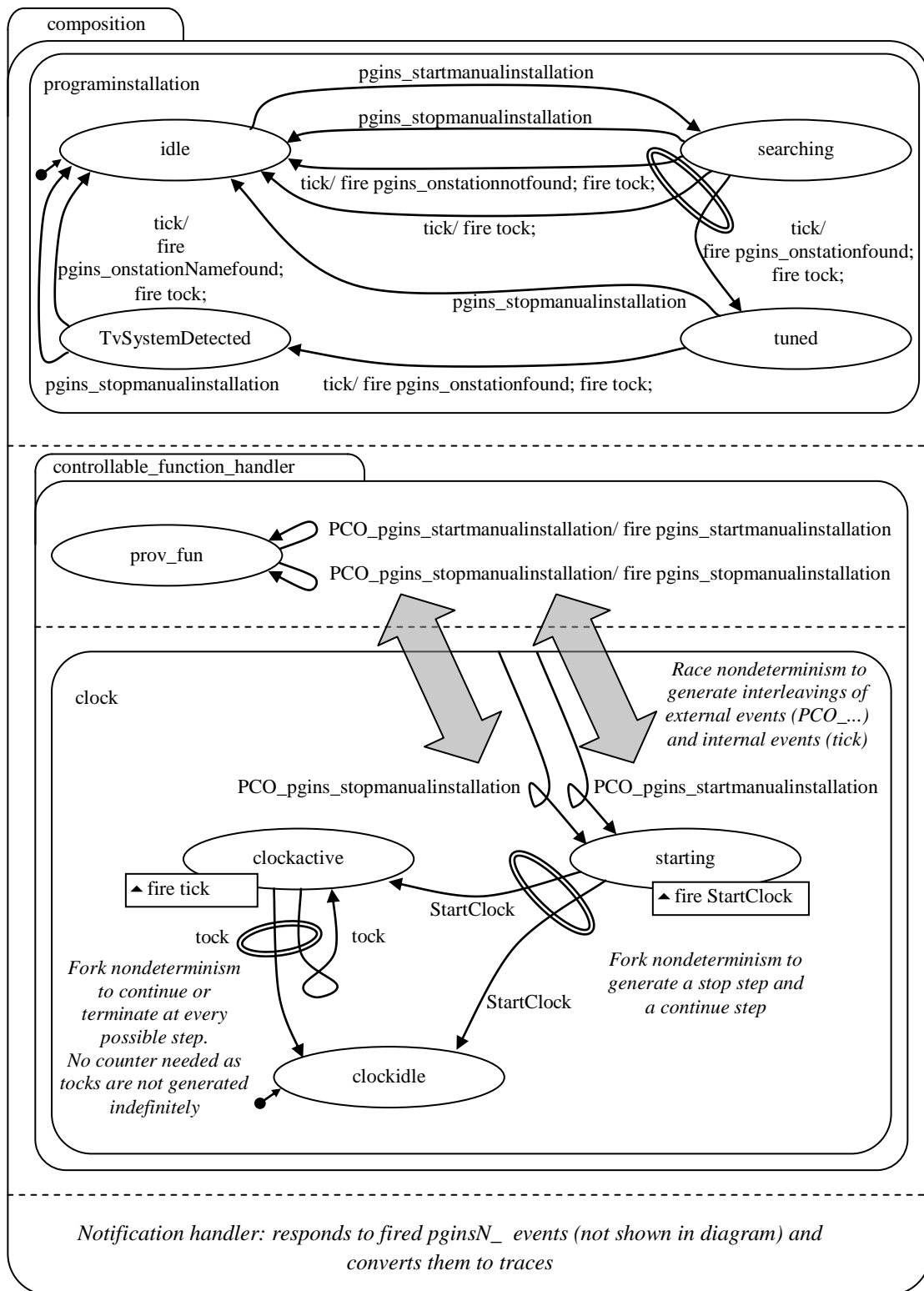
1. Find the carrier
2. Analyse the modulation to find out the TV system (PAL / NTSC / SECAM)
3. Analyse the VBI (vertical blanking interval) data to deduce the station name.

The issues are:

- To use a generic model of the program installation component in any testing configuration or composed-system configuration.
- To obtain all nondeterministic outcomes in the STATECRUNCHER model due to a failure to proceed at any stage.
- To obtain all nondeterministic outcomes in the STATECRUNCHER model due to interleavings of external and internal events.

The Philips report [Trew 03] covers this model, and discusses challenging generic issues in component modelling, such as how to generate interleavings of external and internal events in STATECRUNCHER.

The following model is a simplification of what has been produced. A more extensive model contains details of the tuner.



**Figure 171. Program Installation, simplified, (Tim Trew) [model t4410]**

Points to note:

- The clock generates `tick` events, to which the `programinstallation` states respond, forking on alternatives where they exist.
- The `programinstallation` area fires a `tock` after any response to a `tick`, in order to keep the clock going.
- This clock does not need to limit the number of ticks fired, as the `programinstallation` is not capable of infinite cycling.
- Race nondeterminism is used to generate interleavings of external events (`PCO_...`) and internal events (`tick`). This covers situations where an external event is given, but is preempted by an internal event.
- Fork nondeterminism is used to continue or terminate the clock at every step.

Performance is acceptable: on a 300MHz machine, it takes about 2 seconds to process `PCO_pgins_startmanualinstallation` giving 9 worlds.

Output after event `PCO_pgins_startmanualinstallation` (9 worlds generated).

Wld	program instaln.	Clock	Trace (read in reverse order)	Remarks on program installation
17	idle	idle	[tock, tick/pgins_onstationnotfound, tick in searching, firing tick, PCO_pgins_startmanualinstallation clock, pginsN_onmanualinstallationstarted, pgins_startmanualinstallation in idle, PCO_pgins_startmanualinstallation executed]	Searching did not find a station.
20	idle	active	[firing tick, tock, tick/pgins_onstationnotfound, tick in searching, firing tick, PCO_pgins_startmanualinstallation clock, pginsN_onmanualinstallationstarted, pgins_startmanualinstallation in idle, PCO_pgins_startmanualinstallation executed]	Searching did not find a station. There was an extra tick, with no response.
24	tuned	idle	[tock, tick/pginsN_onstationfound, tick in searching, firing tick, PCO_pgins_startmanualinstallation clock, pginsN_onmanualinstallationstarted, pgins_startmanualinstallation in idle, PCO_pgins_startmanualinstallation executed]	Searched and found a station. Did not proceed to detect the TV system.
32	Tv System Detected	idle	[tock, tick/pginsN_onTvSystemDetected, tick in tuned, firing tick, tock, tick/pginsN_onstationfound, tick in searching, firing tick, PCO_pgins_startmanualinstallation clock, pginsN_onmanualinstallationstarted, pgins_startmanualinstallation in idle, PCO_pgins_startmanualinstallation executed]	Searched, found a station and detected the TV system. Did not proceed to find station name.

40	idle	idle	[tock, tick/pginsN_onStationNameFound, tick in TvSystemDetected, firing tick, tock, tick/pginsN_onTvSystemDetected, tick in tuned, firing tick, tock, tick/pginsN_onstationfound, tick in searching, firing tick, PCO_pgins_startmanualinstallation clock, pginsN_onmanualinstallationstarted, pgins_startmanualinstallation in idle, PCO_pgins_startmanualinstallation executed]	A complete cycle through searching, finding a station, detecting the TV system and finding the station name.
43	idle	active	[firing tick, tock, tick/pginsN_onStationNameFound, tick in TvSystemDetected, firing tick, tock, tick/pginsN_onTvSystemDetected, tick in tuned, firing tick, tock, tick/pginsN_onstationfound, tick in searching, firing tick, PCO_pgins_startmanualinstallation clock, pginsN_onmanualinstallationstarted, pgins_startmanualinstallation in idle, PCO_pgins_startmanualinstallation executed]	A complete cycle with an extra tick, to which there was no response.
44	searching	idle	[PCO_pgins_startmanualinstallation clock, pginsN_onmanualinstallationstarted, pgins_startmanualinstallation in idle, PCO_pgins_startmanualinstallation executed]	Searching, with no further progress.
55	searching	idle	[pginsN_onmanualinstallationstarted, pgins_startmanualinstallation in idle, PCO_pgins_startmanualinstallation executed, PCO_pgins_startmanualinstallation clock]	Searching, with no further progress. Differs from world 44 because of the race (clock wins).
61	searching	active	[pginsN_onmanualinstallationstarted, pgins_startmanualinstallation in idle, PCO_pgins_startmanualinstallation executed, firing tick, PCO_pgins_startmanualinstallation clock]	Searching, with clock winning a race and doing nothing.

**Table 17. Program Installation results**

After the traces have been cleared, there are 6 residual worlds. Then event PCO\_pgins\_stopmanualinstallation can be given, generating 24 worlds (in about 15 seconds on a 300 MHz machine). Space does not permit us to tabulate the results, but we remark that on stopping the installation, a race is run on two transitions on PCO\_stopmanualinstallation, generating interleavings of events pgins\_stopmanualinstallation and tick. The tick first situation could represent a user stopping the installation, but just before the command is seen, the installation completes.

### **Model listing**

```
// Author: Tim Trew
// Test of transition algorithm for clock ticking - can we interleave
// all "wait" events with external events?

// User enters
// SC: pe [PCO_pgins_startmanualinstallation, [composition, sc]]
```

```

// SC: ct
// SC: pe [PCO_pgins_stopmanualinstallation, [composition, sc]]

statechart sc(composition)

set composition (programinstallation, \
                 controllable_function_handler, \
                 notification_handler)

cluster programinstallation (idle, searching, tuned, TvSystemDetected)

/* Program Installation Provided functions */
event composition%%pgins_startmanualinstallation;
event composition%%pgins_stopmanualinstallation;

/* Program Installation notifications */
event composition%%pginsN_onmanualinstallationstarted;
event composition%%pginsN_onmanualinstallationcompleted;
event composition%%pginsN_onmanualinstallationstopped;
event composition%%pginsN_onsearchinprogress;
event composition%%pginsN_onstationfound;
event composition%%pginsN_onstationnotfound;
event composition%%pginsN_onTvSystemDetected;
event composition%%pginsN_onStationNameFound;

state idle { \
  pgins_startmanualinstallation -> searching \
  {trace("pgins_startmanualinstallation in idle"); \
   fire pginsN_onmanualinstallationstarted ; } ; \
  pgins_stopmanualinstallation \
  {trace("pgins_stopmanualinstallation in idle - ignored"); } ; }

state searching { \
  pgins_startmanualinstallation \
  {trace("pgins_startmanualinstallation in searching - ignored"); } ; \
  pgins_stopmanualinstallation -> idle \
  {trace("pgins_stopmanualinstallation in searching"); \
   fire pginsN_onmanualinstallationstopped ; } ; \
  tick -> tuned {trace("tick/pginsN_onstationfound"); \
                fire pginsN_onstationfound; fire tock; } ; \
  tick -> idle {trace("tick/pgins_onstationnotfound"); \
               fire pginsN_onstationnotfound; fire tock; } ; }

state tuned { \
  pgins_startmanualinstallation \
  {trace("pgins_startmanualinstallation in tuned - ignored"); } ; \
  pgins_stopmanualinstallation -> idle \
  {trace("pgins_stopmanualinstallation in tuned"); \
   fire pginsN_onmanualinstallationstopped ; } ; \
  tick -> TvSystemDetected {trace("tick/pginsN_onTvSystemDetected"); \
                            fire pginsN_onTvSystemDetected; \
                            fire tock; } ; }

state TvSystemDetected { \
  pgins_startmanualinstallation \
  {trace("pgins_startmanualinstallation in TvSystemDetected - ignored"); } ; \
  pgins_stopmanualinstallation -> idle \
  {trace("pgins_stopmanualinstallation in TvSystemDetected"); \
   fire pginsN_onmanualinstallationstopped; } ; \
  tick -> idle {trace("tick/pginsN_onStationNameFound"); \
               fire pginsN_onStationNameFound; fire tock; } ; }

// provides functions

```

```

set controllable_function_handler (prov_fun, clock)
  event composition%%PCO_pgins_startmanualinstallation;
  event composition%%PCO_pgins_stopmanualinstallation;

  event composition%%tock;
  event composition%%tick, StartClock;

state prov_fun {
  PCO_pgins_startmanualinstallation
    {trace("PCO_pgins_startmanualinstallation executed");
     fire pgins_startmanualinstallation ; };
  PCO_pgins_stopmanualinstallation
    {trace("PCO_pgins_stopmanualinstallation executed");
     fire pgins_stopmanualinstallation ; };

cluster clock (clockidle, starting, clockactive) {
  PCO_pgins_startmanualinstallation -> clock -> clock.starting
    {trace("PCO_pgins_startmanualinstallation clock"); };
  PCO_pgins_stopmanualinstallation -> clock -> clock.starting
    {trace("PCO_pgins_stopmanualinstallation clock"); };

state clockidle;

state starting {
  upon enter { fire StartClock; };
  StartClock -> clockidle;
  StartClock -> clockactive;

state clockactive {
  upon enter {
    trace("firing tick");
    if (in (::composition.programinstallation.searching))
      {trace("tick in searching"); }
    if (in (::composition.programinstallation.tuned))
      {trace("tick in tuned"); }
    if (in (::composition.programinstallation.TvSystemDetected))
      {trace("tick in TvSystemDetected"); }
    fire tick; }
  /* Fork non-determinism to terminate at every possible step. */
  tock -> clock -> clockactive {trace("tock");};
  tock -> clockidle {trace("tock");};

cluster notification_handler (notif_handler)
/* Turned fired notifications in to traces */
event composition%%pginsN_onchannelfound;

state notif_handler {
  pginsN_onchannelfound -> notif_handler
    {trace ("pginsN_onchannelfound"); };
  pginsN_onmanualinstallationstarted -> notif_handler
    {trace ("pginsN_onmanualinstallationstarted"); };
  pginsN_onmanualinstallationcompleted -> notif_handler
    {trace ("pginsN_onmanualinstallationcompleted"); };
  pginsN_onmanualinstallationstopped -> notif_handler
    {trace ("pginsN_onmanualinstallationstopped"); };
  pginsN_onsearchinprogress -> notif_handler
    {trace ("pginsN_onsearchinprogress") ; };

```

The more extensive model (including the tuner) has been integrated into the TorX tool chain by Nitin Koppalkar at PRI-B. The following diagram, by Nitin Koppalkar, shows the tool chain in action:

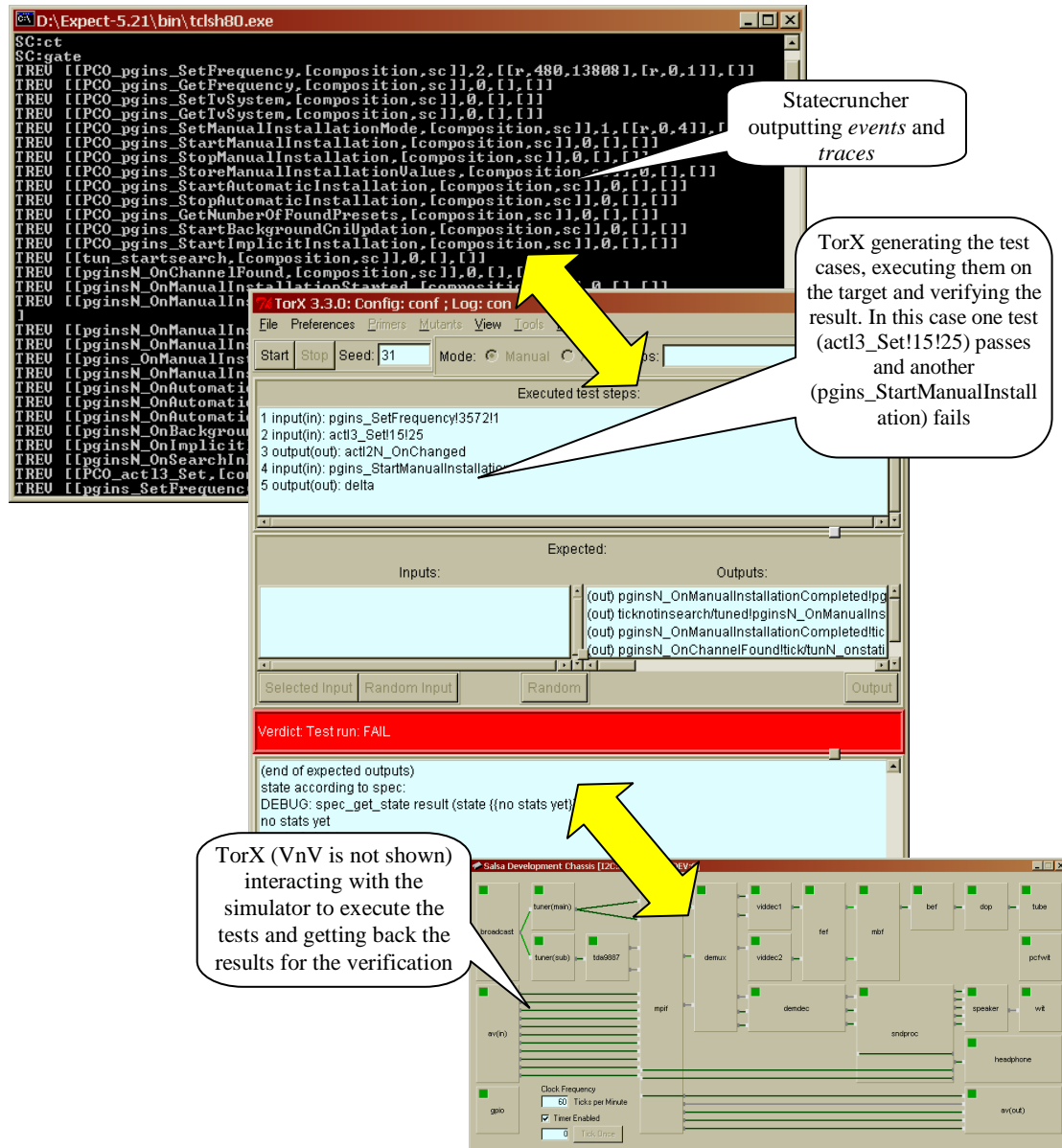


Figure 172. STATECRUNCHER and TorX in action (Nitin Koppalkar)

**Last Status Manager:** Currently (November 2003), PRI-B is working on testing this module, which manages status information, writing it at intervals to non-volatile memory (NVM). At any time, the cache can contain messages that have been written to NVM and messages that still have to be written to NVM, under the constraint that if a message has been written to NVM, all older messages must have also been written to NVM. Later messages may or may not be in NVM, hence nondeterminism. It was considered useful to have an *array* facility to handle the messages in chronological order. It was to meet the needs of this system that arrays were implemented in STATECRUNCHER (in Release 1.04).

### ***Outcomes of the trials of STATECRUNCHER***

We have shown that STATECRUNCHER has been successfully deployed in a live project. The experience of this trial clearly demonstrated STATECRUNCHER's ability to handle all the forms of nondeterminism that were inherently present in the system under test. The successful outcome of these trials has led to a number of reports and continued work using STATECRUNCHER. The following reports have been written or are nearing completion:

On integrating STATECRUNCHER into the TorX tool chain [Koppalkar 02, 03]:

- Nitin Koppalkar and Animesh Bhowmick  
Integration of Generic Explorer with the TorX Tool Chain  
Philips Nat. Lab. Technical Note 2002/387, October, 2002
- Nitin Koppalkar  
Interfacing STATECRUNCHER with TorX for demonstrating the state-based testing technique taking MG-R components for a case study  
Philips Nat. Lab. Technical Note (*under preparation, December 2003*).

On modelling software components in STATECRUNCHER [Trew 03]:

- Tim Trew  
State-based modelling of software components for integration testing  
A practical guide to the creation of STATECRUNCHER models  
Philips Nat. Lab. Technical Note (*under preparation, December 2003*).

We indicate some future directions at the end of this section.

## **10.2 PROLOG as the implementation language**

There is of course a subjective element in stating whether PROLOG is a feasible implementation language for any given purpose. Different people show affinity to different programming languages, and few can claim competence in a really wide range of them. The present author's view is that to build the same STATECRUNCHER system in C would require a significant multiple of the effort taken, although such an undertaking *by a team*, given the present implementation as a precise specification, would not be pointless, as it would lead to



improved performance and greater maintainability in an organisation, because one could then tap into a wider pool of programmers than is the case with a PROLOG implementation. To use an object oriented language could help in many ways, but the hard parts of the transition algorithm are not clearly amenable to an object-oriented approach.

### *Strengths of PROLOG as a programming language*

In the author's estimation, the power of PROLOG (for readers not entirely unfamiliar with PROLOG), lies in the following features:

- **Compact notation.** Although this is arguably a very superficial aspect, it does make for readable programs. They can be overseen with more ease because there is less syntactic overhead (compare the abundant use of brackets in LISP). Examples:
  - Variables have no declaration and their scope is just the one clause they are used in. Symbols beginning with capitals or underscore are variables, and are distinct from those beginning with lower case letters which are *atoms*, i.e. constants. The *and* operator is a comma, and the *or* operator is a semicolon. The result approaches the compactness of the notation for predicates and specifications in discrete mathematics.
  - The notation  $[H|T]$  denotes the head and tail of a list. The head is one element of a list and the tail is conventionally zero or more elements of the list. The term  $[H|T]$  will construct a list from a head and a tail, or split a list into head and tail, or it can be used to check whether an item is a list with at least one element and some tail, (which may be the empty list).
- **Typelessness.** The fact that PROLOG is untyped makes many routines very general, where in C many versions of a function might be needed, one for each type of argument, though this is less of a problem in C++, where a *template* construction can be used.
- **The interpretative nature.** Programs, whether large or very small (e.g. just one *clause*) can be experimented with at the command prompt. PROLOG programs have no header files and compile so fast there is no need for a developer to *build* them, as in non-interpretative languages. The whole of STATECRUNCHER compiles in little more than a second on a modern computer.
- **Unification.** This allows a partially grounded structure to be matched against another one, e.g.  $[a, [B, C]]$  against  $[D, [e|T]]$ . A variable matched against a grounded item is *instantiated* to that item and becomes grounded. The above match succeeds with
  - B = e
  - C = `_G163`
  - D = a
  - T = `[_G163]`

This sort of match is useful e.g. in extracting parts of compiled statements, such as the condition of a transition, where the parse contains structures partly labelled by fixed atoms, with the remaining parse body representing the real parse content to be extracted.

The result of the unification may still contain non-ground terms, as variable *T* is in the above example, though it is constrained to be a list of one element.

- **Backtracking.** This is a search mechanism that will look for a structural match, and satisfaction of further constraints. An example of use might be to find a parsed statement satisfying a certain constraint, such as finding a declaration of a variable of a certain name, or finding a potential transition, then requiring that it satisfy various conditions. An extension to backtracking is to *find all* items satisfying some constraint. Backtracking is also a good mechanism for *generating* many solutions to some requirement, such as permutations.
- **Reversibility.** PROLOG clauses can be written to work in two directions - indeed they will do automatically in many cases, perhaps without the program author realising it. The same simple PROLOG clauses defining how to *append* two lists *L1* and *L2* making *L3*, can also break up a given list *L3* into sublists *L1* and *L2* which when appended, make the given list. It will do this in all possible ways, e.g.  $[a, b, c]$  can be split this way into:
  - $L1 = [], \quad L2 = [a, b, c]$
  - $L1 = [a], \quad L2 = [b, c]$
  - $L1 = [a, b], \quad L2 = [c]$
  - $L1 = [a, b, c], \quad L2 = []$In fact the *append* clause can work with three instantiated parameters to verify that *L1* and *L2* append into *L3*, and even with only *L1* instantiated or only *L2* instantiated or even more unusually with all three parameters uninstantiated.
- **The Definite Clause Grammar (DCG).** This is very convenient way of expressing Backus-Naur grammar rules and recording a parse for them. It is based on processing list structures by specifying what part of a list is used up in the parse, and what part is returned as unused, available for the next term in a grammar rule. It is described very lucidly in [Clocksin]. The implementation of STATECRUNCHER's expression parser shows that use of DCGs is feasible on a large scale (about 20 operator precedences), provided care is taken to maintain efficiency.

### ***PROLOG's run-time performance***

There are two parts to PROLOG's execution performance: compilation and the run-time engine. Although PROLOG's Definite Clause Grammar is well-known for its parsing capability, it is probably for performance reasons that it is not more widely used for full domain-specific-language systems. However, the compilation speed of a STATECRUNCHER model is very acceptable, good even, on a modern (3GHz) machine, where typical illustrative models (as in [StCrManual]), compile in a second or so. Compilation, especially of expressions is certainly felt to be an area where, with more analysis and profiling, the performance could be improved further.

The stress tests in [StCrTest] show that performance is generally good, but with nondeterminism, there are, and always will be, cases of combinatorial explosion. In *deterministic* situations, STATECRUNCHER is fast, by human standards, in all models investigated, including automatically generated large ones.

There are differences between different PROLOG implementations, but the author has been very satisfied with the two chosen for the investigation: [SWI Prolog], which is in the public domain, and [WinProlog], a commercial system. There are not great differences in execution speed, although it can be remarked that the difference between running the WinProlog system as an MS-DOS executable and running in the development environment gives a factor of 2 or 3 difference in performance.

## 10.3 Future directions

Future directions can be seen in *tooling* and in *testing*.

### 10.3.1 The tooling side

#### *Possible enhancements to STATECRUNCHER*

Philips Research has expressed interest in extending STATECRUNCHER with *machine implantation*, whereby state machine templates can be dynamically implanted into a statechart, as described in [StCrFunMod]. This makes whole statecharts recursive, and would solve the problem of how to model (indirectly) synchronous and asynchronous recursive function calls.

A less drastic enhancement to STATECRUNCHER is to implement UML pseudo-states, though these can be simulated with the existing features. Ideally, STATECRUNCHER would keep pace with all developments in UML, as this is becoming the industry standard.

Other possible enhancements are: to support forward chaining of data and lambda transitions (i.e. transitions that take place when some boolean expression becomes true) and to combine cause effect graphing with statecharts.

#### *STATECRUNCHER's performance*

STATECRUNCHER has been subjected to some stress tests, described in detail in [StCrTest]. Some models of regular structure but arbitrary size can be generated by PROLOG programs. Examples are: broad clusters, deep clusters, broad sets, deep sets, intensive nondeterminism, and long chains of fired events. Response times for processing an event as given below are for STATECRUNCHER running under [SWI-Prolog] on a 300 MHz machine. More modern machines can give a factor 10 improvement.

STATECRUNCHER almost always performs well with deterministic models (i.e. no forks in the model, and with race and set transit nondeterminism disabled). Examples:

- Test model  $\tau_{7110}$ , with 25 clusters of 25 leaf-states (625 leaf-states in total), executes a leafstate-to-leafstate transition in 1 second and a cluster-to-cluster one in 2.5 seconds.
- Test model  $\tau_{7120}$ , containing a set of 5 sets each with 5 member clusters of 2 leaf-states, executes an event causing transitioning in all 25 clusters in 1.8 seconds.
- Test model  $\tau_{7180}$  executes a chain of 25 fired events across 25 members of a set in 1.7 seconds.

In nondeterministic situations, models with a few tens of worlds generally perform adequately. The *Program Installation* example (Figure 171) performs well. With larger numbers of worlds (say 100), performance can become a bottleneck, though models have been run leading to world numbers in the thousands after very few events. Set nondeterminism with nested sets appears to degrade performance considerably.

### ***Approaches to increasing STATECRUNCHER's performance***

What options are there for performance improvements? We consider some:

- ***Re-write the program in 'C'***. 'C' is a compile-to-executable (non-interpretative) language which facilitates very precise control over all algorithms, including memory allocation. A disadvantage of this approach is that it would probably be a very time-consuming exercise, though the existence of the PROLOG implementation would provide an unambiguous specification, and would give much guidance on implementation strategy.
- ***Write critical inner loops in 'C'***. One would profile the execution of the PROLOG version to find the critical inner loops. Profiling utilities and an interface mechanism to external code exist for most PROLOG systems. This approach could be very effective, but it is PROLOG-implementation specific. It could be that what is critical to one PROLOG engine is not critical to another. Also, the external interface mechanisms are liable to be specific to the PROLOG system used.
- ***Write one's own subset of PROLOG in 'C'***. By implementing some PROLOG operations as 'C' routines, especially *list* operations, one might be able to produce a system that generally makes use of the existing PROLOG structure, whilst benefiting from the efficiency and controllability of 'C'.
- ***Investigate other PROLOG engines***. There are many suppliers of PROLOG systems. STATECRUNCHER already runs under two PROLOG systems, [SWI-Prolog] and [WinProlog]. This means that a framework for further porting is already in place, with many system-dependent predicates already implemented in a *compatibility library*. The test suite, (mentioned in section 9.3) would help drive the porting process: once all tests run, the serious porting work is likely to be complete.

- ***Tweak PROLOG garbage collection.*** A weakness of PROLOG as a programming language could be that the user does not have adequate control over memory management. The garbage collection algorithm used may not be known. However, most PROLOG systems offer the possibility to make extra garbage collection calls. A few experiments have been done with this, but so far no significant improvements have been observed.
- ***Tailor the coding style to a particular PROLOG engine.*** Some PROLOG suppliers offer guidance on how to write efficient code, though what is good for one system may be bad for another. A case in point is whether to be liberal or sparing with the use of the PROLOG *cut*. One might think that putting in a redundant cut at the end of a deterministic predicate *helps* a PROLOG engine, enabling it to recover many stack frames, but it may *impede* it. This may be because it interferes with *tail recursion optimisation*, where a recursive call at the end of a predicate is executed at the caller's level, rather than by creating a new calling level. A few experiments with removing cuts in the *process set of task sequences in worlds* routine shows that memory requirements become very different (e.g. stack space is traded for heap space), but that there is no drastic performance or capacity change. Another aspect to tailoring code is to make use of *supplied library functions* rather than one's own generic implementations.
- ***Algorithmic experimentation.*** The transition algorithm was described with various algorithmic alternatives, such as the algorithm A / algorithm B options in the main *process set of task sequences in worlds* routine. It could be that a better choice can be found.
- ***Write a front-end cache to STATECRUNCHER*** that pre-explores the state space when the IUT is not executing under real-time constraints, so that when the IUT *is* executing under real-time constraints, a rapid-response test oracle can be given.
- ***Make use of parallel processing*** (e.g. a processor per world). This would be easier at a *macroscopic* level (allocating each extant world visible at *user-event* processing time to a processor) than at a *microscopic* level (allocating each extant world visible at *internal-event* processing time to a processor). As the number of worlds may be larger than the number of processors, some form of dynamic allocation of tasks would be required.

The above list gives many options, but it must be remembered that performance optimisation is in competition with pressure for new features (e.g. as mentioned in this subsection). Moreover, STATECRUNCHER is in competition for resources with the other elements of the tool chain. Should more effort be spent on test generation? Priorities are often determined by the customer.

### ***Perspectives for on-the-fly testing and test generation***

There is scope for research into advanced *primers* (test generators), performing intelligent transition tours and disambiguating IUT states under nondeterminism. STATECRUNCHER at

least enables flattening of nondeterministic UML statecharts, and may be useful for other transformations, e.g. finding an observable NFSM (Nondeterministic Finite State Machine) that is equivalent to an unobservable one (*observable* means that outputs on transitions reveal the new state). STATECRUNCHER could have a role to play as an experimental vehicle for advanced *on-the-fly* testing (Lee's *adaptive* testing) algorithms, which can be more efficient than off-line generated *batch* tests (Lee's *preset* testing). For example, the *homing problem* (see [Lee], p.1095) consists of determining the final state of a machine by giving it a sequence of events and observing the outputs. With on-the-fly testing the homing sequence can be shorter than in the batch case. However, *homing* (which drives the machine into a known state following on from a test) is weaker than *distinguishing* or *verifying* or *identifying* the state after the test, but on-the-fly testing helps here too [Lee, p.1097, p.1105], [Hierons 98]. STATECRUNCHER's command language offers efficient hooks needed by the test generation or other programs.

### 10.3.2 The testing side

#### *Practical problems being tackled*

STATECRUNCHER has been the test oracle tool on which state-based testing at Philips has been focussed for well over a year. The strength of STATECRUNCHER is seen as being in its UML-friendly and intuitive syntax, and its ability to handle nondeterminism, which was the motivation for its development. Other strengths are its support for scoping operators and its *after-landing* transition semantics, both of which facilitate component composition.

PRI-B has shown itself able to use STATECRUNCHER in an advanced testing environment. STATECRUNCHER has been integrated into an end-to-end tool chain, based on TorX, using EXPECT scripts to adapt STATECRUNCHER's interface to that required by TorX. Various components have been selected for modelling and testing.

It has been found that creating some dynamic models from a conventional specification is a particularly skilled task. Part of the difficulty is that this needs to be done in a way that enables *component model* composition to follow the mechanism of *component* composition.

The challenges have been successfully met, and as they have revealed additional needs in STATECRUNCHER, (a socket interface, pruning of worlds on invalid traces, arrays) these have been supplied. The testing activities have also exposed some new problems, in particular the issue of how to handle large numbers of notifications (asynchronous messages) without creating a STATECRUNCHER world for each potential number of notifications.

It is intended to complete this phase of trialling with STATECRUNCHER in 2004. There are plans to make a comparison with another product, Conformiq, of Finnish manufacture. The results of the comparison should be available in the course of 2004.

## **10.4 Final conclusion**

We have presented a state machine system that handles nondeterminism for the purpose of providing a test oracle in a tool chain. It has successfully been transferred to Philips Research India - Bangalore for use on live projects, where it has been deployed for testing of embedded software components with inherent nondeterminism. The successful outcome of these trials to date has led to ongoing use of STATECRUNCHER in testing research within the Philips Electronics organization. We believe that one of the main contributions of this thesis has been to take a research concept from inception through to deployment in an industrial setting.

# 11. Glossary and abbreviations etc.

## 11.1 Greek letters

For compactness, and as in [CHSM], we will often use Greek letters for event names; in the STATECRUNCHER source, these would be spelled out in Roman letters. The English names of the letters are as follows:

$\alpha$ alpha	$\beta$ beta	$\gamma$ gamma	$\delta$ delta
$\epsilon$ epsilon	$\zeta$ zeta	$\eta$ eta	$\theta$ theta
$\iota$ iota	$\kappa$ kappa	$\lambda$ lambda	$\mu$ mu
$\nu$ nu	$\xi$ xi	$\omicron$ omicron	$\pi$ pi
$\rho$ rho	$\sigma$ sigma	$\tau$ tau	$\upsilon$ upsilon
$\varphi$ phi	$\chi$ chi	$\psi$ psi	$\omega$ omega

Table 18. Greek letters

## 11.2 Glossary and abbreviations

**Action:** A STATECRUNCHER term for processing that is associated with a transition (or the entering/exiting of a state). An action can be e.g.  
-a ‘C’-like assignment to a variable  
-the firing of an event  
-the generation of output (a trace).

**Black-box testing:** Testing where system outputs can be observed, but not system internals. In the case of state-based testing, the state (more precisely, *configuration*) of the system will not be directly observable, and must be deduced from *traces* (outputs generated when events are processed).

**Broadcast-event:** An event that is generated within a statechart which can be responded to by the model (transitions can be triggered by it). The STATECRUNCHER keyword to generate a broadcast event is **fire event**.



- Broadcast-event nondeterminism:** Also known as fired-event nondeterminism, this is the form of nondeterminism that arises when an action associated with a transition fires an event, which in turn gives rise (directly or indirectly) to one of the other forms of nondeterminism (e.g. fork, race, set-transit).
- CCS:** The Calculus of Communicating Systems. A process calculus defined by Robin Milner.
- CHSM:** Concurrent Hierarchical finite State Machine. A language implemented by Paul J Lucas [CHSM].
- Cluster:** A hierarchical state and component of a statechart with the understanding that if the cluster is occupied, exactly one of its members must be occupied. It is the XOR-state of Harel.
- Configuration:** The dynamic state of a statechart in a broad sense, comprising: occupancy (occupied/vacant) of the states in the statechart, variable values, cluster history, and trace values.
- CSP:** Communicating Sequential Processes. A process calculus defined by C.A.R. Hoare.
- DCG:** Definite Clause Grammar. This is the standard PROLOG grammar notation, which enables grammar rules to be written in Backus-Naur form.
- Event:** A signal (that has no time duration) which may be responded to in a statechart model by the triggering of transitions.
- Fire:** The act of generating an event in an action associated with a transition: “the action fires the event”. [Compare “triggering a transition”, which may take place when the fired event is processed].
- Fired-event nondeterminism:** Also known as broadcast-event nondeterminism, this is the form of nondeterminism that arises when an action associated with a transition fires an event, which in turn gives rise (directly or indirectly) to one of the other forms of nondeterminism (e.g. fork, race, set-transit).
- Fork nondeterminism:** The form of nondeterminism that arises when an event triggers mutually exclusive transitions in the statechart, and which produce a different outcome.

- FSM:** Finite state machine. We normally mean a flattened state machine of the Mealy type that produces observable outputs on transitions.
- GP4:** Generic Prolog Parsing and Prototyping Package. An underlying layer of PROLOG programs to provide parsing support (especially tokenization and expression parsing).
- GUI:** Graphical User Interface.
- Harness:** A test harness is a tool that contains or accesses a test script so as to obtain tests and their oracle, and communicates with an implementation under test to run the tests. It compares actual with expected output, and logs the results as pass or fail.
- IUT:** Implementation Under Test.
- Leafstate:** A state and a component of a statechart at the lowest hierarchical level.
- LHS:** Left Hand Side.
- Machine engine:** A program that holds a representation of a statechart and a configuration of that statechart, and which can process an event and in so doing calculate and assume the new configuration.
- Meta-event:** An event that is internally generated when a state is exited or entered, and which can be used to trigger transitions in other parts of the statechart.
- NFSM:** Nondeterministic Finite State Machine.
- Nondeterminism:** Dynamic behaviour of a system whereby there is more than one outcome of processing an event. Distinguishing aspects of an outcome are: state occupancy, cluster history, variable values, and traces. For a formal definition of a nondeterministic finite state machine, see section 7.1.
- ONFSM:** Observable Nondeterministic Finite State Machine. For ONFSMs, a unique target state on a transition can be deduced from the output generated by the transition.

- Oracle:** The pre-determined output of the system on a successful test, for comparison purposes with the actual output.
- PCO:** Point of Control and Observation. These are used for systems such as networked and client-server systems where inputs and outputs must be partitioned according to which separate testing point can provide and observe them.
- Primer:** The TorX terminology for the part of the tool chain that decides what events (or transitions) are to be given to the explorer and indirectly to the implementation under test to be processed.
- Race nondeterminism:** The form of nondeterminism that arises when an event triggers transitions in parallel parts of the statechart, and when the order in which these events are processed will affect the outcome.
- RHS:** Right Hand Side.
- Set:** A state and a component of a statechart with the understanding that if the set is occupied, all its members must be occupied. This represents the parallelism of a model. It is the AND-state of Harel.
- Set-action nondeterminism:** The form of nondeterminism that arises when actions (such as variable assignments) in different members of a set are executed, when the order in which this happens affects the outcome.
- Set nondeterminism:** A generic term for set-transit nondeterminism, set-action nondeterminism and set meta-event nondeterminism.
- Set-meta-event nondeterminism:** The form of nondeterminism that arises when elements of a set are exited or entered, (generating enter and exit meta-events), when the order in which this happens affects the outcome.
- Set-transit nondeterminism:** The form of nondeterminism that arises when a *set* is exited or entered, when the order in which the members are exited or entered affects the outcome.
- SRT:** State Relation Table. A table relating input states to output states via events.

- State:** This word is used in two senses according to the context
- a statechart consists of a hierarchy of states, which may be sets, clusters, or leaf-states
  - a state is the occupancy (occupied/vacant) of a state in the above sense.
- Statechart:** A concurrent, hierarchical representation of a dynamic behaviour model consisting of states, events, transitions, and optionally variables and statements for processing them.
- STATECRUNCHER:** A provisional name for a program that compiles statecharts, process events, and provide state or trace information.
- SUT:** System Under Test.
- Trace:** The output generated on processing an event (or transition), corresponding to the expected observable output of the Implementation Under Test.
- Transition:** The relation between the state of a system before and after that system has processed any event that triggers that transition.
- Trigger:** The act of responding to an event by processing an associated transition: “the event triggers the transition”. [Compare “firing an event”, which may take place as an action on the transition].
- UML:** Universal Modelling Language, as set out by the Object Modelling Group. UML is the industry standard for various modelling views on a system. The dynamic modelling view uses statecharts.
- White-box testing:** Testing where system internals can be observed. In the case of state-based testing, the state (more precisely, *configuration*) of the system can be observed directly.

# 12. References

## *STATECRUNCHER documentation and papers by the present author*

*Main Thesis*      [StCrMain]      The Design and Construction of a State Machine System that Handles Nondeterminism

### *Appendices*

Appendix 1      [StCrContext]      Software Testing in Context  
Appendix 2      [StCrSemComp]      A Semantic Comparison of STATECRUNCHER and Process Algebras  
Appendix 3      [StCrOutput]      A Quick Reference of STATECRUNCHER's Output Format  
Appendix 4      [StCrDistArb]      Distributed Arbiter Modelling in CCS and STATECRUNCHER - A Comparison  
Appendix 5      [StCrNim]      The Game of Nim in Z and STATECRUNCHER  
Appendix 6      [StCrBiblRef]      Bibliography and References

### *Related reports*

Related report 1      [StCrPrimer]      STATECRUNCHER-to-Primer Protocol  
Related report 2      [StCrManual]      STATECRUNCHER User Manual  
Related report 3      [StCrGP4]      GP4 - The Generic Prolog Parsing and Prototyping Package (*underlies the STATECRUNCHER compiler*)  
Related report 4      [StCrParsing]      STATECRUNCHER Parsing  
Related report 5      [StCrTest]      STATECRUNCHER Test Models  
Related report 6      [StCrFunMod]      State-based Modelling of Functions and Pump Engines

## *References*

Note: For a separate annotated bibliography, where references have been structured into categories, see [StCrBiblRef], listed above. The references below are those specifically referred to in this thesis.

- [BCS-SIGIST]     Standard for Software Component Testing  
*British Computer Society - Special Interest Group in Software Testing*
- [Beizer]             B. Beizer  
*Software Testing Techniques*  
International Thomson Computer Press, 1990, ISBN 1850328803
- [Bérard]             B. Bérard  
*Systems and Software Verification*  
Springer-Verlag, 2001. ISBN 3-540-41523-8
- [Beveridge]         Jim Beveridge and Robert Wiener  
*Multithreading Applications in Win32. The Complete Guide to Threads*  
Addison-Wesley, 1996, ISBN 0-201-44234-5 (pb)
- [Callahan]             John R. Callahan  
<http://www.cs.wvu.edu/~callahan/interests.html>  
<http://www.ivv.nasa.gov>
- [CdR]                 Côte de Resyste (delivers the TorX tool)  
<http://fmt.cs.utwente.nl/CdR>
- [CdR-iP]             René de Vries, Jan Tretmans, Axel Belinfante, Jan Feenstra, Lex Heerink,  
Loe Feijs, Sjouke Mauw, Nicolae Goga, Arjan de Heer  
*Côte de Resyste in Progress*  
*see the [CdR] site*
- [Chow]                 Tsun S. Chow  
*Testing Software Design Modeled by Finite-State Machines*  
*IEEE Transactions on Software Engineering*, Vol SE-4, No 3, May, 1978

- [CHSM] Paul J. Lucas  
An Object-Oriented System for Implementing Concurrent, Hierarchical,  
Finite State Machines.  
*MSc. Thesis*, University of Illinois at Urbana-Champaign, 1993
- [Clocksin] W.F. Clocksin & C.S. Mellish  
*Programming in Prolog*, 2nd Edition  
Springer Verlag, 1984.
- [CWB] The Edinburgh Concurrency Workbench  
<http://www.dcs.ed.ac.uk/home/cwb/>
- [Dahbura] Anton T. Dahbura, Krishnan K. Sabnani, and M. Ümit Uyar  
*Formal Methods for Generating Protocol Conformance Test Sequences*  
Proceedings of the IEEE, Vol. 78, No. 8, August, 1990
- [CMMI] CMMI-SE/SW, Version 0.2b, Sept 1999  
*Capability Maturity Model - Integrated Systems/Software Engineering*  
CMMI website: <http://www.sei.cmu.edu/cmmi/cmmi.html>
- [Conformiq] <http://www.conformiq.com>
- [Darnell] P.A. Darnell and P.E. Margolis  
C: A Software Engineering Approach  
Springer Verlag, 1988. ISBN 0-387-97389-3 and 3-540-97389-3
- [DejaGnu] R. Savoye  
*The DejaGnu Testing Framework*  
The Free Software Foundation, 1993
- [Du Bousquet] Lydie Du Bousquet, Solofo Ramangalahy, Séverine Simon, César Viho  
Formal Test Automation: The Conference Protocol with TGV/TorX  
Available on the web at the [TorX] site.
- [ECHSM] M.J. Hollenberg  
Extended Hierarchical Concurrent State Machines, Syntax and Semantics  
*Philips Nat. Lab. Draft Report*, 1999

- [Farchi] E. Farchi, A. Hartman, S.S. Pinter  
Using a model-based test generator to test for standard conformance  
*IBM Systems Journal*, Vol. 41, No 1, 2002.
- [Friedman] Galit Friedman, Alan Hartman, Kenneth Nagin, Tomer Shiran  
Projected State Machine Coverage  
IBM Haifa Research Laboratory  
Presentation ISSTA 2002
- [Harel] D. Harel et al.  
On the Formal Semantics of Statecharts  
*Logic in Computer Science*, 2nd Annual Conference, 1987, pp.54-64
- [Hennie] F.C. Hennie  
Fault Detecting for Sequential Circuits  
*Proceedings of the 5th Annual Symposium on Switching Theory and Logical Design*, 1964, pp. 95-110.
- [Hierons 98] Rob M. Hierons  
Adaptive testing of a deterministic implementation against a  
nondeterministic finite state machine  
*The Computer Journal*, 41, 5 (1998), pp. 349-355
- [Hierons 03] Rob M. Hierons  
Generating candidates when testing a deterministic implementation against  
a non-deterministic finite state machine  
*The Computer Journal*, 46, 3, pp. 307-318
- [Hoare] C.A.R. Hoare  
*Communicating Sequential Processes*  
Prentice-Hall, 1985, ISBN 0-13-153271-5, 0-13-153289-8 PBK
- [Hollenberg] M.J. Hollenberg  
Test Templates for PHACT  
*Philips Nat. Lab. Technical Note* 152/99



- [Hong] Hyoung Seok Hong, Jeong Hyun Kim, Sung Deok Cha and Yong Rae Kwon  
Static Semantics and Priority Schemes for Statecharts  
*Proceedings of COMPSAC '95*, IEEE Computer Society Press.
- [IEEE 610.12.1990] *IEEE Standards, Software Engineering*  
Volume I, Customer and Terminology Standards, 1999 Edition
- [ISO 9646-1] International Organization for Standardization  
ISO/IEC 9646-1 (1994)  
Information technology -- Open Systems Interconnection -- Conformance testing methodology and framework -- Part 1: General concepts
- [Jagadeesan] Lalita Jategaonkar Jagadeesan et al.  
Specification-Based Testing of Reactive Software:  
Tools and Experiments. Experience report,  
*Proceedings of the International Conference on Software Engineering*,  
May 1997
- [Koala] R. van Ommering, F. van der Linden, J. Kramer, J. Magee  
The Koala Component model for Consumer Electronics Software  
*IEEE Computer*, March 2000, pp. 78-85.
- [Koppalkar 02] Nitin Koppalkar and Animesh Bhowmick  
Integration of Generic Explorer with the TorX Tool Chain  
*Philips Nat. Lab. Technical Note 2002/387*
- [Koppalkar 03] Nitin Koppalkar  
Interfacing STATECRUNCHER with TorX for demonstrating the state-based testing technique taking MG-R components for a case study  
*Philips Nat. Lab. Draft Report*, December 2003
- [Koymans] Ron Koymans  
An Overview of Automatic Test Generation Techniques for Communication Protocols  
*Philips Nat. Lab. Report RWR-508-re-93558*, November, 1994

- [Kwan] Kwan Mei-Ko  
Graphic Programming Using Odd or Even Points  
Chinese Mathematics 1962, Vol. 1, pp. 273-277
- [Lee] David Lee and Mihalis Yannakakis  
Principles and Methods of Testing Finite State Machines  
*Proceedings of the IEEE*, Vol. 84, No 8, August, 1996
- [Myers] G.J. Myers  
*The Art of Software Testing*  
John Wiley & Sons, 1979. ISBN 0-471-04328-1
- [Petrenko] Alexandre Petrenko, Nina Yevtushenko, Alexandre Lebedev,  
Anindya Das  
Nondeterministic State Machines in Protocol Conformance Testing  
*Protocol Test Systems VI (C-19)*, pp. 363-378, 1994
- [PHACT] L. Heerink and M. Hollenberg  
Conformance Testing Using PHACT  
*Philips Nat. Lab. Technical Note NL-TN 2000/011*
- [Phadke] Madhav S. Phadke  
Planning efficient software tests  
<http://www.stsc.hill.af.mil/crosstalk/1997/10/planning.asp>
- [Sabnani] Krishnan Sabnani and Anton T. Dahbura  
A Protocol Test Generation Procedure  
*Computer Networks and ISDN Systems* 15 (1988), pp. 285-297
- [Schneider] Steve Schneider  
*Concurrent and Real-time Systems, The CSP Approach*  
John Wiley & Sons Ltd, 2000, ISBN 0-471-62373-3
- [Sloane] N.J.A. Sloane  
A library of orthogonal arrays  
<http://www.research.att.com/~njas/doc/OA.html>

- [SPIN] <http://netlib-bell-labs.com/netlib/spin/whatisspin.html>
- [StCr...] *For STATECRUNCHER appendices/references, please see the start of this section.*
- [SWI Prolog] <http://www.swi-prolog.org>
- [Trew 98] Tim Trew  
State-based Testing with WinRunner: the State-Relation Package  
*Philips PRL Internal Note SEA/704/98/05, June 1998*
- [Trew 01] Tim Trew  
Software Component Composition - Still "Plug and Pray?"  
*Proceedings of the 6<sup>th</sup> Philips Software Conference, February, 2001*
- [Trew 03] Tim Trew and Sessaiah Uppala  
State-based modelling of software components for integration testing  
A practical guide to the creation of STATECRUNCHER models  
*Philips Nat. Lab. Technical Note (under preparation).*
- [UML] The Object Management Group website is: <http://www.omg.org>  
UML specifications are available from this website.
- [VnV] Eleen Hollenberg and Erik Mallens  
Cvntestframe User Manual  
*MG-R Software Documentation, v2.0, October 2001.*
- [von der Beeck] Michael von der Beeck  
A Comparison of Statechart Variants  
*Aachen University of Technology, Aachen, Germany*
- [Warren] David H.D. Warren  
Logic Programming and Compiler Writing  
*Software Practice and Experience, Vol. 10, 97-125 (1980).*
- [WinProlog] WinProlog, Logic Programming Associates Ltd  
<http://www.lpa.co.uk>

- [WinRun] WinRunner v4.0/v5.01, Mercury Interactive  
<http://www.merc-int.com/products/winrunguide.html>
- [Yule] D.C. Yule  
Automatic State-Based Testing  
*Philips PRL Technical Note* TN 3611, 1997 / DVD Document V19 C4 S415.
- [Zhang] Fan Zhang and To-yat Cheung  
Optimal Transfer Trees and Distinguishing Trees for testing Observable nondeterministic Finite-State Machines  
*IEEE Transactions on Software Engineering*, Vol. 29, No. 1, Jan. 2003