STATECRUNCHER Test Models

Graham G. Thomason

Report Relating to the Thesis "The Design and Construction of a State Machine System that Handles Nondeterminism"

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

July 2004

STATECRUNCHER Test Models

This document provides diagrams of STATECRUNCHER test models for testing STATECRUNCHER itself, (not for testing an "Implementation Under Test" of some other system). For most test models it will be clear what is being demonstrated or tested. To explain each model in detail, and to show its output, would multiply the size of this report by a considerable factor. That is not necessary, for two reasons: (1) the italicised annotations to the models are intended to clarify subtleties and (2) there is a manual/tutorial that discusses many of the models, often in a simpler form, as part of the training material. Most of the models are exercised in detail under program control in the test suite. The test suite provides an extra resource should it be necessary to see how the model is driven there.

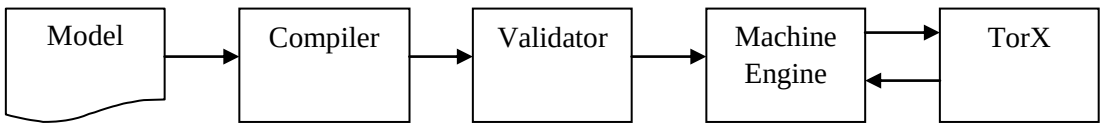
Contents

1. Introduction

This document provides diagrams of STATECRUNCHER test models for testing STATECRUNCHER itself, (not for testing an "Implementation Under Test" of some other system). In addition to these test models, the STATECRUNCHER test suite contains many thousands of tests that do not require any model to be loaded. In fact such lower-level tests form the bulk of the tests for the internal logic and API (Application Programmer Interface). But from the point of view of demonstrating the system, interaction with complete models is most attractive, and a diagram of the model is by far the most expressive way to communicate the functionality being exercised.

The following diagram shows the processes applied to a model as it is compiled, validated and deployed in a testing tool chain such as TorX [http://fmt.cs.utwente.nl/CdR].

Figure 1. Compilation, Validation and Application to a Testing Tool Chain



More details of the parsing process are given in [StCrParsing]. Details of STATECRUNCHER as a whole are given in [StCrMain].

STATECRUNCHER is currently implemented in PROLOG. STATECRUNCHER's own syntax is independent of PROLOG, but occasionally a remark reflects the implementation language. The PROLOG-based test harness used to self-test STATECRUNCHER is described in [StCrGP4].

For most test models it will be clear what is being demonstrated or tested. To explain each model in detail, and to show its output, would multiply the size of this report by a considerable factor. The italicised annotations to the models are intended to clarify subtleties. Most of the models are exercised in detail under program control in the test suite. The test suite provides an extra resource should it be necessary to see how the model is driven there.

1.1 Categories of Models

The models fall into various categories, in order to satisfy testing requirements per phase during development:

- Models designed to test the ***compiler***, but ignoring validator and run-time (machine engine) considerations

- Models designed to test the ***validator***, but not aimed at machine-engine execution. The validator is a kind of back-end to the compiler; it generates a symbol table, cross reference table, and initial data predicates (settings).
- Miscellaneous ***example*** models (e.g. as used in demonstrations and reports), but not attempting any systematic coverage of functionality
- Models designed to *demonstrate* the run-time ***machine engine*** - (1), a feature-by-feature approach, in an illustrative or didactic way, but without attempting to cover every detail.
- Models designed to *systematically test* the run-time ***machine engine*** - (2), where a more structured testing approach has been taken.

Model numbering

Models are numbered by an index such as `t4120` or `c2117`. In the `ci_sc_1.pl` module, a link is set between model number and filename (including path). An example of such a link, using relative path addressing with respect to a 'root' path defined in the boot file, is

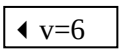
```
ci_file(t5110,'..\StCr3ModelsTest\t5000me\t5110_HelloWorld\HelloWorld').
```

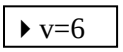
Any one file can be made active for compiling, validating and exercising by setting `ci_current(`*`model-index`*`)` in the `ci_sc_1.pl` file.

File `ci_sc_1.pl` and indices of the kind `t`*`nnnn`* are reserved for test-suite models and are part of the formal STATECRUNCHER release. The user can define more files, e.g. in `ci_sc_2.pl`, using an index such as the `c`*`nnnn`* range. The default `ci_current(`*`model-index`*`)` setting should only be defined once and is defined in file `ci_sc_1.pl`.

The numbering is as follows
- t2000 series: compiler tests
- t3000 series: validator tests
- t4000 series: miscellaneous examples
- t5000 series: machine engine demonstrations
- t6000 series: machine engine systematic tests
- t7000 series: stress tests

1.2 Notation

UML now (v1.5) describes a detailed notation for diagrams, but this report differs in respect of certain features:
- on entry to a state (UML "entry/") is a solid triangle pointing in to the state, e.g. 

- on exit from a state (UML "exit/") is a solid triangle pointing out of the state, e.g. 

- events declared in a part of the hierarchy are denoted by the symbol ↯, e.g. 

- variables are declared in a part of the hierarchy by the symbol ⁞⁞ , e.g.

- PCOs (Points of Control and Observation) are declared by the symbol Ⓟ, e.g. 

2. Testing the Compiler

2.1.1 Compiler coverage aspects

The compiler is mainly concerned with syntax rather than issues of legality of use, such as whether an item has been declared, which are checked by the validator. An exception is that the compiler *is* concerned about a proper hierarchical structure of the statechart, and it will produce an error message (and stop compiling) if there are inconsistencies in the hierarchical structure.

Most situations of erroneous STATECRUNCHER syntax result in a parse where the error is tagged in the parse tree. These situations are extensively tested ***in lower level tests without using a model***. Such tests are not described here. The ***models*** are a system test on the compiler, covering its ability to report the main kinds of error and to proceed appropriately.

The compiler recognises three levels of correctness/error

- statement with no errors
- statement with local errors tagged in the parse tree
- failed statement - the statement could not be parsed at all

Test areas

- Brackets errors
- States and the statechart hierarchy: clusters, sets, leafstates
- Declaration statements (PCOs, events, tags, variables)
- I/O stress: multiple line statements, long files.

2.2 The Compiler Test Models

Here we consider the test aims and error circumstances.

Table 1. Compiler test models

Model (directory) name	Test aim
t2110_braces_er	Error reported on mismatched braces
t2120_round_brack_er	Error reported on mismatched round brackets
t2130_square_brack_er	Error reported on mismatched square brackets
t2210_state_ok	Correct handling of a simple `state` statement
t2211_state2_ok	Correct handling of a more `state` statements

t2215_state_er	Detection of errors in `state` statements
t2220_cluster_ok	Correct handling of a `cluster` statements
t2225_cluster_er	Detection of errors in `cluster` statements
t2230_set_ok	Correct handling of a `set` statements
t2235_set_er	Detection of errors in `set` statements
t2240_struct_ok	Correct handling of a hierarchical statechart structure
t2251_struct_er1	Error in hierarchy structure (1)
t2252_struct_er2	Error in hierarchy structure (2)
t2253_struct_er3	Error in hierarchy structure (3)
t2254_struct_er4	Error in hierarchy structure (4)
t2255_struct_er5	Error in hierarchy structure (5)
t2310_decl_ok	Correct handling of declarations
t2315_decl_er	Detection of errors in declarations
t2320_split_stmt	Handling of a statement split over several lines
t2330_medium	A general medium complexity model
t2340_complex	A general complex model
t2350_longfile	Stress test on a long file
t2360_longstmt	Stress test on a long statement

These models are not put through the validator. The validator is tested independently.

3. Testing the Validator

3.1 Validator Coverage Aspects

The purpose of the validator is to generate certain tables and in so doing to detect certain errors. It generates a ***symbol table*** and a ***cross-reference table***, and also a ***data table*** (containing variable values). For more information on these tables, see [StCrParsing]. Validator coverage is considered from the viewpoint of producing the error messages, and from source code error circumstances. This test approach largely verifies the correctness of the tables. Further testing of the correctness of the tables is done with machine engine tests (described in subsequent sections). The individual tests divide into tests for errors that are detected by symbol table construction and by cross-reference table construction.

Some symbol table coverage aspects

- states
- inbuilt-constants (true, false)
- tags
- variables
- PCOs
- events
- scoped use of the above
- double definition of the above

Some cross-reference table coverage aspects

- variable references in initialization of other variables
- variable references in actions
  - upon enter action
  - upon exit action
  - transition assignment action
- variable references in conditions
- variable references as terms of expression operators
- variable references in library function parameters (e.g. maximum)
- event references by transition
- event references by fired event
- state references by orbit
- state references by target
- state references by the `in()` function
- state references by the `clear()` function

- state references by the `deep_clear()` function
- state references as terms of state-expression operators: `:: $ . %% /\`
- PCO references by event declaration

3.2 Catalogue of Validator Error Messages as Written

The errors fall into the following categories

- warnings
- general errors: version incompatibility, compiler error detection
- type checking
- detection of non-implemented functions
- internal errors (diagnostic error – the program logic *should* preclude these)

Table 2. Validator error messages

`write('** Error (VA-E-001) ** Code is in testing mode: va_testing(yes)')`
`write('** Error (VA-E-002) ** There are compilation errors')`
`write('** Warning (VA-W-003) ** Multiple files loaded')`
`write('** Error (VA-E-004) ** No "object" files loaded')`
`write('** Error (VA-E-005) ** Version incompatibility')`
`write('** Error (VA-E-006) ** Double definition of '),` `  write(SYMB),write(':'),write(MPATH),`
`write('** Error (VA-E-007) ** Uninitialized term(s) in initialization of '),` `  write(SYMBOL),write(':'),write(MPATH),`

`write('** Error (VA-E-008) ** Boolean value error initializing '),`
`  va_err_nltab,`
`  write(SYMBOL),write(':'),write(MPATH),write('.'),`
`  tab(1),`
`  write(VALUE),write(' not in '),write([0,1]),`

`write('** Error (VA-E-009) ** String value error initializing '),`
`  va_err_nltab,`
`  write(SYMBOL),write(':'),write(MPATH),write('.'),`
`  tab(1),`
`  write(VALUE),write(' is not a string'),`

`write('** Error (VA-E-010) ** Range error initializing '),`
`  va_err_nltab,`
`  write(SYMBOL),write(':'),write(MPATH),write('.'),`
`  tab(1),`
`  write(VALUE),write(' not in '),write([LOW,HIGH]),`

`write('** Error (VA-E-011) ** Enum value error initializing '),`
`  va_err_nltab,`
`  write(SYMBOL),write(':'),write(MPATH),write('.'),`
`  tab(1),`
`  write(VALUE),write(' not in '),write(SET),`


```
write('** Error   (VA-E-012) **  Undefined symbol '),
   va_err_nltab,
   write(DSYMBOL),write(':'),write(EPATH),
   va_err_nltab,
   write('in statement '),write(UTYPE),tab(1),
   write(USYMBOL),write(':'),write(UPATH),
```

```
write('** Error   (VA-E-013) **  Undefined symbol of required type'),
   va_err_nltab,
   write(SYMBOL),write(':'),write(EPATH),
   tab(1),
   write('of type '),write(STYPE),
   va_err_nltab,
   write('in statement '),write(UTYPE),tab(1),
   write(USYM),write(':'),write(UPATH),
```

```
write('** Error   (VA-E-014) **  Polyvalent symbol (in overlapping scopes) '),
   write(SYMBOL),
   write(' is used of types '),write(SYMBOLTYPE),
   write(' and '),write(SYMBOLTYPE2),
   va_err_sep,
```

```
write('** Warning (VA-W-015) **  Polyvalent symbol (but scopes are distinct) '),
   write(SYMBOL),
   write(' is used of types '),write(SYMBOLTYPE),
   write(' and '),write(SYMBOLTYPE2),
```

```
write('** Warning (VA-W-016) **  Unreferenced symbol'),
   tab(1),
   write(DSYMBOL),write(':'),write(DPATH),
   va_wrn_nltab,
   write('of type '),write(DTYPE),
```

```
write('** Error   (VA-E-017) **  Type mismatch in assignment '),
   va_err_nltab,tab(4),write('LHS-TYPE '),write(LHS),
   va_err_nltab,write('<assigned>'),
   va_err_nltab,tab(4),
   ( (
        RHS=[typerr,OP,T1,T2],
         write('RHS-TYPE '),write(typerr),
        va_err_nltab,tab(12),write(T1),
        va_err_nltab,tab(8),write(OP),
        va_err_nltab,tab(12),write(T2)
   );(
        write('RHS-TYPE '),write(RHS)
   ) ),
   va_err_nltab,
   write('in statement '),write(UTYPE),tab(1),
   write(USYM),write(':'),write(UPATH),
```

```
write('** Error   (VA-E-018) **  Type mismatch in expression: '),
   va_err_nltab,
   ( (
        DETAIL=[typerr,OP,T1,T2],
        tab(4),write(T1),
        va_err_nltab,write(OP),
        va_err_nltab,tab(4),write(T2)
   );(
        write(DETAIL)
   ) ),
   va_err_nltab,
   write('in statement '),write(UTYPE),tab(1),
   write(USYM),write(':'),write(UPATH),
```

```
write('** Error   (VA-E-019) **  Non-implemented function: '),
   write(FUN),
```

```
write('*** Internal Error (VA-I-500) ***  va_write_pred '),
   write(PRED),
```

A "polyvalent" symbol is one that is used for two or more different ***kinds*** (e.g. an integer and an event). This is tolerated with a warning if the scopes are distinct. If the scopes overlap, then an error is given, since symbol-table look-up (based on symbol and current scope) is ambiguous – more than one entry could be returned as being in scope. This is a separate issue to that of allowing a symbol to be used for two or more different ***scopes***. This is a legal situation which occurs where a symbol has several definitions, usually in of the same kind, but which are distinguished by their scope. Symbol-table look-up is unambiguous, since only the symbol with the innermost scope is taken.

The following are no longer in use: VA-E-001 (testing mode is no longer needed) and VA-E-012 (superseded by VA-E-013). The program logic should prevent VA-I-500 from ever appearing. The remaining error messages are covered in the tests.

3.3 The Validator Test Models

Here we consider the test aims and error circumstances.

Table 3. Validator test models

Model (directory) name	Test aim
t3020_cp_er	Validator error if compiler gave an error
t3031_mult_file1	Validator warning if multiple compiled files loaded
t3032_mult_file2	(used to produce a second file for above)
t3040_no_obj	Validator error if no object file loaded
t3050_vers_incompat	Validator error if file was compiled under an earlier version
t3110_tag_ok	Tag names: normal correct usage, no errors
t3115_tag_er	Tag names: error situations
t3120_var_bool_ok	Boolean variables: normal correct usage, no errors

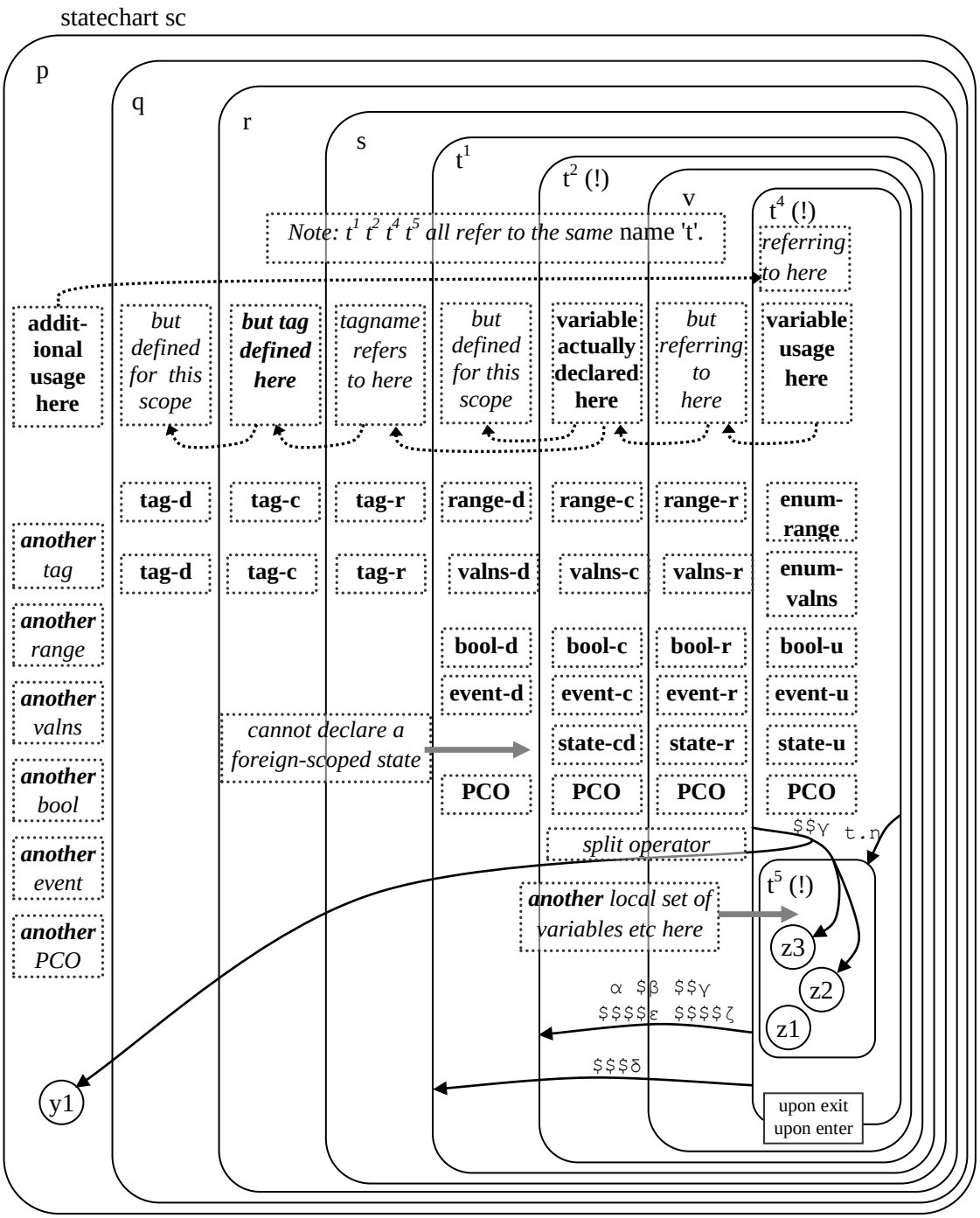
t3125_var_bool_er	Boolean variables: error situations
t3130_var_string_ok	String variables: normal correct usage, no errors
t3135_var_string_er	String variables: error situations
t3140_var_tagrange_ok	Tag-ranged variables: normal correct usage, no errors
t3141_var_tagrange_med	Tag-ranged variables: additional medium model
t3145_var_tagrange_er	Tag-ranged variables: error situations
t3150_var_tagenum_ok	Tag-enumerated variables: normal correct usage, no errors
t3151_var_tagenum_med	Tag-enumerated variables: additional medium model
t3155_var_tagenum_er	Tag-enumerated variables: error situations
t3210_pco_ok	PCOs: normal correct usage, no errors
t3215_pco_er	PCOs: error situations
t3220_evt_ok	Events: normal correct usage, no errors
t3225_evt_er	Events: error situations
t3230_sta_ok	States: normal correct usage, no errors
t3231_sta_basic	States: additional model
t3235_sta_er	States: error situations
t3240_fun_ok	Functions: normal correct usage, no errors
t3245_fun_er	Functions: error situations
t3340_doubdef	Extra double definition tests
t3360_polyvalent	Polyvalent (overloaded) symbol warning/errors
t3370_BasTypChk	Basic Type checking
t3371_AdvTypChk	Advanced type checking
t3910_stxr_ok	A detailed model illustrating scoping issues

Figure 2 following shows a model that tests that items (tags, variables, events, states and PCOs) are correctly addressed where it is necessary to search from the given scope outwards in the state hierarchy (the *outbound search*). It especially tests variables and their declarations, and the declaration of their type. A worst-case scenario is as follows. A variable is used in an expression which is to be evaluated in a certain scope. The variable is operated on by scoping operators, giving a new *evaluated scope* of that variable. But the variable is not found in exactly that scope. However, it is found in a more global scope by the "outbound search". This is the *declared scope* of the variable, although the declaration may have been made in a part of the hierarchy that has yet another scope, but using scoping operators so as to effectively declare as if in the part of the hierarchy that is the declared scope.

When a variable is declared, it has a type defined by the tagname, defining the enumerators or range. The tagname in a variable declaration is itself subject to an evaluated scope and declared scope analogously to the variable declaration.

State scopes can only be defined by means of the place of the state definition in the state hierarchy, but there can be several states of the same name. When a state is referenced, as with variables and tagnames, the effectively referenced state depends on any explicit scoping operations and then the outbound search.

Figure 2. Symbol/cross-reference table: To test tags/variables/events/states/PCOs. [Model t3910_stxr_ok] *(stxr_ok=symbol table and cross-reference table ok)*



Note: The exclamation marks draw attention to names are not part of any syntax.

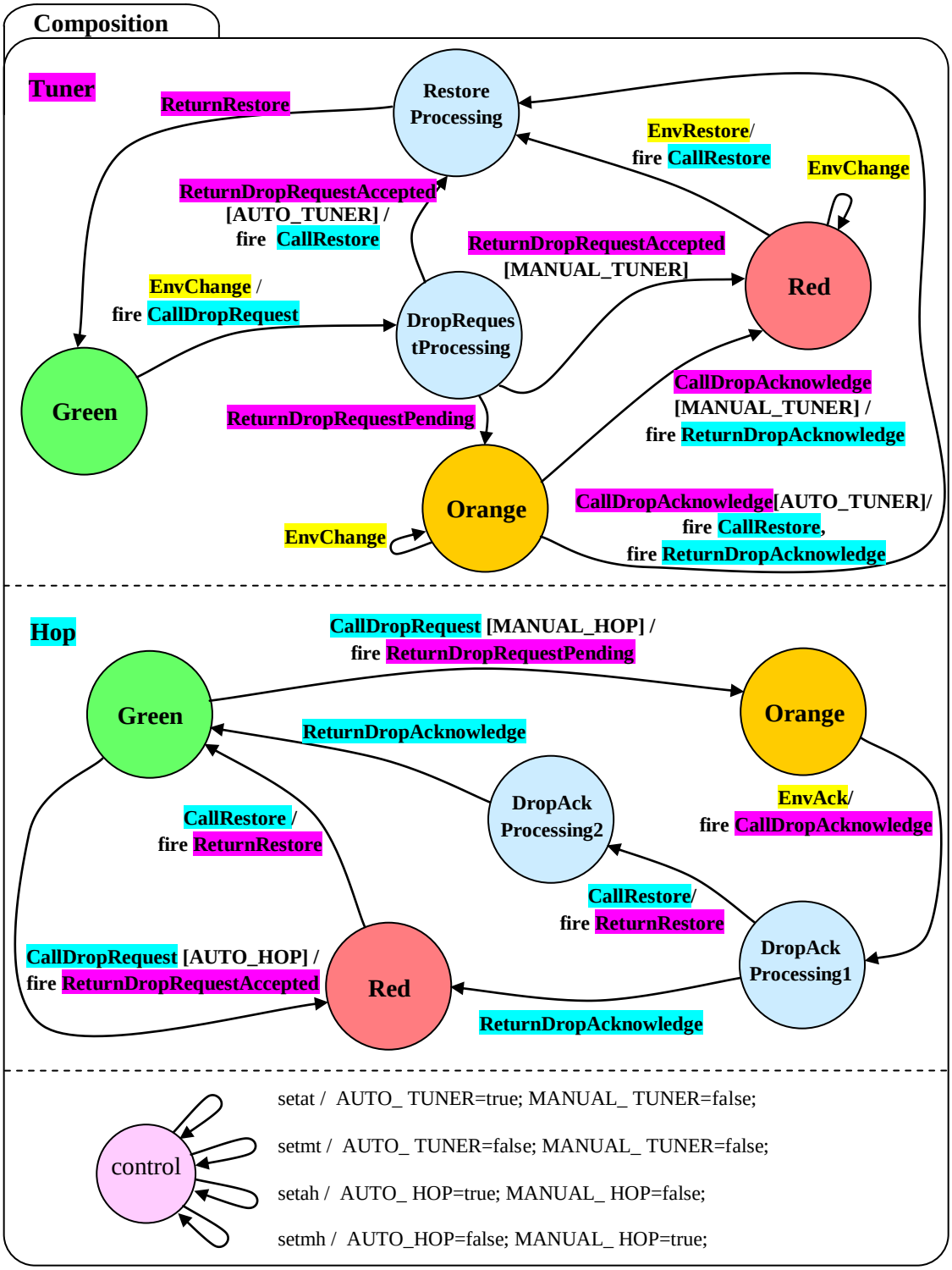
4. Illustrative Examples

These models include examples that have been used in various reports.

- The *Obj_example* model that illustrates object code structure, as exhibited in the *STATECRUNCHER maintenance handbook* (no diagram).
- The *Tie* example of [StCrParsing], (no diagram).
- The *Tuner-Hop* example of a Philips report on component binding[1], p.30 (diagram follows, Figure 3), modelled by Tim trew.
- The *Traces* example in the Transfer Report (diagram follows, Figure 4). The transfer report is a deliverable of the author's PhD registration at the University of Surrey.
- A *Program Installation* model by Tim Trew, for determining the station ID during TV program installation. In this case, the generation of teletext packets is not directly under control of the test harness, and the result of the sequences that might be received is predicted through the `genPckts` state, which exhibits iterative fork nondeterminism on the `next_pkt` event (diagram follows, Figure 5).

[1] G G Thomason
Component Binding in Composite Models for State-based Testing
PRL Technical Note TN 4102, August, 2001

Figure 3. Tuner-Hop (modelled by Tim Trew) [model t4130]



Note colour coding per local event in a component, for Tuner and Hop.

Figure 4. Traces example in transfer report [Model t4140]

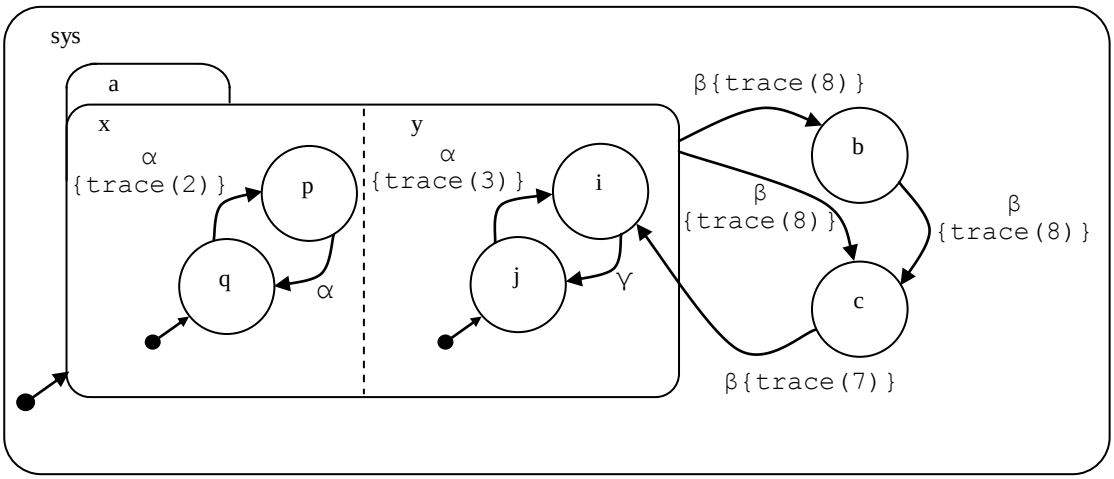
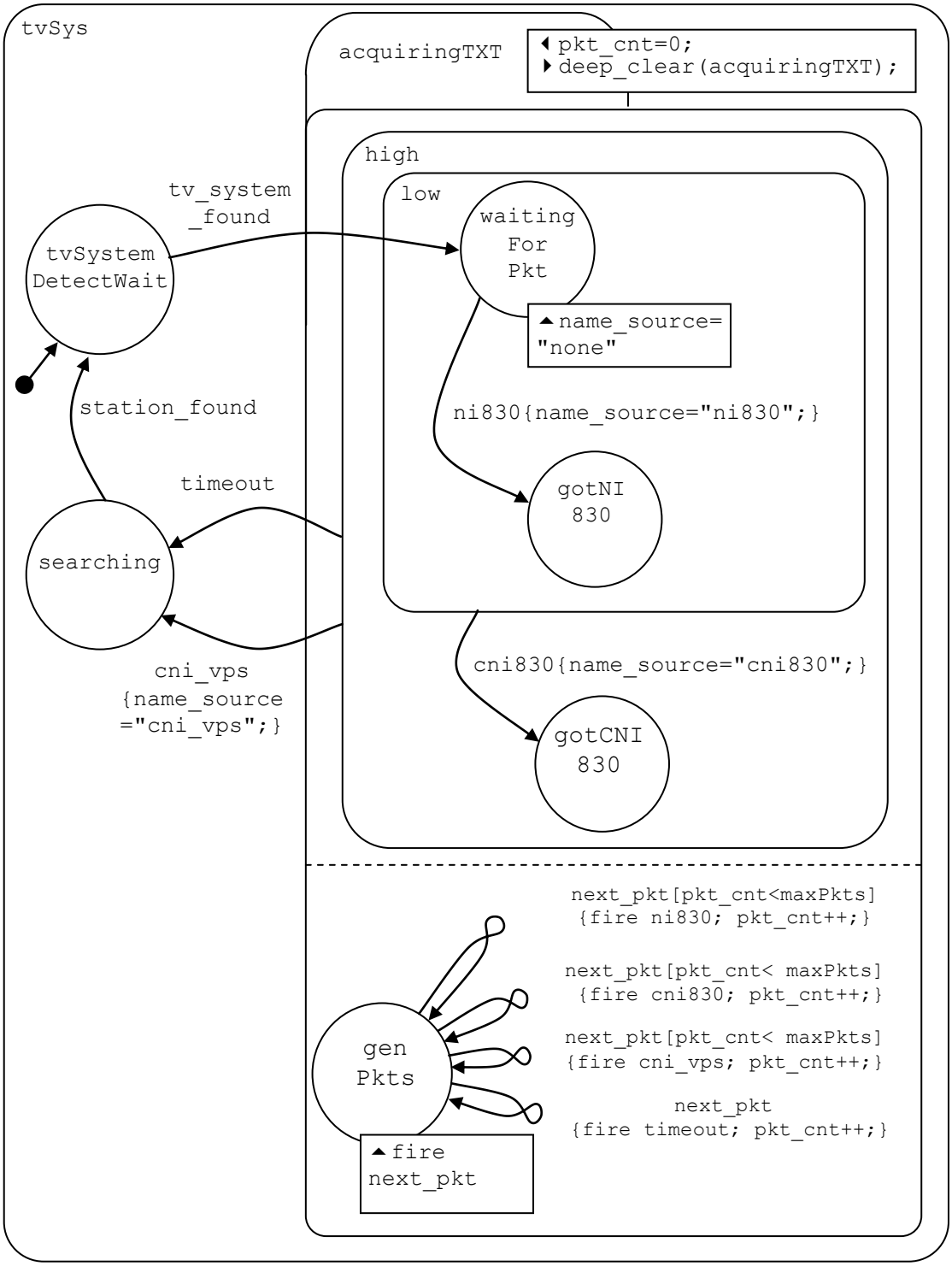


Figure 5. Program Installation (modelled by Tim Trew) [model `t4150`]



The model produces sequences of packets by fork nondeterminism.

Output from this model

The model is driven by turning set-transit nondeterminism off and processing event `tv_system_found`. This can be done interactively, or in a Prolog predicate as follows, where an output file is written in the same directory as the model.

```
go_t4150:-
  me_no_set_tran,                          /* turn set-transit ND off       */
  ci_file(t4150,LOCAL_FILE_NO_EXTN),       /* get model file name           */
  gn_append_atoms(LOCAL_FILE_NO_EXTN,
     '.out.txt',LOCAL_FILE_W_EXTN),        /* add an extension to file name */
  boot_root(sc,BOOT_ROOT),                 /* get boot directory            */
  gn_append_atoms(BOOT_ROOT,
     LOCAL_FILE_W_EXTN,FULL_FILE),         /* make full file name           */
  io_tell(FULL_FILE),                      /* set output to go to this file */
  cs_go(t4150),                            /* load and enter machine        */
  ut_wm,nl,                                /* write machine                 */
  EVENT=[tv_system_found,[sc]],            /* this is the event to process  */
  CALPRARAMS=[],                           /* no parameters to this event   */
  write('About to process '),write(EVENT),nl,nl,
  TASK=[tk_event,[EVENT,CALPARAMS]],       /* wrap the event as a "task"    */
  db_worldbag(INWORLDS),                   /* get the current worlds        */
  me_process_task_in_worlds(TASK,INWORLDS,OUTWORLDS),  /* process task      */
  da_kill_old_worlds,                      /* kill intermediate worlds      */
  ut_wm,                                   /* write machine again           */
  io_told.                                 /* close the file                */
```

This produces an output file `ProgInst.out.txt`. To reduce the output to the essentials (occupied leafstates and key variables), a grep command was executed on it as follows:

```
grep  -E  "(leafstate.*s_occ|name_source|pkt_cnt|^$)"  ProgInst.out.txt  >
grep_out.txt
```

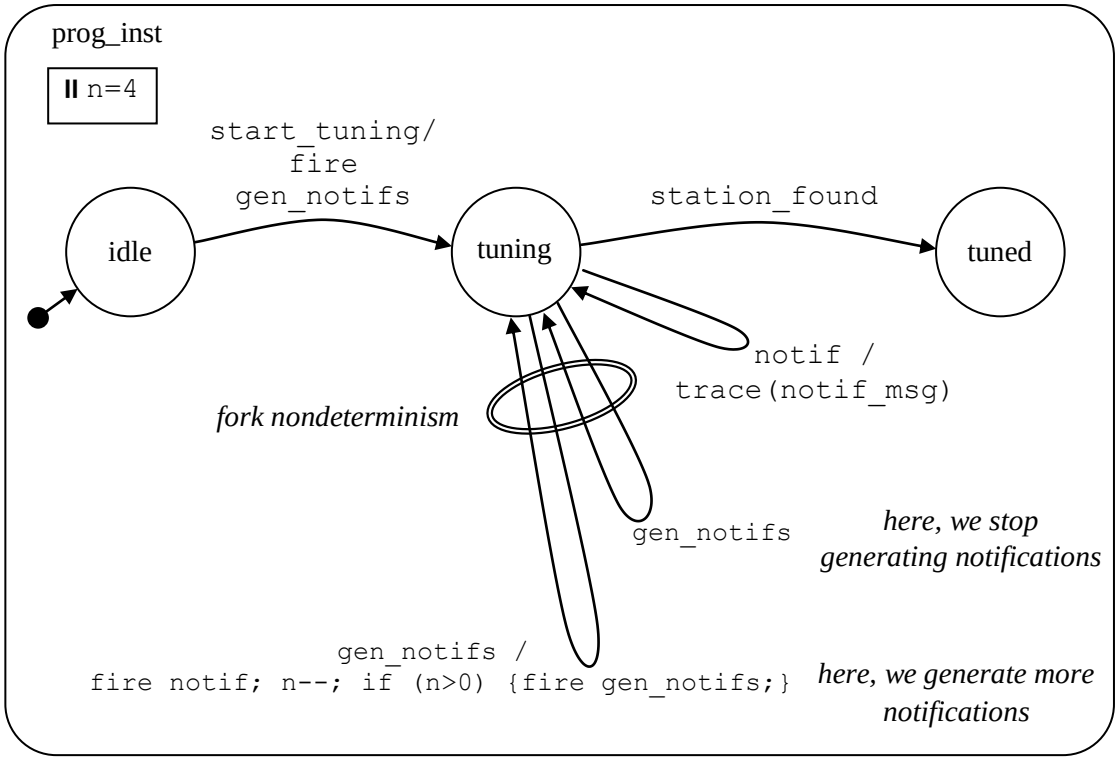
The output (with minor editorial refinements) is as follows

```
SET TRANSIT NONDETERMINISM SWITCHED OFF

9          leafstate searching [tvSys, sc] [s_occ, []]  **
9    VAR  name_source [sc] [vardecl, [string]] =[ex_str, [110, 111 etc]] =none
9    VAR  pkt_cnt [sc] [vardecl, [enumtype, [int1, [sc]]]] =[ex_co, int, 1]

14         leafstate searching [tvSys, sc] [s_occ, []]  **
14   VAR  name_source [sc] [vardecl, [string]] =[ex_str, [99, 110 etc]] =cni_vps
14   VAR  pkt_cnt [sc] [vardecl, [enumtype, [int1, [sc]]]] =[ex_co, int, 1]

22         leafstate searching [tvSys, sc] [s_occ, []]  **
22   VAR  name_source [sc] [vardecl, [string]] =[ex_str, [99, 110 etc]] =cni830
22   VAR  pkt_cnt [sc] [vardecl, [enumtype, [int1, [sc]]]] =[ex_co, int, 2]

27         leafstate searching [tvSys, sc] [s_occ, []]  **
27   VAR  name_source [sc] [vardecl, [string]] =[ex_str, [99, 110 etc]] =cni_vps
27   VAR  pkt_cnt [sc] [vardecl, [enumtype, [int1, [sc]]]] =[ex_co, int, 2]

33         leafstate searching [tvSys, sc] [s_occ, []]  **
33   VAR  name_source [sc] [vardecl, [string]] =[ex_str, [99, 110 etc]] =cni830
33   VAR  pkt_cnt [sc] [vardecl, [enumtype, [int1, [sc]]]] =[ex_co, int, 3]

38         leafstate searching [tvSys, sc] [s_occ, []]  **
38   VAR  name_source [sc] [vardecl, [string]] =[ex_str, [99, 110 etc]] =cni_vps
38   VAR  pkt_cnt [sc] [vardecl, [enumtype, [int1, [sc]]]] =[ex_co, int, 3]

44         leafstate searching [tvSys, sc] [s_occ, []]  **
44   VAR  name_source [sc] [vardecl, [string]] =[ex_str, [99, 110 etc]] =cni830
44   VAR  pkt_cnt [sc] [vardecl, [enumtype, [int1, [sc]]]] =[ex_co, int, 4]

81         leafstate searching [tvSys, sc] [s_occ, []]  **
81   VAR  name_source [sc] [vardecl, [string]] =[ex_str, [110, 105 etc]] =ni830
81   VAR  pkt_cnt [sc] [vardecl, [enumtype, [int1, [sc]]]] =[ex_co, int, 2]

117        leafstate searching [tvSys, sc] [s_occ, []]  **
117  VAR  name_source [sc] [vardecl, [string]] =[ex_str, [110, 105 etc]] =ni830
117  VAR  pkt_cnt [sc] [vardecl, [enumtype, [int1, [sc]]]] =[ex_co, int, 3]

136        leafstate searching [tvSys, sc] [s_occ, []]  **
136  VAR  name_source [sc] [vardecl, [string]] =[ex_str, [110, 105 etc]] =ni830
136  VAR  pkt_cnt [sc] [vardecl, [enumtype, [int1, [sc]]]] =[ex_co, int, 4]
```

With set transit nondeterminism switched on, the following additional output is obtained (due to the action on `genPkts` being executed prior to the action on `waitingForPkt`.

```
158        leafstate searching [tvSys, sc] [s_occ, []]  **
158  VAR  name_source [sc] [vardecl, [string]] =[ex_str, [110, 111 etc]] =none
158  VAR  pkt_cnt [sc] [vardecl, [enumtype, [int1, [sc]]]] =[ex_co, int, 4]

159        leafstate searching [tvSys, sc] [s_occ, []]  **
159  VAR  name_source [sc] [vardecl, [string]] =[ex_str, [110, 111 etc]] =none
159  VAR  pkt_cnt [sc] [vardecl, [enumtype, [int1, [sc]]]] =[ex_co, int, 3]

160        leafstate searching [tvSys, sc] [s_occ, []]  **
160  VAR  name_source [sc] [vardecl, [string]] =[ex_str, [110, 111 etc]] =none
160  VAR  pkt_cnt [sc] [vardecl, [enumtype, [int1, [sc]]]] =[ex_co, int, 2]
```


Figure 6. Notification example [model `t4152`]



This model is discussed in [StCrMain].

5. Testing the Machine Engine: Small Test/Demonstration Models

Ideally, each model would be accompanied by a full explanation, and by the test scripts with expected output. However, space does not permit. The title of each model indicates what is being demonstrated or tested. The test scripts are part of the STATECRUNCHER delivery (see directory `am_sc`). The diagrams give the general reader an overview of STATECRUNCHER functionality and the extent of testing. But the main purpose of the diagrams is as a reference document, serving a certain tutorial function, for discussions amongst STATECRUNCHER users.

Variables and events will always be declared in the diagram if their scope is significant, otherwise their declaration will not necessarily be shown. See Section 1.2 for the notation.

The following models may contain more events and transitions than are marked, to provide direct access to all required states. We call these *omega transitions* – see Section 8.1.1

5.1 Small Deterministic Models

Figure 7. The *hello world* of state models [model `t5110`]

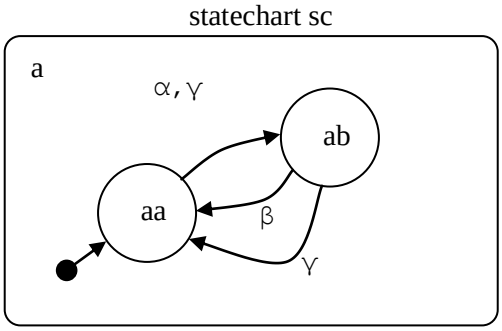
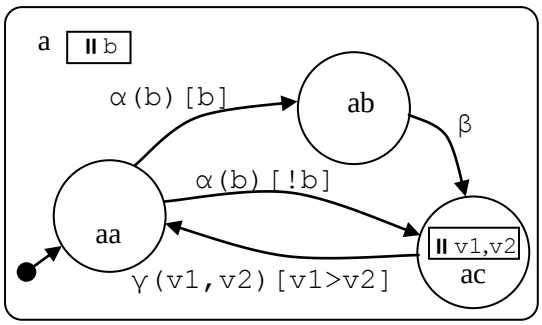


Figure 8. Parameterized, with conditions [models `t5120, t5121, t5122, t5123`]

model `t5120` statechart sc

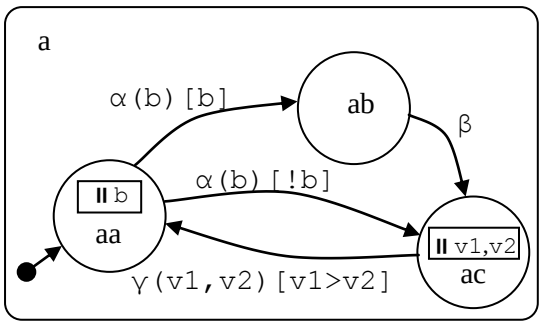


Variables are declared at cluster *and* leafstate scope.

From release 1.05, the *outbound search* technique will find the nearest-scoped variables.

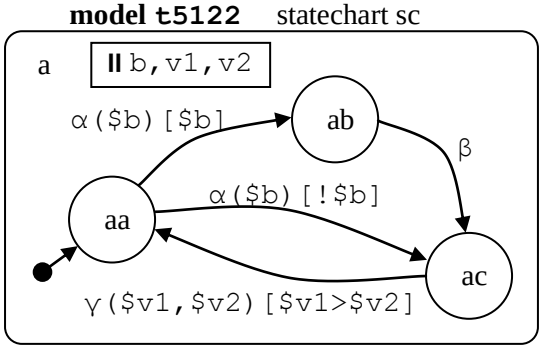
In earlier releases, if the variable was not declared at the specified scope, a hidden variable was created.

model `t5121` statechart sc



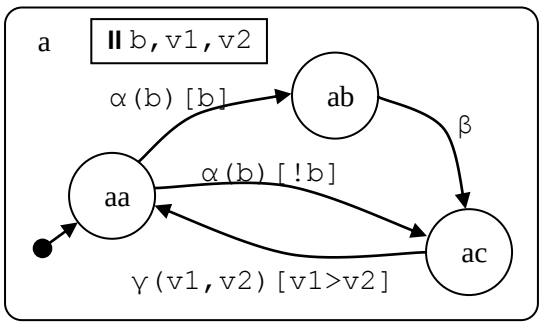
The parameter destinations are local, at leafstate scope. Leafstate scope has to be declared at cluster level with a descend operator (e.g., in `a`, declare `ac.v1`), since there is no place in the syntax to declare at leafstate scope directly.

model `t5122` statechart sc



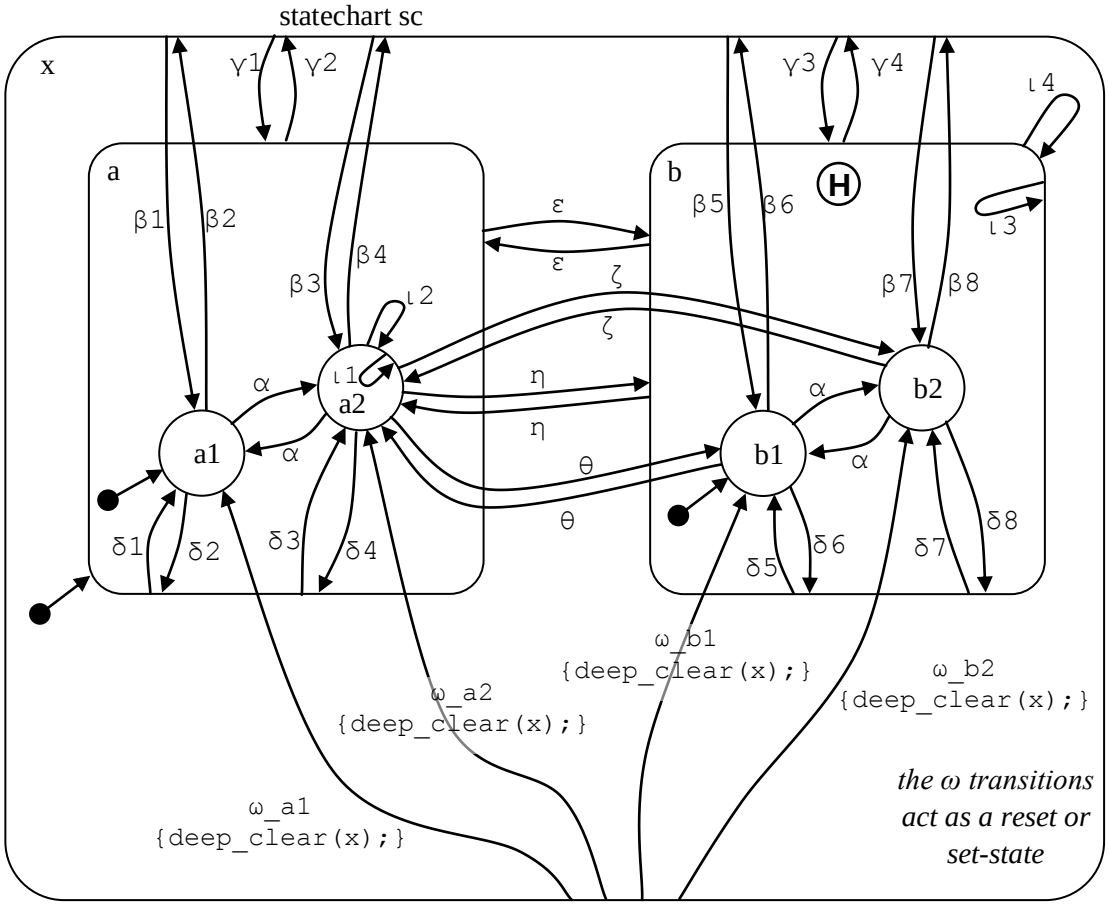
The parameter destinations are at the scope of the cluster. Parameters to events on transitions from leafstates address their destinations using the parent operator, `$`.

model `t5123` statechart sc



The parameter destinations are local - but the destinations are not declared. From release 1.05, the *outbound search* technique will find the nearest-scoped variables. *This arrangement can now be recommended.*

Figure 9. Simple cluster transitions plus history [model `t5130`]



This model also illustrates internal and external self transitions on leaf states and nonleaf states.

Figure 10. Set, but deterministic [model `t5140`]

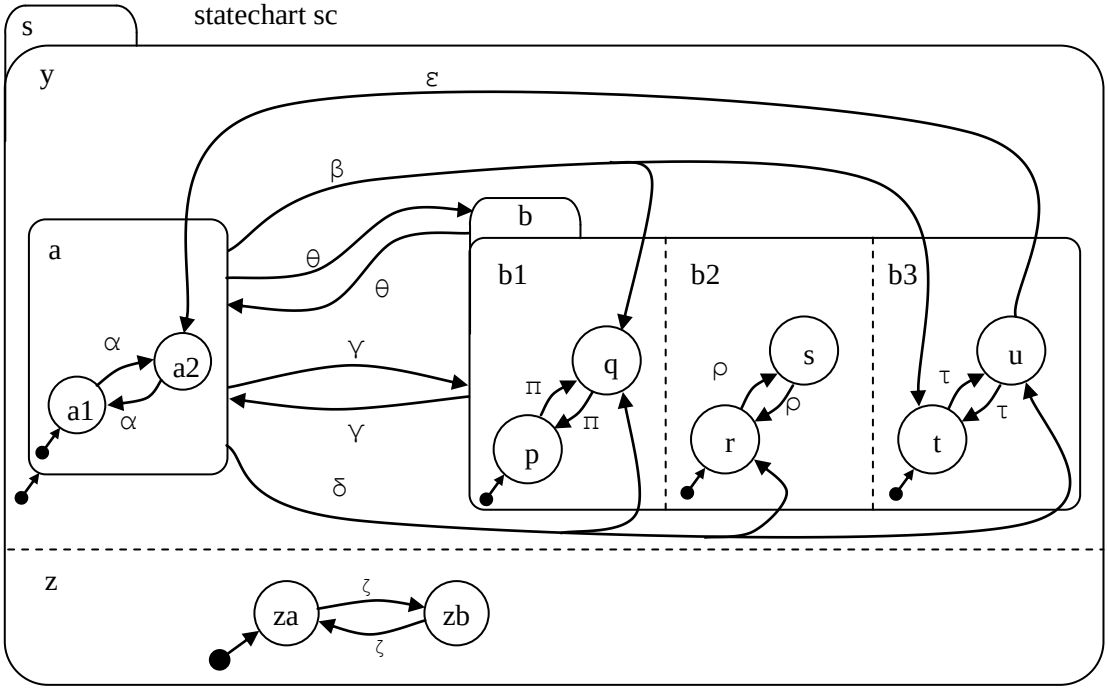
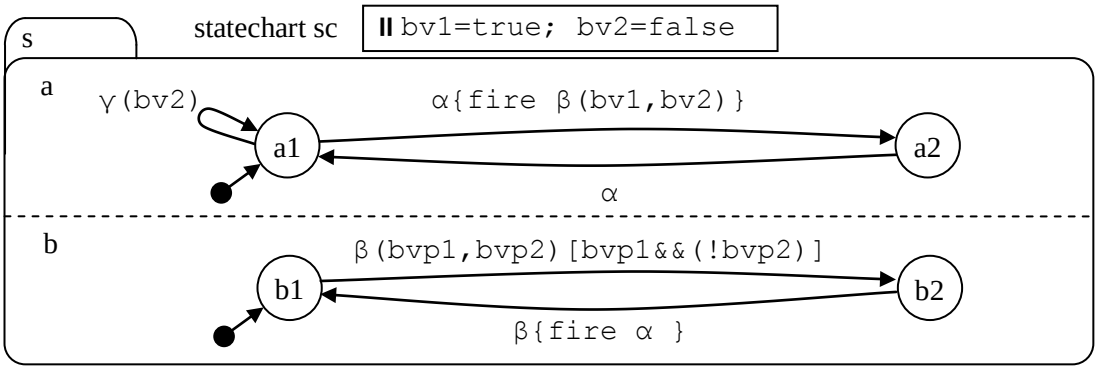


Figure 11. Fired event, but deterministic [model `t5150`]



Model `t5150` explored

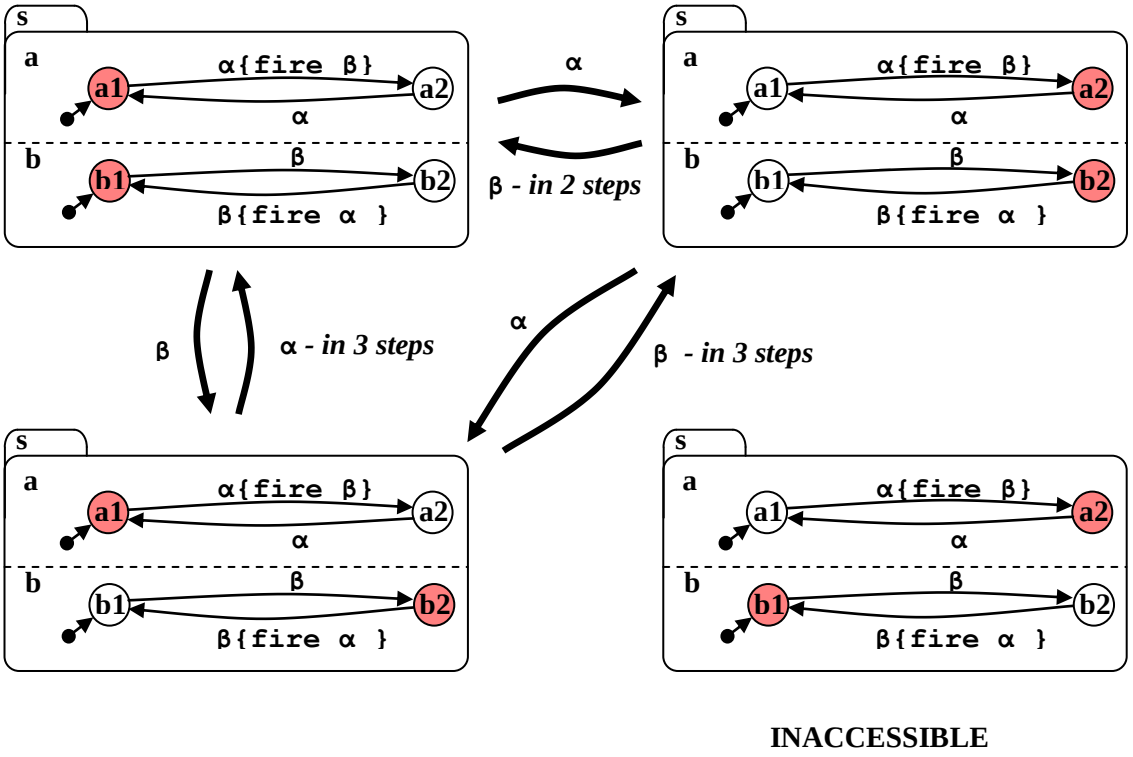


Figure 12. Fired event in series [model `t5152`]

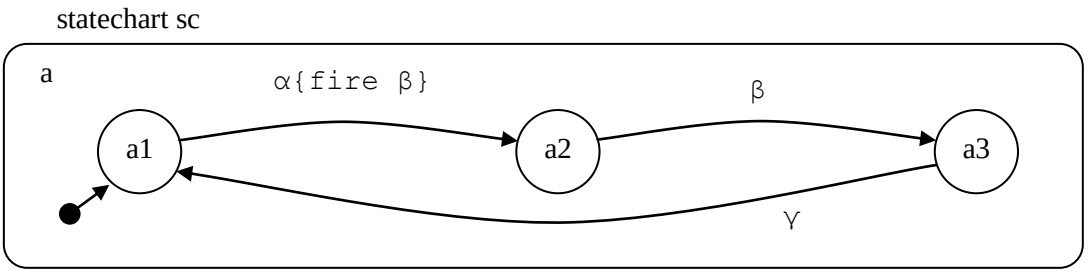


Figure 13. Assignment on transition with overloaded variable names [model `t5160`]

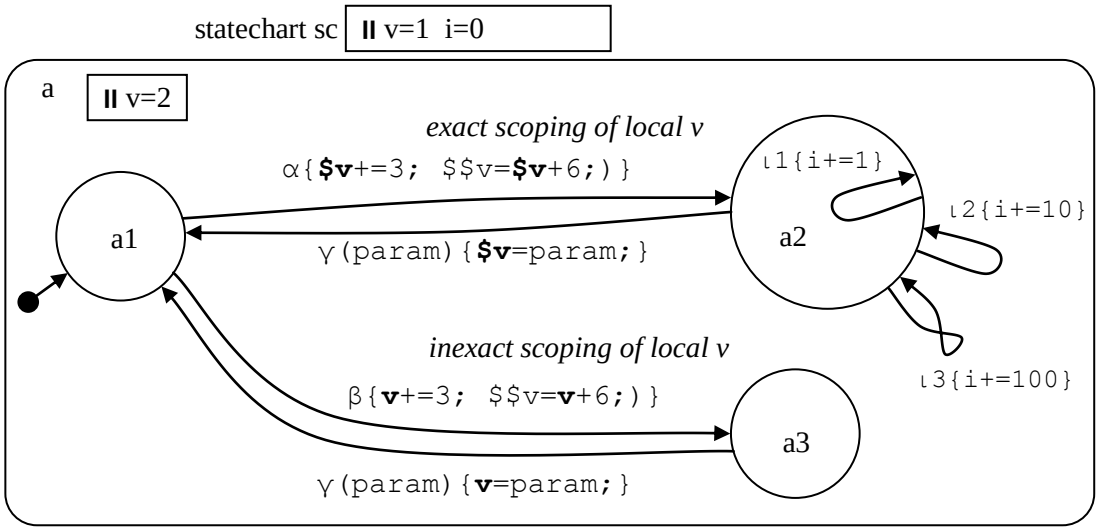


Figure 14. Simple assignment on transition [model `t5161`]

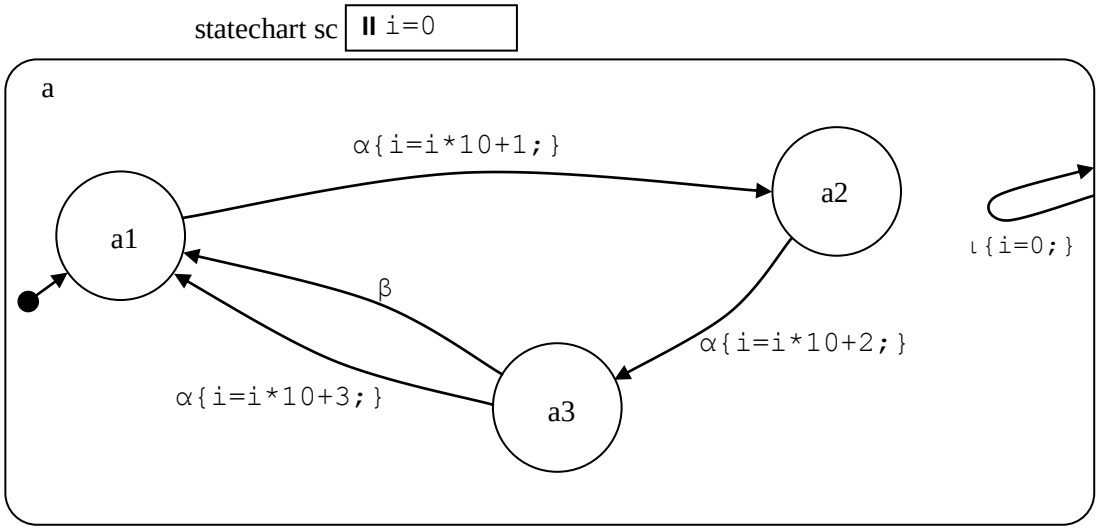
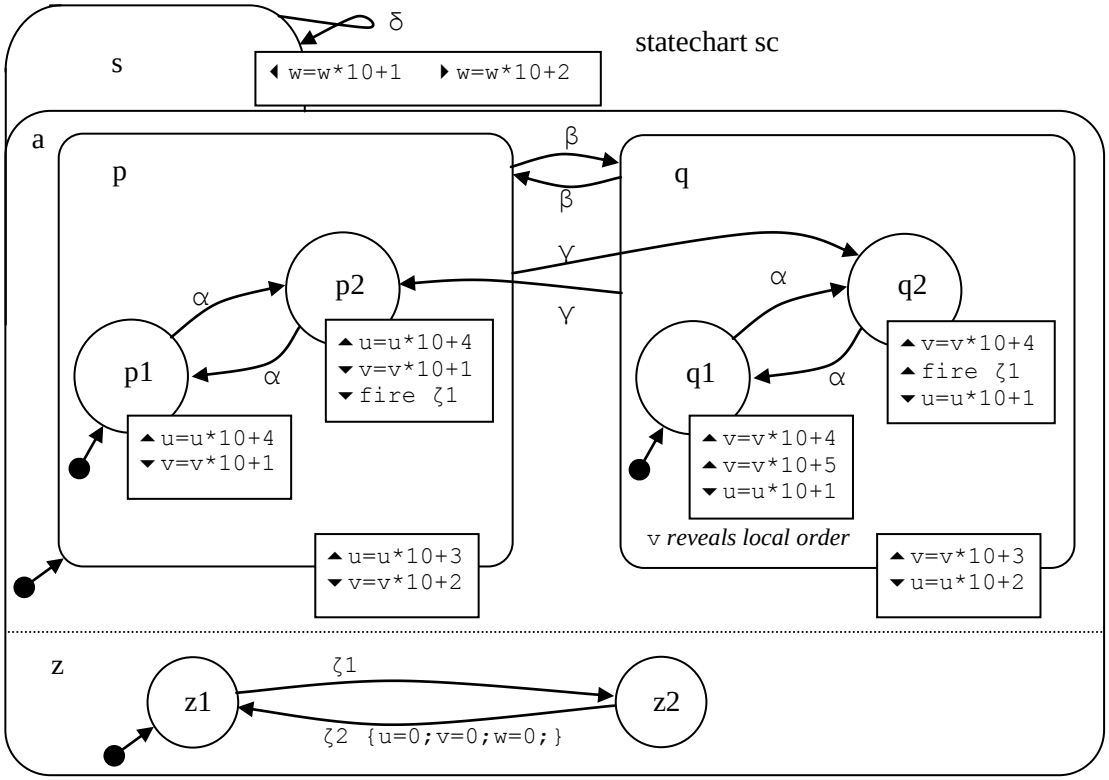


Figure 15. Simple on-enter/ on-exit actions [model `t5170`]



Notes

- Variable v tracks a transition from p to q.
- Variable u tracks a transition from q to p.
- The fired event `ζ1` is only executed in a transition exiting p2 or entering q2.

Figure 16. Simple meta event (state entry/exit) [model `t5180`]

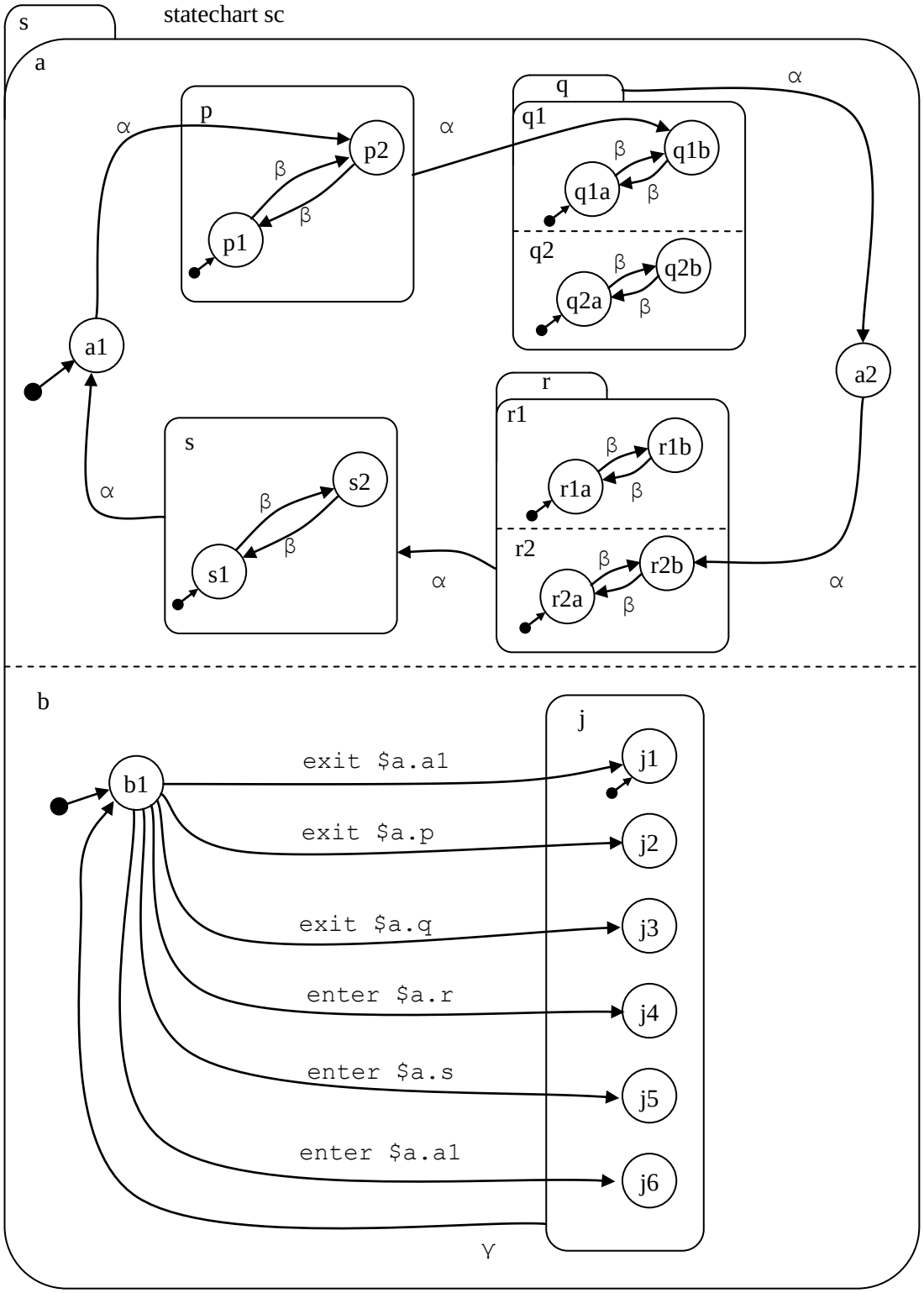


Figure 17. Conditional actions and `in()` function [model `t5190`]

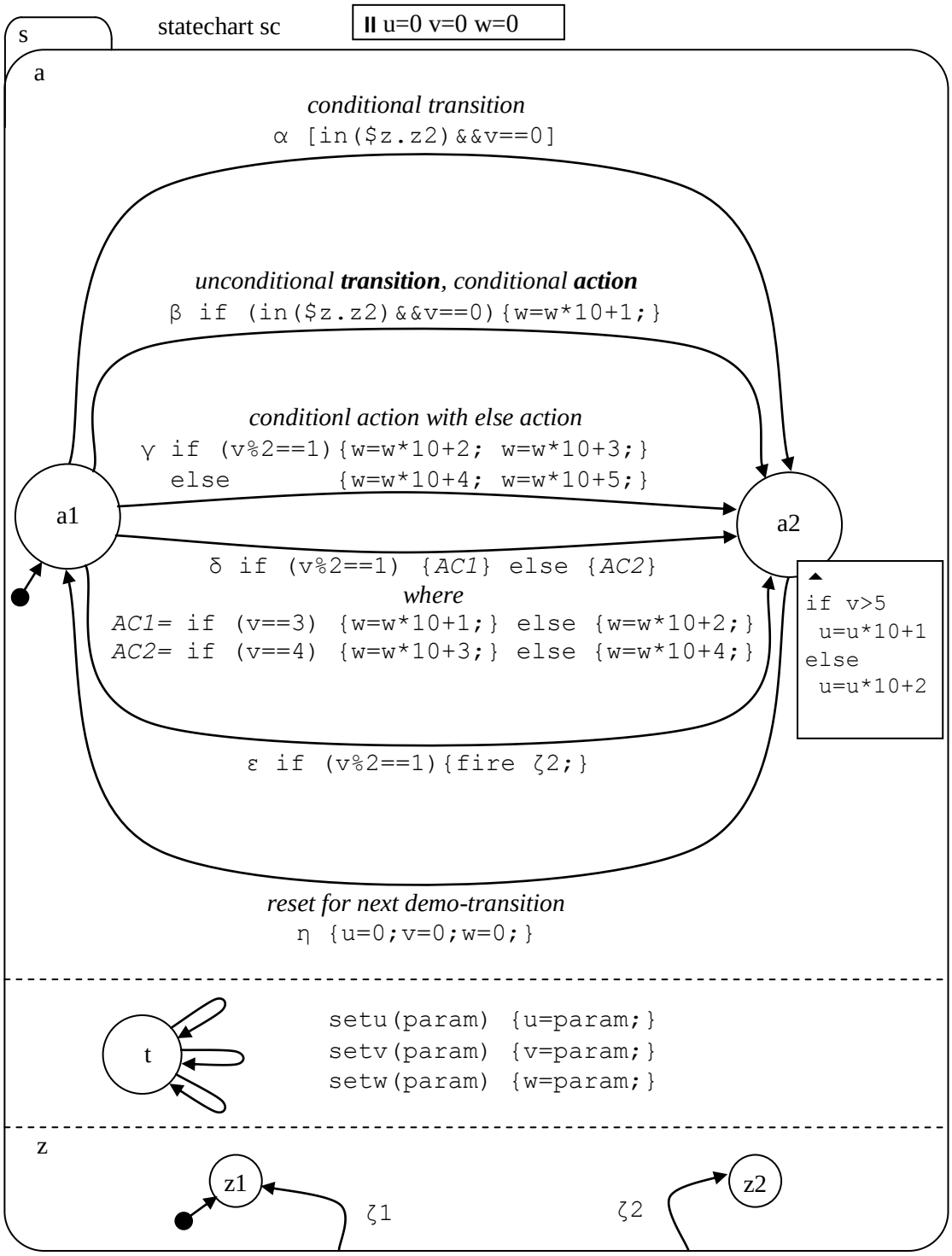


Figure 18. History, Deep History and Clear Functions [model `t5200`]

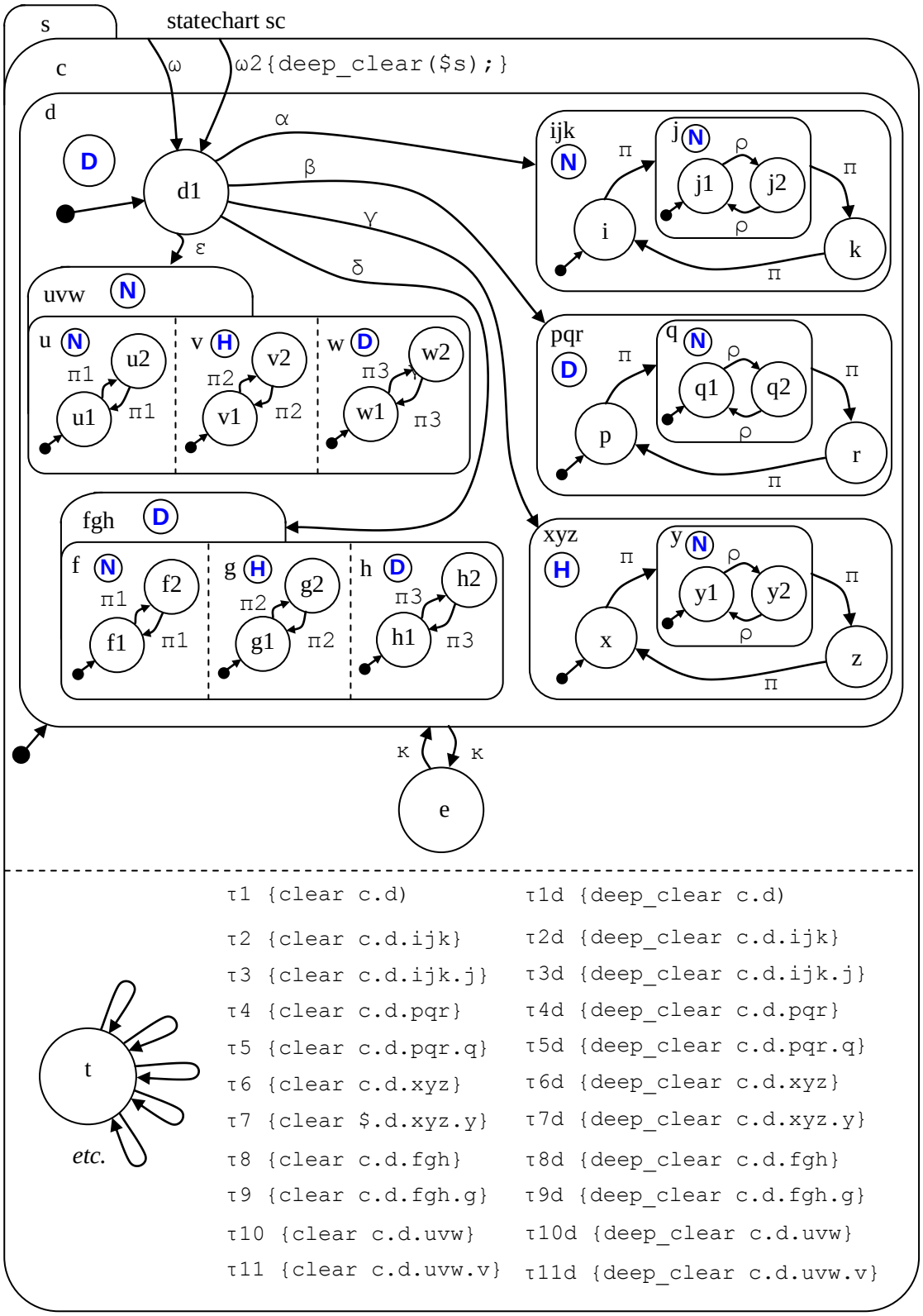


Figure 19. Arithmetic (with scoping) [model `t5210`]

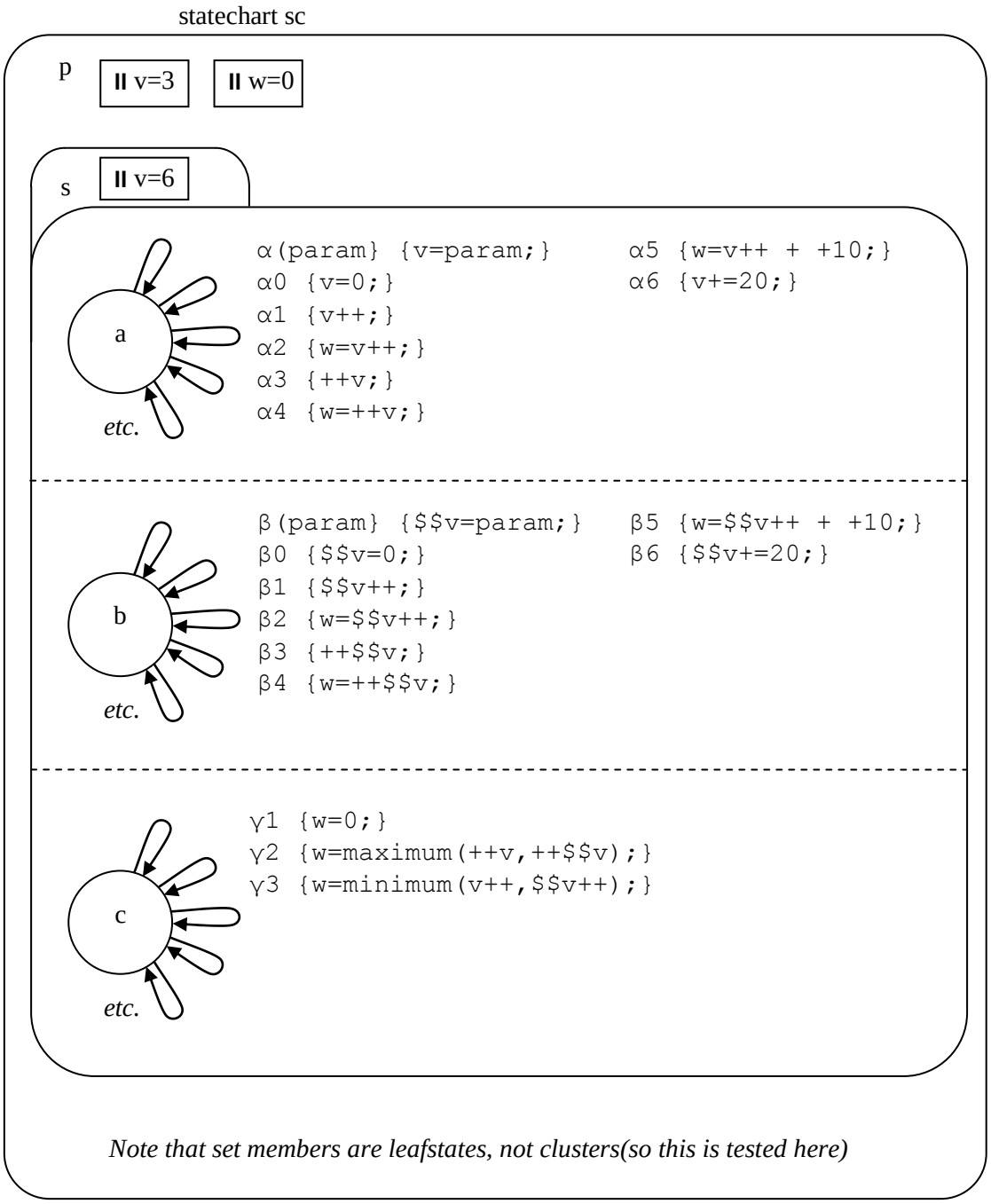


Figure 20. Strings and String Functions [model `t5220`]

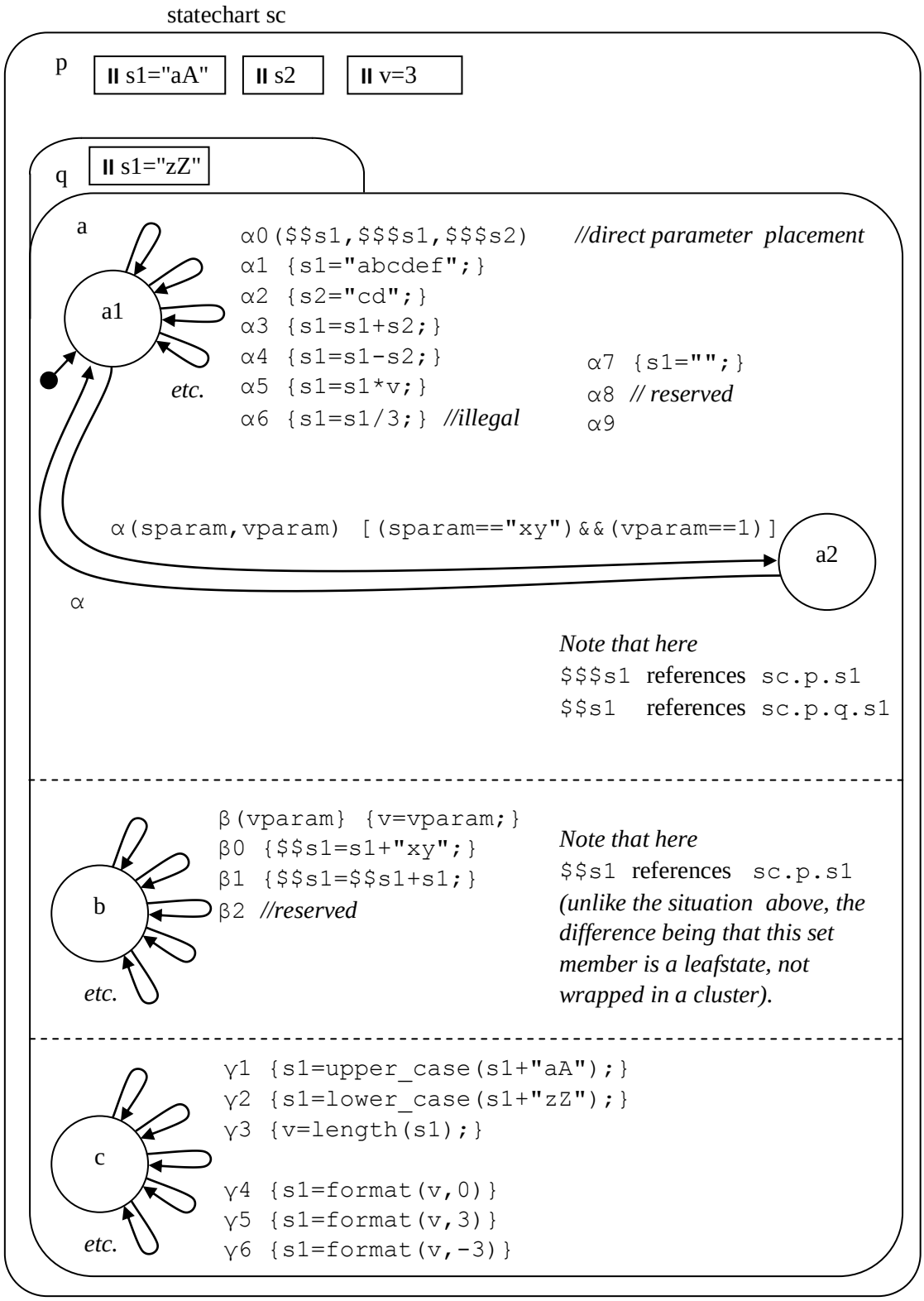


Figure 21. Traces (deterministic) [model `t5230`]

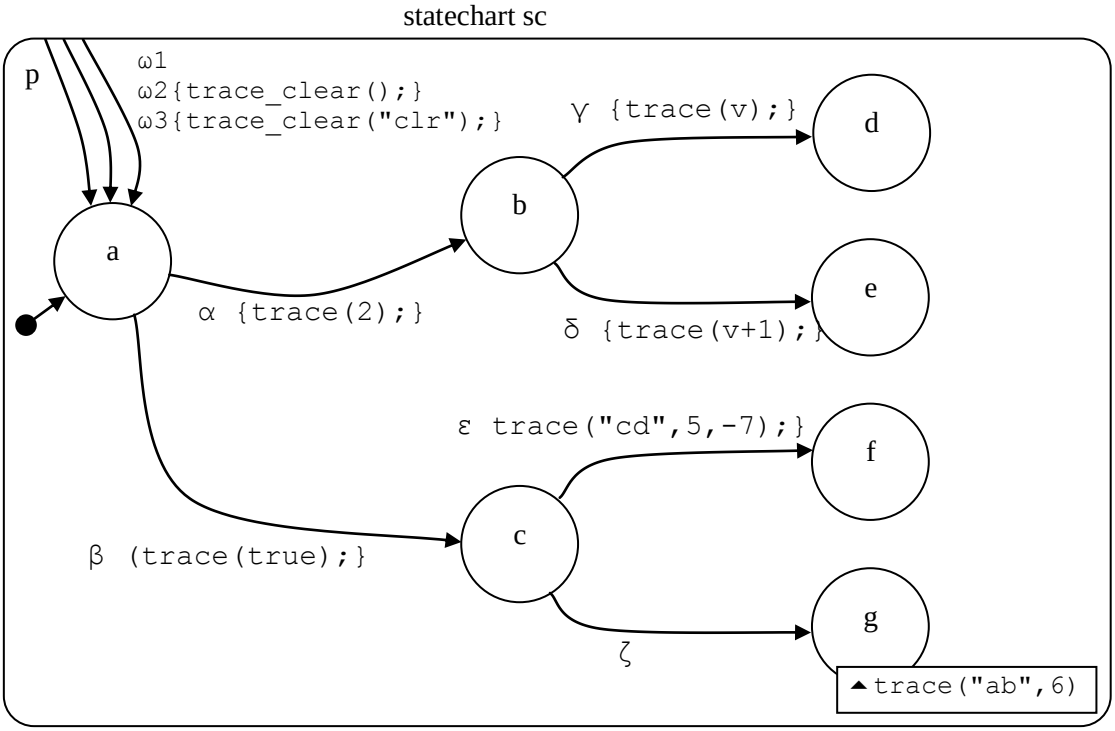


Figure 22. Cycling [model `t5240`]

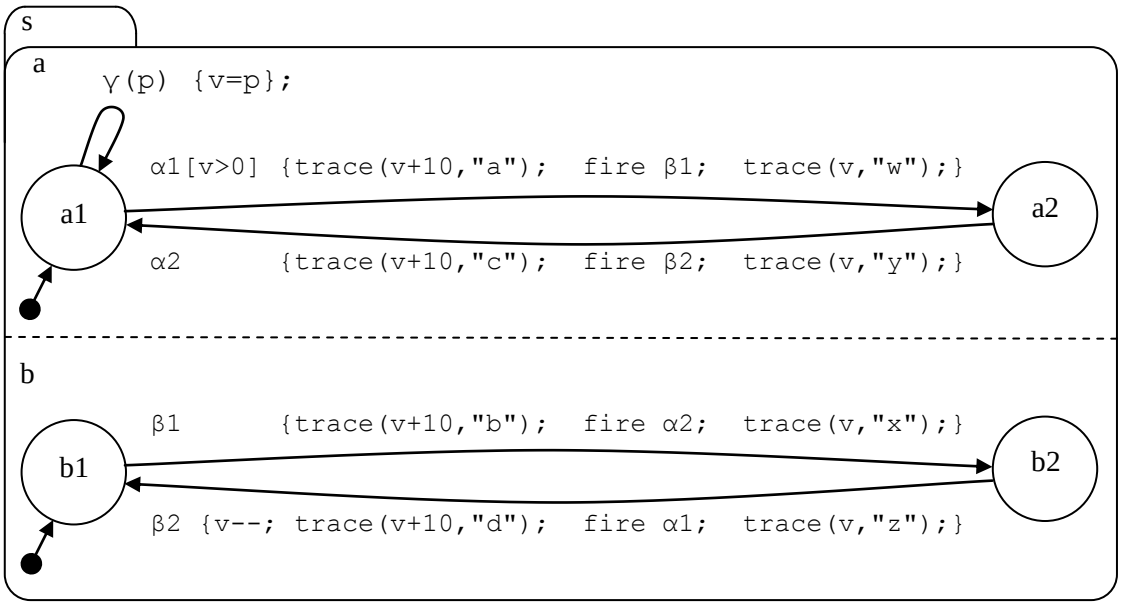
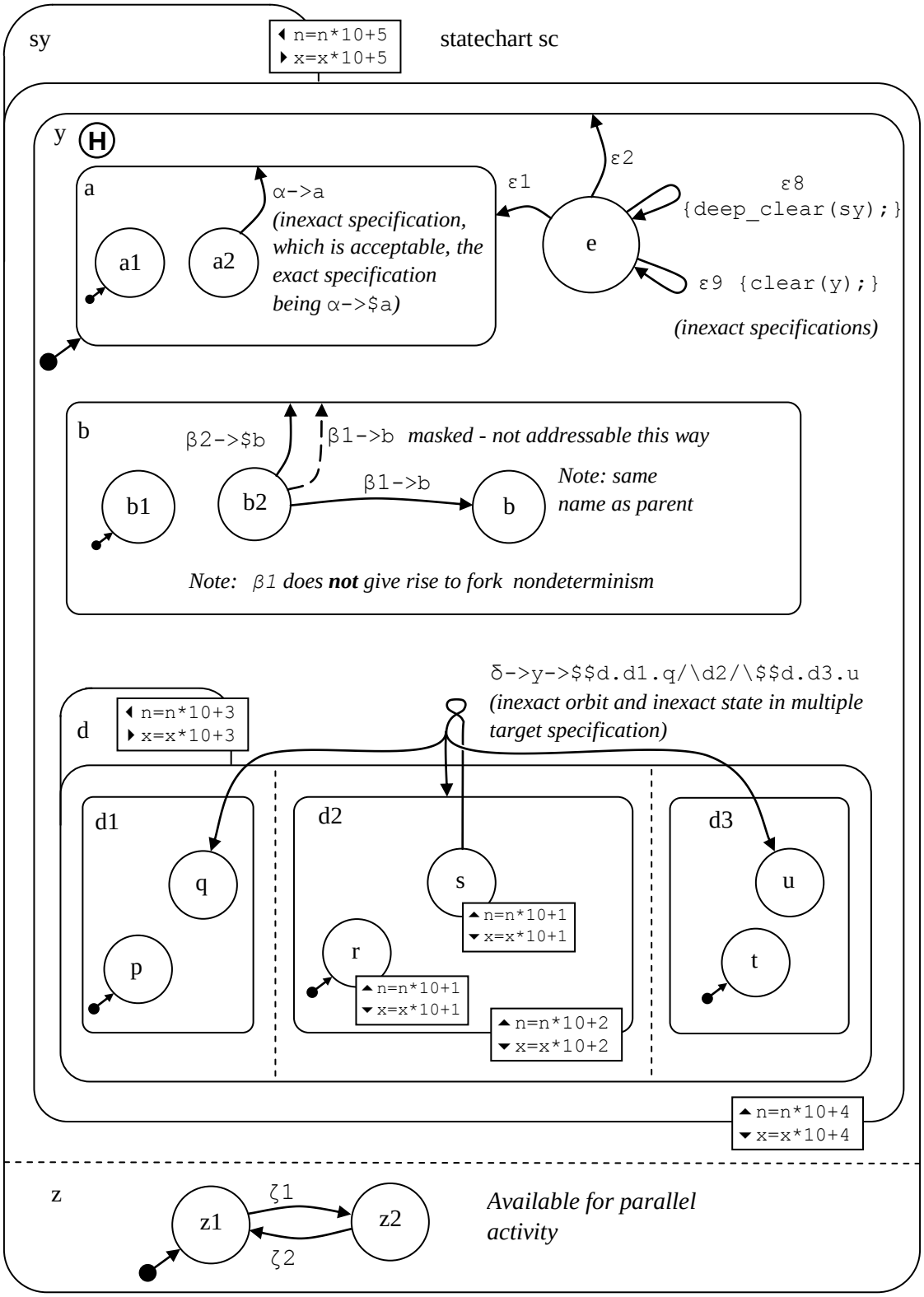


Figure 23. Inexact state scoping - [model `t5250`]



5.2 Small Nondeterministic Models

Figure 24. Set transit nondeterminism only [model `t5410`]

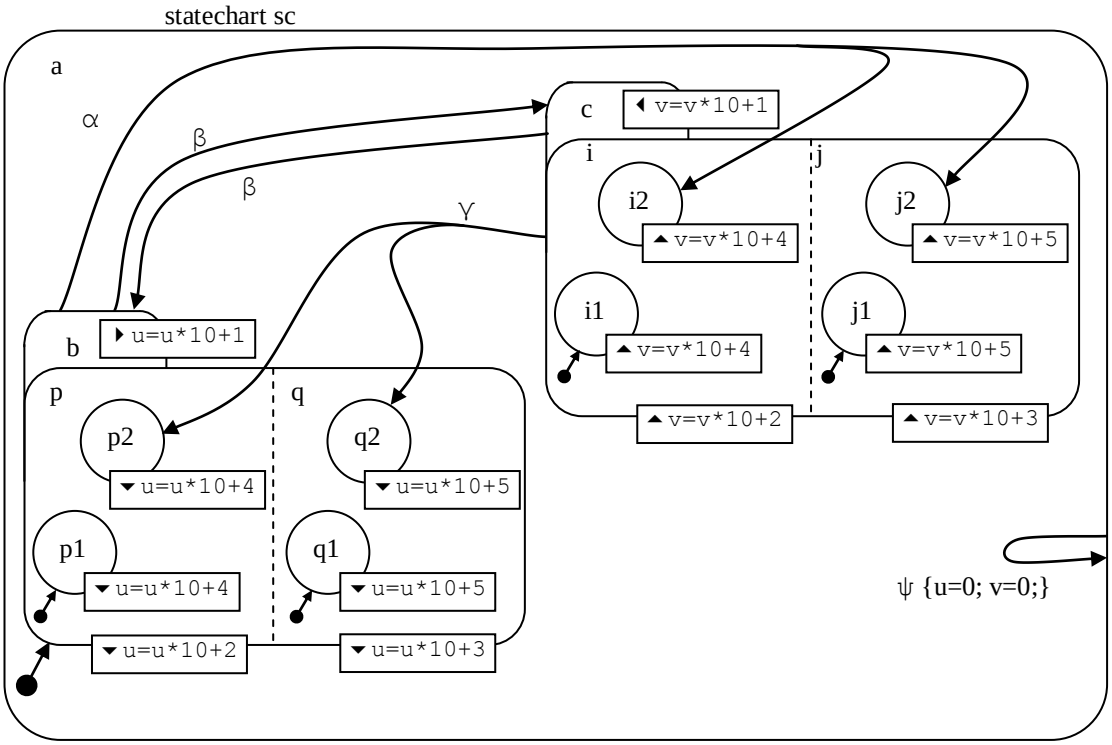
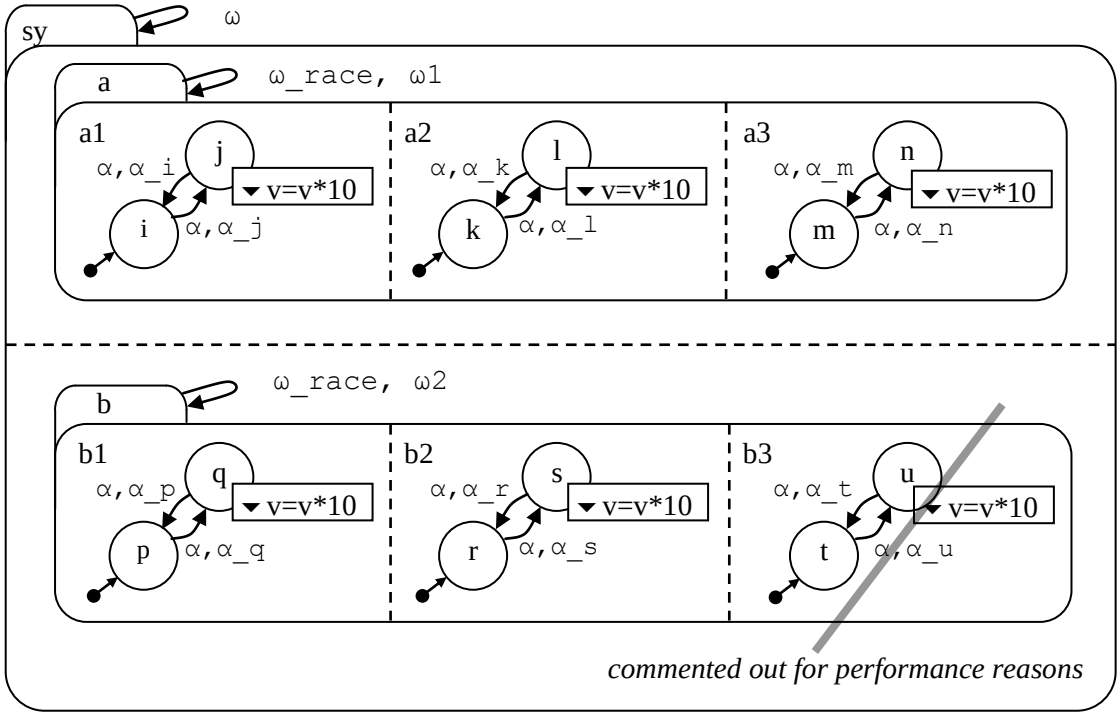


Figure 25. Set Action Nondeterminism [model `t5412`]



When, say, events `α_j`, `α_n`, and `α_s` are given, then `ω` is given, the actions that take place are treated in the same way as set-transit actions on member states.

Notes

- `α`, `α` gives rise to ***race nondeterminism*** on a 5 way race, giving $Perm^{race}(5)$ worlds, i.e. ***10 worlds*** under the `med_set_tran` option. (See Figure 41 and the description following for more explanation about this). This option produces 2n of the n! permutations. This is still ***quite fast***.

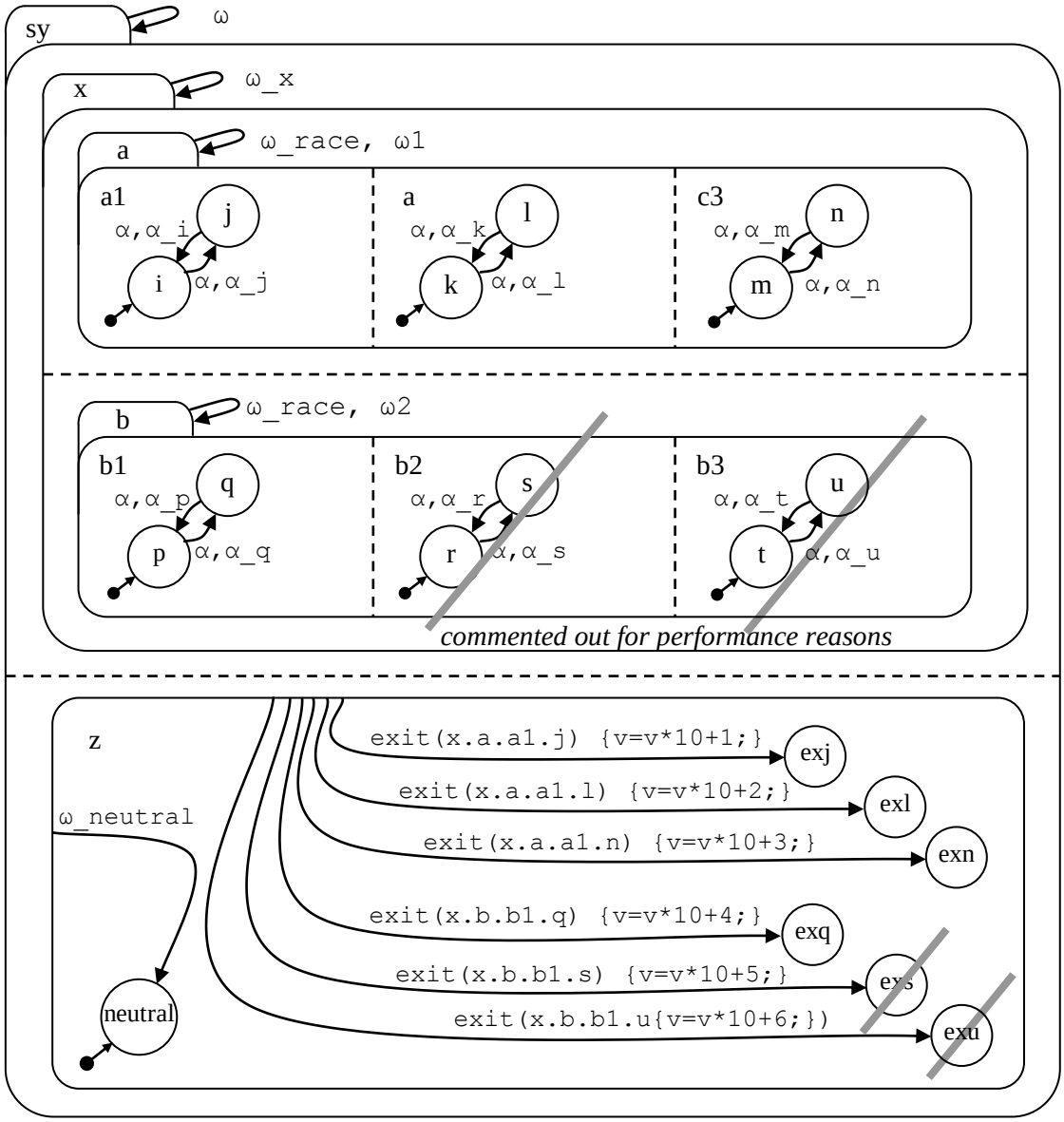
- `α`, `ω` gives rise to ***set-action nondeterminism***, causing permutations on (exit-j and exit-l and exit-n) and on (exit-q and exit-s), and between them, *as if set-transit nondeterminism were involved*, giving $Perm^{set\text{-}tran}(2).Perm^{set\text{-}tran}(3).Perm^{set\text{-}tran}(2)$ =***24 worlds***. ***This is slow***.

- `α`, `ω_race` gives rise to ***mixed race and set-action nondeterminism***, giving $Perm^{set\text{-}tran}(2).Perm^{set\text{-}tran}(3).Perm^{set\text{-}race}(2)$ =***24 worlds. The speed is medium.***

Note on speed

- By *medium*, we mean, typically, a matter of minutes on a 300 MHz machine
- By *slow*, we mean, typically, a matter of 30 mins-2 hours on a 300 MHz machine
- Speeds vary according to
  - the Prolog System
  - whether we run the model under the GP4 test harness or stand-alone
  - what has been run before (under the top-level Prolog prompt), since memory fragmentation (presumably) can degrade performance by one or more orders of magnitude.

Figure 26. Set meta-event nondeterminism [model `t5414`]



Illustrative sequence: `α_j α_n α_s ω_x`, showing permutations of exit meta-events.

- Analogous comments regarding ***race nondeterminism*** versus ***set-meta-event nondeterminism*** apply to those of model t5412, under `med_set_tran` permutations:
  - `α,α` a 4-way race, $Perm^{race}(4)$=8 worlds, fast.
  - `α,ω_x` set-meta-event nondeterminism, $Perm^{set\text{-}tran}(1).Perm^{set\text{-}tran}(3).Perm^{set\text{-}tran}(2)$ =12 worlds, slow.
  - `α,ω_race` rise to mixed race and set-meta-event nondeterminism, giving $Perm^{set\text{-}tran}(1).Perm^{set\text{-}tran}(3).Perm^{set\text{-}race}(2)$ =12 worlds, medium speed.

Figure 27. Fork nondeterminism only [model `t5420`]

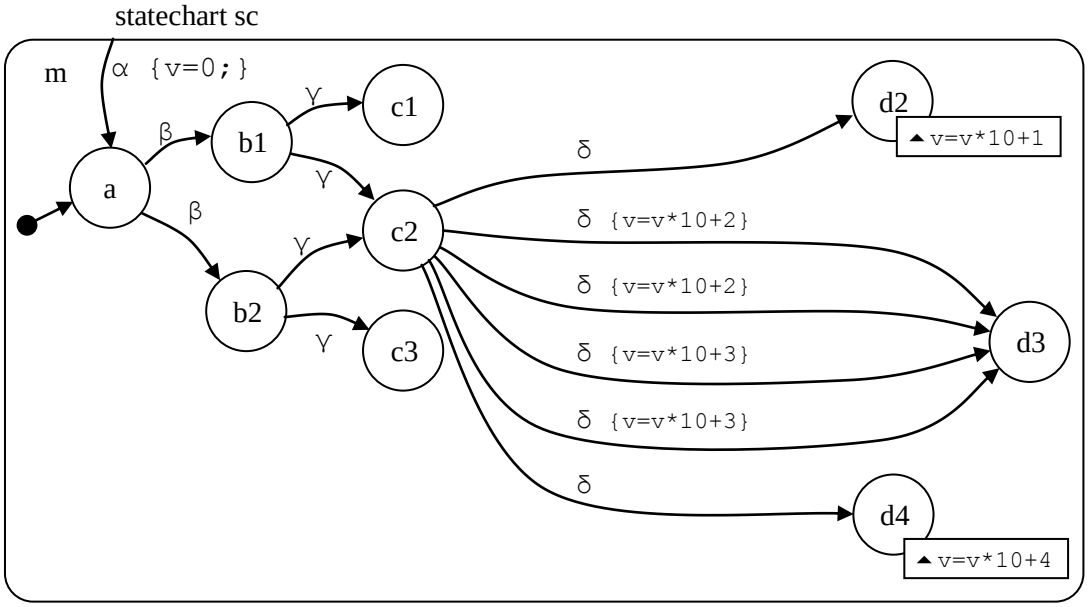
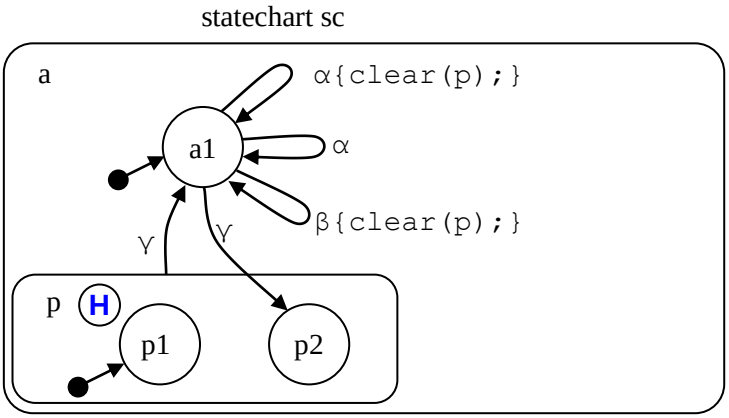


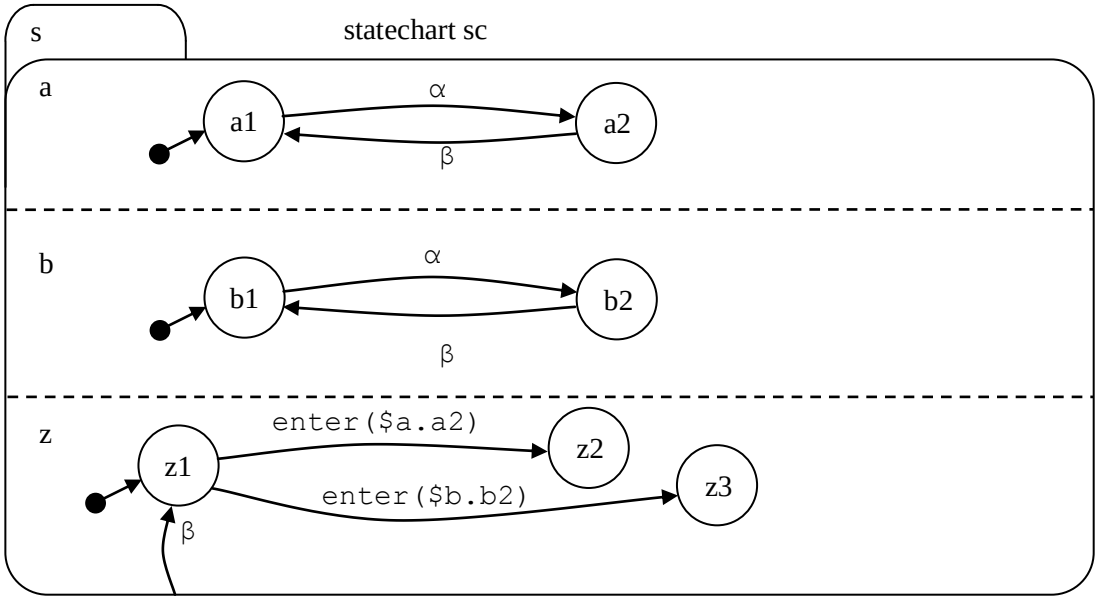
Figure 28. Fork Nondeterminism differentiated by history [model `t5422`]



To effectuate the nondeterminism, execute events as follows

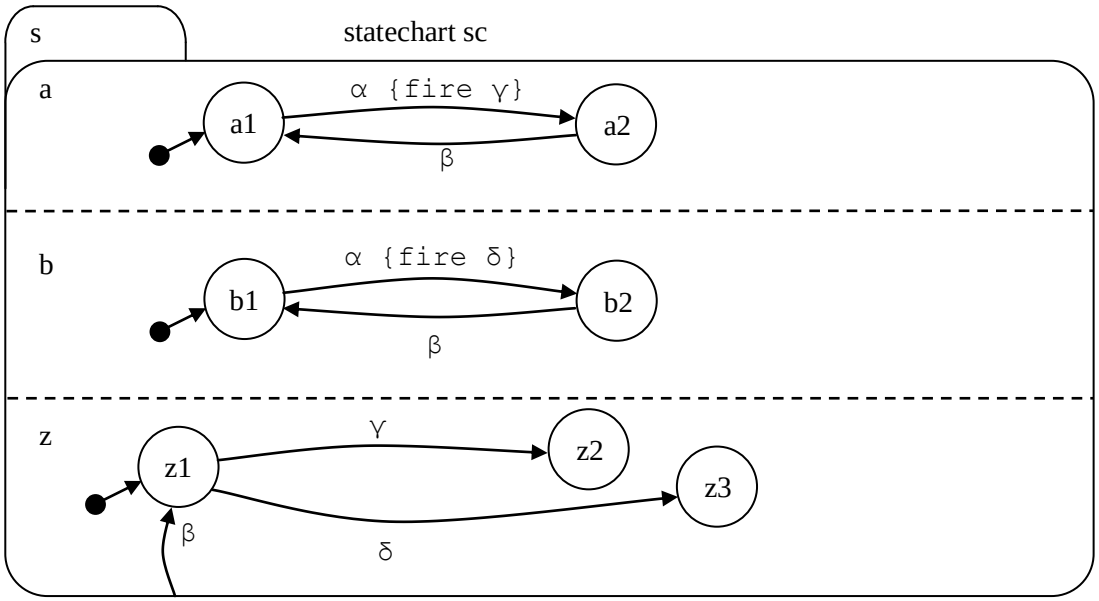
- event γ brings the machine to state `p2`
- event γ brings the machine back to `a1`, with history of cluster `p` recorded
- event α forks on existence of the record of history
- event β of worlds causes reconvergence of worlds by clearing all record of history of cluster p

Figure 29. Race nondeterminism only; winner detected by meta-event [model `t5430`]



note that β resets as a 3-way race, but with same result in each case

Figure 30. Race nondeterminism only - winner detected by fired event [model `t5440`]



note that β resets as a 3-way race, but with same result in each case

Figure 31. Race nondeterminism only - winner detected by variable value [model `t5450`]

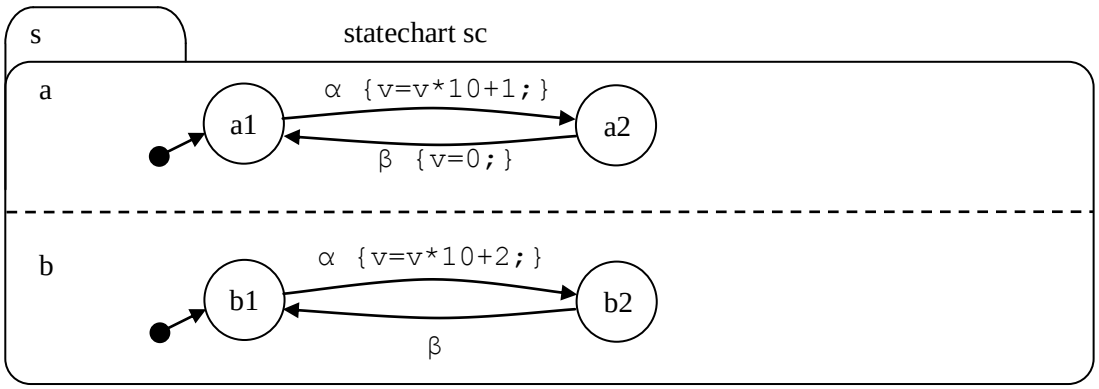
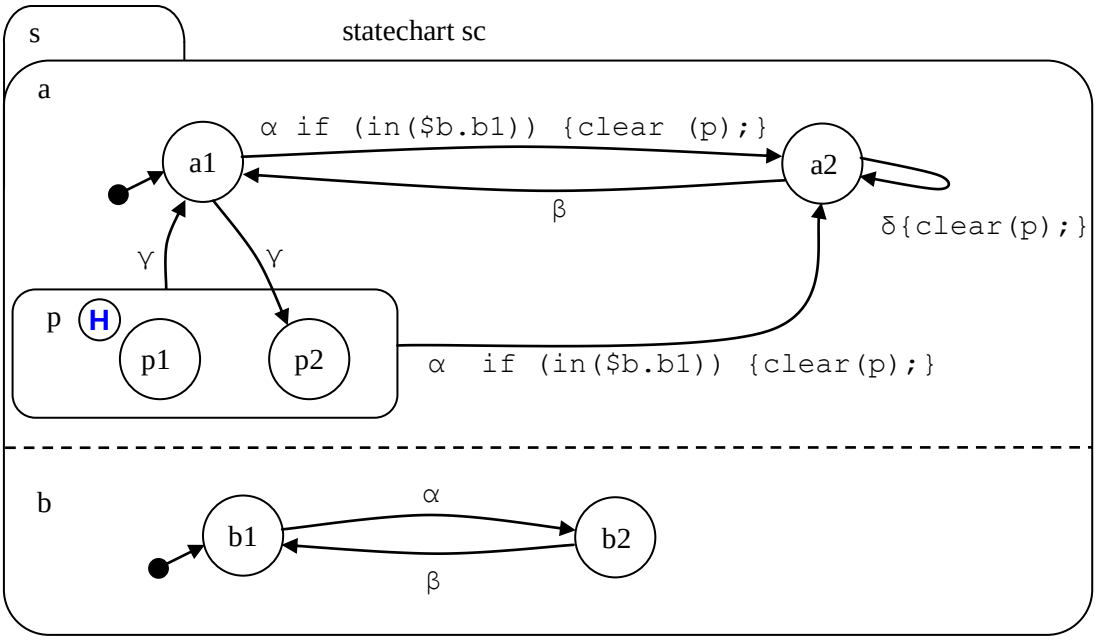


Figure 32. Race nondeterminism - winner detected by history [model `t5460`]



- For a simpler illustration of history in nondeterminism, as a case of fork nondeterminism, see model `t5422`.
- To run the race, process events `gamma`, `gamma`, `alpha`. In one arm of the race, the history of cluster `p` is cleared, in the other it is not cleared (because `b1` is vacant and the conditional action to clear history does not take place).
- Alternatively, events `gamma`, `alpha` are processed. A similar race takes place. In this case history is set on one of the transitions involved in the race, (as opposed to the previous case where history was set up before the race).

Figure 33. Race nondeterminism - winner detected by trace [model `t5470`]

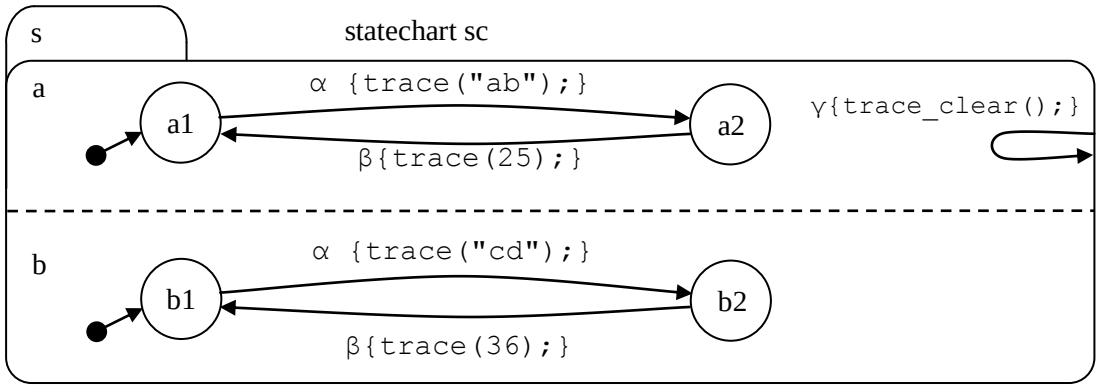


Figure 34. Race to a single target [model `t5472`]

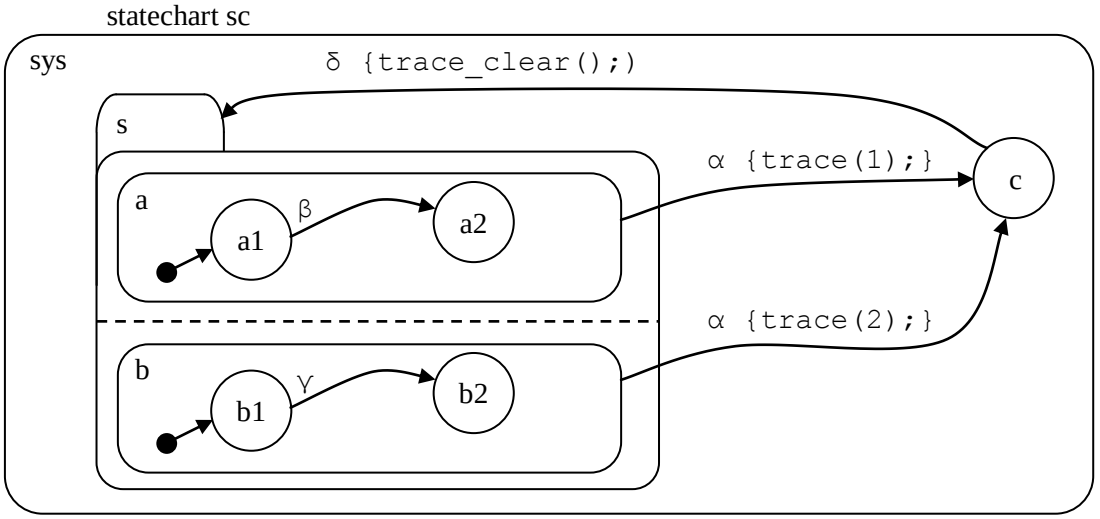


Figure 35. Race to start (mutually exclusive transitions) [model `t5474`]

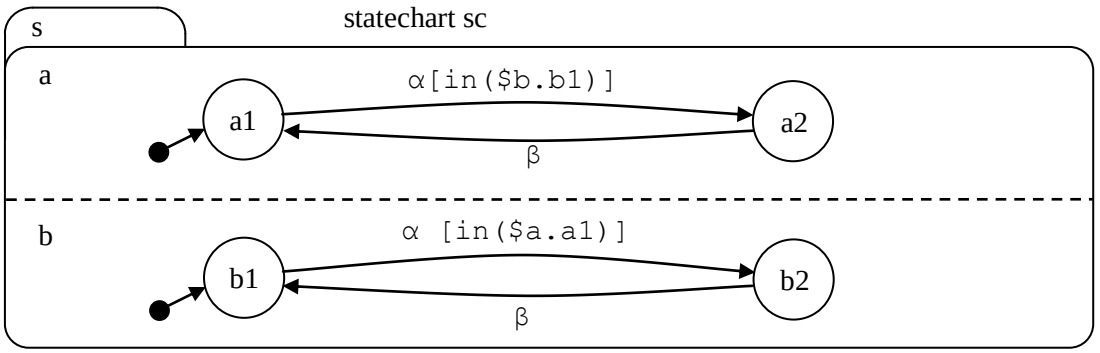
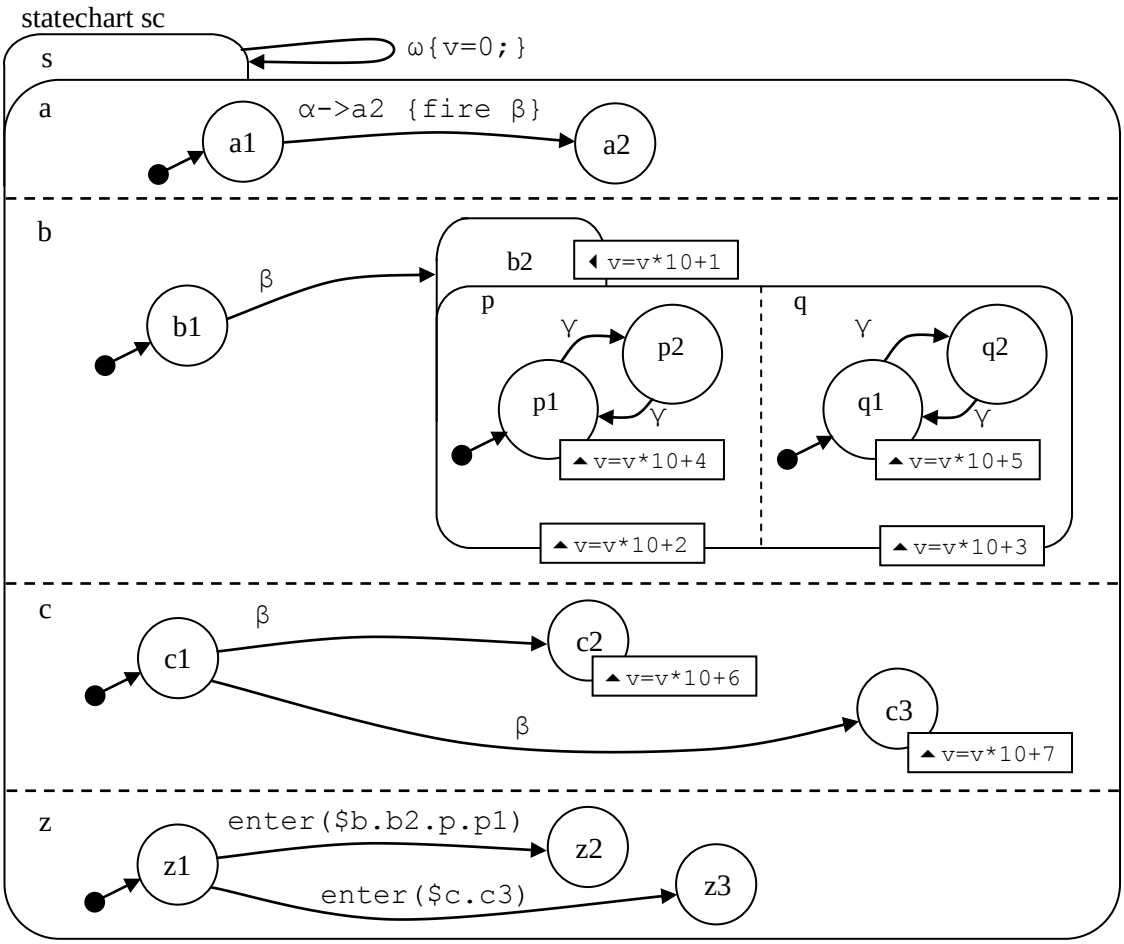
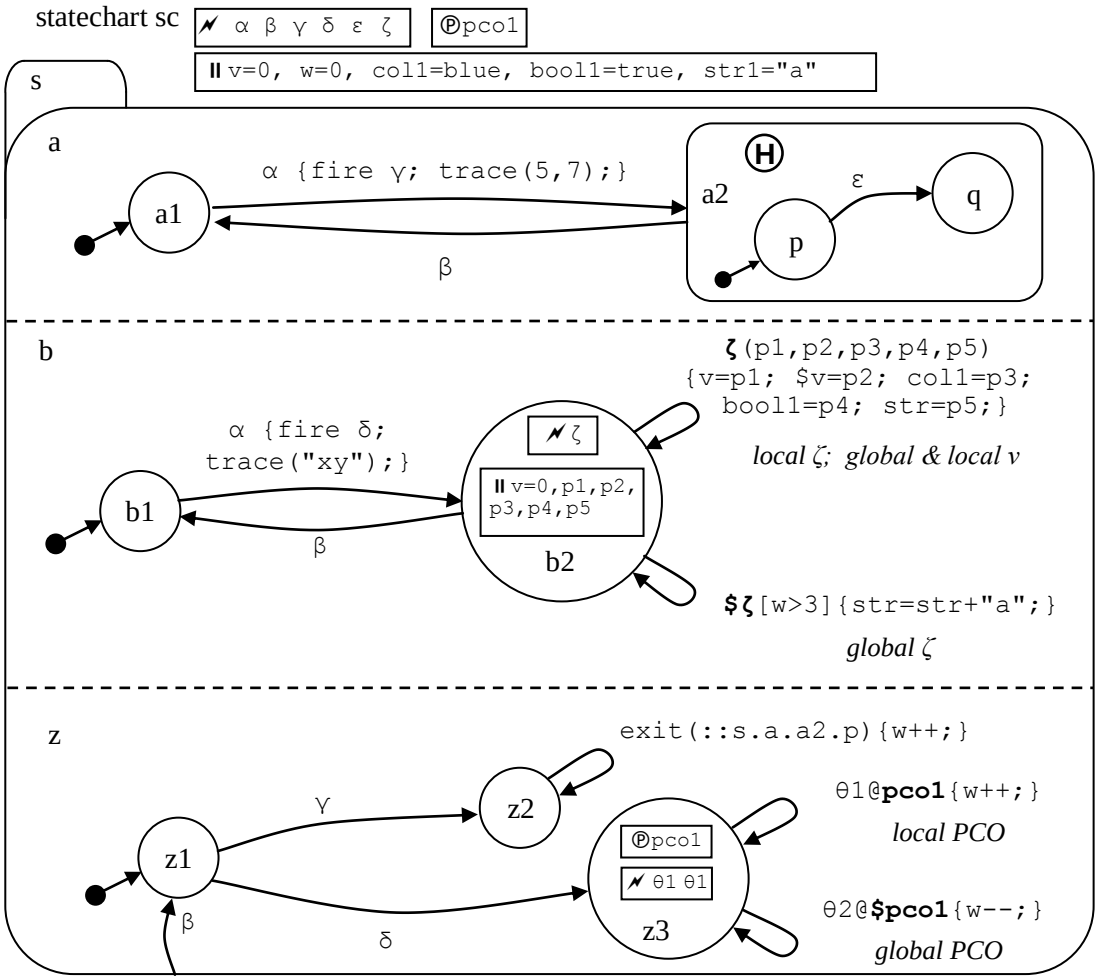


Figure 36. Compact multiple nondeterminism (4 kinds) [model `t5480`]



This model can be used with event β to illustrate set-transit, fork, and race-condition nondeterminism, or with event α to illustrate broadcast-event nondeterminism.

Figure 37. Illustration of all kinds of STATECRUNCHER output [model `t5490`]



Notes

- This model is basically a race on event α between fired events γ and δ, with the winner established by the order of processing fired events γ and δ in member `z` *and* by trace data deposited in members `a` and `b`.
- Scoped events **ζ** and **$ζ**
- Scoped variables **v** and **$v**
- Scoped PCOs **pco1** and **$pco1**
- Note how a nondefault cluster member (`q`) can be entered using event ε the first time and event α from state `a1` using history the second time.
- Note that internally generated events, in our example, `exit(::a.a2.p)` are *not* offered as user suppliable.

This model is used an example to illustrate output that would be used in communication with a primer. (A primer is a program that decides what tests to perform, i.e. what events to process, whereas STATECRUNCHER gives the oracle to these tests).

Figure 38. Transition Prioritization [model `t5500`]

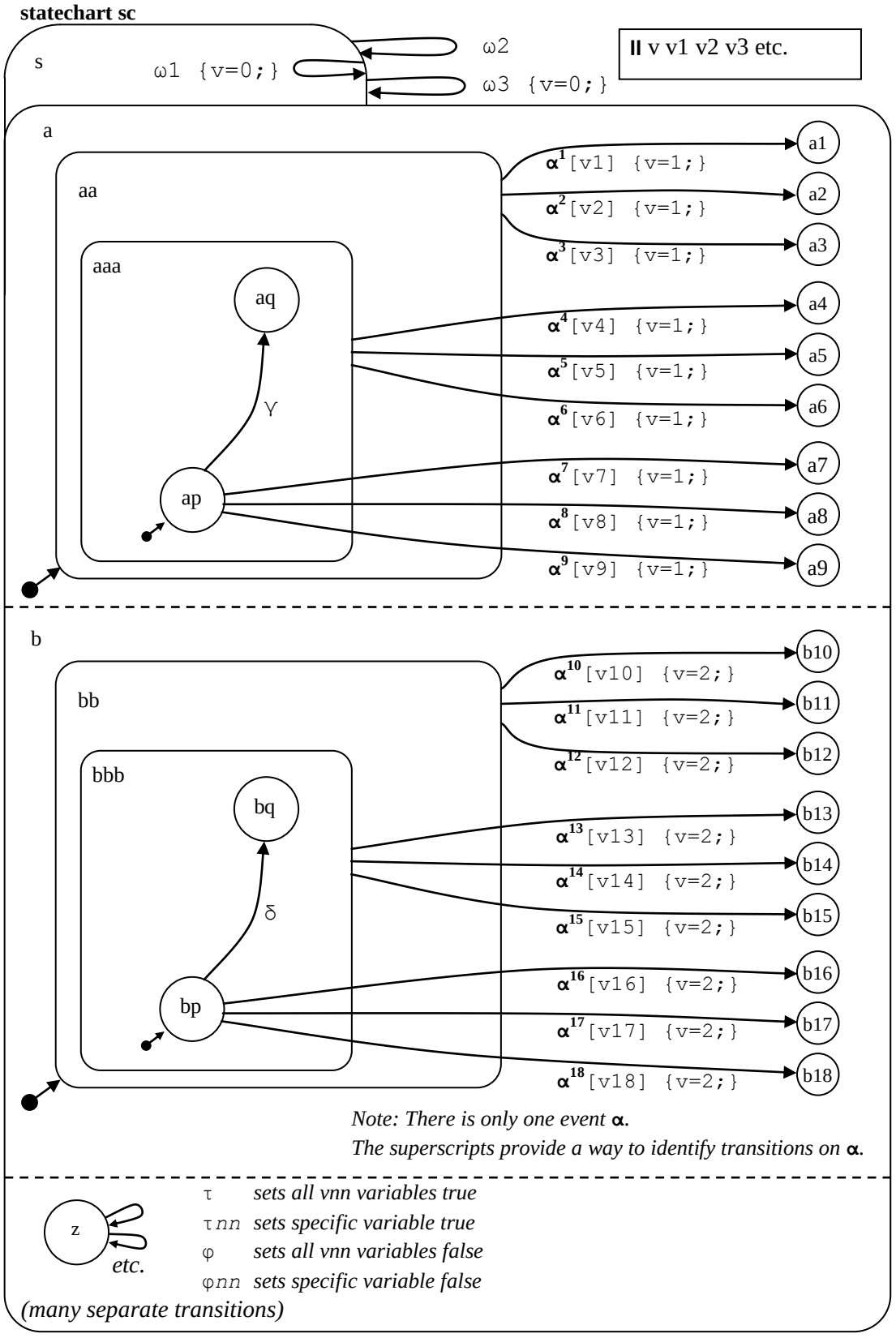


Figure 39. Scoped events illustrated by fork nondeterminism [model `t5510`]

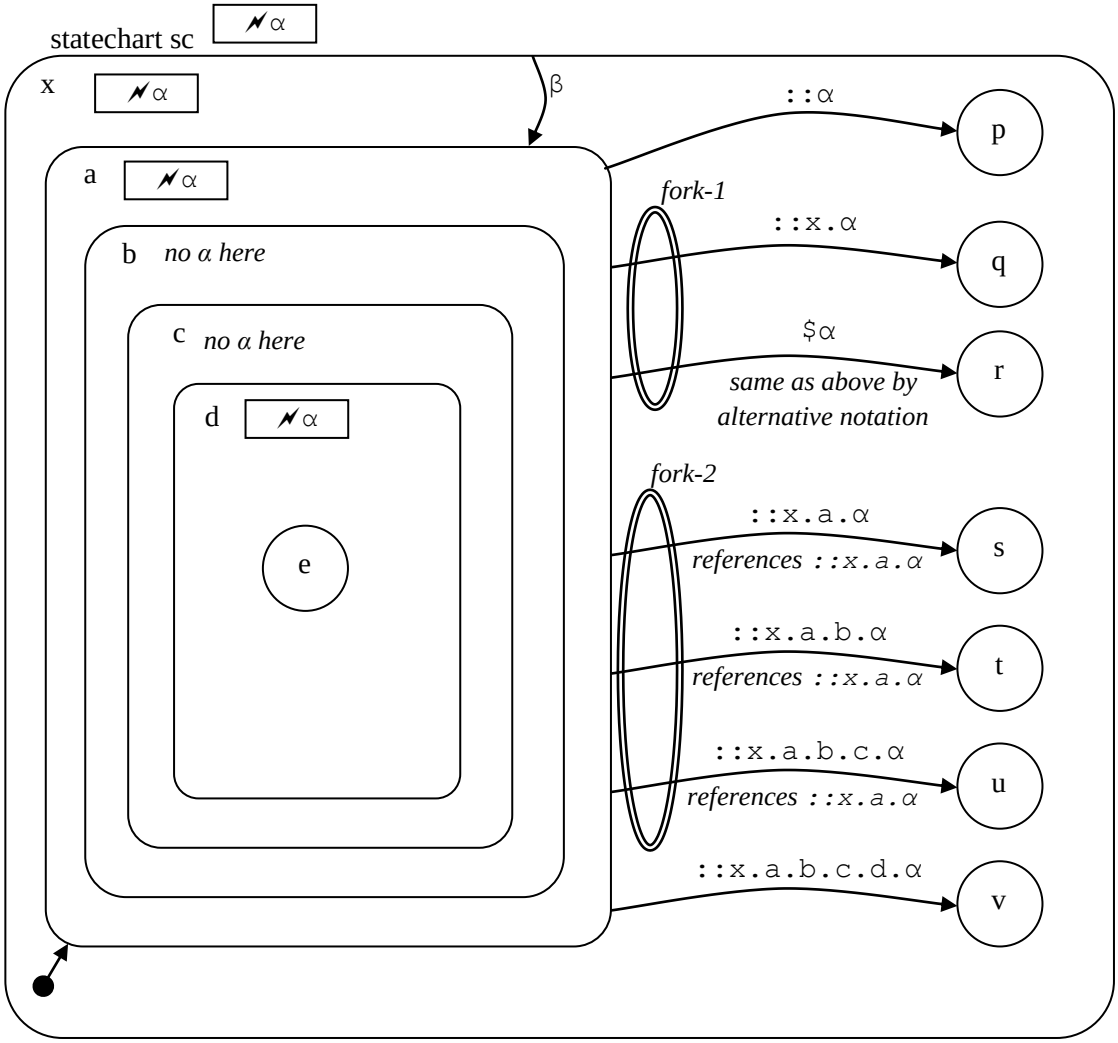
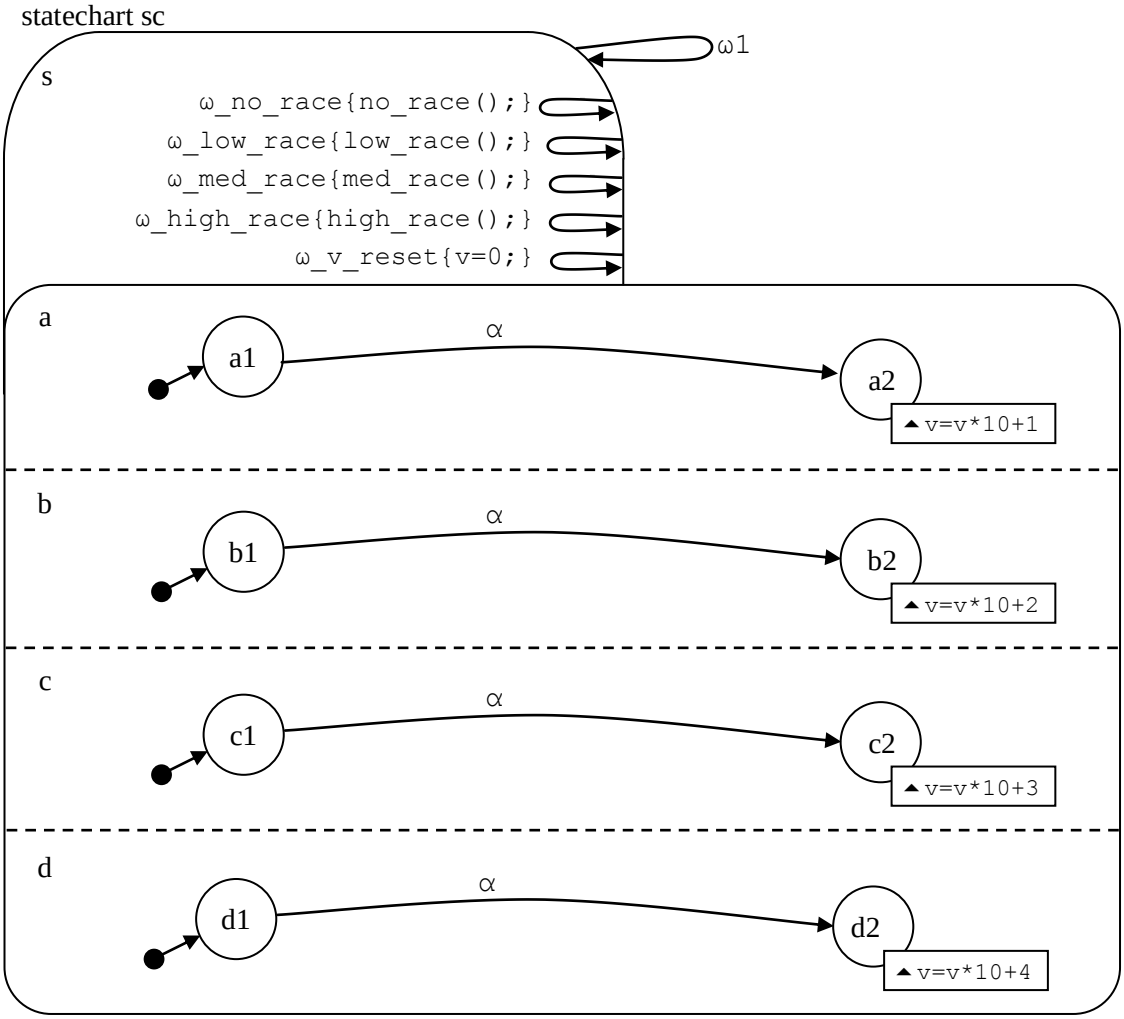


Figure 40. Limited permutation race nondeterminism [model `t5520`]

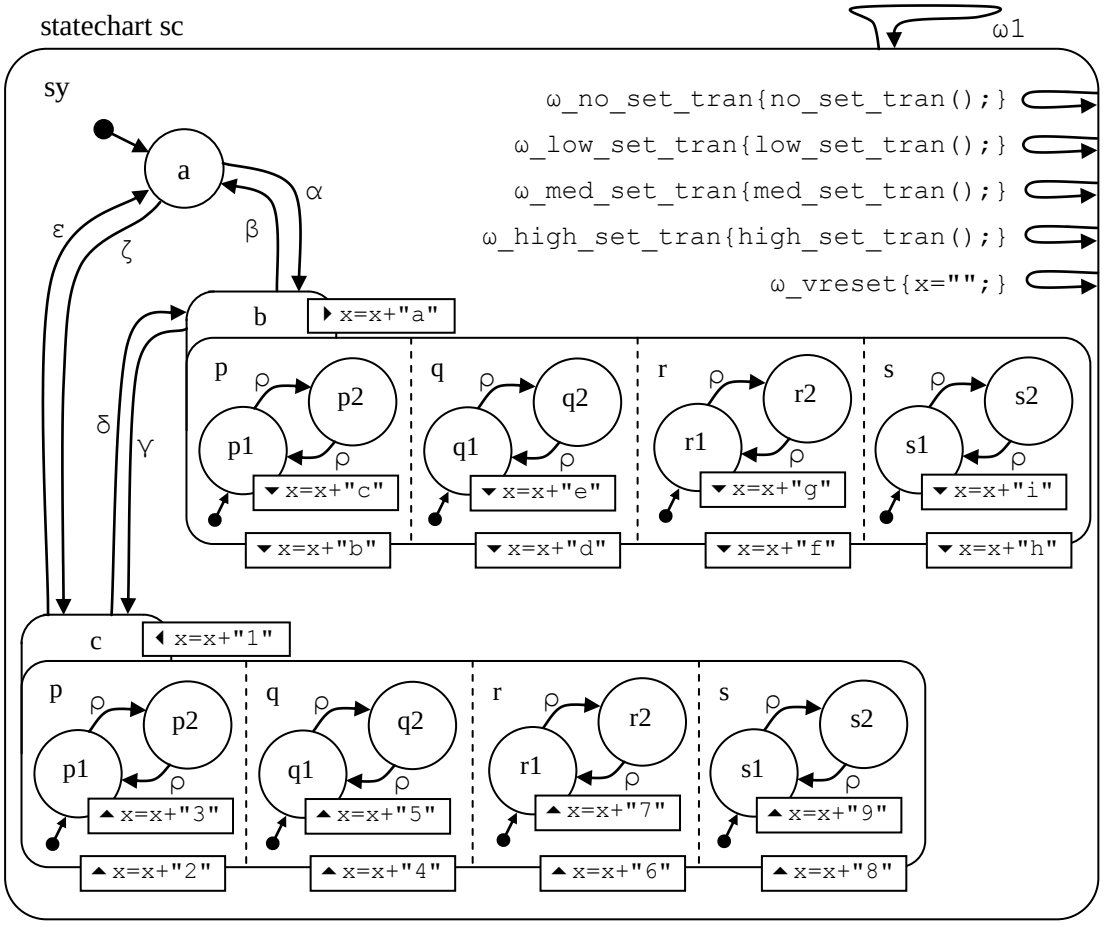


Explanation of the permutation limitations

- **`no_race`**: Only one permutation will be generated. The transition in the first set member will be executed first, then the one in the second set member etc. The permutation using set member names is `abcd`.
- **`low_race`**: Only two permutations will be generated. One is as above, and the other is the reverse of that order. The permutations are `abcd` and `dcba`.
- **`med_race`**: The number of permutations generated is 2n. These permutations are all the cyclic and anticyclic rotation operations on the no-race permutation. The permutations are (cyclic) `abcd`, `bcda`, `cdab`, `dabc`, (and anticyclic) `dcba`, `cbad`, `badc`, `adcb`.
- **`high_race`**: All n! permutations are generated, i.e. 4! = 24 permutations in this case.

These options can be set at a PROLOG prompt by the predicates `me_no_race`, `me_low_race`, `me_med_race` and `me_high_race`. The default is `me_med_race`.

Figure 41. Limited permutation set-transit nondeterminism [model `t5530`]

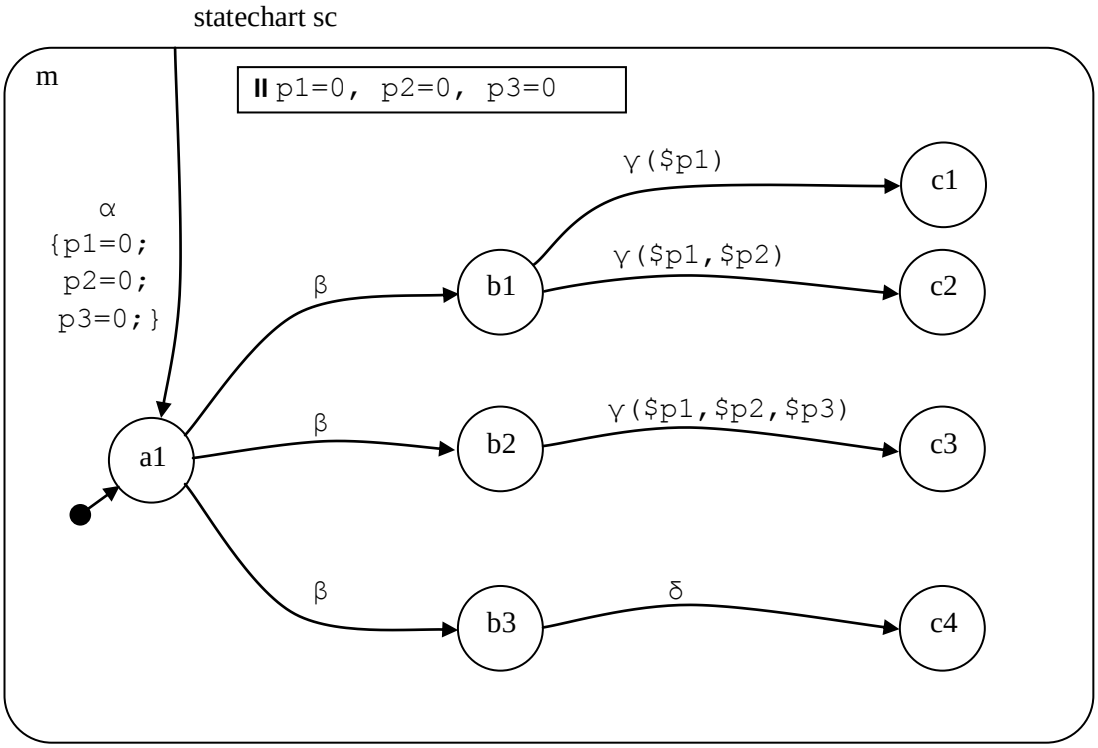


Explanation of the permutation limitations

- **`no_set_tran`**: Only one permutation will be generated. The transition in the first set member will be executed first, then the one in the second set member etc. The permutation using set member names is `pqrs`.
- **`low_set_tran`**: Only two permutations will be generated. One is as above, and the other is the reverse of that order. The permutations are `pqrs` and `srqp`.
- **`med_set_tran`**: The number of permutations generated is 2n. These permutations are all the cyclic and anticyclic rotation operations on the no-set_tran permutation. The permutations are (cyclic) `pqrs`, `qrsp`, `rspq`, `spqr`, (and anticyclic) `srqp`, `rqps`, `qpsr`, `psrq`.
- **`high_set_tran`**: All n! permutations are generated, i.e. 4! = 24 permutations in this case.

These options can be set at a PROLOG prompt by the predicates `me_no_set_tran`, `me_low_set_tran`, `me_med_set_tran` and `me_high_set_tran`. The default is `me_med_set_tran`.

Figure 42. Different transitionable events after nondeterminism [model `t5540`]



Pruning of traces - fork - non-self transitions [model `t5550`]

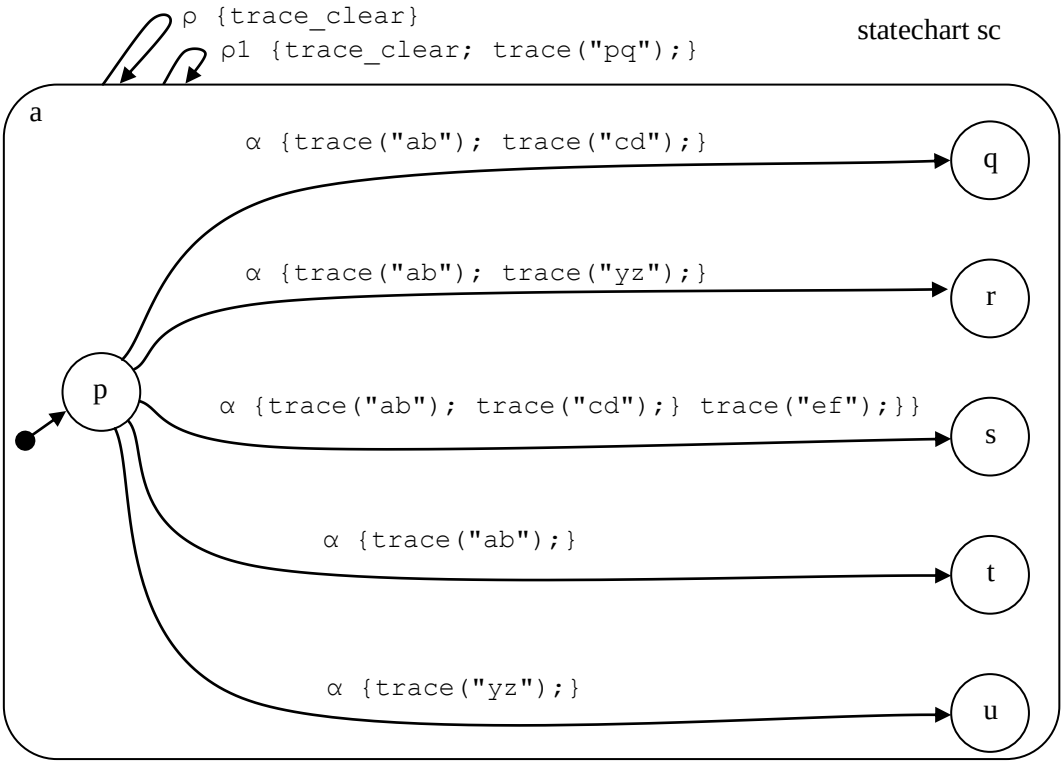
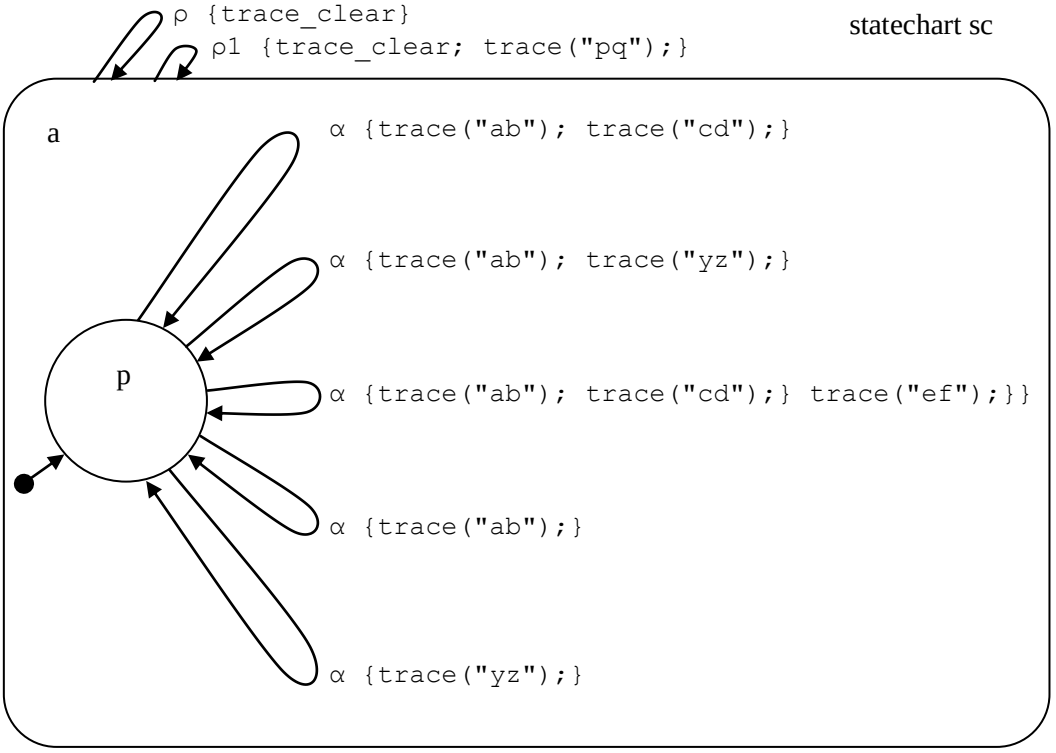


Figure 43. Pruning of traces - fork - self transitions [model `t5555`]



Pruning of traces - race - non-self transitions [model `t5560`]

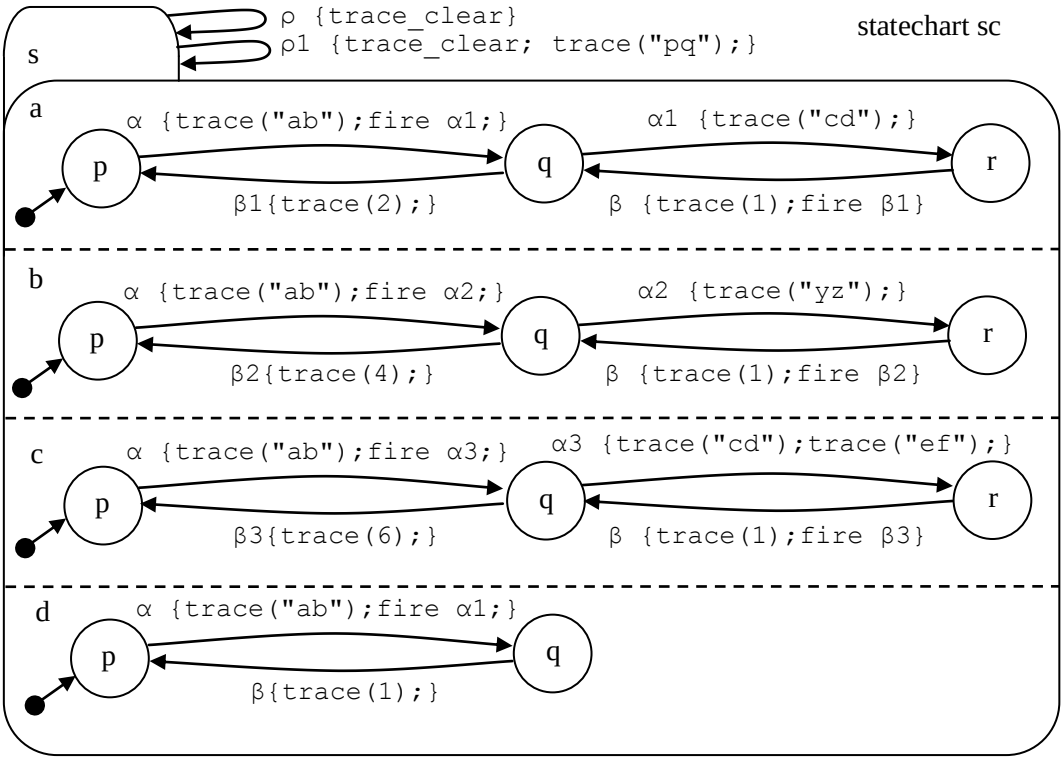


Figure 44. Pruning of traces - race - self transitions[model `t5565`]

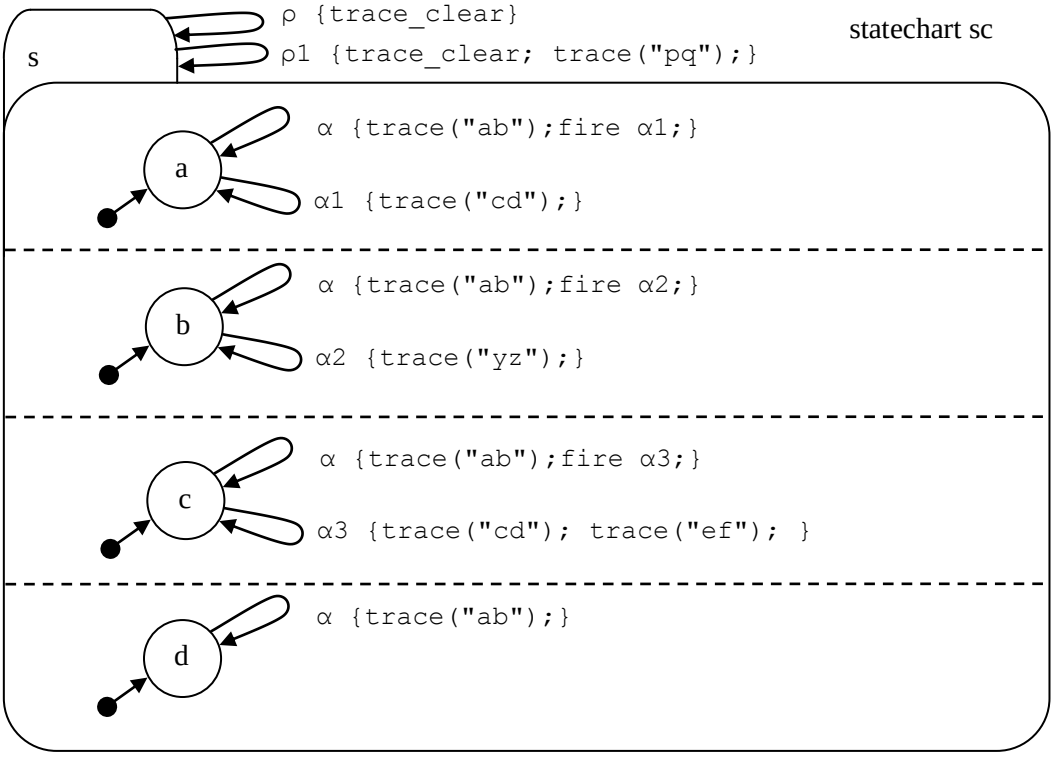
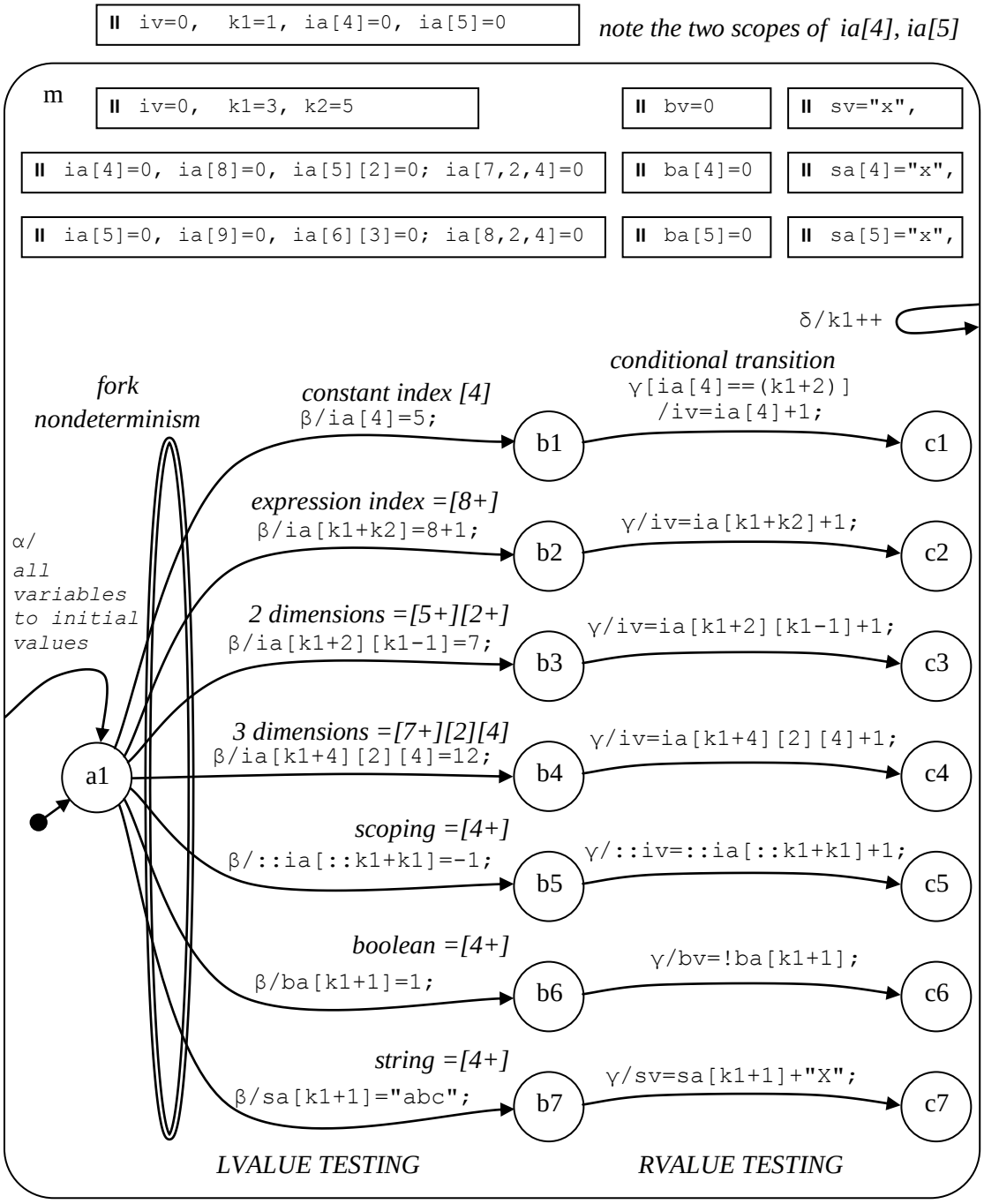


Figure 45. Arrays with fork nondeterminism [model `t5580`]



As at Release 1.04

- Array base (i.e. without index), and all array elements must be declared
- Undeclared array elements may work as regards internal logic, but will not be shown in output, nor be accepted as command input (as from primer).

Test sequence

- `events δ,β,γ,α.` Event δ increments local `k1`, and so some indices, marked by a +.

Figure 46. Simple scoped array [model `t5581`]

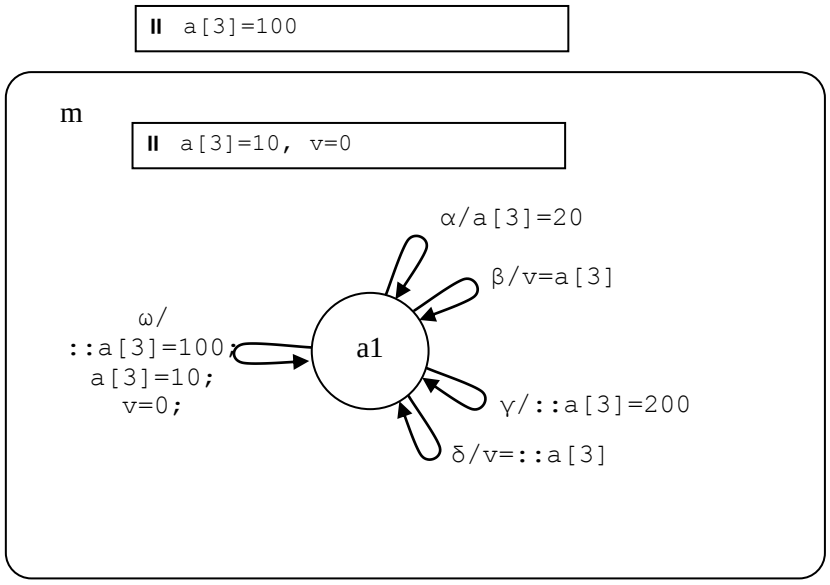
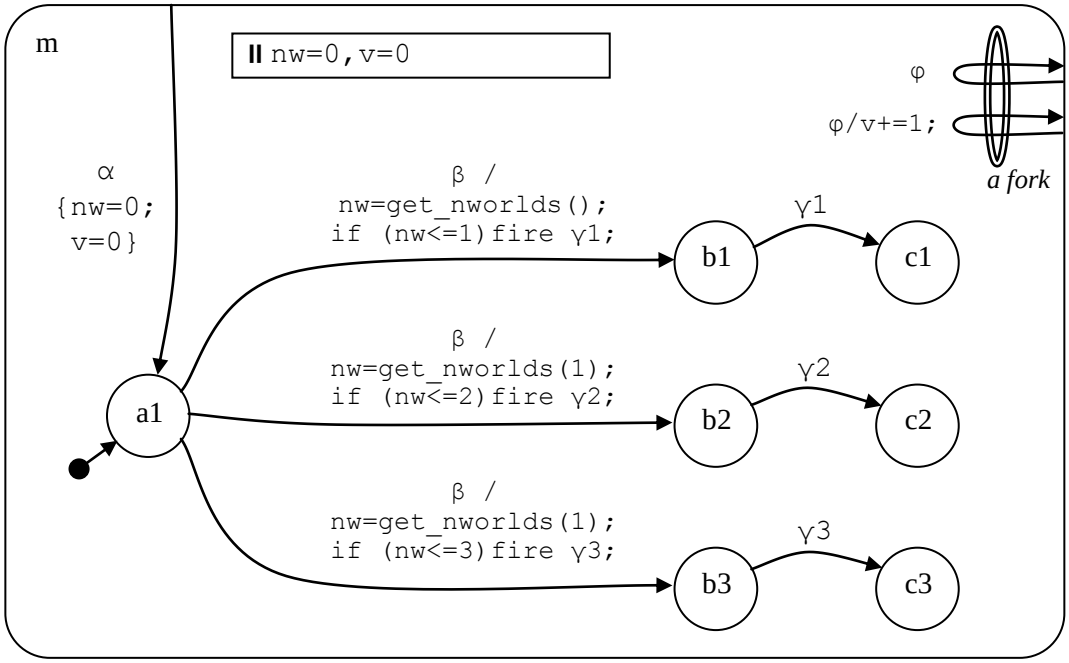
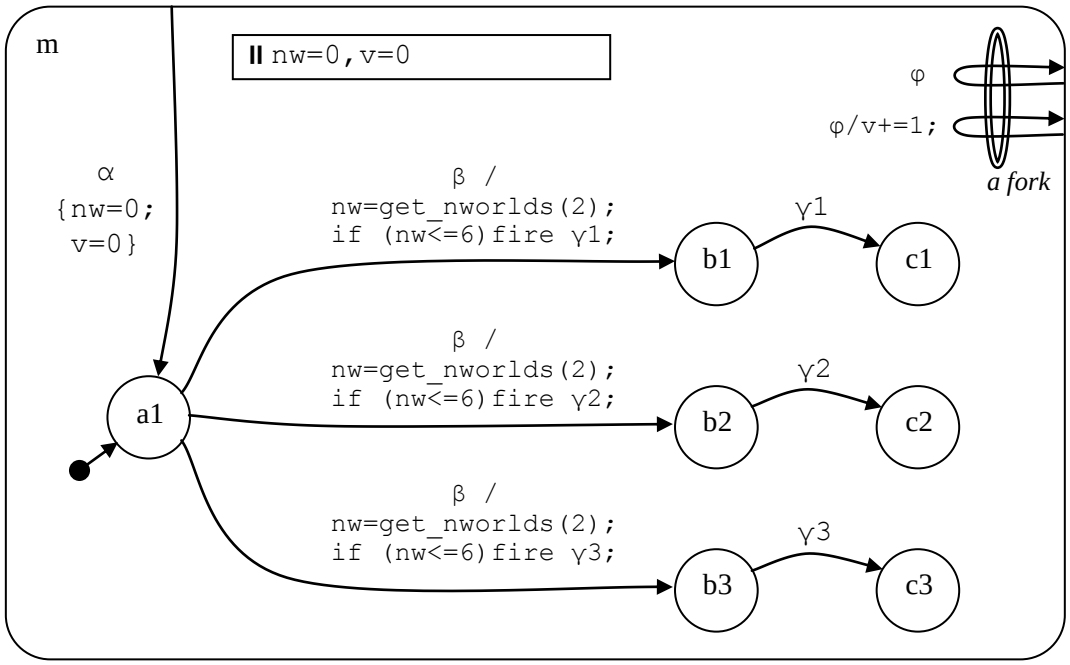


Figure 47. get_nworlds: Get number of worlds (1) [model `t5600`]



Parameter *P1* to get_nworlds: *P1*=1 (default) for command-time number-of-worlds
Illustrative event sequence: φ,β,α

Figure 48. get_nworlds: Get number of worlds (2) [model `t5602`]



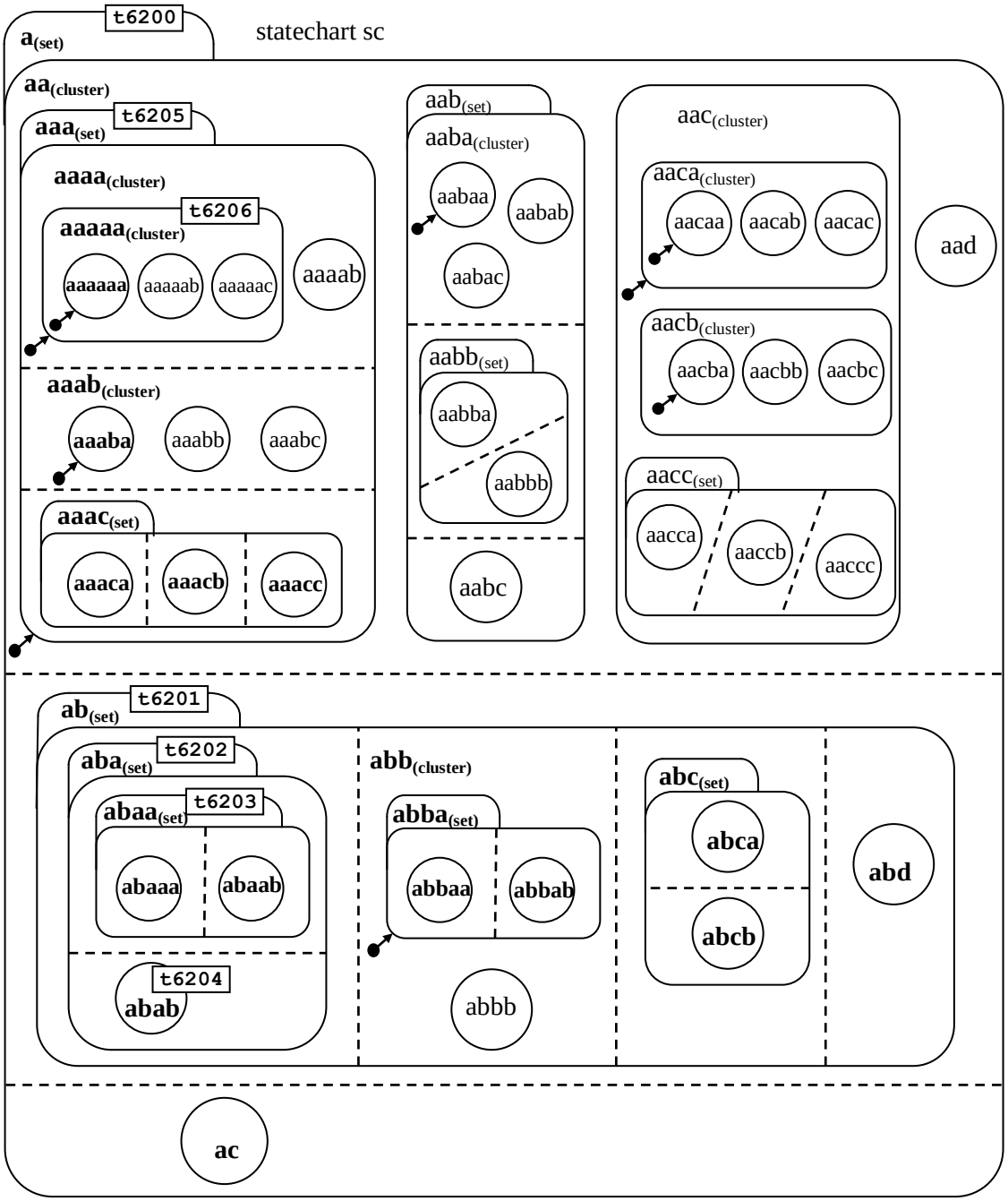
Parameter *P1* to get_nworlds *P1*=2 for execution-time number-of-worlds
This number may be higher than expected due to internal world generation on any action.
Illustrative event sequence: β,α

6. Systematic Test Models

Diagrams with their model numbers follow.

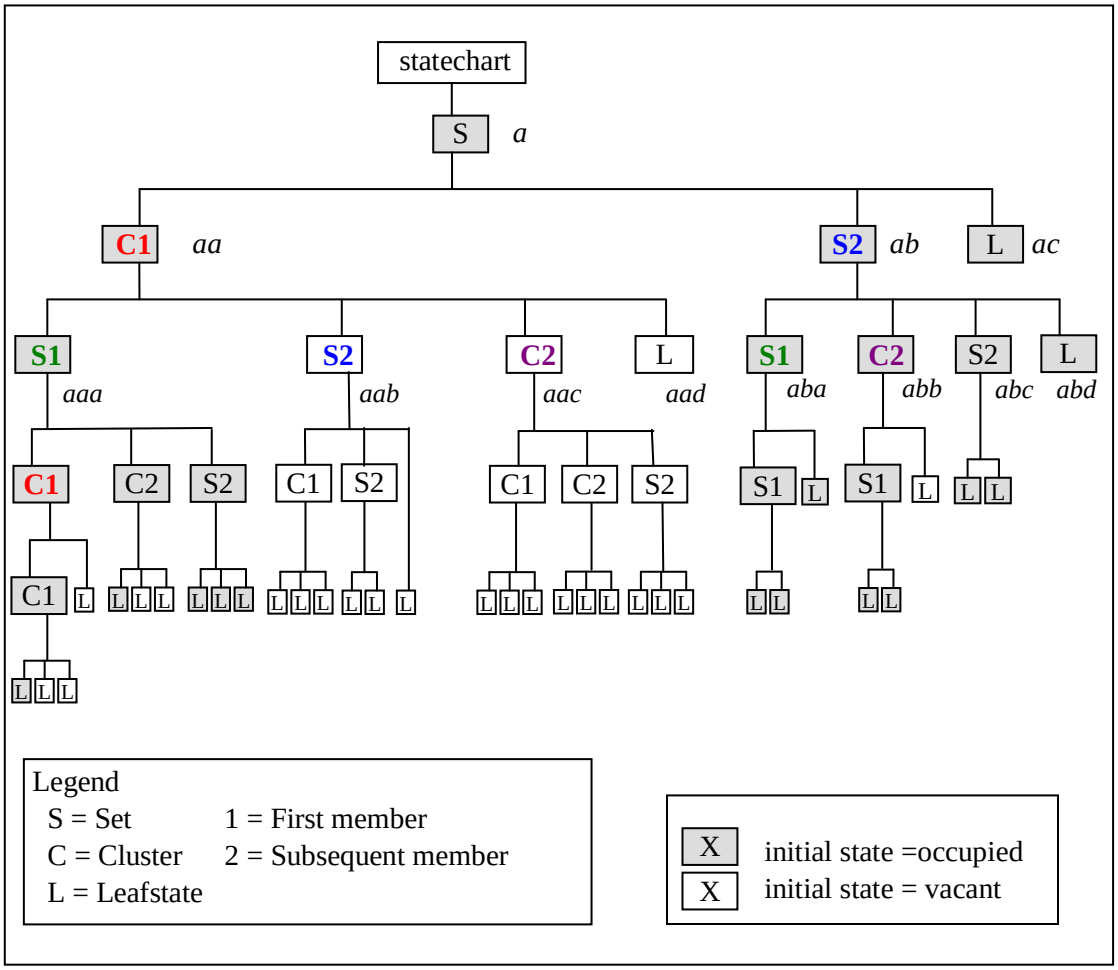
6.1 State Hierarchy and Initial Machine Entry

Figure 49. Hierarchy for initial/directed state entry [model `t6200` & derivatives]



Note: Model `t6200` contains all the above (8 sets in full exit from initial state). Other models contain just part of the full model as indicated, e.g. `t6201` contains just outer set **`ab`** from this model. (5 sets in all).

Figure 50. `t6200` structure



Counting any non-first member as a second member, the above hierarchy contains routes from the top

Set to set/cluster/leafstate

- S1-S1 S1-S2 S2-S1 S2-S2
- S1-C1 S1-C2 S2-C1 S2-C2
- S1-L1 S1-L2 S2-L1 S2-L2

Cluster to set/cluster/leafstate

- C1-C1 C1-C2 C2-S1 C2-S2
- C1-C1 C1-C2 C2-C1 C2-C2
- C1-L1 C1-L2 C2-L1 C2-L2

Occupied/Vacant combinations

- Set_{occ}-Cluster
- Set_{vac}-Cluster
- Set_{occ}-Set
- Set_{vac}-Set
- $Cluster_{occ}$-Cluster
- $Cluster_{vac}$-Cluster
- $Cluster_{occ}$-Set
- $Cluster_{vac}$-Set

The following sequences are also covered

- Set Cluster Set
- Cluster Set Cluster

These are the primary aspects being tested, in respect of "entering initial state".

6.2 Specifying States in Transitions

Figure 51. Specifying States (model `t6220`)

Notes: The notation shown does not include all delimitation (e.g. semicolons)
Exclamation marks on names are attention-drawing, not syntactical
Transitions are shown with explicit target state expressions
Default states are not shown in this diagram

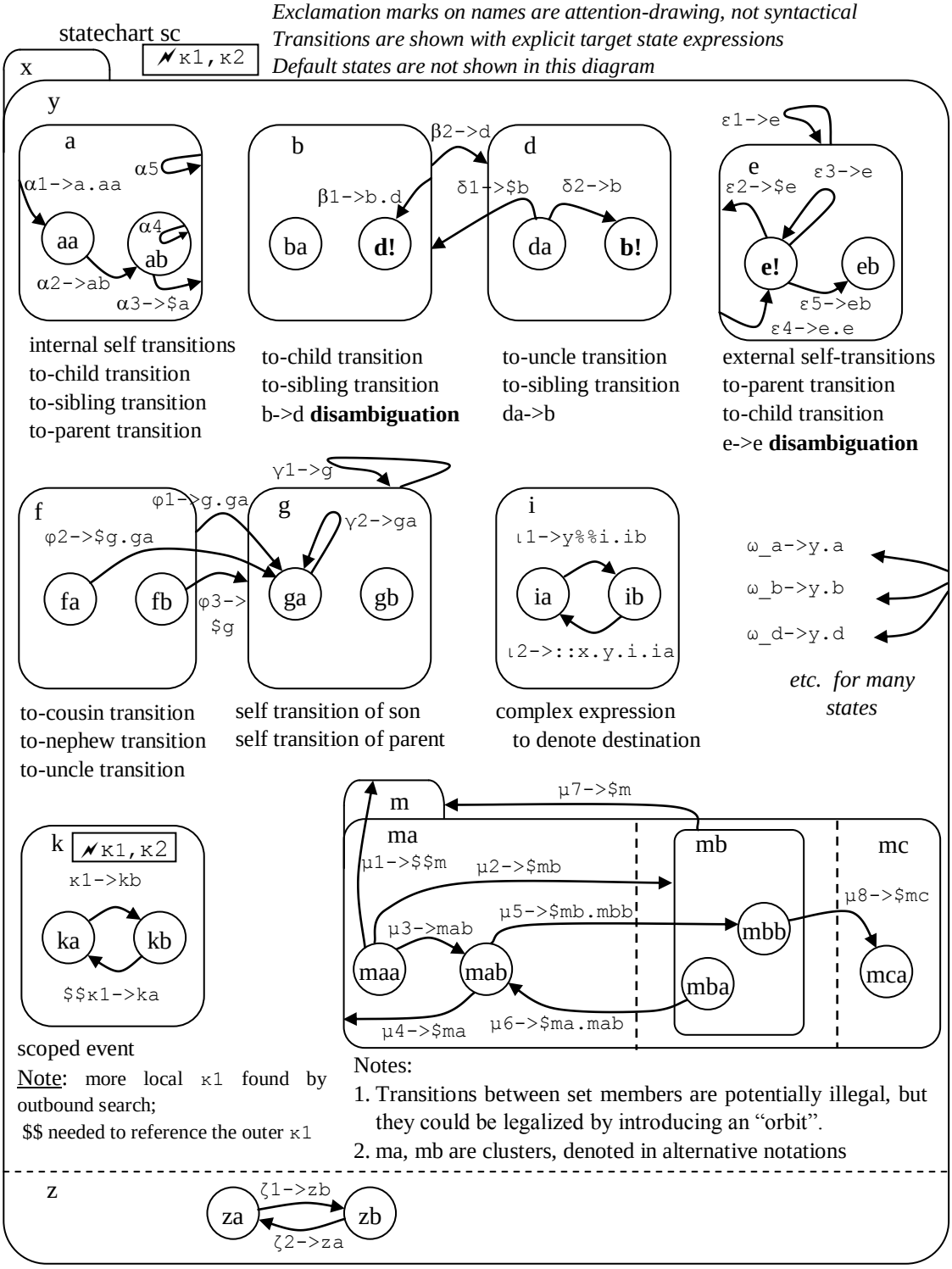
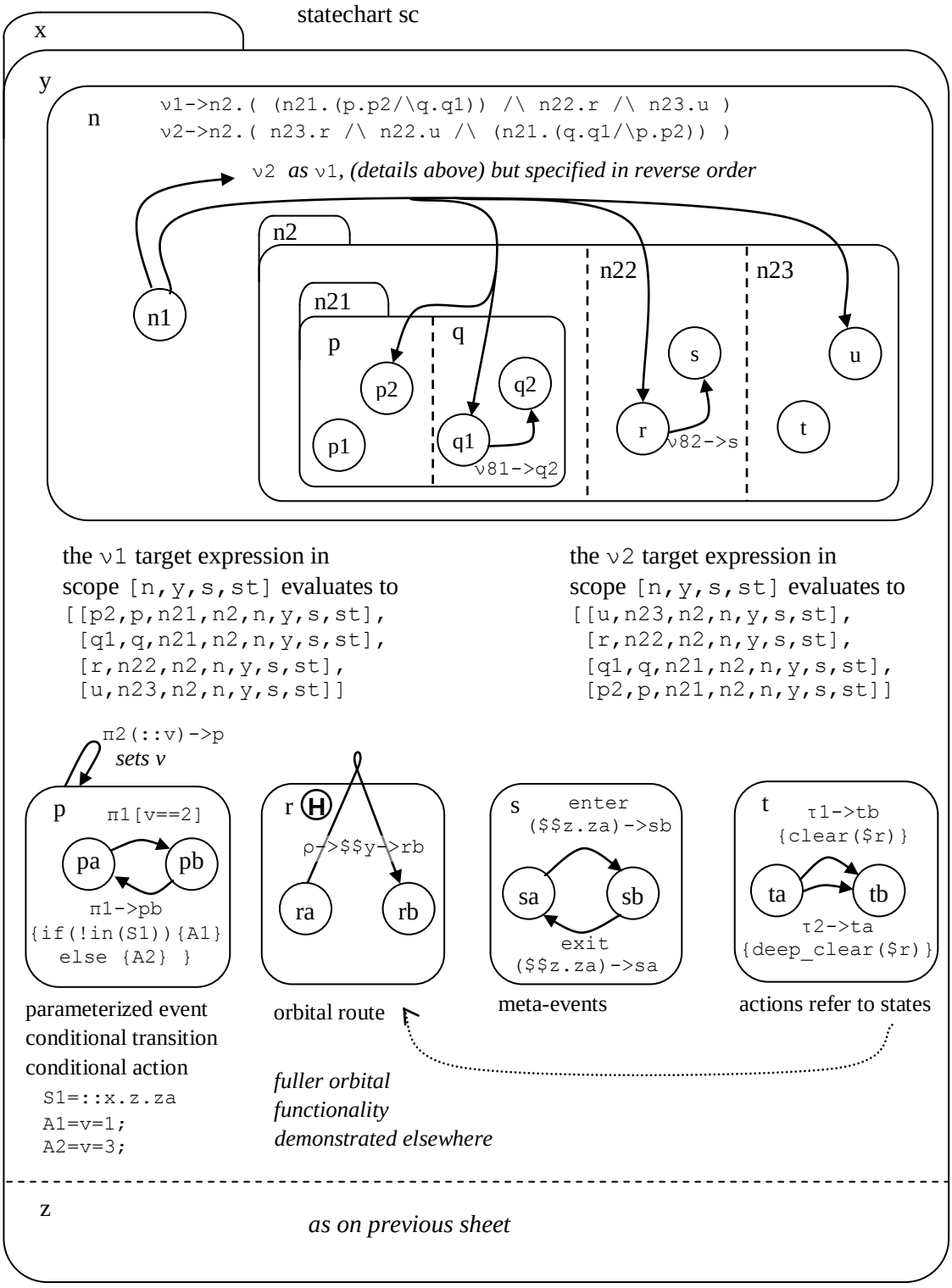


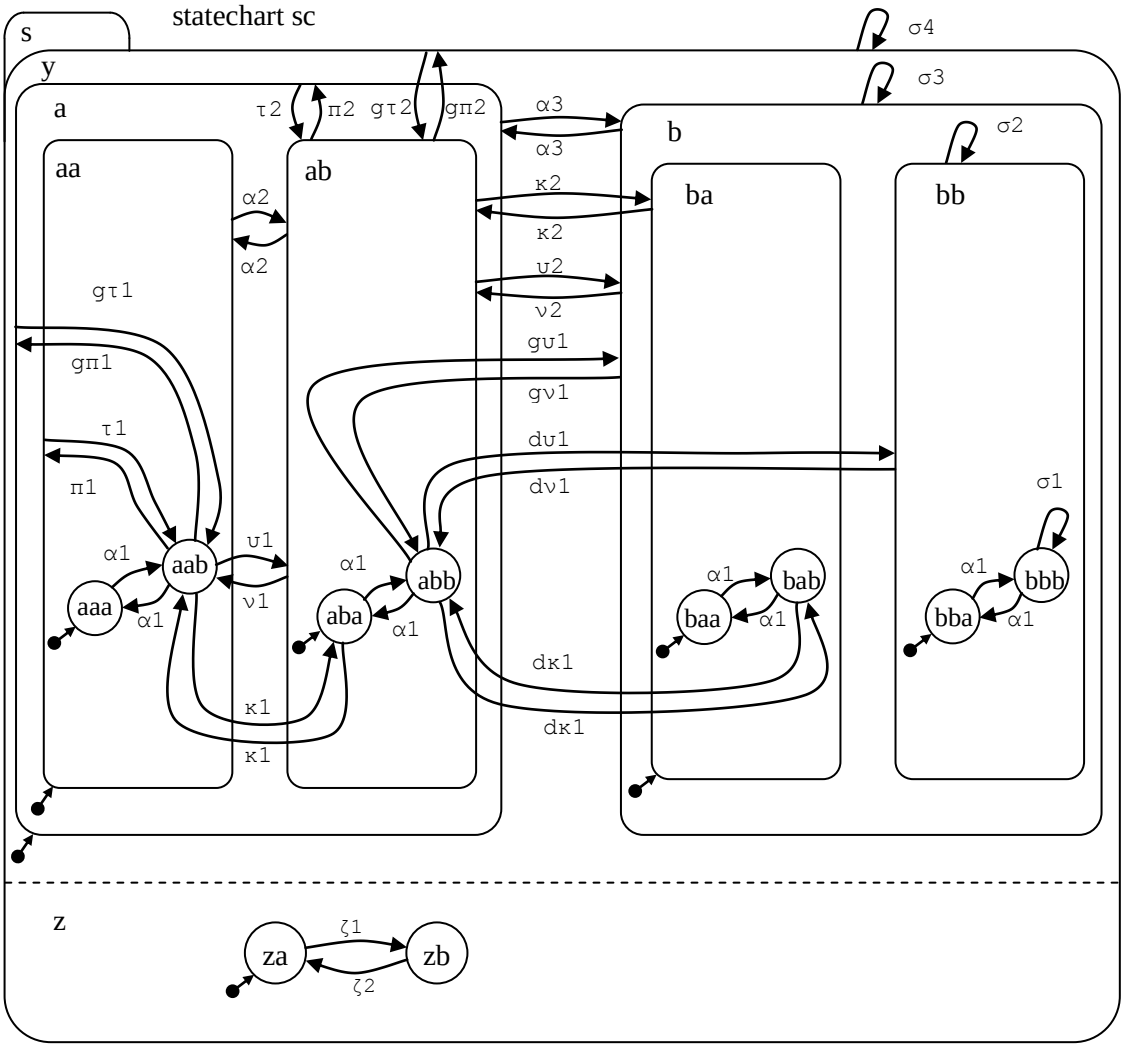
Figure 52. Specifying States - continued



6.3 Deep Nesting

6.3.1 Deep Cluster nesting [model `t6222`]

Figure 53. Deep Cluster Nesting



Notes on event notation (showing destination relation) follow.

Figure 54. Terminology for relationships (with event naming convention)

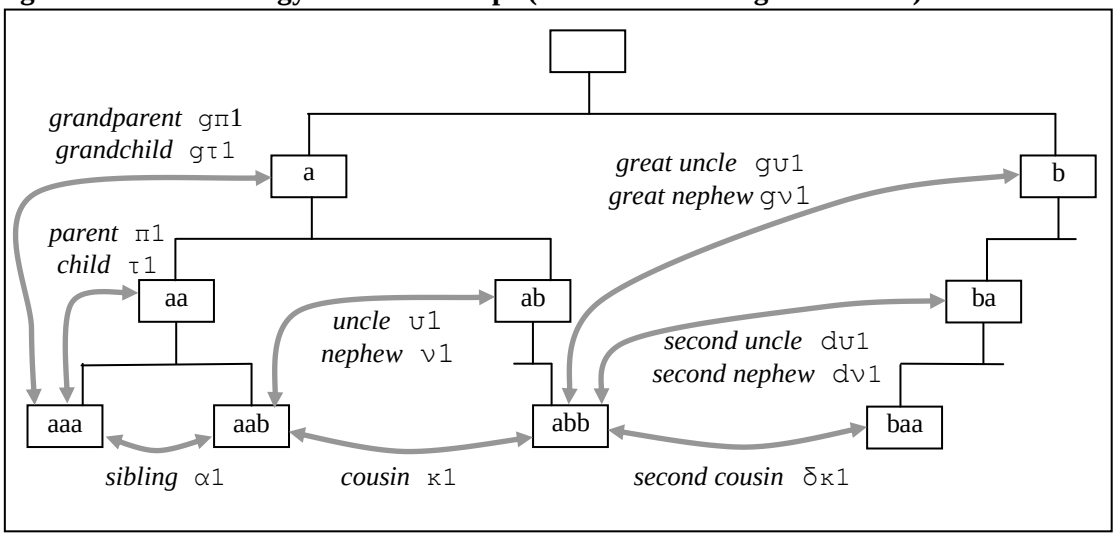


Figure 55. Nonleaf-Nonleaf relationships

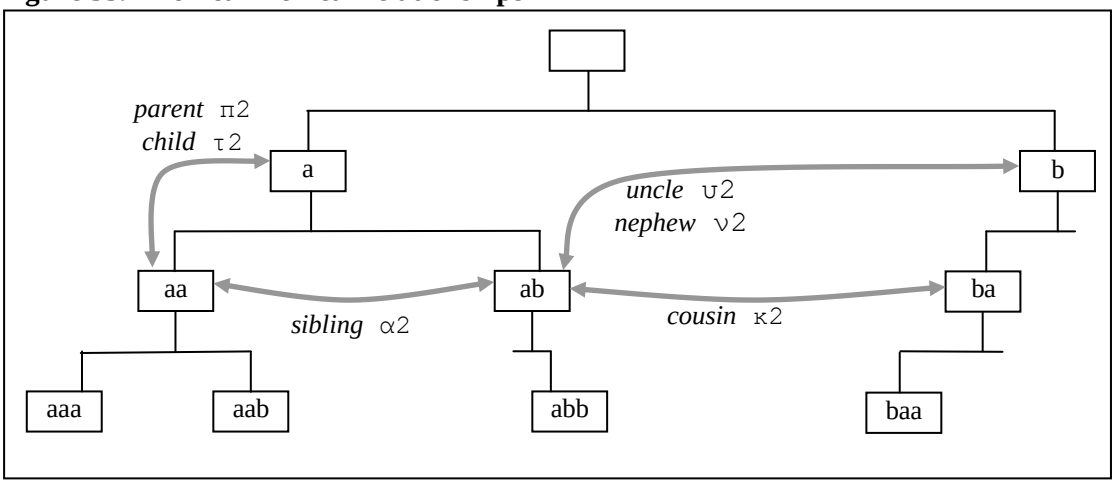


Table 4. Matrix of event names

					one removed				two removed			
to→ from↓	leaf	lp	lgp	lggp	leaf	lp	lgp	lggp	leaf	lp	lgp	lggp
leaf	α1 sibling	π1 parent	gπ1 gr-par	ggπ gr-gr-p	κ1 cousin	υ1 uncle	gυ1 gr-un		dκ1 2cousin	dυ1 2uncle		
lp	τ1 child	α2 sibling	π2 parent	gπ1 gr-par	ν1 nephew	κ2 cousin	υ2 uncle	gυ2 gr-un	dν1 2neph	dκ2 2cousin	dυ2 2uncle	
lgp	gτ1 gr-ch	τ2 child	α3 sibling	π3 parent	gν1 gr-neph	ν2 nephew	κ3 cousin	υ3 uncle		dν2 2neph	dκ3 2cousin	dυ3 2uncle
lggp	ggτ	gτ2	τ3	α4		gν2	ν3	κ4			dν3	dκ4

Legend:
lp=leaf-parent; lgp=leaf-grand-parent; lggp=leaf-great-grand-parent; shaded=not tested

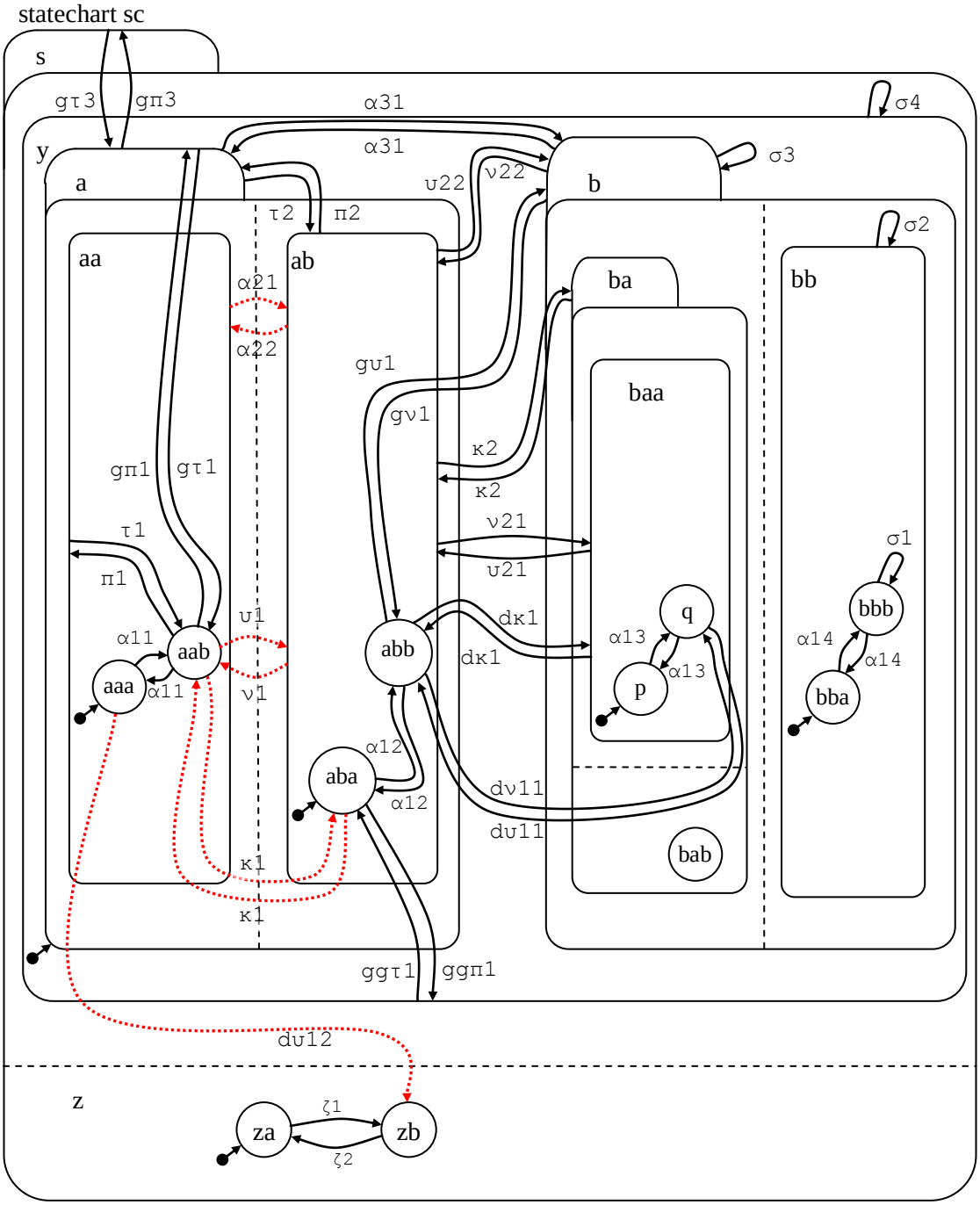
Predicate `gn_relname(FROM,TO,RELATION)` will produce relation names, given machine paths to be read from right to left, e.g.

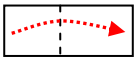
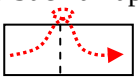
```
gn_relname(
  [u1,u2,u3,u4,u5,u6,u7,u8,u9,c],
  [p1,p2,p3,p4,p5,            c],R).

R=[5,[great,great,great,uncle]]).
```

i.e. `TO` is a *fifth great,great,great,uncle* of `FROM`.

6.3.2 Deep set nesting (model `t6224`)

Figure 56. Deep set nesting



With sets, any direct transition crossing a set member separator () is in principle illegal. Such an apparent transition can be *re-interpreted* as legal transition by introducing an orbit: 

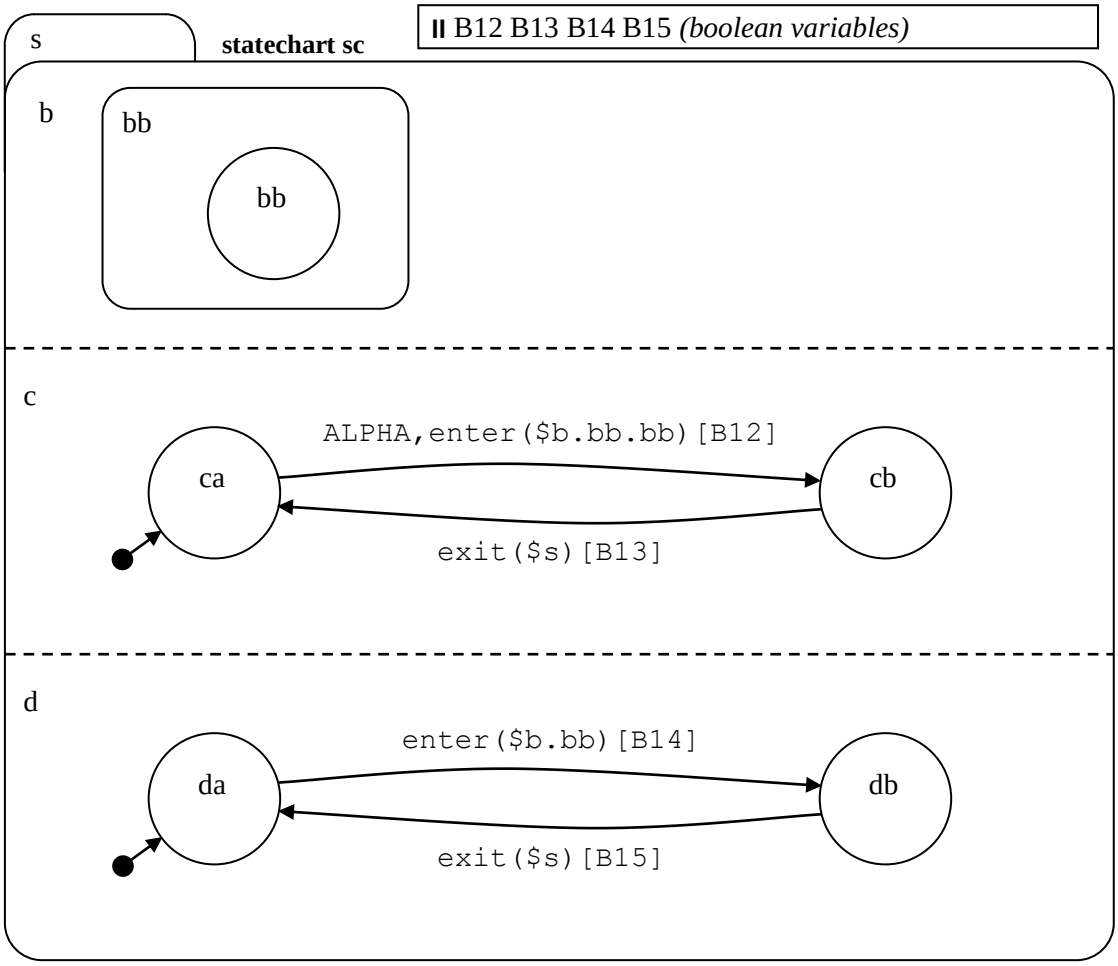
Orbital transitions provide a legal way of transitioning to a set member, as long as any exited set is re-entered.

The elements of a set are normally sets or clusters, so we chiefly use clusters as the innermost set members, with one leaf-state set member for completeness.

The above figure allows for exercising of non-orbital direct-ancestor/direct descendant transitions.

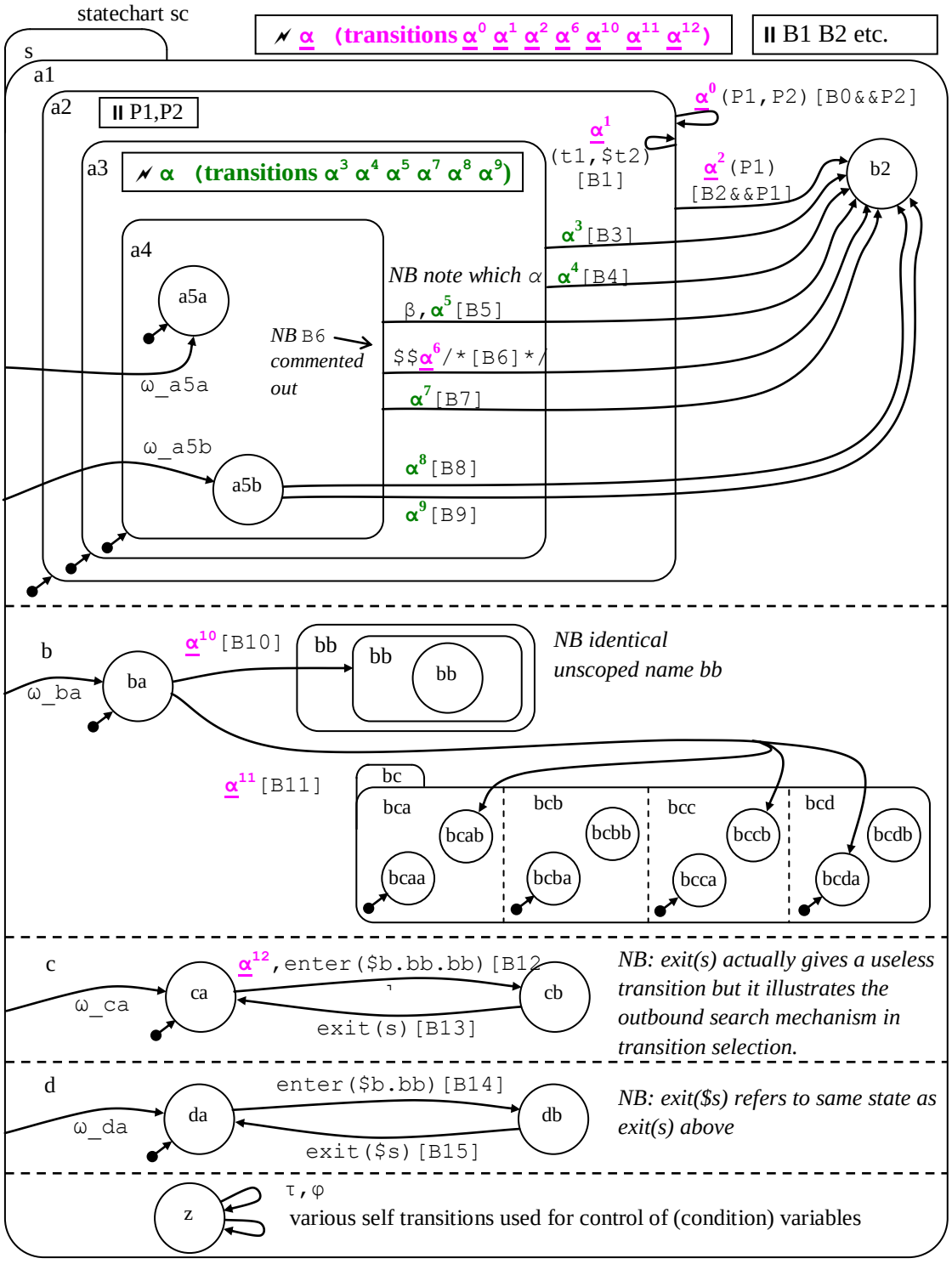
6.4 Transition Selection

Figure 57. Simple Enter-Exit Transition Selection [model `t6230`]



Note: This model is not suitable for user-level driving and is used at an API level.

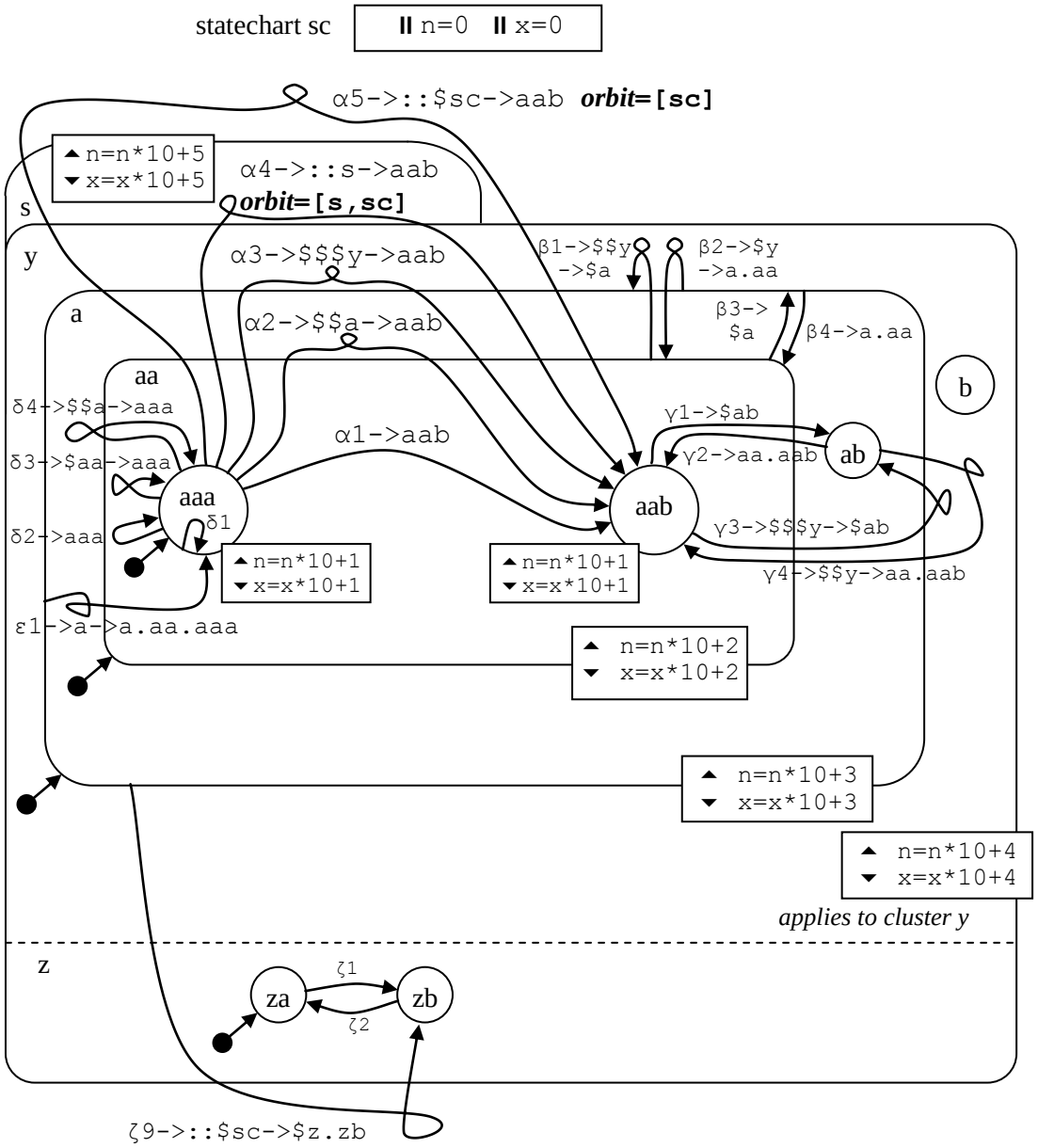
Figure 58. Transition Selection Basics - [model `t6240`]



Notes: This model is a 'legacy' model, used in some low-level tests, but is not particularly suitable as a transition demonstration model due to the diversity of features. There ere two events α, with superscripts added as a means of identifying transitions on them.

6.5 Orbits

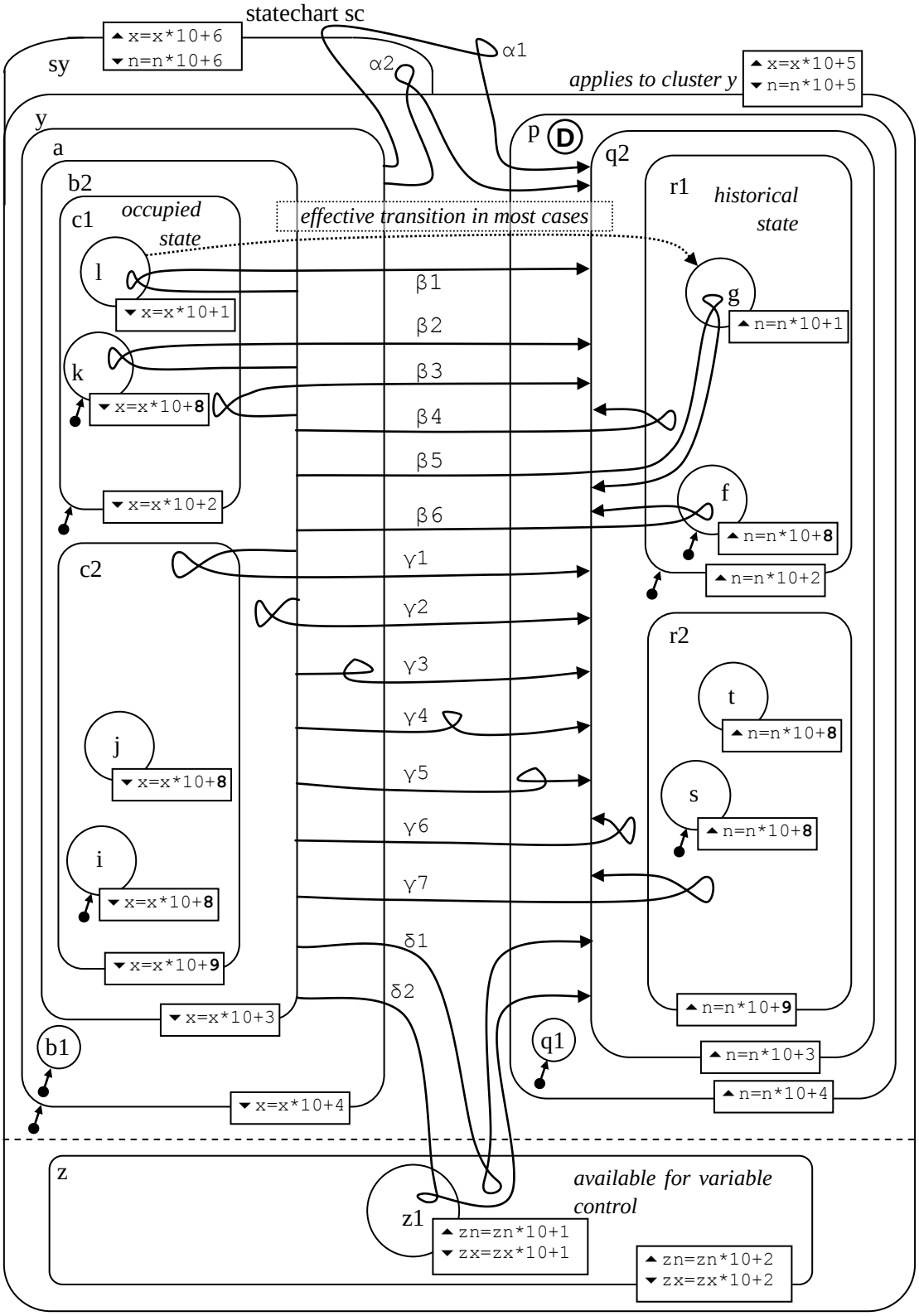
Figure 59. Orbits [model `t6260`]



Notes:

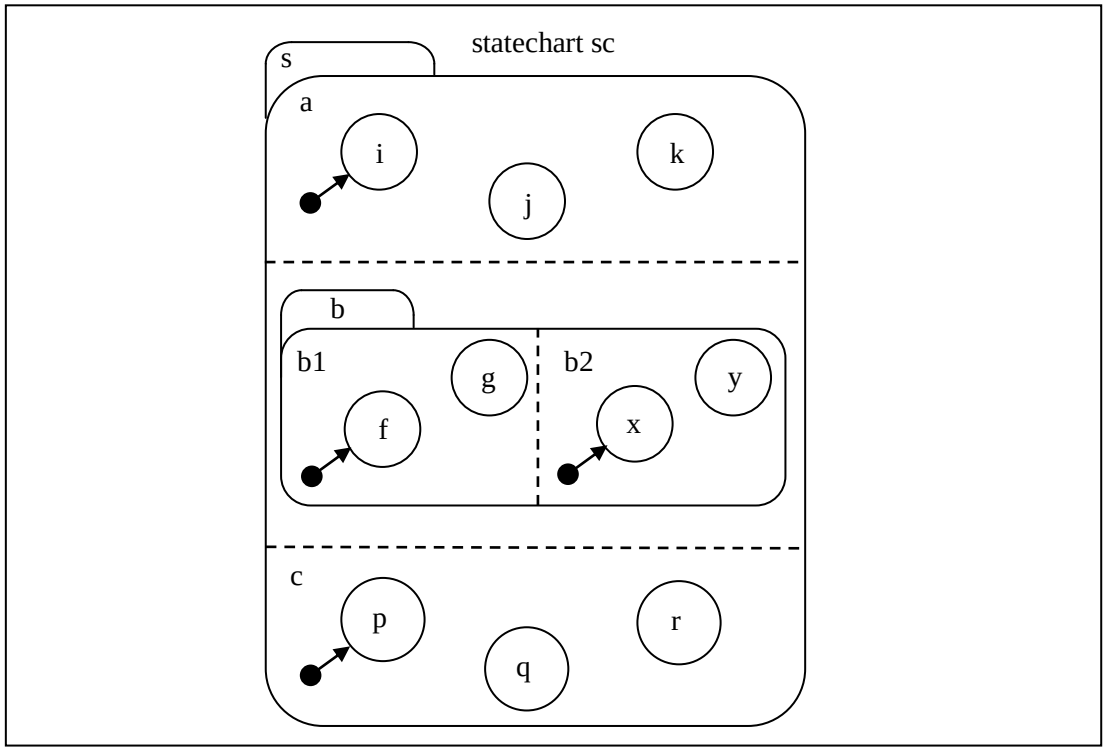
- Variables x and n are reset by ω transitions (specific set-to-state transitions), not shown in the diagram.
- If an orbital transition arc cuts through *n* member-state boundaries, the orbital state can be addressed using *n+1* $-signs.

Figure 60. Orbits - Legalisation of doubtful orbits [model t6264]



6.6 Common Tree Removal

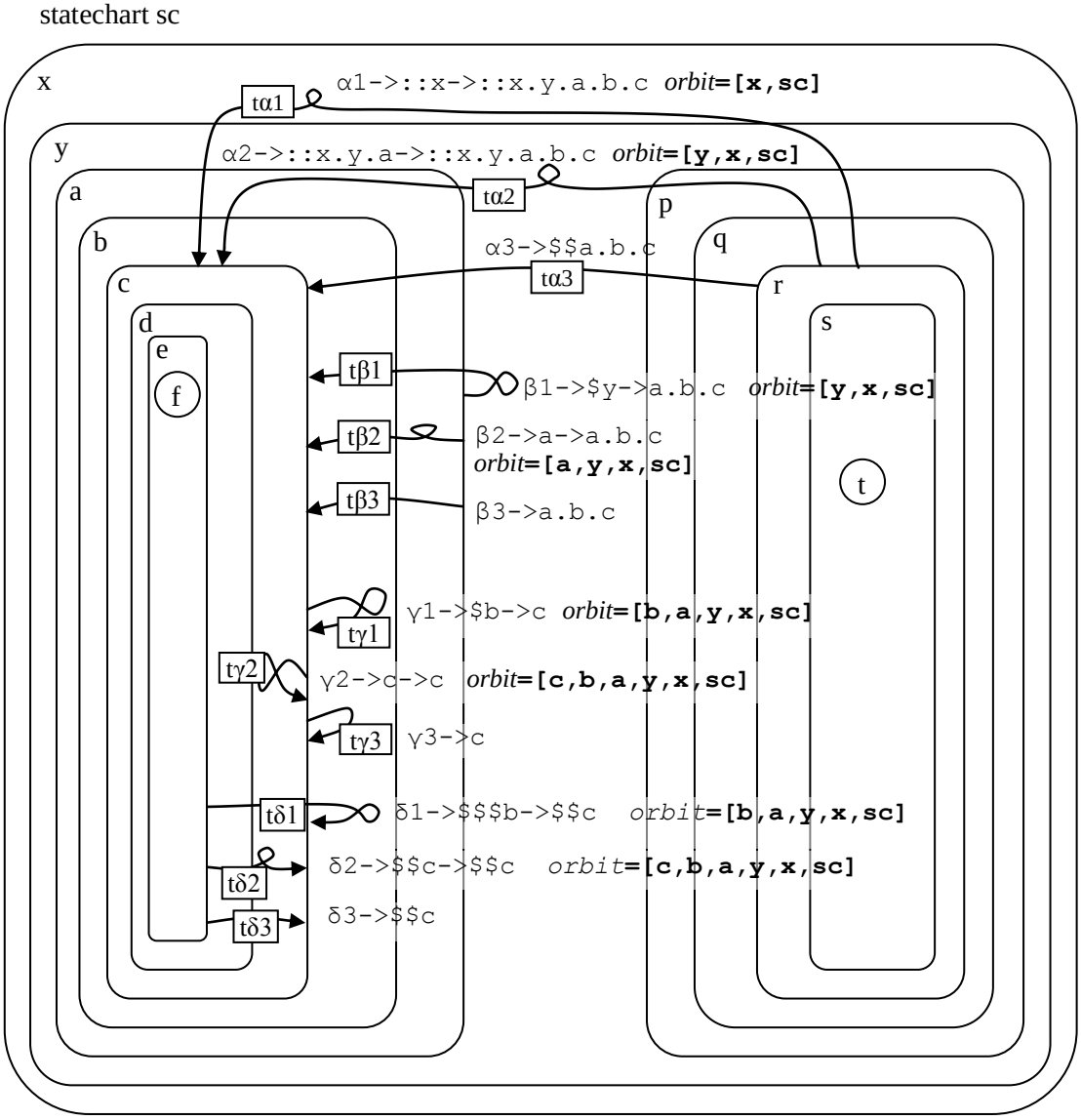
Figure 61. Common tree removal in sets [model `t6270`]



Used without transitions, generating explicit enter/exit trees, in demonstration programs.

6.7 Scope of Enter/Exit Trees

Figure 62. Scope of enter/exit trees [model `t6280`]



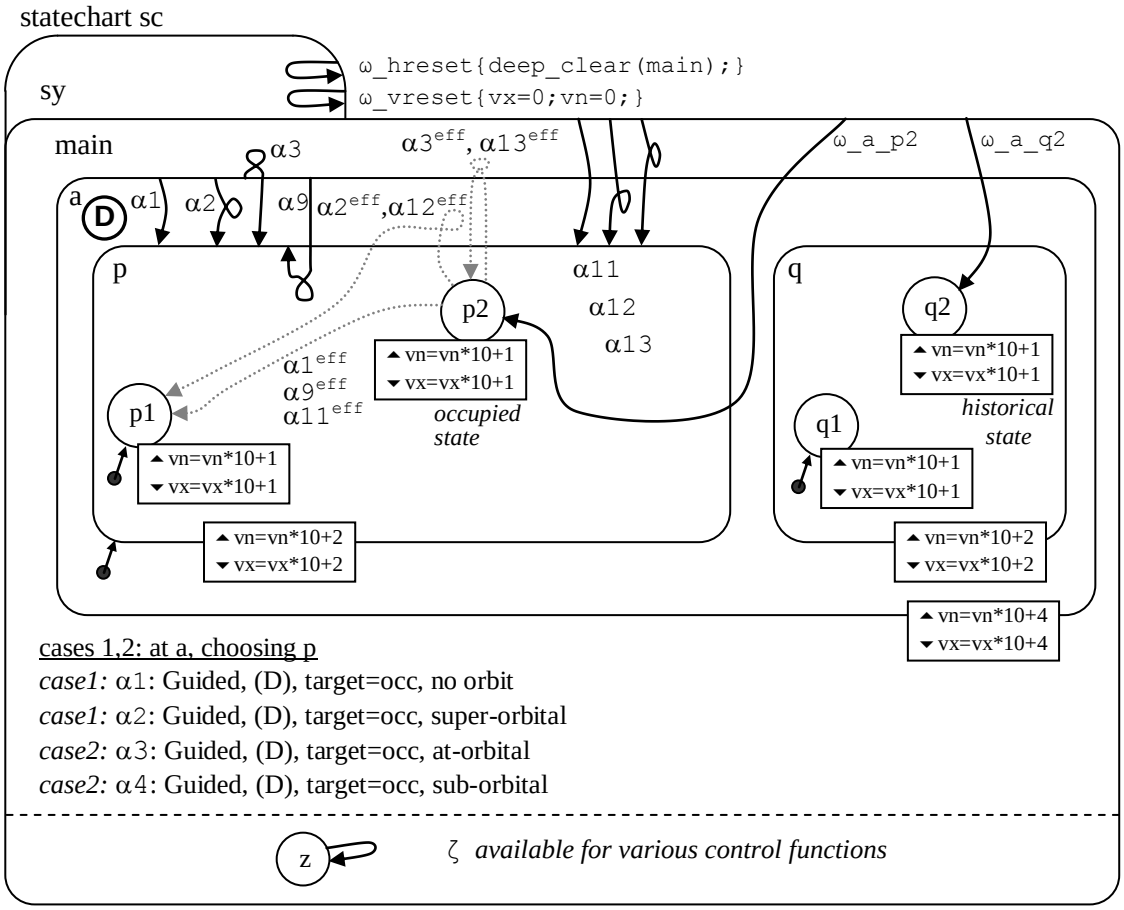
The algorithm tested here is described in [StCrMain] in the section on the transition course.

This model is used for low-level API testing and is not suitable as a high-level demonstration, since the exit and entry tree scopes are not visible at a high level.

This model is also exhibited in the main STATECRUNCHER report [StCrMain], in the section on the transition algorithm, showing the scopes involved.

6.8 Transition Course

Figure 63. Entry tree logic for clusters (1) - [model `t6291`]



Notes

- The algorithm tested here is described in [StCrMain] in the section on the transition course. The terminology is taken from there (case numbers, guided/unguided entry, dho=deep history obligation).
- Notation such as `α12eff`, with a dotted transition arc, refers to the effective transition of the one on event `α12`.
- ***upon enter*** *and* ***upon exit*** *assignments are made throughout the model:*
  - ▲`vn=` *upon enter assignment on entry into state above the symbol*
  - ▼`vx=` *upon exit assignment on exit of state abovethe symbol*

  ▲`vn=vn*10+1` at leaf level; `vn=vn*10+2` at parent; `vn=vn*10+3` at grandparent
  ▼`vx=vx*10+1` at leaf level; `vx=vx*10+2` at parent; `vx=vx*10+3` at grandparent
- ***This model is used at API level as well as high level, and should not be changed lightly***

Figure 64. Entry tree logic for clusters (2) -[model `t6292`]

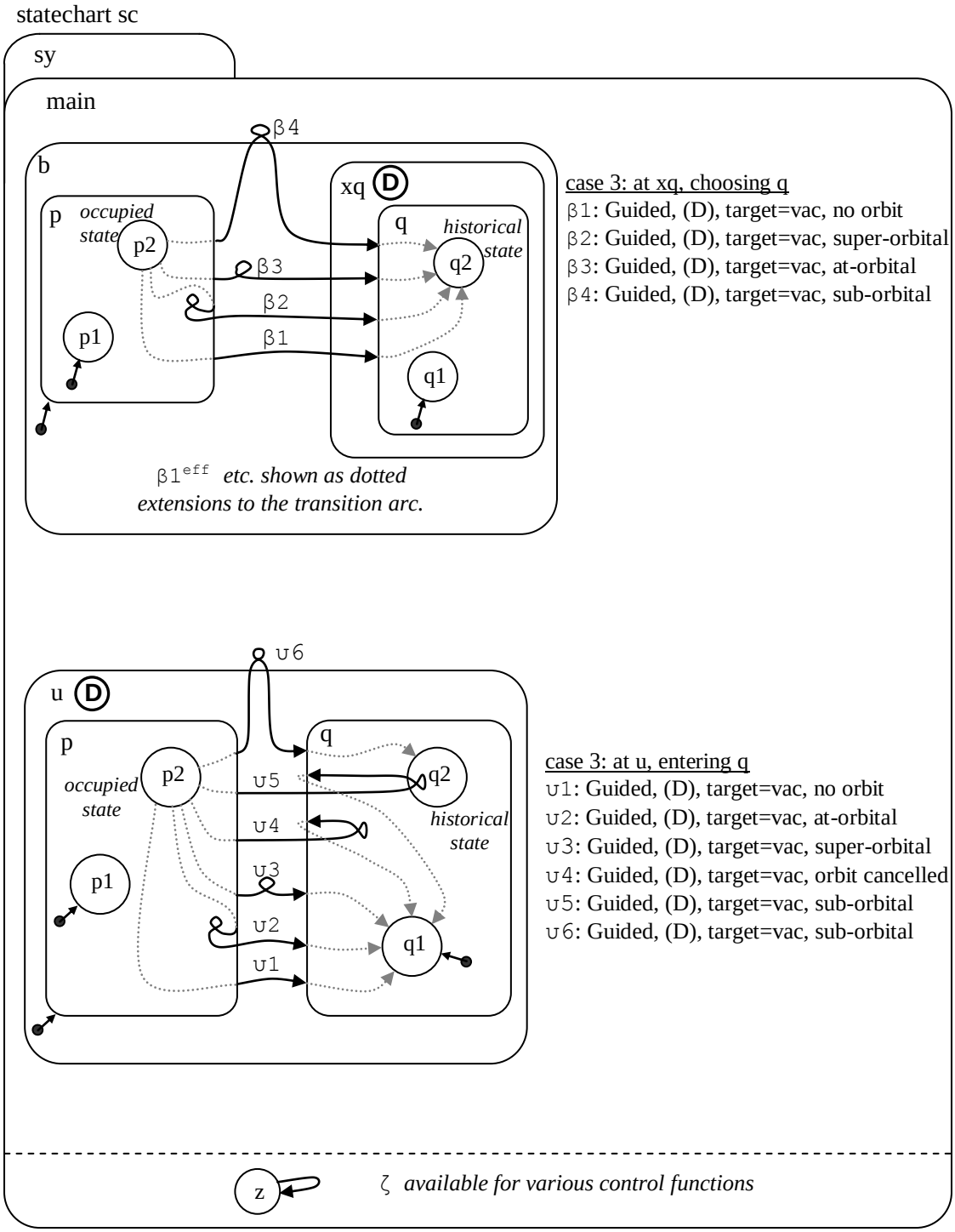


Figure 65. Entry tree logic for clusters (3) - [model `t6293`]

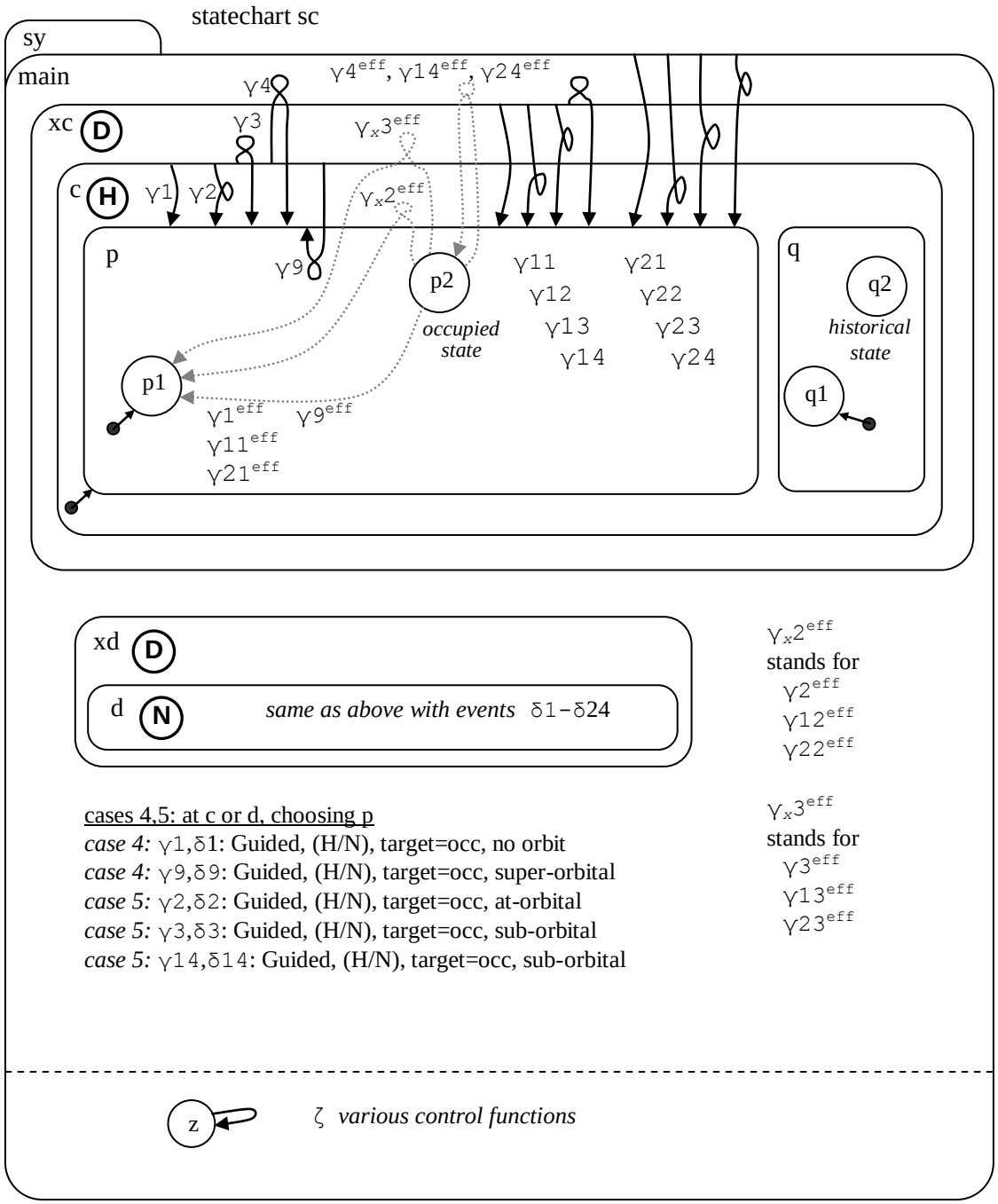


Figure 66. Entry tree logic for clusters (4) - [model `t6294`]

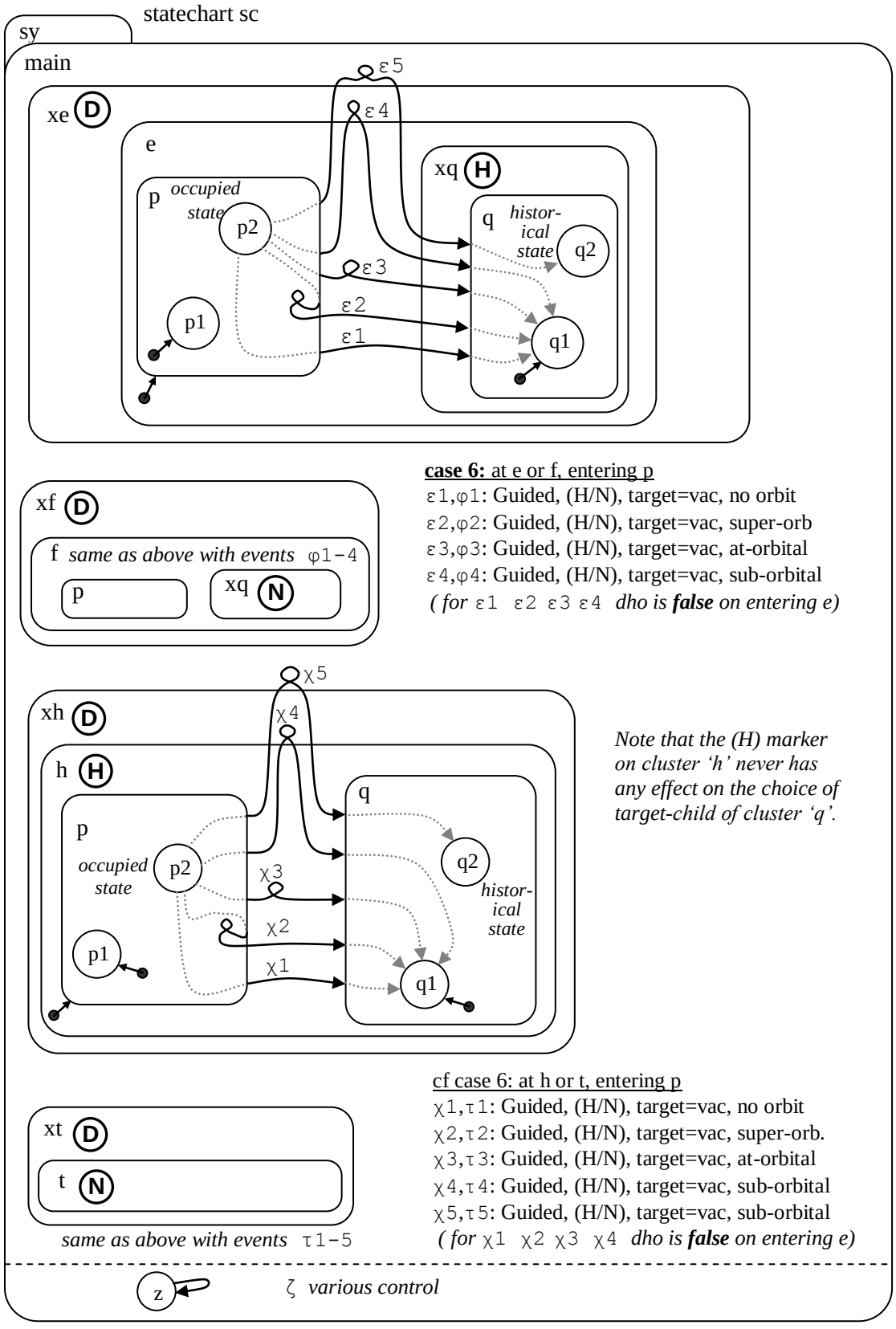
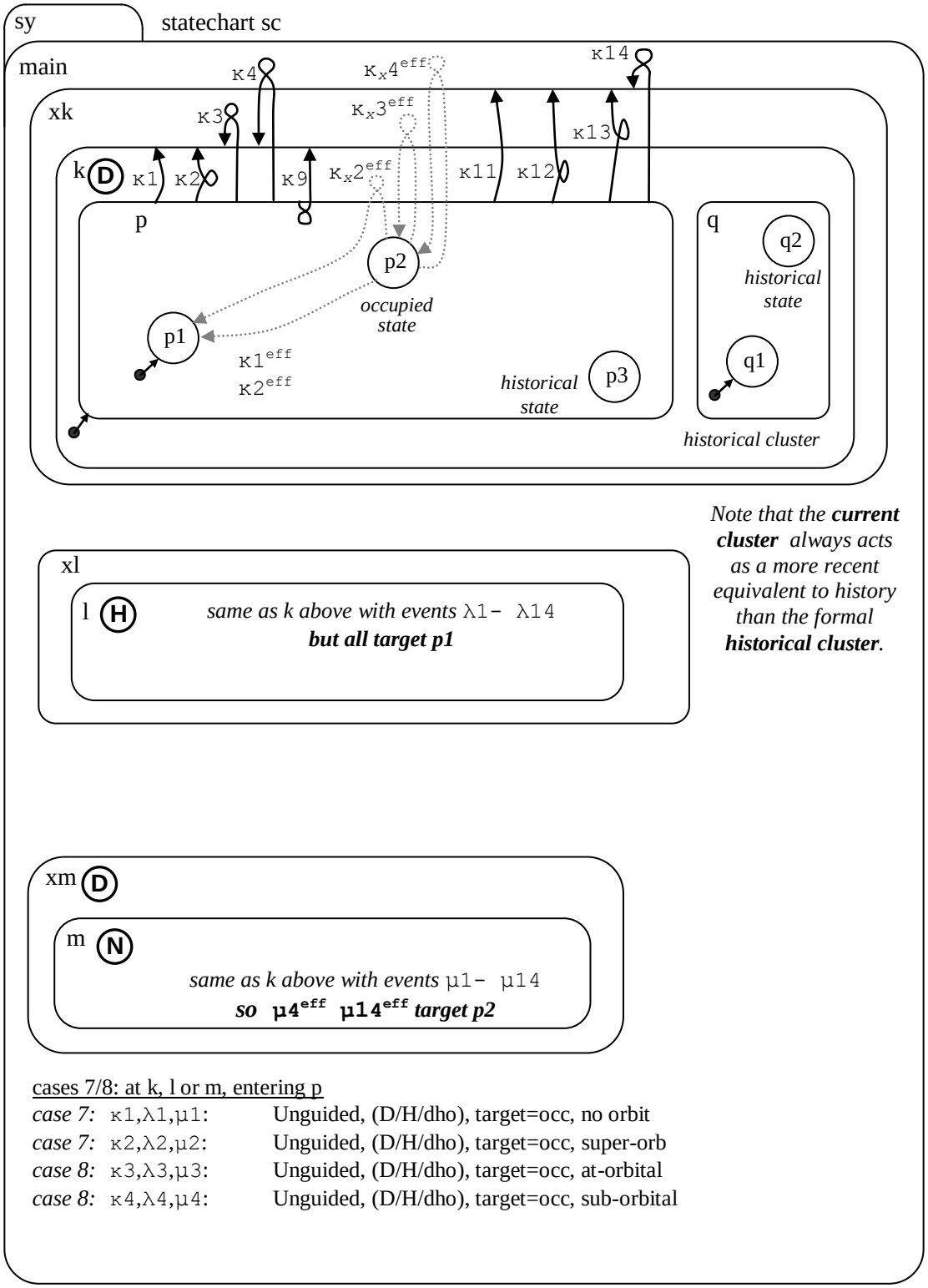


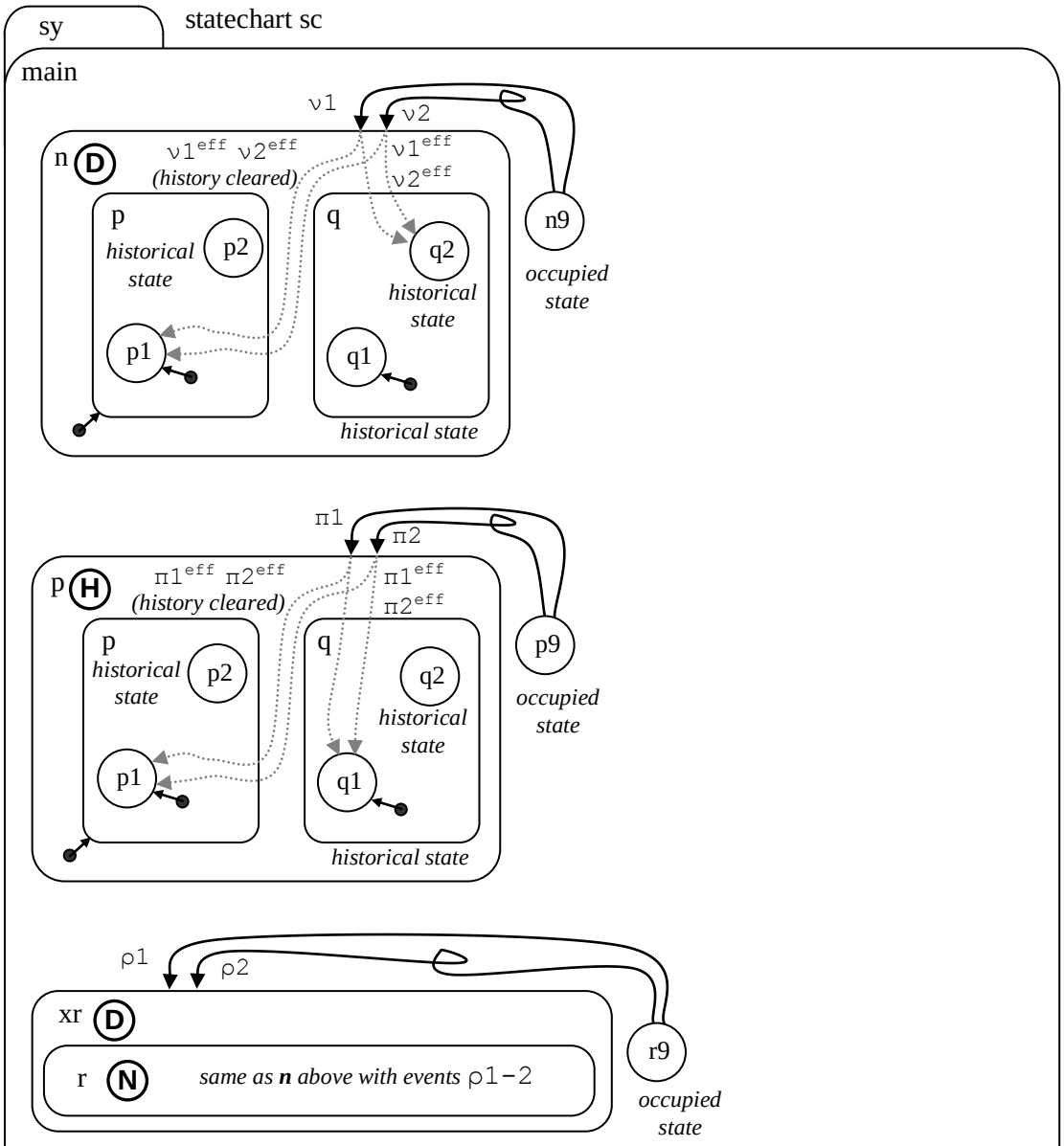
Figure 67. Entry tree logic for clusters (5) - [model `t6295`]



cases 7/8: at k, l or m, entering p

case 7: κ1,λ1,μ1: Unguided, (D/H/dho), target=occ, no orbit
case 7: κ2,λ2,μ2: Unguided, (D/H/dho), target=occ, super-orb
case 8: κ3,λ3,μ3: Unguided, (D/H/dho), target=occ, at-orbital
case 8: κ4,λ4,μ4: Unguided, (D/H/dho), target=occ, sub-orbital

Figure 68. Entry tree logic for clusters (6) - [model `t6296`]



cases 9/10: at n, the outer cluster p, or r, entering the outer cluster p

case 9:	ν1:	Unguided, (D/H/dho), targ=vac, orbital, history available
case 9:	ν2:	Unguided, (D/H/dho), targ=vac, non-orb, history available
case 10:	ν1 history cleared:	Unguided, (D/H/dho), targ=vac, orbital, history not available
case 10:	ν2 history cleared:	Unguided, (D/H/dho), targ=vac, non-orb, history not available

Similarly π1,ρ1 etc

Figure 69. Entry tree logic for clusters (7) - [model `t6297`]

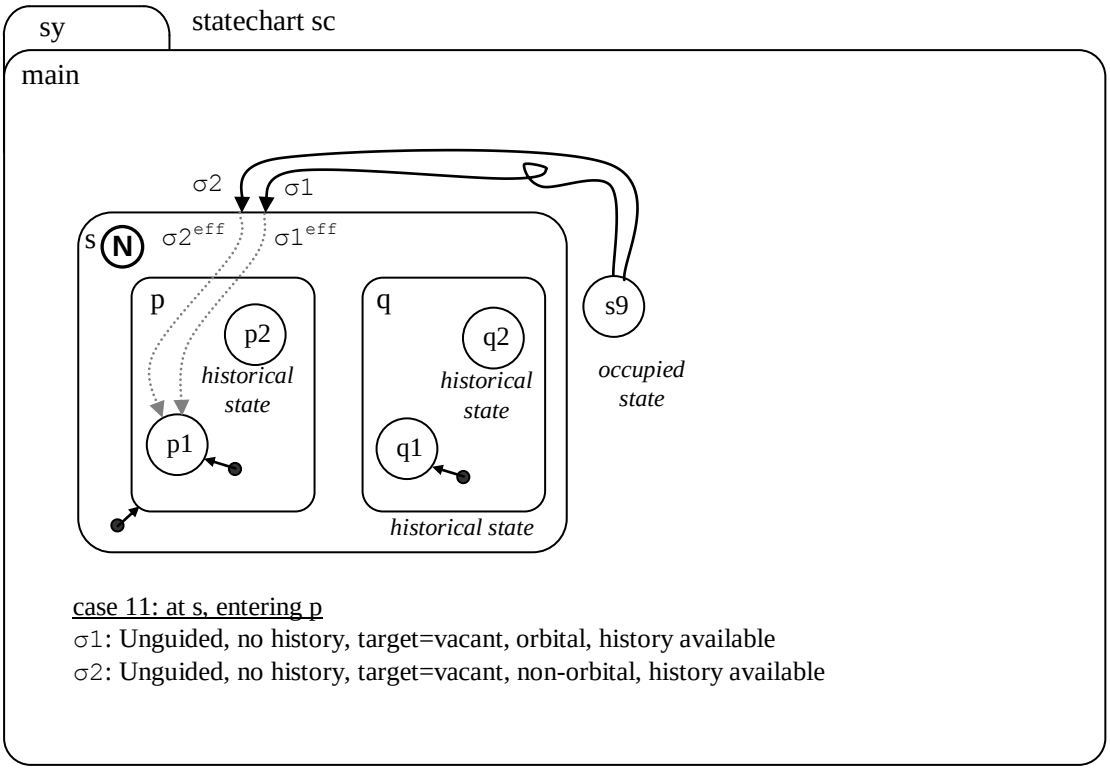
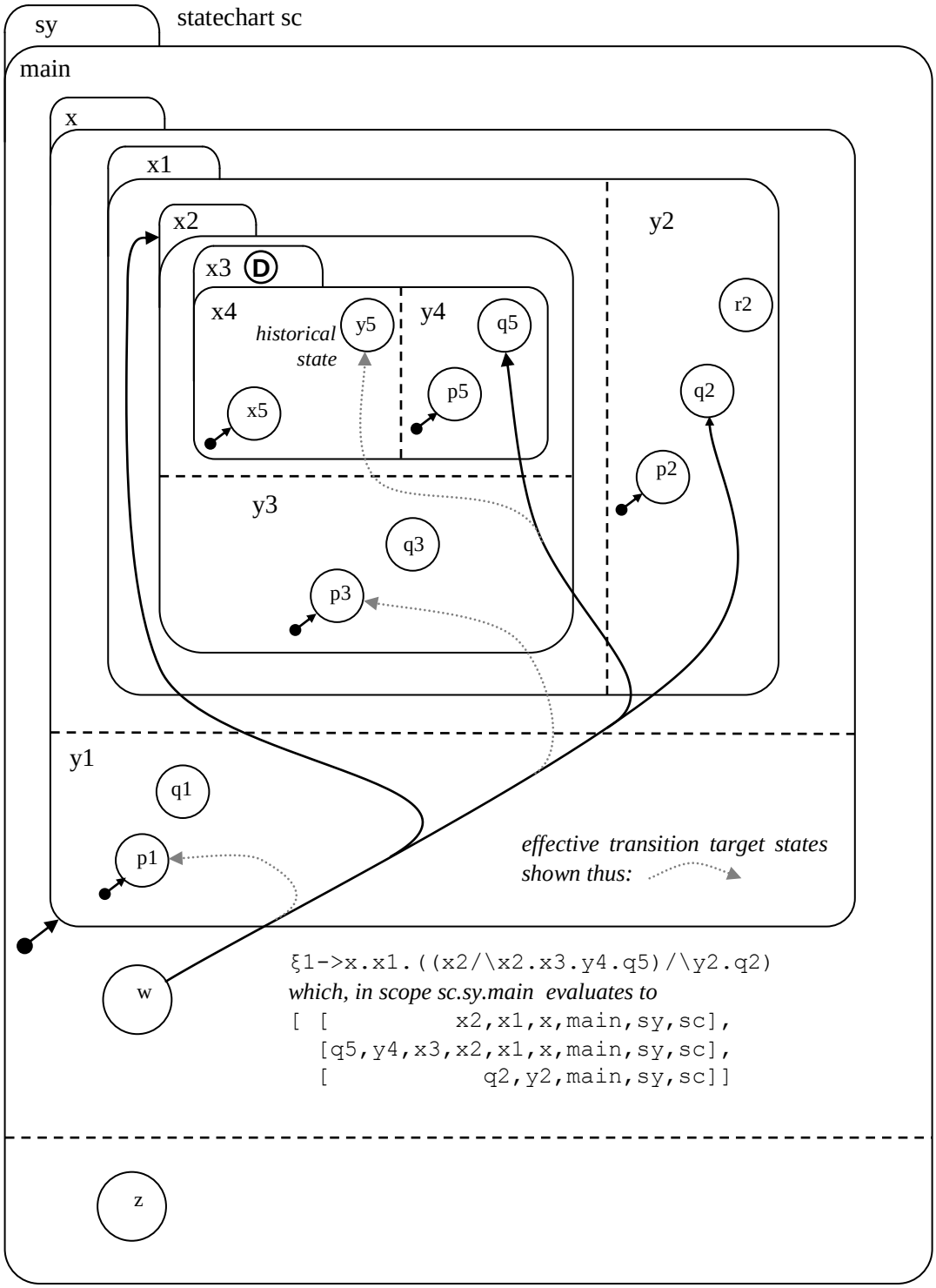


Figure 70. Entry tree logic for sets [model `t6305`]



Note that the target states may be in direct hierarchical (i.e. ancestral) relationship, though in such cases the higher member is redundant in the target specification. This applies to the target `x2` above.

6.9 Exercising Nondeterminism

The 5000-series of models exercises nondeterminism quite extensively. In this section we add a few heavy-duty examples.

6.9.1 Set Transit Nondeterminism

Figure 71. Set Transit Nondeterminism [models `t6310, t6311`]

*Model **`t6310`** is shown (9 sets to exit). Model **`t6311`** contains just member **`b`** (7 sets to exit). Note: event β1, (and more so σ and β2), is likely to cause combinatorial explosion.*

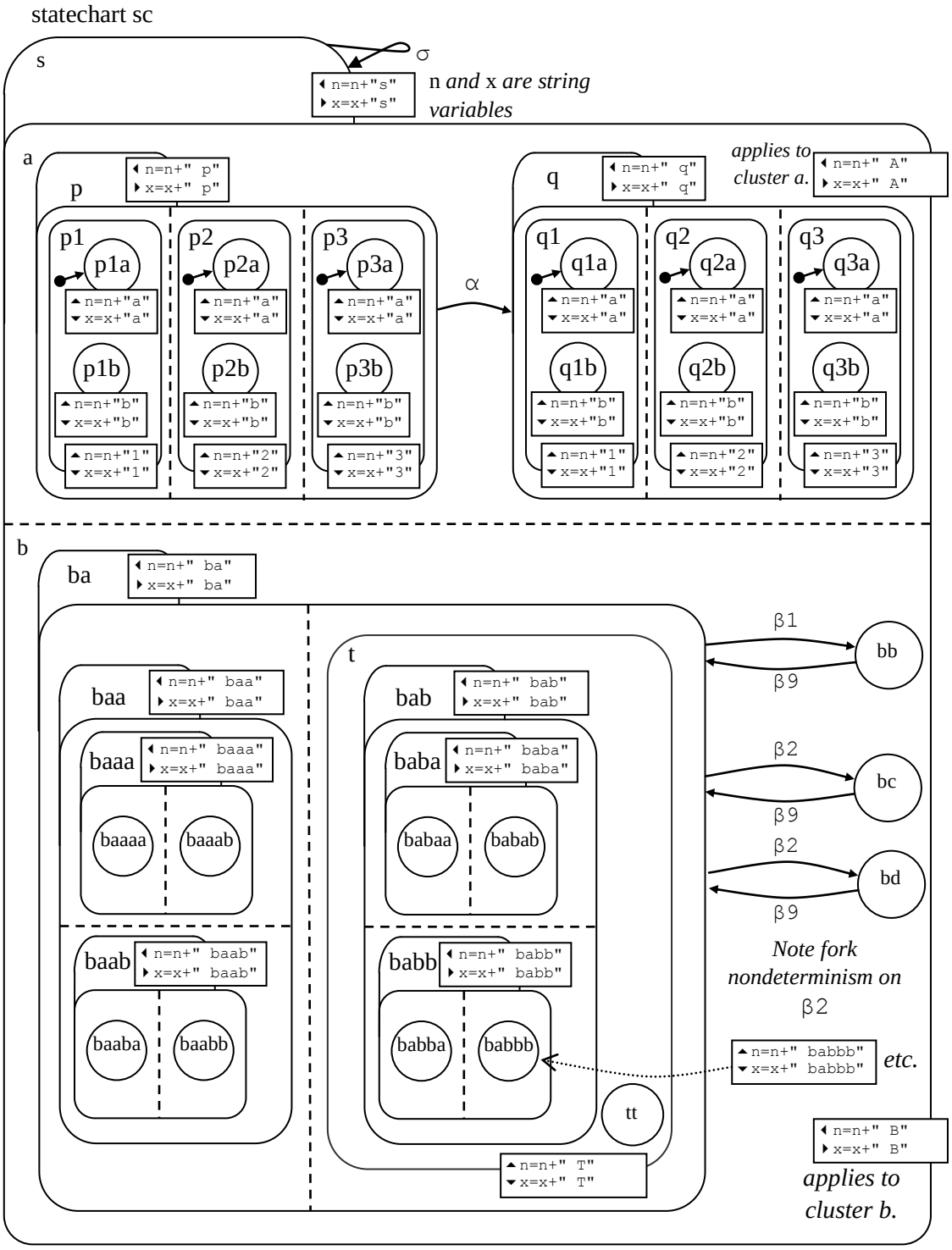
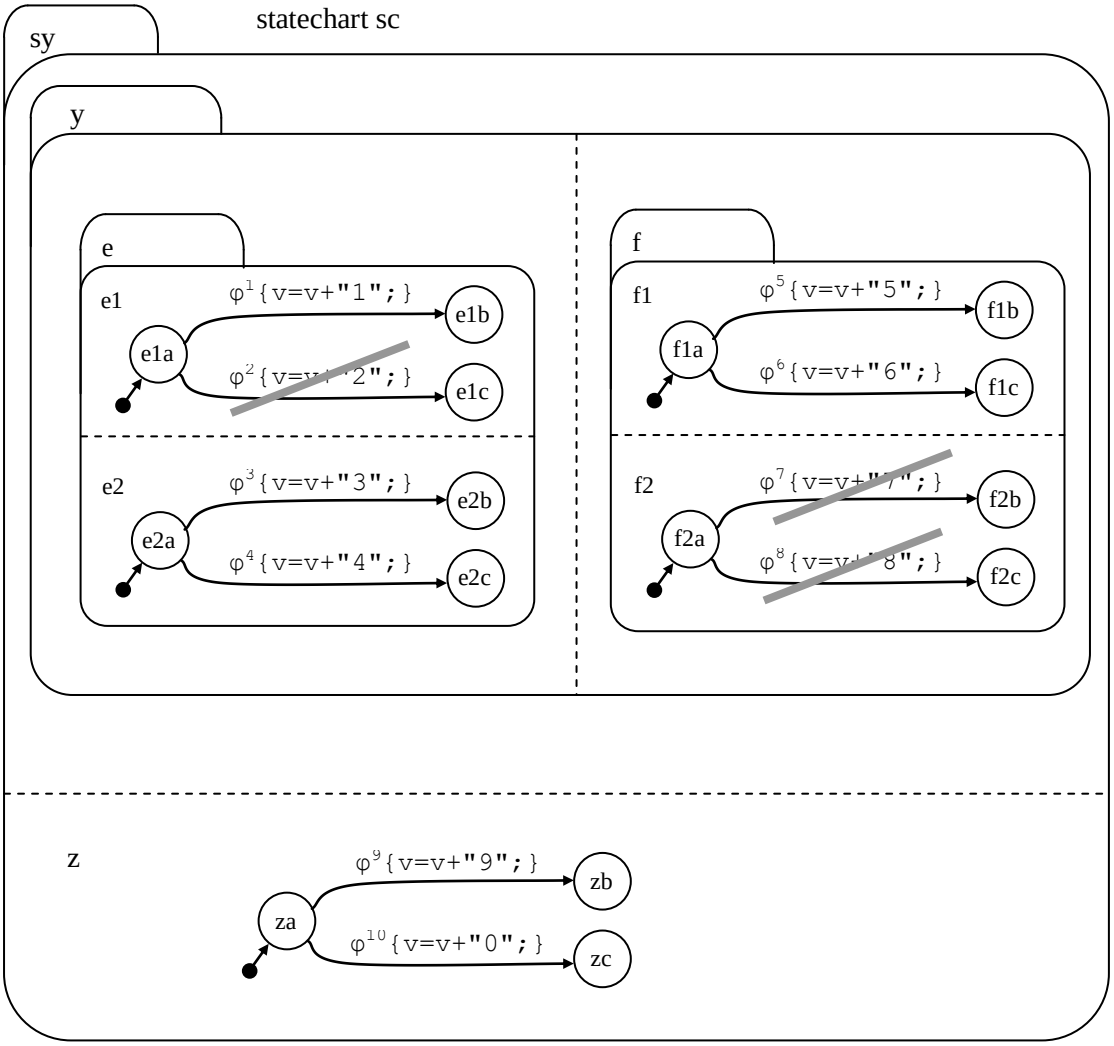


Figure 72. Race and fork nondeterminism in separate sets [models `t6350, t6351`]



The full model **`t6350`** is as shown with all transitions in place, including the ones struck out. We restrict this in **`t6351`** by excluding some transitions as shown by strike-out.

Under the restricted race condition permutation mode

```
med_set_tran
```

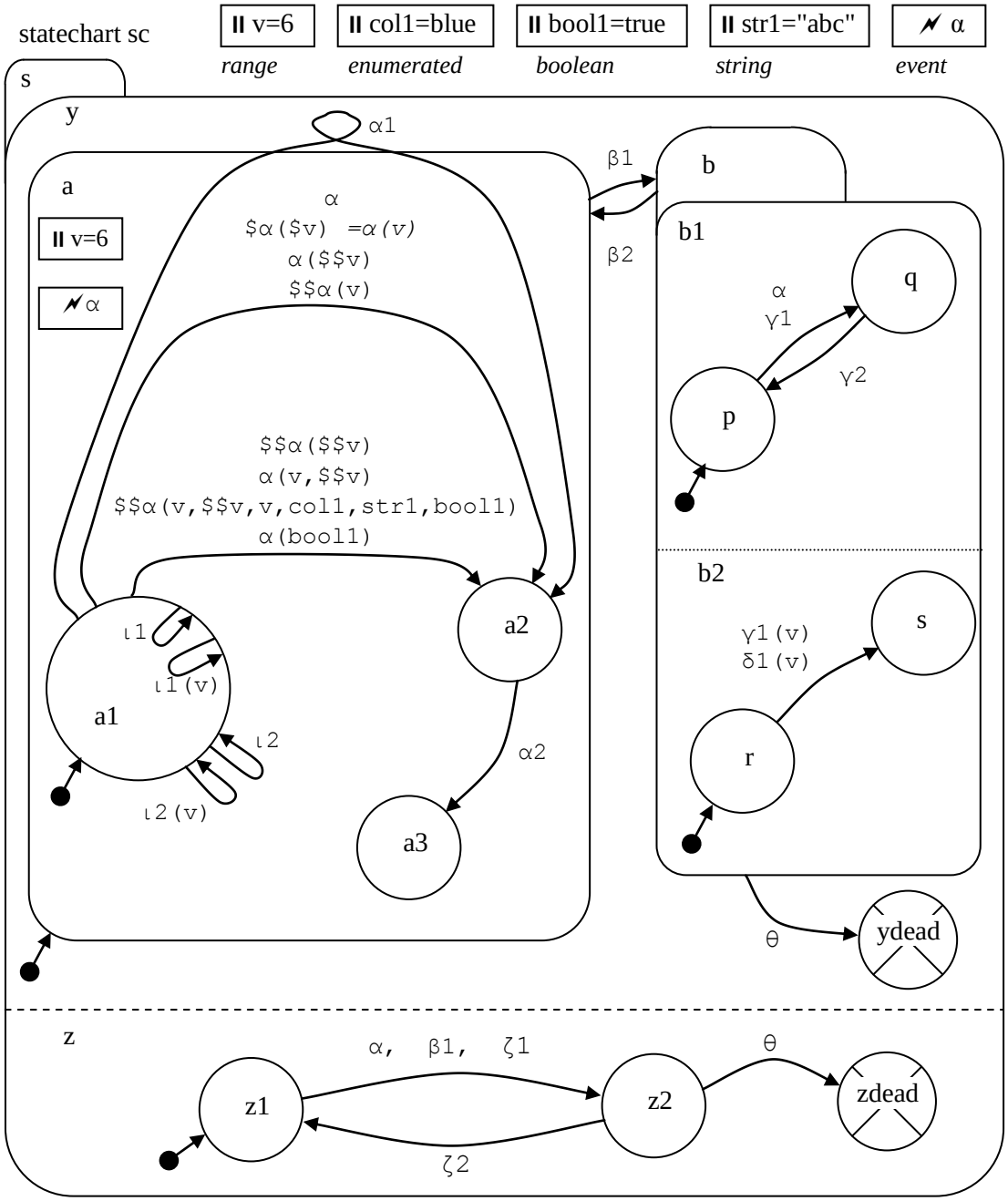
which gives 2n permutations of n elements (see comments following Figure 41), we have the following data on event φ.

Table 5. World generation in model `t6350`

	NF nr. of forks	NR nr. in race	nr of worlds $2^{NF}.2.NR$
Full Model **`t6350`**	5	5	$2^5.2.5 = 320$
Restricted Model **`t6351`**	3	4	$2^4.2.4 = 64$

6.10 Finding Active Events

Figure 73. Finding active events [model `t6410`]



Note that on-transition events and variables are evaluated in source-state scope, states in *state parent* scope. So `$α` references the local event α in the examples in cluster `a` above; `$$α` is needed to reference the global α.

6.11 Upon Exit/Upon Enter

These actions are adequately exercised in model `t5170`.

We reserved model number `t6420` for additional tests if needed.

Figure 74. Upon Enter / Upon Exit [model `t6420`]

6.12 Exercising History

History is adequately tested in model `t5220`.

We reserved model number `t6430` for additional tests if needed.

Figure 75. Model to exercise history [model `t6430`]

7. Stress Testing

7.1 Axes of Stress Testing

The main axes along which stress tests can be constructed are

- size
  - broad but shallow (cluster/set)
  - deep (cluster/set)
- execution speed
  - deterministic situations
  - nondeterministic situations

Some performance statistics are given, but timings may vary with the exact loading on the computer, in terms of core and mass memory, and any additional cpu loading (though the tests were run without any deliberate extra loading).

Notation: In the performance tables that follow, if an event is denoted as ",β" or "then β" this refers to processing event β after some previous event(s) – the context should make it obvious which events. The timing data will apply to the time to process β excluding the time taken by previous events.

7.2 Model Generation.

The models in this section are generated by generation programs.
Some common code for this is located in the `mk_sc` directory alongside the rest of STATECRUNCHER. The model generation modules themselves are located in the test model directory, alongside the place where the model itself is created, e.g. in directory

```
..StCr\StCr3ModelsTest\t7000st\t7110_stress_broad_clusters
```

The generation modules are normally loaded with STATECRUNCHER. A typical predicate to generate a module is

```
mk_t7110(10,12).   // k=10, n=12
```

The test suite *regenerates the models* with the parameters as set in the test scripts.

7.3 Combinatorial Explosion and Limited Permutation

Note that a major cause of slow performance is *combinatorial explosion*, due to the generation of permutations. The permutation options are denoted by FLAGs as follows

Table 6. Flags for permutation control

1 permutation	1 permutation backwards	2 permutations forwards and backwards	2n permutations all cyclic and anticyclic	All n! permutations
f_k1b	f_k1b	f_k2	f_k3a	f_1

The race (transition selection) permutation flag is stored in
`me_permute_trnsel_flag(FLAG).`

The set-transit permutation flag is stored in
`me_permute_settrnd_flag(FLAG).`

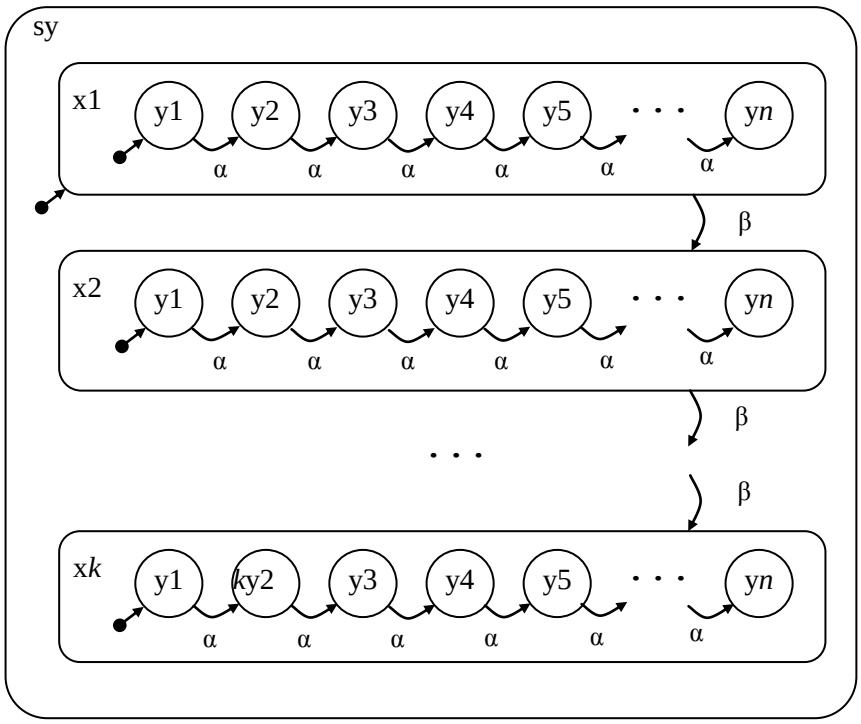
The flags can be set by
`me_set_permute_trnsel_flag(FLAG).` *(FLAG can be **f_k1b**, f_k2, f_k3a, f_1)*
`me_set_permute_settrnd_flag(FLAG).` *(FLAG can be **f_k1a**, f_k2, f_k3a, f_1)*

These forms of nondeterminism can also be switched by

```
me_no_race.      // equivalent to me_set_permute_trnsel_flag(f_k1b).
me_low_race.     // equivalent to me_set_permute_trnsel_flag(f_k2).
me_med_race.     // equivalent to me_set_permute_trnsel_flag(f_k3a).
me_high_race.    // equivalent to me_set_permute_trnsel_flag(f_1).

me_no_set_tran.     // equivalent to me_set_permute_settrnd_flag(f_k1a).
me_low_set_tran.    // equivalent to me_set_permute_settrnd_flag(f_k2).
me_med_set_tran.    // equivalent to me_set_permute_settrnd_flag(f_k3a).
me_high_set_tran.   // equivalent to me_set_permute_settrnd_flag(f_1).
```

See also the descriptions after Figure 40 and Figure 41.

Figure 76. Broad clusters [model `t7110`]

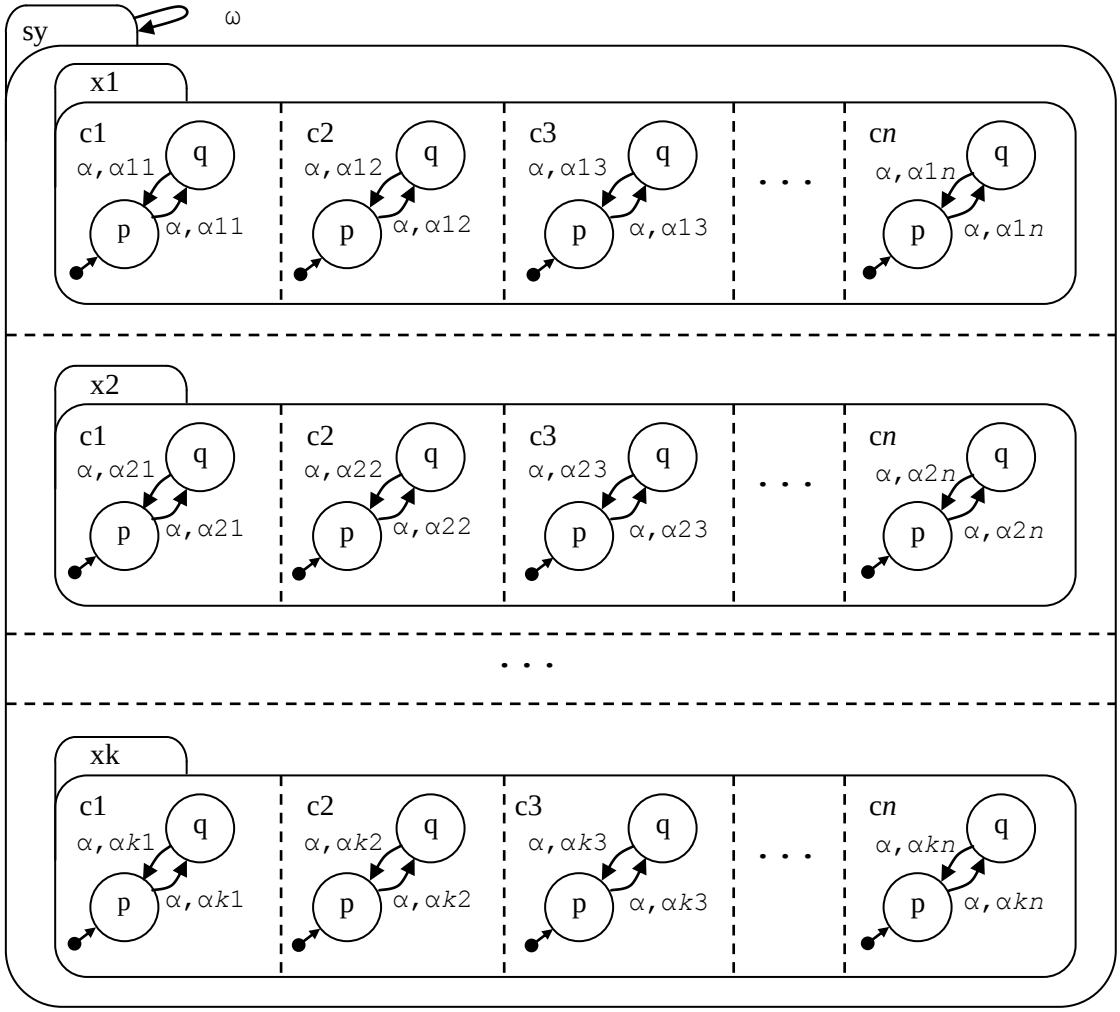


To generate this model: `mk_t7110(20,25).` *// k=20, n=25*

Table 7. Performance statistics of model `t7110`

Model params	Event	PROLOG	Op. System	Processor speed	Perm $Pm^{set\text{-}tran}$	Perm Pm^{race}	Time
(20,25)	α	SWI 5.0.3	Win98	300MHz	f_k3a	f_k3a	1s
(20,25)	*then* β	SWI 5.0.3	Win98	300MHz	f_k3a	f_k3a	2s

Figure 77. Broad sets [model `t7120`]



Note that this model can perform a massive set of transitions on α (but without exiting any sets), or any individual transition on α*xy*

To generate this model: **`mk_t7120(3,4).`**

Table 8. Performance statistics of model `t7120`

Model params	**Event**	**PROLOG**	**Op. System**	**Processor speed**	**Perm Pm$^{set\text{-}tran}$**	**Perm Pm^{race}**	**Time**
(3,4)	α	SWI 5.0.3	Win98	300MHz	f_k3a	f_k3a	6s
..	*then* ω	..	..	..	f_k3a	f_k3a	2m 26s
..	α	..	..	..	*none*	f_k3a	6s
..	*then* ω	..	..	..	*none*	f_k3a	0.2s
..	α	..	..	..	f_k3a	*none*	0.3s
..	*then* ω	..	..	..	f_k3a	*none*	2m 26s

Detailed note

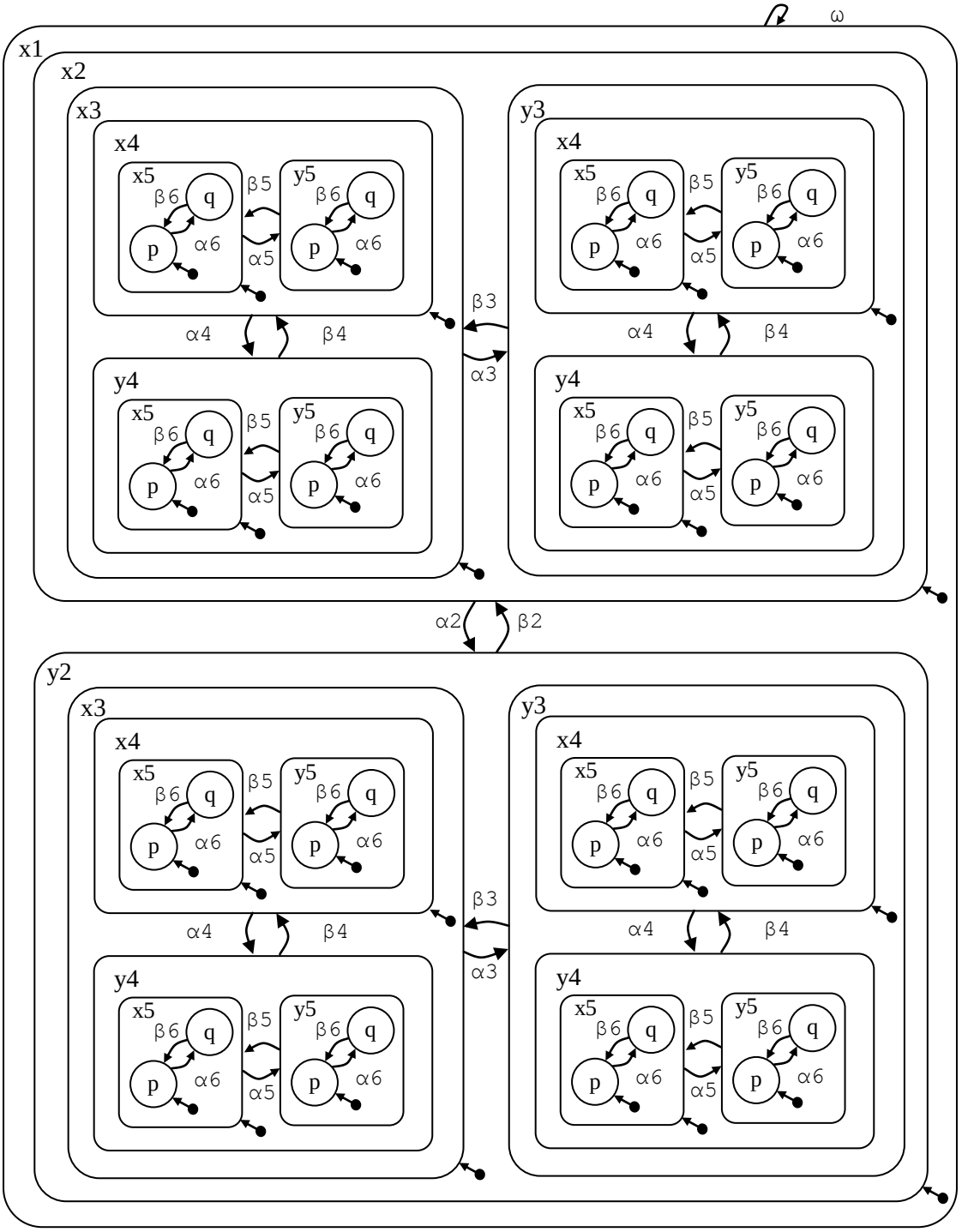
On event ω, permutations of exited leafstates are generated, e.g. a DXLIST (definitive exit list) generated by me_sc_6a.pl: me_process_task_in_world is

```
[[xt_leaf, [q, c1, x1, sy, sc]],
 [xt_leaf, [q, c3, x1, sy, sc]],
 [xt_leaf, [q, c2, x1, sy, sc]],
 [xt_leaf, [q, c1, x3, sy, sc]],
 [xt_leaf, [q, c3, x3, sy, sc]],
 [xt_leaf, [q, c2, x3, sy, sc]],
 [xt_leaf, [q, c1, x2, sy, sc]],
 [xt_leaf, [q, c3, x2, sy, sc]],
 [xt_leaf, [q, c2, x2, sy, sc]]]
```

There are many other permutations of this list.

This list is the basis of generating ***upon-exit actions*** and ***exit meta-events***.

Figure 78. Deep clusters - to level 5 [model `t7130`]

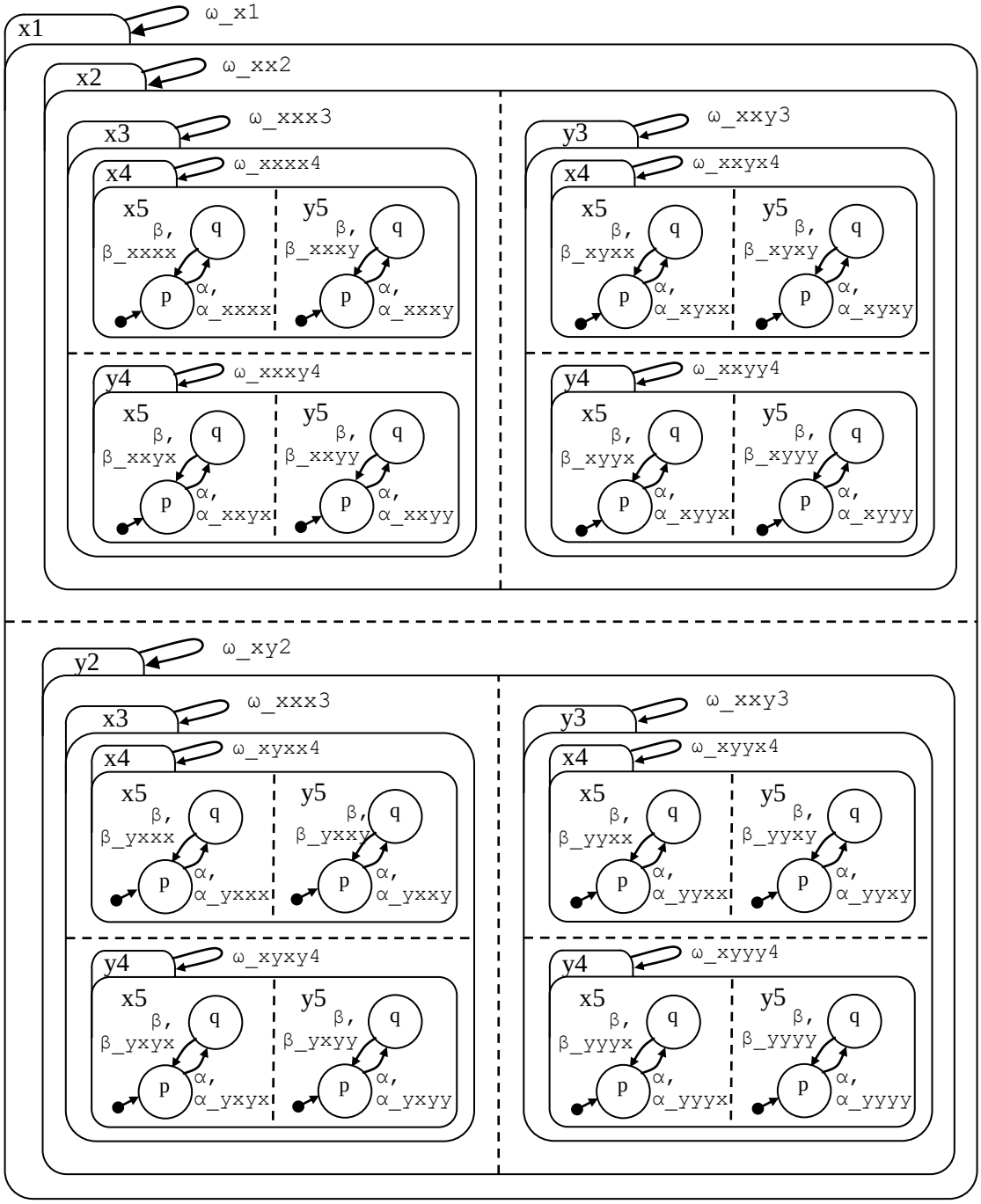


To generate this model: **`mk_t7130(5).`**
To generate a large model: **`mk_t7130(10).`** *// source file 1500 lines*

Table 9. Performance statistics for model `t7130`

Model params	Event	PROLOG	Op. System	Processor speed	Perm $Pm^{set\text{-}tran}$	Perm Pm^{race}	Time
(5)	α2	SWI 5.0.3	Win98	300MHz	f_k3a	f_k3a	0.06s
..	,ω	..	..	..	..	..	0.06s
..	α5	..	..	..	..	..	0.06s
..	,ω	..	..	..	..	..	0.06s
(10)	α2	..	..	..	..	..	20s
..	,ω	..	..	..	..	..	20s
..	α4	..	..	..	..	..	16s
..	,ω	..	..	..	..	..	16s
..	α10	..	..	..	..	..	4 s
..	,ω	..	..	..	..	..	4s

Figure 79. Deep Sets - to level 5 [model `t7140`]

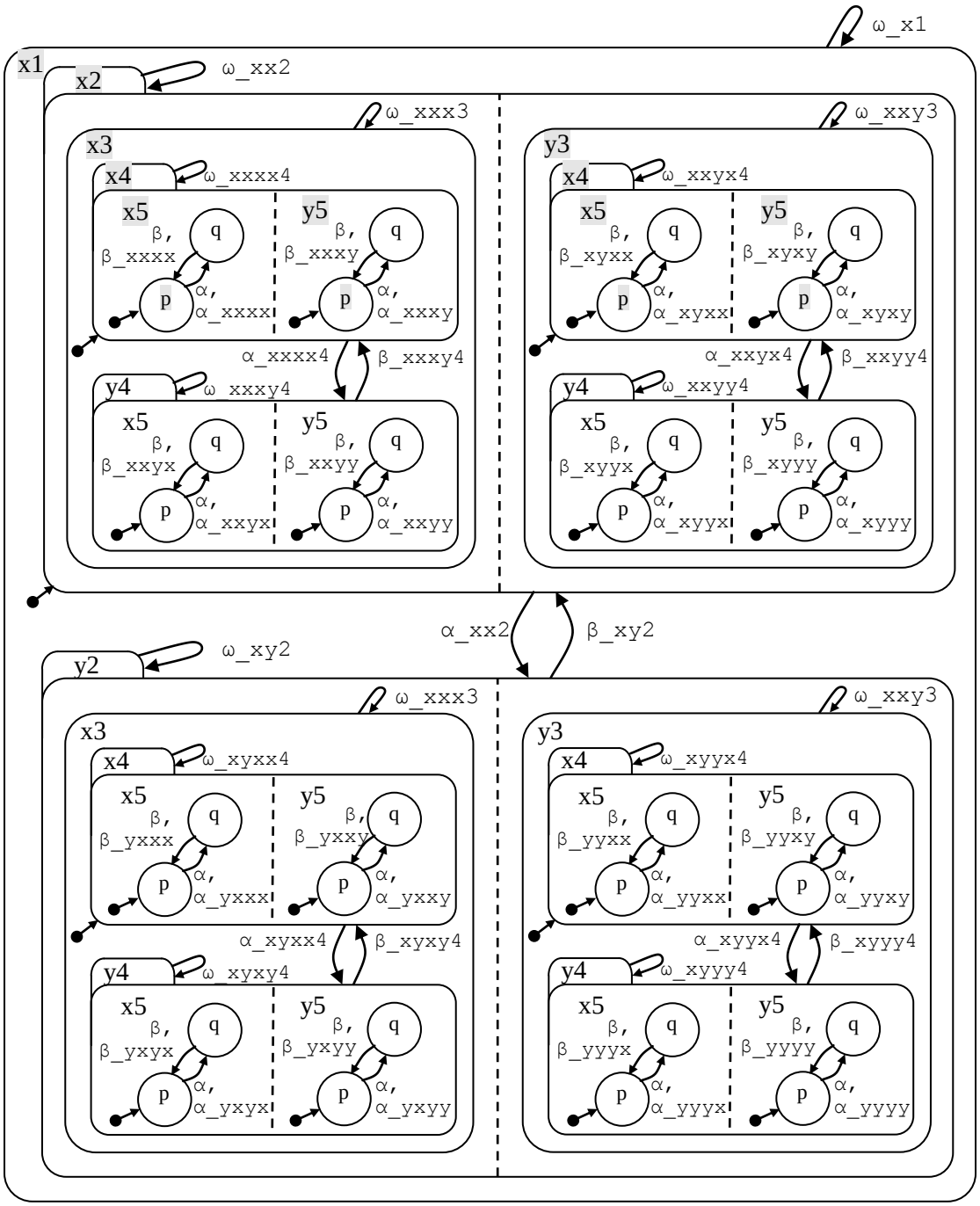


To generate this model: `mk_7140(5)`.

Table 10. Performance statistics for model `t7140`:

Model params	Event	PROLOG	Op. System	Processor speed	Perm $Pm^{set\text{-}tran}$	Perm Pm^{race}	Time or problem
(5)	α	SWI 5.0.3	Win98	300MHz	f_k3a	f_k3a	2m 55s
..	,ω_x1	..	..	..	..	..	global stack
..	,ω_xx2	..	..	..	..	..	1m 9s
..	α_xxxx	..	..	..	..	..	0.7s
..	,ω_x1	..	..	..	..	..	2m 21s
..	α	..	..	..	none	none	4.3 s
..	,ω_x1	..	..	..	..	..	1.6s
..	,ω_xx2	..	..	..	..	..	1.2s
..	α_xxxx	..	..	..	..	..	0.7s
..	,ω_x1	..	..	..	..	..	1.3s

Figure 80. Alternating sets and clusters [model `t7150`]



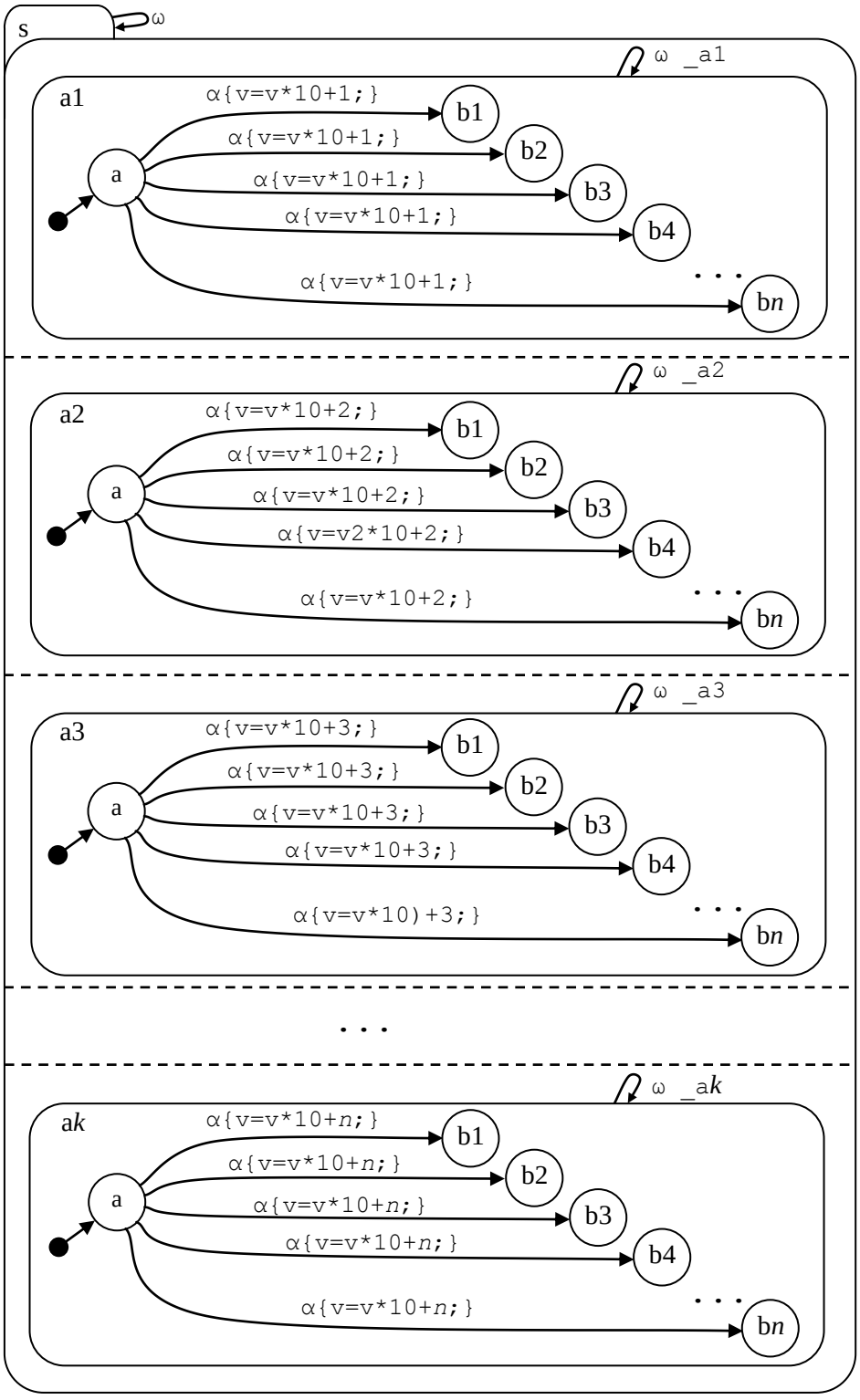
Initial states shaded for clarity.

To generate this model: **`mk_t7150(5).`** *(It is best to use an odd number).*

Table 11. Performance statistics for model `t7150`

Model params	Event	PROLOG	Op. System	Processor speed	Perm $Pm^{set\text{-}tran}$	Perm Pm^{race}	Time or problem
(5)	α	SWI 5.0.3	Win98	300MHz	f_k3a	f_k3a	2.5s
..	,ω_x1	..	..	..	..	..	0.9s
..	α_xx2	..	..	..	..	..	2.5s
..	,ω_x1	..	..	..	..	..	2.5s

Figure 81. Intensive fork and race non-determinism [model `t7160`]



To generate this model: **`mk_t7160(4,5).`** *//k=4, n=5*

Table 12. Number of worlds for model `t7160`

		For k≥3	For k=≤2
• Under f_1 (full) permutations	= n^k . *f_1*(k)	= n^k.(k!)	= n^k.(k!)
• Under f_k3a race-permutations	= n^k. *f_k3a*(k)	= n^k.(2k)	= n^k.(k!)
• Under f_k2 race-permutations	= n^k. *f_2*(k)	= n^k.(2)	= n^k.(k!)

(k,n)	n^k	*f_1*	*f_k3a*	*f_2*	full perm		f_k3a perm		f_k2 perm	
(1,1)	1^1=1	1!=1	1	1	1.1 =	1	1.1 =	1	1.1 =	1
(1,2)	2^1=2	1!=1	1	1	2.1 =	2	2.1 =	2	2.1 =	2
(1,3)	3^1=3	1!=1	1	1	3.1 =	3	3.1 =	3	3.1 =	3
(1,4)	4^1=4	1!=1	1	1	4.1 =	4	4.1 =	4	4.1 =	4
(1,5)	5^1=5	1!=1	1	1	5.1 =	5	5.1 =	5	5.1 =	5
(2,1)	1^2=1	2!=2	2	2	1.2 =	2	1.2 =	2	1.2 =	2
(2,2)	2^2=4	2!=2	2	2	4.2 =	8	4.2 =	8	4.2 =	8
(2,3)	3^2=9	2!=2	2	2	9.2 =	18	9.2 =	18	9.2 =	18
(2,4)	4^2=16	2!=2	2	2	16.2 =	32	16.2 =	32	16.2 =	32
(2,5)	5^2=25	2!=2	2	2	25.2 =	50	25.2 =	50	25.2 =	50
(3,1)	1^3=1	3!=6	6	2	1.6 =	6	1.6 =	6	1.2 =	2
(3,2)	2^3=8	3!=6	6	2	8.6 =	48	8.6 =	48	8.2 =	16
(3,3)	3^3=27	3!=6	6	2	27.6 =	162	27.6 =	162	27.2 =	54
(3,4)	4^3=64	3!=6	6	2	64.6 =	384	64.6 =	384	64.2 =	128
(3,5)	5^3=125	3!=6	6	2	125.6 =	750	125.6 =	750	125.2	250
(4,1)	1^4=1	4!=24	8	2	1.24 =	24	1.8 =	8	1.2 =	2
(4,2)	2^4=16	4!=24	8	2	16.24 =	384	16.8 =	128	16.2 =	32
(4,3)	3^4=81	4!=24	8	2	81.24 =	1944	81.8 =	648	81.2 =	162
(4,4)	4^4=256	4!=24	8	2	256.24 =	6144	256.8 =	512	256.2 =	512
(4,5)	5^4=625	4!=24	8	2	625.24 =	15000	625.8 =	5000	625.2 =	1250
(5,1)	1^5=1	5!=120	10	2	1.120 =	120	1.10 =	10	1.2 =	2
(5,2)	2^5=32	5!=120	10	2	32.120 =	3840	32.10 =	320	32.2 =	64
(5,3)	3^5=243	5!=120	10	2	243.120 =	29160	243.10 =	2430	243.2 =	486
(5,4)	4^5=1024	5!=120	10	2	1024.120=	122880	1024.10=	10240	1024.2=	2048
(5,5)	5^5=3125	5!=120	10	2	3125.120=	375000	3125.10=	31250	3125.2=	6250

Note that event ω does not clear variable data, and so does not reduce to one world. The actual number of worlds is Perm(k), using the selected restricted-permutation generator.

Table 13. Performance statistics for model `t7160`

[SWI-Prolog 5.0.3 / Windows98]:

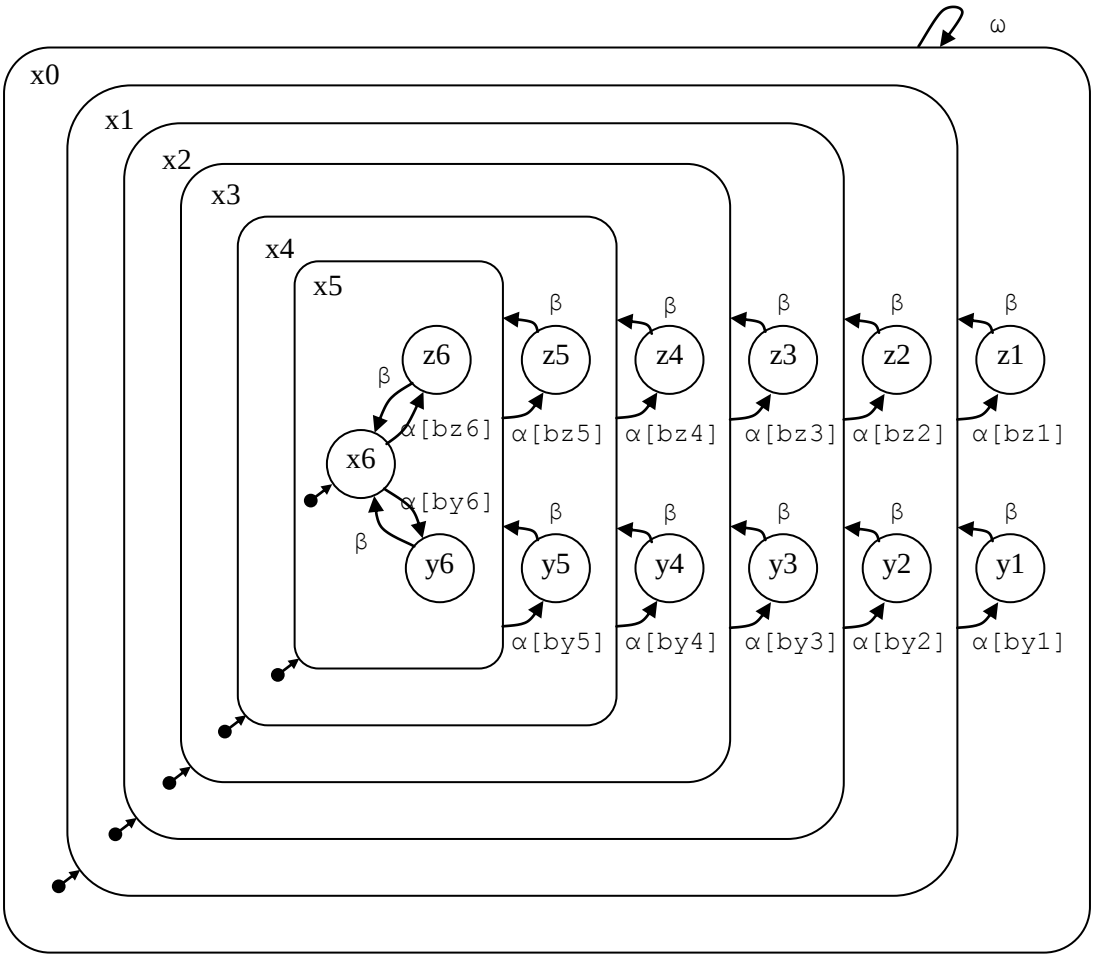
Model params	Event	Nr of Worlds	Processor speed	Perm $Pm^{set\text{-}tran}$	Perm Pm^{race}	Time or problem
(2,3)	α	1→18	300MHz	f_k3a	f_k3a	1.3s
(2,3)	,ω	18→2	..	..	..	0.8s
(2,4)	α	1→32	..	..	..	3.7s
(2,4)	,ω	32→2	..	..	..	1.8s
(3,2)	α	1→48	..	..	..	10.6s
(3,2)	,ω	48→6	..	..	..	10.0s
(3,3)	α	1→162	..	..	..	1m 53s
(3,3)	,ω	162→6	..	..	..	3m 15s
(3,4)	α	1→384	..	..	..	18m 3s
(3,4)	,ω	384→6	..	..	..	11m 0s
(4,2)	α	1→128	..	..	..	1m 59s
(4,2)	,ω	128→8	..	..	..	4m 02
(4,3)	α	1→648	..	..	..	*>1hr 30m*
(4,3)	,ω	648→8	..	..	..	*untested*

[WinProlog 4.010 / Win98]:

(3,4)	α	1→384	..	..	..	*>1hr*
(3,4)	,ω	384→6	..	..	..	*untested*

It was noted that uncompleted event processing, under SWI- and Win- Prolog, involved intense disk activity - it could be that with more core memory the events will complete in *much* less time.

Figure 82. Stressing transition prioritization [model `t7170`]

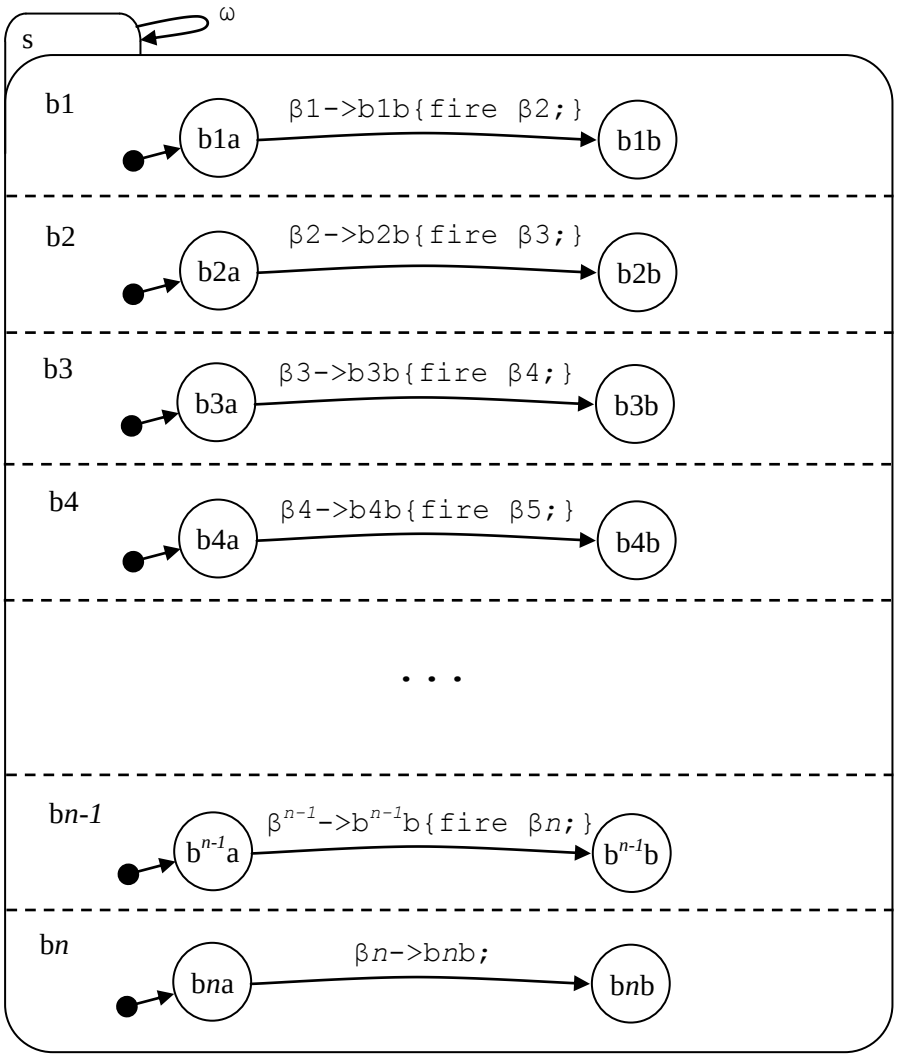


To generate this model : **`mk_t7170(5).`**

Table 14. Performance statistics for model `t7170` [SWI-Prolog 5.0.3 / Win98]:

Model params	Event	Worlds	Processor speed	Perm $Pm^{set\text{-}tran}$	Perm Pm^{race}	Time
(20)	α	1→2	300MHz	f_k3a	f_k3a	0.3s
..	β	2→1	..	..	..	0.3s

Figure 83. Long-chain broadcast-event non-determinism [model `t7180`]

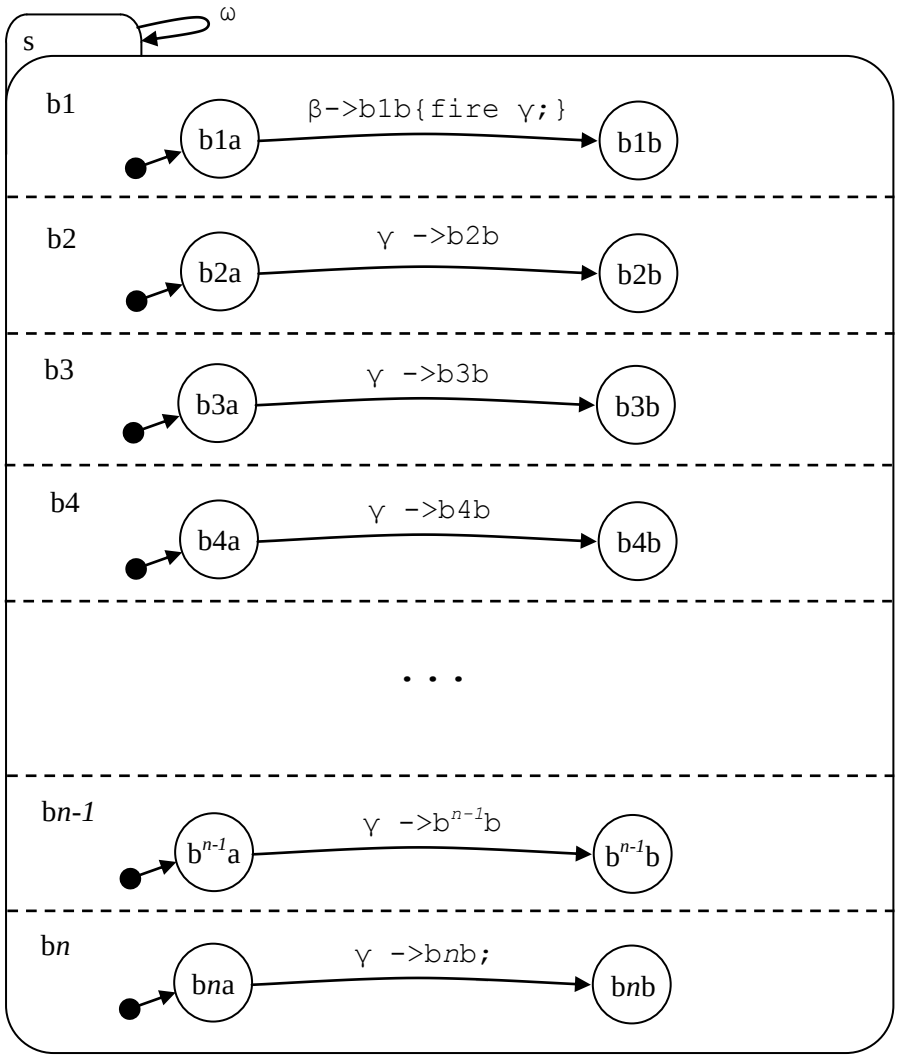


To generate model to b(20): **`mk_t7180(20).`**

Table 15. Performance statistics for model `t7180(20)`

Model params	Event	PROLOG	Op. System	Processor speed	Perm $Pm^{set\text{-}tran}$	Perm Pm^{race}	Time
(20)	β1	SWI 5.0.3	Win98	300MHz	f_k3a	f_k3a	1.0s
(20)	,ω	SWI 5.0.3	Win98	300MHz	f_k3a	f_k3a	16s

Figure 84. Broad broadcast-event nondeterminism [model `t7190`]



To generate this model: **`mk_t7190(20).`**

Table 16. Performance statistics for model `t7190(20)`

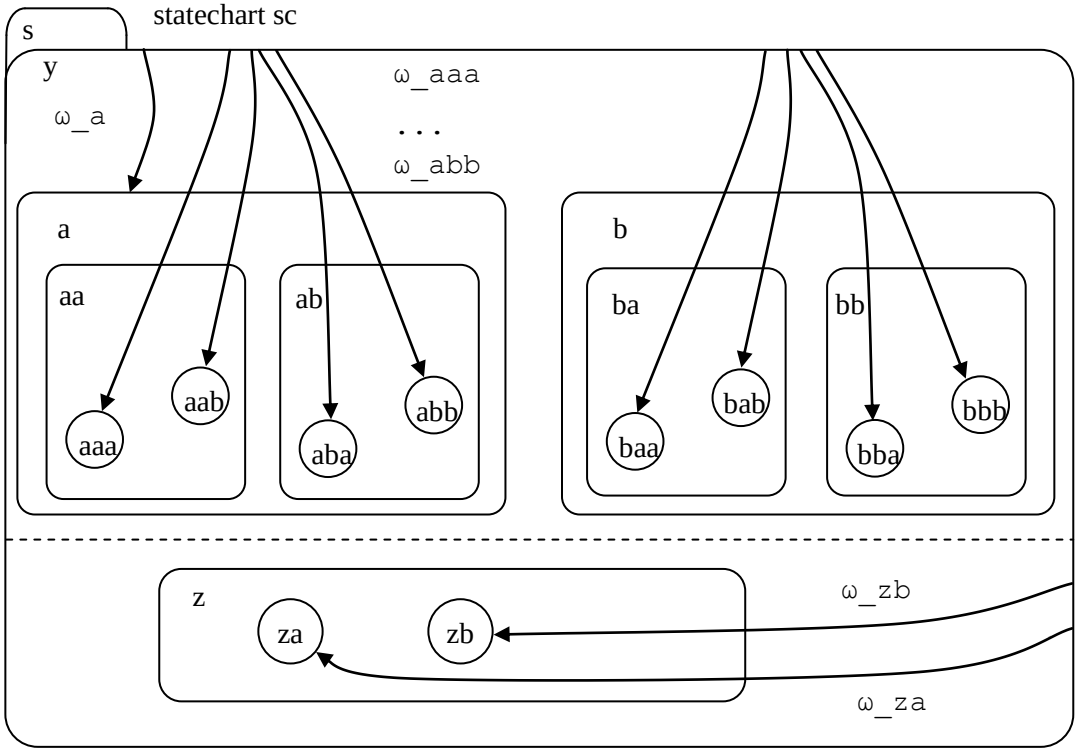
Model params	Event	PROLOG	Op. System	Processor speed	Perm $Pm^{set\text{-}tran}$	Perm Pm^{race}	Time
(20)	β	SWI 5.0.3	Win98	300MHz	f_k3a	f_k3a	1m50s
(20)	,ω	SWI 5.0.3	Win98	300MHz	f_k3a	f_k3a	23s

8. Conventions

8.1.1 Omega Transitions

Statecharts to exercise the state machine engine may contain implicit additional control transitions (named omega...) for the purpose of putting the statechart in a specific state (in particular leafstates) prior to a test.

Figure 85. Omega transitions



Notes

- When priming a model, omega transitions to leaf-states are all that is required. Omega transitions to non-leaf-states are only needed in testing entry to the correct default state. The above diagram shows just one omega-transition (`ω_a`) to a non-leaf-state.
- In order to be unique, the naming of an omega transition may have to incorporate the machine path (not necessary in the above model).
- The event `ω_vreset` is used to reset variables, and `ω_hreset` to clear history, triggering an internal transition at a high level in the hierarchy

9. STATECRUNCHER 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