A Quick Reference of STATECRUNCHER's Output Format

Graham G. Thomason

Appendix 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

Summary

This quick reference was written as a separate appendix to save repeating it in other appendices where STATECRUNCHER models and their output are presented. An appreciation of STATECRUNCHER's output format is particularly a pre-requisite for the following reports:

- The Distributed Arbiter System in CCS [StCrDistArb]
- The Dining Philosophers in CSP [StCrMain]
- The Game of Nim, specified in Z [StCrNim]

STATECRUNCHER was built for the purposes of providing an oracle to state-based tests. It forms part of a tool chain for *testing an implementation* of a system, i.e. for determining whether the implementation under test behaves according to its specified state behaviour, even when it is nondeterministic. STATECRUNCHER *does not generate tests*; it co-operates with a test generator in a tool chain.

Table of Contents

1. STATECRUNCHER's output

This paper serves as an explanation of STATECRUNCHER's output for the systems modeled in various appendices to the main thesis on STATECRUNCHER. We consider a model (Figure 1) which brings out the chief features of the output. The model also illustrates client-server interaction on event α, where member `a` is a client, firing event β to call the server (member `c`), which completes the interaction by firing event `return`..

To also illustrate the STATECRUNCHER language, the model is followed by its source code.

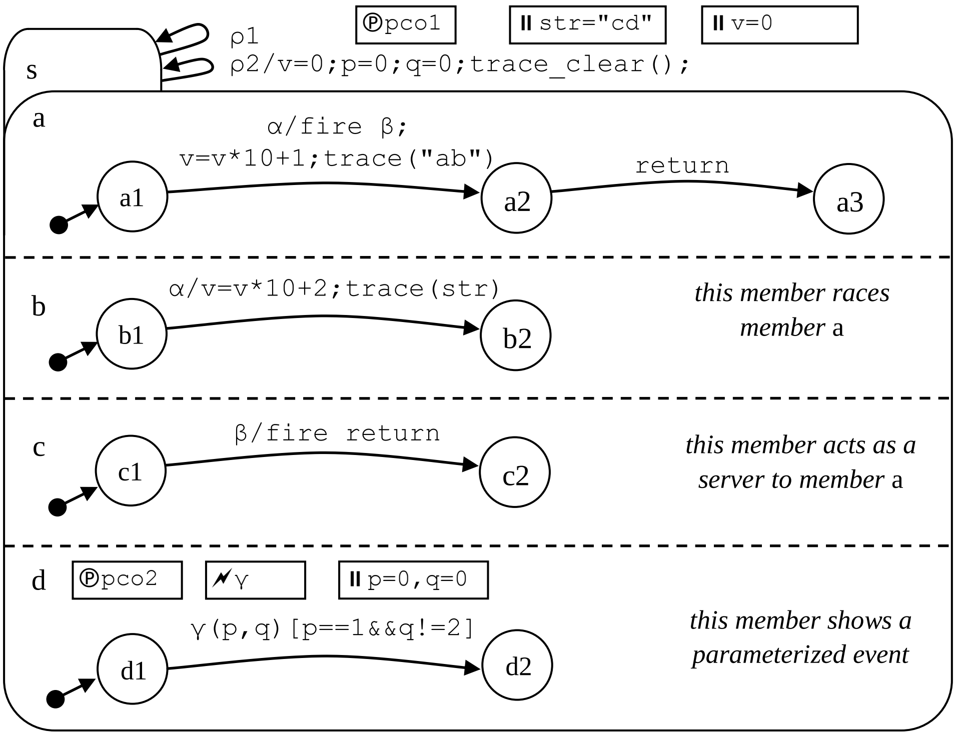

Figure 1. A model to illustrate the chief STATECRUNCHER output `[model t5492]`

Source code of the model `t5492`

```
//--------------------------------------------------------------------------------------
// Module:    all_kinds2.scs.txt
// Author:    Graham Thomason, Philips Digital Systems Laboratories, Redhill
// Date:      2 Aug, 2003
// Purpose:   Statecruncher model: Model to show all kinds of output (2)
//
// Project:   Improving Component Integration
//
// Copyright (C) 2003 Philips Electronics N.V.
//
// Revision History:
//
//-------1---------2---------3---------4---------5---------6---------7---------8-----

statechart sc(s)

PCO pco1;
PCO s.d.pco2;

event alpha;
event rho,rho1;
event beta,return@pco1;
event s.d.gamma@s.d.pco2;

enum int1 {0,..,9};
int1 v=0;

string str="cd";

set s(a,b,c,d)           {rho->s; rho1->s {v=0; d.p=0; d.q=0; trace_clear();}; }
   cluster a(a1,a2,a3)
      state a1           {alpha->a2 {fire beta; v=v*10+1; trace ("ab");}; }
      state a2           {return->a3;}
      state a3;
   cluster b(b1,b2)
      state b1           {alpha->b2 {v=v*10+2; trace(str);}; }
      state b2;
   cluster c(c1,c2)
      state c1           {beta->c2 {fire return;}; }
      state c2;
   cluster d(d1,d2)
      // PCO and events could be declared here, but are declared above
      enum int2 {red=0,orange,yellow,green=5,blue};
      enum int3 {0,..,3};
      int2 p=0;
      int3 q=0;

      state d1           {gamma($p,$q)[p==1 && q!=2]->d2; }
      state d2;
```

Session with model `t5492`

```
| ?- cruncher.
SC:|: mm
SC:|: run t5492
...
SC:|: gc
2    statechart sc
2       set s [sc] = OCC []  **
2          cluster a [s,sc] = OCC []  **
2             leafstate a1 [a,s,sc] = OCC []  **
2             leafstate a2 [a,s,sc] = VAC []
2             leafstate a3 [a,s,sc] = VAC []
2          cluster b [s,sc] = OCC []  **
2             leafstate b1 [b,s,sc] = OCC []  **
2             leafstate b2 [b,s,sc] = VAC []
2          cluster c [s,sc] = OCC []  **
2             leafstate c1 [c,s,sc] = OCC []  **
2             leafstate c2 [c,s,sc] = VAC []
2          cluster d [s,sc] = OCC []  **
2             leafstate d1 [d,s,sc] = OCC []  **
2             leafstate d2 [d,s,sc] = VAC []
2    VAR  INTEGER p [d,s,sc] =0
2    VAR  INTEGER q [d,s,sc] =0
2    VAR  STRING  str [sc] =[99,100] =cd
2    VAR  INTEGER v [sc] =0
2    TRACE =[]
2    TREV [[alpha,[sc]],0,[],[]]
2    TREV [[beta,[sc]],0,[],[pco1,[sc]]]
2    TREV [[gamma,[d,s,sc]],2,[[e,0,1,2,5,6],[r,0,3]],[pco2,[d,s,sc]]]
2    TREV [[rho,[sc]],0,[],[]]
2    TREV [[rho1,[sc]],0,[],[]]

outworlds=[2]
number of outworlds=1
SC:|: pe alpha
SC:|: gc
10   statechart sc
10      set s [sc] = OCC []  **
10         cluster a [s,sc] = OCC []  **
10            leafstate a1 [a,s,sc] = VAC []
10            leafstate a2 [a,s,sc] = VAC []
10            leafstate a3 [a,s,sc] = OCC []  **
10         cluster b [s,sc] = OCC []  **
10            leafstate b1 [b,s,sc] = VAC []
10            leafstate b2 [b,s,sc] = OCC []  **
10         cluster c [s,sc] = OCC []  **
10            leafstate c1 [c,s,sc] = VAC []
10            leafstate c2 [c,s,sc] = OCC []  **
10         cluster d [s,sc] = OCC []  **
10            leafstate d1 [d,s,sc] = OCC []  **
10            leafstate d2 [d,s,sc] = VAC []
10   VAR  INTEGER p [d,s,sc] =0
10   VAR  INTEGER q [d,s,sc] =0
10   VAR  STRING  str [sc] =[99,100] =cd
10   VAR  INTEGER v [sc] =12
10   TRACE =[cd,ab]
10   TREV [[gamma,[d,s,sc]],2,[[e,0,1,2,5,6],[r,0,3]],[pco2,[d,s,sc]]]
10   TREV [[rho,[sc]],0,[],[]]
10   TREV [[rho1,[sc]],0,[],[]]

18   statechart sc
18      set s [sc] = OCC []  **
```

```
18         cluster a [s,sc] = OCC []  **
18            leafstate a1 [a,s,sc] = VAC []
18            leafstate a2 [a,s,sc] = VAC []
18            leafstate a3 [a,s,sc] = OCC []  **
18         cluster b [s,sc] = OCC []  **
18            leafstate b1 [b,s,sc] = VAC []
18            leafstate b2 [b,s,sc] = OCC []  **
18         cluster c [s,sc] = OCC []  **
18            leafstate c1 [c,s,sc] = VAC []
18            leafstate c2 [c,s,sc] = OCC []  **
18         cluster d [s,sc] = OCC []  **
18            leafstate d1 [d,s,sc] = OCC []  **
18            leafstate d2 [d,s,sc] = VAC []
18   VAR  INTEGER p [d,s,sc] =0
18   VAR  INTEGER q [d,s,sc] =0
18   VAR  STRING  str [sc] =[99,100] =cd
18   VAR  INTEGER v [sc] =21
18   TRACE =[ab,cd]
18   TREV [[gamma,[d,s,sc]],2,[[e,0,1,2,5,6],[r,0,3]],[pco2,[d,s,sc]]]
18   TREV [[rho,[sc]],0,[],[]]
18   TREV [[rho1,[sc]],0,[],[]]

outworlds=[10,18]
number of outworlds=2
SC:|:
```

Explanation of the output

The state occupancy configuration is first shown (after command **gc**, get configuration). The lines

```
2             leafstate a1 [a,s,sc] = OCC []  **
2             leafstate a2 [a,s,sc] = VAC []
```

show that in world 2, (the initial world) leafstate `a1` is occupied (emphasized by asterisks) but `a2` is vacant. The item `[a,s,sc]` is the *scope* of these states, which is its place in the statechart hierarchy. Scopes are best read from right to left while descending in the hierarchy. The `[]` after the occupancies are placeholders for the historical state of vacant clusters (never applicable to leafstates, nor to clusters in this model).

Variables are shown in `VAR` lines, of the form:

```
WORLD    VAR  INTEGER|STRING VARIABLE-NAME VARIABLE-SCOPE =VALUE
```

In world 2 we have

```
2    VAR  INTEGER p [d,s,sc] =0
2    VAR  INTEGER q [d,s,sc] =0
2    VAR  STRING  str [sc] =[99,100] =cd
2    VAR  INTEGER v [sc] =0
```

String values are given in two ways: as a list of ASCII values and as characters for printable values.

A *trace* in STATECRUNCHER (unlike CCS/CSP) is a list of output values that have been specifically generated in the model by calling the `trace()` function. Trace values can be integers or strings. In world 2 the trace is empty:

```
2    TRACE =[]
```

Transitionable events are given by `TREV` lines. Consider the transitionable events from the initial model configuration:

```
2    TREV [[alpha,[sc]],0,[],[]]
2    TREV [[beta,[sc]],0,[],[pco1,[sc]]]
2    TREV [[gamma,[d,s,sc]],2,[[e,0,1,2,5,6],[r,0,3]],[pco2,[d,s,sc]]]
2    TREV [[rho,[sc]],0,[],[]]
2    TREV [[rho1,[sc]],0,[],[]]
```

The lines are of the form

```
WORLD TREV [[EVENT,EVENTSCOPE],NPARAMS,PARAM-RANGES,[PCO,PCOSCOPE]]
```

The events also have scope. The events `alpha`, `beta`, `rho` and `rho1` are in the default scope of the statechart: scope `[sc]`. But event `gamma` is in scope `[d,s,sc]`, which is deeper in the hierarchy.

Following the `[`*`EVENT`*`,`*`EVENTSCOPE`*`]` item is *`NPARAMS`*, the number of parameters that can be supplied with the event. In most cases this is none, but for `gamma` it is 2. The information following says that the first parameter can take on enumerated values of 0,1,2,5 or 6. The second parameter can be anything in the range 0 to 3 inclusive. Events taking no parameters have a `[]` for this item. The final item in a `TREV` line is the *`PCO`* (point of control and observation), or `[]` if none was specified in the model. PCOs too can have a scope.

It is also possible to ask STATECRUNCHER for *all* events, not just the transitionable ones (not shown here).

After event `alpha` has been processed (command **`pe alpha`**), there are two worlds, 10 and 18, due to race nondeterminism. Note how the trace values have been set and how the transitionable events have changed.

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