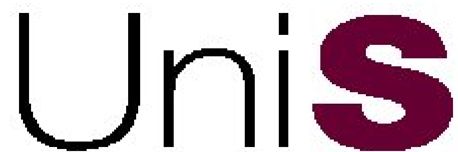
STATECRUNCHER-to-PRIMER Protocol

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

© Graham G. Thomason 2003-2004

Summary

Subject

STATECRUNCHER is a state transition language which, given a dynamic model of a system, provides an ***oracle*** to state based tests. In TorX terminology, STATECRUNCHER is an ‘explorer’. The ***choice of tests*** to perform is delegated to a separate tool, a ‘primer’. This report is concerned with the kinds of messages that will need to be passed between the explorer and primer. A primer and STATECRUNCHER, communicating with an agreed protocol, provide automatic generation of tests and their oracle.

 © Graham G. Thomason 2003-2004

Contents

1. Introduction ... 1
2. STATECRUNCHER and its main output ... 3
3. Primer command syntax ... 9
   3.1 Startup and prompt ... 9
   3.2 Inventory of commands ... 9
   3.3 Additional output ... 14
   3.4 Implementation consideration ... 15
   3.5 Grammar for PROLOG list structures as used in STATECRUNCHER ... 15
4. What the primer might do ... 19
   4.1 Primer requirements ... 19
   4.2 Examples of algorithms as requirements ... 20
5. Module organisation ... 24
6. Conclusions ... 28
7. References ... 29

© Graham G. Thomason 2003-2004

1. Introduction

State-based testing has been applied on many occasions within Philips, e.g. [Baker 95], [Yule 97], and has been very successful in finding defects in software.

Tools used have included State Relation Tables, used in the above examples, with Deja Gnu [DejaGnu] as a test harness. The technique has also been applied under Windows [Trew 98]. One drawback to state-relation tables is that they are hard to read and maintain; a more developer-friendly representation of state behaviour is needed.

Current work is aimed at having a developer- and test- friendly way of specifying state-behaviour, and automating test generation and execution to the highest degree. To this end a convenient state machine language has been developed (STATECRUNCHER) and the TorX tool chain architecture has been adopted from the Côte de Résyste project [CdR].

A complete tool chain for automated state-based testing is as follows:

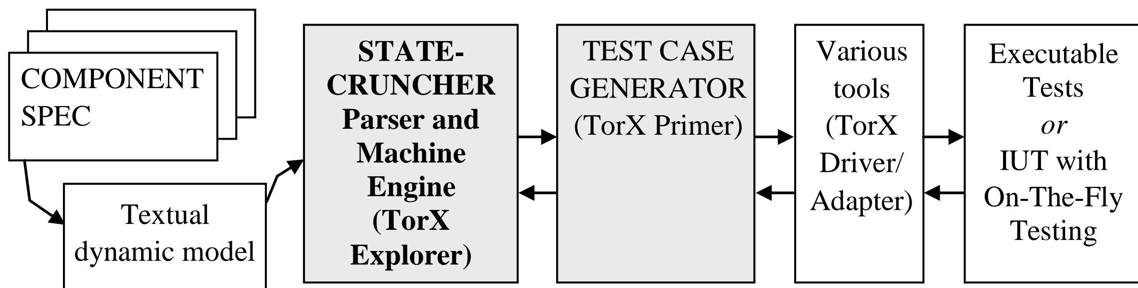


Figure 1. Tool chain for automated testing

This tool chain uses STATECRUNCHER as the oracle to the tests (the *explorer*). STATECRUNCHER has the capability to process *events* (as opposed to transitions) and generate a resultant state configuration and set of variable values. In the case of nondeterminism, STATECRUNCHER will generate all possible outcomes (new state configurations / variable values) in separate *worlds*.

For white-box testing, the state configuration and set of variable values obtained from the STATECRUNCHER will be compared to the IUT values. For black-box testing, both STATECRUNCHER and the IUT will (on at least some events) produce observable outputs called *traces*, which again will be compared per test.

© Graham G. Thomason 2003-2004

The subject of this report is the interface between STATECRUNCHER and the primer (test case generator), which will support both white- and black-box testing. The following dialogue is an informal example of a protocol between the STATECRUNCHER and the primer.

***Primer**:* Go into your initial state and tell me your state and the events to which you can respond.

***STATECRUNCHER**:* Done. Here is my state (including variable values) <...>. I can process events *alpha* and *gamma* (with parameters in the following ranges <...> at the following points of control and observation <...>).

***Primer**:* Process *alpha* and tell me your state and the events to which you can respond.

STATECRUNCHER: I now have 3 worlds (due to nondeterminism), which I will number 3,5,and 6.
World 3 has the following state <...> and can process the following events <...>.
World 5 has the following state <...> and can process the following events <...>.
World 6 has the following state <...> and can process the following events <...>.

Primer: Process event *delta* in all worlds and tell me your state and the events to which you can respond.

STATECRUNCHER: I now have the following worlds and can do the following <...>.

Primer: Go into the following states per world, with variable values indicated, <...> (it is a block of output you gave me earlier) and process event *epsilon*.

STATECRUNCHER: I now have the following worlds in which I can do the following <...>.

It is seen that the basic messages consist of:

- The ***Primer*** saying *process this event.*
- The ***Primer*** saying *go into this state (including variable values), per world.*
- ***STATECRUNCHER*** saying *what it has done* <u>and</u> *what* events it can respond to.

These messages will be examined in more detail in this report, and sketches of test generation algorithms will be presented.

© Graham G. Thomason 2003-2004

2. STATECRUNCHER and its main output

We have seen that STATECRUNCHER needs to be able to present what it has done, i.e. its current state in the broadest sense. Now the current state consists of the following information (which we will illustrate presently):

- The ***occupancy configuration***, i.e. for every hierarchical state, whether that state is occupied or vacant
- The ***historical state*** of clusters. This is recorded because there is an option when entering a cluster to enter the historical member (if known) rather than the default member or a specific member.
- ***Variable values***.
- ***Trace data***. Traces will be illustrated later.

In connection with future extensions, the following will also become part of the state in a broad sense:

- ***Object code*** (not only the compiled statements as such, but also including the derived *symbol table* and perhaps the *cross-reference table*[1]). Object code output will be needed if some projected extensions to STATECRUNCHER are implemented, enabling dynamic 'implantations' of local states, events and variables etc[2]. recursively, so the structure of the hierarchical states may not be fixed.

STATECRUNCHER has the concept of ***worlds*** to represent different nondeterministic outcomes. All the above items will be output ***per world***.

It should be noted that state and variable names also have a ***scope***, and the local name need not be unique. So a variable might be represented by `v1 [bb,aa,sc]` where the variable name is `v1` and it is in a scope defined by entering hierarchical members `sc` (the statechart name) then `aa` (which must be a cluster or set) then `bb` (which must be a cluster, set or leafstate).

[1] The cross-reference table is currently only used in validating to find unreferenced items, and is not used by the run-time engine. However, if a forward-chaining module is added in the future, the cross-reference might be used to efficiently provide access to relevant nodes that are consequent upon any one item of data. Such nodes will reference that data on their right-hand-side.

[2] Also: tagnames (enum declarations), PCOs (points of control and observation), if these are declared in an implantable machine.

© Graham G. Thomason 2003-2004

STATECRUNCHER will also supply what to it is derived data:

- The ***transitionable events***, including how many parameters they take, what the range of values of the parameters is, and at which PCO (point of control and observation) the event can be supplied.
- Some ***summary information*** about what worlds are in existence.

STATECRUNCHER can also supply additional information such as

- all events (not just the transitionable ones)
- time and date

Section gives the 3.2 full repertoire of commands.

We next illustrate STATECRUNCHER's output using a demonstration model that exercises all kinds of output.

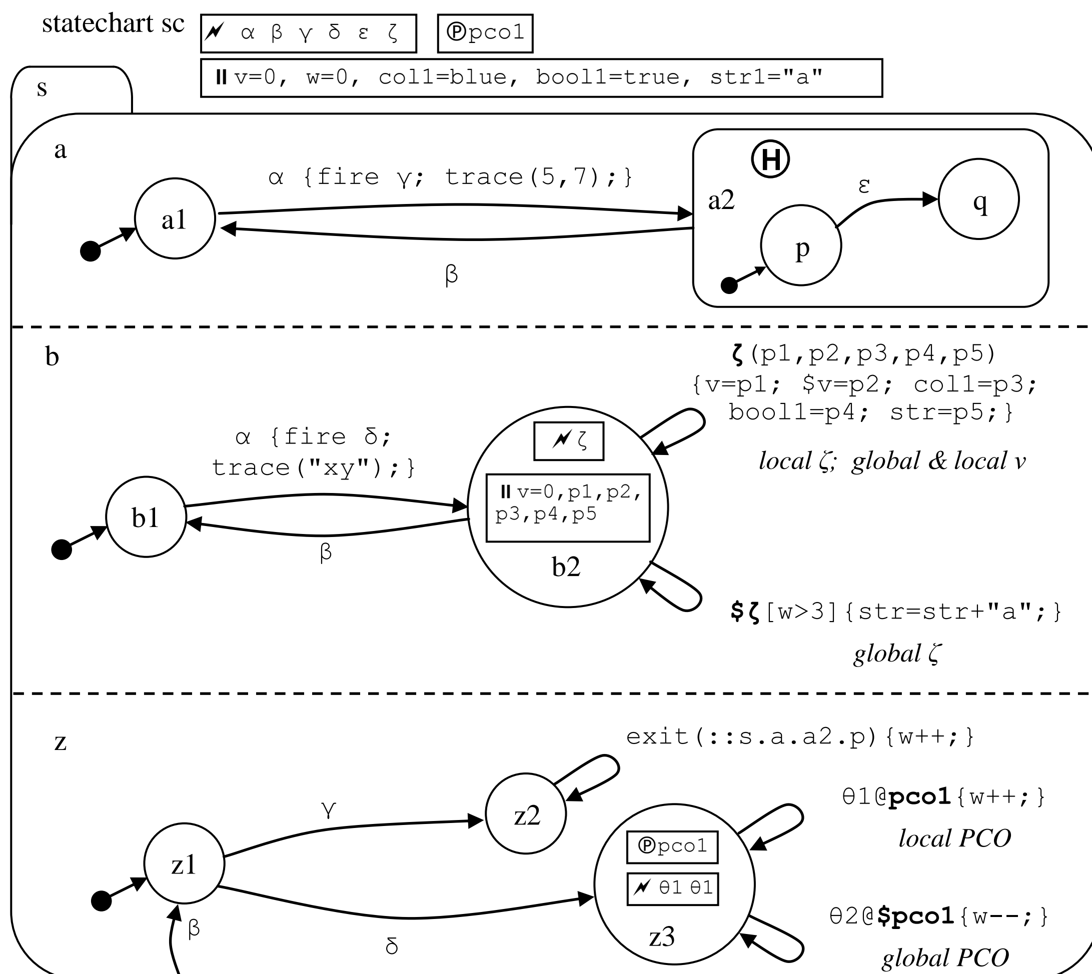


Figure 2. Illustration of all kinds of STATECRUNCHER output [Model t5490]

© Graham G. Thomason 2003-2004

Notes

- This model includes 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`.
- The model illustrates scoped events **ζ** and **$ζ**
- The model illustrates scoped variables **`v`** and **`$v`**
- The model illustrates scoped PCOs **`pco1`** and **`$pco1`**
- 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.
- Internally generated events, in our example, `exit(::a.a2.p)` are *not* offered as user suppliable.

In the output lists that follow, long lines have been re-formatted into two lines for the purpose of this report. Such cases are clear because of the lack of a leading world number on the continuation line.

The initial output for this model, ***from Release 1.03***, after entering it, is as follows.

```
2    statechart sc
2       set s [sc] = OCC []  **
2          cluster a [s,sc] = OCC []  **
2             leafstate a1 [a,s,sc] = OCC []  **
2             cluster a2 [a,s,sc] = VAC []
2                leafstate p [a2,a,s,sc] = VAC []
2                leafstate q [a2,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 z [s,sc] = OCC []  **
2             leafstate z1 [z,s,sc] = OCC []  **
2             leafstate z2 [z,s,sc] = VAC []
2             leafstate z3 [z,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  INTEGER p2 [b2,b,s,sc] =unknown
2    VAR  INTEGER p3 [b2,b,s,sc] =unknown
2    VAR  INTEGER p4 [b2,b,s,sc] =unknown
2    VAR  STRING  p5 [b2,b,s,sc] =unknown
2    VAR  STRING  str [sc] =[98] =b
2    VAR  INTEGER v [b2,b,s,sc] =0
2    VAR  INTEGER v [sc] =0
2    VAR  INTEGER w [sc] =0
2    TRACE =[]
2    TREV [[alpha,[sc]],0,[],[]]
2    TREV [[gamma,[sc]],0,[],[]]
2    TREV [[delta,[sc]],0,[],[]]
2    TREV [[beta,[sc]],0,[],[]]

outworlds=[2]
number of outworlds=1
```

This output is as produced by WinProlog. SWI Prolog adds a space after a comma in a list.

This output shows an active world, in this case world 2, which is the initial world after having been entered (driven into its initial state). The output shows:

- ***States and their occupancies***. The line
```
    2          cluster a [s,sc] = OCC []  **
```
  shows that cluster `a` in scope `[s,sc]` is occupied. This is shown both by the `OCC` indication and the double asterisk. The line also shows that no history (`[]`) has been recorded for this cluster yet (as it has not yet been exited).

- ***Variable data***. The line
```
    2    VAR  INTEGER v [b2,b,s,sc] =0
```
  shows that there is a variable `v` of some integral type in scope `[b2,b,s,sc]` has a value of 0. Details of its allowable range are not given here, but are available by means of a separate command. Note amongst the other variables a string (`str`) with its value as a list of ASCII values (here just `[98]`, the code for "b").

- ***Traces***. The line
```
    2    TRACE =[]
```
  is used to reproduce any trace information traced in a `trace(...)` call (which is allowable in any action). Here there is none, so we see the empty list `[]`.

- ***Transitionable events***. The line
```
    2    TREV [[alpha,[sc]],0,[],[]]
```
  indicates that an event `alpha` can be processed, and that no parameters can be supplied, so the parameter range element is empty (`[]`). The final `[]` indicates that no specific PCO (point of control and observation) is associated with this event. The current version of STATECRUNCHER does not attempt to predict what any transition conditions will evaluate to, and lists all transitions on user-suppliable events from the current configuration.

We now process event `alpha` and examine the output:

```
7    statechart sc
7       set s [sc] = OCC []  **
7          cluster a [s,sc] = OCC []  **
7             leafstate a1 [a,s,sc] = VAC []
7             cluster a2 [a,s,sc] = OCC []  **
7                leafstate p [a2,a,s,sc] = OCC []  **
7                leafstate q [a2,a,s,sc] = VAC []
7          cluster b [s,sc] = OCC []  **
7             leafstate b1 [b,s,sc] = VAC []
7             leafstate b2 [b,s,sc] = OCC []  **
7          cluster z [s,sc] = OCC []  **
7             leafstate z1 [z,s,sc] = VAC []
7             leafstate z2 [z,s,sc] = OCC []  **
7             leafstate z3 [z,s,sc] = VAC []
7    VAR  INTEGER bool1 [sc] =1
7    VAR  INTEGER col1 [sc] =8
7    VAR  INTEGER p1 [b2,b,s,sc] =unknown
7    VAR  INTEGER p2 [b2,b,s,sc] =unknown
7    VAR  INTEGER p3 [b2,b,s,sc] =unknown
7    VAR  INTEGER p4 [b2,b,s,sc] =unknown
7    VAR  STRING  p5 [b2,b,s,sc] =unknown
7    VAR  STRING  str [sc] =[98] =b
```

 © Graham G. Thomason 2003-2004

```
7    VAR  INTEGER v [b2,b,s,sc] =0
7    VAR  INTEGER v [sc] =0
7    VAR  INTEGER w [sc] =0
7    TRACE =[xy,7,5]
7    TREV [[epsilon,[sc]],0,[],[]]
7    TREV [[beta,[sc]],0,[],[]]
7    TREV [[zeta,[b2,b,s,sc]],5,[[r,0,9],[r,0,9],[e,0,7,8,4,8],[r,0,1],
[<string>]],[]]
7    TREV [[zeta,[sc]],0,[],[]]

12   statechart sc
12      set s [sc] = OCC []  **
12         cluster a [s,sc] = OCC []  **
12            leafstate a1 [a,s,sc] = VAC []
12            cluster a2 [a,s,sc] = OCC []  **
12               leafstate p [a2,a,s,sc] = OCC []  **
12               leafstate q [a2,a,s,sc] = VAC []
12         cluster b [s,sc] = OCC []  **
12            leafstate b1 [b,s,sc] = VAC []
12            leafstate b2 [b,s,sc] = OCC []  **
12         cluster z [s,sc] = OCC []  **
12            leafstate z1 [z,s,sc] = VAC []
12            leafstate z2 [z,s,sc] = VAC []
12            leafstate z3 [z,s,sc] = OCC []  **
12   VAR  INTEGER bool1 [sc] =1
12   VAR  INTEGER col1 [sc] =8
12   VAR  INTEGER p1 [b2,b,s,sc] =unknown
12   VAR  INTEGER p2 [b2,b,s,sc] =unknown
12   VAR  INTEGER p3 [b2,b,s,sc] =unknown
12   VAR  INTEGER p4 [b2,b,s,sc] =unknown
12   VAR  STRING  p5 [b2,b,s,sc] =unknown
12   VAR  STRING  str [sc] =[98] =b
12   VAR  INTEGER v [b2,b,s,sc] =0
12   VAR  INTEGER v [sc] =0
12   VAR  INTEGER w [sc] =0
12   TRACE =[7,5,xy]
12   TREV [[epsilon,[sc]],0,[],[]]
12   TREV [[beta,[sc]],0,[],[]]
12   TREV [[zeta,[b2,b,s,sc]],5,[[r,0,9],[r,0,9],[e,0,7,8,4,8],[r,0,1],
[<string>]],[]]
12   TREV [[zeta,[sc]],0,[],[]]
12   TREV [[theta1,[z3,z,s,sc]],0,[],[pco1,[z3,z,s,sc]]]
12   TREV [[theta2,[z3,z,s,sc]],0,[],[pco1,[sc]]]

outworlds=[7,12]
number of outworlds=2
```

The race between the two transitions on event `alpha` has two outcomes depending on the 'winner'. STATECRUNCHER takes both orderings, and so produces two worlds.

We remark on certain output lines:

- The line
  ```
  7    TRACE =[xy,7,5]
  ```
  shows trace information `[xy,7,5]` deposited, to be read from right to left. The order corresponds to the fact that in this world, the transition on event `alpha` from state `a1` (depositing trace integer values 5 and 7) preceded the transition from state `b1` (depositing trace string "xy").

- The following lines relate to transitions from state `b2`:

```
7     TREV [[zeta,[b2,b,s,sc]],5,[[r,0,9],[r,0,9],[e,0,7,8,4,8],
        [r,0,1],[<string>]],[]]
7     TREV [[zeta,[sc]],0,[],[]]
```

  They show two events, both called `zeta`, but in different scopes – a local one and a global one. Note that scopes are read from right to left (if interpreted from outermost in the hierarchy of states to the innermost). Note that `[zeta,[b2,b,s,sc]]` takes five parameters. Ranges are given by `[r,`*`LOWVALUE`*`,`*`HIGHVALUE`*`]`. Enumerations are given by `[e,`*`VALUE`*`,`*`VALUE`*`,`*`VALUE`*`...]`. Booleans are given as a range, `[r,0,1]`. Strings are represented by `[<string>]`.

- Note that world 7 is in state `z2`, but it does not list the event `exit(::s.a.a2.p)` as a transitionable event, since it is not user-suppliable.

- The following lines show events that are attached to PCOs (points of control and observation). The two PCOs have the same name, but are different, because they have different scopes.

```
12    TREV [[theta1,[z3,z,s,sc]],0,[],[pco1,[z3,z,s,sc]]]
12    TREV [[theta2,[z3,z,s,sc]],0,[],[pco1,[sc]]]
```

- If we now process events `epsilon` and `beta`, we will be able to illustrate a historical state.

```
19              cluster a2 [a,s,sc] = VAC q
```

  The state `q` is the historical state of cluster `a2`. It is entered on processing `alpha` again.

```
72                  leafstate q [a2,a,s,sc] = OCC []  **
```

Additional commands will cause additional information to be provided. There are also commands for offering an event for processing and for other functions. These are shown in section 3.

The data that STATECRUNCHER could supply if need be is not limited to the above examples. As already mentioned, it could as a future option supply dynamic 'object code' and related items if necessary.

© Graham G. Thomason 2003-2004

3. Primer command syntax

3.1 Startup and prompt

Startup of the executable version of STATECRUNCHER

On starting up the executable version of STATECRUNCHER, and also after all commands have been processed, the prompt is

```
SC:
```

Under WinProlog, an addition is made to this prompt, and the effective prompt is

```
SC:|:
```

All commands must be on one line.

Startup of the development version of STATECRUNCHER

Under the development version of STATECRUNCHER, the goal

```
statecruncher.
```

or just

```
cruncher.
```

will cause the same startup prompt to be given in the same read-process loop. These predicates are informal ones (in the `aux_load_sc.pl` file) calling the formal predicate

```
cs_read_process.    /* The STATECRUNCHER-PRIMER loop */
```

in file `cs_sc_1.pl`.

3.2 Inventory of commands

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.

Meta-syntax: 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 (/).

© Graham G. Thomason 2003-2004

The important commands that were not possible in previous releases of STATECRUNCHER 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 *showing typical example and/or typical output*

Main processing: high priority black box testing commands

pe …	**process event** *EVENT ?***p=***PARAMETERS ?***t=***EXPECTEDTRACE*
	`pe gamma         p=[4,xy]` *(statechart scope assumed)*
	`pe [alpha,[sc]] p=1`
	`pe [alpha,[sc]]`
	Parameters can also be supplied in STATECRUNCHER internal form, e.g.
	`p=[[ex_co,int,4],[ex_str,[120,121]]]`
	Worlds in direct violation of EXPECTEDTRACE will be killed, but overtrace and undertrace are tolerated.

gt	**get trace** `7   TRACE =[1,2]`
ct	**clear trace** *(this also causes a world merge)*

Main processing: medium priority commands

gae	**get all events** *(whether transitionable or not; not world-related)* `EVENT [theta2, [z3,z,s,sc]] [pco1,[z,s,sc]]`

gate	**get all transitionable events**
	(union from all worlds; no worlds shown)
	`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,[],[]]`

© Graham G. Thomason 2003-2004

gav	**get all variables** *Gets the value-ranges, not the current value per world* 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]
gaw	**get all worlds** *Gets the current worlds* [2,7,8]
gc	**get config** 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>]], [pco1,[z3,z,s,sc]]] outworlds=[2,4] number of outworlds=2
gst	**get symbol table** SYMB delta [sc] eventdecl [] XREF leafstate b1:[b, s, sc] XREF leafstate z1:[z, s, sc]
kill ...	**kill** ***WORLD / WORLDS*** kill 2 kill [2,7,10]
→	***WORLD*** **TRACE** **=** ***TRACE*** *input is as the output of* get config *this does* ***not*** *cause a world merge* *(we will probably issue this kind of command several times before requiring a world merge)*
→	***WORLD STATEKIND STATENAME MPATH = OCCUPANCY HISTORY*** *input is as the output of* get config *this does* ***not*** *cause a world merge (we will probably change more)*
→	***WORLD*** **VAR** ***VARKIND VARIABLENAME MPATH = VALUE*** *input is as the output of* get config *this does* ***not*** *cause a world merge (c.f. WORLD* TRACE *= TRACE)*

cnw	**create new world** *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)*
mw	**merge worlds** *(useful when all trace/state/variable changes have been made)*
gpt	**get processing time** *(timing data is set on processing an event)* exec time=00h 00m 00s 210ms
gd	**get date** *(get date and time)* DATE: 24 Apr 2003 16:01:40/649

Containment of combinatorial explosion: low priority commands

These commands limit the number of permutations used in set transit nondeterminism and race nondeterminism. See [StCrMain] for more explanation.

nst	**no set tran**
lst	**low set tran**
mst	**medium set tran**
hst	**high set tran**
nr	**no race**
lr	**low race**
mr	**medium race**
hr	**high race**

Compilation, loading, start-up, and finish: very low priority

root …	**root** ***ROOTDIRECTORY*** *Sets the root directory to be used with FILENAMEs*
mm	**mode modelnames** *Sets compilation etc. to work with model names. The directory structure must be set up correctly.*
mf	**mode filenames** *(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).*
cp …	**compile** ***FILENAME / MODELNAME*** *(also loads machine, and enters it (as of Rel 1.05))*
ld …	**load** ***FILENAME / MODELNAME*** *(does not enter machine)*
run …	**run** ***FILENAME / MODELNAME*** *=Load and enter machine*
nm	**enter machine** *Machine enters default state*

© Graham G. Thomason 2003-2004

xm	**exit machine** *Leaves a pristine machine ready to be entered*
um	**unload machine** *Removes data and object code*
rm	**reset machine** *=exit and enter*
quit	**quit**

System/diagnostic: very low priority

help	**help**
prolog	**prolog** *Gives a Prolog prompt; enter a Prolog goal*

Table 1. 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 nonexistent 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*

© Graham G. Thomason 2003-2004

Error and warning messages

These 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 2. Error and warning messages

3.3 Additional output

The Prolog system may produce extra output as it loads and abolishes Prolog modules, when a model is loaded or cleared. Typical lines from WinProlog are

```
About to consult
P:\Kwinpro\StCr\StCr2Sand\..\StCr3ModelsTest\t5000me\t5490_all
_kinds_of_output\all_kinds_of_output.sco.pl

# 0.047 seconds to consult
p:\kwinpro\stcr\stcr3modelstest\t5000me\t5490_all_kinds_of_out
put\all_kinds_of_output.sco.pl

# Abolishing
p:\kwinpro\stcr\stcr3modelstest\t5000me\t5490_all_kinds_of_out
put\all_kinds_of_output.sco.pl

# Removing 0 clauses for multifile predicate oc_errorcount / 1
```

Such lines can be ignored by a Primer.

© Graham G. Thomason 2003-2004

3.4 Implementation consideration

A typical command is

```
2    VAR  p1 [b2, B, s, sc] =unknown
```

The items in this line could be regarded as PROLOG terms – they were output as such by STATECRUNCHER.

Question: When reading/parsing commands, do we read PROLOG terms directly, or do we read at a character level and parse?
Answer: We read at a character level and parse, even when reading e.g. PROLOG lists. As we parse, we reconstruct the PROLOG list. The reasons why we must do this are:

- All PROLOG items read with `read()` must be terminated by a dot.
- All capitalised atoms read with `read()` require quoting.

This would all make the input cumbersome, or it would need pre-processing to:

```
2.    'VAR'.  p1. [b2, 'B', s, sc]. '='. unknown.
```

3.5 Grammar for PROLOG list structures as used in STATECRUNCHER

All parses consists of identifiers, integers, particular characters (such as “=”) and lists-of-a-restricted-kind, which we call s_lists. The s_lists contain identifiers, integers, or nested s_lists. They may also be the empty list.

The *parse* of ASCII items in the following syntax descriptions is *the PROLOG item itself*. Identifiers beginning with capital letters are ground items (as if quoted on entry), not variables.

In order for this restricted grammar to work, ***traces must be restricted to identifiers and integers.*** The user must take responsibility for this at present.

Syntax diagrams for the constituent terms needed by commands follow; they correspond very closely to the Prolog Definite Clause Grammar implementation.

© Graham G. Thomason 2003-2004

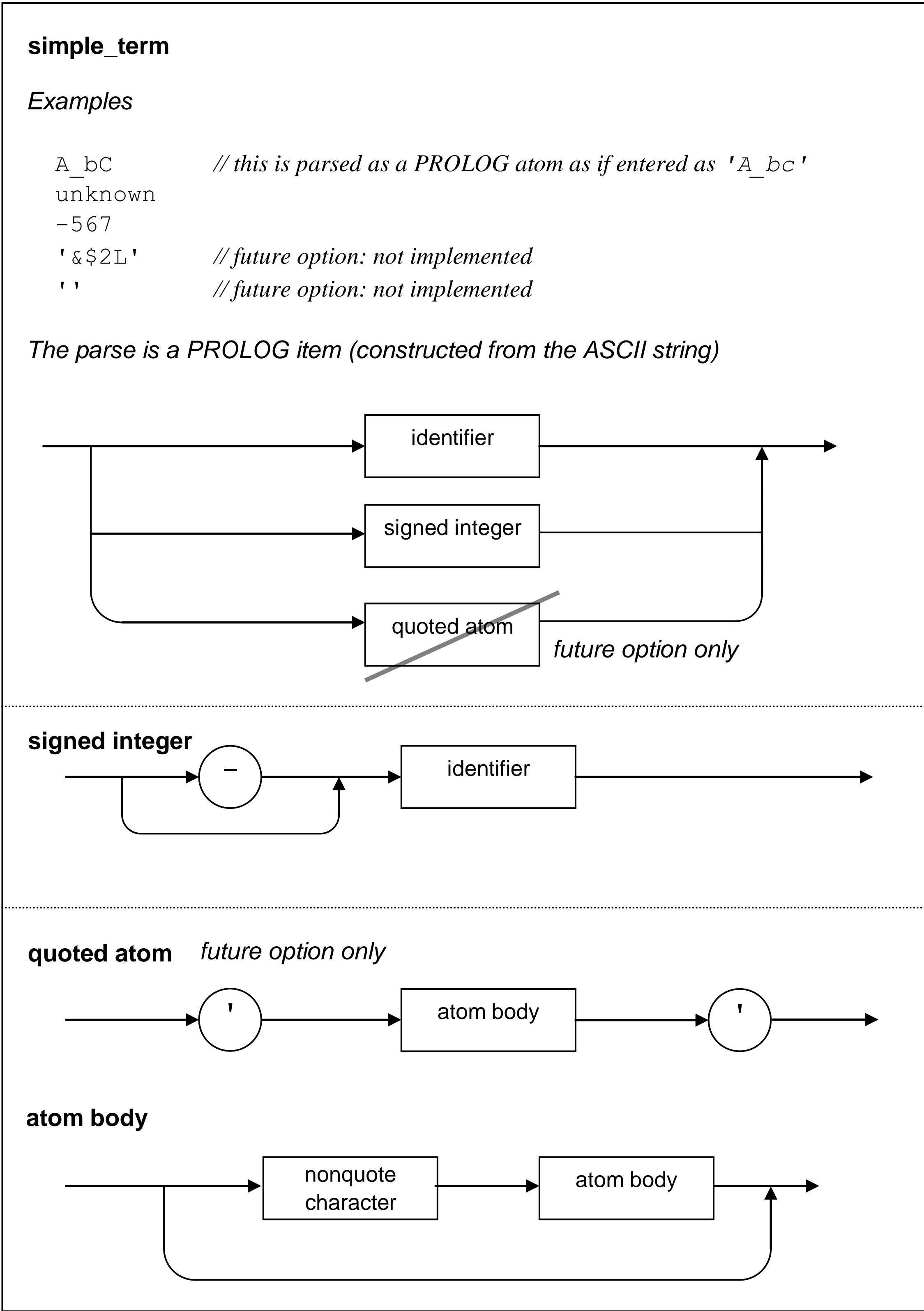


Figure 3. simple_term

© Graham G. Thomason 2003-2004

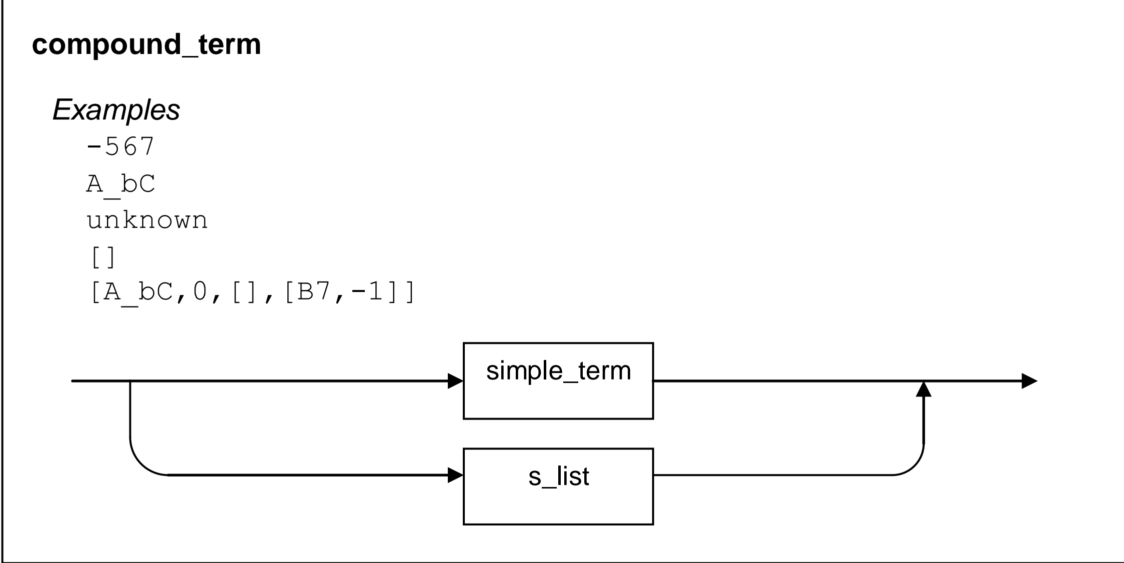


Figure 4. compound_term

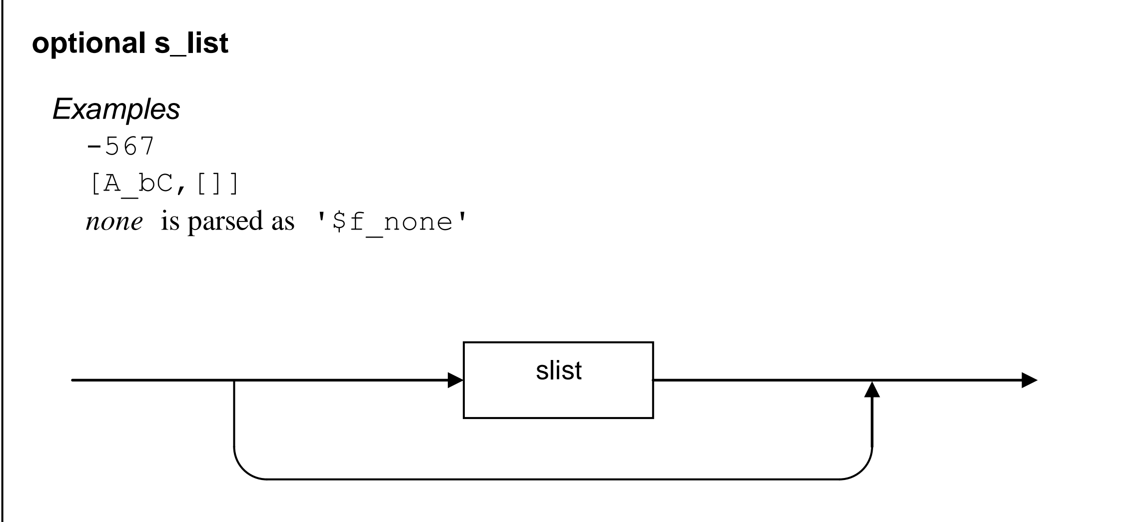


Figure 5. optional s_list

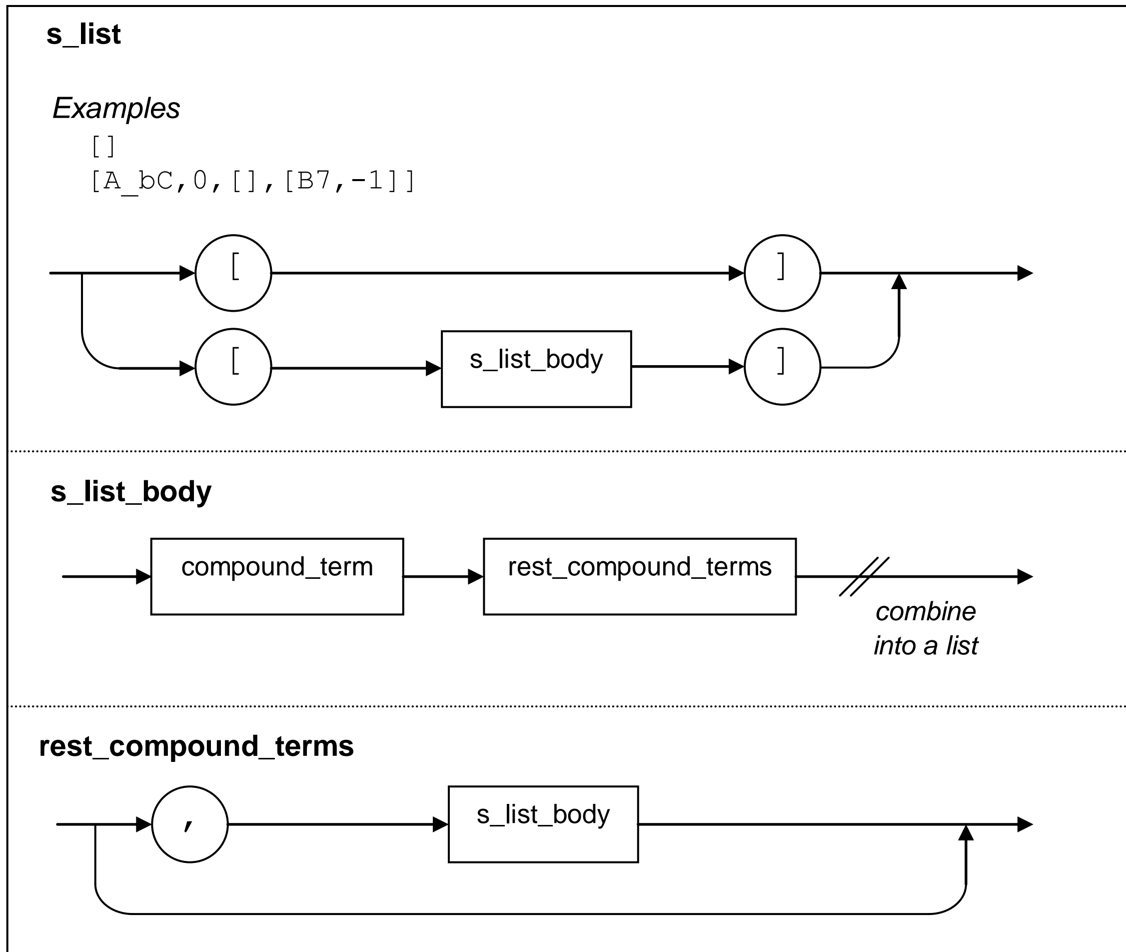


Figure 6. s_list

© Graham G. Thomason 2003-2004

4. What the primer might do

4.1 Primer requirements

STATECRUNCHER Release 1.03 is an independent executable program (so not requiring any implementation-language environment to be pre-loaded, such as a Prolog system). The Primer will typically be a separate executable program in a language such as Prolog, C++, TCL or Perl, with the ability to store and retrieve large amounts of data (since for some tasks it will be storing many state configurations), and select specific items from that data.

Messages to and from STATECRUNCHER are supplied via standard input and standard output via a pipe. If a socket connection is required, a simple standard_io-to-socket program can be inserted in the chain.

The primer should be able to do the following:

- ***Explore*** (i.e. find, reach, and enter) every state in the entire state space. This will involve marking states as visited and returning to previous state configurations.

- Support constructions for ***white box testing***, in which we ***exercise every event***, if it gives rise to a transition, from ***every state*** with ***every value*** assigned to ***every variable***.

- Support constructions for ***black-box testing***, where trace data is analysed, and conformity of the IUT is proved (or otherwise) by generating event sequences whereby expected trace output will disambiguate and uniquely identify the original target state.

- Include commands to ***output*** information about what it has decided to do with the results of transitions.
  - It should be possible to produce a test script
  - It should be possible to do on-the-fly testing and logging in a test report

- Implement any ***pruning algorithm*** e.g. one devised to explore a useful subspace of the total space.

- Where appropriate, the primer will make use of PCO (point of control and observation) information, so as to support testing of distributed systems. Even where the system is not distributed, it may be that not all events can be generated by the test harness, so PCOs will have a wider use.

4.2 Examples of algorithms as requirements

Note: These algorithms are in pseudo-code, not in STATECRUNCHER's command language, or in any other language itself.

4.2.1 Exploration algorithms

We start with the algorithm given as specified in [Raptis 99]. The pseudo-code below is basically as specified in that report, reworded slightly, but with the significant change that we process *events*, not *transitions*. This is a depth first algorithm to find and reach every state in the state space. The basic algorithm ignores data value issues. We extend the algorithm to handle data and non-determinism subsequently.

Exploration algorithm [deterministic, just events]

```
Enter INITIAL STATE
Push INITIAL STATE on stack
While stack not empty
   Pop SOURCE STATE off stack
   Do next observable EVENT -> TARGET STATE
   If there are unprocessed events from SOURCE STATE
      Push SOURCE STATE on stack
   If TARGET STATE unvisited
      Push TARGET STATE on stack
      Mark TARGET STATE as visited
```

 © Graham G. Thomason 2003-2004

Equivalent exploration algorithm [Alg-1: deterministic, just events]

Here we use recursion rather than stack operations

```
EXPLORE(INITIAL STATE)

PROC EXPLORE(STATE)
Mark STATE as visited
For each EVENT                        // all from same current state, so ...
   Do EVENT -> TARGET_STATE           // obtain target state without losing cur state
   If the TARGET_STATE is unvisited
      EXPLORE(TARGET_STATE)
ENDPROC
```

Prolog demo of the principle for a small model

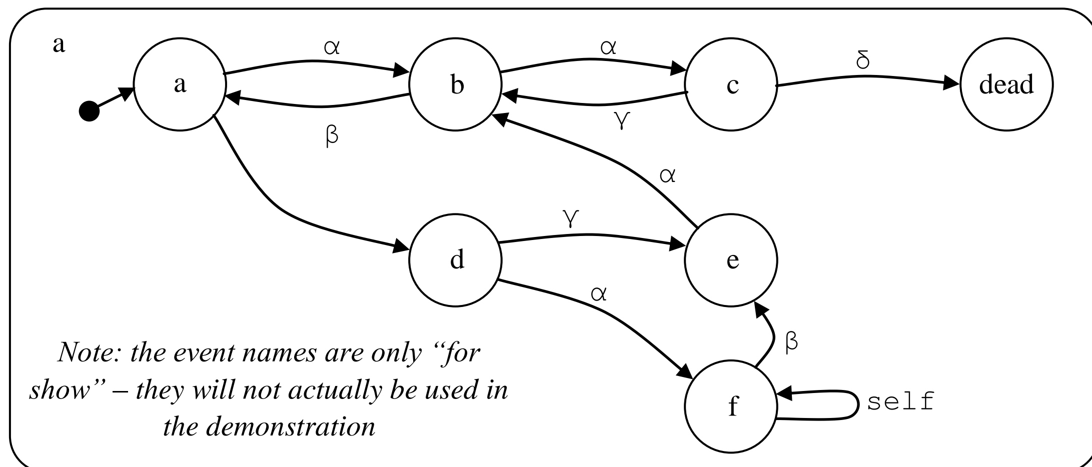


Figure 7. Prolog demo of *explore* for a small model

© Graham G. Thomason 2003-2004

```
/*-------------------------------------------------------------------------*/
/* TRANSITIONS                                                             */
/*-------------------------------------------------------------------------*/
transition(a,alpha,b).
transition(a,beta,d).
transition(b,alpha,c).
transition(b,beta,a).
transition(c,gamma,b).
transition(c,delta,dead).
transition(d,gamma,e).
transition(d,alpha,f).
transition(e,alpha,b).
transition(f,self,f).
transition(f,beta,e).

/*-------------------------------------------------------------------------*/
/* USER TOP LEVEL GOAL                                                     */
/*-------------------------------------------------------------------------*/
goa:-
   clear,
   explore(a),
   writevisited.

/*-------------------------------------------------------------------------*/
/* EXPLORE FROM STATE X                                                    */
/*     EVENT is not used but we keep it                                    */
/*-------------------------------------------------------------------------*/
explore(X):-
  markvisited(X),
  transition(X,EVENT,TARGET),
    ( (
        visited(TARGET)        /* have been here before, will backtrack  */
    );(
        not(visited(TARGET)),  /* have not been here before              */
        explore(TARGET)        /* recursively explore                    */
    ) ),
    fail.                      /* backtrack to next transition           */
explore(_).                    /* end up by succeeding                   */

/*-------------------------------------------------------------------------*/
/* UTILITIES                                                               */
/*-------------------------------------------------------------------------*/
writevisited:-
   visited(X),
   write(X),
   tab(1),
   fail.
writevisited:-
   nl.

markvisited(X):-
   assertz(    visited(X) ).

clear:-
   retractall( visited(_) ).
```

Figure 8. Prolog code of "explore" demo

Output: `a b c dead d e f`

© Graham G. Thomason 2003-2004

Extended exploration algorithm [Alg-2: deterministic, event-parameters, variables]

```
PROC EXPLORE(VSTATE)          // A VSTATE is a (variable × state) configuration
Mark VSTATE as visited
For each EVENT                     // All from same current state
  For each EVENT-PARAMETER-VALUE // or combination if many parameters
     Do EVENT -> TARGET_VSTATE    // obtain target state without losing cur state
     If the TARGET_VSTATE is unvisited
        EXPLORE(TARGET_VSTATE)
ENDPROC
```

Extended exploration algorithm [Alg-3: *non*deterministic, event-parameters, variables]

```
PROC EXPLORE(BSTATE)          // A BSTATE is a set of VSTATES
Mark BSTATE as visited
For each EVENT                     // All from same current state
  For each EVENT-PARAMETER-VALUE // or combination if many parameters
     Do EVENT -> TARGET_BSTATE    // obtain target state without losing cur state
     If the TARGET_BSTATE is unvisited
        EXPLORE(TARGET_BSTATE)    //
ENDPROC
```

Notes

- We do not mark individual world states as visited, but *sets of worlds* as visited, as *precisely that set* is visited.

Random Testing

```
For N=1 To NUMBER-OF-TESTS         // could loop as long as time permits
  SELECT RANDOM EVENT              // consider all events, transitionable or not
      For each EVENT-PARAMETER-VALUE
           SELECT RANDOM VALUE IN RANGE
           PROCESS EVENT
```

Here a random event is selected from the set of all events that can be supplied to the implementation. We do not restrict ourselves to transitionable events, since it is necessary to test that non-transitionable events do not cause a transition in the implementation under test.

5. Module organisation

This chapter is a brief overview of how the primer-related software is organised in modules. The overview will be of use in the event of software maintenance.

Review: STATECRUNCHER software is organised in the following directories:

gp4

`aa`	*Prolog /Operating-System Dependent code*
`ar`	*Arithmetic*
`cp`	*Compiler (generic part)*
`ex`	*Expression parser*
`gn`	*General routines*
`io`	*I/O routines*
`p1`	*Pass-1 parsing for the main STATECRUNCHER language*
`re`	*Regular expressions*
`tf`	*Test framework*

stcr2sand

`ac_sc`	*Compilation of test models (as a test sub-suite in itself)*
`am_sc`	*Transition testing on test models*
`ap_sc`	*Application specific naming (e.g. compiler header text)*
`boot_sc`	*Boot loader (and auxiliary loader)*
`ci_sc`	*command interface (for compilation settings)*
`cs_sc`	***command shell (for PRIMER-STATECRUNCHER commands)***
`da_sc`	*data access (primitive API routines used in transitioning and world merging)*
`ev_sc`	*evaluator (for arithmetic and scoping expressions)*
`fu_sc`	*evaluator for functions*
`me_sc`	*machine engine*
`mk_sc`	*"make" routines for scalable stress-testing models*
`op_sc`	*operator definitions (for expressions)*
`sc_sc`	*command routines for compilation*
`sy_sc`	*syntax in DCG form*
`ut_sc`	*utilities*
`va_sc`	*validator (can be considered the second phase of compilation)*
`zt_sc`	*extra area for patches, experiments*

The `cs_sc` directory (in bold above) contains the *PRIMER - STATECRUNCHER command* handler. Any lower-level routines called in the code can be identified by their two-letter

© Graham G. Thomason 2003-2004

prefix, such as me_ for machine engine, as indicated above. The files in the `cs_sc` directory and their contents, in brief, are:

`cs_sc_1.pl` *The underlying parsing predicates, parsing from ASCII:*

`cs_p1_opt_compound_term`	*optional compound term ("slist"/simple term)*
`cs_p1_compound_term`	*compound term*
`cs_p1_opt_slist`	*optional "slist" (recursive list structure)*
`cs_p1_slist`	*"slist"*
`cs_p1_simple_term`	*simple term*
`cs_p1_literal`	*literal text*
`cs_p1_sint`	*optionally signed integer*
`cs_p1_identifier`	*identifier*
`cs_p1_atom`	*quoted/nonquoted atom*
`cs_p1_opt_wspace_seq`	*optionalwhite space sequence*
`cs_p1_wspace_seq`	*white space sequence*
`cs_p1_wspace_item`	*white space item*
`cs_p1_any_text_long_atomized`	*Backtrackable "any text" as an atom*
`cs_p1_any_text_long`	*Backtackable "any text" as a list*

`cs_sc_2a.pl` *The parser for `cs_line` - a line of input (as from PRIMER)*

`cs_line` *for* `process event` *`EVENT`* `p=`*`PARAMETERS`* `t=`*`EXPECTED_TRACE`*
`cs_line` *for* `get trace`
`cs_line` *for* `clear trace`
`cs_line` *for* `get all events`
`cs_line` *for* `get all transitionable events`
`cs_line` *for* `get all variables`
`cs_line` *for* `get all worlds`
`cs_line` *for* `get config`
`cs_line` *for* `get symbol table`
`cs_line` *for* `kill world(s)`
`cs_line` *for* `WORLD TRACE=`*`TRACE`*
`cs_line` *for SET/CLUSTER/LEAFSTATE lines*
`cs_line` *for VAR lines*
`cs_line` *for* `create new world`
`cs_line` *for* `merge worlds`
`cs_line` *for* `get processing time`
`cs_line` *for* `get date`

`cs_sc_2b.pl` *The parser for `cs_line` continued*

`cs_line` *for* `no set tran`
`cs_line` *for* `low set tran`
`cs_line` *for* `medium set tran`
`cs_line` *for* `high set tran`

© Graham G. Thomason 2003-2004

`cs_line` *for* `no race`
`cs_line` *for* `low race`
`cs_line` *for* `medium race`
`cs_line` *for* `high race`
`cs_line` *for* `root` *`ROOTDIRECTORY`*
`cs_line` *for* `mode modelnames | filenames`
`cs_line` *for* `compile` *`FILENAME|MODELNAME`*
`cs_line` *for* `load` *`FILENAME|MODELNAME`*
`cs_line` *for* `run` *`FILENAME|MODELNAME`*
`cs_line` *for* `exit machine`
`cs_line` *for* `unload machine`
`cs_line` *for* `reset machine`
`cs_line` *for* `quit`
`cs_line` *for* `help`
`cs_line` *for* `prolog`

`cs_sc_3.pl` *the main loop, calling:*
`cs_read_process` *read line and process it, calling:*
`cs_read_command` *read a command from standard input*
`cs_process_ascii_line` *parse and process line, calling:*
`cs_line` *parse line (in `cs_sc_2a/b`)*
`cs_process_parsed_line` *process parsed line calling:*
`cs_process_command` *process command (in `cs_sc_4a/b`)*

`cs_sc_4a.pl`
`cs_process_command` *process the commands parsed in `cs_sc_2a`*
`cs_sc_4b.pl`
`cs_process_command` *process the commands parsed in `cs_sc_2b`*

`cs_sc_4c.pl` *Auxiliaries for command processing:*
`cs_create_old_world` *re-create a world that has existed*
`cs_create_new_world` *create a new world*
`cs_get_pvalue` *Convert a STATECRUNCHER wrapped value to a plain value*
`cs_get_scvalue` *Convert a plain value to a STATECRUNCHER wrapped value*

`cs_sc_8.pl` *The old Release 1.02 command-transition loop*
`cs_sc_8_socket.pl` *The socket version of the above (SWI-PROLOG only)*

`cs_sc_z...pl` *Test routines*

© Graham G. Thomason 2003-2004

The old Release 1.02 loop

The old Release 1.02 loop uses the commands

    cs_loop_wp(MODELNUMBER)
    cs_loop_np(MODELNUMBER)

to process events in a loop on a pre-compiled and indexed model, with and without event parameters respectively. These commands are called directly by aliases, the informal predicates:

    craft(MODELNUMBER)
    craftnp(MODELNUMBER)

which are defined in the `aux_load_sc.pl` file in the `boot_sc` directory.
In this loop, commands and events are read as PROLOG terms. This has the disadvantage that any errors are reported by the Prolog system, not application code. (It is possible to write error handlers, but they would be Prolog-system dependent, and it would typically not be easy to recover to the right place in the command loop).

For the ***socket version*** of the Release 1.02 loop, the `aux_load_sc.pl` file must be edited to load `cs_sc_8_socket.pl` instead of load `cs_sc_8.pl`. Use of the socket interface is not supported with the executable version of STATECRUNCHER.

Maintenance note

For all commands, we require
- a parse
- a test suite for the parse
- an update to the local number of tests for parse predicates
- a handler
- a test suite for the handler
- an update to the local number of tests for handler predicates
- an entry in the help command
- an update to the global number of tests
- documentation in this report

6. Conclusions

This report has shown the main principles of test case generation using STATECRUNCHER as an oracle, and the syntax of a PRIMER-STATECRUNCHER dialogue.

The next stage of the project is to examine existing TorX tools and protocols so as to match STATECRUNCHER to the tool chain. This work is being undertaken jointly by PDSL-Redhill (under the auspices of Nat.Lab.-IST) and IST-Bangalore. STATECRUNCHER Release 1.02 has already been integrated into the TorX chain using a socket interface [Koppalkar], but complete end-to-end integration was subsequently achieved, early in 2003. It is anticipated that STATECRUNCHER 1.03 will enable more powerful testing algorithm to be deployed.

An alternative option is to produce a chain of independent programs without using TorX, connecting them by pipes, sockets and Expect Scripts as appropriate [Expect].

A possible scenario where the primer is split in two parts is follows

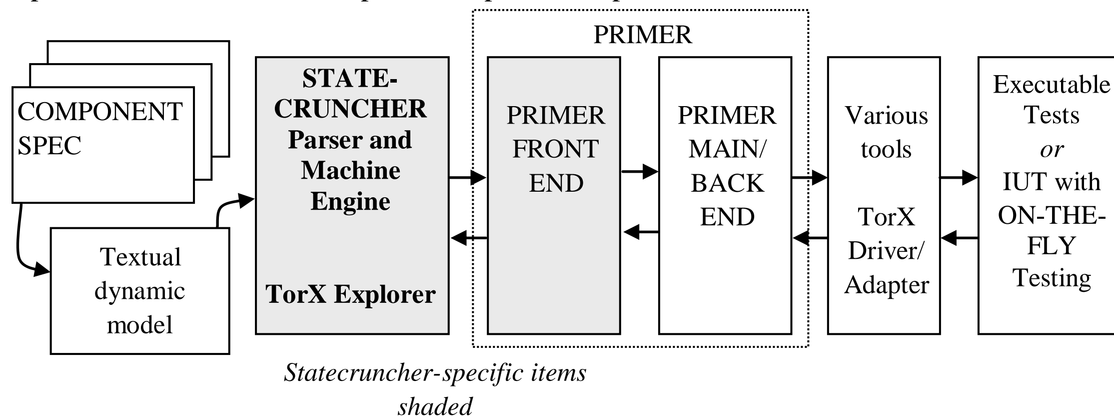


Figure 9. STATECRUNCHER specific vs. TorX generic tool chain components

Test-case generation is of interest to universities (e.g. Brunel, –see [Hierons 98]). What has been lacking is industrial tool support for the 'academic' (but perfectly valid) algorithms. An opportunity is arising to for useful co-operation in which the academic work can be harnessed to the benefit of industries such as ours. It is therefore essential that the tool chain and its protocol will ultimately be capable of supporting the most powerful adaptive on-the-fly test generation algorithms under nondeterminism.

© Graham G. Thomason 2003-2004

7. 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 3	[StCrSemComp]	A Semantic Comparison of STATECRUNCHER and Process Algebras
Appendix 4	[StCrOutput]	A Quick Reference of STATECRUNCHER's Output Format
Appendix 5	[StCrDistArb]	Distributed Arbiter Modelling in CCS and STATECRUNCHER - A Comparison
Appendix 7	[StCrNim]	The Game of Nim in Z and STATECRUNCHER
Appendix 8	[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

© Graham G. Thomason 2003-2004

References

[Baker 95] M.L. Baker and D.C. Yule
Automation of Software Testing:
A Case Study on a Real-Time Embedded System
PRL Technical Note 3373, September 1995

[CdR] Côte de Résyste
http://fmt.cs.utwente.nl/CdR
Côte de Resyste (COnformance TEsting of REactive SYSTEms) is a research and development project (1998-2002) funded by the Dutch Technology Foundation STW (http://www.stw.nl/), and is a collaboration between:
- the University of Eindhoven (http://www.tue.nl)
- the University of Twente (http://www.utwente.nl/)
- Philips (http://www.philips.com)

[CdR-iP] René de Vries, Jan Tretmans, Axel Belinfante, Jan Feenstra, Lex Heerink, Loe Feijs, Sjouke Mauw, Nicolae Goga, Arjan de Heer
Côte de Résyste in Progress
see the [CdR] site

[CHSM] P.J. Lucas
An Object-Oriented System for Implementing Concurrent, Hierarchical, Finite State Machines.
MSc. Thesis, University of Illinois at Urbana-Champaign, 1993

[DejaGnu] R. Savoye
The DejaGnu Testing Framework
The Free Software Foundation, 1993

[Expect] Don Libes
Exploring Expect
O'Reilley & Associates, 1995, ISBN 1-56592-090-2

[Hierons 98] R.M. Hierons
Adaptive testing of a deterministic implementation against a nondeterministic finite state machine
The Computer Journal, *41*, 5 pp 349-355
Available from the author's home page: www.brunel.ac.uk/~csstrmh

[Koppalkar] Nitin Koppalkar and Animesh Bhowmick
Integration of Generic Explorer with the TorX Tool Chain
Nat Lab Technical Note 2002/387

 © Graham G. Thomason 2003-2004

[Raptis] D. Raptis
Generation of Test Sequences from FSM's
PRL Technical Note 3683, March 1998

[Trew 98] T.I .P. Trew
State-based Testing with WinRunner: the State-Relation Package
PRL Internal Note SEA/704/98/05, June 1998

[Yule 97] D.C. Yule
Automatic State-Based Testing
PRL Technical Note TN 3611, 1997 /
DVD Document V19 C4 S415.

© Graham G. Thomason 2003-2004