Appendices

To the Thesis
The Design and Construction of a State Machine System that Handles Nondeterminism

Graham G. Thomason

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

July 2004

© Graham G. Thomason 2003-2004
Software Testing in Context

Graham G. Thomason

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

Unis

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

This paper presents a broad approach to testing – an approach that theoretically could be adapted and applied to a typical software project. It is a condensation of what appear to be the best state-of-the-art practical testing techniques. We cover module testing, integration testing, and system testing; white-box testing and black-box testing; automated test execution and automated test generation.

The purpose of the paper is to situate state-based-testing in a broad testing context.

A word of caution is in place. Owing to the variety of techniques presented, one would be unwise to attempt them all on any one project, as this could easily lead to an overload of tooling and lack of focus on key testing issues for the particular project in hand.
Contents

1. A general testing approach ........................................................................................................1
   1.1 Code checking .................................................................................................................2
   1.2 Module testing ..............................................................................................................3
   1.3 Integration testing .........................................................................................................7
   1.4 System testing .............................................................................................................8
2. Non-functional testing .............................................................................................................9
3. Automated test execution .....................................................................................................10
4. Automated test generation ...................................................................................................17
   4.1 State-based testing .......................................................................................................17
   4.2 Decision tables ...........................................................................................................19
   4.3 Cause-effect graphing (CEG) ....................................................................................27
   4.4 Syntax testing .............................................................................................................36
   4.5 Orthogonal arrays .....................................................................................................42
   4.6 Other model-based testing .........................................................................................42
   4.7 Random testing ..........................................................................................................42
   4.8 Summary of automated test generation .................................................................42
5. Abbreviations .......................................................................................................................43
   5.1 Testing-related abbreviations ...................................................................................43
   5.2 Other abbreviations used ..........................................................................................43
6. References ............................................................................................................................44

© Graham G. Thomason 2003-2004 iii
1. A general testing approach

The V-model for the software development life-cycle is well-known. The testing phases of this model are shown in Figure 1.

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

- **Code checking in general**: Static analysis can reveal bad coding style and possible pitfalls. Dynamic techniques can check for memory leaks and can provide code coverage, such as statement coverage, described in more detail later.

- **Module testing**: The question to be answered is: Does the implementation correspond to the design? Modules are usually single functions, or a small number of tightly coupled functions designed against a single specification. Exercise code statements and branches. Use code instrumentation to check for coverage of these. Also include a memory leak check in the tests. Module testing is typically white-box testing - we have a knowledge of the code structure and use it to guide us in designing test cases, and we have detailed controllability and observability of the module.

- **Integration testing**: The question to be answered is: Is the design internally consistent? Exercise interfaces between modules. Measure call-pair coverage (i.e. every call and every return from it). Integration testing is typically black-box testing - some modules may even be only available as object code, and the only way we can test the integrated system is via the published interfaces.

- **System testing**: The question to be answered is: Does the system satisfy the project requirements? This will typically be a black-box testing activity, since the requirements do not normally specify internal controllability and observability, but
rather the operations and their outputs which to which the end-user has access. For some kinds of system, a part of system testing will be volume testing. For example, a set-top box will need to be tested with large quantities of MPEG streams, and a Global Positioning System will need to be tested with large quantities of sampled radio-front-end (intermediate frequency) satellite data.

Tests suites are best structured, where possible, as a set of individually self-sufficient test cases, defining their own pass/fail criterion (rather than e.g. comparing output with that of previous runs). Some tests will address robustness under error situations.

For each form of testing, it may be advantageous not to test against the specification directly, but to produce a test specification, and test against that. In this way, we admit that we are not testing everything (or every combination of things), but we do make explicit what combinations of things we are testing.

![Figure 2. The Principle of a Test Specification](image)

In addition to functional testing, there is non-functional testing, which is largely a form of system testing. This is considered in chapter 2. Further chapters address test automation.

We now consider the aims of each form of testing in a little more detail.

### 1.1 Code checking

At the bottom of the V model is coding. As code is produced (or perhaps upgraded from prototype to production status), it should be subjected to some static analysis. This could be:

- Automated static analysis, e.g. for C and C++, by the [QAC] product. This analysis will reveal poor coding style and many potential bugs. It also provides code complexity metrics. Experience shows that complex code in terms of its branching and looping structure (having a high cyclomatic complexity metric) is much more liable to have bugs than one which is less complex.
- Code reviews by peers. This is often regarded as being as valuable as testing.

Code may also be subject to dynamic analysis. The following can be used when testing:

- Memory leak and array bounds checking (using e.g. [Purify]).
- Code instrumentation for statement or branch coverage checking.
- Data flow testing. A tool tracks the use of variables, and reports on suspect use.
We discuss code instrumentation and data flow testing in little more detail in section 1.2.

1.2 Module testing

Modules are tested against a module specification, and we aim to cover all statements or all branches in our tests. There is a saying that if in your tests you haven't executed any lines of code, you might as well rip them out of the product, because they are a good as defective. Statement coverage is essential, branch coverage is desirable, but there are various levels of detail of branch coverage, which we briefly discuss. Then we give some advice on how module tests could be designed.

Code coverage is obtained by instrumenting the code, so that when it is executed, apart from executing its own function, it also produces a log or trace of what code was executed.

Example (from [McCabe])
The program is regarded as segments (between potential branches or function calls), which are numbered by a node number. The node numbers are recorded on execution.

Uninstrumented

```c
define _mcrepco2 
if (Getstate() > 0) {return Fred(); }
```

Instrumented

```c
if ( (_mcrepco2(1662,1663,(GetState() > 0) != 0) ) )
    {return Fred();}
```

The call to _mcrepco2 contains the evaluated condition in the third argument, so that the relevant node number (the first or second argument) can be logged according to whether it is true or false, and so the resultant boolean value can be returned into the if statement.

A table is then produced with coverage results, e.g.

<table>
<thead>
<tr>
<th>Module Name</th>
<th># Branch</th>
<th># Covered</th>
<th>% Covered</th>
</tr>
</thead>
<tbody>
<tr>
<td>LsdSyncDec::GetResource</td>
<td>8</td>
<td>5</td>
<td>62.5</td>
</tr>
<tr>
<td>LsdSyncDec::OpenSession</td>
<td>11</td>
<td>5</td>
<td>45.5</td>
</tr>
</tbody>
</table>

Table 1. Example of a coverage table

Before we discuss forms of statement and branch coverage, we must discuss a factor that interferes with measurement of some of them. C and many other languages use short-circuit evaluation of boolean expressions. Short-circuit evaluation skips evaluating operands where they do not contribute to the expression result. The problem that arises is

- Not all combinations of boolean terms are relevant - but in the context of short-circuit evaluation we know that, and do not count them against us in terms of the coverage percentage.
- Boolean operands could be function calls that may have side effects. So they cannot safely be evaluated in instrumented code if they would not be evaluated under normal uninstrumented circumstances. So we cannot measure some forms of coverage.
In the examples below, our typical condition is

\[ \text{if } (x==0 \text{ || } y==0 \text{ || } z==0) \ldots \]

(For simplicity we do not call functions here).

The naming for code coverage is not universally standardised; we take frequently used names. The most commonly met forms of coverage that one could attempt to cover are:

1. **Statement coverage.** This is achieved if the if statement is executed at all.

2. **BDC: Branch decision coverage**
   Full coverage is obtained by any expressions that make the entire boolean expression true and false.

3. **BCC: Branch condition coverage.** The individual terms (not the variables) in the boolean expression must be made true and false at some time. So \(x==0\) must be true and must be false on occasions, as must \(y==0\) and \(z==0\). But we are not concerned about combinations, or even whether the branch is taken.

4. **BDC/BCC:** The union of BDC and BCC.

5. **MC/DC: Modified condition decision coverage.** Each boolean operand must individually affect the outcome of the decision. Four combinations would suffice for values of \(x, y, \) and \(z\) (using \(t=\text{true}, f=\text{false}, x=\text{don't care}\)): \((f,f,f), (t,f,f), (f,t,f), (f,f,t)\). In general this requires \(n+1\) tests for \(n\) boolean operands. Under *short-circuit evaluation*, this form of coverage can be measured on the understanding that it really is done in the context of short circuit evaluation. So \((f,f,f), (t,x,x), (f,t,x), (f,f,t)\) gives full coverage. However, with all \(x=t\) in practice, say, it would not necessarily give full coverage if the terms in the expression were re-ordered, though with all \(x=f\) it would. MC/DC coverage in the short-circuit context is called *masking-MC/DC* and in the long-circuit context it is called *unique cause MC/DC*.

6. **BCCC: Branch condition combination coverage.** This requires that the boolean operands take on all values in all combinations, i.e. \((f,f,f), (f,f,t), (f,t,f), (f,t,t), (t,f,f), \ldots\). In general this involves \(2^n\) tests for \(n\) boolean operands. Under *short-circuit evaluation*, this form of coverage can be sensitized for, but not all measured.

7. **LCSAJ: Linear Code Sequence And Jump** coverage. This may appear to be like branch testing, but it differs in that it requires that loops are executed in ways that branches do not require.

8. **Path coverage.** For full coverage, all paths through the program are taken. The enormous number of paths in a typical module makes this impracticable.
In practice BDC is often chosen where testing time is very limited. BCC is very weak on its own, as it does not force branch decision. BDC/BCC appears to be offered by many inexpensive tools. MC/DC is potentially very powerful (it exposes the weakness of the above-mentioned coverage criteria) but takes quite some work (but so does BDC/BCC). MC/DC is required as part of the US Department of Defense standard DO-178B. BCCC is excessive in most cases, and impracticable with short-circuiting languages such as C. LC/SAJ is powerful and should be feasible in many cases. Not all are supported by all tools.

Data flow coverage
This form of coverage is not based on statements, but on data flow as variables are Defined (created, initialized, or written to in an assignment), Used (as a Predicate in a condition, or in a Calculation in the right hand side of an assignment), and Killed (e.g. by going out of scope). A coverage requirement might be that every path from Definition to Use is exercised. Many more paths are useful. Anomalies are looked for such as DK (why define and kill without using?) or KU (definitely a bug - an undefined value is being used). Reference: [Beizer, ch.5].

There are many other forms of coverage - see for example [BCS Sigist].

How should module tests be designed?
The module under test will often be isolation tested, where all modules it calls are stubbed. Stubbing is replacing real modules by small modules with pre-cooked return values, preferably controllably by the test script. This gives more control over the module than when it is not stubbed.

Sometimes there is opportunity for automatic test generation, especially for state-based testing, decision table testing and cause-effect graphing (discussed later). But often module tests will be hand crafted. The tests will typically be matter of supplying various sets of parameter values in a function call. Global data may also play a role. Parameter values should be divided into equivalence classes, based on critical boundaries. Then 'grazing' values should be taken in and just out of each equivalence class. For example, if an equivalence class is the range -9..-4 (inclusive), test at least with values -10, -9, -4, -3. Correct error handling for out-of-range values should be checked.

Specific points of attention for numerical systems
Calculation-intensive applications have the potential for many numerical errors. Points of attention could be

- Finding all divisions in expressions and looking for possible sensitization of division by zero
- Looking for overflow / underflow / sign flip - perhaps in mid-expression - (perhaps detect it by assertion)
- Looking for int / unsigned / long int / unsigned long int / float / double / long double mixes in expressions and review them (maybe static analysis can help).
- Looking for all subtractions in expressions, and anticipate insufficient precision. The result of (large number)-(another similarly large number), e.g. 123456789.12-
123456789.13, producing a very small number, is subject to great loss of precision, because much of the available precision was used up in storing the parts of the numbers that were subtracted away.

- Subjecting the module to massive feeds of data (volume testing) around critical expressions where it is claimed that dangerous values of variables cannot occur, with dense assertions in the codes; also continue to look for values indicative of overflow/underflow/sign flip (loss of precision due to subtraction might be hard to detect by assertion). The data might be:
  - random data
  - artificial data representing unusual circumstances.

After the tests have been designed, scripted and run the, the coverage figures can be analysed, and ways should be devised to sensitise for branches that were not taken. Occasionally, extra test software (such as special stubbing) is required to do this, because the error condition might be hard or impossible to sensitise from calling parameters alone.

**Code coverage targets**

What coverage targets should be set? Safety critical industries would require 100% MC/DC coverage. A paper claiming the experimental effectiveness of MC/DC is [Dupuy]. However, it does require considerably more effort than BDC/BCC, which are more commonly taken as norms. It is sometimes infeasible to sensitise for coverage certain parts of code, especially some error handling code, except by artificially forcing it.
1.3 Integration testing

Integration testing is the testing of interfaces between modules. It is important, because if it is not done, errors will occur in system testing which will be hard to diagnose, because it will not be clear exactly what caused the failures. What may happen is that after the defective statement was executed, no failure was yet caught and more statements were executed, and memory blocks became overwritten, destroying evidence.

In integration tests, we do not attempt to reproduce the coverage of module testing. What we do concentrate on is module-to-module interfacing and interaction. Potential causes of integration errors in a system, and how to address them, are described in [Trew 99], covering:

- **Incompatibilities between actual and formal parameter ranges.**
  
  - Test with boundary values.

- **Errors in large scale state behaviour**
  
  - Reach all states. Make all transitions, perhaps all pairs of transitions

- **Interpretation of parameter values, (e.g. in interpretation of units, of array offsets, in enumerated values, a defect caused by a make file bug)**
  
  - Exercise all call pairs (tooling can give the call pair coverage)

- **Parameter ordering. Parameters of the same type may be inadvertently exchanged**
  
  - Exercise all call pairs (tooling can give the call pair coverage)

- **Dependencies on shared global data. Is the data used consistently? Is it always initialised?**
  
  - Structured data-flow tests
  
  *or*
  
  - Volume test with high levels of activity, and check for integrity of the data

- **Re-entrancy (direct recursion, indirect recursion).**
  
  - Visualisation tools will reveal it

- **Race conditions**
  
  - (State-based) test under all preconditions.
  
  - Ensure design (and code) employs a handshake

- **Deadlock**
  
  - Rigorous design inspections
  
  - Volume testing.

It is seen that exercising call pairs (client-server calls) and state-based testing can play an important role, as does design/code inspection.
1.4 System testing

System testing addresses the question of whether the system meets the customer's or project manager's requirements. Even perfect module and integration testing, with 100% coverage figures, will not protect against swathes of missing functionality. System testing is against requirements and system level analysis documents, and obviously the approach is very application specific. The use of a test specification (see Figure 2) is particularly useful here. Many tests of a fully integrated system should be centred around the user - i.e. they should be use cases.

Use cases

Use cases are part of UML. For the UML baseline, see [Catalysis, Ch. 4]. Use cases are important in system testing, because, if well chosen, they exercise the software in the way it is likely to be used in practice. Use cases are part of the [PHASST] approach in Philips, where they are described as follows:

A use case describes the system's behaviour under various conditions as the system responds to a request from its users. The system user, primary actor in use case terminology, interacts with the system to achieve some goal. Each use case is a high level description of the group of scenarios which may be triggered when a particular set of conditions holds. It also includes a set of conditions that are valid when the sequence of events associated with any of the scenarios in the set is completed.
2. Non-functional testing

According to [Evans], reporting for the BCS SIGiST, functional areas are concerned with what a product does, and non-functional areas are concerned with how well the product behaves, including whether a product is enjoyable to use and perceived as trustworthy.

The list of non-functional testing techniques from [Evans] and [TestingStds] is as follows:

- Memory Management
- Performance
- Stress Procedure
- Reliability
- Security
- Interoperability
- Usability
- Portability
- Compatibility
- Maintainability
- Recovery
- Installability
- Configuration
- Recovery
- Disaster Recovery
- Conversion

The SIGiST is currently (2003) in the process of elaborating on these concepts. Each project needs to review which of the above are applicable and how to address them in the light of its own context of use.
3. Automated test execution

The techniques described here apply across different levels of testing (module, integration, system testing).

Testing should normally be automated where possible. Humans become weary of e.g. repeatedly following written test instructions manually and checking output by eye. But even a collection of diverse test programs can be difficult to manage. The best kind of test suite is one in which

- All tests are called in a uniform way
- Every test calls the Implementation Under Test (IUT) and examines the IUT output directly in the script.
- Every test defines its own pass/fail criterion
- Every test logs the test name or number and a pass or fail indication.
- If possible, the test script supplies values to stubbed modules, so that all relevant data to a test comes from the test script, and is not distributed among special stub routines.

A basic way of automating test execution is illustrated in the following figure:

![Automated test execution diagram]

There are two levels at which tests may be scripted:

- Hard-linking the test script to the IUT (Implementation Under Test). In this case, the tests are direct function calls and tests on return values or on global data. A tool that supports this kind of testing, and also gives coverage data, is Cantata [Cantata].
- Communicating with the IUT at the executable level. A good public domain tool for communicating via Standard Input and Standard Output is DejaGnu. [DejaGnu].

Examples of Cantata and DejaGnu in use are now given.
**Cantata**

Cantata is a commercial test harness from IPL. It is suitable for C testing. There is a sister product called Cantata++ which is suitable for C or C++ testing, which is more actively promoted by the company. We show what is essentially involved in writing test cases in Cantata.

In the example below, we are testing some function `myfunc` which takes an integer parameter and returns an integer. This function calls another function, which is artificially called `stub`, since it will be stubbed. The figure below shows a Cantata test script, including stubs for the stubbed function, and instructions on how the stub is to be used on each call to it.
Cantata test case example

```c
extern int myfunc(int); // IUT Declaration:
  // a function taking and returning an int

int myfunc_P1;         // Variable to hold the parameter value
int R_myfunc;          // Variable to hold the return value
int E_R_myfunc;        // Variable to hold the expected return value

/*** Test Case ***/
START_TEST(2);

myfunc_P1=10;         // Initialize input parameter to myfunc
E_R_myfunc=20;        // Set expected return value

EXECUTE_BY_REF("myfunc","stub#1;stub#1;stub#2");
R_myfunc=myfunc(myfunc_P1);

DONE();
CHECK_S_INT("myfunc return", R_myfunc, E_R_myfunc);
END TEST();
```

Stub definition example

```c
int stub (int pi)
{
  int ret_val;
  START STUB ("stub1");
  switch (ACTION)()
  {
    case 1:
      CHECK_U_INT("pi",pi,30);
      retval=TRUE;
      break;
    case 2:
      CHECK_U_INT("pi",pi,40);
      retval=FALSE;
      break;
    default:
      ILLEGAL_ACTION();
      break;
  }
  END_STUB();
  return (ret_val);
}
```

Figure 4. Cantata test case example

The above example shows how function `myfunc` is tested. The test calls it with a parameter value of 10, and expects a return value of 20. The function calls another function, `stub`, which takes an integer parameter and returns a boolean. We stub this function by defining pre-cooked return values (TRUE and FALSE) based on the calling parameter. The stub definition allows us to check that calling parameter is 30 or 40 depending on which occasion
the stub was called. The test case itself specifies (by "stub#1;stub#1;stub#2") that the stub is expected to be called 3 times, twice under case 1 conditions, then once under case 2 conditions. Under case 1 conditions we expect stub to be called with parameter value 30 and we return the pre-cooked value TRUE. Under case 2 conditions we expect stub to be called with parameter value 40 and we return the pre-cooked value FALSE.

Any deviations from the expected values in the stub or in the return value of myfunc will cause the test to report a failure.

The test report is of the following format:

<table>
<thead>
<tr>
<th>Test Script</th>
<th>Checks Errors</th>
<th>Checks Passed</th>
<th>Checks Failed</th>
<th>Checks Warning</th>
<th>Checks Failed</th>
<th>Stubs Failed</th>
<th>Paths Failed</th>
<th>Assertions Failed</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTE 0 0 0 0 0 0 0 0</td>
<td>PASS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001 0 2 0 0 0 0 0 0</td>
<td>PASS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>002 0 3 1 0 0 0 0 0</td>
<td>&gt;&gt;FAIL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ANS 0 2 0 1 0 0 0 0</td>
<td>PASS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total 0 7 1 1 0 0 0 0</td>
<td>&gt;&gt;FAIL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

_PTE_ stands for Pre-Test Errors.  
_ANS_ stands for analysis check warning (the user can define a coverage measure as a check).
DejaGnu

DejaGnu [DejaGnu] is a layer on top of Expect [Expect-DL], which is a layer on top of TCL (Tool Command Language) [TCL].

<table>
<thead>
<tr>
<th>DejaGnu</th>
<th>Provides test suite management and Pass/Fail logging</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expect</td>
<td>Allows spawning of programs and communication with them via standard I/O. Also handles timeout.</td>
</tr>
<tr>
<td>TCL</td>
<td>An interpretative scripting language, designed for general use</td>
</tr>
</tbody>
</table>

Figure 5. TCL, Expect and Deja Gnu

TCL and Expect can both be learnt from [Expect-DL]. There is also a detailed book on TCL, [TCL], by its creator, John Ousterhout.

DejaGnu is well established on Unix Systems, and has been ported to Windows for use under CYGWIN [CYGWIN]. A separate port of Expect to Windows (by Gordon Chaffee) also exists. Both versions are pointed to by [Expect-Nist]. DejaGnu was used on the Philips G+4 set-top box platform project.

The essence of DejaGnu testing is to spawn the IUT (Implementation Under Test) and talk to its via standard input and standard output. If the IUT does not respond within a certain time, a timeout can catch this in DejaGnu.

![Diagram of DejaGnu](https://via.placeholder.com/150)

Figure 6. DejaGnu

DejaGnu communicates with an executable program, the IUT or a program relaying I/O to and from the IUT. So the IUT could be on the same computer as DejaGnu, or on another machine. In the latter case, DejaGnu would spawn e.g. a serial line program or a socket program communicating with the actual IUT. This scheme is suitable for testing the IUT on a target board, providing the necessary glue code is in place. DejaGnu (being in essence EXPECT) can spawn more than one program and control them independently if necessary.
A *calc* demonstration program is supplied with DejaGnu. It would not be confused with the proper Unix *calc* program, because of its verbose commands, *add* and *multiply*. It has the following behaviour:

```
% calc
  calc: add 2 3
  5
  calc: add 1 2 3
  Usage: add #1 #2
  calc: multiply 3 4
  12
  calc: multiply 2 4
  12
  calc: quit
% 
```

Note that the program produces a prompt after any other output. *Notice its bug!*

**Excerpts from a DejaGnu Test Script (as supplied - it could be improved)**

```
spawn calc

expect_after {
  -re "\[
      \r]*$prompt$" {
    fail "$test (bad match)"
  }
  timeout {
    fail "$test (timeout)"
  }
}

set test add1
send "add 3 4\n"
expect {
  -re "7+$prompt$" (pass $test)
}

set test add2
send "add 1 2 3\n"
expect {
  -re "Usage: add #1 #2+$prompt$" (pass $test)
}

set test multiply2
send "multiply 2 4\n"
expect {
  -re "8+$prompt$" (pass $test)
}
```

The script first spawns the *calc* program. The *calc* program will then run internally, without a window, obtaining input from Expect and writing output to Expect. The
expect_after statements in the script are effectively extensions to expect statements discussed below. Each test consists of setting a test name and sending an ASCII string to the calc program. Then the script waits for (expects) output from calc, which may match the regular expression defined. If this happens, the test is passed by a call to the DejaGnu pass function. If the text from calc for any test does not match the expect regular expression, but does match the expect_after regular expression, then control is passed to the associated statements before returning to the next test. In this example, two possibilities for expect_after have been defined: one for when some text at least ending in the calc prompt has been obtained, and one for a timeout when all else fails. Both the expect_after situations are fails, but are logged with a different annotation.

The log after running these tests

```plaintext
=== calc tests ===

spawn calc
calc: Running
./testsuite/calc.test/calc.exp ...
------------------------------------------
add 3 4
7
calc: PASS: add1
------------------------------------------
add 1 2 3
Usage: add #1 #2
calc: PASS: add2
------------------------------------------
multiply 2 4
12
calc: FAIL multiply2 (bad match)
------------------------------------------

=== calc Summary ===

# of expected passes 2
# of unexpected failures 1
```
4. Automated test generation

We have discussed how the test framework must support *automated test execution* (as far as possible) for all testing phases. Under some circumstances it may be possible to deploy *automated test generation* as well. The generated tests may be generated as a batch, in which case the same testing set-up can be used as for automated test execution. A more advanced form of automated test generation is *on-the-fly automated test generation*, where what later tests are generated depends on the results earlier tests.

- Automatic generation of tests is possible where the specifications are in a formalism with which a test generator can work:
  - state-based tests (derived from a state-transition diagram)
  - decision tables
  - cause-effect graphing
  - syntax testing
- Another form of automated testing is
  - random testing

4.1 State-based testing

The state behaviour of a system is described by a statechart, as in the dynamic model of UML. The elements of the model are

- states (in a hierarchy)
- events
- transitions (these connect source state(s) to target state(s) on an event; we say an event *triggers* a transition).

Below is an example from a smart-card manager:
Statecharts like this are valuable in pinning down the specifications and in providing a good handle for testing, whether by hand-crafted tests or by automatic test generation. To test against a statechart like this, we need to at least cover all transitions. Deeper coverage could be obtained by requiring transition pairs.

To automate the process, we need two key programs (best kept separate)
- A test generator that says what events are to be processed
- A test oracle to the tests that says what the new state is (or what outputs were expected). The oracle program may entail a language to describe the statechart, a compiler and a run-time machine engine for that language. STATECRUNCHER [StCrMain] is such an oracle.

For white-box testing, we are able to examine the state of the IUT and test against states. For black box testing, we test against outputs. The figure below illustrates white box state-based testing.
The [TorX] architecture has a more explicit test case generator in a tool chain as follows (with TorX terminology at the top, and more conventional terminology below).

![Diagram of test script and state behavior model](image)

**Figure 8. State based testing basics**

The TorX tool chain forms the basis of investigations by Philips Research Bangalore in the use of the TorX toolchain using STATECRUNCHER [StCrMain] as the oracle.

Very large numbers of tests can be generated using state-based testing, though the nature of the tests is often very unlike that of hand-crafted ones. This is especially true where there is parallelism in the model. The technique has been effective in finding defects in a DVD system and in the G+4 set-top box platform.

### 4.2 Decision tables

Decision tables directly relate combinations of inputs to multiple outputs.

Inputs are called the *condition stub*.

Outputs are called the *action stub*.

<table>
<thead>
<tr>
<th>Rule (e.g. from Requirements Specification)</th>
<th>Condition stub</th>
<th>Action stub</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C1</td>
<td>C2</td>
</tr>
<tr>
<td></td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td>false</td>
<td>x</td>
</tr>
</tbody>
</table>

© Graham G. Thomason 2003-2004
Use x for "don't care" in the condition stub.

The decision table represents a (usually pruned) tree:

```
      C1
     /  \
    no  yes
   /   /  \
  no  no  yes
 /   /  /  \
(1,1) (yes, no) (no, yes) (no, no) (yes, yes)
```

Figure 10. Decision table as a tree

Check the decision table for:
- completeness (no undetermined outputs)
- consistency (no contradictions)
- good sense (review activity)

In principle generate all input combinations

<table>
<thead>
<tr>
<th>Condition stub</th>
<th>Action stub</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>A1</td>
</tr>
<tr>
<td>C2</td>
<td>A2</td>
</tr>
<tr>
<td>C3</td>
<td></td>
</tr>
<tr>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>x-&gt;true</td>
</tr>
<tr>
<td>true</td>
<td>x-&gt;false</td>
</tr>
<tr>
<td>false</td>
<td>x-&gt;true</td>
</tr>
<tr>
<td>false</td>
<td>x-&gt;false</td>
</tr>
<tr>
<td>false</td>
<td></td>
</tr>
</tbody>
</table>

Table 3. Combinations in a decision table

An x does not mean "don't care" to the tester!! In principle generate all input combinations, (so whenever an x occurs, generate the true and false value).

Use the decision table as an oracle to the tests. To generate the tests:
- For small decision tables, the test cases can be generated by hand.
- Decision tables are a simple case of CEG (Cause Effect Graphing), and a CEG tool can be used (see section 4.3).
- Rules from the requirements specification can be expressed in a rule or logic based program such as PROLOG.

The following example illustrates how PROLOG can be used to generate the tests.
Robot Arm Example

A robot has three kinds of gripper:
• magnet
• sucker
• parallel fingers

The following rules to determine how to pick up an object:
• A magnet can only be used on ferrous objects
• A magnet requires an accessible upper surface
• A sucker requires a smooth object
• A sucker requires an accessible upper surface
• Parallel fingers require a rough object
• Parallel fingers require accessible parallel faces

---

1 This example was suggested to the author for an exercise with an expert system shell by an engineer at Agfa-Gevaert in Antwerp in 1985.
PROLOG program to derive test cases from rules

/*-----------------------------------------------*/
/* Module: robot1.pl */
/* Author: Graham Thomason, Philips Research Laboratories, Redhill */
/* Date: 10 Jun, 1999 */
/* Purpose: Example of unpruned decision table generation */
/* */
/* Copyright (C) 1999 Philips Electronics N.V. */
/*-----------------------------------------------*/

/* Representation of an object */
/* ============== */
/* An object is of the format [AUS,APF,FERROUS,SMOOTH] */
/* * AUS = Accessible Upper Surface */
/* * APF = Accessible Parallel faces */
/* * FERROUS = is ferrous */
/* * SMOOTH = is smooth */
/* * Each item in this list can be 't' (true) or 'f' (false) */
/* * if SMOOTH=f, then we say the object is rough */
/*-----------------------------------------------*/

/* Rules for picking up by different robot arms */
/* Self explanatory predicate names */
/* */
/* Parameters */
/* X (In) The object being examined for picking */
/* * For representation of the object, see comment above */
/* * VAL (Out) = 'y' (yes) or 'n' (no) according to the pickability */
/*-----------------------------------------------*/
pickByMagnet(X,y):-
    hasAccUpSurf(X),
    isFerrous(X),
    !.
pickByMagnet(X,n).

pickBySucker(X,y):-
    hasAccUpSurf(X),
    isSmooth(X),
    !.
pickBySucker(X,n).

pickByFingers(X,y):-
    hasAccParFaces(X),
    isRough(X),
    !.
pickByFingers(X,n).

/* Testing for different properties in object */
/* ----------------------------------------- */
/* The predicates take an object as their parameter and succeed if: */
/* * hasAccUpSurf(X): if X has an accessible upper surface */
/* * hasAccParFaces(X): if X has an accessible parallel faces */
/* * isFerrous(X): if X is ferrous */
/* * isSmooth(X): if X is smooth */
/* * isRough(X): if X is rough */
/* hasAccUpSurf */
hasAccUpSurf(X):-
   X=[t,_,_,_].

/* hasAccParFaces */
hasAccParFaces(X):-
   X=[_,t,_,_].

isFerrous(X):-
   X=[_,_,_,_].

isSmooth(X):-
   X=[_,_,_].
isRough(X):-
   X=[_,_,_].

/* Generate all objects (on backtracking) */
/* generates [f,f,f,f], [f,f,f,t], [f,f,t,f], etc. */
/* */
obj([AUS,APF,FERROUS,SMOOTH]):-
   ausVal(AUS), /* accessible upper surface value */
   apfVal(APF), /* accessible parallel faces value */
   ferrousVal(FERROUS), /* ferrous value */
   smoothVal(SMOOTH). /* smooth value */

ausVal(X):- tfVal(X).
apfVal(X):- tfVal(X).
ferrousVal(X):- tfVal(X).
smoothVal(X):- tfVal(X).

tfVal(f).
tfVal(t).

/* main loop */
/* */
/* Writes abbreviated keywords vertically */
/* */
/* AUS=Accessible Upper Surface (object has) */
/* MAG=Magnet (object is pickable pickable by) */
/* etc. */
/* */
go:-
   write('  A A F S M S F'),nl,
   write('  U P E M A U I'),nl,
   write('  S F R O G C N'),nl,
   fail.
go:-
   obj(X), /* loop over all objects */
   pickByMagnet(X,PBM),
   pickBySucker(X,PBS),
   pickByFingers(X,PBF),
   write(X),tab(1),write(PBM),tab(1),write(PBS),tab(1),write(PBF),nl,
   fail.
go.

© Graham G. Thomason 2003-2004
Output (with minor reformatting to facilitate annotation)

?- go.

### Object Properties
- Accessible upper surface
- Accessible parallel faces
- Ferrous
- Smooth

\[ f = \text{property is false} \]
\[ t = \text{property is true} \]

### Gripper possibilities
- Magnet suitable
- Sucker suitable
- Parallel fingers suitable

\[ n = \text{no, this gripper is not suitable} \]
\[ y = \text{yes, this gripper is suitable} \]

Figure 11. Robot arm output
Karnaugh maps

Decision tables can also be represented as grids or spreadsheets (with 2 inputs) or as cubes (with 3 inputs - but then separate planes are drawn) or as hypercubes for more inputs. These diagrams are called Karnaugh maps. Adjacent cells with the same output value, but with at least one input value held constant, reveal where a group of outputs is not dependent on all inputs, and so showing where decision logic can be simplified. The figure below shows the Karnaugh map for the robot arm, with colour coding to show grouping.

INPUTS: 4 binary variables (values t and f)
- [AccUpSurf, ParFaces, Ferrous, Smooth]

OUTPUTS: 3 binary variables (values y and n)
- [CanUseMagnet, CanUseSucker, CanUseFingers]

![Karnaugh map diagram](image)

**Figure 12. Karnaugh map**

© Graham G. Thomason 2003-2004
From the Karnaugh map a decision table with don't cares can be constructed. The same colour code as in the Karnaugh map is used below. Where 2 Karnaugh map cells form a group, there will be one don't care, and where 4 cells form a group, there will be 2 don't cares.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Upper Surface</td>
<td>Parallel Faces</td>
</tr>
<tr>
<td>f</td>
<td>f</td>
</tr>
<tr>
<td>f</td>
<td>t</td>
</tr>
<tr>
<td>f</td>
<td>t</td>
</tr>
<tr>
<td>t</td>
<td>x</td>
</tr>
<tr>
<td>t</td>
<td>x</td>
</tr>
<tr>
<td>t</td>
<td>f</td>
</tr>
<tr>
<td>t</td>
<td>f</td>
</tr>
<tr>
<td>t</td>
<td>t</td>
</tr>
<tr>
<td>t</td>
<td>t</td>
</tr>
</tbody>
</table>

Table 4. Robot gripper decision table

Decision tables are a feed-forward technique. They are applicable where there is no obvious memory in the logic, in contrast to state-based testing where states represent memory so that the same event can have a different effect at different times because of the state. However, it is possible to model simple state models as decision tables, where parallel states become condition stub items, the event becomes another condition stub item, and the action stub items are the new states.
4.3 Cause-effect graphing (CEG)

Cause-effect graphing is described in detail in [Myers, p.56]. The technique consists of establishing a relationship between inputs and outputs where the logic is more than a simple decision table. There is typically a network of logical gates (with their own CEG symbols), under constraints (shown by dotted lines below).

![Diagram of CEG]

Figure 13. A CEG

The constraint *one* above indicates that exactly one of the inputs *B* and *F* must be true, and *G requires* *H* indicates that for *G* to be true, *H* must be true.

The idea is to test key input combinations of each gate. The complexities arise from:

- The need to avoid combinatorial explosion, so to combine tests efficiently.
- The presence of constraints, such as one input requiring a truth-value of another to make sense. For example if one input is (*x>*0) and another is (*x>*6), it is not possible to have the first true and the second false.
- Observability issues. If intermediate nodes are not observable, the output of a gate must be propagated through the network. This puts sensitization requirements on other gates. This is not always logically possible - leaving certain gate combinations untestable (unless extra observability/controllability measures are taken).

The output of test cases from a CEG tool is similar to that of decision tables.

There is a commercial tool for generating CEGs:

- A tool originally called SoftTest from Bender and Associates, then apparently under Borland called Caliber-RBT and now under Nohau called Caliber-RM.
The following pages show how CEGs can be used to test the colour of a teletext object. Teletext objects are used to overwrite parts of a teletext page, but with quite complex rules to govern the colour of the new text.

We take specifications from the standard (ETS 300 706, May 1997), paragraph 13, page 98.

Example of an object overwriting underlying text:

<table>
<thead>
<tr>
<th>THE</th>
<th>FAST</th>
<th>DOG</th>
<th>AND</th>
</tr>
</thead>
<tbody>
<tr>
<td>L A Z Y</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| THE | L A Z Y | DOG | AND |

One application of teletext objects is to place an advertisement in a certain place on a set of pages, without the need to re-code the pages individually.

There are 3 kinds of object, plus underlying text, with highest-to-lowest priority as follows:

- Passive
- Adaptive
- Active
- Underlying text

We consider the 3 kinds of object in turn.

---

1 The test cases are for illustrative purposes. Absolute accuracy cannot be guaranteed, though care has been taken with them, as, due to changing testing priorities, these tests have not actually been deployed.
Active Objects

- Colour change affects underlying text ("AND")
- Until underlying text changes colour, ("CAT")
- Colour change stays in effect to end of row (not end of object range)

Active object example

<table>
<thead>
<tr>
<th>bTHERE</th>
<th>FAT</th>
<th>D0G</th>
<th>AND</th>
<th>rCAT</th>
<th>gRAN1N</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAZY</td>
<td>COW</td>
<td>FOX</td>
<td>SAT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

--range of object cells addressed by object--

<table>
<thead>
<tr>
<th>THE</th>
<th>LASYCOW</th>
<th>ANDFOX</th>
<th>SAT</th>
<th>IN</th>
</tr>
</thead>
<tbody>
<tr>
<td>blue</td>
<td>blue</td>
<td>pink</td>
<td>pink</td>
<td>red</td>
</tr>
</tbody>
</table>

Object does not have an initial colour change

-> Underlying Colour

Object sets a new colour

Colour change stays in effect

Underlying text sets new colour

Active object sets a new colour

This stays in effect to end of row

Figure 14. Active object example
Adaptive Objects

- Colour depends on
  - Colour set by adaptive object
  - Else as set by previous active object
  - Else colour of underlying text,
- Underlying col change gets overridden
- Colour changes end at end of object

Adaptive object example

<table>
<thead>
<tr>
<th>bTHE</th>
<th>FAST</th>
<th>DOG</th>
<th>ANDrCAT</th>
<th>gRAN</th>
<th>IN</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAZY</td>
<td>COW</td>
<td>FOX</td>
<td>SAT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p</td>
<td>y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

--range of object cells addressed by object--

<table>
<thead>
<tr>
<th>bTHE</th>
<th>FAST</th>
<th>DOG</th>
<th>ANDrCAT</th>
<th>gRAN</th>
<th>IN</th>
</tr>
</thead>
<tbody>
<tr>
<td>THE</td>
<td>LAZY</td>
<td>COW</td>
<td>AND</td>
<td>FOX</td>
<td>SAT</td>
</tr>
<tr>
<td>blue</td>
<td>blue</td>
<td>pink</td>
<td>pink</td>
<td>pink</td>
<td>yellow</td>
</tr>
</tbody>
</table>

Object does not have an initial colour change

-> Underlying colour

Object sets a new colour

Colour change stays in effect on under-lying text

Object overrides underlying colour change

Object sets a new colour.

Does not remain in force after end of object

Figure 15. Adaptive object example
Passive objects

- If no object colour specified, displayed colour=WHITE (highest priority inherits nothing)
- Where no character defined in object, underlying text retains its colour.
- Colour changes end at end of object

Figure 16. Passive object example

| b | T | H | E | F | A | S | T | D | O | G | A | N | D | r | C | A | T | g | R | A | N | I | N |
| L | A | Z | Y | C | 0 | W | A | N | D | F | O | X | S | A | T | 1 | N |

| p | y |

--range of object cells addressed by object--

| T | H | E | L | A | Z | Y | C | 0 | W | A | N | D | F | O | X | S | A | T | 1 | N |
| blue | white | pink | blue | pink | yellow | green |

Object does not have an initial colour change

- \( \rightarrow \) WHITE

Object sets a new colour

Object overrides underlying colour

Object overrides underlying colour change

Does not remain in force after end of object

Figure 17. Passive object example
TITLE 'Teletext Objects'.

NODES

/* Char is before any object'.
CharAfterPAS = 'Char is after a PASSIVE object'.
CharAfterADP = 'Char is after an ADAPTIVE object'.
CharAfterACT = 'Char is after an ACTIVE object'.
CharInPAS = 'Char is in a PASSIVE object'.
CharInADP = 'Char is in an ADAPTIVE object'.
CharInACT = 'Char is in an ACTIVE object'.

ExplicitObjChar = 'Char is explicitly overwritten in the object'.
UnderlyingColChange = 'Underlying text changes colour under the object'.
ObjColSet = 'Object has set colour'.

/* Intermediate Nodes */

AfterObjDispColUnder = 'Char after object- Display in underlying col'.
AfterObjDispColObj = 'Char after object- Display in object colour'.
InObjDispColUnder = 'Char in object- Display in underlying col'.
InObjDispColObj = 'Char in object- Display in object colour'.
InObjDispColWhite = 'Char in object- Display in white'.

/* Effects */

DispColUnder = 'Display the char in the underlying colour'.
DispColObj = 'Display the char in the last colour set by the object'.
DispColWhite = 'Display the char White'.

/* Constraints */

CONSTRAINTS

ONE (CharBeforeObj, CharAfterPAS,CharAfterADP,CharAfterACT, CharInPAS,CharInADP,CharInACT).
MASK (CharBeforeObj,ExplicitObjChar,UnderlyingColChange,ObjColSet).
MASK (CharAfterPAS,ExplicitObjChar).
MASK (CharAfterADP,ExplicitObjChar).
MASK (CharAfterACT,ExplicitObjChar).

/* Relations */

AfterObjDispColUnder:-
CharAfterPAS

/* */
OR CharAfterADP
OR (CharAfterACT AND NOT ObjColSet)
OR (CharAfterACT AND ObjColSet AND UnderlyingColChange).

AfterObjDispColObj:-
CharAfterACT AND ObjColSet AND NOT UnderlyingColChange.

InObjDispColUnder:-
(CharInACT AND ObjColSet AND UnderlyingColChange)
OR (CharInACT AND NOT ObjColSet)
OR (CharInADP AND NOT ObjColSet)
OR (CharInPAS AND ObjColSet AND NOT ExplicitObjChar)
OR (CharInPAS AND NOT ObjColSet AND NOT ExplicitObjChar).

InObjDispColObj:-
(CharInACT AND ObjColSet AND NOT UnderlyingColChange)
OR (CharInADP AND ObjColSet)
OR (CharInPAS AND ObjColSet AND ExplicitObjChar).

InObjDispColWhite:-
CharInPAS AND NOT ObjColSet AND ExplicitObjChar.

DispColUnder:-CharBeforeObj OR AfterObjDispColUnder OR InObjDispColUnder.
DispColObj:- AfterObjDispColObj OR InObjDispColObj.
DispColWhite:-InObjDispColWhite.

/* -----[End of script] -----*/
SoftTest “definition matrix”

The parameter settings for each of 15 tests are seen from the table produced, below. The first test, TEST#01, says that a character after the end of a passive object (and so not before or in any object), where no colour was set in the object, but where the colour of the underlying text in the range of the object did change, is displayed in the underlying colour. The three observable output properties are marked (obs).

<table>
<thead>
<tr>
<th>Causes:</th>
<th>T T T T T T T T T T T T T T T</th>
</tr>
</thead>
<tbody>
<tr>
<td>CharAfterPAS</td>
<td>T F F F F F F F F F F F F F F</td>
</tr>
<tr>
<td>CharAfterADP</td>
<td>F T F F F F F F F F F F F F F</td>
</tr>
<tr>
<td>CharAfterACT</td>
<td>F F T T F F F F F F F F T F F</td>
</tr>
<tr>
<td>ObjColSet</td>
<td>F F F T T F F T F M T F T T T</td>
</tr>
<tr>
<td>UnderlyingColChange</td>
<td>T T F T T F T F T M T F T F T</td>
</tr>
<tr>
<td>CharInACT</td>
<td>F F F T T F F F F F F F F F F</td>
</tr>
<tr>
<td>CharInADP</td>
<td>F F F F F T F F F F F F F F F</td>
</tr>
<tr>
<td>CharInPAS</td>
<td>F F F F F T F F F F F F F F F</td>
</tr>
<tr>
<td>ExplicitObjChar</td>
<td>M M M M F F F F M F T M F T</td>
</tr>
<tr>
<td>CharBeforeObj</td>
<td>F F F F F F F F T F F F F F F</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Effects:</th>
<th>T T T T F F F F F F F F F F</th>
</tr>
</thead>
<tbody>
<tr>
<td>AfterObjDispColUnder</td>
<td>T T T T F F F F F F F F F F</td>
</tr>
<tr>
<td>AfterObjDispColObj</td>
<td>F F F F F F F F F F F F F F</td>
</tr>
<tr>
<td>InObjDispColUnder</td>
<td>F F F F T T T T T T T T T T</td>
</tr>
<tr>
<td>InObjDispColObj</td>
<td>F F F F F F F F F F F F F F</td>
</tr>
<tr>
<td>InObjDispColWhite</td>
<td>F F F F F F F F F F F F F F</td>
</tr>
<tr>
<td>DispColUnder {obs}</td>
<td>T T T T T T T T T T T T T T</td>
</tr>
<tr>
<td>DispColObj {obs}</td>
<td>F F F F F F F F F F F F F F</td>
</tr>
<tr>
<td>DispColWhite {obs}</td>
<td>F F F F F F F F F F F F F F</td>
</tr>
</tbody>
</table>

Figure 18. SoftTest definition matrix
Limitations of SoftTest & CEGs

- SoftTest is not a test harness
  - It does not claim to be.
  - The tests are also output as an ASCII file and can be converted to a scripting language for use with a test harness.

- In SoftTest, the number of tests is so highly optimized that it may fail to generate tests that distinguish two inputs. For example if there is an input A to one gate, and B to another, we may find that A and B are always set to true and false together.

- CEGs are just one approach to systematic testing. They are not likely to be sufficient on their own, and should be supplemented by other forms of testing.
4.4 Syntax testing

Reference: [Beizer, ch 9] explains how syntax testing is not only applicable to formal computer languages, because software systems often have hidden languages. These may be

- a user input language
- a data format with many (perhaps nested) options (e.g. bmp files, avi files, mpeg files)
- an inter-process communication convention
- an API calling sequence convention
- communication protocols

Our example below is for C, but many systems that are not languages like C exhibit hidden languages that can be tested by syntax testing. The syntax may be represented diagrammatically as a railroad diagram, which defines the grammar, e.g.

![Syntax graph - a bit of C](image)

Test generation possibilities

- Generate legal productions of the grammar and feed them to the IUT.
- Mutate the grammar, generate productions of that, filtering them out if they happen to also be parsable by the original grammar, feed these to the IUT and check that they are recognized as error situations.
The oracle, if any, must come from some additional information, perhaps manually supplied, or embedded in the grammar. Even if no oracle to the tests is supplied, the tests have value in testing the robustness of the system. Value can be added by putting assertions in the code.

We now show how syntax coverage can be obtained using a Prolog program. The example illustrates how Prolog Definite Clause Grammars can serve two purposes:

- Obtaining a parse of input
- Generating productions from the grammar

The example generates sentences where several simple sentences can also be conjoined to make one long sentence of the kind:

*the boy likes the girl and the girl eats a pear and ...*

There is additional code to prevent sentences of the type

- the A likes the A
- the A likes the B and the A likes the B
Listing

 Validator: sentence.pl

 Author: Graham Thomason, Philips Research Laboratories, Redhill

 Date: 10 Jun, 1999

 Project: S/W Testing

 Purpose: Example of syntax-based test generation

 Copyright (C) 1999 Philips Electronics N.V.

 EXTERNALS used by this module

 ggtlib:io_pp

 NonTerminals

 sentence

 sentence(DEPTH,[sentence,S1])-->
 simple_sentence(S1).

 sentence(DEPTH,[sentence,S1,C|RESTLIST])-->
 simple_sentence(S1),
 conjunction(C),
 {(NEWDEPTH is DEPTH+1)},
 {{(NEWDEPTH <= 3) ; (NEWDEPTH > 3,!,fail)}},
 sentence(NEWDEPTH,S2),
 {(S2=_[|RESTLIST])},
 {(gn_not(gn_member(S1,RESTLIST)))}.

 simple_sentence([simple_sentence,NP,VP])-->
 noun_phrase(NP),
 verb_phrase(VP),
 {(NP=_[..,N],N=[..,NW],
  VP=_[..,NP2],NP2=_[..,N2],N2=[..,NW2],
  NW\=NW2)}. /* not the same noun in both places */

 noun_phrase([noun_phrase,A,N])-->
 article(A),
 noun(N).

 verb_phrase([verb_phrase,V,NP])-->

verb(V),
noun_phrase(NP).

/*-----------------------------------------------*/
/* Terminals                                     */
/* ==========                                    */
/*-----------------------------------------------*/
article([article,A])—>
{(article(A))},
[A].

article(the).
/* article(a). */
noun([noun,N])—>
/* noun/3 */
{(noun(N))},
[N].
noun(boy).
noun(girl).
/* noun(cherry). */
/* noun(pear). */

verb([verb,V])—>
{(verb(V))},
[V].
verb(knows).
verb(likes).

conjunction([conjunction,C])—>
{(conjunction(C))},
[C].

/*-----------------*/
/* conjunction(but). */
/*-----------------*/

/* Simple tests */
/*--------------*/
tterm:-tarti,tnoun,tverb,tconj.
tarti:-article(P,[X],[]),write(P),tab(1),write(X),nl,fail.
tarti.

tnoun:-noun(P,[X],[]),write(P),tab(1),write(X),nl,fail.
tnoun.

tverb:-verb(P,[X],[]),write(P),tab(1),write(X),nl,fail.
tverb.

tconj:-conjunction(P,[X],[]),write(P),tab(1),write(X),nl,fail.
tconj.

tmp1:-noun_phrase(P,X,[]),io_pp(P),tab(1),write(X),nl,nl,fail.
tmp1.

tmp2:-noun_phrase(P,X,[]),tab(1),write(X),nl,fail.
tmp2.

tvp1:-verb_phrase(P,X,[]),io_pp(P),tab(1),write(X),nl,nl,fail.
tvp1.

tvp2:-verb_phrase(P,X,[]),tab(1),write(X),nl,fail.
tvp2.

tssl:-simple_sentence(P,X,[]),io_pp(P),tab(1),write(X),nl,nl,fail.
tssl.

© Graham G. Thomason 2003-2004 39
tss2: -simple_sentence(P,X,[]), tab(1),write(X),nl,nl,fail.
tss2.

tsen1: -sentence(1,P,X,[]),write(P),nl,io_pp(P),tab(1),write(X),nl,nl,fail.
tsen1.

tsen2: -sentence(1,P,X,[]),tab(1),write(X),nl,fail.
tsen2.

/*---------------[end of module sentence.pl]--------------------*/

Coverage Output

| ?- tsen2.
| [the, boy, knows, the, girl]
| [the, boy, likes, the, girl]
| [the, girl1, knows, the, boy]
| [the, girl, likes, the, boy]
| [the, boy, knows, the, girl, and, the, boy, likes, the, girl]
| [the, boy, knows, the, girl, and, the, girl, knows, the, boy]
| [the, boy, knows, the, girl, and, the, girl, likes, the, boy]
| [the, boy, knows, the, girl, and, the, boy, likes, the, girl, and, the, girl, knows, the, boy]
| [the, boy, knows, the, girl, and, the, girl, likes, the, boy, and, the, boy, likes, the, girl]
| [the, boy, knows, the, girl, and, the, girl, likes, the, boy, and, the, boy, knows, the, girl]
| [the, boy, knows, the, girl, and, the, girl, likes, the, boy, and, the, boy, likes, the, girl, and, the, girl, knows, the, boy]
| [the, boy, knows, the, girl, and, the, girl, likes, the, boy, and, the, boy, likes, the, girl, and, the, girl, knows, the, boy]
| [the, boy, knows, the, girl, and, the, girl, likes, the, boy, and, the, boy, knows, the, girl]
| [the, boy, likes, the, girl, and, the, girl, knows, the, boy]
| [the, boy, likes, the, girl, and, the, girl, likes, the, boy]
| [the, boy, likes, the, girl, and, the, girl, knows, the, boy, and, the, boy, knows, the, girl]
| [the, boy, likes, the, girl, and, the, girl, likes, the, boy, and, the, boy, knows, the, girl]
| [the, boy, likes, the, girl, and, the, girl, knows, the, boy, and, the, boy, likes, the, girl, and, the, girl, knows, the, boy]
| [the, boy, likes, the, girl, and, the, girl, likes, the, boy, and, the, boy, knows, the, girl, and, the, girl, knows, the, boy]
| [the, boy, likes, the, girl, and, the, girl, likes, the, boy, and, the, boy, likes, the, girl, and, the, girl, knows, the, boy]
| [the, boy, likes, the, girl, and, the, girl, knows, the, boy, and, the, boy, likes, the, girl]
| [the, boy, likes, the, girl, and, the, girl, knows, the, boy, and, the, girl, likes, the, boy]
| [the, boy, likes, the, girl, and, the, girl, knows, the, boy, and, the, girl, likes, the, boy, and, the, boy, likes, the, girl]
| [the, boy, likes, the, girl, and, the, girl, knows, the, boy, and, the, girl, likes, the, boy, and, the, boy, knows, the, girl]
| [the, boy, likes, the, girl, and, the, girl, knows, the, boy, and, the, girl, likes, the, boy, and, the, boy, likes, the, girl, and, the, girl, knows, the, boy]
| [the, boy, likes, the, girl, and, the, girl, knows, the, boy, and, the, girl, likes, the, boy, and, the, girl, knows, the, boy, and, the, boy, likes, the, girl]
| [the, boy, likes, the, girl, and, the, girl, knows, the, boy, and, the, girl, likes, the, boy, and, the, girl, knows, the, boy, and, the, boy, knows, the, girl]
| [the, boy, likes, the, girl, and, the, girl, knows, the, boy, and, the, girl, likes, the, boy, and, the, girl, knows, the, boy, and, the, boy, knows, the, girl, and, the, girl, knows, the, boy]
| [the, boy, likes, the, girl, and, the, girl, knows, the, boy, and, the, girl, likes, the, boy, and, the, girl, knows, the, boy, and, the, boy, knows, the, girl, and, the, girl, knows, the, boy, and, the, boy, likes, the, girl]
| [the, boy, likes, the, girl, and, the, girl, knows, the, boy, and, the, girl, likes, the, boy, and, the, girl, knows, the, boy, and, the, boy, knows, the, girl, and, the, girl, knows, the, boy, and, the, boy, likes, the, girl, and, the, girl, knows, the, boy]
| [the, boy, likes, the, girl, and, the, girl, knows, the, boy, and, the, girl, likes, the, boy, and, the, girl, knows, the, boy, and, the, boy, knows, the, girl, and, the, girl, knows, the, boy, and, the, boy, likes, the, girl, and, the, girl, knows, the, boy, and, the, boy, knows, the, girl]
| [the, boy, likes, the, girl, and, the, girl, knows, the, boy, and, the, girl, likes, the, boy, and, the, girl, knows, the, boy, and, the, boy, knows, the, girl, and, the, girl, knows, the, boy, and, the, boy, knows, the, girl, and, the, girl, knows, the, boy]
| [the, boy, likes, the, girl, and, the, girl, knows, the, boy, and, the, girl, likes, the, boy, and, the, girl, knows, the, boy, and, the, boy, knows, the, girl, and, the, girl, knows, the, boy, and, the, boy, knows, the, girl, and, the, girl, knows, the, boy, and, the, boy, likes, the, girl]
| [the, boy, likes, the, girl, and, the, girl, knows, the, boy, and, the, girl, likes, the, boy, and, the, girl, knows, the, boy, and, the, boy, knows, the, girl, and, the, girl, knows, the, boy, and, the, boy, knows, the, girl, and, the, girl, knows, the, boy, and, the, boy, knows, the, girl]
| [the, boy, likes, the, girl, and, the, girl, knows, the, boy, and, the, girl, likes, the, boy, and, the, girl, knows, the, boy, and, the, boy, knows, the, girl, and, the, girl, knows, the, boy, and, the, boy, knows, the, girl, and, the, girl, knows, the, boy, and, the, boy, knows, the, girl, and, the, girl, knows, the, boy]
| [the, boy, likes, the, girl, and, the, girl, knows, the, boy, and, the, girl, likes, the, boy, and, the, girl, knows, the, boy, and, the, boy, knows, the, girl, and, the, girl, knows, the, boy, and, the, boy, knows, the, girl, and, the, girl, knows, the, boy, and, the, boy, knows, the, girl, and, the, girl, knows, the, boy, and, the, boy, knows, the, girl]
| [the, boy, likes, the, girl, and, the, girl, knows, the, boy, and, the, girl, likes, the, boy, and, the, girl, knows, the, boy, and, the, boy, knows, the, girl, and, the, girl, knows, the, boy, and, the, boy, knows, the, girl, and, the, girl, knows, the, boy, and, the, boy, knows, the, girl, and, the, girl, knows, the, boy, and, the, boy, knows, the, girl, and, the, girl, knows, the, boy]
| [the, boy, likes, the, girl, and, the, girl, knows, the, boy, and, the, girl, likes, the, boy, and, the, girl, knows, the, boy, and, the, boy, knows, the, girl, and, the, girl, knows, the, boy, and, the, boy, knows, the, girl, and, the, girl, knows, the, boy, and, the, boy, knows, the, girl, and, the, girl, knows, the, boy, and, the, boy, knows, the, girl, and, the, girl, knows, the, boy, and, the, boy, knows, the, girl]
Example of a parse

| ?- sentence(P, [the, boy, likes, the, girl, and, the, girl, knows, the, boy]), io_pp(P).
  sentence
    simple_sentence
      noun_phrase
        article
        the
        noun
        boy
      verb_phrase
        verb
        likes
      noun_phrase
        article
        the
        noun
        girl
    conjunction
    and
    simple_sentence
      noun_phrase
        article
        the
        noun
        girl
      verb_phrase
        verb
        knows
      noun_phrase
        article
        the
        noun
        boy
  P = [sentence, [simple_sentence, [noun_phrase, [article, the], [noun, boy]], [verb_phrase, [verb, likes], [noun_phrase, [article, the], [noun, girl]]]], [conjunction, and],
      [simple_sentence, [noun_phrase, [article, the], [noun, girl]], [verb_phrase, [verb, knows], [noun_phrase, [article, the], [noun, boy]]]]]

| ?- 

Code for the above pretty print formatter is as in [Clocksin, p.81].
4.5 Orthogonal arrays

Suppose a routine needs testing with 4 parameters, (A, B, C, and D), each of which can take 3 values (1, 2, and 3). Exhaustive testing would require running $3^4=81$ tests. But suppose we find it adequate that all pairwise combinations of parameter values are taken. A table can be found satisfying this with 9 entries of values of the 4 parameters as follows:

<table>
<thead>
<tr>
<th>ABCD</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
</tr>
<tr>
<td>1223</td>
</tr>
<tr>
<td>1332</td>
</tr>
<tr>
<td>2122</td>
</tr>
<tr>
<td>2231</td>
</tr>
<tr>
<td>2313</td>
</tr>
<tr>
<td>3133</td>
</tr>
<tr>
<td>3212</td>
</tr>
<tr>
<td>3321</td>
</tr>
</tbody>
</table>

For pairwise coverage as above we speak of orthogonal arrays of strength 2. If we had required that all triples of parameters should be covered for all combinations of values, the strength would be 3 and so on. See also [Sloane]; the above array is equivalent to the one at http://www.research.att.com/~njas/oadir/oa.9.4.3.2.txt.

4.6 Other model-based testing

Of the UML models, the dynamic model (state-based testing) is probably the most amenable to automated testing. But use cases, message sequence diagrams, collaboration diagrams etc. are also being used to derive tests. The [Agedis] project addresses model based testing including such models. There is also a very rich website on model-based testing maintained by Harry Robinson, with UML-based testing featuring prominently, [Robinson].

4.7 Random testing

Random testing can also be useful. In this case there is no precise oracle to the tests. However, by densely larding the code with assertions (which act as oracles in a way), the tests have value in testing the robustness of the system.

4.8 Summary of automated test generation

Automated test generation requires formal specifications such as a UML model, a decision table, a cause effect graph, or the grammar rules of a language. Large numbers of tests can be generated. State based testing has proved to be particularly effective in finding defects in practice. Sometimes the techniques, which could be used for automated test generation, can be applied by hand (e.g. for a small statechart or decision table).
5. Abbreviations

5.1 Testing-related abbreviations

- BCC: Branch Condition Coverage
- BCS: British Computer Society
- BDC: Branch Decision Coverage
- BCCC: Branch Condition Combination Coverage
- CEG: Cause Effect Graphing
- IUT: Implementation Under Test
- LCSAJ: Linear Code Sequence and Jump (coverage)
- MC/DC: Modified Condition / Decision Coverage
- PHASST: Philips Approach to Structured System Testing. See [PHASST]
- QAC: Probably: Quality Assessment for C. See [QAC]
- SIGiST: Special Interest Group in Software Testing
- TCL: Tool Command Language

5.2 Other abbreviations used

- API: Application Programmer Interface
- GNU: Gnu's Not Unix - see http://www.gnu.org
- GUI: Graphical User Interface
- MPEG: Moving Picture Experts Group
- UML: Unified Modelling Language
6. 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 3 [StCrGP4] GP4 - The Generic Prolog Parsing and Prototyping Package *(underlies the STATECRUNCHER compiler)* |
| Related report 4 [StCrParsing] STATECRUNCHER Parsing |
| Related report 5 [StCrTest] STATECRUNCHER Test Models |
| Related report 6 [StCrFunMod] State-based Modelling of Functions and Pump Engines |
References

[Agedis] www.agedis.de

[BCS Sigist] Standard for Software Component Testing
British Computer Society Special Interest Group in Testing

[Beizer] Boris Beizer
Software Testing Techniques

[Catalysis] D.F. D'Souza
Objects, Components and Frameworks with UML


[Clocksin] W.F. Clocksin & C.S. Mellish
Programming in Prolog
Springer Verlag, 1981. ISBN 3-540-11046-1


[CYGWIN] www.cygwin.com
Cygwin is a Linux-like environment for Windows. It consists of two parts:
• A DLL (cygwin1.dll) which acts as a Linux emulation layer providing substantial Linux API functionality.
• A collection of tools, which provide Linux look and feel.

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

[Dupuy] Arnaud Dupuy and Nancy Leveson
An Empirical Evaluation of the MC/DC Coverage Criterion on the Hete-2 Satellite Software
DASC (Digital Aviation Systems Conference), Oct 2000

[Evans] Isabel Evans
Non-functional testing techniques
Don Libes
Exploring Expect

http://expect.nist.gov/

G.J. Myers
The Art of Software Testing

Madhav S. Phadke
Planning efficient software tests

Elena Pérez-Miñana
PHASST (Philips Approach To Structured System Testing)
Philips PDSL Document 1207, 2003

http://www.rational.com
http://www.ibm.com

http://www.programmingresearch.com/main.htm

Harry Robinson
Model Based Testing Home Page (maintained by)
http://www.geocities.com/model_based_testing

N.J.A. Sloane
A library of orthogonal arrays

John Ousterhout
TCL and the TK Toolkit

http://www.testingstandards.co.uk/definitions.htm
(Gives a summary of non-functional testing techniques)

[Trew 98] Tim Trew
State-based Testing with WinRunner: the State-Relation Package
Philips PRL Internal Note SEA/704/98/05, 1998

[Trew 99] Tim Trew
The aims of integration testing
Philips PRL Technical Note 3922, 1999

Appendix 2

A Semantic Comparison of STATECRUNCHER and Process Algebras

An Appendix to the Thesis
The Design and Construction of a State Machine System that Handles Nondeterminism

Graham G. Thomason
A Semantic Comparison of
STATECRUNCHER and Process Algebras

Graham G. Thomason

Appendix to the Thesis “The Design and Construction of a State Machine System that Handles Nondeterminism”

UniS

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

This paper discusses the essential differences in the STATECRUNCHER approach to composition and synchronisation of processes, and to nondeterminism, to that of the process algebras CCS and CSP. It is a pre-requisite to the papers mentioned below, covering ground common to them.

In separate papers a more detailed discussion of specific case studies, taken from the CCS and CSP literature, is given. Those papers show working STATECRUNCHER models of the systems, covering their statechart diagram, source code, and output from sessions running the models. A comparison of the STATECRUNCHER model with the CCS or CSP specification is given. An additional study shows how a Z specification relates to STATECRUNCHER concepts. The case studies in those papers are:

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

Reminder of the motivation for STATECRUNCHER

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. Comparison of terminology ................................................................. 1

2. Composition of processes .................................................................. 4

3. Parallelism other than call/return composition ................................. 15

4. Nondeterminism ................................................................................ 18

5. Concluding remarks .......................................................................... 23

References ........................................................................................... 24
1. Comparison of terminology

In STATECRUNCHER terminology the main concepts in a statechart are

- states
- events
- transitions
- actions
- traces
- variables
- assignments to variables

In order to concentrate on the essentials, we do not discuss here other refinements such as multiple target states, orbital transitions, conditional transitions, conditional actions, references to state occupancies, meta-events, parameterised events, and upon-entry/upon exit actions. These are described in [StCrMain, StCrParsing]. Mention will be made, however, of PCOs (points of control and observation), as a fixed attribute to an event.

The STATECRUNCHER terminology is different to that of CCS and CSP. STATECRUNCHER actions and traces are not the same as those of CCS and CSP. The STATECRUNCHER terminology corresponds more closely to that of tools used within Philips over the years such as [CHSM] and [TorX]. For this reason, a comparison is now offered. We start with a review of STATECRUNCHER terminology.

A very simple STATECRUNCHER model is shown in the figure below:

![Figure 1. States, events, transitions and actions](image_url)
The above diagram models a system as having:

- three states: a, b and c
- four events: α, β, γ, and δ
- four transitions: t1, t2, t3 and t4
- three actions: fire δ and trace("ab") and v=v+2

At any one time, a system modelled by the above state-transition diagram will be in one and only one state. That state is called the occupied (or active) state. The others are vacant (or inactive). Since in general, in more complex models, several states can be occupied (due to parallelism and hierarchy), we speak of an occupancy configuration.

The main relationships between these are expressed as follows:

- an event triggers a transition, for example, α triggers t1.
- a transition occasions any actions on that transition. There are actions on transition t2.
- an action does one of the following:
  - fires an event, for example an action on transition t2 fires event δ. In the above model, nothing responds to δ, but if there were a parallel part of the statechart, or even another transition from state b triggered by event δ, the response would be made.
  - generates a trace
  - makes an assignment to a variable

When a transition occasions an action, we may speak of the transition itself firing the event, generating the trace, or making the assignment, e.g. “transition t2 fires event δ”.

In STATECRUNCHER, an event may occur at any time, but a transition will only take place if the source state of the transition is occupied. STATECRUNCHER has commands to tell it to provide the set of all events and the set of transitionable events.

STATECRUNCHER traces are specific outputs on a transition that the modeller decides to record, so that the model can output them on request. They typically correspond to observable outputs of a system under test, and are important in black-box testing, where the states and internally generated events cannot be observed. On this basis, in the above figure, only transition t2 produces output; the others are silent, and the only way to try to prove they have taken place is to drive the machine on through t2 by an event sequence.
Summary of approximate equivalences

<table>
<thead>
<tr>
<th>STATECRUNCHER</th>
<th>CCS</th>
<th>CSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>(state of an) agent. [Milner, p.19]: “Rather than distinguishing between two concepts - agent and state - we find it convenient to identify them, so that both agent and state will always be understood to mean agent in some state.”</td>
<td>process</td>
</tr>
<tr>
<td>event</td>
<td>action, handshake [Milner, p.17]</td>
<td>event</td>
</tr>
<tr>
<td>transition</td>
<td>transition, as in $A' \xrightarrow{\text{get}} A$ [Milner, p.38]</td>
<td>transition [Hoare, p.34], as a pictorial aid. Note: $x \rightarrow P$ describes an agent that can engage in event $x$ and become agent $P$.</td>
</tr>
<tr>
<td>action</td>
<td>probably best modelled as an output action</td>
<td>probably best modelled as an output action</td>
</tr>
<tr>
<td>trace</td>
<td>probably best modelled as an output action with which a user can engage</td>
<td>probably best modelled as an output action with which a user can engage</td>
</tr>
<tr>
<td>(sequence of processed events)</td>
<td>trace</td>
<td>trace</td>
</tr>
</tbody>
</table>

Table 1. Approximate equivalences in terminology

This table serves as a rough guide and an alert that the terminology is used differently in the different systems. The differences in approach will become more apparent as processes, and their composition, are discussed.

The ways in which nondeterminism is handled by the different systems is considered in section 3.
2. Composition of processes

In CSP and CCS, processes are combined by sharing events.

For CSP, Hoare says [Hoare p.65-66]

When two processes are brought together, the usual intention is that they will interact
with each other. These interactions may be regarded as events that require simultaneous
participation of both the processes involved.

The CSP operator for composition is \( \parallel \). The expression \( P \parallel Q \), is initially introduced for the case
where processes \( P \) and \( Q \) have the same alphabet [Hoare, p.66 1.8], i.e. the same set of events,
though this is relaxed in a generalisation [Hoare, p.69]. We will adopt the generalised version
of the operator in our discussions that follow as in so many realistic examples interacting
processes only share some of their events, namely the ones where they engage each other.
(Hoare perhaps unwittingly uses the generalised operator before introducing it, in his example
\( X \square \) [Hoare, p.66], where the alphabet of \( F O O L C U S T \) lacks event \( \text{small} \), which is in the
alphabet of \( V M C \) [Hoare, p.30]). More than two processes can be assembled using this
commutative and associative operator, e.g. \( P \parallel Q \parallel R \).

CSP also has an interleaving operator \( \parallel \parallel \), [Hoare, p.119]. In the expression \( P \parallel \parallel Q \), only one
process will engage in any action. If both processes can engage in an action, a
nondeterministic choice is made between them. There is no notion of processes engaging one
another.

In CCS, the composition symbol is \( \mid \), as in \( \text{Jobber} \mid \text{Hammer} \), [Milner, p.29], where these
particular agents share events for picking a hammer up and putting a hammer down. It is
possible to have several instances of one agent, giving an expression such as \( \text{Jobber} \mid \text{Jobber} \mid \text{Hammer} \).
CCS allows two (and only two) processes to synchronise by performing an action
and a complementary action together (e.g. \( c \) and \( \bar{c} \)), regarded as the handshake action \( \tau \).
Milner describes the handshake and composition operator "\( \mid \)" along the following lines
[Milner, p.39]:

\[
\text{if } A' \xrightarrow{\tau} A \text{ and } B \xrightarrow{\tau} B' \\
\text{then} \\
A' \parallel B \xrightarrow{\tau} A \parallel B'
\]

The event \( \tau \) is internal to the composite agent [Milner, p.39], and it is used to describe the
internal synchronisation action of any pair of complementary actions.
We note that in CCS, event $\tau$ does not necessarily take place when it potentially can. The composite agent may perform a $\tau$ action which results from $(c,\bar{c})$ communication between its components [Milner, p.40].

Restriction on $c$, (and so implicitly on $\bar{c}$), which is denoted by $\\{c\}$ or just $\{c\}$, excludes independent execution of $c$ and $\bar{c}$. It is a nondeterministic eventuality as to whether event $\tau$ is actually performed.

STATECRUNCHER will allow parallel parts of a statechart to share events, but this is not the same as CCS synchronisation, because there is no notion of event complements. STATECRUNCHER composition can best be achieved with a fired-call-event / fired-return-event paradigm, as follows. We then consider this composition paradigm in relation to process algebras.

**STATECRUNCHER's composition paradigm**

The standard paradigm for composing software components using STATECRUNCHER is to regard one component as a client (or caller) and one as a server (or callee). An event is fired by the client to call the server, and a return event is fired by the server to the client.

This has been elaborated on in detail, with some novel ideas, in [StCrFunMod].

The following figure illustrates the principle:

![Diagram](image)

**Figure 2. Client-server composition in STATECRUNCHER**

STATECRUNCHER's composition paradigm is closely analogous to the function call and return of imperative languages such as 'C'. The making of the function call is modeled by a fired event, the response to this is modeled by a transition on the event that was fired. The return statement is modeled by fired return event, and the response to this is modeled by a transition on the return event. If there are many such calling sequences in a model, return names can be made unique to a server function by affixing the function name to the event (e.g. return_max) or by putting the return event in a sufficiently local scope (using
STATECRUNCHER's scoping capabilities - described in [StCrMain] - but not further discussed here).

From the initial configuration, when event $\alpha$ occurs, the client transitions to state $C2$ and fires event $\beta$. This causes the server to make a transition. In this example the server has immediately completed its work, and it immediately fires event $\text{return}$. This causes the client to transition to state $C3$. The whole sequence is regarded as atomic to STATECRUNCHER, in the sense that no other event can interrupt it.

In STATECRUNCHER, the interaction on event $\alpha$ definitely takes place. There is no nondeterminism involved as in the case of a $\tau$ event in CCS, where the transition only may take place. This is because we are typically modelling function calls and their return. However, if in CCS the only event that can take place is $\tau$, then it can be argued that it should be considered deterministic.

We would not expect event $\beta$ to be generated except by a client of the particular server. The name $\beta$ would typically correspond to a server function name. There might, however, be several clients. We consider that situation later.

If the composition is a server to some higher level component, then the $\alpha$/fire $\beta$ construction will be repeated at a higher level (e.g. $\delta$/fire $\alpha$). It need not concern us as it is a repeat under different names of what we have seen. Alternatively, $\alpha$ is at the top level and is user supplied.

The transition semantics are important to allow this paradigm to work. A transition is taken to completion before its actions are executed. This ensures that no participating transition is blocked by its source state not being ready (i.e. occupied) for execution. So the transition on event $\text{return}$ can take place because its source state $C2$ will be occupied.

The individual models of the client and server can be experimented with separately under STATECRUNCHER. But in the absence of, say, the server, an event $\text{return}$ for the client will need to be given at prompt level by the user. Events should be attached to a point of control and observation (PCO). Event $\beta$ and $\text{return}$ would be put on an inter-component PCO, which can only be used in module testing. Under integration testing, this PCO and the events on it become internalized, or restricted or hidden, in CCS/CSP terminology, as the composition only admits to events such as event $\alpha$. 

© Graham G. Thomason 2003-2004
The STATECRUNCHER composition paradigm is analogous to the composition of Process X-machines, as described in [Stannett]. The paper has:

```
.../setvalue(100,this) → ack_t/
```

**object-machine**

```
setvalue(x,who) / t = p / ack[who];
```

**class-machine**

**Figure 3.** PXM assignment to a static class variable by an object

The STATECRUNCHER analogue is:

```
composition
```

```plaintext
client

α / fire_setvalue(100) → C2 ack_serv / ...

C1

S1

server

setvalue(p) / t = p; fire ack_serv;
```

**Figure 4.** STATECRUNCHERS composition paradigm making an assignment

Here, we have not made the ack_serv event unique to the specific caller as in the paper (the **this** keyword). Since this server does not support recursion, the server can only be serving one client at a time, so it is sufficient for ack_serv to be unique to the **server**; it cannot then be confused with the acknowledgement from any other server serving a different function. In [StCrFunMod], we propose a composition mechanism for recursive state machines, where the returned acknowledgement need not have a unique name at all, and targets its caller by means of scoping operators.

The STATECRUNCHER composition paradigm has been used to compose models of Koala components [Koala]. Koala is a static component binding tool used by Philips for embedded software. STATECRUNCHER is being used to test some Koala television components. In Koala representation, the component binding would be drawn thus:

**Figure 5.** Koala components
A more realistic server in practice

Figure 2 is conceptually the simplest possible example of client server composition. There could have been additional states and transitions in the server before the return event was given, in which case the client would be in state \( C2 \) for a while until the server was able to fire \( return \). In such a case, the server might look like this:

It would be normal for a server to end up in its default state when a client has been served and returned to, as follows:

Referring back to Figure 2: we do not return to the default state (S1); instead we are in a different state (S2) after the call, as this makes the calling paradigm as such a little clearer. One could think of the server as requiring some form of reset before it can be used again (not shown in the model).

Parameters can be passed back and forth by means of STATECRUNCHER's parameterized event mechanism. The issues of multiple clients, unique naming, and re-entrant or recursive calls is dealt with in [StCrFunMod].

Under this general system, a model of the server can be combined with \( any \) client that calls it with the agreed event \( \beta \) and which expects a return event \( return \). Similarly the client could be combined with a different server as long as the interface was defined in the same way.

STATECRUNCHER's composition paradigm and process algebras

Let us examine the properties of Figure 2 and consider how to model it in a process algebra.

It has three STATECRUNCHER events, \( \alpha, \beta \) and \( return \). It has two STATECRUNCHER actions, \( fire \beta \) and \( fire \ return \). Questions we will be considering are:

- Should the \( fire \) actions be considered events in a process algebra?
• If so, should some of the events be paired off, into (event, complementary-event) pairs?
• Which events should be restricted (recalling that return and return are on an inter-component PCO)?

We first consider the composition from a CCS perspective.

The composition could be modeled using the CCS ~ combinator [Milner, p.68], giving Client~Server. We show this using the CCS port diagrams [Milner, p.17], which are similar to Koala diagrams.

This composition can be defined by:

\[
\text{Client-Server} \equiv (\text{Client}[\text{mid1}/\text{return}, \text{mid2}/\overline{\beta}] \mid \text{Server}[\text{mid1}/\overline{\text{return}}, \text{mid2}/\beta])\backslash\{\text{mid1, mid2}\}
\]

We see that a fired event on a transition becomes an output event in a CCS model. Where CCS restricts events, STATECRUNCHER allows for them to be labeled as inter-component (i.e. internal after composition) by means of a PCO. In this way, a test generator (or Primer), when communicating with STATECRUNCHER, can be instructed whether or not to exercise these events. Event α would be on a global PCO, or at on a PCO denoting a higher level of component aggregation.

CCS allows for replacement of simultaneous complementary events by τ, the “perfect action” [Milner p.39]. In our model, the transitions on β and ̅β would be replaced by τ. When α takes place, τ must follow; nothing external can intervene (as it would spoil the paradigm). This is
in accordance with CCS semantics, for although \( \tau \) can lead to nondeterminism in an expression with a leading \( \tau \) term such as
\[
A \triangleq \alpha.A + \tau.b.A \quad \text{[Milner, p.42]}
\]
it is nevertheless permissible to eliminate \( \tau \) when preceded by another event:
\[
\alpha.\tau.P \triangleq \alpha.P \quad \text{[Milner, p.41]}
\]
so we can be sure that \( \tau \) takes place in our composition after event \( \alpha \).

By analogy with CCS, the STATECRUNCHER's \( \text{fire} \beta \) and transition in response to \( \beta \) are as good as simultaneous. This is a fair way to view STATECRUNCHER, since the transition semantics do not allow an intervening event. So we see a close parallel with CCS's notion of synchronization.

**What if there are several clients?**

Other clients of Server can also exist, but not be used simultaneously if there is just one instantiation of the server. Simultaneous outstanding server calls require the recursive state machine techniques of [StCrFunMod]. But provided the server is used sequentially, a STATECRUNCHER construction such as the following is useful:

![Diagram of several clients](image-url)

**Figure 10. Several clients**
In CCS, this can be modeled as follows:

![Multiple Clients in CCS - Port diagram](image1)

**Figure 11.** Multiple Clients in CCS - Port diagram

![Multiple Clients composed in CCS - Port diagram](image2)

**Figure 12.** Multiple Clients composed in CCS - Port diagram

In this case we have the CCS composition

\[(\text{Client1} \mid \text{Client2} \mid \text{Server}) \{ \beta, \text{return} \}\]

The Server can synchronize with either client, as in the single client case. The clients never synchronize with each other.

**What if there are several servers?**

A server typically represents a ‘C’-like function, and functions have unique names, and it is this name, \(\beta\) say, that will be in the fire \(\beta\) construction in STATECRUNCHER. So it is unlikely that the composition construction will be used with several servers - the system is rather nonsensical. Were this to be the case, however, the fire \(\beta\) action would synchronize with all servers. This cannot be modeled directly in CCS, as only two processes can synchronize. The fire return construction on return from the servers would be performed.
redundantly from all but the first server. This configuration does not appear to have any practical application.

**CCS state view**
The model of Figure 2 can be represented as follows in CCS, but we introduce additional states between the execution of α and the firing of β.

**Client**

C1 ≜ α.C1X  
C1X ≜ β.C2  
C2 ≜ return.C3

**Server**

S1 ≜ β.S1X  
S1X ≜ return.S2

The composition restricts on β and return: (and so also on their complements)

COMP ≜ C1 | S1 \{β, return\}

The composition is shown in the figure below.

![CCS state transition diagram of client-server model](image)

**Figure 13. CCS state transition diagram of client-server model**
Internal \( \tau \) events arising from \( \beta/\bar{\beta} \) and \( \text{return}/\text{return} \) events cause transitioning across states C1X, C2 and S1X, making them unobservable externally.

**Modeling the interaction in CSP**

The processes that engage are similar to those of CCS, but without complementary actions. If the processes had to share the same alphabet, we would compose with \( C1X|S1 \) in the diagram below. The complete composition would be a process \( \alpha \rightarrow (C1X|S1) \). But with the generalized composition operation, we can compose with \( C1|S1 \). The generalized composition operator would be essential if the client or server had additional states with their own events as in Figure 6 and Figure 7.

---

**Figure 14. CSP state transition diagram of client-server model**

The main differences in the approaches to composition are:

**In CCS:**
- Only 2 processes can participate in an interaction. They do this with complementary actions, which can be internalized into the internal event \( \tau \).
- The internal event \( \tau \) may or may not take place and so gives rise to nondeterminism.

**In CSP:**
- There is no distinction between an event and its complement. Using the generalized composition operator as discussed, any number of processes with at least one some common event can be composed, but then all *must* participate in any such common event.
• So if one component of a composition is not at some stage able to respond to an event for
the interaction, it will prevent the interaction.

• There is no ε nondeterminism.

In STATECRUNCHER:
• There is no symmetry between and fire β and β.
  ° There can be several places where β is generated (including the user).
  ° There can be several transitions triggered by β.
  ° Some of these may have nothing to do with the composition. However, the event
    name would typically be reserved for the composition. It could be put on an inter-
    component PCO as a means of indicating that it is not available for independent
    generation.

• When β is generated, all transitions triggered will in principle take place, though they can
  be invalidated if at execution time due to preceding actions if their source state has been
  vacated or their guard condition has become false.
3. Parallelism other than call/return composition

The following examples show some situations that can arise with parallel systems, where the separate machines may influence the other's behaviour in a way other than an engagement in the sense considered in the previous section. We discuss them from the STATECRUNCHER perspective.

Figure 15. Simple parallelism

This machine represents a composition of clusters $a$ and $b$ in parallel. Cluster $b$ is drawn with its own cluster boundary for clarity as to the tail of the transition on $\gamma$. In the initial state, in states $a_1$ and $b_1$, the machine can respond to event $\alpha$. The result will be that the clusters are in states $a_2$ and $b_2$. This model may be appropriate under some circumstances, but there has been no notion of interaction or synchronisation.

Event $\alpha$ is not always processed in both clusters. If after first processing event $\alpha$, we proceed to process event $\gamma$, the occupancy configuration is $\{a_2, b_1\}$. Now event $\alpha$ will only cause a transition in cluster $b$. 

© Graham G. Thomason 2003-2004
In fact, STATECRUNCHER interprets the above model as a race. Interleavings will be created. If we add some STATECRUNCHER actions to the transitions, this becomes apparent.

\[
\begin{align*}
\alpha / v = v * 10 + 1 \\
\beta
\end{align*}
\]

Figure 16. Simple parallelism as a race

In Figure 16, there is a variable \( v \) initialised to 0. The transition on \( \alpha \) from \( a1 \) causes a digit 1 to be appended to the value of \( v \). The transition on \( \alpha \) from state \( b1 \) causes a digit 2 to be appended to the value of \( v \). The result is 12 or 21 depending on the interleaving, i.e. who wins the race. STATECRUNCHER's nondeterminism handling produces a set of results, and so produces both values. Although we call this race nondeterminism, it is equivalent to fork nondeterminism in a flattened state space:

\[
\begin{align*}
\alpha / v = v * 10 + 2 \\
\beta
\end{align*}
\]

Figure 17. Part of flattened state space of the race model

The states are the Cartesian product of state occupancies in members \( a \) and \( b \) and with the values of variable \( v \). The nondeterminism on event \( \alpha \) from the initial state is seen as a fork.
The decision to evaluate conditions (guards) on transitions prior to executing them can lead to a blocked start, as in the following model.

![Figure 18. Blocked start](image)

Each transition on $\alpha$ has a condition that the other cluster must be in its default state. So it appears that from the initial state of the composition, both transitions can take place. But since the transitions are executed sequentially, one will invalidate the other. STATECRUNCHER will produce two outcomes, one in \{a1,b2\} and one in \{a2,b1\}. The reasons for the choice of semantics are explored in the main thesis, but we give another example here showing why transitions cannot just be started in parallel:

![Figure 19. Parallel start problem](image)

In this model, it appears that the two transitions on $\alpha$ can take place in parallel, but their target states are in conflict. They are members of the same cluster, and so cannot both become occupied. STATECRUNCHER's semantics are that after one transition, all conditions on the next are re-evaluated. The result is that STATECRUNCHER's two interleavings give a world in state $p$ and a world in state $q$ (set $s$ is exited completely on either transition).
4. Nondeterminism

Nondeterminism in STATECRUNCHER

Finite state machines (FSMs) are often described without reference to the hierarchical structures of a UML or STATECRUNCHER statechart (in UML: concurrent and non-concurrent composite states; in STATECRUNCHER: sets and clusters). This is because the hierarchical structure is just a convenient way of expressing a mathematically equivalent flattened state space. When the hierarchy is introduced, the terminology changes from FSMs to statecharts, but the two are equivalent. A state in the flattened state space is an element of the Cartesian product of parallel states in the statechart. Only leafstates need be considered, because the occupancies of their ancestors is a derivative of that of the leafstates. If the statechart contains history, variables and traces, then these must also present as terms in the Cartesian product in defining flattened states.

An example has already been given in Figure 16 and Figure 17 where the effect of event \(\alpha\) from the initial state is seen as race nondeterminism in the statechart and fork nondeterminism in the flattened state machine. The flattened state names are sequences (sequence brackets omitted for brevity). In the flattened state space, the only form of nondeterminism is fork nondeterminism.

From that example, it is seen that just as the hierarchical states of a statechart offer convenience in representing the state space, so some nondeterministic semantics (in this case, for the race) offer convenience in representing FSM nondeterminism. STATECRUNCHER simply structures the nondeterminism into various categories that are easy to visualize in a statechart.

STATECRUNCHER supports the following forms of structured nondeterminism:
- fork
- race
- set-transit
- set action
- set meta-event
- fired event (or broadcast event) nondeterminism

These are described in detail in [StCrMain]. After processing an event STATECRUNCHER produces a world per distinct state configuration, which, in flattened state space terms, is equivalent to a world for every possible resultant flattened state.
We develop the notion of a world more formally, working from the definition of a NFSM (Nondeterministic Finite State Machine) given by [Hierons]:

An NFSM $M$ is defined by a tuple $(S, s_i, h, X, Y)$ in which
- $S$ is a set of states
- $s_i$ is the initial state
- $h$ is the state transition function
- $X$ is the input alphabet
- $Y$ is the output alphabet

Given an NFSM $M$, $S_M$ shall denote the state set of $M$. When $M$ receives an input value $x \in X$, while in state $s \in S$, a transition is executed producing an output value $y \in Y$ and moving $M$ to some state $s' \in S$. The function $h$ gives the possible transitions and has the type $S \times X \rightarrow \mathcal{P}(S \times Y)$ where $\mathcal{P}$ denotes the power set operator. ... An NFSM $M$ is completely specified if, for each $s \in S$ and $x \in X$, $|h(s,x)| \geq 1$. $M$ is deterministic if for each $s \in S$ and $x \in X$, $|h(s,x)| \leq 1$.

What in Hierons' description is the notion of $M$ being in state $s$, is to STATECRUNCHER having an occupancy configuration $s$, and other dynamic properties, where an occupancy configuration gives the occupancy (occupied or vacant) of every state. Several states can be occupied, due to parallelism (modelled by a STATECRUNCHER set), and hierarchy (the fact that a parent of an occupied state is also an occupied state). Remark: the occupancy of non-leaf states can be derived from that of their child states (by the set and cluster rules), so, given the hierarchical structure, the occupancy configuration need only explicitly comprise the set of occupied leaf states.

The 'other dynamic properties' which $s$ must comprise are cluster history and variable values.

In our definitions below, we define $\mathcal{F}(A \times B) \subseteq \mathcal{P}(A \times B)$ to be the set of all functions from $A$ to $B$.

A STATECRUNCHER statechart is therefore $(C, V, P, s_i, v_i, p_i, X, Y, h)$ where
- $C$ is a hierarchy of states (sets, clusters and leaf states), from which we can easily derive
  - $S$, the set of all states
  - $P$, the set of all clusters, $P \subseteq S$
- $V$ is a set of variables
- $s_i$ is the initial state
- $v_i$ is a function giving the initial variable values, $V \rightarrow Z$, where $Z$ is the set of integers
- $p_i$ is a function giving the initial history values per cluster, $S \rightarrow S$
- $X$ is the input alphabet (a set of events in STATECRUNCHER)
- $Y$ is the output alphabet (a set of trace elements in STATECRUNCHER)
- $h$ is the state transition function

$$h : [S \times \mathcal{F}(V \times Z) \times \mathcal{F}(P \times S)] \times X \rightarrow \mathcal{P}([S \times \mathcal{F}(V \times Z) \times \mathcal{F}(P \times S)] \times Y),$$

where the $\mathcal{F}(V \times Z)$ term represents all the variables with their values.
The domain and range of $h$ can be represented as

$$
\text{domain}(h) : [S \times \mathcal{F}(V\times Z) \times \mathcal{F}(P\times S)] \times Y = W \times X
$$

$$
\text{range}(h) : \mathcal{P}(S \times \mathcal{F}(V\times Z) \times \mathcal{F}(P\times S)] \times Y) = \mathcal{P}(W \times Y)
$$

When an event is processed in many worlds, a new set of worlds is produced. To represent this, we define a multi-input-world transition function:

$$
H : \mathcal{P}(W \times X) \rightarrow \mathcal{P}(W \times Y)
$$

$$
H(A) = \bigcup_{B \in A} h(B)
$$

In a practical situation, the elements of the range of $H$ will all contain the same event in all the Cartesian product terms.

Remark: in the actual STATECRUNCHER implementation, traces also distinguish worlds, so we should strictly say that dynamic the configuration $d$ of a statechart is of type

$$
S \times \mathcal{F}(V\times Z) \times \mathcal{F}(P\times S) \times Y^*
$$

where $Y^*$ is the set of strings consisting of elements of $Y$, (including the empty sequence). So this could be considered to be the actual type of the range of the transition function $h$. However, the most efficient mode of operation is to clear traces and merge worlds between processing events; if this is not done, old and new traces are concatenated. Traces do not impinge on the transition algorithm. With this understanding, we discount the traces in a dynamic state, so we can more closely map to the description given by Hierons.

**Comparison of nondeterminism**

We take an example of fork nondeterminism:

![Figure 20. Simple fork nondeterminism](image)

If state $a$ is occupied, STATECRUNCHER offers the user a choice of transitionable events:

- event $\alpha$
- event $\beta$
In STATECRUNCHER terminology, we say that event $\alpha$ leads to nondeterminism. STATECRUNCHER takes care of the nondeterministic outcomes without user interaction, by returning the set of all possible outcomes. If event $\alpha$ is selected, two worlds are produced, one in state $b$ and one in state $c$, as described in [StCrMain]. In general there will be more than one world beforehand in which to process an event, and the event is processed in all of them.

**Nondeterminism in CSP**

In CSP, a choice between different events (e.g. $\alpha$ and $\beta$) is expressed by the choice operator ($\mid$). So one process may be defined by

$$ (\alpha \rightarrow P \mid \beta \rightarrow Q). $$

This is not nondeterminism, since the environment can control such a process by the event given.

The choice operator ($\mid$) is not an operator on processes [Hoare, p31]. It is syntactically incorrect in CSP to write

$$ (\alpha \rightarrow P \mid \alpha \rightarrow Q). $$

A process that behaves like $P$ or $Q$ where the environment has no control over the choice, is written using the nondeterministic or operator ($\sqcap$), [Hoare p.102], which Schneider calls the internal choice operator [Schneider, p.24]. We can write

$$ (\alpha \rightarrow P) \sqcap (\alpha \rightarrow Q). $$

CSP has another potentially nondeterministic operator, the general choice operator ($\sqcup$) [Hoare, p.106], which Schneider calls the external choice operator [Schneider, p.20]. The expression

$$ (P \sqcup Q) $$

denotes a process which the environment can control, provided this is done by the first event. If only $P$ can engage with the event, then $P$ is selected. If only $Q$ can engage with the event, then $Q$ is selected. If both can engage with the event, then the choice is nondeterministic.

As mentioned previously, CSP has an interleaving operator $||$, and in the expression $P||Q$, if both processes can engage in an action, a nondeterministic choice is made between them. But unlike with $(P \sqcup Q)$, no process is discarded, and the interleaving of two processes remains.

Compare again the ordinary CSP composition operator $||$, whereby in $P||Q$, both processes must participate if the event is in both their alphabets.

We see that $(P \sqcup Q)$, is nondeterministic, $(P \sqcup Q)$ and $(P || Q)$ can be nondeterministic, and $(P || Q)$ is deterministic (inasmuch as $P$ and $Q$ are themselves deterministic).
Nondeterminism in CCS

CCS combines two agent expressions with the summation operator (+). This can be nondeterministic. From [Milner, p.20]:

The agent \( P + Q \) behaves either like \( P \) or like \( Q \); as soon as one performs its first action, the other is discarded. Often the environment will only permit one of these alternatives [...]. But if both alternatives are permitted, then \( P + Q \) is non-deterministic; that is, it may behave like \( P \) on one occasion and like \( Q \) on another.

CCS [Milner, p85] allows defining equations such as

\[ B \equiv a.B1 + a.B1' \]

where the same action occurs in more than one term on the right hand side.

CCS has additional nondeterminism on agent composition, because the internal transition \( \tau \) may or may not occur [Milner p.40]. There can be several event-complement pairs that can give rise to different internal transitions. This means that several combinations of nondeterminism are possible:

![Figure 21. CCS combinations of nondeterminism](image)

Figure 21. CCS combinations of nondeterminism
5. Concluding remarks

Concluding remarks on composition of processes

CCS is rather different to CSP and STATECRUNCHER, in that only two processes can participate in an interaction, but the event - event complement concept does match up with the STATECRUNCHER fired event mechanism for composition when there is one or more clients and one server. The CCS $\tau$ event may give rise to nondeterminism, where none would be present in the STATECRUNCHER composition paradigm as presented.

CSP does not have the two-process restriction of CCS, but there is no direct STATECRUNCHER counterpart to the way in which one participating process can prevent others from engaging (which happens in CSP when that process cannot respond to a particular event in the shared alphabet). Such a prevention mechanism is not required for simple client-server composition, (but constructs can be created as necessary - for a semaphore see the example of the dining philosophers in [StCrMain]).

Concluding remarks on nondeterminism

As mentioned, the forms of STATECRUNCHER nondeterminism (e.g. race nondeterminism), are simply convenient constructs for use in a structured way when dealing with a statechart structure containing hierarchy (clusters and sets) and concurrency (sets). These constructs are all effectively fork nondeterminism in an equivalent flattened model, and so are nothing new for the purposes of this comparison.

It is seen that the CCS summation operator (+) and the CSP internal choice operator (\null) express nondeterminism in the STATECRUNCHER sense, but the operands must be processes not events, so the model of figure Figure 20 has to be expressed as separate processes rather than one process. This is effectively no more than a syntactic requirement of CCS and CSP. The semantics of CCS and CSP can lead to further nondeterministic situations, where a STATECRUNCHER model would typically contain a nondeterministic fork.
References

STATECRUNCHER documentation and papers by the present author

Main Thesis
StCrMain The Design and Construction of a State Machine System that Handles Nondeterminism

Appendices

Appendix 1 StCrContext Software Testing in Context
Appendix 2 StCrSemComp A Semantic Comparison of STATECRUNCHER and Process Algebras
Appendix 3 StCrOutput A Quick Reference of STATECRUNCHER's Output Format
Appendix 4 StCrDistArb Distributed Arbiter Modelling in CCS and STATECRUNCHER - A Comparison
Appendix 5 StCrNim The Game of Nim in Z and STATECRUNCHER
Appendix 6 StCrBiblRef Bibliography and References

Related reports

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

[Bruns] Glenn Bruns
Distributed Systems Analysis with CCS

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

[Harel 87] D. Harel et al.
On the Formal Semantics of Statecharts
Logic in Computer Science, 2nd Annual Conference, 1987, pp.54-64

[Hierons] R.M. Hierons
Adaptive testing of a deterministic implementation against a
nondeterministic finite state machine.
1998

[Hoare] C.A.R. Hoare
Communicating Sequential Processes

[Koala] R. van Ommering, F. van der Linden, J. Kramer, J. Magee
The Koala Component model for Consumer Electronics Software

[Milner] Robin Milner
Communication and Concurrency

[Schneider] Steve Schneider
Concurrent and Real-time Systems, The CSP Approach
Mike Stannett and A.J.H. Simons  
Complete Behavioural testing of Object-Oriented Systems using CCS-Augmented X-Machines  
Test Report CS-02-04, Dept. of Computer Science, United Kingdom  

The Côte de Réyste Project delivers the TorX tool  
http://fmt.cs.utwente.nl/CdR  

The Object Management Group website is: http://www.omg.org  
UML specifications are available from this website.
Appendix 3

A Quick Reference of STATECRUNCHER's Output Format

An Appendix to the Thesis
The Design and Construction of a State Machine System that Handles Nondeterminism

Graham G. Thomason
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”

UniS

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

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

References
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]

© Graham G. Thomason 2003-2004
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.

statechart sc(s)
PCO pcol;
PCO s.d.pco2;
event alpha;
event rho,rhol;
event beta,returnOpcol;
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; rhol->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] = OCC [] **
10 leafstate a2 [a,s,sc] = VAC []
10 leafstate a3 [a,s,sc] = VAC []
10 cluster b [s,sc] = OCC [] **
10 leafstate b1 [b,s,sc] = OCC [] **
10 leafstate b2 [b,s,sc] = VAC []
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 [] **

© Graham G. Thomason 2003-2004
cluster a [s,sc] = OCC [] **  
leafstate a1 [a,s,sc] = VAC []  
leafstate a2 [a,s,sc] = VAC []  
leafstate a3 [a,s,sc] = OCC [] **  
cluster b [s,sc] = OCC [] **  
leafstate b1 [b,s,sc] = VAC []  
leafstate b2 [b,s,sc] = OCC [] **  
cluster c [s,sc] = OCC [] **  
leafstate c1 [c,s,sc] = VAC []  
leafstate c2 [c,s,sc] = OCC [] **  
cluster d [s,sc] = OCC [] **  
leafstate d1 [d,s,sc] = OCC [] **  
leafstate d2 [d,s,sc] = VAC []  

VAR INTEGER p [d,s,sc] =0  
VAR INTEGER q [d,s,sc] =0  
VAR STRING str [sc] = [99,100] =cd  
VAR INTEGER v [sc] =21  
TRACE = [ab, cd]  
TREV [[gamma, [d,s,sc]], 2, [[e,0,1,2,5,6], [r,0,3]], [pco2, [d,s,sc]]]  
TREV [[rho, [sc]], 0, [], []]  
TREV [[rhol, [sc]], 0, [], []]  

outworlds=[10,18]  
number of outworlds=2

SC: I:

Explanation of the output

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

leafstate a1 [a,s,sc] = OCC [] **  
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

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

TRACE =[]
Transitionable events are given by TREV lines. Consider the transitionable events from the initial model configuration:

- $TREV[[\alpha, [sc]], 0, [], []]$
- $TREV[[\beta, [sc]], 0, [], [pcol, [sc]]]$
- $TREV[[\gamma, [d, s, sc]], 2, [[e, 0, 1, 2, 5, 6], [r, 0, 3]], [pcol2, [d, s, sc]]]$
- $TREV[[\rho, [sc]], 0, [], []]$
- $TREV[[\rho1, [sc]], 0, [], []]$

The lines are of the form

\[ \text{WORLD TREV } \left[ \left[ \text{EVENT, EVENTSCOPE} \right], \text{NPARAMS, PARAM-RANGES, [PCO, PCOSCOPE]} \right] \]

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 $[\text{EVENT, EVENTSCOPE}]$ item is $\text{NPARAMS}$, 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 $\text{PCO}$ (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 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
Appendix 4

Distributed Arbiter Modelling in CCS and STATECRUNCHER - A Comparison

An Appendix to the Thesis
The Design and Construction of a State Machine System that Handles Nondeterminism

Graham G. Thomason
Distributed Arbiter Modelling in CCS and
STATECRUNCHER - A Comparison

Graham G. Thomason

Appendix to the Thesis “The Design and Construction of a State Machine System that Handles Nondeterminism”

UniS

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

In this paper we show how a system taken from the CCS literature can be modelled in STATECRUNCHER. An understanding of STATECRUNCHER is assumed, but for the purposes of this paper, most of STATECRUNCHER functionality will not seem strange to anyone familiar with UML dynamic modelling [UML], since that is the basis of the language.

We take a system that is neither too trivial nor too complex to serve as a good case study: the distributed arbiter system as described by Bruns [Bruns p.19]. For the definitive book on CCS by its designer, see [Milner].

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. Pre-requisite reading ................................................................................................................1

2. The distributed arbiter ..............................................................................................................2  
   2.1 Description of the problem ...............................................................................................2  
   2.2 STATECRUNCHER notation and conventions .............................................................4  
   2.3 The single arbiter ...............................................................................................................6  
   2.4 Two distributed arbiters ...................................................................................................9  
   2.5 Two distributed arbiters with users ...............................................................................12  
   2.6 Flattened models .............................................................................................................14  

3. The STATECRUNCHER distributed arbiter in CCS .........................................................25  

4. Source code of models ...........................................................................................................29  
   4.1 Source code of the single distributed arbiter [model t4300] ........................................29  
   4.2 Source code of the two distributed arbiters (John and Mary) [model t4301] .............31  
   4.3 Source code of the distributed arbiter with clients [model t4302] ..............................33  
   4.4 Source code of single flattened distributed arbiter [model t4310] ............................36  
   4.5 Source code of flattened distributed arbiters [model t4311] ........................................37  
   4.6 Source code of flattened distributed arbiter with clients [model t4312] .................39  

References .......................................................................................................................................41
1. Pre-requisite reading

Two separate appendices are pre-requisite reading to this appendix. They are:

- *A Semantic Comparison of STATECRUNCHER and Process Algebras* [StCrSemComp]
- *A Quick Reference of STATECRUNCHER's Output Format* [StCrOutput]

The first also describes differences in terminology between STATECRUNCHER and CCS, and compares their semantics and the way they compose separate state machines into a system. The nondeterministic features of STATECRUNCHER are discussed.
2. The distributed arbiter

2.1 Description of the problem

The purpose of this paper is to show how a system taken from the CCS literature can be modelled in STATECRUNCHER. We take a system that is neither too trivial nor too complex to serve as a good case study: The distributed arbiter system as described by Bruns [Bruns p.19]. For the definitive book on CCS by its designer, see [Milner].

The purpose of an arbiter is to manage a serially reusable resource, which we will henceforth just call the resource. If the resource is free, it can be allocated. If the resource is allocated, any other client has to wait (at least) until the resource is released.

Suppose there are two clients for a resource, and these clients run on separate machines in a network, and suppose that communication between the machines is regarded as expensive (probably in terms of time, affecting response times), so communication should be restricted to when it is essential. In this case, the combined arbiter can be constructed out of two single arbiters who share a token which gives the right to allocate the resource. Requests for the token, and replying to the requests by passing the token or saying not-OK will be only be performed if they are essential. So if a client on one machine obtains a resource, and then releases it and requests it again several times, without the other client requesting the resource, no traffic between the machines will ensue. For simplicity, the arbiter does not allow cancelling an unfulfilled request which has been placed for a resource - once a resource is requested, the client will either get it immediately or must wait for it. (That is how a normal program using a disk server will work, anyway). Figure 1 is a schematic of the distributed system.

![Figure 1. Distributed arbiters and clients for a resource](image_url)
The CCS notation and state-transition diagram given for a single distributed arbiter are given in [Bruns, p.21] - but we first work in native STATECRUNCHER mode in designing a distributed arbiter. We return to Bruns's model after we have shown the STATECRUNCHER models.

We start with a single arbiter, instances of which run on each distributed machine. We call this arbiter in isolation **Me**, and its counterpart **You**, as a kind of template of the arbiter, but later in compositions we will name them John and Mary.

The following colour coding for events and PCOs (points of control and observation) will be used:
- **green** for a user (i.e. client) event that does not in itself interact with an arbiter, but which may be the event which makes the client want to interact with an arbiter.
- **blue** for client-arbiter interaction events corresponding to **requesting**, **releasing** and **acquiring** a resource from an arbiter. The **requesting** and **releasing** events are ones a user would supply; **acquiring** is one that would be supplied to a user.
- **red** for inter-arbiter events
- **black** for internal events to a single arbiter (if used)

Client-arbiter events: the client of the single arbiter can supply events

- **MeReqRes** Client tells Me-arbiter that it requests a resource
- **MeRelRes** Client tells Me-arbiter that it is releasing a resource
- **MeAcqRes** The Me-arbiter tells the client that the resource has been acquired

The PCO for these events is **ClientMePco**.

Inter-arbiter events: the events that occur between the arbiter and its counterpart are:

- **MeReqTok** I request the token
- **MePass** I pass the token to you
- **MeSayNok** I tell you you can't have the token

The PCO for these events is **InterArbMePco**.

- **YouReqTok** You request the token
- **YouPass** You pass the token to me
- **YouSayNok** You tell me I can't have the token

The PCO for these events is **InterArbYouPco**.

A STATECRUNCHER model of a single arbiter may or may not make use of internal events. That is perhaps a matter of taste. In this paper, we show two approaches, one with internal events and one without. If the first one seems unnecessary, when you come to it on page 7, skip it by going to page 16. To the outside world, which includes the other arbiter and the client, the behaviour is identical.
The following internal events are used in the relevant models, such as that of Figure 4. They are of local scope, and do not need a Me prefix.

TryTok  
I ask another part of myself whether I have the token, and if not, I tell that part of myself to try and get it. The responses may be TryOk (I already have the token), YouSayNok (after asking you for the token, I get a negative response), or AcqTok (I ask you for the token and you pass it to me).

TryOk  
I tell another part of myself that the try for the token succeeded (because I already had the token).

AcqTok  
I tell another part of myself that the try for the token succeeded (because I could get it from the You-arbiter).

ResetWant  
I tell another part of myself that the You-arbiter need no longer be considered wanting to obtain the token from me.

These events have a null PCO (denoted by [] in STATECRUNCHER).

The arbiter was initially modelled by the author with explicit internal parallelism for
- the state of resource allocation: Idle, Requested, Waiting, Alloc(ated)
- whether the arbiter possesses the token or not: NotHaveTok(en), HaveTok(en)
- whether the arbiter actually needs the token: NotNeedTok(en), NeedTok(en)
- whether the other arbiter wants the token: OtherNotWantTok(en), OtherWantTok(en).

We can flatten (explore) such a model. The Cartesian product of states is potentially 4x2x2x2 = 32, but actually only 6 can exist under proper sequences of events. The flattened model, which is without internal parallelism is presented later, and as in STATECRUNCHER it produces less output, we will mainly use it. The reader may regard the flattened model as more intuitive from the start or prefer the internally-parallel approach; we will first show the model with internal parallelism.

2.2 STATECRUNCHER notation and conventions

A few details of the model notation and STATECRUNCHER semantics are now explained for convenience. Details are available in the main STATECRUNCHER reports.

STATECRUNCHER's composite states are called clusters and sets. A cluster corresponds to a UML non-concurrent composite state, and to Harel's XOR-states [Harel]. A set corresponds to a UML concurrent composite state, and to Harel's AND-states.

STATECRUNCHER's “after-landing” transition semantics are essential. By this we mean that the actions associated with a transition are carried out after the target states of the transition has been entered, and the pure transition as such is complete. (The alternative is to carry out transition actions in “mid-flight”). In Figure 2, if we are in states state1 and state8, and event alpha is given, the resultant states will be state3 and state9. This is the basis of client-server communication modelling: when the client processes to event alpha, it needs to call the server, which is done by firing event beta. The return event from the server is event gamma.
The whole process is carried out without allowing other events to intervene. (Under mid-flight semantics, the transition on gamma would not take place, because its source state, state2, would not be occupied).

![Diagram](image)

**Figure 2. “After landing” semantics**

STATECRUNCHER supports actions on transitions, as in Mealy machines, and actions on exiting or entering states (compare Moore machines). The only kind of action we are concerned with here is the **fired event**. (In other models, a variable assignment is a common action). The diagrammatic notation used here for on-entry actions is as follows with the arrow pointing into the state.

![Diagram](image)

**Figure 3. On entry symbol**

Transitions can be triggered by internal STATECRUNCHER events, - the exiting and entering of states in a parallel part of the model. Such events are denoted by `enter(state)` and `exit(state)`.

Conditional **transitions** have a condition in square brackets. Conditional actions are represented by `if (condition) action` with optionally `else action`. Tests can be made for the occupancy of a parallel state using the `in(state)` function.

States are addressed using scoping operators. In brief, these are:

- `$x$` go up a state then down into $x$
- `x%®y` go up until you reach $x$, then take $y$
- `a:x` descend through $a$ and $x$
- `:x` start at statechart level and take $x$

© Graham G. Thomason 2003-2004
They can be combined into expressions.

Traces in STATECRUNCHER (which are different in concept to those of CCS) are outputs that are observable at black-box level. They are generated in the model by the function \( \text{trace}(\text{expression}) \), which can occur in any STATECRUNCHER action.

The commands to STATECRUNCHER that we will be using are

?- crunccher. \hspace{1cm} \textit{enter the STATECRUNCHER read-process loop}

SC:mm \hspace{1cm} \textit{set modelname mode}

SC:run t4300 \hspace{1cm} \textit{run a model (by modelname, not filename, in this mode)}

SC:pe event \hspace{1cm} \textit{process an event}

SC:gc \hspace{1cm} \textit{get configuration}

SC:gt \hspace{1cm} \textit{get trace}

Reminder for users on how to exit the system

SC:quit \hspace{1cm} \textit{quit the STATECRUNCHER read-process loop}

?- halt. \hspace{1cm} \textit{exit Prolog}

2.3 The single arbiter

We now present the single arbiter, and show a session running it. Its STATECRUNCHER source, and that of other models, is given at the end of this paper.
Figure 4. Single arbiter [model t4300]

© Graham G. Thomason 2003-2004
Session with single arbiter [model t4300]

?- cruncher.
SC: mm
SC: run t4300
...
SC: gc

statechart sc
  set Me [sc] = OCC [] **
  cluster Res [Me, sc] = OCC [] **
  leafstate Idle [Res, Me, sc] = OCC [] **
  leafstate Requested [Res, Me, sc] = VAC []
  leafstate Waiting [Res, Me, sc] = VAC []
  leafstate Alloc [Res, Me, sc] = VAC []
  cluster Have [Me, sc] = OCC [] **
  leafstate NotHaveTok [Have, Me, sc] = OCC [] **
  leafstate HaveTok [Have, Me, sc] = VAC []
  cluster Need [Me, sc] = OCC [] **
  leafstate NotNeedTok [Need, Me, sc] = OCC [] **
  leafstate NeedTok [Need, Me, sc] = VAC []
  cluster OtherWant [Me, sc] = OCC [] **
  leafstate OtherNotWantTok [OtherWant, Me, sc] = OCC [] **
  leafstate OtherWantTok [OtherWant, Me, sc] = VAC []

TRACE =[]
TREV [[MeReqRes, [sc]], 0, [ClientMePco, [sc]]]
TREV [[TryTok, [sc]], 0, [], []]
TREV [[YouPass, [sc]], 0, [], [InterArbYouPco, [sc]]]
TREV [[YouReqTok, [sc]], 0, [], [InterArbYouPco, [sc]]]

outworlds=[2]
number of outworlds=1
SC: pe MeReqRes
SC: gc

statechart sc
  set Me [sc] = OCC [] **
  cluster Res [Me, sc] = OCC [] **
  leafstate Idle [Res, Me, sc] = VAC []
  leafstate Requested [Res, Me, sc] = OCC [] **
  leafstate Waiting [Res, Me, sc] = VAC []
  leafstate Alloc [Res, Me, sc] = VAC []
  cluster Have [Me, sc] = OCC [] **
  leafstate NotHaveTok [Have, Me, sc] = OCC [] **
  leafstate HaveTok [Have, Me, sc] = VAC []
  cluster Need [Me, sc] = OCC [] **
  leafstate NotNeedTok [Need, Me, sc] = VAC []
  leafstate NeedTok [Need, Me, sc] = OCC [] **
  cluster OtherWant [Me, sc] = OCC [] **
  leafstate OtherNotWantTok [OtherWant, Me, sc] = OCC [] **
  leafstate OtherWantTok [OtherWant, Me, sc] = VAC []

TRACE =[]
TREV [[AcqTok, [sc]], 0, [], []]
TREV [[TryOk, [sc]], 0, [], []]
TREV [[YouSayNok, [sc]], 0, [], [InterArbYouPco, [sc]]]
TREV [[TryTok, [sc]], 0, [], []]
TREV [[YouPass, [sc]], 0, [], [InterArbYouPco, [sc]]]
TREV [[YouReqTok, [sc]], 0, [], [InterArbYouPco, [sc]]]

outworlds=[5]
number of outworlds=1
SC: pe YouSayNok
SC: gc

statechart sc
  set Me [sc] = OCC [] **
  cluster Res [Me, sc] = OCC [] **
  leafstate Idle [Res, Me, sc] = VAC []
  leafstate Requested [Res, Me, sc] = VAC []
  leafstate Waiting [Res, Me, sc] = OCC [] **
2.4 Two distributed arbiters

We now compose a system from two single arbiters, as shown in the following figure:
Figure 5. Two distributed arbiters (John and Mary) [model t4301]
Session with two distributed arbiters [model t4301]

Output not shown in full, as it is rather lengthy, and equivalent to that of model t4311, shown later.

?- cruncher.
SC:mm
SC:run t4301
...
SC:pe GiveJohnTok
SC:pe MaryReqRes
SC:pe JohnReqRes
SC:gc

statechart sc

19 set Cmp [sc] = OCC [] **
19 set John [Cmp, sc] = OCC [] **
19   leafstate Idle [Res, John, Cmp, sc] = VAC []
19   leafstate Requested [Res, John, Cmp, sc] = VAC []
19   leafstate Waiting [Res, John, Cmp, sc] = OCC [] **
19   leafstate Alloc [Res, John, Cmp, sc] = VAC []
19   cluster Have [John, Cmp, sc] = OCC [] **
19   leafstate NotHaveTok [Have, John, Cmp, sc] = OCC [] **
19   leafstate HaveTok [Have, John, Cmp, sc] = VAC []
19   cluster Need [John, Cmp, sc] = OCC [] **
19   leafstate NotNeedTok [Need, John, Cmp, sc] = VAC []
19   leafstate NeedTok [Need, John, Cmp, sc] = OCC [] **
19   cluster OtherWant [John, Cmp, sc] = OCC [] **
19   leafstate OtherWantTok [OtherWant, John, Cmp, sc] = OCC [] **
19   leafstate OtherWantTok [OtherWant, John, Cmp, sc] = VAC []
19 set Mary [Cmp, sc] = OCC [] **
19   cluster Res [Mary, Cmp, sc] = OCC [] **
19   leafstate Idle [Res, Mary, Cmp, sc] = VAC []
19   leafstate Requested [Res, Mary, Cmp, sc] = VAC []
19   leafstate Waiting [Res, Mary, Cmp, sc] = VAC []
19   leafstate Alloc [Res, Mary, Cmp, sc] = OCC [] **
19   cluster Have [Mary, Cmp, sc] = OCC [] **
19   leafstate NotHaveTok [Have, Mary, Cmp, sc] = VAC []
19   leafstate HaveTok [Have, Mary, Cmp, sc] = OCC [] **
19   cluster Need [Mary, Cmp, sc] = OCC [] **
19   leafstate NotNeedTok [Need, Mary, Cmp, sc] = VAC []
19   leafstate NeedTok [Need, Mary, Cmp, sc] = OCC [] **
19   cluster OtherWant [Mary, Cmp, sc] = OCC [] **
19   leafstate OtherWantTok [OtherWant, Mary, Cmp, sc] = VAC []
19   leafstate OtherWantTok [OtherWant, Mary, Cmp, sc] = OCC [] **

TRACE = []
19 TREV [[AcqTok, [John, Cmp, sc]], 0, [], []]
19 TREV [[TryTok, [John, Cmp, sc]], 0, [], []]
19 TREV [[MaryPass, [Cmp, sc]], 0, [], [InterArbMaryPco, [Cmp, sc]]]
19 TREV [[MaryReqTok, [Cmp, sc]], 0, [], [InterArbMaryPco, [Cmp, sc]]]
19 TREV [[MaryRelRes, [sc]], 0, [], [ClientMaryPco, [sc]]]
19 TREV [[TryTok, [Mary, Cmp, sc]], 0, [], []]
19 TREV [[JohnReqTok, [Cmp, sc]], 0, [], [InterArbJohnPco, [Cmp, sc]]]
19 TREV [[ResetWant, [Mary, Cmp, sc]], 0, [], []]
19 TREV [[GiveJohnTok, [sc]], 0, [], []]

outworlds=[19]
number of outworlds=1
SC:pe MaryRelRes
...

© Graham G. Thomason 2003-2004
2.5 Two distributed arbiters with users

The following figure shows users (clients) composed into a system with two distributed arbiters:

![Diagram of two distributed arbiters with users](image)

**Figure 6. Two distributed arbiters with users (black box view) [model t4302]**

In this model the clients need the resource when they process events alpha and gamma. The clients release the resource on events beta and delta. We drive the system using these events only.
Session with two distributed arbiters with users

Reminder: the TRACE is read from right to left.

Read

\textbf{JAcq} as the user of arbiter John acquired the resource
\textbf{JRel} as the user of arbiter John released the resource
\textbf{MACq} as the user of arbiter Mary acquired the resource
\textbf{MRel} as the user of arbiter Mary released the resource

?- cruncher.
SC:mm
SC:run t4302
...
SC:gt
2  TRACE =[]
SC:pe  giveJohnTok
SC:gt
3  TRACE =[]
SC:pe  alpha
SC:gt
10  TRACE =[JAcq]
SC:pe  gamma
SC:gt
17  TRACE =[JAcq]
SC:pe  beta
SC:gt
36  TRACE =[MACq, JRel, JAcq]
SC:pe  delta
SC:gt
40  TRACE =[MRel, MACq, JRel, JAcq]

SC:gc
40  statechart sc
40  set Cmp [sc] = OCC [] **
40  set SysJ [Cmp, sc] = OCC [] **
40  set John [SysJ, Cmp, sc] = OCC [] **
40  cluster Res [John, SysJ, Cmp, sc] = OCC [] **
40  leafstate Idle [Res, John, SysJ, Cmp, sc] = OCC [] **
40  leafstate Requested [Res, John, SysJ, Cmp, sc] = VAC []
40  leafstate Waiting [Res, John, SysJ, Cmp, sc] = VAC []
40  leafstate Alloc [Res, John, SysJ, Cmp, sc] = VAC []
40  cluster Have [John, SysJ, Cmp, sc] = OCC [] **
40  leafstate HaveTok [Have, John, SysJ, Cmp, sc] = OCC [] **
40  leafstate HaveRes [Have, John, SysJ, Cmp, sc] = VAC []
40  cluster Need [John, SysJ, Cmp, sc] = OCC [] **
40  leafstate NotHaveTok [Need, John, SysJ, Cmp, sc] = OCC [] **
40  leafstate NotHaveRes [Need, John, SysJ, Cmp, sc] = VAC []
40  cluster OtherWant [John, SysJ, Cmp, sc] = OCC [] **
40  leafstate OtherWantTok [OtherWant, John, SysJ, Cmp, sc] = OCC [] **
40  leafstate OtherWantRes [OtherWant, John, SysJ, Cmp, sc] = VAC []
40  set SysM [Cmp, sc] = OCC [] **
40  set Mary [SysM, Cmp, sc] = OCC [] **
40  cluster Res [Mary, SysM, Cmp, sc] = OCC [] **
40  leafstate Idle [Res, Mary, SysM, Cmp, sc] = OCC [] **
40  leafstate Requested [Res, Mary, SysM, Cmp, sc] = VAC []
40  leafstate Waiting [Res, Mary, SysM, Cmp, sc] = VAC []

© Graham G. Thomason 2003-2004
2.6 Flattened models

The diagrams following show an alternative model to the distributed arbiter, using just a cluster. We first show the flattened states of the model of Figure 4, then a new model based on the flattened states.
Figure 7. The model flattened (explored, unfolded)
Figure 8. Single flattened distributed arbiter [model t4310]
Session with single flattened arbiter

| ~? cruncher.
SC:|: mm
SC:|: run t4310
...
SC:|: gc
2  statechart sc
2  cluster Me [sc] = OCC [] **
2  leafstate m1_IdleNoTok [Me,sc] = OCC [] **
2  leafstate m2_ReqdTok [Me,sc] = VAC []
2  leafstate m3_Waiting [Me,sc] = VAC []
2  leafstate m4_AllocPlain [Me,sc] = VAC []
2  leafstate m5_IdleWithTok [Me,sc] = VAC []
2  leafstate m6_AllocOtherWant [Me,sc] = VAC []
2  TRACE =[]
2  TREV [[MeReqRes,[sc]],0,[],[ClientMePco,[sc]]]

outworlds=[2]
number of outworlds=1
SC:|: pe JohnReqRes
SC:|: gc
2  statechart sc
2  cluster Me [sc] = OCC [] **
2  leafstate m1_IdleNoTok [Me,sc] = OCC [] **
2  leafstate m2_ReqdTok [Me,sc] = VAC []
2  leafstate m3_Waiting [Me,sc] = VAC []
2  leafstate m4_AllocPlain [Me,sc] = VAC []
2  leafstate m5_IdleWithTok [Me,sc] = VAC []
2  leafstate m6_AllocOtherWant [Me,sc] = VAC []
2  TRACE =[]
2  TREV [[MeReqRes,[sc]],0,[],[ClientMePco,[sc]]]

outworlds=[2]
number of outworlds=1
SC:|: pe MarySayNok
SC:|: gc
2  statechart sc
2  cluster Me [sc] = OCC [] **
2  leafstate m1_IdleNoTok [Me,sc] = OCC [] **
2  leafstate m2_ReqdTok [Me,sc] = VAC []
2  leafstate m3_Waiting [Me,sc] = VAC []
2  leafstate m4_AllocPlain [Me,sc] = VAC []
2  leafstate m5_IdleWithTok [Me,sc] = VAC []
2  leafstate m6_AllocOtherWant [Me,sc] = VAC []
2  TRACE =[]
2  TREV [[MeReqRes,[sc]],0,[],[ClientMePco,[sc]]]

outworlds=[2]
number of outworlds=1
SC:|: pe MaryPass
SC:|: gc
2  statechart sc
2  cluster Me [sc] = OCC [] **
2  leafstate m1_IdleNoTok [Me,sc] = OCC [] **
2  leafstate m2_ReqdTok [Me,sc] = VAC []
2  leafstate m3_Waiting [Me,sc] = VAC []
2  leafstate m4_AllocPlain [Me,sc] = VAC []
2  leafstate m5_IdleWithTok [Me,sc] = VAC []
2  leafstate m6_AllocOtherWant [Me,sc] = VAC []
2  TRACE =[]
2  TREV [[MeReqRes,[sc]],0,[],[ClientMePco,[sc]]]

outworlds=[2]
number of outworlds=1
SC:|:
Figure 9. Two flattened distributed arbiters [model t4311]
Session with two flattened distributed arbiters

| ?- cruncher.
SC|:  mm
SC|:  run t4311
...
SC|:  pe GiveJohnTok
SC|:  gc
3 statechart sc
3    set Cmp [sc] = OCC [] **
3    cluster John [Cmp,sc] = OCC [] **
3      leafstate m1_IdleNoTok [John,Cmp,sc] = VAC []
3      leafstate m2_ReqTok [John,Cmp,sc] = VAC []
3      leafstate m3_Waiting [John,Cmp,sc] = VAC []
3      leafstate m4_AllocPlain [John,Cmp,sc] = VAC []
3      leafstate m5_IdleWithTok [John,Cmp,sc] = OCC [] **
3      leafstate m6_AllocOtherWant [John,Cmp,sc] = VAC []
3    cluster Mary [Cmp,sc] = OCC [] **
3      leafstate m1_IdleNoTok [Mary,Cmp,sc] = OCC [] **
3      leafstate m2_ReqTok [Mary,Cmp,sc] = VAC []
3      leafstate m3_Waiting [Mary,Cmp,sc] = VAC []
3      leafstate m4_AllocPlain [Mary,Cmp,sc] = VAC []
3      leafstate m5_IdleWithTok [Mary,Cmp,sc] = VAC []
3      leafstate m6_AllocOtherWant [Mary,Cmp,sc] = VAC []
3  TRACE =[]
3  TREV [[MaryReqTok,[Cmp,sc]],0,[],[InterArbMaryPco,[Cmp,sc]]]
3  TREV [[JohnReqRes,[sc]],0,[],[ClientJohnPco,[sc]]]
3  TREV [[MaryReqRes,[sc]],0,[],[ClientMaryPco,[sc]]]
3  TREV [[GiveJohnTok,[sc]],0,[],[]]

outworlds=3
number of outworlds=1
SC|:  pe MaryReqRes
SC|:  gc
6 statechart sc
6    set Cmp [sc] = OCC [] **
6    cluster John [Cmp,sc] = OCC [] **
6      leafstate m1_IdleNoTok [John,Cmp,sc] = OCC [] **
6      leafstate m2_ReqTok [John,Cmp,sc] = VAC []
6      leafstate m3_Waiting [John,Cmp,sc] = VAC []
6      leafstate m4_AllocPlain [John,Cmp,sc] = VAC []
6      leafstate m5_IdleWithTok [John,Cmp,sc] = VAC []
6      leafstate m6_AllocOtherWant [John,Cmp,sc] = VAC []
6    cluster Mary [Cmp,sc] = OCC [] **
6      leafstate m1_IdleNoTok [Mary,Cmp,sc] = VAC []
6      leafstate m2_ReqTok [Mary,Cmp,sc] = VAC []
6      leafstate m3_Waiting [Mary,Cmp,sc] = OCC [] **
6      leafstate m4_AllocPlain [Mary,Cmp,sc] = VAC []
6      leafstate m5_IdleWithTok [Mary,Cmp,sc] = VAC []
6      leafstate m6_AllocOtherWant [Mary,Cmp,sc] = VAC []
6  TRACE =[]
6  TREV [[JohnReqRes,[sc]],0,[],[ClientJohnPco,[sc]]]
6  TREV [[MaryRelRes,[sc]],0,[],[ClientMaryPco,[sc]]]
6  TREV [[JohnReqTok,[Cmp,sc]],0,[],[InterArbJohnPco,[Cmp,sc]]]
6  TREV [[GiveJohnTok,[sc]],0,[],[]]

outworlds=6
number of outworlds=1
SC|:  pe JohnReqRes
SC|:  gc
9 statechart sc
9    set Cmp [sc] = OCC [] **
9    cluster John [Cmp,sc] = OCC [] **
9      leafstate m1_IdleNoTok [John,Cmp,sc] = VAC []
9      leafstate m2_ReqTok [John,Cmp,sc] = VAC []
9      leafstate m3_Waiting [John,Cmp,sc] = OCC [] **
9      leafstate m4_AllocPlain [John,Cmp,sc] = VAC []

© Graham G. Thomason 2003-2004
leafstate m5_IdleWithTok [John,Cmp,sc] = VAC []
leafstate m6_AllocOtherWant [John,Cmp,sc] = VAC []
cluster Mary [Cmp,sc] = OCC [] **
leafstate m1_IdleNoTok [Mary,Cmp,sc] = VAC []
leafstate m2_ReqdTok [Mary,Cmp,sc] = VAC []
leafstate m3_Waiting [Mary,Cmp,sc] = VAC []
leafstate m4_AllocPlain [Mary,Cmp,sc] = VAC []
leafstate m5_IdleWithTok [Mary,Cmp,sc] = VAC []
leafstate m6_AllocOtherWant [Mary,Cmp,sc] = OCC [] **
TRACE =[
TREV [[MaryPass,[Cmp,sc]],0,[],[InterArbMaryPco,[Cmp,sc]]]
TREV [[MaryRelRes,[sc]],0,[],[ClientMaryPco,[sc]]]
TREV [[GiveJohnTok,[sc]],0,[], []]
outworlds=[9]
number of outworlds=1
SC:|: pe MaryRelRes
SC:|: gc
statechart sc
  set Cmp [sc] = OCC [] **
  cluster John [Cmp,sc] = OCC [] **
    leafstate m1_IdleNoTok [John,Cmp,sc] = VAC []
    leafstate m2_ReqdTok [John,Cmp,sc] = VAC []
    leafstate m3_Waiting [John,Cmp,sc] = VAC []
    leafstate m4_AllocPlain [John,Cmp,sc] = OCC [] **
    leafstate m5_IdleWithTok [John,Cmp,sc] = VAC []
    leafstate m6_AllocOtherWant [John,Cmp,sc] = VAC []
  cluster Mary [Cmp,sc] = OCC [] **
    leafstate m1_IdleNoTok [Mary,Cmp,sc] = OCC [] **
    leafstate m2_ReqdTok [Mary,Cmp,sc] = VAC []
    leafstate m3_Waiting [Mary,Cmp,sc] = VAC []
    leafstate m4_AllocPlain [Mary,Cmp,sc] = VAC []
    leafstate m5_IdleWithTok [Mary,Cmp,sc] = VAC []
    leafstate m6_AllocOtherWant [Mary,Cmp,sc] = VAC []
TRACE =[
TREV [[JohnRelRes,[sc]],0,[],[ClientJohnPco,[sc]]]
TREV [[MaryReqTok,[Cmp,sc]],0,[],[InterArbMaryPco,[Cmp,sc]]]
TREV [[MaryReqRes,[sc]],0,[],[ClientMaryPco,[sc]]]
TREV [[GiveJohnTok,[sc]],0,[], []]
outworlds=[11]
number of outworlds=1
SC:|: 
Flattened model and session with users - Repeat of Figure 6, [model t4312]
```
leafstate m2_ReqdTok [Mary, SysM, Cmp, sc] = VAC []
leafstate m3_Waiting [Mary, SysM, Cmp, sc] = VAC []
leafstate m4_AllocPlain [Mary, SysM, Cmp, sc] = VAC []
leafstate m5_IdleWithTok [Mary, SysM, Cmp, sc] = VAC []
leafstate m6_AllocOtherWant [Mary, SysM, Cmp, sc] = VAC []
cluster UserM [SysM, Cmp, sc] = OCC [] **
leafstate NotHaveRes [UserM, SysM, Cmp, sc] = OCC [] **
leafstate Waiting [UserM, SysM, Cmp, sc] = VAC []
leafstate HaveRes [UserM, SysM, Cmp, sc] = VAC []

TRACE = []
TREV [[MaryReqTok, [Cmp, sc]], 0, [], [InterArbMaryPco, [Cmp, sc]]]
TREV [[JohnReqRes, [sc]], 0, [], [ClientJohnPco, [sc]]]
TREV [[alpha, [sc]], 0, [], [UserJPCo, [sc]]]
TREV [[MaryReqRes, [sc]], 0, [], [ClientMaryPco, [sc]]]
TREV [[gamma, [sc]], 0, [], [UserMPCo, [sc]]]
TREV [[GiveJohnTok, [sc]], 0, [], []]
```

```
outworlds=3
number of outworlds=1
SC: |: pe alpha
SC: |: gc
statechart sc
set Cmp [sc] = OCC [] **
set SysJ [Cmp, sc] = OCC [] **
cluster John [SysJ, Cmp, sc] = OCC [] **
leafstate m1_IdleNotok [John, SysJ, Cmp, sc] = VAC []
leafstate m2_ReqdTok [John, SysJ, Cmp, sc] = VAC []
leafstate m3_Waiting [John, SysJ, Cmp, sc] = VAC []
leafstate m4_AllocPlain [John, SysJ, Cmp, sc] = OCC [] **
leafstate m5_IdleWithTok [John, SysJ, Cmp, sc] = VAC []
leafstate m6_AllocOtherWant [John, SysJ, Cmp, sc] = VAC []
cluster UserJ [SysJ, Cmp, sc] = OCC [] **
leafstate NotHaveRes [UserJ, SysJ, Cmp, sc] = VAC []
leafstate Waiting [UserJ, SysJ, Cmp, sc] = VAC []
leafstate HaveRes [UserJ, SysJ, Cmp, sc] = OCC [] **
set SysM [Cmp, sc] = OCC [] **
cluster Mary [SysM, Cmp, sc] = OCC [] **
leafstate m1_IdleNotok [Mary, SysM, Cmp, sc] = OCC [] **
leafstate m2_ReqdTok [Mary, SysM, Cmp, sc] = VAC []
leafstate m3_Waiting [Mary, SysM, Cmp, sc] = VAC []
leafstate m4_AllocPlain [Mary, SysM, Cmp, sc] = VAC []
leafstate m5_IdleWithTok [Mary, SysM, Cmp, sc] = VAC []
leafstate m6_AllocOtherWant [Mary, SysM, Cmp, sc] = VAC []
cluster UserM [SysM, Cmp, sc] = OCC [] **
leafstate NotHaveRes [UserM, SysM, Cmp, sc] = OCC [] **
leafstate Waiting [UserM, SysM, Cmp, sc] = VAC []
leafstate HaveRes [UserM, SysM, Cmp, sc] = VAC []

TRACE = [JAcq]
TREV [[JohnRelRes, [sc]], 0, [], [ClientJohnPco, [sc]]]
TREV [[MaryReqTok, [Cmp, sc]], 0, [], [InterArbMaryPco, [Cmp, sc]]]
TREV [[beta, [sc]], 0, [], [UserJPCo, [sc]]]
TREV [[MaryReqRes, [sc]], 0, [], [ClientMaryPco, [sc]]]
TREV [[gamma, [sc]], 0, [], [UserMPCo, [sc]]]
TREV [[GiveJohnTok, [sc]], 0, [], []]
```

```
outworlds=7
number of outworlds=1
SC: |: pe gamma
SC: |: gc
statechart sc
set Cmp [sc] = OCC [] **
set SysJ [Cmp, sc] = OCC [] **
cluster John [SysJ, Cmp, sc] = OCC [] **
leafstate m1_IdleNotok [John, SysJ, Cmp, sc] = VAC []
leafstate m2_ReqdTok [John, SysJ, Cmp, sc] = VAC []
leafstate m3_Waiting [John, SysJ, Cmp, sc] = VAC []
leafstate m4_AllocPlain [John, SysJ, Cmp, sc] = VAC []
leafstate m5_IdleWithTok [John, SysJ, Cmp, sc] = VAC []
```
leafstate m6_AllocOtherWant [John,SysJ,Cmp,sc] = OCC [] **
leafstate NotHaveRes [UserJ,SysJ,Cmp,sc] = VAC []
leafstate Waiting [UserJ,SysJ,Cmp,sc] = VAC []
leafstate HaveRes [UserJ,SysJ,Cmp,sc] = OCC [] **
set SysM [Cmp,sc] = OCC [] **
cluster Mary [SysM,Cmp,sc] = OCC [] **
leafstate m1_IdleNoTok [Mary,SysM,Cmp,sc] = OCC [] **
leafstate m2_ReqdTok [Mary,SysM,Cmp,sc] = VAC []
leafstate m3_Waiting [Mary,SysM,Cmp,sc] = VAC []
leafstate m4_AllocPlain [Mary,SysM,Cmp,sc] = VAC []
leafstate m5_IdleWithTok [Mary,SysM,Cmp,sc] = VAC []
leafstate m6_AllocOtherWant [Mary,SysM,Cmp,sc] = VAC []
cluster UserM [SysM,Cmp,sc] = OCC [] **
leafstate NotHaveRes [UserM,SysM,Cmp,sc] = VAC []
leafstate Waiting [UserM,SysM,Cmp,sc] = VAC [] **
leafstate HaveRes [UserM,SysM,Cmp,sc] = VAC []
TRACE = [JAcq]
TREV [[JohnRelRes, [sc]], 0, [], [ClientJohnPco, [sc]]]
TREV [[beta, [sc]], 0, [], [UserJPco, [sc]]]
TREV [[JohnPass, [Cmp,sc]], 0, [], [InterArbJohnPco, [Cmp,sc]]]
TREV [[MaryAcqRes, [sc]], 0, [], [ClientMaryPco, [sc]]]
TREV [[GiveJohnTok, [sc]], 0, [], []]
outworlds=11
number of outworlds=1
SC:|: pe beta
SC:|: gc
statechart sc
set Cmp [sc] = OCC [] **
set SysJ [Cmp,sc] = OCC [] **
cluster John [SysJ,Cmp,sc] = OCC [] **
leafstate m1_IdleNoTok [John,SysJ,Cmp,sc] = OCC [] **
leafstate m2_ReqdTok [John,SysJ,Cmp,sc] = VAC []
leafstate m3_Waiting [John,SysJ,Cmp,sc] = VAC []
leafstate m4_AllocPlain [John,SysJ,Cmp,sc] = VAC []
leafstate m5_IdleWithTok [John,SysJ,Cmp,sc] = VAC []
cluster UserJ [SysJ,Cmp,sc] = OCC [] **
leafstate NotHaveRes [UserJ,SysJ,Cmp,sc] = VAC []
leafstate Waiting [UserJ,SysJ,Cmp,sc] = VAC []
leafstate HaveRes [UserJ,SysJ,Cmp,sc] = VAC []
set SysM [Cmp,sc] = OCC [] **
cluster Mary [SysM,Cmp,sc] = OCC [] **
leafstate m1_IdleNoTok [Mary,SysM,Cmp,sc] = OCC [] **
leafstate m2_ReqdTok [Mary,SysM,Cmp,sc] = VAC []
leafstate m3_Waiting [Mary,SysM,Cmp,sc] = VAC []
leafstate m4_AllocPlain [Mary,SysM,Cmp,sc] = VAC [] **
leafstate m5_IdleWithTok [Mary,SysM,Cmp,sc] = VAC []
leafstate m6_AllocOtherWant [Mary,SysM,Cmp,sc] = VAC []
cluster UserM [SysM,Cmp,sc] = OCC [] **
leafstate NotHaveRes [UserM,SysM,Cmp,sc] = VAC []
leafstate Waiting [UserM,SysM,Cmp,sc] = VAC [] **
leafstate HaveRes [UserM,SysM,Cmp,sc] = VAC []
TRACE = [MAcq,JRel,JAcq]
TREV [[JohnReqRes, [sc]], 0, [], [ClientJohnPco, [sc]]]
TREV [[beta, [sc]], 0, [], [UserJPco, [sc]]]
TREV [[MaryRelRes, [sc]], 0, [], [ClientMaryPco, [sc]]]
TREV [[JohnReqTok, [Cmp,sc]], 0, [], [InterArbJohnPco, [Cmp,sc]]]
TREV [[delta, [sc]], 0, [], [UserMPco, [sc]]]
TREV [[GiveJohnTok, [sc]], 0, [], []]
outworlds=17
number of outworlds=1
SC:|: pe delta
SC:|: gc
statechart sc
set Cmp [sc] = OCC [] **
set SysJ [Cmp, sc] = OCC [] **
cluster John [SysJ, Cmp, sc] = OCC [] **
leafstate m1_IdleNoTok [John, SysJ, Cmp, sc] = OCC [] **
leafstate m2_ReqdTok [John, SysJ, Cmp, sc] = VAC []
leafstate m3_Waiting [John, SysJ, Cmp, sc] = VAC []
leafstate m4_AllocPlain [John, SysJ, Cmp, sc] = VAC []
leafstate m5_IdleWithTok [John, SysJ, Cmp, sc] = VAC []
leafstate m6_AllocOtherWant [John, SysJ, Cmp, sc] = VAC []
cluster UserJ [SysJ, Cmp, sc] = OCC [] **
leafstate NotHaveRes [UserJ, SysJ, Cmp, sc] = OCC [] **
leafstate Waiting [UserJ, SysJ, Cmp, sc] = VAC []
leafstate HaveRes [UserJ, SysJ, Cmp, sc] = VAC []
set SysM [Cmp, sc] = OCC [] **
cluster Mary [SysM, Cmp, sc] = OCC [] **
leafstate m1_IdleNoTok [Mary, SysM, Cmp, sc] = VAC []
leafstate m2_ReqdTok [Mary, SysM, Cmp, sc] = VAC []
leafstate m3_Waiting [Mary, SysM, Cmp, sc] = VAC []
leafstate m4_AllocPlain [Mary, SysM, Cmp, sc] = VAC []
leafstate m5_IdleWithTok [Mary, SysM, Cmp, sc] = OCC [] **
leafstate m6_AllocOtherWant [Mary, SysM, Cmp, sc] = VAC []
cluster UserM [SysM, Cmp, sc] = OCC [] **
leafstate NotHaveRes [UserM, SysM, Cmp, sc] = OCC [] **
leafstate Waiting [UserM, SysM, Cmp, sc] = VAC []
leafstate HaveRes [UserM, SysM, Cmp, sc] = VAC []
TRACE = [MRel, MAcq, JRel, JAcq]
TREV [[JohnReqRes, [sc]], 0, [], [ClientJohnPco, [sc]]]
TREV [[alpha, [sc]], 0, [], [UserJPco, [sc]]]
TREV [[JohnReqTok, [Cmp, sc]], 0, [], [InterArbJohnPco, [Cmp, sc]]]
TREV [[MaryReqRes, [sc]], 0, [], [ClientMaryPco, [sc]]]
TREV [[gamma, [sc]], 0, [], [UserMPco, [sc]]]
TREV [[GiveJohnTok, [sc]], 0, [], []]

outworlds=20
number of outworlds=1
SC: ;
3. The STATECRUNCHER distributed arbiter in CCS

The description of the model in Figure 8 is as follows, with the following renaming applied with respect to that figure:

\[
\begin{align*}
\text{MeReqRes} & \rightarrow \text{ReqRes} \\
\text{MeRelRes} & \rightarrow \text{RelRes} \\
\text{MeAcqRes} & \rightarrow \text{AcqRes} \\
\text{MeReqTok} & \rightarrow \text{ReqTok} \\
\text{YouReqTok} & \rightarrow \text{ReqTok} \\
\text{MePass} & \rightarrow \text{Pass} \\
\text{YouPass} & \rightarrow \text{Pass} \\
\text{MeSayOk} & \rightarrow \text{Ok} \\
\text{YouSayOk} & \rightarrow \text{Ok} \\
\text{MeSayNok} & \rightarrow \text{Nok} \\
\text{YouSayNok} & \rightarrow \text{Nok}
\end{align*}
\]

Here is the description:

\[
\begin{align*}
m_1_{\text{IdleNoTok}} & \overset{\text{def}}{=} \text{ReqRes} \cdot \text{ReqTok} \cdot m_2_{\text{ReqdTok}} \\
m_2_{\text{ReqdTok}} & \overset{\text{def}}{=} \text{Nok} \cdot m_3_{\text{Waiting}} + \text{Pass} \cdot \text{AcqRes} \cdot m_4_{\text{AllocPlain}} \\
m_3_{\text{Waiting}} & \overset{\text{def}}{=} \text{Pass} \cdot \text{AcqRes} \cdot m_4_{\text{AllocPlain}} \\
m_4_{\text{AllocPlain}} & \overset{\text{def}}{=} \text{RelRes} \cdot m_5_{\text{IdleWithTok}} + \text{ReqTok} \cdot \text{Nok} \cdot m_6_{\text{AllocOtherWant}} \\
m_5_{\text{IdleWithTok}} & \overset{\text{def}}{=} \text{ReqRes} \cdot \text{AcqRes} \cdot m_4_{\text{AllocPlain}} + \text{ReqTok} \cdot \text{Pass} \cdot m_1_{\text{IdleNoTok}} \\
m_6_{\text{AllocOtherWant}} & \overset{\text{def}}{=} \text{RelRes} \cdot \text{Pass} \cdot m_1_{\text{IdleNoTok}}
\end{align*}
\]

We now consider the model given by Bruns.
Figure 10. CCS Model in [Bruns]
Bruns's model is similar to the model in Figure 8. We make the following remarks

- Bruns has three minor typographical errors in his Figure 2.5:
  - rel₁ from state II should read req₁,
  - state G1 (G for Got?) should read A₁ (Allocated)
  - state OG₁ should read OA₁
- The ok event is initiated when one arbiter acquires the resource. It is pointless, because it checks for a request for the resource when there is no need to do so. An arbiter is told when there is request for a resource by the nok event; any initiative taken in asking about a request is also expensive (because it involves inter-arbiter communication).
- The nok event is a request for the token. This will be triggered by a request for the resource when the arbiter does not have the token.

A comparison

We have named STATECRUNCHER states and events as seems natural in that language. The naming relationships between the STATECRUNCHER and Bruns's CCS model are:

- Bruns's 1 and 2 are replaced by Me and You for events, but we avoid such suffixes for states, which implicitly apply to the Me machine shown. When a second arbiter is introduced, states are by distinguished making names local to an arbiter.
- We distinguish YouPass and MePass as separate events
- We likewise distinguish who says Ok and Nok with YouSayOk, MeSayOk, YouSayNok and MeSayNok.
- Our m2_Reqd_Tok and m3_Waiting states in Figure 8 are combined by Bruns into state SI (resource requested).

We could merge the MePass and YouPass events into one event Pass, since no confusion would arise, but they are better considered as separate events. Similarly the other inter-arbiter events. They have separate origins - in separate computers even. In STATECRUNCHER it is convenient to give them separate names, since they can then be separately declared in their own machine, albeit with composition scope (through the use of scoping operators).

Bruns combines two arbiter agents with CCS calculus; STATECRUNCHER combines server and client by wrapping both in a set. STATECRUNCHER offers scoping operators for PCOs, events, states and variables so that these items can have local or composition scope (see the use of the %% operator in the arbiter-pair models).

Conclusion

This paper has shown how a typical client-server application is modelled in STATECRUNCHER, providing a direct comparison with a well-known example in the literature. Both STATECRUNCHER and CCS are amenable to the problem, but the emphasis is different:
STATECRUNCHER is a state machine engine providing the white box or black box the oracle to tests and does not support calculus manipulations; CCS is a calculus which is used to prove properties of composed systems, and is supported by Concurrency Workbench.
4. Source code of models

4.1 Source code of the single distributed arbiter [model t4300]

```
// Module: d_arb.scs.txt
// Author: Graham Thomason, Philips Digital Systems Laboratories, Redhill
// Date: 07 June, 2003
// Purpose: Statecruncher model: SINGLE DISTRIBUTED ARBITER (cf Glenn Bruns CCS, p.21)
// Project: Improving Component Integration
// Copyright (C) 2003 Philips Electronics N.V.
// Revision History:
//-------------------------2--------3---------4--------5--------6--------7--------8--------

statechart sc(Me)

PCO ClientMePco; // For client-arbiter events
PCO InterArbMePco; // For inter-arbiter events to Me
PCO InterArbYouPco; // For inter-arbiter events to You
// no PCO for internal events

// Acq=Acquire
// Rel=Release
// Req=Request
// Res=Resource
// Tok=Token

// No need for the initial GiveMeTok event, because YouPass is legal

event MeReqRes @ClientMePco; // Client event to Me
event MeRelRes @ClientMePco; // Client event to Me
event MeAcqRes @ClientMePco; // Client event from Me

event MeReqTok @InterArbMePco; // InterArbiter
event MePass @InterArbMePco; // InterArbiter
event MeSayNok @InterArbMePco; // InterArbiter

event YouReqTok @InterArbYouPco; // InterArbiter
event YouPass @InterArbYouPco; // InterArbiter
event YouSayNok @InterArbYouPco; // InterArbiter

event TryTok; // Local to this arbiter
event TryOk; // Local to this arbiter
event AcqTok; // Local to this arbiter
event ResetWant; // Local to this arbiter

set Me(Res,Have,Need,OtherWant)

cluster Res(Idle,Requested,Waiting,Alloc)
  state Idle (MeReqRes->Requested {fire TryTok;})
  state Requested (AcqTok,TryOk->Alloc; YouSayNok->Waiting;)
```
state Waiting {AcqTok->Alloc;}
state Alloc {upon enter {fire MeAcqRes;} MeRelRes->Idle \ 
{if (in($$Me.OtherWant.OtherWantTok)) {fire MePass;}};

cluster Have (NotHaveTok,HaveTok)
state NotHaveTok { 
   TryTok {fire MeReqTok;}; \ 
   YouPass->HaveTok {fire AcqTok;}; 
state HaveTok { 
   TryTok {fire TryOk;}; \ 
   MePass->NotHaveTok {fire ResetWant;}; 
}

cluster Need(NotNeedTok,NeedTok)
state NotNeedTok { 
   YouReqTok {fire MePass;}; \ 
   exit($$Me.Res.Idle)->NeedTok; 
state NeedTok { 
   YouReqTok {fire MeSayNok;} ; 
   enter($$Me. Res.Idle)->NotNeedTok; 
}

cluster OtherWant(OtherNotWantTok,OtherWantTok)
state OtherNotWantTok { 
   YouReqTok [ in($$Me.Need.NeedTok) \ 
               && in($$Me.Have.HaveTok)->OtherWantTok;] 
state OtherWantTok { ResetWant->OtherNotWantTok; }

//--------------------[end of module]---------------------------

© Graham G. Thomason 2003-2004
4.2 Source code of the two distributed arbiters (John and Mary)
[model t4301]
TryTok {fire TryOk;};
MePass->NotHaveTok {fire ResetWant;}; 

cluster Need(NotNeedTok,NeedTok)
  state NotNeedTok {
    YouReqTok {fire MePass;};
    exit($$Me.Res.Idle)->NeedTok;
  }
  state NeedTok {
    YouReqTok {fire MeSayNok;};
    enter($$Me.Res.Idle)->NotNeedTok;
  }

cluster OtherWant(OtherNotWantTok,OtherWantTok)
  state OtherNotWantTok {
    YouReqTok |
      in($$Me.Need.NeedTok)
      && in($$Me.Have.HaveTok) -> OtherWantTok;
  }
  state OtherWantTok {ResetWant->OtherNotWantTok;}

//-------------------------------[end of module]-------------------------------------------
4.3 Source code of the distributed arbiter with clients [model t4302]

---

// Module: d_arb_client.scs.txt
// Author: Graham Thomason, Philips Digital Systems Laboratories, Redhill
// Date: 07 June, 2003
// Purpose: Statecruncher model: DISTRIBUTED ARBITER WITH CLIENTS
// (as in Glenn Bruns CCS, p.21)
// Project: Improving Component Integration
// Copyright (C) 2003 Philips Electronics N.V.
// Revision History:

-------------1---------2---------3---------4---------5---------6---------7---------8-----

statechart sc(Cmp)

PCO UserJPco;                  // For user events
PCO UserMPco;                  // For user events

PCO ClientJohnPco;             // For Client-to-arbiter events
PCO ClientMaryPco;             // For Client-to-arbiter events

// Cmp=Composition
// Acq=Acquire
// Rel=Release
// Req=Request
// Res=Resource
// Tok=Token

event alpha@UserJPco;          // User event in UserJ
event beta@UserJPco;           // User event in UserJ

event gamma@UserMPco;          // User event in UserM
event delta@UserMPco;          // User event in UserM

event JohnReqRes @ClientJohnPco;  // Client event to John(arbiter)
event JohnRelRes @ClientJohnPco;  // Client event to John(arbiter)
event JohnAcqRes @ClientJohnPco;  // Client event from John(arbiter)
event MaryReqRes @ClientMaryPco;  // Client event to Mary(arbiter)
event MaryRelRes @ClientMaryPco;  // Client event to Mary(arbiter)
event MaryAcqRes @ClientMaryPco;  // Client event from Mary(arbiter)

// INITIAL MANUAL EVENT TO BE GIVEN
// It preserves symmetry between John/Mary
// (otherwise reverse state order to get opposing default states)
// It is GLOBAL for ease of entry
// (we could have used event Composition%%JohnAcqTok)
event GiveJohnTok;

// ReqTok and AcqTok events are locally defined in composition scope

set Cmp(SysJ,SysM) {GiveJohnTok}->{Cmp.SysJ.John.Have.HaveTok;)

set SysJ(John,UserJ)

set(Res,Have,Need,OtherWant)

set(Res,Have,Need,OtherWant)

PCO Cmp%%InterArbJohnPco;       // For inter-arbiter events to John
event Cmp%%JohnReqTok@InterArbJohnPco;      // InterArbiter, Composition scope
event Cmp%%JohnPass@InterArbJohnPco;        // InterArbiter, Composition scope

© Graham G. Thomason 2003-2004
event Cmp%%JohnSayNok@InterArbMaryPco; // InterArbiter, Composition scope

event TryTok; // Local to this arbiter
event TryOk; // Local to this arbiter
event AcqTok; // Local to this arbiter
event ResetWant; // Local to this arbiter

cluster Res(Idle, Requested, Waiting, Alloc)
  state Idle { JohnReqRes->Requested (fire TryTok); }
  state Requested { AcqTok,TryOk->Alloc; JohnSayNok->Waiting; }
  state Waiting { AcqTok->Alloc; }
  state Alloc { upon enter {fire JohnAcqRes; } JohnRelRes->Idle \
    (if (in($$John.OtherWant.OtherWantTok)) (fire JohnPass)); }

cluster Have (NotHaveTok, HaveTok)
  state NotHaveTok {
    TryTok {fire JohnReqTok;};
    MaryPass->HaveTok (fire AcqTok;);
  }
  state HaveTok {
    TryTok {fire TryOk;};
    JohnPass->NotHaveTok {fire ResetWant;};
  }

cluster Need (NotNeedTok, NeedTok)
  state NotNeedTok {
    MaryReqTok {fire JohnPass;};
    exit($$John.Res.Idle)->NeedTok;
  }
  state NeedTok {
    MaryReqTok {fire JohnSayNok;}
    enter($$John.Res.Idle)->NotNeedTok;
  }

cluster OtherWant (OtherNotWantTok, OtherWantTok)
  state OtherNotWantTok {
    MaryReqTok [in($$John.Need.NeedTok) \&\& in($$John.Have.HaveTok)]->OtherWantTok;
  }
  state OtherWantTok {ResetWant->OtherNotWantTok;}

cluster UserJ (NotHaveRes, Waiting, HaveRes)
  state NotHaveRes { alpha->Waiting [fire JohnReqRes;]; }
  state Waiting { JohnAcqRes->HaveRes; }
  state HaveRes { upon enter {trace('JAq');} \ 
    upon exit {trace('JRel');} \ 
    beta->NotHaveRes {fire JohnRelRes;};}

// ----- as above except alpha,beta,gamma,delta and...
// ... Mary for John and vice versa everywhere -----

set SysM(Mary, UserM)

set Mary(Res, Have, Need, OtherWant)

PCO Cmp%%InterArbMaryPco; // For inter-arbiter events to Mary
event Cmp%%MaryReqTok @InterArbMaryPco; // InterArbiter, Composition scope
event Cmp%%MaryPass @InterArbMaryPco; // InterArbiter, Composition scope
event Cmp%%MarySayNok @InterArbJohnPco; // InterArbiter, Composition scope

event TryTok; // Local to this arbiter
event TryOk; // Local to this arbiter
event AcqTok; // Local to this arbiter
event ResetWant; // Local to this arbiter

cluster Res(Idle, Requested, Waiting, Alloc)
  state Idle { MaryReqRes->Requested (fire TryTok); }
  state Requested { AcqTok, TryOk->Alloc; JohnSayNok->Waiting; }
  state Waiting { AcqTok->Alloc; }
  state Alloc { upon enter {fire MaryAcqRes; } MaryRelRes->Idle \
    (if (in($$Mary.OtherWant.OtherWantTok)) (fire MaryPass)); }

cluster Have (NotHaveTok, HaveTok)
state NotHaveTok {
    TryTok {fire MaryReqTok;};
    JohnPass->HaveTok {fire AcqTok;};
}
state HaveTok {
    TryTok {fire TryOk;};
    MaryPass->NotHaveTok {fire ResetWant;};
}

cluster Need(NotNeedTok, NeedTok)
state NotNeedTok {
    JohnReqTok {fire MaryPass;};
    exit($$Mary.Res.Idle)->NeedTok;
}
state NeedTok {
    JohnReqTok {fire MarySayNok;};
    enter($$Mary.Res.Idle)->NotNeedTok;
}

cluster OtherWant(OtherNotWantTok, OtherWantTok)
state OtherNotWantTok {
    JohnReqTok [ in($$Mary.Need.NeedTok) && in($$Mary.Have.HaveTok)]->OtherWantTok;
}
state OtherWantTok {ResetWant->OtherNotWantTok;}

cluster UserM(NotHaveRes, Waiting, HaveRes)
state NotHaveRes {gamma->Waiting {fire MaryReqRes;};}
state Waiting {MaryAcqRes->HaveRes;}
state HaveRes {upon enter {trace("MAcq");} \ 
    upon exit {trace("MRel");} \ 
    delta->NotHaveRes {fire MaryRelRes;};}

//------------------------[end of module]-----------------------------
4.4 Source code of single flattened distributed arbiter [model t4310]

---

// Module:     d_arbf.scs.txt
// Author:     Graham Thomason, Philips Digital Systems Laboratories, Redhill
// Date:       25 June, 2003
// Purpose:    Statecruncher model: SINGLE DISTRIBUTED ARBITER FLAT MODEL
//             (cf Glenn Bruns CCS, p.21)
// Project:    Improving Component Integration
// Copyright (C) 2003 Philips Electronics N.V.

statechart sc(Me)

PCO ClientMePco; // For client-arbiter events
PCO InterArbMePco; // For inter-arbiter events to Me
PCO InterArbYouPco; // For inter-arbiter events to You
// no PCO for internal events

// Acq=Acquire (not used in flattened model)
// Rel=Release
// Req=Request
// Res=Resource
// Tok=Token

event MeReqRes ©ClientMePco; // Client event to Me
event MeRelRes ©ClientMePco; // Client event to Me
event MeAcqRes ©ClientMePco; // Client event from Me

event MeReqTok ©InterArbMePco; // InterArbiter
event MePass ©InterArbMePco; // InterArbiter
event MeSayNok ©InterArbMePco; // InterArbiter

event YouReqTok ©InterArbYouPco; // InterArbiter
event YouPass ©InterArbYouPco; // InterArbiter
event YouSayNok ©InterArbYouPco; // InterArbiter

// No local events in the flattened model

cluster Me( m1_IdleNoTok, m2_ReqdTok, \
         m3_Waiting, m4_AllocPlain, \
         m5_IdleWithTok, m6_AllocOtherWant )

state m1_IdleNoTok { MeReqRes->m2_ReqdTok (fire MeReqTok};

state m2_ReqdTok { YouSayNok->m3_Waiting; \n                YouPass->m4_AllocPlain; }

state m3_Waiting { YouPass->m4_AllocPlain;}

state m4_AllocPlain { upon enter [fire MeAcqRes]; \n                      MeRelRes->m5_IdleWithTok; \n                      YouReqTok->m5_AllocOtherWant {fire MeSayNok};}

state m5_IdleWithTok { YouReqTok->m1_IdleNoTok [fire MePass]; \n                      MeRelRes->m4_AllocPlain; }

state m6_AllocOtherWant { MeRelRes->m1_IdleNoTok (fire MePass); }

//------------------------{end of module}----------------------------------------
4.5 Source code of flattened distributed arbiters [model t4311]

---

// Module: d_arbf_pair.scs.txt
// Author: Graham Thomason, Philips Digital Systems Laboratories, Redhill
// Date: 25 June, 2003
// Purpose: Statecruncher model: TWO DISTRIBUTED ARBITERS FLATTENED
// (cf Glenn Bruns CCS, p.21)

// Project: Improving Component Integration
// Copyright (C) 2003 Philips Electronics N.V.

// Revision History:
//
//-------1--------2--------3--------4--------5--------6--------7--------8------

statechart sc(Cmp)

PCO ClientJohnPco; // For user-to-arbiter events
PCO ClientMaryPco; // For user-to-arbiter events

// Acq=Acquire (not used in flattened model)
// Rel=Release
// Req=Request
// Res=Resource
// Tok=Token

event JohnReqRes ©ClientJohnPco // Client event to John(arbiter)
event JohnAcqRes ©ClientJohnPco // Client event from John(arbiter)
event MaryReqRes ©ClientMaryPco // Client event to Mary(arbiter)
event MaryRelRes ©ClientMaryPco // Client event from Mary(arbiter)
event MaryAcqRes ©ClientMaryPco // Client event from Mary(arbiter)

// No local events in the flattened model

// INITIAL MANUAL EVENT TO BE GIVEN
event GiveJohnTok;

set Cmp(John,Mary) {
	GiveJohnTok[in(Cmp.John.m1_IdleNoTok) && in(Cmp.Mary.m1_IdleNoTok)]
	->Cmp.John.m5_IdleWithTok;
	// The above condition is evaluated at execution time and does not
	// prevent the event appearing as a potential transitionable event
}

cluster John( m1_IdleNoTok, m2_ReqdTok,
	m3_Waiting, m4_AllocPlain,
	m5_IdleWithTok, m6_AllocOtherWant )

PCO Cmp%%InterArbJohnPco; // For inter-arbiter events to John

event Cmp%%JohnReqTok @InterArbJohnPco; // InterArbiter, Composition scope
event Cmp%%JohnPass @InterArbJohnPco; // InterArbiter, Composition scope

cluster JohnSayNok @InterArbMaryPco; // InterArbiter, Composition scope

state m1_IdleNoTok { JohnReqRes->m2_ReqdTok {fire JohnReqTok;}; }
state m2_ReqdTok { MarySayNok->m3_Waiting; 
	MaryPass->m4_AllocPlain; }
state m3_Waiting { MaryPass->m4_AllocPlain; }
state m4_AllocPlain { upon enter {fire JohnAcqRes;}
	JohnRelRes->m5_IdleWithTok;
	MaryReqTok->m6_AllocOtherWant {fire JohnSayNok;}; }
state m5_IdleWithTok { MaryReqTok->m1_IdleNoTok {fire JohnPass;}; 
	JohnReqRes->m4_AllocPlain; }

© Graham G. Thomason 2003-2004
state m6_AllocOtherWant {  JohnRelRes->m1_IdleNoTok {fire JohnPass;} ;  }

//----- as above, but Mary for John and vice versa everywhere -----  

cluster Mary(  m1_IdleNoTok, m2_ReqdTok,  
m3_Waiting, m4_AllocPlain,  
m5_IdleWithTok, m6_AllocOtherWant )    

PCO  Cmp%%InterArbMaryPco;  // For inter-arbiter events to Mary  
event Cmp%%MaryReqTok @InterArbMaryPco;  // InterArbiter, Composition scope  
event Cmp%%MaryPass @InterArbMaryPco;  // InterArbiter, Composition scope  
event Cmp%%MarySayNok @InterArbJohnPco;  // InterArbiter, Composition scope

state m1_IdleNoTok {  MaryReqRes->m2_ReqdTok {fire MaryReqTok;} ;  }  
state m2_ReqdTok {  JohnSayNok->m3_Waiting;  
JohnPass->m4_AllocPlain;  }  
state m3_Waiting {  JohnPass->m4_AllocPlain;  }  
state m4_AllocPlain {  upon enter {fire MaryAcqRes;}  
    MaryRelRes->m5_IdleWithTok;  
    JohnReqTok->m6_AllocOtherWant {fire MarySayNok;} ;  }  
state m5_IdleWithTok {  JohnReqTok->m1_IdleNoTok {fire MaryPass;} ;  
    MaryReqRes->m4_AllocPlain;  }  
state m6_AllocOtherWant {  MaryRelRes->m1_IdleNoTok {fire MaryPass;} ;  }

="/end of module"
4.6 Source code of flattened distributed arbiter with clients [model t4312]

// Module: d_arbf_client.scs.txt
// Author: Graham Thomason, Philips Digital Systems Laboratories, Redhill
// Date: 25 June, 2003
// Purpose: Statetracer model: DISTRIBUTED (FLAT) ARBITER WITH CLIENTS
// (cf Glenn Bruns CCS, p.21)
// Project: Improving Component Integration
// Copyright (C) 2003 Philips Electronics N.V.
// Revision History:
// 1-2-3-4-5-6-7-8

statechart sc(Cmp)

PCO UserJPco; // For user events
PCO UserMPco; // For user events
PCO ClientJohnPco; // For user-to-arbiter events
PCO ClientMaryPco; // For user-to-arbiter events

// Acq=Acquire (not used in flattened model)
// Rel=Release
// Req=Request
// Res=Resource
// Tok=Token

event alpha@UserJPco; // User event in UserJ
event beta@UserJPco; // User event in UserJ

event gamma@UserMPco; // User event in UserM
event delta@UserMPco; // User event in UserM

event JohnReqRes@ClientJohnPco; // Client event to John(arbiter)
event JohnRelRes@ClientJohnPco; // Client event to John(arbiter)
event JohnAcqRes@ClientJohnPco; // Client event from John(arbiter)
event MaryReqRes@ClientMaryPco; // Client event to Mary(arbiter)
event MaryRelRes@ClientMaryPco; // Client event to Mary(arbiter)
event MaryAcqRes@ClientMaryPco; // Client event from Mary(arbiter)

// No local events in the flattened model

// INITIAL MANUAL EVENT TO BE GIVEN
event GiveJohnTok;

set Cmp(SysJ,SysM) { GiveJohnTok \[in(Cmp.SysJ.John.ml_IdleNoTok) && in(Cmp.SysM.Mary.ml_IdleNoTok)\] \->Cmp.SysJ.John.m5_IdleWithTok; }

set SysJ(John,UserJ)

cluster John( ml_IdleNoTok, m2_ReqdTok, \m3_Waiting, m4_AllocPlain, \m5_IdleWithTok, m6_AllocOtherWant )
PCO Cmp%%InterArbJohnPco; // For inter-arbiter events to John
event Cmp%%JohnReqTok ©InterArbJohnPco; // InterArbiter, Composition scope
event Cmp%%JohnPass ©InterArbJohnPco; // InterArbiter, Composition scope
event Cmp%%JohnSayNok ©InterArbMaryPco; // InterArbiter, Composition scope

state m1_IdleNoTok  { JohnReqRes->m2_ReqdTok {fire JohnReqTok;} ; }
state m2_ReqdTok   { MarySayNok->m3_Waiting; \  
                      MaryPass->m4_AllocPlain;  )
state m3_Waiting   { upon enter {fire JohnAcqRes;} \  
                      JohnRelRes->m5_IdleWithTok; \  
                      MaryReqTok->m6_AllocOtherWant {fire JohnSayNok;}; )
state m4_AllocPlain { upon enter {fire JohnAcqRes;} \  
                        JohnRelRes->m5_IdleWithTok; \  
                        MaryReqTok->m6_AllocOtherWant {fire JohnSayNok;}; )
state m5_IdleWithTok  { JohnReqTok->m1_IdleNoTok {fire JohnPass;}  \  
                         JohnReqRes->m4_AllocPlain;  )
state m6_AllocOtherWant { JohnRelRes->m1_IdleNoTok {fire JohnPass;}  )

cluster UserJ(NotHaveRes,Waiting,HaveRes)
state NotHaveRes  { alpha->Waiting {fire JohnReqRes;;});
state Waiting    { MaryAcqRes->HaveRes;}
state HaveRes    {upon enter {trace("JAcq");} \  
                      upon exit {trace("JRel");} \  
                      beta->NotHaveRes {fire JohnRelRes;;});

//----- as above, but Mary for John and vice versa everywhere ----- 
set SysM(Mary,UserM)

cluster Mary( m1_IdleNoTok, m2_ReqdTok, \  m3_Waiting, m4_AllocPlain, \  m5_IdleWithTok, m6_AllocOtherWant )
PCO Cmp%%InterArbMaryPco; // For inter-arbiter events to Mary

state m1_IdleNoTok  { MaryReqRes->m2_ReqdTok {fire MaryReqTok;} ; }
state m2_ReqdTok   { JohnSayNok->m3_Waiting; \  
                      JohnPass->m4_AllocPlain;  )
state m3_Waiting   { JohnPass->m4_AllocPlain;  )
state m4_AllocPlain { upon enter {fire MaryAcqRes;} \  
                        MaryRelRes->m5_IdleWithTok; \  
                        JohnReqTok->m6_AllocOtherWant {fire MarySayNok;}; )
state m5_IdleWithTok  { JohnReqTok->m1_IdleNoTok {fire MaryPass;}  \  
                         JohnReqRes->m4_AllocPlain;  )
state m6_AllocOtherWant { MaryRelRes->m1_IdleNoTok {fire MaryPass;}  )

cluster UserM(NotHaveRes,Waiting,HaveRes)
state NotHaveRes  { gamma->Waiting {fire MaryReqRes;;});
state Waiting    { MaryAcqRes->HaveRes;}
state HaveRes    {upon enter {trace("MAcq");} \  
                      upon exit {trace("MRel");} \  
                      delta->NotHaveRes {fire MaryRelRes;;});

//------------------------------[end of module]-----------------------------

© Graham G. Thomason 2003-2004
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 3 [StCrGP4] GP4 - The Generic Prolog Parsing and Prototyping Package (underlies the STATECRUNCHER compiler)
Related report 4 [StCrParsing] STATECRUNCHER Parsing
Related report 5 [StCrTest] STATECRUNCHER Test Models
Related report 6 [StCrFunMod] State-based Modelling of Functions and Pump Engines
References

[Bruns]  Glenn Bruns
Distributed Systems Analysis with CCS

[CWB]  The Edinburgh Concurrency Workbench
http://www.dcs.ed.ac.uk/home/cwb/

[Harel 87]  D. Harel et al.
On the Formal Semantics of Statecharts
Logic in Computer Science, 2nd Annual Conference, 1987, pp.54-64

[Milner]  Robin Milner
Communication and Concurrency

UML specifications are available from this website.
Appendix 5

The Game of Nim in Z and STATECRUNCHER

An Appendix to the Thesis
The Design and Construction of a State Machine System that Handles Nondeterminism

Graham G. Thomason
The Game of Nim in Z and STATECRUNCHER

Graham G. Thomason

Appendix to the Thesis “The Design and Construction of a State Machine System that Handles Nondeterminism”

UniS

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

In this paper we show how a system taken from the Z literature can be modelled in STATECRUNCHER. An understanding of STATECRUNCHER is assumed, but for the purposes of this paper, most of STATECRUNCHER's functionality will not seem strange to anyone familiar with UML dynamic modelling [UML], since that is the basis of the language.

We take a fairly easy example that nevertheless illustrates the essence of Z: the Game of Nim as described by McMorran and Powell [McMorran p.224]. This example covers a relation (a total function) and \(\Delta\) and \(\Xi\) operations on schemas.

We are not concerned with a strategy for winning, though a simple one exists. We are concerned with specifying how the game is played and when a player has won.

STATECRUNCHER has been built for the purposes of providing an oracle to state-based tests. It has sufficient expressive power to capture the game of Nim in its entirety in a fairly intuitive way.

\[1\] When there is one pile left, the only winning position (i.e. after the winning player's turn) is when there is just one stick in the pile. If there are two or three piles left, winning positions are determined as follows. Express the number of sticks in each pile in binary. Add these binary numbers in column-by-column modulo-2 arithmetic (so there is no carry from one column to another). If the result is zero, the position is a winning one. For the starting position (5, 6, and 7 sticks), the modulo-2-sum is \(101+110+111 = 100\). So by taking 4 sticks from any pile, a winning position is obtained.
# Table of Contents

1. Nim in Z.......................................................................................................................... 1
2. Nim in STATECRUNCHER.............................................................................................. 4
3. Source listing of the STATECRUNCHER model........................................................... 11
4. References ...................................................................................................................... 13
1. Nim in Z

The description of the exercise given in [McMorran, p.118] is:

The game of Nim is played by two people. The game starts with three piles containing five, six and seven sticks respectively. Each player plays alternately. On each turn, the player removes some sticks from one pile. The loser is the player who removes the last stick (the other player is the winner).

![Nim - the starting position](image)

We are asked to write a formal specification of the game state and a Play schema. We must distinguish between

- Game Ended
- Game Continues
- Illegal Play

A Nim specification in Z along the lines of [McMorran] follows, but we add the notion of players necessarily taking turns. The players are John and Mary. The player is not supplied as a parameter, but any move is attributed to the player whose turn it is when making the move. The additions in the specification below with respect to [McMorran] are in marked by a double line in the margin. The maker of the Z font used is indicated in reference [Z font].

Reminder of some less common terminology used:

- The range of a relation $R$ is denoted by $\text{ran } R$.
- The range restriction relation $R > S$ is the subset of $R$ where the range is restricted to $S$.

We will call the piles $A$, $B$ and $C$.

\[ \text{pileid} ::= A \mid B \mid C \]
We will call the players John and Mary.

player::=John | Mary

The Game state is a mapping from pileid to the number of sticks in that pile, and a mapping from the player to a truth value of whether it is their turn or not. It is only one player's turn at any time.

A player may make a valid or invalid move (requesting too many sticks). A valid move will complete the game or leave sticks still available.

code::=ok | error | fin

The input values for Play are a pile identity, \( p \)?, and the number of sticks the player wishes to take, \( \text{take} \)?. A return code, \( rc \), shows the result.

The play is permitted if there are enough sticks.

The set of piles is updated by decrementing the count for \( p \)? by \( \text{take} \)\?. The turn mapping is updated by negating the truth value associated with each player. For the new turn mapping, we could have negated each player's turn explicitly.
\[ \text{turn} = \text{turn} \oplus \{\text{John} \rightarrow \neg(\text{turn John})\} \oplus \{\text{Mary} \rightarrow \neg(\text{turn Mary})\} \]

The play is prohibited if there are too few sticks:

\[
\begin{align*}
\text{PlayErr} & \quad \Xi \text{Game} \\
\text{Parameters} & \\
\text{pile} p \? < \text{take} \? \\
rc ! = \text{error}
\end{align*}
\]

The game is complete when all piles are empty:

\[ \text{Ended} \equiv [\text{Game} | \text{ran pile} = \{0\}] \]

Any intermediate state, (that is, where there are sticks on the table) we will call \textit{Open}:

\[ \text{Open} \equiv [\text{Game} | \text{ran pile} \neq \{0\}] \]

A play that leaves the game in an \textit{Open} state can be described thus:

\[ \text{PlayMore} \equiv [\text{PlayOK} | \text{Open} \land rc ! = \text{ok}] \]

A play that ends the game can be described thus

\[ \text{PlayLast} \equiv [\text{PlayOK} | \text{Ended} \land rc ! = \text{fin}] \]

We can now describe a play

\[ \text{Play} \equiv \text{PlayMore} \lor \text{PlayLast} \lor \text{PlayErr} \]
2. Nim in STATECRUNCHER

Figure 2 shows how Nim can be modelled in STATECRUNCHER.

Following the figure, a description of the model is given, then a session running the model is reproduced.

The following appendix is recommended reading prior to studying the output produced by the STATECRUNCHER models:

- *A Quick Reference of STATECRUNCHER's Output Format* [StCrOutput]

The source code of the model given at the end of this paper. It corresponds to the figure in almost every detail.
statechart sc

II (variables)
i1=5 nr of sticks in pile 1
i2=6 nr of sticks in pile 2
i3=7 nr of sticks in pile 3

II (variables)
p=0 pile from which sticks are taken
take=0 number of sticks taken

Play

GeneralTake /
if (p==1) {i1-=take; trace("OK");}
if (p==2) {i2-=take; trace("OK");}
if (p==3) {i3-=take; trace("OK");}
if (i1==0 && i2==0 && i3==0 && in(Play.John)) {trace("John Wins");}
if (i1==0 && i2==0 && i3==0 && in(Play.Mary)) {trace("Mary Wins");}

PileError / trace("PileError");
nSticksError / trace("nSticksError");

JohnTake(::p,::take)
[ p<1 || p>3 ] /
fire PileError

JohnTake(::p,::take)
[ (p==1 && i1<<take)
|| (p==2 && i2<<take)
|| (p==3 && i3<<take) ] /
fire nSticksError;

MaryTake(::p,::take)
[ (p==1 && i1>=take)
|| (p==2 && i2>=take)
|| (p==3 && i3>=take) ] /
fire GeneralTake

MaryTake(::p,::take)
[ (p==1 && i1>=take)
|| (p==2 && i2>=take)
|| (p==3 && i3>=take) ] /
fire GeneralTake

MaryTake(::p,::take)
[ (p==1 && i1<take)
|| (p==2 && i2<take)
|| (p==3 && i3<take) ] /
fire nSticksError;

JohnTake(::p,::take)
[ (p==1 && i1<take)
|| (p==2 && i2<take)
|| (p==3 && i3<take) ] /
fire GeneralTake

MaryTake(::p,::take)
[ (p==1 && i1<take)
|| (p==2 && i2<take)
|| (p==3 && i3<take) ] /
fire nSticksError;

MaryTake(::p,::take)
[ (p==1 && i1<take)
|| (p==2 && i2<take)
|| (p==3 && i3<take) ] /
fire nSticksError;

JohnTake(::p,::take)
[ (p==1 && i1<take)
|| (p==2 && i2<take)
|| (p==3 && i3<take) ] /
fire GeneralTake

MaryTake(::p,::take)
[ (p==1 && i1<take)
|| (p==2 && i2<take)
|| (p==3 && i3<take) ] /
fire nSticksError;

MaryTake(::p,::take)
[ (p==1 && i1<take)
|| (p==2 && i2<take)
|| (p==3 && i3<take) ] /
fire nSticksError;

Figure 2. Nim [model t4320]

© Graham G. Thomason 2003-2004
A description of the STATECRUNCHER model, with the relationship to the Z specification

The piles are held in variables $i_1$, $i_2$, $i_3$ respectively (cf. the Z pileid::= $A$ | $B$ | $C$).

The player whose turn it is, is held by a leafstate John or Mary being occupied (with the other one being vacant). The cluster (OR state) construction ensures that only one state is occupied. In Z this was $\#(pl : player \mid turn \ player) = 1$.

The $\Delta$ operations in $PlayOK$ correspond to transitions between states John and Mary. These always involve a legal number of sticks being taken. The events triggering the transitions are JohnTake and MaryTake, with parameters that are stored in $p$ and $take$, corresponding to $p$? and $take$? in the Z specification.

The $\Xi$ operations in $PlayErr$ correspond to self transitions on states John and Mary. These are error moves which result in a STATECRUNCHER trace to this effect, with no transitions between states. The error code of the Z specification
code::=ok | error | fin
is reflected in the most recent STATECRUNCHER TRACE which can be:
OK, nSticksError, PileError, JohnWins or MaryWins.

The self-transitions on the internal events GeneralTake, PileError and nSticksError are the equivalent of a subroutine of imperative languages such as C. They execute the mechanics of a move that could come from two places, with either John or Mary initiating them.

The game ends when the STATECRUNCHER TRACE indicates this by giving the winner - no more events should be given - the model is only valid up to this point. (Any more events are traced as being in error if attempted. We could have disabled such events by introducing a new state $\text{Ended}$ and transitioning to it as an additional action to tracing the winner). If the trace does not indicate a winner, the game continues. We can also see from the pile values $i_1$, $i_2$ and $i_3$ whether the game has ended. Comparing with the Z specification, we have

In the Z specification, the game is complete when all piles are empty:
$$\text{Ended} \equiv \exists \text{Game} \mid \text{ran pile} = \{0\}$$

In the STATECRUNCHER model, the game is complete when
$$i_1 = 0 \land i_2 = 0 \land i_3 = 0$$
or when the most recent TRACE is JohnWins or MaryWins.
In the Z specification, any intermediate state is called Open:

\[ \text{Open} \triangleq [\text{Game} \mid \text{ran pile} \neq \{0\}] \]

In the STATECRUNCHER model, an Open state is seen by

\[ i_1! = 0 \mid i_2! = 0 \mid i_3! = 0 \]

or when the most recent TRACE is not JohnWins and not MaryWins.

In the Z specification, a play that leaves the game in an Open state is:

\[ \text{PlayMore} \triangleq [\text{PlayOK} \mid \text{Open'} \land \text{rc} ! = \text{ok}] \]

In the STATECRUNCHER model, this corresponds to, for example

event MaryTake or JohnTake (according to which is transitionable)

after which

\[ i_1! = 0 \mid i_2! = 0 \mid i_3! = 0 \]

and (last TRACE) = OK (but even disallowed moves leave the game open)

In the Z specification, a play that ends the game can be described thus

\[ \text{PlayLast} \triangleq [\text{PlayOK} \mid \text{Ended} \land \text{rc} ! = \text{fin}] \]

In the STATECRUNCHER model, this corresponds to, for example

event MaryTake or JohnTake (according to which is transitionable)

after which

\[ i_1==0 \land i_2==0 \land i_3==0 \]

and (last TRACE) = MaryWins or JohnWins

In the Z specification, we describe a play as

\[ \text{PlayMore} \triangleq \text{PlayMore} \lor \text{PlayLast} \lor \text{PlayErr} \]

In the STATECRUNCHER model, this is just

event MaryTake or JohnTake (according to which is transitionable)

provided the game has not ended, which is as far as the model is valid.
Session with Nim [model t4320]

Only essential explanations of STATECRUNCHER's output are given here. For more detail, refer to [StCrOutput].

Transitionable events are given by TREV lines. The only events that can be supplied from an external perspective are those at PCO [external, sc]. This will give just one event from the set {JohnTake, MaryTake} at any stage of playing the game. An event is supplied for processing in this model by a command
\[ \text{pe } event \ p= [\text{param1}, \text{param2}] \]
where event is JohnTake or MaryTake and param1 is the pile (the p of Z) and param2 is the number of sticks to take (the take of Z).

The player whose turn it is, is evident from the transitionable event offered, but it can also be seen from the occupied leafstate, in the leafstate John and leafstate Mary lines (OCC=occupied, VAC=vacant).

The number of sticks per pile is seen in the VAR INTEGER i1/i2/i3 lines.

The move status is indicated by the TRACE lines. The TRACE is read from right to left.

Further notes on the output, but which are not essential to understanding the game play are:
- terms in many lines such as [sc] and [Play, sc] give the scope (i.e. position in the statechart hierarchy) of an item. Events can be supplied without scope - in that case statechart scope is assumed.
- the TREV lines contain event names (with scope), then the number of parameters, then the ranges of parameters then the PCO of the event.

The session shows moves being made, including disallowed moves involving a disallowed pile of a disallowed number of sticks. The events supplied, and error codes just produced in the TRACE, are shown in bold font.

```
SC:|: run t4320
...
SC:|: gc
2  statechart sc
2   cluster Play [sc] = OCC [] **
2   leafstate John [Play,sc] = OCC [] **
2   leafstate Mary [Play,sc] = VAC []
2  VAR INTEGER i1 [sc] =5
2  VAR INTEGER i2 [sc] =6
2  VAR INTEGER i3 [sc] =7
2  VAR INTEGER p [sc] =0
2  VAR INTEGER take [sc] =0
2  TRACE =[]
2  TREV [[JohnTake, [sc]], 2, [[r,0,3],[r,0,7]], [external, [sc]]]
2  TREV [[PileError, [sc]], 0, [], [internal, [sc]]]
2  TREV [[nSticksError, [sc]], 0, [], [internal, [sc]]]
2  TREV [[GeneralTake, [sc]], 0, [], [internal, [sc]]]
outworlds=2]
number of outworlds=1
```
SC: | pe JohnTake p=[2,4]
SC: |: gc
6 statechart sc
6   cluster Play [sc] = OCC [] **
6   leafstate John [Play,sc] = VAC []
6   leafstate Mary [Play,sc] = OCC [] **
6   VAR INTEGER i1 [sc] =5
6   VAR INTEGER i2 [sc] =2
6   VAR INTEGER i3 [sc] =7
6   VAR INTEGER p [sc] =2
6   VAR INTEGER take [sc] =4
6   TRACE = [OK]
6   TREV [[MaryTake, [sc]],2,[[r,0,3],[r,0,7]], [external, [sc]]]
6   TREV [[PileError, [sc]], 0, [], [internal, [sc]]]
6   TREV [[nSticksError, [sc]],0, [], [internal, [sc]]]
6   TREV [[GeneralTake, [sc]], 0, [], [internal, [sc]]]

outworlds=6
number of outworlds=1
SC: | pe MaryTake p=[2,3]
SC: |: gc
9 statechart sc
9   cluster Play [sc] = OCC [] **
9   leafstate John [Play,sc] = VAC []
9   leafstate Mary [Play,sc] = OCC [] **
9   VAR INTEGER i1 [sc] =5
9   VAR INTEGER i2 [sc] =2
9   VAR INTEGER i3 [sc] =7
9   VAR INTEGER p [sc] =2
9   VAR INTEGER take [sc] =3
9   TRACE = [nSticksError,OK]
9   TREV [[MaryTake, [sc]],2,[[r,0,3],[r,0,7]], [external, [sc]]]
9   TREV [[PileError, [sc]], 0, [], [internal, [sc]]]
9   TREV [[nSticksError, [sc]],0, [], [internal, [sc]]]
9   TREV [[GeneralTake, [sc]], 0, [], [internal, [sc]]]

outworlds=9
number of outworlds=1
SC: | pe MaryTake p=[2,2]
SC: |: gc
13 statechart sc
13   cluster Play [sc] = OCC [] **
13   leafstate John [Play,sc] = VAC []
13   leafstate Mary [Play,sc] = OCC [] **
13   VAR INTEGER i1 [sc] =5
13   VAR INTEGER i2 [sc] =2
13   VAR INTEGER i3 [sc] =7
13   VAR INTEGER p [sc] =2
13   VAR INTEGER take [sc] =2
13   TRACE = [OK,nSticksError,OK]
13   TREV [[MaryTake, [sc]],2,[[r,0,3],[r,0,7]], [external, [sc]]]
13   TREV [[PileError, [sc]], 0, [], [internal, [sc]]]
13   TREV [[nSticksError, [sc]],0, [], [internal, [sc]]]
13   TREV [[GeneralTake, [sc]], 0, [], [internal, [sc]]]

outworlds=13
number of outworlds=1
SC: | pe JohnTake p=[3,7]
SC: |: gc
17 statechart sc
17   cluster Play [sc] = OCC [] **
17   leafstate John [Play,sc] = VAC []
17   leafstate Mary [Play,sc] = OCC [] **
17   VAR INTEGER i1 [sc] =5
17   VAR INTEGER i2 [sc] =2
17   VAR INTEGER i3 [sc] =7
17   VAR INTEGER p [sc] =3
17   VAR INTEGER take [sc] =7

© Graham G. Thomason 2003-2004
17 TRACŒ {OK,OK,nSticksError,OK}
17 TREV [(MaryTake,[sc]),2,[{r,0,3},{r,0,7}],[external,[sc]]]
17 TREV [(PileError,[sc]),0,{},[internal,[sc]]]
17 TREV [(nSticksError,[sc]),0,{},[internal,[sc]]]
17 TREV [(GeneralTake,[sc]),0,{},[internal,[sc]]]

outworlds=[17]
number of outworlds=1
SC:: pe MaryTake p=[4,1]

20 statechart sc
20 cluster Play [sc] = OCC () **
20 leafstate John [Play,sc] = VAC () **
20 leafstate Mary [Play,sc] = OCC () **
20 VAR INTEGER i1 [sc] =5
20 VAR INTEGER i2 [sc] =0
20 VAR INTEGER i3 [sc] =0
20 VAR INTEGER p [sc] =3
20 VAR INTEGER take [sc] =7
20 TRACŒ = [nSticksError,OK,OK,nSticksError,OK]
20 TREV [(MaryTake,[sc]),2,[{r,0,3},{r,0,7}],[external,[sc]]]
20 TREV [(PileError,[sc]),0,{},[internal,[sc]]]
20 TREV [(nSticksError,[sc]),0,{},[internal,[sc]]]
20 TREV [(GeneralTake,[sc]),0,{},[internal,[sc]]]

outworlds=[20]
number of outworlds=1
SC:: pe MaryTake p=[1,4]

24 statechart sc
24 cluster Play [sc] = OCC () **
24 leafstate John [Play,sc] = VAC () **
24 leafstate Mary [Play,sc] = OCC () **
24 VAR INTEGER i1 [sc] =1
24 VAR INTEGER i2 [sc] =0
24 VAR INTEGER i3 [sc] =0
24 VAR INTEGER p [sc] =1
24 VAR INTEGER take [sc] =4
24 TRACŒ = [OK,nSticksError,OK,OK,nSticksError,OK]
24 TREV [(JohnTake,[sc]),2,[{r,0,3},{r,0,7}],[external,[sc]]]
24 TREV [(PileError,[sc]),0,{},[internal,[sc]]]
24 TREV [(nSticksError,[sc]),0,{},[internal,[sc]]]
24 TREV [(GeneralTake,[sc]),0,{},[internal,[sc]]]

outworlds=[24]
number of outworlds=1
SC:: pe JohnTake p=[1,1]

29 statechart sc
29 cluster Play [sc] = OCC () **
29 leafstate John [Play,sc] = VAC () **
29 leafstate Mary [Play,sc] = OCC () **
29 VAR INTEGER i1 [sc] =0
29 VAR INTEGER i2 [sc] =0
29 VAR INTEGER i3 [sc] =0
29 VAR INTEGER p [sc] =1
29 VAR INTEGER take [sc] =1
29 TRACŒ = [Mary Wins,OK,OK,nSticksError,OK,OK,nSticksError,OK]
29 TREV [(MaryTake,[sc]),2,[{r,0,3},{r,0,7}],[external,[sc]]]
29 TREV [(PileError,[sc]),0,{},[internal,[sc]]]
29 TREV [(nSticksError,[sc]),0,{},[internal,[sc]]]
29 TREV [(GeneralTake,[sc]),0,{},[internal,[sc]]]

outworlds=[29]
number of outworlds=1
SC::

© Graham G. Thomason 2003-2004
3. Source listing of the STATECRUNCHER model

-------------------
// Module:    Nim.scs.txt
// Author:    Graham Thomason, Philips Digital Systems Laboratories, Redhill
// Date:      11 July, 2003
// Purpose:   StateCruncher model: The Game of Nim (McMorran & Powell "Z.." p118,224)
// Copyright (C) 2003 Philips Electronics N.V.
// Revision History:
//-------------------

statechart sc(Play)

PCO internal,external;

event JohnTake,MaryTake@external;

enum int3 {0, . ., 3};
enum int7 {0, . ., 7};

int7 il=5, i2=6, i3=7; // Sticks remaining on each pile
int3 p=0;          // Pile from which sticks are taken
int7 take=0;       // Number of sticks taken from pile

cluster Play(John,Mary) {
    /* If Mary took the last stick, we are now in John, and John wins */
    PileError  { trace("PileError");   };
    nSticksError { trace("nSticksError"); };
    GeneralTake
    { if (p==1) { il-=take; trace("OK"); } }
    if (p==2) { i2-=take; trace("OK"); } 
    if (p==3) { i3-=take; trace("OK"); } 
    if (il==0 && i2==0 && i3==0 && in(Play. John))  {trace("John Wins");}
    if (il==0 && i2==0 && i3==0 && in(Play.Mary))  {trace("Mary Wins");}
};

// The occupied cluster state indicates whose turn it is

state John (JohnTake{::p,::take})
    [ (p==1 && il>=take)| | (p==2 && i2>=take) | | (p==3 && i3>=take) ] -> Mary
    {fire GeneralTake;  };

JohnTake{::p,::take} /*internal transition*/
    [ p<1 | | p>3 ]
    {fire PileError;} ;

JohnTake{::p,::take} /*internal transition*/
    [ (p==1 && il<take)| | (p==2 && i2<take) | | (p==3 && i3<take) ]
    {fire nSticksError;} ;

© Graham G. Thomason 2003-2004
state Mary {MaryTake(::p,::take)
   \[ (p==1 \&\& i1>=take) \| (p==2 \&\& i2>=take) \| (p==3 \&\& i3>=take) \]
   -> John
   {fire GeneralTake; }

   MaryTake(::p,::take) /*internal transition */
   \[ p<1 \| p>3 \]
   {fire PileError; }

   MaryTake(::p,::take) /*internal transition */
   \[ (p==1 \&\& i1<take) \| (p==2 \&\& i2<take) \| (p==3 \&\& i3<take) \]
   {fire nSticksError; }

}\-----------------------[end of module]----------------------------------------
4. 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 3 [StCrGP4] GP4 - The Generic Prolog Parsing and Prototyping Package (*underlies the STATECRUNCHER compiler*)
Related report 4 [StCrParsing] STATECRUNCHER Parsing
Related report 5 [StCrTest] STATECRUNCHER Test Models
Related report 6 [StCrFunMod] State-based Modelling of Functions and Pump Engines
References

[McMorran] Mike McMorran and Steve Powell
Z Guide for Beginners

[Hayes] Ian Hayes (editor)
Specification Case Studies

UML specifications are available from this website.

[Z font] Shareware by Lubos Mikusiak, lmikusia@ingr.com, available from the site http://www.informatikforum.org
Appendix 6

Bibliography and References

An Appendix to the Thesis
The Design and Construction of a State Machine System that Handles Nondeterminism

Graham G. Thomason
Bibliography and References

Graham G. Thomason

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

UniS

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

This annotated bibliography accompanies the thesis on *The Design and Construction of a State Machine System that Handles Nondeterminism* (called STATECRUNCHER) and is divided into five parts: (1) internal Philips publications relating to (conformance) testing, setting a backdrop; (2) systems and formalisms supporting state machines; (3) publications relating to state machines; (4) supporting projects / products / information of relevance to testing; (5) a consistent set of STATECRUNCHER references. In addition to state-based techniques, various other model-based testing techniques are touched upon within the various categories.
Table of Contents

1. Introduction ......................................................................................................................... 1
   1.1 Categorisation of references ......................................................................................... 1
   1.2 Abbreviations and definitions used in this appendix ................................................... 2
2. Internal Philips publications .................................................................................................. 3
3. Systems and formalisms supporting state machines or related models .............................. 8
4. Publications relating to verification, testing and/or state machines .................................... 19
5. Supporting projects / products / information ...................................................................... 38
6. STATECRUNCHER references ......................................................................................... 44
1. Introduction

1.1 Categorisation of references
The references have been arranged in categories, then alphabetically, as follows

- Internal Philips publications relevant to validation and verification (testing)
- Systems and formalisms supporting state machines and other model-based testing techniques
- Publications relating to state machines and other model-based testing techniques
- Supporting projects/products/information of relevance to testing
- The STATECRUNCHER references.

The Philips reports show some of the history in the company of state-based conformance testing, as a backdrop to the development of STATECRUNCHER.

Under *systems supporting state machines*, we include *model checking systems*, because whether or not they offer a simulation facility, they internally run some state machine engine. We will distinguish two kinds of tool in our annotations (rather than introducing separate categories): *model checkers* and *simulators/test oracles*. The corresponding activities may be called *validation* and *verification/testing* respectively, though 'verification' is often used of model checking, and we often meet the phrase 'verifying properties'. A software system needs a *design* and an *implementation*, and both need a separate kind of tool and activity to ensure the quality of the final system.

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

A slogan popular in Philips in the 1990s was: *Doing the right thing and doing things right*. This is like saying: *validating* the properties of the design, and *verifying* (testing) that the implementation conforms to the design. Both are extremely important, but distinct, though an occasional tool (e.g. SPIN) is suited to both.

We also note in our annotations whether a state-based testing system is of the *Labelled Transition System* (LTS) type or *(Mealy)* *Finite State Machine* (FSM) type. The former has
affinities with CCS and CSP; event sequences are the traces, and events are partitioned into input events and output events. The FSM approach defines a separate output alphabet. FSMs produce output on transitions, the trace of such systems. [Tretmans] regards the precise relation between testing theories based on the two approaches as an aspect of further study. STATECRUNCHER was designed as a test oracle, and the main thrust of the thesis is that its design will help in testing. Nevertheless it could be used to validate properties, given the aid of an additional tool communicating with it, because it offers facilities which can help in exploring state spaces. STATECRUNCHER is more geared to the FSM approach than the LTS approach. In [StCrSemComp] we make some comparisons with the process algebras. Some papers describe work where the implementation language is SDL; this corresponds more to an LTS approach than an FSM one, because input and output messages are both analogous to events.

The main scope of the bibliography is state-machine systems (and how they have been used), whether commercial, proprietary, or academic, principally in a testing context, but also in a validating context. Test generation algorithms are surveyed, as being STATECRUNCHER's nearest neighbour in a tool chain. In addition we give some references for UML-based modelling other than dynamic modelling, and we mention a few other testing techniques: cause effect graphing, orthogonal array testing.

Under supporting projects/products/information we cover various tools, which, although they may appear to be a disparate collection, have proved to be of especial value in constructing testing tools and synthesizing tool chains. PROLOG features prominently in the list, being the implementation language of STATECRUNCHER.

Finally, the STATECRUNCHER references form a consistent set of documents describing the system from various angles at its latest release (1.05).

### 1.2 Abbreviations and definitions used in this appendix

We use abbreviations and technological terms, where not explained, sparingly in the annotations, but the following are so commonly needed as to be useful:

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Black box</td>
<td>Used of a state machine, this means that states themselves are not directly observable, but outputs on transitions are, and it is from these that a state may be deduced.</td>
</tr>
<tr>
<td>FSM</td>
<td>Finite State Machine</td>
</tr>
<tr>
<td>IUT</td>
<td>Implementation Under Test</td>
</tr>
<tr>
<td>LTS</td>
<td>Labelled Transition System</td>
</tr>
<tr>
<td>NFSM</td>
<td>Nondeterministic Finite State Machine</td>
</tr>
<tr>
<td>OSI</td>
<td>Open Systems Interconnection</td>
</tr>
<tr>
<td>SUT</td>
<td>System Under Test</td>
</tr>
</tbody>
</table>

© Graham G. Thomason 2003-2004
2. Internal Philips publications

The Philips laboratories involved are:
- PRL (Philips Research Laboratories, Redhill)
- PDSL-R (Philips Digital Systems Laboratories, Redhill)
- Nat. Lab. (Natuurkundig Laboratorium, Philips Research Laboratories, Eindhoven)
- PRI-B (Philips Research India - Bangalore).

These reports cover state-based testing and related issues in various ways: early studies, tooling approaches, transition tour approaches, and case studies.

Automation of Software Testing: A Case Study on a Real-Time Embedded System
PRL Technical Note 3373, September 1995

This report describes early work within Philips Research to automate testing of two Interactive TV applications (an interactive quiz show and interactive shopping –both teletext based). The work featured:
- state-based testing, using the public domain tool [DejaGnu] as a test harness, with custom code being written in Expect/TCL. The state behaviour was defined using state-relation tables.
- code coverage, using the [McCabe] toolset.

Out of 1400 tests, 76 failed. Two major errors relate to a requirements omission and an implementation omission. The combination of the two techniques makes it possible to see how much code is exercised by a state model. Branch coverage (stronger than statement coverage) figures in modules varied from 26% to 100%. The low figures were often where error recovery code had not been exercised; more tests could be devised to increase the coverage.

[ECHSM] M.J. Hollenberg
Extended Hierarchical Concurrent State Machines, Syntax and Semantics
Nat. Lab. Report, version 0.4, 25 October, 1999

This is a document describing the syntax for an ECHSM (Extended Concurrent Hierarchical finite State Machine) language. The syntax is an extension to that of [CHSM]. The semantics are practically "as in [CHSM]". The purpose of the language
is to flatten ECHSM's to FCHSM's (see [FCHSM]) for use with [PHACT]. The grammar has been largely adopted by STATECRUNCHER, with extensions, and with the semantics extended for nondeterminism.

[FCHSM] M.J. Hollenberg
Flattened Concurrent State Machines, Syntax and Semantics
A language for describing flattened concurrent hierarchical state machines, derived from ECHSM's (see [ECHSM]), for use with [PHACT].

[GFET] G.G. Thomason
A GUI Front End for Testing
Program GFET (Multi-threaded version)
User Manual, Version 2.0/5.0
PRL Technical Note 3875, July 1999
A tool to give a Windows user interface to embedded software that does not have a user interface. It allows for control of 10 threads on which portions of software can be run. It provides easy implementation of stubbed functions as dialogue boxes. This enables the software to be tested using button-pressing, edit-box-communicating Windows software testing tools, such as WinRunner [WinRun] to test embedded software. The test script may make use of a state-relation package [Trew 98].

[Koppalkar 02] Nitin Koppalkar and Animesh Bhowmick
Integration of Generic Explorer with the TorX Tool Chain
This report describes how STATECRUNCHER, being an explorer in [TorX] terms, can be integrated into the TorX tool chain. The actual integration took place later, when STATECRUNCHER had a socket interface.

[Koppalkar 03] Nitin Koppalkar
Interfacing STATECRUNCHER with TorX for demonstrating the state-based testing technique taking MG-R components for a case study
This report shows STATECRUNCHER in the [TorX] tool chain in action testing a TV software component.
The report describes the relevant state of the art (at the time of writing) in conformance testing, with explanations of test sequence generation by the T, D, W and U methods: transition tours, distinguishing sequences, characterisation sets and unique I/O sequences, and extensions to these. Tooling is SDL, LOTOS and Estelle based, with TTCN used as a test definition format.

This report describes how a state-based model written in [CHSM] can be flattened, and then have its variables expanded, to give final output in a Flattened State Machine language to be used as input to [PHACT]. The flattening process takes place by driving CHSM through its state space. The concepts were used in testing American digital television (DTV '98).

PHACT (Philips Automated Conformance Tester) is built on a proprietary state-based testing tool, KPN's Conformance Kit. KPN [http://www.kpn.com] is a large Dutch telecom company, the main successor to the Dutch PTT. PHACT does not support hierarchy (so hierarchical state models must be flattened). It has been used to test an MPEG source decoder (DIVAS) and American digital TV (DTV'98). Some handling of nondeterministic situations can be managed by defining intermediate states [p.41].

The problem addressed in this report is that of generating transition tours round a state transition diagram. A tour is then effectively a black-box test sequence, since it does not rely on being able to set any state directly, (which would be white-box control). The problem of generating the tour is known as the Chinese Postman Problem. Part of the solution is to solve an assignment problem. For an optimal solution, Raptis refers us to the Hungarian solution, Christos H. Papadimitriou and Kennett Steiglitz, Combinatorial Optimization: Algorithms and Complexity, Prentice Hall, 1982. This has cubic complexity. Raptis presents a faster algorithm for a non-optimal, but near-optimal solution, with some experimental results.
[Raptis 99a] D. Raptis

A Modelling and Testing Approach for Horizontal Communication in the TV Platform

PRL Technical Note 3893, April, 1999

This report describes how [CWB] (Concurrency Workbench) was used to model the state-based behaviour of the composition of two formal software components given their interface specifications. The components handle parts of an end-to-end analogue signal flow: a tuner and high-end output processor. The interactions of such components are only with adjacent components (horizontal communication) - so obviating the need for a manager program that knows the whole configuration. This scheme facilitates system synthesis from components, but integration testing is needed to ensure it works.

[Raptis 99b] D. Raptis

Modelling and Validation of Concurrent Programs using CCS


This report shows how CCS agents, with and without value passing, can be designed to model data types, variables and algorithms. Semaphores and Peterson's algorithm for mutual exclusion are described as examples. A pre-processor using a Unix sed script is described for translating from a user-friendly syntax to CCS. An introduction to verification of model properties as supported by CTL*, rather than the modal mu calculus of CCS, is given.

[Thomason] G.G. Thomason

Component Binding in Composite Models for State-based Testing

PRL Technical Note TN 4102, August, 2001

The aim of this report is to identify how systems built from software components will need to be tested. A tool chain is required which can automatically generate and execute tests —in particular integration tests. The generation side must use models of the behaviour of individual components and of their binding which ‘wires up’ the complete system, and produces tests and their ‘oracle’ from the model —which may incorporate several alternative results in the event of nondeterminism. Solutions are explored involving compositions of STATECRUNCHER models, using a preprocessor to make model bindings in the same way that system bindings are made.


State-based Testing with WinRunner: the State-Relation Package

PRL Internal Note SEA/704/98/05, June 1998

This package, allows a WinRunner [WinRun] test script to loop over tests defined by state relation tables and so execute state-based tests.
This presentation describes the difference between ordinary object-oriented development and component development and the impact of that on testing. The need for good, structured integration testing is all the more important. (State based testing can be expected to be a major part of this).

This report addresses the practicalities of using STATECRUNCHER to model systems of software components.

This is a Philips proprietary test harness for embedded systems with a host side part and a target side part.

This illustrates the effectiveness of state-based testing. In a DVD player, errors (sometimes many) were found in every module tested – even though this was after hand-crafted conventional tests had been run. The modules were: the Loader Subsystem, the Media Access module, the CD-DA Playback module, and the VCD Playback module.
3. Systems and formalisms supporting state machines or related models

[Agedis] www.agedis.de

A consortium project headed by IBM Research Laboratory, Haifa, with the aim of "...automating software testing and improving the quality of software while reducing the expense of the testing phase... by developing a methodology and tools for the automation of software testing in general, with emphasis on distributed, component-based software systems". A publication Model based test generation tools by Alan Hartman gives a list of the main tools available. Commercial tools: [TVEC], [Conformiq], [Reactis], Icontract, [Tau], Testmaster, Unitek. Proprietary tools: [GOTCHA-TCBeans], Ucbt-Salt, [ASML], [PTK]. Academic tools: Spectest, Mulsaw, Toster, TGV/CADP, [TorX]/CADP, [Cow_Suite].

[Argos] F. Maraninchi
The Argos Language: Graphical Representation of Automata and Description of Reactive Systems
IEEE Workshop on Visual Languages, Kobe, Japan, October 1991

Argos supports the graphical development of statecharts. The graphical items correspond to a syntax, which directs the graphical editor. Nondeterminism is detected so that the user can remove it. The system supports UML-like models, including (synchronous) broadcast events. Verification is performed in an environment called Argonaute, using an automaton comparator called Aldebaran, for which the following reference is given: J.C. Fernandez, An Implementation of an Efficient Algorithm for Bisimulation Equivalence, Science of Computer Programming, vol. 13, 2-3, May, 1990. That article and additional information on Aldebaran can be found on the internet at the INRIA (Institut National de Recherche en Informatique et en Automatique) site: http://www.inrialpes.fr

http://www.artisansw.com/products/professional_overview.asp

From the Real Time Studio Professional web page

"Already an acknowledged leader in providing modelling support for system engineers, ARTiSAN has added a powerful set of new enhancements to its system validation functionality, so that engineers can:

• Build and simulate advanced state models for system behaviour:"
• Use events straight from the system architecture model
• Add timers and timed events
• Use drag/drop to populate state triggers, actions and guards
• Verify system response to external and internal events before building”

The transition semantics appear to be in agreement with UML.

[ASML]  
(Abstract State Machine Language)  
http://research.microsoft.com/fse/asml  
The above site includes a 76-page tutorial for download. ASML is “an executable specification language based on the theory of Abstract State Machines...good for testers...”. The language is very reminiscent of imperative languages, (such as ‘C++’ – ASML has classes), rather than the reactive systems approach of other state machine systems such as [STATEMATE]. It has processing blocks divided into steps, allowing parallelism within steps, where updated variable values only take effect after a step. The notion of state is simply related to variable values at the end of a step, and transitions are the act of processing a step. The language includes sets and sequences, and maps (equivalent to associative arrays of Perl, or hash tables in database systems) Nondeterminism can be specified, but the system then makes one choice. There is support for predicate logic, e.g. for all...holds and exists...where. Microsoft state that ASML is being used for conformance checking. For a paper on Sequential Abstract State Machines, see [Gurevich].

[Caliber]  
http://www.nohau.se/products/kravhantering.html  
A cause-effect graphing tool that has been used at Philips, originally called SoftTest from Bender and Associates, then apparently under Borland called Caliber-RBT and now under Nohau called Caliber-RM. Cause-effect graphs are described in [Myers].

[CCS]  
Calculus of Communicating Systems  
A process calculus. See [Milner], [Bruns]

[CHSM]  
Paul J. Lucas  
An Object-Oriented System for Implementing Concurrent, Hierarchical, Finite State Machines  
MSc. Thesis, University of Illinois at Urbana-Champaign, 1993  
http://homepage.mac.com/pauljlucas/software.html  
CHSM stands for Concurrent Hierarchical finite State Machines, and (in context) Lucas's implementation of a language for them. The concurrency and hierarchy are expressed as 'sets' and 'clusters'. It allows for transition actions, which may be broadcast (i.e. fired) events. The language is easy to grasp, and although apparently not designed with testing applications in mind, it is at a suitable level for ordinary developers and testers to use. The language is implemented by conversion to C++ using the Unix tools YACC and LEX. CHSM supports embedded C++ in a source
model. CHSM prevents transition ‘cycling’ (potentially possible through broadcast events) by only allowing any one transition to be taken once during the processing of a top-level event. A CHSM model may contain nondeterministic transitions, but the system will take just the first one it finds.

A commercial tool supporting batch and on-the-fly testing, based on UML dynamic models.

[Cow Suite] Francesca Basanieri, Antonio Bertolino, Eda Marchetti
The Cow_Suite Approach to Planning and deriving Test Suites in UML Projects
Instituto di Elaborazione della Informazione, Pisa
Cow_suite tools generate test cases from UML diagrams, based on the analysis of Use Case diagrams and Sequence Diagrams. No translation into an intermediate notation is needed. A cost-weighted strategy is used, assigning weights to nodes of derived trees, to select the most ‘important’ test cases from all possible use cases and message sequences. The user can choose either a fixed number of tests, or fixed functional coverage. Managers provide ‘importance’ criteria. Cow_suite does not execute tests; for this a separate driver is required.

[CSP] Communicating Sequential Processes
A process calculus. See [Hoare], [Schneider].

[CTL] Computation Tree Logic.
This temporal logic is embodied in a language called CTL*. See [Emerson], [Bérard].

[CWB] The Edinburgh Concurrency Workbench
http://www.dcs.ed.ac.uk/home/cwb/
This tool expresses its designs in the Calculus of Communicating Systems (CCS). It is a powerful tool, and is popular as a research tool, but it is not aimed at the ordinary software developer in industry. It supports nondeterminism at a transition level, so that the user can choose between transitions even where some of them are triggered off the same event. (Contrast this with STATECRUNCHER, which supports nondeterminism at the event level, relieving the user of the need to detect and manage multiple nondeterministic transitions in their own loop).

[Design/CPN] Design/Coloured Petri Nets
Initially developed by Meta Software Corp, Cambridge MA USA, and the CPN Group at the University of Århus, Denmark. Available from http://www.daimi.au.dk/designCPN
Design/CPN allows one to edit, simulate and verify large hierarchical coloured Petri nets ([Bérard, Ch14]). Since Petri nets can be used to model state-based systems (see [Murata]), the tool can be used to verify them.

[ESPRESS] Engineering of safety-critical embedded systems
http://www.first.gmd.de/~espress

“ESPRESS aims to increase productivity in developing complex, safety-critical, embedded systems and enhance the reliability of such systems by the development of a methodological tool-supported software technology for specific application areas covering the whole life-cycle. The project focuses on the application area of automobile electronics and traffic light control. ... Essential features are the explicit separation of specifications into functional and safety relevant parts, the combination of graphical (statecharts) and formal methods (Z) as well as verification, code-generation, systematic testing and automatic test evaluation.”

Tool support is based on STATEMATE. See [Büssow] for a description of the formalism used: $\mu$SZ. See [Fuhrmann] for another ESPRESS publication, on the verification of STATEMATE statecharts via the CSP verification tool [FDR].

[Estelle] ISO 9074 (draft)
http://www.estelle.org

Estelle is an ISO Formal Description Technique, i.e. a specification language, for concurrent distributed systems. Compare [LOTOS], a companion ISO standard, and [SDL], an ITU standardized language, with which it has some commonality. Estelle is based on modules and interaction points, and uses the asynchronous (non-blocking) send for intermodule interaction, and also shared variables.

Estelle is championed by the LOR, département LOgiciels-Réseaux, (Department of Network Software)
http://www-lor.int-evry.fr/
LOR has produced EDT = Estelle Development Toolset.
For a tutorial, see [Budkowski].

[FDR] Failures Divergences Refinement checker
A CSP-based model checker from Formal Systems Europe:
http://www.fsel.com/
A companion tool is [Probe].

A proprietary IBM tool “designed to assist testers in developing, executing and organizing function tests direct against Application Program Interfaces (APIs) and software protocols written in Java, C or C++”. The tool has been used in the [Agedis] project. The test process is one of producing a state machine model of system specifications from which an abstract test suite is generated by GOTCHA. This is
translated into test scripts by TCBEANS which are run via an executor or on-the-fly. (Compare [TorX]). From the file system example, it appears that the user must write switch statements in an imperative language to produce the state machine model, but a UML modelling language has been defined in the [Agedis] project. Nondeterminism support is claimed (no details given).

[LOTOS] ISO/IEC standard 8807
LOTOS (Language of Temporal Ordering Specification) is an ISO Formal Description Technique, i.e. a specification language, for concurrent distributed systems. Compare [Estelle]. It has historical connections with CCS and CSP. It is algebraic, using processes, events, ordering operators etc. Synchronisation is by shared events as in CSP. Nondeterminism is implicit in parallelism (various interleavings), or can be specified by offering the same event name more than once with the choice operator \( [\] \) (example from Kenneth Turner, Univ. of Stirling):

\[
(eat\_out; \text{CHINESE MEAL})[\] (eat\_out; \text{INDIAN MEAL})
\]

Many implementations of LOTOS exist. LOTOS has been used as the explorer element of the [TorX] tool chain.

http://www.telelogic.com/products/objectgeode/articles.cfm#simulation

The above downloadable paper describes state-base testing from the perspective of exploring the state space of a model written in SDL (Specification and Description Language): *Automated Test Generation with ObjectGeode Test Composer, Alain Kerbrat*.

Abstract: This paper presents the advanced features provided by ObjectGeode Test Composer, a Test Suite generator for conformance testing of distributed systems:
- Test purposes generation based on structural coverage,
- Test cases generation based on state space exploration,
- Interactive and batch generation,
- Test suite structuring and production


[PLTL] Propositional Linear Temporal Logic
[Probe] Process Behaviour Explorer
A tool to interpret and animate CSP specifications from Formal Systems Europe:
http://www.fsel.com/
A companion tool is [FDR].

[PROMELA] (PROcess MEta LAnguage)
http://cm.bell-labs.com/cm/cs/what/spin/Man.Quick.html
The language allows for the dynamic creation of concurrent processes. Communication via message channels can be specified to be synchronous or asynchronous. Support is provided by [SPIN], which can perform random or interactive simulations of the system’s execution or exhaustive verification of the system’s state space (e.g. checking for the absence of deadlocks). PROMELA has been used as the explorer in the [TorX] tool chain.

[PTK] see [BakerP]
A Motorola in-house tool used to generate conformance tests (SDL or TTCN) from Message Sequence Charts (MSCs) and Process Data Unit specifications (PDUs).

http://www.rational.com/products/rose/real_time/rtrose.jsp
From the website on Rational Rose RealTime:
“Developers of embedded, real-time and network systems software applications develop some of the coolest code for the most technologically challenging products and systems. Because of this, they face several challenges that other development environments don’t. Many times, this type of software is highly event-driven, concurrent, and often distributed. Stringent requirements must be met for latency, throughput, and dependability. Capturing and effectively communicating designs for such systems can be tough without the right tools. Rational Rose RealTime for Windows or UNIX is the best solution for accelerating your devices & embedded systems software development projects quickly, easily and completely.”
The transition semantics appear to be as described in UML books.

[Reactis] http://reactive-systems.com
A graphical tool that supports “a large subset of the discrete-time subset of Simulink and Stateflow”. It may also interact with MATLAB for calculations. For Simulink, Stateflow and MATLAB, see http://www.mathworks.com. Simulink is strong in numerical algorithms and is aimed at control systems design, signal processing, and communication systems. Stateflow is the state-transition tool. Apart from many features apparently equivalent to UML statecharts, it supports temporal logic and “schedules transitions and events using temporal operators ("before", "after", "at", "every").” In Reactis, state-transition diagrams are shown graphically, and input events can be selected from a source, the default being random events, which it is
admitted (on the Getting Started web pages) may lead to poor coverage. State-transition coverage is indicated by green and red colouring of the diagram. Features appear to be geared to interactive simulation: oscilloscope-like windows showing real-time progress of numerical outputs, variable watching, breakpoints, and stepping through model execution.


RHAPSODY is a CASE-tool from I-Logix. From the web-page:
   Rhapsody is an enterprise-wide visual programming environment that allows corporations to build and deploy real-time embedded systems and software applications. Rhapsody is designed and optimized for the special needs of the embedded market. Real-time behavioral semantics, target real-time operating system support, model/code associativity, design-level debugging, and production quality code generation increase developer productivity. Rhapsody customers regularly report design cycle reduction of more than 30%, even on the first project.

The semantics of RHAPSODY (and STATEMATE) are described in [Harel-96].

[RSML] Requirements State Machine Language

RSML is Mealy-machine based (actions on transitions). See [Heimdahl] for a description of its semantics, and [Leveson] for its origins. [Von der Beeck] gives the following earlier reference with the same title as [Leveson]:

N. Leveson, M. Heimdahl, H. Hildreth, J. Reese
Requirements Specification for Process Control Systems

RSML allows for state arrays. Messages can be sent between separate state machines. It supports timing functions. The semantics allow for looping round transitions. Although developed as a specification language, a simulator for RSML has been built by Heimdahl.

[SDL] Specification and Description Language

This language has been standardized by the ITU (International Telecommunications Union) as ITU-Z.100 and Z.105. It uses asynchronous message (=signal) passing between processes. It supports objects and inheritance. The basic graphical symbols represent the following items: state, message output (send), message input (consume), message save (if not consumed), task (perform some action). The notation is convenient for constructing a state transition diagram in small, page-sized portions at a time. Nondeterminism can arise where different interleavings of message arrival are possible.

[SMV] Symbolic Model Verifier

A model checking tool developed by K.L. McMillan under the guidance of E.M. Clarke at Carnegie-Mellon University. It uses CTL* as its temporal logic language.
(see [Emerson], and uses binary decision diagrams in its implementation. Summarised in [Bérard, Ch.12].
SMV is available from
http://www.cs.cmu.edu/~modelcheck/smv.html
The following site is a tutorial and gives an example of modelling a semaphore: Model checking lecture notes by Marsha Chechik (U. Toronto)
www.cs.toronto.edu/~chechik

[SPIN] A simulation and verification tool.
SPIN was mainly developed by G.J. Holzmann at Bell Labs. The following site gives a general description, many theoretical references, workshop information etc.
From [Bérard, p.139]: SPIN was designed for simulation and verification of distributed algorithms. The systems must first be described in [PROMELA]. Spin has two modes: (1) simulation (2) property-checking (using PLTL). Key feature: state space reduction mechanisms, on-the-fly verification and hashing (allowing it to work with 10^7+ states). SPIN was used in the [TorX] tool chain for on-the-fly conformance testing in the Côte de Resyste project (also ref. [Torx]), using a PROMELA description of the model, supporting nondeterminism.

[Stateflow] see [Reactis]

STATEMATE is a statechart system from I-Logix. From the web-page:
I-Logix' Statemate MAGNUM is the most comprehensive graphical modeling and simulation tool for the rapid development of complex embedded systems. Statemate MAGNUM provides a direct and formal link between user requirements and software implementation by allowing the user to create a complete, executable specification. Operating on an engineering workstation or PC, Statemate MAGNUM creates a visual, graphical specification that clearly and precisely represents the intended functions and behavior of the system being specified. This specification may be executed, or graphically simulated, so the system engineer can explore what if scenarios to determine if the behavior and the interactions between system elements are correct. These scenarios can be captured and included in Test Plans which are later run on the embedded system to ensure that what gets built meets what was specified. This executable specification is also used to communicate with the customer or end user to confirm that the specification meets their requirements.
The semantics of STATEMATE are described in [Harel-96].

Harel's statecharts and I-Logix's STATEMATE differ from UML's interpretations. Even Rhapsody, from I-Logix, conforms to the UML view. The main differences are
1) The form of parallelism allows for variables to be altered in one place, but retain their original value when used in another place. UML assumes a specific sequence.

2) Harel (and CHSM) prioritize giving the outermost transitions on the same event priority; UML takes an object-oriented derived-class-overrides view and gives the inner transition priority:

[TGV]  
http://www.irisa.fr/pamapa/VALIDATION/TGV/TGV.html

"TGV (Test Generation with Verification technology) is a prototype for the generation of conformance test suites for protocols. It is based on the model of input/output (labelled) transition systems (IOLTS) and uses algorithms coming from verification technology. TGV has been developed in collaboration with Vérimag Grenoble and uses libraries of the César-Aldébaran Distribution Package (CADP) developed by Verimag Grenoble and VASY from Inria Rhône Alpes. A first prototype has been connected to the GEODE tool (Verilog) and allows the production of test suites in the TTCN format (Tree and Tabular Combined Notation) from SDL specifications." [Du Bousquet] describes the use of TGV in conjunction with [TorX], for random testing.

[TorX]  
Côte de Resyste project: http://fmt.cs.utwente.nl/CdR  
TorX tool: http://fmt.cs.utwente.nl/tools/torx/torx-intro.1.html

TorX comes from the Côte de Resyste (COnformance TEsting of REactive SYSTEms) project, a research and development project (1998-2002) funded by the Dutch Technology Foundation STW (http://www.stw.nl/). It 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)

It aims to develop methods and techniques to build a tool for specification-based testing in an automated way based on formal methods. Based on formal testing theory and languages (LOTOS, SDL, TTCN, PROMELA...), the approach is the Labelled Transition System one, with a partition between outputs and (always enabled) inputs. It defines conformance of an implementation \( i \) to a specification \( S \) as:

- \[ i \text{ ioco} S \equiv \forall \sigma \in \text{Straces}(s) : \text{out}(i \text{ after } \sigma) \subseteq \text{out}(s \text{ after } \sigma) \]

Tretmans explains this as: \( i \text{ ioco}-\text{conforms to } s \) iff

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

A test suite \( T \) is sound if \( i \text{ ioco } s \Rightarrow i \text{ passes } T \).

A test suite \( T \) is exhaustive if \( i \text{ passes } T \Rightarrow i \text{ ioco } s \).

TorX is a tool chain, supporting on-the-fly testing, consisting of an Explorer-Primer-Driver-Adapter-IUT, as follows:
A commercial set of tools integrating requirements and test, listed by the [Agedis] consortium. One mode of testing is model-based testing. The web pages do not elaborate on models supported, (UML dynamic models?). The T-VEC “tabular modeler” is derived from the US Naval Research Center's SCR (Software Cost Reduction) model, which is a requirements formalism, amenable to model checking, e.g. by SPIN.

(The Object Management Group Website)
UML specifications (v. 1.5, November, 2003) are available from the website.
Section 2.12 is on State Machines, which are a subpackage of the Behavioral Elements Package, which also includes Collaborations, Use Cases and Activity Graphs.

UML is a visual modelling language rather than a visual programming language [section 1.5.1.1, pp.1-7], so a direct comparison with STATECRUNCHER is not always possible. STATECRUNCHER is close to UML in semantics, and it is certainly our aim to align STATECRUNCHER as precisely as possible with UML if we have the opportunity for future developments. We note the following features of UML:

- **Change events** (lambda transitions), e.g. transitions triggered by data taking on a certain value. There are semantic issues as to when data is allowed to trigger such a transition.

- **Deep history and shallow history vertices** (i.e. as transition targets, also known as pseudo-states, so that different transitions can target a composite state individually invoking deep history, shallow history or no history). These are on STATECRUNCHER's wish-list.

- **Joins, forks, junctions and choices.** STATECRUNCHER can accommodate joins using the in(...) function as a guard. STATECRUNCHER has forks (the split operator). STATECRUNCHER can implement the functionality of junctions and choices using multiple transitions.

- **Deferrable events.** Not supported in STATECRUNCHER.

- **Do Activities,** describing processing associated with being in a state.
- **Synch states**, used for ordering forks and joins.
- **Time Events**. Such an event can express expiry of a deadline. STATECRUNCHER does not have any special constructs for expressing time.
- **Firing priorities**. Transitions originating from a substate has priority over a transition originating from any of its containing states. STATECRUNCHER now (Release 1.03 and higher) conforms to this.

[VVT-RT]  
**Validation, Verification and Test of Real Time Systems**

A tool from Verified Systems International GmbH, Bremen, in co-operation with the Bremen Institute of Safe Systems (BISS) within the Center for Computing Technology (TZI) at Bremen University. It is based on CSP [Hoare]. For a paper on an application of it, see [Schlinghoff].

© Graham G. Thomason 2003-2004
4. Publications relating to verification, testing and/or state machines

[Alhir] Sinan Si Alhir
UML in a Nutshell
This book contains intense, concise detail on UML. Chapter 11 covers statechart diagrams. It elaborates on compound transitions (decision branching), and splitting/synchronizing control.

[BakerP] Paul Baker, Paul Bristow, Clive Jervis, David King and Bill Mitchell
Automatic Generation of Conformance Tests from Message Sequence Charts
System and Software Engineering Research Lab (UK), Motorola Labs
The paper describes how the PTK tool (Motorola proprietary) is used to generate conformance tests from Message Sequence Charts (MSCs) and Protocol Data Unit specifications (PDUs). PTK generates SDL of TTCN scripts. Interleaving semantics of MSCs are used to generate all traces of events. Nondeterminism is handled by generating separate scripts for separate outcomes, with one precise outcome giving a test result of pass, and alternatives giving a test result of inconclusive. This makes it possible to check that all nondeterministic outcomes have been obtained (but it is not explained how they might be stimulated).

[BCS-SIGIST] Standard for Software Component Testing
British Computer Society - Special Interest Group in Software Testing
This document contains a great number of definitions and descriptions of testing terms and metrics. It defines State Transition Coverage as follows: For single transitions, the coverage metric is the percentage of all valid transitions exercised during the test. This is known as 0-switch coverage. For \( n \) transitions, the coverage measure is the percentage of all valid sequences of \( n \) transitions exercised during the test. This is known as \((n-1)\) switch coverage.
[Beizer] B. Beizer  
Software Testing Techniques, 2nd edition  
A very good introduction to practical software testing in general, covering various testing techniques. Chapter 11 is on States, State Graphs, and Transition Testing. The book introduces a tabular representation of transitions. It contains good advice on what to model (p.389). All examples are presented as flat deterministic finite state machines.

[Belinfante] Axel Belinfante  
Formal Test Automation: A Simple Experiment  
(A [TorX] / Côte de Resyste report)
This paper describes TorX in use, with test scenarios specified in LOTOS, PROMELA and SDL, testing a conference protocol.

[Bérard] B. Bérard  
Systems and Software Verification  
This excellent book describes in turn automata, temporal logic, model checking, symbolic model checking, and timed automata. It is concerned with model checking, i.e. proving properties of a model, (so verifying a design), rather than testing a model against an implementation. The temporal logic languages CTL* and PLTL are used. Amongst the tools described are: SMV, SPIN and Design/CPN, (and some timed/real-time tools).

[Binder] Robert V. Binder  
Testing objects: State-based testing: Sneak paths and conditional transitions  
Object Magazine, October 1995, pp. 87-89
This article illustrates the practical need to test an object (it also applies to a system) with messages that should not be accepted (what STATECRUNCHER calls non-transitionable events), and to check that the state has not changed. This is, of course, in addition to normal transitioning tests. A bank account example is given. Code which allows an illegal transition is a called a sneak path; it could be deliberate for the purposes of theft or sabotage. An equivalent situation arises with transitions having a condition that evaluates to false. There is a discussion on how to handle illegal messages at a coding level.
The authors take a statechart model of an aircraft control system with commands `climb, descent, flaps_down, flaps_up, terminate, level`. The approach requires a deterministic specification and implementation. Unlike the STATECRUNCHER case, events can be combined and negated in labelling a transition: `command \land \neg terminate`. The approach is a black-box one, because states are distinguished using a \textit{characterisation set}, described here as a path which exists from one state but not another.

**[Booch]**
Grady Booch, James Rumbaugh, Ivar Jacobson

\textit{The Unified Modelling Language User Guide}

A tutorial by the original developers of UML. Chapters 21 and 24 are on State Machines and Statechart Diagrams.

**[Brinksma]**
Ed Brinksma

\textit{Testing Transition Systems: An Annotated Bibliography.}
University of Twente, The Netherlands, Formal Methods and Tools Group.
\url{http://fmt.cs.utwente.nl}

This paper covers developments in formal testing theory and formal test generation. Test generation products mentioned: TVEDA, TGV, TestComposer (SDL-based; all have fed into \textsc{ObjectGeode}); VVT-RT (which uses CSP), SaMsTaG and AUTOLINK (which derive tests from SDL).

**[Bruns]**
Glenn Bruns

\textit{Distributed Systems Analysis with CCS}

A book that teaches CCS with many examples (arbiter, triple-modular redundancy and others). Complementary to [Milner], which is the authoritative text.

**[Budkowski]**
A. Budkowski, P. Dembinski, M. Diaz

\textit{ISO Standardized Description Technique Estelle}

This is a tutorial on [Estelle], available from
\url{http://www-lor.int-evry.fr/idemcop/uk/est-lang/download/short-estelle-tutorial.pdf}
The μSZ notation is used in the [ESPRESS] project. It combines Z and Harel-style statecharts. Process classes are: *data space* (variables), *operational behaviour* (statechart structure and transitions), *behavioural constraints* (can be specified with a temporal logic), *structural embedding* (aggregations of instances of classes).

An early paper on obtaining and measuring state coverage. Discusses the use of $P$, a set of input sequences to take a machine to every source state of a transition and to trigger that transition. $P$ can be obtained from $T$, a testing tree, which is a recursive exploration of the state space from everywhere not seen before. Discusses further $W$, the characterization set, a set of input sequences capable of distinguishing the behaviours of every pair of states in a minimal finite state machine.

This project aims at making software component systems self-testable and run-time using Built-In Testing (BIT) facilities. These facilities are structured as additional interfaces to the components, a *provides* interface to test and a *requires* interface to notify. A tester component might contain corresponding interfaces that are bound to both of these interfaces. A small extra size overhead in the components is regarded as acceptable, as in the case of VLSI chips. Both interface contract and quality of service (QoS) can be tested. QoS testing is continuous verification against e.g. deadlock, time constraint violation, data corruption, user conformance, memory leaks or conflicts. An example of contract testing is actually *state transition testing*, in this case, of a stack (sections 3.3.1.1 - 3.3.1.2 of the Deliverable D3 document).

The context is FSMs. This paper gives an overview of the four main methods of generating test sequences for such deterministic FSMs: (1) the *transition tour* (the T method), (2) *distinguishing sequences* (the D method), (3) *characterizing sequences* (the W-method) and (4) *unique I/O sequences* (the U method). These are illustrated by worked examples.
A Côté de Resyste report (see [TorX], [Tretmans]), and so labelled transition system based. [SPIN] is used with PROMELA specifications, allowing for large state spaces. Nondeterminism is handled in an on-the-fly algorithm (section 3). Quiescence (no output) is also accepted if it is valid.

This paper describes the first experiment with [TGV] and [TorX] in combination. The system tested was a multicast protocol implementation (a kind of chatbox), specified in LOTOS. Manually generated and random testing were compared. An on-the-fly technique was used. Of 25 mutant systems (i.e. with seeded errors), manual testing found all but one. Random testing found all mutants.

This paper argues for the testing effectiveness of obtaining the boolean expression coverage criterion known as MC/DC (Modified Condition / Decision Coverage), as defined in the USA Department of Defense standard D0178B. In this standard, test cases are generated such that each term in the expression is shown to be capable of independently affecting the value of the whole expression. For an application in state-based testing, see [Offutt].

Chapter X Machines is the seminal publication on X-Machines. These are state machines that operate on data of type X as they transition.
E.A. Emerson and J.Y. Halpern

"Sometimes" and "Not Never" revisited:

On branching versus linear time temporal logic.


E. Farchi, A. Hartman and S.S. Pinter

Using a model-based test generator to test for standard conformance


This article describes state-based testing of a stack, a file system and a Java exception handler, and how the state explosion problem was avoided by using projection state and projection transition coverage as a means of specifying test criteria.

Kay Fuhrmann, Jan Hiemer

Formal Verification of STATEMATE Statecharts

An [ESPRESS] publication. A technique is given whereby STATEMATE statecharts are translated into CSP for verification with the [FDR] model checking tool. The hard part appears to be the translation of STATEMATE's step semantics.

Susumu Fujiwara, Gregor v. Bochmann

Test Selection Based on Finite State Models

IEEE Transactions on Software Engineering, Vol. 17, No 6, June 1991

The context is principally deterministic FSMs. The paper presents an optimization to the W method (see [Chow]), called the partial W method. The optimization is based on using an identification set to identify a state, rather than the characterization set. The identification set is a state-dependent subset of the characterization set. (If an identification set consists of a single sequence, it is equivalent to a UIO approach). Good worked examples are given. There is a discussion of the following testing issues: (A) implementations having more states than the specification, (B) issues arising from incomplete specifications, (C) synchronization of distributed systems, (D) specifications including data flow, (E) nondeterministic implementations and/or specifications, and (F) OSI protocol conformance testing.
The paper presents a test selection method for testing nondeterministic systems. The approach is the labelled transition system one, not the finite state machine one. A successful test run proceeds through all actions specified without deadlocking.

Sequential algorithms are related to Abstract State Machines by a correspondence between variable values and abstract state, though these states can be interpreted as structures of mathematical logic, and as memory. States are transformed in computation steps, which are related to transitions. Nondeterminism is seen as the environment making a choice. "Nondeterministic algorithms are special interactive programs (section 9)." [ASML] is a tool embodying the notions of Abstract State Machines. As with ASML, the nature of Abstract State Machines as described has an imperative rather than reactive character, (reinforced by the examples of Eratosthenes' sieve and Euclid's greatest common divisor algorithms).

This paper has effectively laid the foundations for modern approaches to state modelling. It elaborates on the concept of 'statecharts' (as opposed to flat state diagrams) which Harel had recently introduced [D.Harel, Statecharts: A Visual Formalism for Complex Systems, Science of Computer Programming, 8, 1987]. Harel's statecharts have XOR (called OR in [Harel 96]) and AND components, default states, history, and broadcast events. The paper discusses the semantics of statecharts using the concept of micro-steps, discussing such difficulties as the value of shared variables that can, in principle, be assigned simultaneously possibly conflicting values. Nondeterministic situations are recognized, and some constructs are introduced to resolve them to a deterministic course of action. The concepts of this paper led to the commercial product STATEMATE.

The paper underlies [CHSM] and so indirectly also STATECRUNCHER.
This paper gives the semantics that the I-Logix products \(\text{STATEMATE}\) MAGNUM and \(\text{RHAPSODY}\) employ. The products can be used for testing and for code synthesis. A notable feature is prioritized transitions. Nondeterminism is handled as follows:

- Conflicting transitions [pp.16-17] (STATECRUNCHER's fork nondeterminism) result in the generation of sets of steps (transitions and static reactions, the latter being equivalent to additional transitions). The selection can be carried out interactively by the user, or by specifying a selection criterion at the start. The dynamic tests tool will try out all the different possibilities in an exhaustive fashion. The code synthesized by the software code generator will select the first possibility it finds that is enabled and will proceed to execute it.

- Racing conditions [pp. 24-25]. Where there are multiple orderings (such as Fig. 25, where \(t_2\) and \(t_3\) race), the paper states that \(\text{STATEMATE}\) reports a racing condition.

The paper addresses completeness and consistency in a statechart. Statecharts are modelled as functions. The language used is \(\text{RSML}\), which is Mealy-machine based (actions on transitions). Robustness is defined by [p.363]: (1) every state must have a behaviour (transition) defined for every input; (2) the logical OR of the conditions on every transition out of any state must form a tautology; (3) every state must have a timeout. This is called \(d\)-completeness. The transition relation is made to behave as a function. In this way determinism is imposed in \(d\)-completeness. Completeness checking is maintained in composition of state machines.

The approach is Mealy FSMs, though in the guise of circuits that take inputs of 0 or 1 and produce outputs of 0 or 1. It is an early paper introducing and synthesizing distinguishing sequences as a means of state checking.
[Hierons 98] Rob 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

This paper shows how an implementation that is known to be a deterministic state
machine can be tested against a nondeterministic model of it. The paper introduces \( d \)-
distinguishing sequences, that distinguish two states in an NFSM provided the
implementation is deterministic (although it is not known how). On-the-fly tests learn
from the observed behaviour and so adapt the test generation accordingly.

STATECRUNCHER, in conjunction with other programs communicating with it, could
be of assistance in implementations of algorithms like this, perhaps by exploring a
nondeterministic UML model and helping find \( d \)-distinguishing sequences.

[Hierons 03] Rob M. Hierons

Generating Candidates when testing a deterministic implementation
against a Non-deterministic Finite State Machine

The Computer Journal, 46, 3, pp. 307-318

The paper addresses the problem of testing an implementation that is known to be
deterministic against a nondeterministic specification. A candidate is a deterministic
FSM that is generated from the nondeterministic specification and the
implementation. It has the property that if the implementation conforms to the
candidate, the implementation conforms to the specification. Tests can then be
derived from the candidate, using test generation algorithms for deterministic FSMs.

[Hoare] C.A.R. Hoare

Communicating Sequential Processes,

This book describes CSP, (Communicating Sequential Processes): a process algebra
(or calculus) for specifying state behaviour in terms of processes and events. There
are various operators for parallel composition of processes. Ordinary engagement of
two or more processes is based on sharing of events in their 'alphabet'. There are
operators \( (\Pi, \Pi) \) for nondeterministic compositions. Algebraic laws enable rewriting,
simplification and comparison of process expressions.
This paper defines static semantics of statecharts and identifies types of nondeterminism. The semantics allow for conjunctions two or more simultaneous events and their negations, e.g. \( \alpha \land \neg \beta \), (unlike STATECRUNCHER). Nondeterminism in a statechart is identified as:

- **external** nondeterminism, where with two simultaneous events the system can have differing resultant states.
- **internal** nondeterminism, where there are different resulting states after processing one event.

The paper also discusses invalid transitions, with formal properties for valid transitions, and the use of priority when there are simultaneous events.

This paper describes an unusual combination of *model checking* and *implementation testing*. A temporal logic specification is made of the system, defining safety properties. From this, finite state machines (FSMs) that accept input-output traces that violate the safety properties are automatically generated. From the FSMs, test inputs are generated, and the IUT is checked for whether the safety properties are violated by these inputs, and if so, an alert is given. The specification may be nondeterministic, but this is not elaborated on. Examples given: an elevator system and a telephone switching system.

This paper describes “algorithms for the generation of test sequences from non-deterministic finite state machines (NFSMs). The test sequences are synchronizing sequences (SS), transferring sequences (TS) and unique input/output (UIO) sequences.” An SS may not exist, but in practice for protocols they usually do. Compared to a (strongly connected) deterministic situation, the following issues arise: A TS does not always exist because it may not always be possible to transfer deterministically to this state. The UIO has to check a set of states, not just one
expected state. The SS and TS can be regarded executing the test and the UIO as getting extra output to verify the result.

[Kwan] Kwan Mei-Ko
Graphic programming Using Odd or Even Points

The paper shows how to generate a postman's route, i.e. a transition tour. The Chinese postman problem is so named in honour of the author.

[Lee 96] David Lee and Mihalis Yannakakis
Principles and Methods of Testing Finite State Machines
Proceedings of the IEEE, Vol. 84, No 8, August, 1996

This paper gives a good overview of testing based on Mealy machines (actions on transitions, not on state exit/entry). The paper states explicitly that it does not cover validation and verification (model checking), which are distinct from testing. Key concepts: distinguishing sequence of events to identify states; unique input/output (UIO) sequence of events to verify some particular state; checking sequence to test for conformance of a black box to its specification. The paper also describes characterization sets (see [Chow]) which distinguish pairs of states, and transition tours (see the Philips report [Raptis 98]). Nondeterminism is mentioned, but the main exposition focuses on deterministic machines.

Requirements Specification for Process Control Systems
IEEE Transactions on Software Engineering, vol. 20, no. 9, Sept 1994

The paper describes how the need for a specification language for safety-critical systems led to [RSML], and describes RMSL semantics. RSML is based on Harel’s statechart notation, with some omissions where the complexity did not warrant them, and some extensions to allow for the requirements needing to be expressed. The application considered is an aircraft collision avoidance system. A simulator for RSML was built by Heimdahl.

[Li] J Jenny Li, Hong Liu, Rudolph E. Seviora
Constructing Automated Protocol Testing Oracles to Accommodate Specification Nondeterminism
Sixth International Conference on Computer Communications and Networks (ICCCN '97), September 22 - 25, 1997, Las Vegas, NV

The paper describes an SDL-based implementation of a nondeterministic test oracle. For local nondeterminism (like a STATECRUNCHER a fork), a construct ALL that supports AND-states is introduced, a counterpart to ANY in the specification. (AND-states are alternative nondeterministic states, not Harel's parallel states of the same
designations). For global nondeterminism (like a STATECRUNCHER race), permutations of signal arrival orders are needed, apparently also handled by the AND-states. The method was trialled with a small protocol serving 60 nodes. Test generation was random testing. The maximum number of 'concurrent' states generated was 1442.

[Lüttgen 00] Gerald Lüttgen, Michael. von der Beeck and Rance Cleaveland
A Compositional Approach to Statecharts Semantics
Presented at FSE (Foundations of Software Engineering) 2000,
San Diego
http://www.cs.virginia.edu/fse8/
Available from Cleaveland

The paper discusses the semantics of a statecharts composed of smaller statecharts. The approach is one of micro-step semantics as in [Harel], on the ticking of a global clock, from which the macro-composition is recovered, (rather than the sequenced approach of UML). It also has the concept of more than one conjoined event, or absence of an event, (e.g. $a \land \neg b$), on a transition.

[Milner] Robin Milner
Communication and Concurrency

This book describes CCS: the Calculus of Communicating Systems, a process algebra (or calculus), for specifying state behaviour in terms of processes and events. Ordinary engagement of two processes (no more than that) is based an event and its complement being possible, giving rise to a possible internal transition $\tau$, (so introducing potential nondeterminism). The ordinary summation operator, (+), specifies alternative behaviours, which may include nondeterministic choices on the same event. Algebraic laws enable rewriting, simplification and comparison of process expressions.

[Murata] Tadao Murata
Petri Nets: Properties, Analysis and Applications
Proceedings of the IEEE, Vol 77, No 4, April, 1989

The paper gives a comprehensive survey of what the title proposes, with 315 references. Petri nets can be used to model deterministic and nondeterministic finite state machines [p.544]. Property checking (of Petri nets themselves rather than state machines) is discussed (e.g. liveness and safety) [p.550, p.555]. Many applications apart from state-machine related ones are discussed. Higher level nets, including coloured Petri nets (for which an implementation now exists, see [Design/CPN]), are described.
[Myers 79] G.J. Myers
The Art of Software Testing
This is an early, but still popular, book on standard software testing techniques. It is strong on cause-effect graphing (in Chapter 4), a major complementary testing technique to state-based testing. A future research area will probably be to combine cause effect graphing and state based modelling, perhaps in connection with parameterized events.

[Offutt] Jeff Offutt
Generating tests from UML specifications
George Mason University, Fairfax VA 22030, USA
This paper describes a tool called UMLTEST, which takes Rational Rose UML specifications of state machines, requiring that they be deterministic, and generates test cases at full predicate and transition pair coverage level. By full predicate, the author means that the guard (or enabling) condition on the transition is exercised according to a boolean expression coverage criterion known as MC/DC (Modified Condition / Decision Coverage), as defined in the USA Department of Defense standard DO178B. In this standard, test cases are generated such that each term in the expression is shown to be capable of independently affecting the value of the whole expression. The tool was empirically evaluated against a cruise control system with seeded faults, all of which were found, which was better than with just transition pair or statement coverage testing.

[Ostroff 89] Jonathan S. Ostroff
Temporal Logic for Real-Time Systems
The book describes ESMs (Extended State Machines), which, unlike statecharts, contain communication channels over which events are executed, Manna-Pnueli temporal logic, RTTL (Real Time Temporal Logic), and a proof system associated with this, PS-RTTL. The perspective is property checking, not testing.

[Petrenko] Alexandre Petrenko, Nina Yevtushenko, Alexandre Lebedev, Anindya Das
Nondeterministic State Machines in Protocol Conformance Testing
Protocol Test Systems VI (C-19), pp. 363-378, 1994
This paper describes test suite generation for NFSMs, introducing the concept of r-distinguishing sequences to distinguish states in an observable NFSM.
This is a popular article explaining *orthogonal arrays*. Suppose a routine needs testing with 4 parameters, (A,B,C, and D), each of which can take 3 values (1,2, and 3). Exhaustive testing would require running $3^4=81$ tests. But suppose we find it adequate that all pairwise parameter value combinations are taken. A table can be found satisfying this with 9 entries of values of the 4 parameters as follows:

<table>
<thead>
<tr>
<th>ABCD</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
</tr>
<tr>
<td>1223</td>
</tr>
<tr>
<td>1332</td>
</tr>
<tr>
<td>2122</td>
</tr>
<tr>
<td>2231</td>
</tr>
<tr>
<td>2313</td>
</tr>
<tr>
<td>3133</td>
</tr>
<tr>
<td>3212</td>
</tr>
<tr>
<td>3321</td>
</tr>
</tbody>
</table>

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

---

[Robinson 00] Harry Robinson

**Intelligent Test Automation**

*Software Testing and Quality Engineering, Sept/Oct 2000, pp. 24-32*

This popular article makes the practical case for model-based testing using four amusing cartoons.

[Robinson www] Harry Robinson

**Model Based Testing Home Page (maintained by)**

http://wwwgeocities.com/model_based_testing

This is a popular website with many articles on model-based testing.

[Sabnani] Krishnan Sabnani and Anton T. Dahbura

**A Protocol Test Generation Procedure**


The context is Mealy FSMs. The paper describes the UIO (unique I/O sequence) method of checking states, so that the target state of all transitions can be checked.
This paper reports on the use of the [VVT-RT] tool to test a safety-critical application: a thermal control unit of the X-ray satellite ABRIXAS. A target system is tested against CSP specifications. All possible execution sequences (presumably of inputs, i.e. events) were executed. The results were to find incomplete parts of specifications and several bugs, including a hardware problem, where EEPROMs did not meet their specification.

A book on CSP, good for learning CSP, that is complementary to [Hoare], which is the authoritative text.

In the context of deterministic Mealy FSMs, the paper presents results for test sequences using a transition tour, validating the target state of each transition with a UIO (Unique I/O sequence), built into the tour, with the refinement that the best UIO is chosen (where there are several options), so as to produce an optimised tour.

“This paper proposes a revised semantic interpretation of UML Statechart Diagrams which ensures, under the specified design rules, that Statecharts may be constructed to have true compositional properties.” The example of an automatic gearbox is given, and the issue of concurrent events at different compositional levels is discussed. We remark that in STATECRUNCHER, the issue of concurrent, interrupting or conflicting events does not arise, as any triggered transition is processed to completion as regards state occupancies, before any associated actions, which will have been collected, are processed from a consistent and stable configuration.
N.J.A. Sloane
A library of orthogonal arrays
For a description of orthogonal arrays, see [Phadke].

Mike Stannett and A.J.H. Simons
Complete Behavioural testing of Object-Oriented Systems using CCS-Augmented X-Machines
Test Report CS-02-04, Dept. of Computer Science, United Kingdom
The paper combines X-Machines and [CCS], generating a new behavioural specification and modelling language, CCS-XM. A form of communicating X-machine, communicating in the CCS sense, not in the shared memory sense, is defined: a Process X-machine (PXM). The analysis of the way PXMs communicate is analogous to STATECRUNCHERS composition mechanism. The paper has:

The STATECRUNCHER analogue is:

Here, we have not made the ack_serv event unique to the specific caller as in the paper (the this keyword). Since this server does not support recursion, the server can only be serving one client at a time, so it is sufficient for ack_serv to be unique to the server; it cannot then be confused with the acknowledgement from any other server serving a different function. In [StCrFunMod], we propose a composition mechanism for recursive state machines, where the returned acknowledgement need not have a unique name at all, and targets its caller by means of scoping operators.

A large list of testing researchers, with web links.
Jan Tretmans

Test Generation with Inputs, Outputs and Repetitive Quiescence

Department of Computer Science, University of Twente

"...A test generation algorithm is given which is proved to produce a sound and exhaustive test suite from a specification, i.e. a test suite that fully characterizes the set of correct implementations". This paper underlies the later [TorX] publications. The approach is the labelled transition system one, not the finite state machine one. Publications by Jan Tretmans are listed/summarised/downloadable as the case may be at: http://fmt.cs.utwente.nl/publications/tretmans.pap.html

Michael von der Beeck

A Comparison of Statechart Variants

Aachen University of Technology, Aachen, Germany

This paper uses a set of distinctive features to make a detailed comparison of 21 statechart variants. These are: [RSML] (Leveson), [Argos] (Maraninchi), and statecharts indicated by developers/designers only (sometimes with collaborators): Harel, Huizing, Pnueli, Hooman, Classen, Maggioli-Schettini, Day, Peron, Keston, von der Beeck. All but one of these statecharts allows for the specification of nondeterminism, but the only description of handling of nondeterminism given is to resolve the potential nondeterminism to a deterministic choice.

If we attempt to characterise STATECRUNCHER by von der Beeck's criteria, we have

- (1) Perfect synchrony: Yes, there is no buffering of events, but when one event fires another, output is generated in particular orderings of on-state-exit actions, on-transition actions, on-state-entry actions etc.
- (2) Self-triggering: No. Two transitions triggered by $\alpha$/fire $\beta$ and $\beta$/fire $\alpha$ will not spontaneously take place - they require a separate generation of an initial $\alpha$ or $\beta$.
- (3) Negated trigger event: No. There is no concept of negated events, or conjunction of events, such as $\alpha\neg\beta$. Events can only be offered sequentially, and triggered transitions are seen as a set of sequences representing interleaving.
- (4) Effect of a transition is contradictory to its cause: Not applicable, because there is no concept of triggering from a negated event. A transition $-\alpha$/fire $\alpha$ is not specifiable.
- (5) Inter-level transition: Yes. Source and (multiple) target states of a transition can all be in at any level in the hierarchy (provided the transition is not illegal).
- (6) State reference: Yes. This is the in(... function.
- (7) Compositional semantics, Self-termination: Yes, inasmuch as a client-server paradigm exists for composition, mirroring formal software component composition. The client and server can be tested independently, and the inter-
component events can be hidden by attaching them to a PCO (point of control and observation) that indicates that they are not external events in compositions. Discussed in [StCrSemCom]. **Self-termination** is supported, but it is not needed as an inter-level work-around.

- **(8) Operational versus denotational semantics:** Denotational, inasmuch as we specify the exact transition algorithm in a computer-independent way, and an abstract-model-independent way.

- **(9) Instantaneous state:** Yes. This is the knock-on effect in a chain of transitions, and states are simultaneously entered and exited, regarding the whole chain of execution as being atomic, and so conceptually instantaneous, to the user.

- **(10) Durability of events:** No. events are discrete, and have no duration.

- **(11) Parallel execution of transitions:** Yes. parallel execution of transitions is supported, but with selectable interleavings. The article regards this feature as being contradictory to (9), but we have explained and qualified our interpretation of these points

- **(12) Transition refinement:** Not applicable, because we support instantaneous states, giving the equivalence of transition sequences.

- **(13) Multiply entered or exited instantaneous state:** Yes. This is the cycling issue, which we regard as advantageous (provided it is bounded), especially in conjunction with nondeterminism, for reasons given in [StCrMain].

- **(14) Infinite sequence of transition executions at an instant in time:** Not prohibited. A useless infinite loop could theoretically be detected, at the cost of execution time resources (performance and memory). We leave it up to the user not to program an infinite loop, as it were, as is the case in a language such as ‘C’.

- **(15) Determinism:** Nondeterminism is well-supported, this being STATECRUNCHER's speciality.

- **(16) Priorities for transition execution:** UML-style specialization priority (i.e. transitions on inner elements of the hierarchy) is currently implemented.

- **(17) Pre-emptive versus non-pre-emptive interrupt:** Not applicable, as it involves simultaneous events, whereas in STATECRUNCHER all user events are offered sequentially.

- **(18) Distinguishing internal from external events:** There is no formal distinction, except that a different PCO (point of control and observation) can be attached to each kind of event. Events that can be generated internally in an IUT are modelled by having them generated as fired events on the preceding transition in the STATECRUNCHER model, using nondeterministic constructs if the internal events only may occur.

- **(19) Time specification, timeout, timed transition:** No time support. Time handling is regarded as a test generator or test driver/harness affair (e.g. when we wish to wait for the SUT to perhaps execute an internal event). STATECRUNCHER
can indicate that this is the situation by providing an event called e.g. \texttt{wait}, which has this special meaning.

- \textit{(Feature items - semantics, when not covered by the above)}
  - True concurrency: No.
  - Discrete/continuous time: Discrete

- \textit{(Feature items - syntax)}
  - Graphical/Textual: Textual.
  - Negated trigger event: No
  - Timeout event: No
  - Timed transition item: No
  - Disjunction of trigger events: No
  - Trigger condition: Yes
  - State reference: Yes
  - Assignment to a variable: Yes
  - Inter-level transition: Yes
  - History mechanism: Yes

Other statechart features that could be included in a comparison are (1-10 supported by \texttt{STATECRUNCHER}): (1) multiple target states, (2) orbital transitions, (3) traces, (4) nondeterministic worlds, (5) scoping operators, (6) points of control and observation, (7) upon enter and upon exit actions, (8) entering and exiting of states as internally generated events, (9) parameterised events, (10) a command language supporting: (i) output of transitionable or all events, (ii) re-instatement of previous worlds (iii) creation of new worlds, (iv) explicit killing of worlds, (v) implicit killing of worlds on trace violations, etc. Some features not currently supported by \texttt{STATECRUNCHER}: (A) lambda transitions (i.e. transitions on data values, not requiring events), (B) recursive state machine implantation.

---

[Zhang] Fan Zhang and To-yat Cheung

\textbf{Optimal Transfer Trees and Distinguishing Trees for Testing Observable Nondeterministic Finite-State Machines}


The approach is the \textit{finite state machine} one, not the \textit{labelled transition system} one. Testing a black box NFSM involves bringing it into a specific state, for which a transfer tree (TT) is required, and then verifying that it is in the correct state by further transitioning, using diagnosis/distinguishing trees (DTs). This paper investigates for \textit{observable} NFSMs (different outputs generated on forks from the same event to different states) how, when weights (or probabilities) are assigned to nondeterministic transitions, TTs can be constructed to have a minimal expected value of weights over all paths, or to have minimal maximum of the weights. A similar problem for a certain kind of DT is also addressed.

---

© Graham G. Thomason 2003-2004
5. Supporting projects / products / information

[Beveridge] Jim Beveridge and Robert Wiener
Multithreading Applications in Win32. The Complete Guide to
Threads
A very good book on threads in Windows 32 systems. An example of using
semaphores to protect against deadlock in the dining philosophers problem is given.
(This problem is also considered by [Hoare], [Schneider] and many other textbooks
on logic and parallelism).

[Boley] Harold Boley
Relationships between Logic Programming and XML
Proceedings of the 14th Workshop Logische Programmierung,
Würzburg, Jan. 2000
The relevance of this paper is that it describes the nearest application of Prolog to a
compiler-related field that we find in recent conferences on applications of Prolog
(see [INAP 2001]), though for an early paper on the subject, see [Warren]. The paper
shows how XML documents might be represented as PROLOG clauses and vice-
versa, covering not just PROLOG facts but relationships with non-ground terms. The
application to XML query languages is discussed, where a response can be that
Prolog structures are nondeterministically enumerated.

[Bratko] Ivan Bratko
PROLOG Programming for Artificial Intelligence
Addison-Wesley, ISBN 0-201-41606-9
This book on PROLOG has an artificial intelligence slant. It is good on advanced tree
structures and searching.

[Callahan] John R. Callahan
http://www.cs.wvu.edu/~callahan/interests.html
Callahan, and also the Nasa Goddard IV&V facility, (http://www.ivv.nasa.gov)
interpret verification and validation in the following contexts:
• Verification: Are we building the product right?
• Validation: Are we building the right product?

These are useful interpretations, corresponding to testing and property checking, but are by no means universally understood this way. Compare [IEEE 610.12.1990] and [CMMI].

[Clocksin 84] W. F. Clocksin & C. S. Mellish
Programming in Prolog
Springer Verlag, 1981. ISBN 3-540-11046-1

This is a standard Prolog book, using Edinburgh syntax. It is very well structured, and it clearly explains all constructs of the language with elementary examples.

[CMMI] CMMI-SE/SW, Version 0.2b, Sept 1999
Capability Maturity Model - Integrated Systems/Software Engineering

CMMI website: http://www.sei.cmu.edu/cmmi/cmmi.html

We seek definitions of validation and verification, and find:
• Validation (v.2, p.109): The purpose of validation is to confirm that a product fulfills its intended use when placed in its intended environment.
• Product Verification (v.2, p.106): The purpose of Product Verification is to assure that work products meet the specified requirements

The distinction between property checking and implementation testing does not appear to be made in these definitions. But see [Callahan] for a useful distinction.

[CYGWIN] www.cygwin.com

CYGWIN is a public-domain Linux-like environment for Windows. It consists of two parts: (1) a DLL (cygwin1.dll) which acts as a Linux emulation layer providing substantial Linux API functionality; (2) A collection of tools, which provide Linux look and feel. CYGWIN provides a platform for the popular test harness [DejaGnu].

[Darnell] Peter A. Darnell and Philip E. Margolis
C: A Software Engineering Approach

The ANSI C railroad syntax diagrams in this standard ‘C’ textbook give the basis of the expression grammar of STATECRUNCHER. In STATECRUNCHER an extension was used, and the left-recursive diagrams were transformed into a non-left recursive feed-forward grammar for parsing as a PROLOG DCG (Definite Clause Grammar), as described in [StCrGP4].
This is an example of a public domain test harness, originally developed for Unix, using [TCL] (Tool Command Language) and [EXPECT]. It spawns a program (or several) and works by sending lines of input to its standard input, and receives standard output. It tests for a pattern match on the standard output or registers a timeout. Pass or Fail is logged per test, typically according to the success or failure of a pattern match. Philips has used it for state-based testing using state relation tables, from which tests are generated using a program written in TCL, effectively sending events and receiving the target states, matching against the tabular oracle. See [Savoye] for the manual.

Expect is a very powerful scripting language, built on [TCL], capable of spawning many processes and communicating with them independently via standard input and standard output. It is the underlying layer of the test harness [DejaGnu]. It is also useful for writing glue code in chains of testing tools, e.g. for converting one format or protocol to another, and is used as such in the integration of STATECRUNCHER into the [TorX] tool chain. The book on the language, written by its creator, is [Libes].

IEEE Standards, Software Engineering
We seek definitions of validation and verification, and find:

- Validation (p.80): The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements.

- Verification (p.81): (1) The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase. (2) Formal proof of program correctness.

The distinction between property checking and implementation testing does not appear to be made in these definitions. But see [Callahan] for a useful distinction.

Don Libes
Exploring Expect
The book by the creator of [EXPECT] describing [TCL] and EXPECT.

The 14th International Conference of Applications of Prolog
We examine the programme of this conference (and some previous years) to see what PROLOG is being used for, and whether it has been used as a compiler for what might be called a *domain specific language*, whether in the testing domain or any other. The session streams at this conference were:

- Supporting Organisational Learning: Knowledge Management and Case-based Reasoning
- Deductive Databases and Knowledge Management
- Web Applications for the Legal Domain
- Logic Programming for Natural Language Processing
- Practical Applications of Controlled Natural Languages
- Optimization and Simulation of Complex Industrial Systems. Extensions and Applications of Constraint-Logic Programming
- Business Opportunities in Advanced Technologies
- Decision Support in Medicine and Health Care
- Rule-Based Data Mining

Invited talks were on *Making decisions with incomplete information* (Donald Nute) and *The Rule Markup Language: RDF-XML Data Model, XML Schema Hierarchy, and XSL Transformations*, (Harold Boley). The latter is perhaps as close to the compiler domain as anything presented. For this subject area, see the related article [Boley]. For an article on the use of PROLOG for compilation, see [Warren].

R. van Ommering, F. van der Linden, J. Kramer, J. Magee  
*The Koala Component model for Consumer Electronics Software*  
*IEEE Computer, March 2000, pp. 78-85.*

Koala is a static-binding component model, used for Philips TV software. The initial trialling of STATECRUNCHER is with Koala components and compositions of them.

Richard O'Keefe  
*The Craft of Prolog*  
*MIT Press. ISBN 0-262-15039-5*  

A good PROLOG book with a particularly good section on the PROLOG 'cut'.
[Ousterhout] TCL and the TCL Toolkit
John K Ousterhout
The above book is by the creator of [TCL] (Tool Command Language). TCL is a
powerful scripting language, underlying [EXPECT] and the [DejaGnu] test harness.

[Savoye] R. Savoye
The DejaGnu Testing Framework
The Free Software Foundation, 1993
This is a manual for the [DejaGnu] public domain test harness.

[Sterling] Leon Sterling & Ehud Shapiro
The Art of Prolog
A good PROLOG book with many detailed examples, and useful guidance on good
PROLOG programming style.

A public domain PROLOG, used in addition to [WinProlog] for developing
STATECRUNCHER.

[Tau] www.telelogic.com
A commercial tool by Telelogic for [TTCN] testing, with support for e.g. TCP/IP,
RS-232 and “almost any target operating system”

[TCL] Tool Command Language
TCL is a powerful scripting language, underlying [EXPECT] and the [DejaGnu] test
harness. It is described in [Ousterhout] and [Libes].

[TTCN] The Tree and Tabular Combined Notation
ISO (the International Organisation for Standardisation) /
IEC (International Electrotechnical Commission) standard 9646-3
A format and methodology for describing conformance tests, designed especially in
connection with telecommunications standards and OSI protocols. Batch-generated
state-based tests can be represented in TTCN. The basic structure is a depth first tree
of alternatives (so supporting nondeterminism). A tutorial is available on the web by
Mazen Malek
http://www.item.ntnu.no/~malek/research/TTCNcourse
This paper showed the feasibility of using PROLOG as an implementation language for compilers at an early date. The principle is illustrated for a ‘toy’ assembler, but the most important techniques are covered, including expression parsing with two operator precedences. The DCG (Definite Clause Grammar) technique is used, but without the more compact notation (the \( \rightarrow \) operator, which hides systematically repeated parameters) which was later introduced into the PROLOG language. Computer memory and speed were restricting factors at the time; Warren considered memory the greater limitation. For STATECRUNCHER, a few megabytes of memory are needed, and speed is perhaps a limitation on PC machines below 300 MHz, corresponding to pre-1998 manufacture.

WinProlog, Logic Programming Associates Ltd
http://www.lpa.co.uk
This is a version of PROLOG which was used for the development of STATECRUNCHER, on a PC (in addition to SWI-Prolog).

WinRunner v4.0/v5.01, Mercury Interactive
http://www.merc-int.com/products/winrunguide.html
A tool for Graphical-User-Interface-based testing of Window products. Philips has an extension, informally known as Deja Gnu-Y-Trewl, [Trew 98], to support state-relation tables. Another Philips tool that is useful in conjunction with WinRunner is GFET [GFET], which gives a graphical user interface to software that otherwise does not have one.
6. 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 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
The Design and Construction of a State Machine System that Handles Nondeterminism

Graham G. Thomason

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

UniS

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

July 2004

© Graham G. Thomason 2003-2004
Summary

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

Thanks are due to my supervisors:

- Timotheo Trew, ars cui summa est, studium doctrinae pudorque. quem magni artifices semper dicunt magistrum. doctior hoc nemo est; potest quem vincere nemo programmata qui noverit probare.


- To Dr. David Pitt, who has the greatest skill, enthusiasm for his subject, and modesty. Great academics always call him the expert. No-one is more scholarly than he; no-one who knows how to formally specify a system can surpass him.

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

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

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

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

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

1. Introduction ...............................................................................................................................1
   1.1 Context of the work ...........................................................................................................1
   1.2 The problem to be considered ..........................................................................................1
   1.3 A peek at the result of the work ......................................................................................2
   1.4 What STATECRUNCHER is not ...................................................................................2
   1.5 The structure of this thesis ............................................................................................4

2. Software testing in context .......................................................................................................5

3. State-based testing and STATECRUNCHER overview .......................................................9
   3.1 States and events ...............................................................................................................9
   3.2 Deterministic state-based testing ...................................................................................12
   3.3 Nondeterministic testing ..............................................................................................25
   3.4 STATECRUNCHER and the TorX tool chain .............................................................26
   3.5 Alternative modelling techniques to state-based modelling ........................................26
   3.6 Summary of this section .................................................................................................28

4. Nondeterminism .....................................................................................................................29
   4.1 Review of nondeterministic testing ...............................................................................29
   4.2 Fork nondeterminism .....................................................................................................30
   4.3 Race nondeterminism .....................................................................................................33
   4.4 Set transit nondeterminism .............................................................................................35
   4.5 Fired-event and multiple nondeterminism .....................................................................36
   4.6 Set-action nondeterminism ............................................................................................38
   4.7 Set-meta-event nondeterminism .....................................................................................39
   4.8 Effects of nondeterminism .............................................................................................41
   4.9 Worlds ............................................................................................................................42
   4.10 Containment of combinatorial explosion .....................................................................44
   4.11 Test generation under nondeterminism .........................................................................63
   4.12 Summary of this section .................................................................................................64

5. STATECRUNCHER as a language ......................................................................................66
   5.1 General syntax .................................................................................................................66
   5.2 STATECRUNCHER statements ....................................................................................68
   5.3 Basic syntax of statechart / cluster / set and (leaf-)states in a hierarchy .......................69
   5.4 More about hierarchical states ........................................................................................70
   5.5 Declarations and scoping ...............................................................................................75
   5.6 Expressions, operators and functions .............................................................................80
   5.7 Review of items parsed as expressions ...........................................................................99
   5.8 Transition block .............................................................................................................101

6. Algorithmic sequencing .........................................................................................................118
   6.1 Cycling ............................................................................................................................119
Appendix 6  [StCrBiblRef]  Bibliography and References

Related reports
Related report 1  [StCrPrimer]  STATECRUNCHER-to-Primer Protocol
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
1. Introduction

1.1 Context of the work

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

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

1.2 The problem to be considered

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

1.3 A peek at the result of the work

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

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

![Figure 1. STATECRUNCHER in a tool chain](image)

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

1.4 What STATECRUNCHER is not

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

**STATECRUNCHER is not a property checker**

We distinguish two kinds of tool: model checkers and simulators/test oracles. The corresponding activities may be called property checking and testing respectively. A software system needs a design and an implementation, and both need a separate kind of tool and activity to ensure the quality of the final system¹.

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

- The design must guarantee certain properties, e.g. safety, liveness, fairness, freedom from deadlock. Given a formal design, such as a statechart with properties attached to states, and a formulation of the properties required in a system, a model checker can attempt to prove them. Two possible limitations are: the expressiveness of the property language (typically a temporal logic), and the size of the state space (though some techniques allow for vast numbers of states).

- Given a design, the system must be implemented, and the implementation tested. Televisions, mobile phones etc. are a combination of hardware and software. The concept of being in a state means much more to a real system than to a simulator: mobile phone transmitters may be switched on, threads may be waiting for semaphores, buffers should have certain content, such as a teletext page. Testing involves making sure that these things that should happen really do happen. The state model tells us what it is that should happen.

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

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

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

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

**STATECRUNCHER is not a test generator**

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

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

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

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

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

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

- **Code checking in general**: Static analysis can reveal bad coding style and possible pitfalls. Dynamic techniques can check for memory leaks and can provide code coverage, such as statement coverage, described in more detail in [StCrContext].

- **Module testing**: The question to be answered is: Does the implementation correspond to the design? Modules are usually single functions, or a small number of tightly coupled functions designed against a single specification. Exercise code statements and branches. Use code instrumentation to check for coverage of these. Also include a memory leak check in the tests. Module testing is typically white-box testing - we have a knowledge of the code structure and use it to guide us in designing test cases, and we have detailed controllability and observability of the module.
• **Integration testing:** The question to be answered is: Is the design internally consistent? Exercise interfaces between modules. Measure call-pair coverage (i.e. every call and every return from it). Integration testing is typically *black-box testing*—some modules may even be only available as object code, and the only way we can test the integrated system is via the published interfaces.

• **System testing:** The question to be answered is: Does the system satisfy the project requirements? This will typically be a *black-box testing* activity, since the requirements do not normally specify internal controllability and observability, but rather the operations and their outputs to which the end-user has access. For some kinds of system, a part of system testing will be *volume testing*. For example, a set-top box will need to be tested with large quantities of MPEG streams, and a Global Positioning System will need to be tested with large quantities of sampled radio-front-end (intermediate frequency) satellite data.

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

**Automated test execution**

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

![Diagram](image)

**Figure 3. Automated test execution**

There are good commercial and public domain test harnesses. A Unix-based public-domain test harness with which Philips Research has considerable experience is *Deja Gnu* [DejaGnu]. A commercial tool for GUI-driven testing under Windows NT with which Philips Research is also familiar is *WinRunner* [WinRun]. A Philips Research tool to give an
(embedded) multi-threaded application so that it can be tested using WinRunner is GFET [GFET].

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

![Diagram: Automated test generation]

- **Model of System under Test**
  - `test script generation` or on-the-fly testing
  - **Test script**
  - **Test Harness**
  - **System Under Test**
  - **Test report**

**Figure 4. Automated test generation**

The kinds of model that are most used for automated test generation are:
- A state behaviour model, or statechart, such as the UML dynamic model
- A cause-effect graph (or a decision table, which is a simple form of cause effect graph)
- A grammar of a language or protocol for syntax testing
- Orthogonal arrays for parameter/property interaction testing.

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

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

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

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

3.1 States and events

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

Under this modelling technique, if a system is “in a particular state”, it will remain so indefinitely until an event occurs. In other words, the notion of a state entails durability - the state exists over a period of time. Even if a system enters a particular state $s_1$ and there is an event ready and waiting to cause a change of state (say to state $s_2$), we still regard the moment when the system is in state $s_1$ as a point at which the system has become stable in terms of its state behaviour. At such a point, the state of the system (in a wide sense) will map to a state in our model of the system.

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

While the system is processing an event, at a modelling level we do not talk about its state, while still recognising that the system will assume ‘states’ at a detailed level which we do not model. At a modelling level we regard processing an event ideally as an infinitely fast and atomic activity, whilst recognising that real-world implementations require time to process events.
If an event would appear to require duration, the situation should probably be modelled by two events \((\text{start } x \text{ and stop } x \text{ events})\) and an intermediate state \((\text{doing } x)\).

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

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

![Figure 5. Specific system-life-cycle – abstract example](image)

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

![Figure 6. Specific system-life-cycle – concrete example: Television](image)

We have a concept of a state as something durable until an event is presented and processed. Systems characterized by this kind of behaviour are called \textit{reactive systems}, since they do
nothing until they react to an event. For a computer system, this suggests that the system is actually idle (as regards machine processing cycles) when it has settled into a state. However, this need not be the case. For example, a multi-threaded application might be modelled with states which represent the fact that low priority threads are running - such a system would still be able to react to events which interrupt at a higher priority. It may even be necessary to represent cpu-bound tasks as states, perhaps using several states so as to model events as having been recorded but unable to be processed until the task completes.

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

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

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

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

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

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

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

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

3.2.1 White box testing

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

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

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

![Figure 7. White-box state-based testing](image)

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

How should the state behaviour be represented? Early work used a state-relation table, or SRT, in which entries in the first columns define initial states, a middle column contains the event, and the latter columns define final states, i.e. states after processing the event. The use of wildcards can help keep the table size reasonable. An SRT has affinities with a decision table. At Philips, a program by David Yule has been used to obtain an oracle to state behaviour this way, to test inter alia a DVD player and a set-top box. As an example (without parallelism), the figure below shows a dynamic model of a smart card reader, followed by part of an SRT representing it.
The state-relation table below represents part of the above model, using the notation “?” for a wildcard, and “#0” for as in the first column.

<table>
<thead>
<tr>
<th>Start State</th>
<th>Event</th>
<th>Result State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disconnected</td>
<td>Connect</td>
<td>No_Card</td>
</tr>
<tr>
<td>Disconnected</td>
<td>?</td>
<td>#0</td>
</tr>
<tr>
<td>?</td>
<td>Disconnect</td>
<td>Disconnected</td>
</tr>
<tr>
<td>No_Card</td>
<td>InsertCard</td>
<td>Resetting</td>
</tr>
<tr>
<td>No_Card</td>
<td>?</td>
<td>#0</td>
</tr>
<tr>
<td>?</td>
<td>RemoveCard</td>
<td>No_Card</td>
</tr>
<tr>
<td>Error</td>
<td>ErrorHandled</td>
<td>Resetting</td>
</tr>
</tbody>
</table>

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

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

### 3.2.2 Statechart systems

A diagram showing states and transitions is called a state-transition diagram. Statecharts extend the basic notion with hierarchical structure, to be described in detail later, but evident in Figure 9, which is a statechart. Such a representation provides a compact and intuitive means to express all the relationships between states, events, and new states after processing.
the event. Statecharts were first proposed and used by David Harel [Harel]. We now consider the primitives of a statechart in more detail.

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

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

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

\[ \beta, \delta \]

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

\[ \beta, \delta \quad \text{t22} \]

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

The above diagram models a system as having:
- three states: a, b and c. The initial state is a (symbol \( \bullet \)).
- five events: \( \alpha \), \( \beta \), \( \gamma \), \( \delta \) and \( \varepsilon \).
- four transitions: \( t1 \), \( t2 \), \( t3 \) and \( t4 \).

At any one time, a system modelled by the above state-transition diagram will be in one and only one state. That state is called the occupied (or active) state. The others are vacant (or inactive).
Transitions whose source states are vacant (at the time an event occurs) do not cause any state transitioning to take place – they are inapplicable (or invalid) in the current state.

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

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

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

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

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

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

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

\(^1\) e.g. hierarchical prioritisation, where an inner transition masks an outer one or vice versa
State behaviour modelling is part of the UML (Unified Modelling Language) dynamic view. It is not particularly onerous to prepare STATECRUNCHER models using a text editor. But an alternative way might be to use CASE (Computer Aided Software Engineering) tools to draw the diagram, and use them to export the state machine view in textual form. Utilities could then be written as necessary to convert exported descriptions to STATECRUNCHER code.

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

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

**Clusters**

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

![Cluster and transition target notation](model_t4160)

**Source code:**

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

© Graham G. Thomason 2003-2004
The syntax of STATECRUNCHER is such that target states are by default a sibling of the source state. Non-sibling target states need more precise specification than just their name, giving their scope. Parent scope is specified using the operator "\$". A grandparent scope would be designated by "\$\$". Descent into child states is achieved using the operator ".". Grandchildren would be designated using this operator twice, e.g. cl.d.grch. Note that on event theta a transition will take place from anywhere in cluster cl to member state d.

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

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

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

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

Sets

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

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

```plaintext
statechart sc
    cluster sys [sc] = OCC [ ] **
    leafstate a [sys, sc] = OCC [ ] **
    set b [sys, sc] = VAC [ ]
    cluster b1 [b, sys, sc] = VAC [ ]
    leafstate p [b1, b, sys, sc] = VAC [ ]
    leafstate q [b1, b, sys, sc] = VAC [ ]
    cluster b2 [b, sys, sc] = VAC [ ]
    leafstate r [b2, b, sys, sc] = VAC [ ]
    leafstate s [b2, b, sys, sc] = VAC [ ]
    cluster b3 [b, sys, sc] = VAC [ ]
    leafstate t [b3, b, sys, sc] = VAC [ ]
    leafstate u [b3, b, sys, sc] = VAC [ ]

TRACE = []
TREV [ [theta, [sc]], 0, [ ], [ ] ]
TREV [ [gamma, [sc]], 0, [ ], [ ] ]
TREV [ [beta, [sc]], 0, [ ], [ ] ]
TREV [ [delta, [sc]], 0, [ ], [ ] ]
```

On processing event beta, the output is:

```plaintext
statechart sc
    cluster sys [sc] = OCC [ ] **
    leafstate a [sys, sc] = VAC [ ]
    set b [sys, sc] = OCC [ ] **
    cluster b1 [b, sys, sc] = OCC [ ] **
    leafstate p [b1, b, sys, sc] = VAC [ ]
    leafstate q [b1, b, sys, sc] = OCC [ ] **
    cluster b2 [b, sys, sc] = OCC [ ] **
    leafstate r [b2, b, sys, sc] = OCC [ ] **
    leafstate s [b2, b, sys, sc] = VAC [ ]
    cluster b3 [b, sys, sc] = OCC [ ] **
    leafstate t [b3, b, sys, sc] = OCC [ ] **
    leafstate u [b3, b, sys, sc] = VAC [ ]

TRACE = []
TREV [ [pi, [sc]], 0, [ ], [ ] ]
TREV [ [rho, [sc]], 0, [ ], [ ] ]
TREV [ [tau, [sc]], 0, [ ], [ ] ]
TREV [ [gamma, [sc]], 0, [ ], [ ] ]
TREV [ [theta, [sc]], 0, [ ], [ ] ]
```

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

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

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

```
enter($x.y)->c
```

![Figure 13. Internal event](image)

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

```
α->c {i=i*j+3; }
```

```
\[ n=n*10+1 \]
\[ x=x*10+1 \]
```

![Figure 14. Variable assignment](image)

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

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

```
α[{i+j>4}&&!in($x.y)]->c
```

![Figure 15. Conditional transition](image)
Events can be **parameterized**. The destinations for the parameters are listed in round brackets. A parameter may be used in the condition of the transition triggered by the parameterized event. In this example, care has been taken that there should be no nondeterminism.

![Diagram](image)

**Figure 16. Parameterized event**

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

![Diagram](image)

**Figure 17. Fired events**

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

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

![Diagram](image)

**Figure 18. Conditional action**

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

Transitions can be **internal**. This means that there will be no state change, but any actions on the transition will be executed. In Figure 19, on event $\beta$ the internal transition will take place provided cluster $p$ is in the occupied state.
Transitions can have an *orbital* trajectory. In the figure below, the transition on event $\beta$ causes cluster $p$ to be exited and re-entered, whereas the transition on event $\alpha$ does not. This is reflected in the resultant occupied member of cluster $q$. Orbits can be to any height in the hierarchy, and are specified as event->orbital-state->target-state. In the diagram, the loop in the transition arc emphasizes the orbit.

When a cluster is exited, the member that was occupied is stored as the *historical* state. UML uses pseudo-states to indicate entering clusters either recursively (deep history) or just at the top cluster level. History can be (deep-) cleared. STATECRUNCHER currently marks a cluster with a (deep-) history marker (as in CHSM), indicating how the cluster is to be entered if a transition targets it. A deep-history cluster can be shallow-history-entered by deep-clearing its child history, or default-entered by deep-clearing its own history. UML's pseudo-states may be implemented in the future, where transitions can individually specify whether a (deep-) historical state is to be entered or not. In Figure 21, the transition on $\alpha$ will cause the historical states of cluster $b$ to be entered. Initially this is state $b_1$, but if the last occupied states were $b_2$ and $q$, then these would now be entered. However, event $\epsilon$ clears $b$'s history, and if this has happened since exiting $b$, then the transition on $\alpha$ will target member $b_1$. 

© Graham G. Thomason 2003-2004
3.2.4 Black box testing

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

![Figure 22. Black box testing - compare traces](image)

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

States can be checked in various ways; for deterministic systems, the best-known methods are:

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

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

Although most theoretical articles describe a finite state machine in terms of a machine without hierarchy or parallelism, a concurrent hierarchical statechart can be *flattened* (or *unfolded*), since any configuration of state occupancies, variable values and historical states
can be regarded as a single flattened state. So the theoretical results are fully applicable to statecharts.

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

![Trace on a transition](image)

Figure 23. Trace on a transition

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

3 \text{TRACE} = [44]

### 3.2.5 Points of control and observation

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

### 3.3 Nondeterministic testing

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

![Testing with a nondeterministic oracle](image)

Figure 24. Testing with a nondeterministic oracle
It is a major issue to discuss the ways in which the different worlds can come about. The subject is addressed in section 4, where we meet fork nondeterminism, race nondeterminism and other forms of nondeterminism.

### 3.4 STATECRUNCHER and the TorX tool chain

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

![TorX tool chain diagram](image)

**Figure 25. TorX tool chain**

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

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

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

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

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

Figure 26. A cause-effect graph

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

Syntax testing

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

Orthogonal arrays

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

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

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

3.6 Summary of this section

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

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

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

4.1 Review of nondeterministic testing

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

![Diagram showing comparison between Model and IUT-Implementation Under Test]

**Figure 27. Review of testing with a nondeterministic oracle**

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

Fork nondeterminism occurs where there are several transitions on the same event from the same source state. The figure below illustrates fork nondeterminism on events β, γ and δ.

![Diagram of fork nondeterminism](model u5420)

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

After processing event β, the configuration as given by STATECRUNCHER is as follows.

```plaintext
statechart sc
  cluster m [sc] = OCC [] **
  leafstate a [m, sc] = VAC []
  leafstate b1 [m, sc] = VAC []
  leafstate b2 [m, sc] = OCC [] **
  leafstate c1 [m, sc] = VAC []
  leafstate c2 [m, sc] = VAC []
  leafstate c3 [m, sc] = VAC []
  leafstate d2 [m, sc] = VAC []
  leafstate d3 [m, sc] = VAC []
  leafstate d4 [m, sc] = VAC []

VAR INTEGER v [sc] =0
TRACE =[]
TREV [[gamma, [sc]], 0, [], []]
```
STATECRUNCHER has produced 2 worlds. Space does not permit us to reproduce the output on processing events $\gamma$ and $\delta$.

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

An issue in fork nondeterminism

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

![Diagram of fork nondeterminism with same source state]

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

But how is the following situation to be handled? The transitions are named $t_1$ and $t_2$.

![Diagram of hierarchical issue]

**Figure 30. Hierarchical issue**

There are three ways this could be handled:

(1) We could say it is fork nondeterminism, with one world ending up in state $m$. $b_1$ and the other in state $b_2$. 

© Graham G. Thomason 2003-2004
(2) We could say that we prioritize and override by specialisation, saying that \( t_1 \) takes precedence, because its source state is deeper in the hierarchy, and it masks \( t_2 \). In this case, the model is deterministic. This is the approach taken by UML, and is in line with overriding member methods in C++ derived classes.

(3) We could say that we prioritize and override by the more external transition, saying that \( t_2 \) takes precedence and masks \( t_1 \). In this case, the model is again deterministic. This approach has the advantage that an external transition cannot be affected by (perhaps poorly understood) internals of a deeply embedded machine. This is the approach taken by [CHSM].

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

![Figure 31. Forced prioritisation reversal giving specialisation](image)

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

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

![Figure 32. Forks in a hierarchy with conditional transitions](image)

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

STATECRUNCHER proceeds as follows:
- Under an event, collect all possible transitions on it in the entire statechart hierarchy
- Evaluate all their conditions
• Find all innermost layers of the hierarchy that have at least one transition attached with a satisfied condition
• Take all satisfied transitions from these layers.

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

![Diagram](https://via.placeholder.com/150)

**Figure 33. Equivalent for hierarchical nondeterminism**

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

### 4.3 Race nondeterminism

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

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

![Diagram](https://via.placeholder.com/150)

**Figure 34. Race - winner determined by variable value**
Race nondeterminism is a convenient way of expressing what would be fork nondeterminism in a flattened (or unfolded) model. The above model is equivalent to the following one:

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

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

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

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

Consider the following model:

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

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

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

22 VAR STRING u [sc] = [49, ...] =1234567890
23 VAR STRING u [sc] = [51, ...] =3412569078
If we transition back to set a with event gamma, say, then variable v will track another 4 orderings. And these will be done in the 4 existing worlds. That will produce 16 worlds. The last lines of output are:

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>157</td>
<td>var string u [sc] = [49, ...] =1234569078</td>
</tr>
<tr>
<td>157</td>
<td>var string v [sc] = [51, ...] =3412567890</td>
</tr>
<tr>
<td>157</td>
<td>trace = []</td>
</tr>
<tr>
<td>157</td>
<td>trev [[omega, [sc]], 0, [], []]</td>
</tr>
<tr>
<td>157</td>
<td>trev [[beta, [sc]], 0, [], []]</td>
</tr>
<tr>
<td>157</td>
<td>trev [[alpha, [sc]], 0, [], []]</td>
</tr>
</tbody>
</table>

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

The order of transit in this last world was:

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

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

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

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

### 4.5 Fired-event and multiple nondeterminism

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

The following figure shows a model exhibiting fork, race, set-transit and fired-event nondeterminism in concert. The action of the transition on event alpha is to fire event beta. Event beta triggers three transitions, which are explicitly named t1, t2 and t3. These give rise to a fork and race. The set of sequences produced is: \{<t1,t2>, <t2,t1>, <t1,t3>, <t3,t1>\}. Transition t1 gives rise to set-transit nondeterminism on entering set b2. In one set of worlds states p and p1 will be entered before states q and q1, and in another set of worlds this will be the other way around. The net result of processing event alpha is therefore to generate 8
worlds. The order in which transitions and set-member entry is done is recorded in the variable \( v \), since each assignment to this variable adds a unique digit to the end of the current value.

An example world generated on event \( \alpha \) is:

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

Figure 38. Four kinds of nondeterminism in concert [model \( t5480 \)]
The value of $v$ (=612435) shows that transition $t_2$ was chosen from the fork of $t_2$ and $t_3$, and that it was executed before $t_1$ in the race. This is corroborated by the occupancies of $c_2$ and $z_2$. The value of $v$ shows that order of entering set $b_2$ and its members is: $b_2, p, p_1, q, q_1$. The other seven worlds have values of $v$ of 613524, 135246, 124356, 712435, 713524, 135247, and 124357, with the corresponding state occupancies of $c_2$, $c_3$, $z_2$, and $z_3$.

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

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

Separate control of race and set permuting is offered.

### 4.6 Set-action nondeterminism

Processing a single transition may lead to actions taking place in several set members, even though no set member may be entered or exited. This could be seen as a special case of set-transit nondeterminism, but we consider it separately. The actions will be hierarchically grouped (or bracketed) and permuted as for set-transit nondeterminism. The example below contains a set of sets, and suffers from the beginnings of poor performance due to the many permutations involved. For that reason, part of the model has been commented out.
Figure 39. Set action nondeterminism [model t5412]

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

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

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

4.7 Set-meta-event nondeterminism

This is similar to set-action nondeterminism. In our example below, we have a set containing a set containing two more sets, and we are not surprised to see poor performance, which is why part of the model has been commented out. Processing a single transition may lead to broadcast meta-events taking place relating to several set members, even though no set
member may be entered or exited. The meta-events will be hierarchically grouped (or bracketed) and permuted as for set-transit nondeterminism. Example:

![Diagram](image)

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

After event $\alpha$, any of events $\omega, \omega_x$, or $\omega_{race}$ will cause exiting of states, generating exit meta-events, triggering transitions in cluster $z$. Note that the transitions on the meta events respond from any state in cluster $z$, not just neutral. So all exit meta-events under consideration are recorded, in order. Events $\omega$ and $\omega_x$ cause hierarchically grouped orderings as with set action nondeterminism, producing in this case $3! \times 11! \times 2! = 12$ orderings. Event $\omega_{race}$ will generate 12 worlds by a different mechanism: the transitions on this event are sequenced in two orderings by race nondeterminism, and one of the transitions produces 6 orderings by set nondeterminism. As it happens, $\omega_{race}$ is faster to process than
If after event $\omega_x$, we again process event $\alpha$, a similar reset to the initial states in sets $a$ and $b$ occurs, but now all transitions race each other, and $4!=24$ worlds are produced (when all permutations are enabled, i.e. under high race), or under medium race, 8 worlds.

4.8 Effects of nondeterminism

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

4.8.1 State-occupancy nondeterminism

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

![Figure 41. State occupancy nondeterminism](image)

4.8.2 Variable-value nondeterminism

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

![Figure 42. Variable-value nondeterminism](image)

4.8.3 Trace-value nondeterminism

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

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

4.9 Worlds

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

The forks are emphasized as usual by the double ellipses. The first fork is on event β, where the fork leads to two different target states. Then on event γ there is another fork, but with
two transitions from different source states (b1 and b2) converging on the same target state. A duplicate world will be discarded, and there will be 3 resultant worlds. On event δ, two worlds do not respond (those in states c1 and c3); these will be left intact. Departing from the world where c2 is occupied, there are 5 transitions, but they only lead to 4 new worlds, because two transitions lead to an identical world. In all, there are 6 worlds after event delta. The model can effectively be reset by event alpha, which will be processed in all worlds, but will take them to the same configuration, and duplicates will be removed, leaving one world.

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

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

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

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

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

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

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

- **Representation explosion.** The representation of the state space may require explicitly defining a large number of states. The use of statechart hierarchy and parallelism often mitigates this problem. If that is not the case, such a situation would suggest that the application being modelled is intrinsically complex or extensive.

- **Effective state space explosion.** Although there may be a compact representation of a model, using statechart facilities, there may still be a vast number of distinct effective states in the model. These would be explicit in a flattened (or unfolded) model.

- **Coverage explosion.** The testing technique may require visiting a large number of states, or executing a vast number of transitions, in order to achieve certain coverage requirements.

- **World explosion.** The number of nondeterministic worlds may become large.

The first of these, representation explosion, appears to be an application-specific issue. We briefly consider effective state space explosion and coverage explosion. The fourth level,
world explosion is very pertinent to STATECRUNCHER and we will describe in some detail the ways in which the design of STATECRUNCHER addresses the issue.

4.10.1 Effective state space explosion

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

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

4.10.2 Coverage explosion

Some test coverage criteria are:
- Reach every state of the flattened machine
- Take every transition arc in a flattened machine

Even though a state space is large, it may be acceptable to traverse it in a limited way. A possibility is:
- Use Projected State Machine Coverage [Friedman, Farchi]. In this technique, states are grouped into equivalence classes. Each equivalence class is a single state in the projected machine.

Specific forms of projected state machine coverage would be to:
- Reach and vacate every leaf-state of the hierarchical statechart, i.e. to ensure that every leafstate is occupied in some test, and that every leafstate is vacant in some test, whilst remaining indifferent to which leafstates are occupied in combination with which other ones (and to variable/history values).
- Take every transition arc in the hierarchical statechart, but again showing indifference to the circumstances (occupancy of states in parallel parts of the machine etc.).
- To regard all leafstates in clusters as equivalent.
More research is needed to ascertain whether these forms of limited coverage are useful in practical testing. Useful ones will find (almost) as many faults as would have been found with more exhaustive testing.

4.10.3 World explosion

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

World explosion arises from large number of worlds that can be generated, as follows:
- There will be $f$ worlds for a fork with $f$ prongs
- There will be $r!$ worlds for a race between $r$ transitions
- There will be $s!$ worlds for a set operation involving $s$ set members.

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

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

We explain these more detail below.

4.10.3.1 The granularity of set-transit nondeterminism

The number of worlds STATECRUNCHER generates on set-transit nondeterminism has been kept within reasonable bounds by avoiding excessive orderings of transition steps. This is illustrated using the figure below:
On transition $\alpha$, two interleavings of the on-entry actions are set up (assume $v=0$):
- enter $c$, enter $i$, enter $i2$, enter $j$, enter $j2$. Variable $v$ will be set to 12435.
- enter $c$, enter $j$, enter $j2$, enter $i$, enter $i2$. Variable $v$ will be set to 13524.

We do not generate the following interleavings (or any others):
- enter $c$, enter $i$, enter $j$, enter $i2$, enter $j2$. Variable $v$ would be set to 12345.
- enter $c$, enter $i$, enter $j$, enter $j2$, enter $i2$. Variable $v$ would be set to 12354.
- enter $c$, enter $j$, enter $i$, enter $i2$, enter $j2$. Variable $v$ would be set to 13245.
- enter $c$, enter $j$, enter $i$, enter $j2$, enter $i2$. Variable $v$ would be set to 13254.

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

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

When a transition enters several sets, the permutations take place at each hierarchical level, illustrated from Figure 48 below.
The outer set members are A (with members a1 and a2) and B (with members b1, b2 and b3). The following permutations will be generated:

- within S: <A,B> and <B,A>
- within A: <a1,a2> and <a2,a1>
- within B: <b1,b2,b3>, <b2,b3,b1>, <b3,b1,b2>, <b3,b2,b1>, <b2,b1,b3>, <b1,b3,b2>

The net orderings on leafstate entry are therefore:

- <a1,a2,b1,b2,b3>, <a1,a2,b2,b3,b1>, <a1,a2,b3,b1,b2>, <a1,a2,b3,b2,b1>,
  <a1,a2,b2,b1,b3>, <a1,a2,b1,b3,b2>
- <a2,a1,b1,b2,b3>, <a2,a1,b2,b3,b1>, <a2,a1,b3,b1,b2>, <a2,a1,b3,b2,b1>,
  <a2,a1,b2,b1,b3>, <a2,a1,b1,b3,b2>
- <b1,b2,b3,a1,a2>, <b2,b3,b1,a1,a2>, <b3,b1,b2,a1,a2>, <b3,b2,b1,a1,a2>,
  <b2,b1,b3,a1,a2>, <b1,b3,b2,a1,a2>
- <b1,b2,b3,a2,a1>, <b2,b3,b1,a2,a1>, <b3,b1,b2,a2,a1>, <b3,b2,b1,a2,a1>,
  <b2,b1,b3,a2,a1>, <b1,b3,b2,a2,a1>

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

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

4.10.3.2 Limited permutation generation

Race condition nondeterminism and set-transit nondeterminism require, in principle, the generation of all permutations of a set of transitions. Different orderings of transitions can
lead to different resultant states or different values of variables. A sequence of assignments, (each of which could be attached to separate transitions) such as
\[ v = v \times 10 + 1; \ v = v \times 10 + 2; \ v = v \times 10 + 3; \]
gives a different result for each order of execution of the three assignments.

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

<table>
<thead>
<tr>
<th>( n )</th>
<th>( n! )</th>
<th>( (n!)^2 )</th>
<th>( (n!)^3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>24</td>
<td>576</td>
<td>13,824</td>
</tr>
<tr>
<td>5</td>
<td>120</td>
<td>14,400</td>
<td>1,728,000</td>
</tr>
<tr>
<td>7</td>
<td>5,040</td>
<td>25,401,600</td>
<td>1.2802 x 10^{11}</td>
</tr>
<tr>
<td>10</td>
<td>3,628,800</td>
<td>1.3168 x 10^{13}</td>
<td>4.7784 x 10^{15}</td>
</tr>
</tbody>
</table>

Table 2. Factorial growth

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

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

\(<a,b,c,d>, <a,d,c,b>, <b,d,c,a>, <c,b,a,d>, <c,d,a,b>, <d,b,a,c>\)

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

**The \([n,k]\) problem**

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

\[1\] This set was found by Alistair Willis.
What we have shown above is a solution to the problem of selecting a subset \( G \) of the permutation of a set of 4 elements, such that all permutations of 3 elements of the set retain their relative ordering in some element of \( G \). We call this a solution to the \([4,3]\) problem.

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

Some terminology and context

Sets

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

Power set

We denote the power set of a set \( S \) by \( \mathcal{P}(S) \).

Sequences

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

\[ Q = \langle a, b, c, d \rangle \]. Its head is ‘a’ and its tail is \( \langle b, c, d \rangle \).

Precedence

For a sequence

\[ A = \langle a_i \rangle_{i=1}^k \ (k \geq 2) = \langle a_1, a_2, a_3, \ldots, a_k \rangle \]

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

Embedding

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

To be more precise, for any sequences \( A \) and \( B \)

\[ A = \langle a_i \rangle_{i=1}^k = \langle a_1, a_2, a_3, \ldots, a_k \rangle \]

\[ B = \langle b_i \rangle_{i=1}^n = \langle b_1, b_2, b_3, \ldots, b_n \rangle \]

Sequence \( \langle a_i \rangle \) embeds into \( \langle b_i \rangle \) if

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

\[ \forall \ r \in [1..n] \cdot a_r = b_{f(r)} \]

Clearly \( |A| \leq |B| \); one sequence cannot embed into a smaller sequence.
We use the notation $A \text{ embeds in } B$ to denote that sequence $A$ embeds into sequence $B$.

**Example of embedding**

\[ \langle b,c,e \rangle \text{ embeds in } \langle a,b,c,d,e,f \rangle \]

**The permutation function $\text{Perm}$**

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

**Example of the permutation function**

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

**Useful subsets of a permutation set**

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

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

\[ S^k = \{ s \in \mathcal{P}(S) \mid |s| = k \} \]

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

The embedding property that must hold is:

\[ \forall s : S^k \bullet ( \forall p : \text{Perm}(s) \bullet ( \exists g : G \bullet p \text{ embeds in } g ) ) \]

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

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

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

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

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

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

\[ S = \{a, b, c, d\} \]

\[ \text{P= Perm}(S) = \{ \]
\[ \langle a,b,c,d \rangle, \langle a,b,d,c \rangle, \langle a,c,b,d \rangle, \langle a,c,d,b \rangle, \langle a,d,b,c \rangle, \langle a,d,c,b \rangle, \]
\[ \langle b,a,c,d \rangle, \langle b,a,d,c \rangle, \langle b,c,a,d \rangle, \langle b,c,d,a \rangle, \langle b,d,a,c \rangle, \langle b,d,c,a \rangle, \]
\[ \langle c,a,b,d \rangle, \langle c,a,d,b \rangle, \langle c,b,a,d \rangle, \langle c,b,d,a \rangle, \langle c,d,a,b \rangle, \langle c,d,b,a \rangle, \]

© Graham G. Thomason 2003-2004
Subset $G$ consists of the bold sequences above: $G = \{G_1, G_2, G_3, G_4, G_5, G_6\}$, where

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

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

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

The individual elements of $S^3$ are:

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

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

$S^3_{1,1} = \langle a, b, c \rangle \text{ embeds in } G_1$  $S^3_{1,2} = \langle a, c, b \rangle \text{ embeds in } G_2$
$S^3_{1,3} = \langle b, a, c \rangle \text{ embeds in } G_3$  $S^3_{1,4} = \langle b, c, a \rangle \text{ embeds in } G_4$
$S^3_{1,5} = \langle c, b, a \rangle \text{ embeds in } G_5$  $S^3_{1,6} = \langle c, a, b \rangle \text{ embeds in } G_6$

$S^3_{2,1} = \langle a, b, d \rangle \text{ embeds in } G_1$  $S^3_{2,2} = \langle a, d, b \rangle \text{ embeds in } G_2$
$S^3_{2,3} = \langle b, a, d \rangle \text{ embeds in } G_3$  $S^3_{2,4} = \langle b, d, a \rangle \text{ embeds in } G_4$
$S^3_{2,5} = \langle c, a, d \rangle \text{ embeds in } G_5$  $S^3_{2,6} = \langle c, b, a \rangle \text{ embeds in } G_6$

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

**Optimal solution to the $[n,2]$ problem**

Given a set $S$

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

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

$G = \{G_1, G_2\} = \{\langle s_1, s_2, s_3, \ldots, s_n \rangle, \langle s_n, s_{n-1}, s_{n-2}, \ldots, s_1 \rangle\}$
Theorem

For any set $S$, $|S| \geq 2$, and set $G$ as defined above:

$$\forall s : S^2 \bullet \left[ \forall p : \text{Perm}(s) \bullet \left( \exists g : G \bullet \text{embeds in} g \right) \right]$$

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

Proof

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

$$S_{uv} = \langle s_p, s_q \rangle$$

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

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

□

Example:

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

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

$S_{uv} = \langle d, b \rangle$, which embeds into $G_2$.

Sub-optimal solution to the $[n,3]$ problem ($n \geq 3$)

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

Given a set $S$

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

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

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

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

$$\langle s_i^i \rangle_{i=n+1}^{2n+1} \text{ when } i > n+1, i \leq 2n \text{ (anticyclic)}$$

Examples of $G$

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

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

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

$G_5 = \langle s_5, s_1, s_2, s_3, s_4 \rangle$  $G_6 = \langle s_5, s_4, s_3, s_2, s_1 \rangle$

$G_7 = \langle s_4, s_3, s_2, s_1, s_5 \rangle$  $G_8 = \langle s_3, s_2, s_1, s_5, s_4 \rangle$

$G_9 = \langle s_2, s_1, s_5, s_4, s_3 \rangle$  $G_{10} = \langle s_1, s_5, s_4, s_3, s_2 \rangle$
S={a,b,c,d,e}

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

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

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

**Theorem**

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

\[ \forall s : S^3 \bullet [ \forall p : \text{Perm}(s) \bullet ( \exists g : G \text{ embeds in } g ) ] \]

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

**Proof**

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

\[ S^3_{uv}=\langle s_p,s_q,s_r \rangle \]

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

- if \( p=n \), then \( p+1=1 \)
- if \( p=1 \), then \( p-1=n \)

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

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

\[ \langle s_p,s_{p+1},...,s_{p+1} \rangle \] (prior to explicit modulo adjustment)

Three cases come into view after making modulo adjustments:

- \( G_{c1}: \langle s_p,s_{p+1},...,s_{m+1},...,s_{p+1} \rangle \) (\( p \neq n, p \neq 1 \))
- \( G_{c2}: \langle s_m,s_{1},...,s_{n-1} \rangle \) (\( p=n \))
- \( G_{c3}: \langle s_1,s_2,...,s_n \rangle \) (\( p=1 \))

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

\[ \langle s_p,s_{p-1},...,s_{p-1} \rangle \] (prior to explicit modulo adjustment)

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

- \( G_{a1}: \langle s_p,s_{p-1},...,s_{1},s_{m+1},...,s_{p+1} \rangle \) (\( p \neq n, p \neq 1 \))
- \( G_{a2}: \langle s_m,s_{n-1},...,s_{1} \rangle \) (\( p=n \))
- \( G_{a3}: \langle s_{1},s_{2},...,s_{n} \rangle \) (\( p=1 \))

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

- **Case 1:** \( p < q, q < r, r > p \). \( <s_p,s_q,s_r> \) embeds into the cyclic case \( G_{c3} \)
- **Case 2:** \( p < q, q > r, r < p \). \( <s_p,s_q,s_r> \) embeds into the cyclic case \( G_{c1} \)
- **Case 3:** \( p > q, q < r, r < p \). \( <s_p,s_q,s_r> \) embeds into the cyclic case \( G_{c1} (p \neq n) \) or \( G_{c2} (p=n) \)
- **Case 4:** \( p > q, q > r, r < p \). \( <s_p,s_q,s_r> \) embeds into the anticyclic case \( G_{a2} \)

© Graham G. Thomason 2003-2004
Case 5: \( p > q, q < r, r > p \)  
\(<s_p, s_q, s_r> \) embeds into the anticyclic case \( G_{a_1} \)

Case 6: \( p < q, q > r, r > p \)  
\(<s_p, s_q, s_r> \) embeds into the anticyclic case \( G_{a_1} (p=1) \) or \( G_{a_3} (p=1) \)

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

Case 1: \( p < q, q < r, r > p \). \( p=10, q=20, r=30 \)
Case 2: \( p < q, q > r, r < p \). \( p=20, q=30, r=10 \)
Case 3: \( p > q, q < r, r < p \). \( p=30, q=10, r=20 \)
Case 4: \( p > q, q > r, r < p \). \( p=30, q=20, r=10 \)
Case 5: \( p > q, q < r, r > p \). \( p=20, q=10, r=30 \)
Case 6: \( p < q, q > r, r > p \). \( p=10, q=30, r=20 \)

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

**Application in STATECRUNCHER**

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

For control of **race nondeterminism**:

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

Table 3. **Control of race nondeterminism**

For control of **set nondeterminism**:

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

Table 4. **Control of set nondeterminism**
Set nondeterminism consists of set-transit nondeterminism, set-action nondeterminism and set-meta-event nondeterminism; they are all controlled by the same setting (the suffix _tran is a little misleading in this respect).

4.10.3.3 Pruning worlds based on traces

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

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

![Figure 49. Reduction of task processing](image-url)
### Implementation in STATECRUNCHER Release 1.05

<table>
<thead>
<tr>
<th>Abbrev. Command</th>
<th>Command showing typical example and/or typical output</th>
</tr>
</thead>
</table>

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

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

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

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

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

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

Which of these should be permitted?

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

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

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

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

STATECRUNCHER currently applies a very lenient strategy of allowing everything except a flagrant trace violation. This can be changed if required.
Test models t5550, t5555, t5560, t5565 (q.v. in [StCrTest]) can be used for experimentation.

**Application in test strategies**

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

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

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

### 4.10.4 The notification example - and containment approaches

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

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

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

```
<table>
<thead>
<tr>
<th>client function</th>
<th>thread starts</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>initial call</td>
</tr>
<tr>
<td>progress notification</td>
<td></td>
</tr>
<tr>
<td>progress notification</td>
<td></td>
</tr>
<tr>
<td>completion notification</td>
<td>server function</td>
</tr>
</tbody>
</table>
```

**Figure 50. Notifications**
A problem involving notifications is a good example of parallelism, two threads being active in parallel. It has the added difficulty that notifications are events that are generated by the system under test, rather than being events that are offered to the system under test. The result is that the system behaves nondeterministically - the number of notification events that will be generated is not known a priori by the state model. This is not a problem until the potential number of notifications becomes large, which is exactly what happens in the example we investigate: a TV program installation example. Program installation for one channel is a process of searching for a station with the tuner, reporting with notifications that that the search is in process. If a station is found, it will be registered. If no station can be found, the TV remains untuned. The program installation process can be stopped at any time.

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

![Diagram](https://example.com/diagram.png)

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

On the `start_tuning` event, the TV searches for a station by tuning. During the search, notifications are generated, representing “search in progress”. These notifications can be used to fill a progress bar. From the above model STATECRUNCHER generates worlds containing various numbers of notifications. When a station is found, the `station_found` event is generated. A fuller model would allow for stopping the program installation, and for failure to find a station.
In a composite system of program installation and tuner as above, the start_tuning event is under the tester's control, but thenotif and station_found events are generated internally to the IUT (Implementation Under Test), ultimately by hardware. The problem arises that a large number of notifications could be generated. The above model caters for up to 4 notification messages by using fork nondeterminism on event gen_notifs to generate:

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

The STATECRUNCHER traces corresponding to this are:

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

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

The following improvements in efficiency are possible:

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

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

A third technique is:

- Pruning worlds based on traces (section 4.10.3.3).

Repetitions

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

\[
\text{TRACE} = [\text{notif_msg*}] 
\]

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

**Conversion of IUT traces to model events**

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

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

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

© Graham G. Thomason 2003-2004
Figure 53. Conversion of traces to events

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

4.10.5 Summary of containment techniques

The following summarises the ways described for containing combinatorial explosion.

**Compact representation of a large number of states and transitions**
- The use of hierarchy and concurrency: STATECRUNCHER's clusters and sets
- Binary decision diagrams are efficient, and are used in SPIN.

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

**Limited state machine coverage in testing**
- Projection coverage.

**Nondeterministic restriction of world explosion**
- Fork nondeterminism: not controllable except by excision of forks in model source code
- Race nondeterminism: A race with \( n \) competitors can be set to
  - no race (1 interleaving)
  - low race (2 interleavings)
  - medium race (2\( n \) interleavings)
  - high race (\( n! \) interleavings)
• Set nondeterminism. Where there are $n$ set member operations, the nondeterminism can be set to
  - no set tran ($I$ interleaving)
  - low set tran ($2n$ interleavings)
  - medium set tran ($2n$ interleavings)
  - high set tran ($n!$ interleavings)
Also
  - The transition semantics avoid micro-orderings of set entry/exit
  - The hierarchical permutation technique, applied to nested sets, reduces the number of interleavings.

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

**Handling notifications**
• Allow for repetitions of a notification in one pseudo-trace
• Conversion of traces to pseudo-events

### 4.11 Test generation under nondeterminism

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

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

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

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

© Graham G. Thomason 2003-2004
Tretmans, in a presentation on Côte de Resyste [CdR], (where inputs and outputs are both events, and traces are sequences of processed events, as in CSP) defines conformance of an implementation \(i\) to a specification \(S\) as:

\[
i \ioco S \iff \forall \sigma \in \text{straces}(S) : \text{out}(i \text{ after } \sigma) \subseteq \text{out}(S \text{ after } \sigma)
\]

Tretmans explains this as: \(i\) \ioco-conforms to \(S\) iff

- if \(i\) produces output \(X\) after trace \(\sigma\), then \(S\) can produce \(X\) after \(\sigma\)
- if \(i\) cannot produce any output after trace \(\sigma\), then \(S\) cannot produce any output after \(\sigma\), (quiescence).

A test suite \(T\) is sound if \(i \ioco S \Rightarrow i\) passes \(T\).

A test suite \(T\) is exhaustive if \(i\) passes \(T \Rightarrow i \ioco S\).

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

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

### 4.12 Summary of this section

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

STATECRUNCHER may be able to play a role in adaptive, on-the-fly testing, but this is a subject for further consideration and research. STATECRUNCHER can certainly flatten UML-style state spaces, and may be useful as a test oracle in adaptive testing too. For example, if after a test STATECRUNCHER has been left with more than one world (all with the same trace, but differing in internal state), and if it is known that the implementation is deterministic, then there may be very efficient disambiguating sequences of events (\(d\)-distinguishing sequences, [Hierons 98]) which could be applied to the IUT and STATECRUNCHER, after
which STATECRUNCHER would be pruned to the matching world only. However, this does not prune the underlying model, only the data it has produced. A future very advanced possibility would be for STATECRUNCHER to allow for adaptation of its model, whereby states and transitions can be created and destroyed.

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

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

5.1 General syntax

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

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

5.1.1 General syntax conventions

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

1. Statements currently must be written on a line of their own, and only on one line, except that a continuation character, the backslash, ",", may be used at the end of a line to denote continuation onto the next line. Use of the backslash may be repeated over many lines. Avoid having anything (e.g. spaces, comments) following the continuation character on the same line; it must be the last character of the line.

2. STATECRUNCHER syntax is case sensitive throughout. Language keywords must be specified in the correct case. User-defined names (identifiers) must be consistent with respect to case.

3. Identifiers are user-defined names of states, events, variables etc. The rules are:
   - Identifiers must not be a language keyword, transition label or function name.
   - Language keywords are:
Keywords reserved for transition labels are:

<table>
<thead>
<tr>
<th>Cost</th>
<th>Name</th>
<th>Time</th>
<th>Utility</th>
</tr>
</thead>
<tbody>
<tr>
<td>lk_cost</td>
<td>lk_name</td>
<td>lk_time</td>
<td>lk_utility</td>
</tr>
</tbody>
</table>

Function names are:

<table>
<thead>
<tr>
<th>Function</th>
<th>Function</th>
<th>Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs</td>
<td>cast</td>
<td>format</td>
<td>get_nworlds</td>
</tr>
<tr>
<td>high_race</td>
<td>high_set_tran</td>
<td>length</td>
<td>lower_case</td>
</tr>
<tr>
<td>low_race</td>
<td>low_set_tran</td>
<td>maximum</td>
<td>med_race</td>
</tr>
<tr>
<td>med_set_tran</td>
<td>minimum</td>
<td>no_race</td>
<td>no_set_tran</td>
</tr>
<tr>
<td>upper_case</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Identifiers must begin with a letter (upper case or lower case) or an underscore. This is optionally followed by a sequence containing upper or lowercase letters, decimal digits and underscores.

4. Numbers are in accordance with their representation in C. Real numbers are not currently supported in any STATECRUNCHER statement.

Examples of integer constants:

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

Examples of character constants:

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

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

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

<table>
<thead>
<tr>
<th>Character</th>
<th>Character</th>
<th>Character</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>space (32)</td>
<td>alert (7)</td>
<td>backspace (8)</td>
<td>horizontal tab (9)</td>
</tr>
<tr>
<td>line feed (10)</td>
<td>vertical tab (11)</td>
<td>form feed (12)</td>
<td></td>
</tr>
</tbody>
</table>

1 Additionally, the suffixes for long and unsigned or both may be appended, e.g. 1231(long) 123L(long) 123u(unsigned) 123U(unsigned) 123ul(unsigned long) 123UL(unsigned long). However, these do not alter the internal representation.

2 For normal use the white space characters are space and horizontal tab. An embedded backspace does not remove the preceding character. Line feed and/or carriage return may not be possible as embedded characters as they may be absorbed in the line read process. DOS and Unix have different end-of-line conventions. The user need not normally be concerned about this. Some text editors may not allow embedding of some of these characters in a file.
Comments (see below) also count as white space. White space can be omitted where that does not lead to an erroneous tokenization or parse. For example, if there is no white space between the keyword `cluster` and the identifier `volume`, a new identifier `clustervolume` is formed, so white space is required. But after brackets, commas, operators, semicolons etc., no white space is required.

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

5.2 STATECRUNCHER statements

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

![STATECRUNCHER Statements Diagram]

In the sections following, these statements are considered individually.
5.3 Basic syntax of statechart / cluster / set and (leaf-)states in a hierarchy

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

![Diagram of statechart syntax](image)

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

The statenames block contains the names of the member states of the cluster or set. The statements defining these member states must occur immediately after their parent. This gives the entire hierarchy a depth-first structure, as will also be seen in the example that follows.
If there is an error in defining the member states (because the child states announced in a parent state do not actually occur, or do not occur in the right place), this is flagged as a machine path error. The machine path is the hierarchical path from the statechart level down the hierarchy to a state at some place in the hierarchy.

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

The transition block is considered in section 5.8.

### 5.4 More about hierarchical states

#### 5.4.1 Statecharts

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

#### 5.4.2 Clusters

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

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

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

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

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

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

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

- \( \mathbf{N} \): no history (default)
- \( \mathbf{H} \): (shallow) history
- \( \mathbf{D} \): deep history

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

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

- The transition on \( \tau 1 \) causes cluster a to be exited. The transition on \( \tau 2 \) causes it to be re-entered, and as cluster a has a deep history marker, it and all descendants will be assume the previous occupancy (for example, states ab, abc and abc, showing the applicability of history in a cluster without a history marker).

- The transition on \( \sigma 2 \) causes cluster aa to be exited. The transition on \( \sigma 1 \) causes it to be re-entered. The deep history marker in cluster a is not effective, as cluster a is not being re-entered on this transition. Since cluster aa does not have a history marker, the default member state is taken: this is state aaa.

- The transition on \( \sigma 4 \) causes cluster ab to be exited. The transition on \( \sigma 3 \) causes it to be re-entered. The history marker in cluster ab indicates that the historical member is to be entered. Suppose this is abc. Cluster abc is duly entered, followed by its default member: state abc.

Notes:

- History data can be cleared (as an action - described later) using the functions clear(state Expr) and deep_clear(state Expr).

- A set (to be described) cannot be marked with a history marker, but it can be marked with a deep history marker.

- History also impinges on the 'transition course' under more complex circumstances – such as transitions targeting a parent state of the source state – to be described later.

- STATECRUNCHER may be changed in the future to handle UML pseudo-states, where it is the transition, not the cluster, that specifies how history is to be handled. But STATECRUNCHER can simulate these, since all clusters can be marked with deep history, and history can be cleared beforehand when the historical states are not required.
5.4.4 Sets

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

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

![Figure 59. Notation for members of sets](image)

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

![Figure 60. Set with transitions](image)
Transitions can have multiple targets so as to specify which states within set members are entered. They can also effectively have multiple source states, indicating that the transition requires all the source states to be occupied, but this must be modelled in STATECRUNCHER as a transition from one of the source states with a condition attached, testing for occupancy of the others. Conditions are described later.

Figure 62. Transitions with multiple source and multiple target states

Sets and history

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

5.4.5 Example of hierarchical states

In the figure below, default states are marked in bold font. The source code is shown alongside.
5.5 Declarations and scoping

STATECRUNCHER supports the following declared items

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

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

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

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

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

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

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

```plaintext
event alpha;
```

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

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

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

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

### 5.5.1 State declarations

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

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

```
Figure 65. PCO declarations

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

```
Figure 66. Event declaration

Examples
event alpha;
event beta,$$gamma,delta@pco1;
event $$epsilon,zeta@$@pco2;
```
Events are not declared with parameters, but, as will be seen, transitions can be labelled with events and their parameters.

5.5.3 Types and variables

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

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

Type declarations and variable declarations are separate statements.

```
type declaration

enum tag-expression enum body ;

enum body

{ integer , . . , integer } { value-name = integer } identifier

tag-expression ::= identifier or scoped name expression

Figure 67. Type declaration
```

Examples of type declarations
```
enum channel {14,...,18};
enum colour {red=6,blue,green=9};
enum $$channels {90,...,99};
enum $$colour {white,red=6,blue,green=9};
```
variable declaration

\[
\text{var-expression} ::= \text{identifier or scoped name expression}
\]

\[
\text{expr ::= scoped arithmetic expression; includes fixed constants true and false}
\]

Examples of variable declarations

\[
\begin{align*}
\text{bool } b1; \\
\text{bool } b1, b2 = \text{true}, b3 = \text{false}, b4 = b2 \&\& \neg b3; \\
\text{bool } $$b1 = \text{false}; \\
\text{channel } \text{fav}_{\text{channel}} = 15, \text{your}_{\text{channel}} = \text{fav}_{\text{channel}} + 2; \\
\text{$channel } $$\text{favourite}_{\text{channel}} = 91; \\
\text{colour } \text{tie}_{\text{col}}, \text{sock}_{\text{col}} = \text{maximum}(\text{red}, \text{green}, \text{blue}); \\
\text{colour } $$\text{tie}_{\text{col}}, $$\text{sock}_{\text{col}} = $$\text{red}; \\
\text{$$colour } $$\text{my}_{\text{tie}_{\text{col}}} = $$\text{colour}_{\text{of the day}};
\end{align*}
\]

5.5.4 PCOs, events and variables in diagrams

Since PCOs, events and variables can also have the same name in different scopes, it may be desirable to show where they are declared. We do that with the ★, ★ and ♂ symbols. In the absence of any symbol, the names can be considered unique and in scope, though it is not specified whether they are global or local.
In the above figure, there are
- PCO declarations in scopes sc and sc.a
- event declarations in scopes sc, sc.a.aa, and sc.a.ab.
- variable declarations in scopes sc, sc.a, sc.a.aa, and sc.a.ab.

The effect of the event declarations is that the δ1, δ2 labels on transitions refer to different events according to the scope of the transition source state. Similarly, there are two PCOs called pco1, which must be distinguished. Similarly again, any expressions using variables (not shown on the diagram) would address the appropriate variable vl.

5.6 Expressions, operators and functions

5.6.1 Expression parsing

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

Expressions in different contexts can be allocated different operator sets, and parsed using the GP4 parser – details of this are given in [StCrGP4]. We give a summary and a flavour of that here, by showing a left-recursive grammar and its transformation into a feed-forward grammar for expressions. The grammar terminals are tokens from a lexical pass performed by GP4, which include constants, identifiers and strings, but not operators, which are identified at expression parsing time. Expressions and terms are parameterized according to their
precedence level, i.e. the level of operator precedence that is being parsed, with higher precedence expressions forming terms at the lower level concerned. A few features that are not pure syntax were introduced:

- Expression grammar rules are parameterized with a precedence level, which is the precedence level of the operators used to combine terms in the grammar rule for the expression at that level.
- Term sequences are also parameterized with an associativity parameter.
- Some small non-grammar operations are performed, indicated by //>. Examples:
  - to left associate, which basically transforms \([a+b-c+d]\) into \([[a+b]-c]+d\]
  - to test for a property, or assign a parameter (such as \(ASSOC=yfx\)).

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

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

The diagrams below are not claimed to be an original exhibition of a general expression grammar, but Figure 71, Figure 72 and Figure 73 were constructed from first principles by the author from the left-recursive grammar of Figure 70, which is a variation of the expression grammar for ‘C’ given in [Darnell]. (A moderate amount of searching and enquiry amongst compiler colleagues failed to come up with anything explicitly similar, apart from the early example of [Warren], though it could be argued that many parsers, though outwardly not similar, effectively implement what is shown here). For that reason, the approach may have some original aspects of some interest to others in a related field.
Figure 70. Left recursive grammar (requiring transformation)
Figure 71. GP4 expressions - feed-forward grammar (1)
The cut, fail combination is reached if the input stream cannot be parsed as expression(N+1).

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

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

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

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

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

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

These operators and functions can be mixed seamlessly in expressions.

Operators have the following attributes:
- A symbol, e.g. +, &
- A name, used internally, which distinguishes between operators of like symbol, e.g. mplus (monadic plus), dplus (dyadic plus).
• A precedence (also called priority). Higher precedence operators bind their arguments before lower precedence ones. Note that this does not mean that they will necessarily be evaluated sooner, although this is sometimes perforce the case. Example \( a-b+c*d+e = a-b+(c*d)+e \), since multiplication has a higher precedence than addition and subtraction.

• A position. This can be
  - **prefix** (as in \(-x\))
  - **postfix** (as in \(i++\))
  - **dyadic infix** (as in \(a+b\))
  - **post-circumfix** (as in the brackets of function call operator, e.g. \(\text{maximum}(a,b)\)).
    Note how these operators come in two parts.
  - **triadic infix** (as in \(a?b:c\))—but this is not currently supported.

• An associativity. This can be
  - left associative: \(a+b+c+d\) is equivalent to \(((a+b)+c)+d\)
  - right associative: \(a=b=c=d\) is equivalent to \(a=(b=(c=d))\)

• An arity. This gives the number of arguments to the operator. It can be
  - monadic: \(-a\)
  - dyadic: \(a+b\)
  - triadic: \(a?b:c\)—but not currently supported.

• Some semantics. The STATECRUNCHER arithmetic and logical operators are commonly known, being mainly compatible with ‘C’.

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

### 5.6.2.1 Arithmetic operators

The following operators are supported:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Symbol</th>
<th>Arity</th>
<th>Precedence</th>
<th>Associativity</th>
<th>Position</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Primary Suffixes</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Array indexing</td>
<td>[ ]</td>
<td>dyadic</td>
<td>18</td>
<td>none</td>
<td>circumfix</td>
</tr>
<tr>
<td>Function call</td>
<td>( )</td>
<td>dyadic</td>
<td>17</td>
<td>none</td>
<td>circumfix</td>
</tr>
<tr>
<td><strong>Various monadic</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>plus</td>
<td>+</td>
<td>monadic</td>
<td>16</td>
<td>right</td>
<td>prefix</td>
</tr>
<tr>
<td>minus</td>
<td>-</td>
<td>monadic</td>
<td>16</td>
<td>right</td>
<td>prefix</td>
</tr>
<tr>
<td>logical not</td>
<td>!</td>
<td>monadic</td>
<td>16</td>
<td>right</td>
<td>prefix</td>
</tr>
<tr>
<td>post increment</td>
<td>++</td>
<td>monadic</td>
<td>16</td>
<td>left</td>
<td>postfix</td>
</tr>
<tr>
<td>post decrement</td>
<td>--</td>
<td>monadic</td>
<td>16</td>
<td>left</td>
<td>postfix</td>
</tr>
<tr>
<td>pre increment</td>
<td>++</td>
<td>monadic</td>
<td>16</td>
<td>left</td>
<td>postfix</td>
</tr>
<tr>
<td>pre decrement</td>
<td>--</td>
<td>monadic</td>
<td>16</td>
<td>left</td>
<td>postfix</td>
</tr>
<tr>
<td>Multiplicative</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------------------------------</td>
<td>----------</td>
<td>---------</td>
<td>-------</td>
<td>-------</td>
<td></td>
</tr>
<tr>
<td>multiplication</td>
<td>*</td>
<td>dyadic</td>
<td>15</td>
<td>left</td>
<td></td>
</tr>
<tr>
<td>division</td>
<td>/</td>
<td>dyadic</td>
<td>15</td>
<td>left</td>
<td></td>
</tr>
<tr>
<td>modulo</td>
<td>%</td>
<td>dyadic</td>
<td>15</td>
<td>left</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Additive</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>addition</td>
<td>+</td>
<td>dyadic</td>
<td>14</td>
<td>left</td>
</tr>
<tr>
<td>subtraction</td>
<td>-</td>
<td>dyadic</td>
<td>14</td>
<td>left</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Relational</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>less than or equal</td>
<td>&lt;=</td>
<td>dyadic</td>
<td>12</td>
<td>left</td>
</tr>
<tr>
<td>greater than or equal</td>
<td>&gt;=</td>
<td>dyadic</td>
<td>12</td>
<td>left</td>
</tr>
<tr>
<td>less than</td>
<td>&lt;</td>
<td>dyadic</td>
<td>12</td>
<td>left</td>
</tr>
<tr>
<td>greater than</td>
<td>&gt;</td>
<td>dyadic</td>
<td>12</td>
<td>left</td>
</tr>
<tr>
<td>equal</td>
<td>==</td>
<td>dyadic</td>
<td>12</td>
<td>left</td>
</tr>
<tr>
<td>not equal</td>
<td>!=</td>
<td>dyadic</td>
<td>12</td>
<td>left</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Logical</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>short-circuit and</td>
<td>&amp;&amp;</td>
<td>dyadic</td>
<td>7</td>
<td>left</td>
</tr>
<tr>
<td>xor</td>
<td>^^</td>
<td>dyadic</td>
<td>6</td>
<td>left</td>
</tr>
<tr>
<td>equivalence</td>
<td>! ^^</td>
<td>dyadic</td>
<td>6</td>
<td>left</td>
</tr>
<tr>
<td>short-circuit or</td>
<td></td>
<td></td>
<td></td>
<td>dyadic</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Assignment</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>assign</td>
<td>=</td>
<td>dyadic</td>
<td>2</td>
<td>right</td>
</tr>
<tr>
<td>multiply-assign</td>
<td>*=</td>
<td>dyadic</td>
<td>2</td>
<td>right</td>
</tr>
<tr>
<td>divide-assign</td>
<td>/=</td>
<td>dyadic</td>
<td>2</td>
<td>right</td>
</tr>
<tr>
<td>modulo-assign</td>
<td>%=</td>
<td>dyadic</td>
<td>2</td>
<td>right</td>
</tr>
<tr>
<td>add-assign</td>
<td>+=</td>
<td>dyadic</td>
<td>2</td>
<td>right</td>
</tr>
<tr>
<td>subtract-assign</td>
<td>-=</td>
<td>dyadic</td>
<td>2</td>
<td>right</td>
</tr>
</tbody>
</table>

**Table 6. Arithmetic operators**

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

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

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

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

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

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

Two examples showing state referencing follow.

![Figure 74. Scoping example - states (1)](image)

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

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

\[ \text{event} \rightarrow \text{a} \]
as that does not distinguish the different targets.

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

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

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

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

With all three transitions present, transition t2 could be specified by just `event -> $a`

since that specifies ‘a’ in the scope of state s, and a2 is the nearest state of that name in state s. Similarly transition t2 could be specified by just `event -> $a`

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

```
statechart sc

m

 a
  \alpha1->a,aa
  aa
  \alpha2->ab
  ab
  \alpha3->$a

f

fa
\varphi1->g,ga

fb

\varphi2->$g,ga

\varphi3->$g
g

g a
  \varphi3->g

gb
```

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

The statechart level is the outermost named level, and *global* PCOs, events, tagnames and variables are declared in this scope by putting their declarations between the statechart statement and the first `state` statement. More *local* PCOs, events, tagnames and variables are declared either by putting their declarations immediately after the `state` statement of the required scope, or by placing the declarations elsewhere, but applying scoping operators to specify their effective scope.
Every state/PCO/event/tagname/variable that is declared is in scope to its descendants, unless a descendant defines a new item of the same name, in which case the most local item is in scope by default. Looking at this from the perspective of an item being referenced: the item will be found by an outbound search, starting at the current scoping level, and, if the item is not found to have been declared there, backing out one hierarchical level at a time until the item is found. This means that scoping operators are not needed to address the most local name.

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

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

**Design of scoping operators**

There are four scoping operators:

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

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

- operator symbols
- operator precedence
- operator associativity

The operators are defined as follows:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Symbol</th>
<th>Arity</th>
<th>Precedence</th>
<th>Associativity</th>
<th>Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>parent scope</td>
<td>$</td>
<td>monadic</td>
<td>19</td>
<td>right</td>
<td>prefix</td>
</tr>
<tr>
<td>statechart scope</td>
<td>::</td>
<td>monadic</td>
<td>19</td>
<td>right</td>
<td>prefix</td>
</tr>
<tr>
<td>named child scope</td>
<td>.</td>
<td>dyadic</td>
<td>20</td>
<td>right</td>
<td>infix</td>
</tr>
<tr>
<td>named ancestor scope</td>
<td>%</td>
<td>dyadic</td>
<td>20</td>
<td>right</td>
<td>infix</td>
</tr>
</tbody>
</table>

Table 7. Scoping operators
It is good to realise that there is a major difference in the way scoping operators work compared with arithmetic operators. Arithmetic operators apply their own operation after evaluating their arguments (which they do by a recursive call to the evaluator). For example, a simplified PROLOG predicate to evaluate the monadic minus operation on a parameter P1 might be:

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

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

Similarly for dyadic operations (simplified):

```
ev_expr(MPATH, [[ex_dyadic, dminus], P1, P2], V) :-
    ev_expr(MPATH, P1, W1), /* evaluate P1 */
    ev_expr(MPATH, P2, W2), /* evaluate P2 */
    V is W1 - W2, /* operator's own action */
!.
```

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

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

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

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

---

1 Various factors ignored here: error conditions, details of type and wrapping of data, and overloading of the operator (i.e. different actions on different types of data).
Similarly for a dyadic scoping operator. The following operator evaluates its first argument (P1), as a required addition to the machine path, so as to make the scope more local. The second argument (P2) is then evaluated in the new scope.

\[
\text{ev_com_expr}(\text{MPATH}, [[\text{ex_dyadic, descend}], P1, P2], V) :- \\
\quad \text{ev_com_expr}(\text{MPATH}, P1, V1), \\
\quad V1 = [\text{ID, }], \\
\quad \text{MPATH2} = [\text{ID, MPATH}], \\
\quad \text{ev_com_expr}(\text{MPATH2}, P2, V), \\
\quad !.
\]

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

\[\text{aa}\cdot\text{bb}\cdot\text{cc}\cdot\text{dd}\]

is equivalent to

\[\text{aa}\cdot(\text{bb}\cdot(\text{cc}\cdot\text{dd}))\]

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

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

Similarly

\[\text{aa%\%bb%\%cc%\%dd}\]

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

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

\[\$\$\$\text{aa}\]

which is equivalent to

\[\$(\$(\text{aa}))\]

backs out three levels then evaluates \(\text{aa}\).

The expression

\[\text{::\$aa}\]
backs out to the outermost shell, then backs out one more level, which in STATECRUNCHER is
admissible, as the "::" operator backs out to the statechart level, from which it is
possible to back out once more to the absolute level.

The expression
$$::aa$$
would normally be pointless, as it backs out one level before performing a global back-out
operation.

These monadic and dyadic operators combine with dyadic operations binding tighter, so that
$$aa.bb.cc$$
which is equivalent to
$$\$(aa.(bb.cc))$$
means back out two levels, then enter aa then enter bb then enter cc. The rule is emerging
that the expression is to be interpreted as a sequence of actions in left-to-right reading
order.

One consideration is that dyadic operators have a higher precedence than monadic ones,
which is fine for expressions such as
$$sa.ba.bb.cc$$
but it means that brackets are needed for adjacent dyadic-monadic accumulations, e.g.
$$cc\\%(\$(dd.var2))$$
which is to be read as: back-out to parent cc, then back out twice more, then descend into dd,
then evaluate var2 in this scope.

Scoping operators have a higher precedence than non-scoping ones. An example of a
combined expression, extending the above example, is:
$$var1 + cc\\%(\$(dd.var2))$$
which is to be read as: evaluate var1, back-out to parent cc, then back out twice more, then
descent into dd, then evaluate var2 in this new scope, then finally add together with the
evaluation of var1.

5.6.2.3 The split operator
This operator is used to define multiple target states of transitions. STATECRUNCHER allows
transitions to specify targets in more than one member of a set. This can take place at various
hierarchical levels, so requiring a target state tree. This is illustrated in the figure below.
Figure 76. Multiple target states

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

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

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Symbol</th>
<th>Arity</th>
<th>Precedence</th>
<th>Associativity</th>
<th>Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>split</td>
<td><code>/\</code></td>
<td>dyadic</td>
<td>14</td>
<td>left</td>
<td>infix</td>
</tr>
</tbody>
</table>

Table 8. Split operator

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

\[ a \ /\ b \ /\ c \ /\ d = ((a /\ b) /\ c) /\ d. \]

**A restriction**

The left hand side of the "." and "%" operators should not be a term which has already been split, (although such a thing does make sense), since such a construction is unusual and the evaluator does not currently support it. So, in the figure below, it would not be permissible to write
\[
\alpha \rightarrow a.((a\backslash ab).x)
\]
Instead, the following should be used:
\[
\alpha \rightarrow a.(aa.x\backslash ab.x)
\]

![Figure 77. Restriction in use of the split operator](image)

**Evaluation of the split operator**

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

<table>
<thead>
<tr>
<th>Evaluation Scope</th>
<th>Expression</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 [bb,aa]</td>
<td>dd/\ee</td>
<td>[[dd,bb,aa], [ee,bb,aa]]</td>
</tr>
<tr>
<td>2 [bb,aa]</td>
<td>pp.dd/\ee</td>
<td>[[dd,pp,bb,aa], [ee,bb,aa]]</td>
</tr>
<tr>
<td>3 [bb,aa]</td>
<td>(pp.dd)/\ee</td>
<td>[[dd,pp,bb,aa], [ee,bb,aa]]</td>
</tr>
<tr>
<td>4 [bb,aa]</td>
<td>pp.(dd/\ee)</td>
<td>[[dd,pp,bb,aa], [ee,pp,bb,aa]]</td>
</tr>
<tr>
<td>5 [bb,aa]</td>
<td>pp.(dd/(ee/\ff.\gg))</td>
<td>[[dd,pp,bb,aa], [ee,pp,bb,aa], [gg,ff,pp,bb,aa]]</td>
</tr>
<tr>
<td>6 [bb,aa]</td>
<td>pp.((dd/\ee)/\ff.\gg)</td>
<td>[[dd,pp,bb,aa], [ee,bb,aa], [gg,ff,pp,bb,aa]]</td>
</tr>
<tr>
<td>7 [bb,aa]</td>
<td>pp.((dd$/\ee)/\ff.\gg)</td>
<td>[[dd,pp,bb,aa], [ee,bb,aa], [gg,ff,pp,bb,aa]]</td>
</tr>
<tr>
<td>8 [bb,aa]</td>
<td>pp.((dd/\ee)/(ff.\ff.\ff.\gg.\hh))</td>
<td>[[dd,pp,bb,aa], [ee,pp,bb,aa], [f2,ff,pp,bb,aa], [hh,gg,pp,bb,aa]]</td>
</tr>
<tr>
<td>9 [bb,aa]</td>
<td>pp.(dd/\ee)/(ff.\ff.\ff.\gg.\hh)</td>
<td>[[dd,pp,bb,aa], [ee,pp,bb,aa], [f2,ff,bb,aa], [hh,gg,bb,aa]]</td>
</tr>
<tr>
<td>10 [cc,bb,aa]</td>
<td>$$$pp.(dd.\ee.\ff/$\gg.\hh.\ii)</td>
<td>[[ff,ee,dd,pp,aa], [ii,gg,aa]]</td>
</tr>
<tr>
<td>11 [cc,bb,aa,sc]</td>
<td>::pp.(dd.\ee.\ff/$\gg.\hh.\ii)</td>
<td>[[ff,ee,dd,pp,sc], [ii,gg,sc]]</td>
</tr>
<tr>
<td>12 [cc,bb,aa,sc]</td>
<td>::$pp/$$dd</td>
<td>[[pp], [dd,aa,sc]]</td>
</tr>
</tbody>
</table>
The target of transition $\alpha$ in Figure 76 is represented by
$$aa.(p.pb.pba/\backslash t.q.(qa.qaa/\backslash qb.qbb))$$
in evaluation scope
$$[a,s,sc]$$
evaluating to
$$[[pba,pb,p,aa,a,s,sc],$$
$$[qaa,qa,q,t,aa,a,s,sc],$$
$$[qbb,qb,q,t,aa,a,s,sc]]$$

### 5.6.3 Functions

#### 5.6.3.1 Arithmetic functions

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

<table>
<thead>
<tr>
<th>Basic arithmetic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs($P1$)</td>
<td>absolute value of a number</td>
</tr>
<tr>
<td>maximum(list)</td>
<td>maximum of several numbers, e.g. $i=$maximum($v1,v2+1,v3$)</td>
</tr>
<tr>
<td>minimum(list)</td>
<td>minimum of several numbers, e.g. $i=$minimum($v1,v2+1,v3$)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>String related</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>format($P1$,P2)</td>
<td>Format integer expression $P1$ as text. $P2$ is the field width: -ve for left justify, 0 for just fit, +ve for right justify.</td>
</tr>
<tr>
<td>length($P1$)</td>
<td>length of string</td>
</tr>
<tr>
<td>lower_case($P1$)</td>
<td>convert string to lower case</td>
</tr>
<tr>
<td>upper_case($P1$)</td>
<td>convert string to upper case</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Casting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cast($P1$)</td>
<td>$i=$cast($j$) allows an assignment that would otherwise be a type mismatch</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tracing</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>trace(list)</td>
<td>add parameter(s) to the trace list</td>
</tr>
<tr>
<td>trace_clear()</td>
<td>clear the trace list</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>System information</th>
<th>Description</th>
</tr>
</thead>
</table>
get_nworlds(P1) | get_nworlds() or get_nworlds(1) gets the number of worlds at the start of event processing. get_nworlds(2) gets the dynamic number of worlds.

<table>
<thead>
<tr>
<th>Nondeterminism control</th>
</tr>
</thead>
<tbody>
<tr>
<td>no_race()</td>
</tr>
<tr>
<td>low_race()</td>
</tr>
<tr>
<td>med_race()</td>
</tr>
<tr>
<td>high_race()</td>
</tr>
<tr>
<td>no_set_tran()</td>
</tr>
<tr>
<td>low_set_tran()</td>
</tr>
<tr>
<td>med_set_tran()</td>
</tr>
<tr>
<td>high_set_tran()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Special functions taking a state-expression argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>in(P1)</td>
</tr>
<tr>
<td>clear(P1)</td>
</tr>
<tr>
<td>deep_clear(P1)</td>
</tr>
</tbody>
</table>

Table 10. Functions

5.6.3.2 Special functions

The evaluation of most functions proceeds as follows:

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

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

**in**

The function

\[
in(state-expression)\]

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

**clear and deep_clear**

The function

\[
clear(state-expression)\]

removes history data from the specified state.

The function

\[
deep_clear(state-expression)\]
removes history data from the specified state and all its child states recursively down the hierarchy.

**trace**

The function

\[
\text{trace}(\text{expression})
\]

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

### 5.6.4 Type compatibility in expressions

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

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

Example

\[
\text{colour mycolour = yourcolour;}
\]

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

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

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

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

### 5.6.5 Type compatibility in functions

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

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

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

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

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

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

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

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

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

The following (rather concocted) example shows the potential complexity of scoping operators and the outbound search mechanism to find the nearest variable and its type in scope.
Figure 78. Complex tagname/variable scoping
5.8 Transition block

Transition blocks are part of state statements.

5.8.1 Transition block overview

Figure 79. Overview of transition block
5.8.2 Transition block syntax

**transition block**

```
transition block
  { enter block exit block transition } transition block
```

**enter block**

```
upon enter action block enter block
```

**exit block**

```
upon exit action block exit block
```

**transition**

```
meta-event condition route action block label block transition
```

*if no route or action block, first square bracket must introduce a condition*

**meta event**

```
event expression ( parameter list ) transition event expression ( state expression )
event expression ( state expression )
```

*Figure 80. Transition block syntax (1)*

© Graham G. Thomason 2003-2004
Transition block syntax continued:

**condition**

```
[
  boolean expression
]```

**route**

```
->
  state expression
  ORBITAL STATE
  state expression
  disallowing the split operator, "/\"
```

```
  state expression
  TARGET STATE
  state expression
  allowing the split operator, "/\"
```

**action block**

```
{
  expression
  statement
  fire
  event expression
  destination
  parameter list
  action
  block
  else
  action
  block

  if
  ( boolean expression
  )
  action
  block
}
```

**label block**

```
[
  label-name
  = expression
]```

**Figure 81. Transition block syntax (2)**
5.8.3 Detailed examples of transition block functionality

Remark

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

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

Reminder

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

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

The following figure shows some common examples of transitions. Self-transitions are explained later in this section.
Notes: Exclamation marks on names are attention-drawing, not syntactical. Transitions are shown with explicit target state expressions.

Figure 82. Specification of states
5.8.3.2 A model illustrating internal events

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

![Statechart diagram showing state transitions and meta events.]

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

Source code of the model

```plaintext
statechart sc(s)
event alpha,beta,gamma;
set s(a,b)
   cluster a(a1,p,q)
        state a1 (alpha->p.p2;)
        cluster p(p1,p2) (alpha->q.q2;)
        state p1 (beta->p2;)
        state p2 (beta->p1;)
    
    cluster q(q1,q2) (alpha->a1;)
        state q1 (beta->q2;)
        state q2 (beta->q1;)

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

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

© Graham G. Thomason 2003-2004
5.8.3.3 Conditional transitions and conditional actions

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

![Diagram](image-url)

**Points to note**

- There is a conditional transition on $\alpha$.
- There is a conditional action on the transition on $\beta$, and also on entering state $a_2$.
- The transition on $\gamma$ has an else part.
- The transition on $\delta$ has nested conditional actions.
- The conditional action of the transition on $\varepsilon$ fires an event, putting cluster $z$ in state $z_2$.
- We can set the value of $v$ (used in the conditions) using the setv event.
- We can reset variables and states using the $\eta$ event.

© Graham G. Thomason 2003-2004
Source code of the above model:

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

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

set s(a,z)
    cluster a(a1,a2) {setv(v);  eta->a.al {u=v=w=0; fire zetal;};
    state a1
        (alpha [in($z.xxx.z2) && (v==0)]->a2;
        beta-> a2 if (in($z.z2) && (v==0)) {w=w*10+1;} );
        gamma-> a2 if (v%2==1) {w=w*10+2;w=w*10+3;}
            else {w=w*10+4;w=w*10+5;} );
        delta-> a2 if (v%2==1)
            {if (v==3) (w=w*10+1; }  else (w=w*10+2;}}
            else 
                {if (v==4) (w=w*10+3; else (w=w*10+4; ) } ;
        epsilon->a2 if (v%2==1) {fire zeta2;});
    state a2 {upon enter {  if(v>5) (u=u*10+1;}  else (u=u*10+2;}}
    cluster z(z1,z2) {zeta2->z.z2; zetal->z.z1;}
    state z1;
    state z2;
```

5.8.3.4 Route; orbit; internal and external self-transitions

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

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

- There is no difference between leafstate and non-leafstate internal self-transitions. If they are valid and there is an action attached to them, the action is performed (see transitions on ζ1 and ζ1 below).
- Internal transitions cannot be orbital (the transition on ζ2 is unspecifiable).
External self-transitions are drawn outside the state.

- If they are on a nonleaf state, they can cause transitions to default states, (but not in clusters with history, because the current state is counted as the historical state). This applies to the self-transition on ε3 when state p2 is occupied below.
- If they are on a leafstate, nothing is exited or entered (unless the self-transition is orbital), but actions are executed, and they behave like internal transitions (see transitions on ζ1 and ζ3).
- External self transitions can be orbital (to any height of orbit). In this case they always cause exiting and entering to the height of the orbit (transitions on ζ4 and ε4).
- How is the transition on ε2 to be interpreted? As an internal orbital transition it is undefined and unspecifiable in STATECRUNCHER. It can, however, be regarded as an external transition, a shorthand for what might otherwise be drawn as the transition on ε5. This is specifiable in STATECRUNCHER and the meaning is to exit from whatever deeper states are occupied as far as the orbit, and to re-enter states according to the transition course algorithm as described in section 7.5.

Internal orbital self-transitions (as on ζ2, and as on ε2 if it were to be regarded as internal) are currently unspecifiable. However, they could be given a syntax such as

ε2 ->@shallow_internal
e2 ->@deep_internal

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

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

Figure 85. Orbits and self-transitions, [model u5170b]
Source of this model

statechart sc(a)

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

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

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

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

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

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

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

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

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

Points to note

- Variable v tracks a transition from p to q. Variable u tracks a transition from q to p. The on-transition actions simply add digit 0 to u and v by multiplying by 10. This gives us a complete record of the order of the actions that take place during a transition. The variables can be reset without any transitioning by executing event omega.
- If there are upon enter actions and upon exit actions, the upon enter actions must be specified first.
- An example of orbital notation is delta-> p \$ sc -> q. More detail is given later in this section.
So far, we have been precise about the orbital state. Where states have unique names, the operators can be omitted and the correct state will be found by the outbound search for the nearest state in scope. So we can also specify the example as simply \texttt{delta->sc->q}.

More on orbital transitions

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

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

\[ \begin{array}{c}
\text{event} \Rightarrow \text{orbital\_state} \Rightarrow \text{target\_state}
\end{array} \]

Note that an orbital transition is \textit{not} achieved by specifying the target state in any particular way: a transition on event \texttt{a1} in Figure 86 below might be specified as any of the following:

\begin{itemize}
  \item \texttt{a1 -> aab}
  \item \texttt{a1 -> $aa.aab}$
  \item \texttt{a1 -> $$a.aa.aab}$
  \item \texttt{a1 -> ::s.y.a.aa.aab}$
\end{itemize}

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

Referring to Figure 86, note that it is possible to have an orbital from-superstate transition (transition on event \texttt{e1}).

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

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

Such orbital data is ignored by \texttt{STATECRUNCHER}.

© Graham G. Thomason 2003-2004
Figure 86. Orbital transitions [model t6260]

Useful rules on orbital states

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

- Two examples of evaluated orbits are shown, for the transitions on $\alpha 4$ and $\alpha 5$. Evaluated orbits are machine paths, here in PROLOG list notation, to be read from right to left when descending in the hierarchy.

- To specify the very highest orbital level, the state expression $:\$:sc is used. The reason for this is that $\$:arg evaluates arg in the statechart level (i.e. machine path sc), not at an absolute root level (machine path []). This convention is convenient for statechart-global declarations such as $\$:alpha, $\$:var1. But to specify an orbital state at statechart level it is admittedly not so convenient. Since $\$: must take an argument, it will be the statechart name, and the evaluation scope must be further back still, which is effected by the $.$.

5.8.4 UML pseudo-states

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

![Figure 87. Pseudo-states (option for possible future implementation)
5.8.5 Illegal transitions

The following figure illustrates some examples of illegal transitions.

![Statechart diagram showing illegal transitions](image)

Figure 88. Illegal transitions
Categories of (potentially) illegal transitions

- Set member to co-member: the transitions on $\beta_1$ etc. Such transitions can be legalized by raising the orbit.
- Illegal route: the transitions on $\gamma_1$, $\gamma_2$, $\gamma_3$ do not have a straight-out straight-in route.
- Multiple target states include cluster co-members: the transition on $\varphi$.

Detection of illegal transitions

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

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

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

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

5.8.6 Actions

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

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

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

Under the current semantics (discussed in section 6), actions in one action block take place sequentially.
Knock-on effects of fired event actions

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

![Statechart diagram](image)

Figure 89. Knock-on effects of fired event actions

5.8.7 Labels

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

- the execution time taken in performing a transition. It could be based on an actual measurement. This enables transition tour algorithms to optimize test cases against execution time.

- the probability of a transition in the case of fork nondeterminism. This could make some optimizations in testing strategy possible; see [Zhang].
- the cost of a transition, if there are factors other than execution time that make a transition expensive (or cheap). Any transition requiring manual intervention or observation would probably be classed as very expensive.
- a name for the transition
- a usefulness factor indicating how important it is felt that such a transition should be taken in a test suite.

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

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

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

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

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

Sequencing issues concern:
- When conditions on transitions are evaluated.
- The use of an original or current value of a variable.
- The ordering of processing of on-transition actions.
- The ordering of processing of upon-exit actions.
- The ordering of processing of upon-enter actions.
- The ordering of generation and processing internal meta-events.
The design of algorithms to meet the requirements is a matter of identifying the micro-steps of the transition algorithm and sequencing them in the right order. Some algorithms introduce extra restrictions on transitions, e.g. blocking them when other transitions are in certain phases of execution, but our final choice of algorithm does not require any special restrictions.

6.1 Cycling

Consider the transitions of the figure below:

![Figure 90. Cycling](image)

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

![Figure 91. Cycling with termination](image)

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

© Graham G. Thomason 2003-2004
We note that CHSM prevents cycling by ‘marking’ transitions [CHSM]. CHSM processes fired (‘broadcast’) events after exiting all states on the initiating transition, but before entering any states. Every time a transition is ‘taken’, it is excluded from further participation in the processing ensuing from the initiating transition.

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

**Prevention of infinite cycling**

If no protection is built into a system to prevent infinite cycling, then the system will probably crash on a heap or stack overflow condition, though it is conceivable that some kinds of infinite loops will run indefinitely without consuming memory. Given that we do not mark transitions as taken, or block states, infinite cycling could be prevented by recognising that a configuration of state occupancies, state histories, variable values and traces has been seen before in the cycle. However, this is computationally expensive, as it involves comparing the configuration of a machine (which may be quite extensive) with a number of recorded configurations (which may be quite high). A weakened version of this is to evaluate a hash function of the full state, and to store and compare against that instead. If the co-domain of the hash function is effectively a set of say $2^{54}$ ($\approx 10^{16}$) pseudo-random numbers, then the probability of a false positive match compares favourably with the probability of the user being struck by lightning in a year ($\approx 10^{-8.5}$). A weaker method still is to count transitions executed within the compass of an initiating transition, and to put a maximum, say 100, on the number of ensuing transitions. The initial version of STATECRUNCHER for simplicity will not contain protection against cycling, thus leaving the responsibility with the user (as with looping in conventional programming languages).

**6.2 Maintaining machine integrity**

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

- **exit meta-events**
  These are meta-events that are generated when a state is exited. Other transitions may be triggered by this event.
These are specified as part of a state's transition block, but they belong rather to the state than any one transition. They are the actions that are executed when the state is exited, and can contain events to be fired.

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

© Graham G. Thomason 2003-2004
A major algorithm design issue is when to process these processing steps. As will become apparent, it is not a good idea to execute any of these actions as they occur. Instead, it is better to collect the actions first, and execute at some other time. This gives us various possibilities as to exactly when to execute them, and what other precautions need to be taken.

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

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

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

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

Figure 97 shows one way in which, unless precautions are taken, performing transition actions too early can lead to breakdown of the statechart integrity. Suppose state aaa is
occupied. On event $\alpha$, state $aaa$ is exited. If we immediately process the $\text{exit}(aaa)$ meta-event (and so exit state $aa$ and enter state $ab$), and then return to the transition on $\alpha$, we also end up in state $ac$, and so break the cluster rule that only one member can be occupied.

![Figure 97. Integrity threat (1)](image)

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

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

![Figure 98. In flight and after landing](image)

A simple test for whether a statechart system uses an in-flight or after-landing approach, is as follows:
If the system uses an in-flight transition algorithm, then the event $\beta$ will have no effect (unless the algorithm is adapted in some way). If the after-landing approach is taken, then fired event $\beta$ will trigger a knock-on transition.

6.3 An in-flight approach

6.3.1 In-flight state blocking

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

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

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

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

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

The example below shows an elaboration of the previous example where some set-co-members remain unblocked.
On processing transition $t_1$, the whole of member $a$ of set $s$ will be blocked except set member $d_2$. There is no blocking of member $z$, so the transition there (transition $t_4$, on exiting $d_1$) can take place. Nor does it affect state $d_2$, as it is not an ancestor of the source or target state of our original transition $t_1$ on $a$. So the transition $t_2$ from $d_1$ to $d_2$ can in principle be triggered. Transition $t_3$ is invalid in this situation.

**Orbital transitions**

An orbital transition is blocked if its orbital level takes it to a blocked state. In the figure below, as transition $t_1$ takes place, transition $t_2$ becomes blocked, because state $d_1$ becomes blocked, so it cannot be exited or entered.

![Figure 102. Blocking of orbital transitions](image)

**Unblocking of states**

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

![Figure 103. Unblocking example](image)

**History**

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

The record of a historically occupied child can conveniently always be set whether or not the History/Deep History markers indicate that it is required. The issue of whether to make use of
this data is resolved on cluster entry. This policy is robust in the event of changes to the algorithm.

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

Under ‘the conditions’ we understand
- the requirement that a source state is occupied
- the requirement that the boolean condition expression, (or guard), evaluates to true.

The options are:
- at collection time only
- at execution time only
- on both occasions

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

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

In the following figure, at transition collection time, two transitions on \( \alpha \) are valid. This gives rise to race nondeterminism, so that transition sequences \(<t_1, t_2>\) and \(<t_2, t_1>\) will be prepared. If the condition \([v==0]\) is re-evaluated at execution time, then in the world which processes \(<t_1, t_2>\), transition \(t_2\) will not take place.

![Figure 104. Race with arithmetic transition condition](image)

The following figure shows that there is a need for condition re-evaluation, at least as regards the occupancy requirement. There is race nondeterminism. However, owing to statechart integrity considerations, one transition must invalidate the other. This can be achieved by in-flight blocking or execution time re-evaluation of the conditions (including the source state occupancy). Under nondeterministic processing, a world will be generated in which \(c\) is the
final occupied state and a world will be generated in which cl is final occupied state. It is simply not possible to proceed on the basis that (perhaps just in some world) both transitions must take place.

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

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

The following figure shows that collection of transitions is a one-off process. Suppose event β occurs when state e is occupied. Although transition t8 becomes eligible for processing, it must not be processed on the same occurrence of event β that triggered t7, because when β occurs, state f is vacant.

**Conclusion on condition evaluation**

In view of the threats to machine integrity in race condition situations, we opt for condition evaluation at collection and execution time.
6.3.3 Mutual order of actions and meta-events

We consider the best order in which to process transition actions and meta-events relative to each other, the categories being:

- *exit* meta-events
- upon *exit* actions
- transition actions
- *enter* meta-events
- upon *enter* actions

The order is relevant, because

- Variables are always referenced in the latest context – not, say, the context just prior to the transition. So if a variable is modified by one collected action, a subsequent collected action will see the modified value.
- States are also referenced in the latest context, and may become occupied or vacant through a certain action, so that subsequent fired or generated events do not trigger a transition which they would otherwise have triggered.

Within the context of one transition, each action, however ordered, will be completed before the next one is executed, so it will never be the case that one action causes new blocked states to come into effect and be ‘seen’ by subsequent actions in the list of collected actions. Note, however, that knock-on actions, (actions associated with transitions triggered by events that were fired as an action of an original transition) will typically see more blocked states.

It is clear that *upon exit* actions should precede *upon enter* actions and that these should take place in the order in which they were generated. Similarly *upon exit* meta-events should precede *upon enter* meta-events.

Where in the sequence should the transition actions be executed? Candidate orderings are:

1. 1st transition actions, 2nd *exit* and *upon exit* actions, 3rd *enter* and *upon enter* actions
2. 1st *exit* and *upon exit* actions, 2nd transition actions, 3rd *enter* and *upon enter* actions
3. 1st *exit* and *upon exit* actions, 2nd *enter* and *upon enter* actions, 3rd transition actions

Option (2) has an intuitive feel to it. Note that *actually entering* the target state can never be invalidated by earlier actions, because it has already shadow-taken-place. One disadvantage is that transition actions cannot override on-enter actions. This would be useful, as on-entry actions are generic to many transitions. So if an on-entry action is \( v = v \mod 3 \), (the modulo function) but for a specific transition we would like \( v \) to be set to 5, we cannot do it this way. A work-around is to cancel the on-entry actions and re-write all relevant transition actions to include the appropriate assignment to \( v \). This argument lends support to option 3.

However, we feel that user-intuitiveness is important, and provisionally choose option (2).
As to the question of the order of exit meta-events versus upon exit actions, we choose to do the upon exit actions first. Similarly concerning the order of enter meta-events versus upon enter actions, we choose to do the upon enter actions first.

Where a hierarchy is exited, the exit meta-event and upon exit actions for one level are performed before those of the next level up. Similarly where a hierarchy is entered the enter meta-event and upon enter actions for one level are performed before those of the next level down.

The ordering of all aspects of transition processing for the in-flight approach is therefore:

1. shadow exit (all relevant states), collecting exit meta-events and upon exit actions
2. shadow enter (all relevant states) collecting enter meta-events and upon enter actions
3. block ancestors
4. execute upon exit actions (loop with next step)
5. execute exit meta-events (inner loop to previous step for each hierarchical level)
6. execute transition actions
7. execute upon enter actions (loop with next step)
8. execute enter meta-event (inner loop to previous step for each hierarchical level)
9. unblock ancestors
10. execute real exit (loop with next step)
11. set history (inner loop to previous step for each hierarchical level)
12. execute real enter (all relevant states)

We illustrate this with an example:

![Figure 108. Order of actions on hierarchical entry/exit](image)

The ordering, with bracketed reference to the above numbering, will be:

1. (1a) shadow exit ab
2. (1b) shadow exit a
3. (2a) shadow enter b
4. (2b) shadow enter ba
5. (3a) execute upon exit (ab) action (fire ξ1)
6. (4a) execute meta-event exit (ab)
7. (3b) execute upon exit (a) action (fire ξ2)
8. (4b) execute meta-event exit (a)
9. (6) execute on-transition action (fire β)
10. (7a) execute upon enter (b) action (fire ζ3)
11. (8a) execute meta-event enter (b)
12. (7b) execute upon enter (ba) action (fire ζ4)
13. (8b) execute meta-event enter (ba)
14. (10a) real exit ab
15. (10b) real exit a
16. (11b) set history of a
17. (12a) real enter b
18. (12b) real enter ba

**Major disadvantages of the in-flight approach**

The problems with the in-flight approach are that by blocking states:

- it prevents cycling. The fact that this is so can be seen by reference to Figure 90. The transition on α fires β, which triggers a transition involving states which will not be blocked, so that transition can take place. However, the transition on γ will not take place because its source and target states are blocked. Cycling has been found to be useful in generating a number of interleaved traces.

- it may prevent knock-on transitions as in Figure 99. In-flight approaches that do not prevent (all) knock-on effects may be possible. It will be seen that knock-on transitions are essential to composition of models (section 6.5)

Given that the ability to cycle under well-constructed circumstances is desirable, and the relative complexity of blocking and unblocking states, we examine an alternative approach (in the next sub-section), which we will adopt.

**6.4 An after-landing approach**

**6.4.1 After landing ordering**

In this approach, the transition actions are executed after the initiating transition has actually entered the target states. Fired events (and other actions, and meta-events) are processed after completion of exit and enter processing of the transition that fired them. Processing them may be done by an in-line call at the end of processing the original transition, or by placing the new event as a job in a buffer, which we could call a *joblist*, for a read-execute loop.

*The actual implementation in STATECRUNCHER is an in-line call, elaborated on in Figure 140, (p.170).*
The net effect in either case is that, referring to Figure 108 again, we have a new ordering such as the following:

1. real exit ab
2. real exit a
3. set history of a
4. real enter b
5. real enter ba
6. execute upon exit(ab) action (fire ζ1)
7. execute meta-event exit(ab)
8. execute upon exit(a) action (fire ζ2)
9. execute meta-event exit(a)
10. execute on-transition action (fire β)
11. execute upon enter(b) action (fire ζ3)
12. execute meta-event enter(b)
13. execute upon enter(ba) action (fire ζ4)
14. execute meta-event enter(ba)

In Figure 109, the transition on α will be processed to completion, while its action (fire β) will be collected and executed afterwards.

A more complex example shows how multiple fired events and their consequent actions are sequenced:

Figure 109. After-landing equivalence
processing a

Note that the list of events to be processed is built up by depth-first traversal of the nested fired events, but that all are processed at a top-level after completion of the previous one – there is never anything to be re-visited for a previous event. For the processing order to actually make a difference in our example, there would have to be more detail in the model, such as variable assignments on the self-transitions, but we keep the example simple.

6.4.2 Condition evaluation

The discussions under the in-flight approach on race conditions apply equally well to the after-landing approach, as they are not concerned with fired events. The conclusion there, that conditions on transitions processed in transition sequences must be re-evaluated at execution time, applies to the after-landing approach too. An example is given illustrating the time reference of the \texttt{in(\ldots)} function below.

The after-landing approach views transitions such as the one on $\beta$ in Figure 111 from the point in time of completion of the transition on $\alpha$. The transition on $\beta$ will be accepted. This may or may not correspond to the user's instinctive idea of when conditions are evaluated.

6.5 Client-server composition and PCOs

In this section, we see how the after-landing approach enables us to model one software component or function calling another using fired events.
Points to note

- **STATECRUNCHER's composition paradigm** is closely analogous to the function call and
  return of imperative languages such as 'C'.
  - The **making** of the function call is modeled by a fired event
  - The **response** to this is modeled by a transition on the event that was fired
  - The **return statement** is modeled by fired return event
  - The **response** to this is modeled by a transition on the return event that was fired.
  If there are many such calling sequences in a model, return names can be made unique to
  a server function by affixing the function name to the event (e.g. return_max) or by
  putting the return event in a sufficiently local scope (using STATECRUNCHER's scoping
  capabilities).

- The client can be seen as an independent state machine, which can be driven through its
  cycle with events α and return. It does not care who it is that responds to its firing of β,
  nor who it is that provides the return event. A different server to the one shown might
  be connected to the client, e.g. with more states and transitions between its initial and
  final states (S1 and S2). Similarly, the server is independent of its client, except for the
  agreed interface of β and return.

- Event α is supplied externally to the client and server. Events β and return are part of
  the agreed interface between the client and server. We indicate this by putting the events
  on different PCOs. STATECRUNCHER's output will reveal the PCOs so that a test generator
  program can distinguish, and if required, restrict itself to certain PCOs only. We put α on
  pco_ext (for external) and β on pco_comp (for composition). If we had more events
  local to the server only, say, we could put them on pco_serv and so on, but we have
  kept this model to the basics.

- The scheme would not work with the in-flight approach, because the return event would
  not be eligible when needed.
For a discussion of these semantics in relation to the process algebras CSP and CCS, see [StCrSemComp].

6.6 Conclusions on the sequencing in the transition algorithm

Given a requirement to allow cycling and composition between parallel machines, the conclusions for the best approach to the transition algorithm are:

- An after-landing approach
- Condition re-evaluation for transitions at the time they are executed.
7. The transition algorithm

7.1 The formal statechart and the nondeterministic transition function

Finite state machines (FSMs) are often formally described without reference to the hierarchical structures of a Harel or UML or STATECRUNCHER statechart (Harel's AND- and XOR-states; in UML's concurrent and non-concurrent composite states; STATECRUNCHER's sets and clusters). This is because the hierarchical structure is just a convenient way of expressing a mathematically equivalent flattened state space. When the hierarchy is introduced, the terminology changes from FSMs to statecharts, but the two are equivalent. A state in the flattened state space is an element of the Cartesian product of parallel states in the statechart. Only statechart leafstates need be considered, because the occupancy of their ancestors is a derivative of that of the leafstates. If the statechart contains history, variables and traces, then these must also present as terms in the Cartesian product in defining flattened states.

Just as the hierarchical states of a statechart offer convenience in representing the state space, so the structured forms of nondeterminism offer convenience in representing what is equivalent to FSM nondeterminism in the flattened state space. STATECRUNCHER simply structures the nondeterminism into various categories that are easy to visualize in a statechart. As has been seen, STATECRUNCHER supports the following forms of structured nondeterminism, all equivalent to fork nondeterminism in the flattened state space.

- fork
- race
- set-transit
- set action
- set meta-event
- fired event (or broadcast event) nondeterminism.

We gave an example of flattened race nondeterminism in Figure 35.

After processing an event STATECRUNCHER produces a world per distinct state configuration, which, in flattened state space terms, is equivalent to a world for every possible resultant flattened state.
We develop the notion of a world more formally, working from the definition of a NFSM (Nondeterministic Finite State Machine) given by [Hierons 98]:

An NFSM $M$ is defined by a tuple $(S, s_i, h, X, Y)$ in which

- $S$ is a set of states
- $s_i$ is the initial state
- $h$ is the state transition function
- $X$ is the input alphabet
- $Y$ is the output alphabet

Given an NFSM $M$, $S_M$ shall denote the state set of $M$. When $M$ receives an input value $x \in X$, while in state $s \in S$, a transition is executed producing an output value $y \in Y$ and moving $M$ to some state $s' \in S$. The function $h$ gives the possible transitions and has the type $S \times X \rightarrow \mathcal{P}(S \times Y)$ where $\mathcal{P}$ denotes the power set operator. ... An NFSM $M$ is completely specified if, for each $s \in S$ and $x \in X$, $|h(s, x)| \geq 1$. $M$ is deterministic if for each $s \in S$ and $x \in X$, $|h(s, x)| \leq 1$.

What in Hierons' description is the notion of $M$ being in state $s$, is to STATECRUNCHER having an occupancy configuration $s$, and other dynamic properties, where an occupancy configuration gives the occupancy (occupied or vacant) of every state. Several states can be occupied, due to parallelism (modelled by a STATECRUNCHER set), and hierarchy (the fact that a parent of an occupied state is also an occupied state). Remark: the occupancy of non-leaf states can be derived from that of their child states (by the set and cluster rules), so, given the hierarchical structure, the occupancy configuration need only explicitly comprise the set of occupied leaf states.

The 'other dynamic properties' which $s$ must comprise are cluster history and variable values.

In our definitions below, we define $\mathcal{F}(A \times B) \subseteq \mathcal{P}(A \times B)$ to be the set of all functions from $A$ to $B$.

A STATECRUNCHER statechart is therefore $(C, V, P, s_i, v_i, p_i, X, Y, h)$ where

- $C$ is a hierarchy of states (sets, clusters and leafstates), from which we can easily derive
  - $S$, the set of all states
  - $P$, the set of all clusters, $P \subseteq S$
- $V$ is a set of variables. We assume the range of values is finite - it is determined by practical limitations.
- $s_i$ is the initial state
- $v_i$ is a function giving the initial variable values, $V \rightarrow Z$, where $Z$ is the set of integers
- $p_i$ is a function giving the initial history values per cluster, $P \rightarrow S$
- $X$ is the input alphabet (a set of events in STATECRUNCHER)
- $Y$ is the output alphabet (a set of trace elements in STATECRUNCHER)
- $h$ is the state transition function

$$h : [S \times \mathcal{F}(V \times Z) \times \mathcal{F}(P \times S)] \times X \rightarrow \mathcal{P}([S \times \mathcal{F}(V \times Z) \times \mathcal{F}(P \times S)] \times Y),$$

where
° the $\mathcal{F}(V \times Z)$ term represents all the variables with their values
° the $\mathcal{F}(P \times S)$ term represents all the clusters with their histories
° the [...] bracketing on the LHS and RHS is introduced because of the commonality of these terms; they are the STATECRUNCHER worlds, which we can denote by $W$. There may be no worlds in existence.

We could add to this definition

- $Q$ the set of PCOs
- $A$ the set of actions

and a way of attaching them to other components of the statechart, but PCOs are effectively a simple attribute to events, and actions can be absorbed into the transition function, since they occur on transitions and influence the final configurations.

The domain and range of $h$ can be represented as

$$\text{domain}(h): [S \times \mathcal{F}(V \times Z) \times \mathcal{F}(P \times S)] \times Y = W \times X$$

$$\text{range}(h): \mathcal{P}([S \times \mathcal{F}(V \times Z) \times \mathcal{F}(P \times S)] \times Y) = \mathcal{P}(W \times Y)$$

When an event is processed in many worlds, a new set of worlds is produced.

To represent this, we define a multi-input-world transition function:

$$H: \mathcal{P}(W \times X) \rightarrow \mathcal{P}(W \times Y)$$

$$H(A) = \bigcup_{B \in A} h(B)$$

In a practical situation, the elements of the domain of $H$ will all contain the same event in all the Cartesian product terms.

Remark: in the actual STATECRUNCHER implementation, traces also distinguish worlds, so we should strictly say that the dynamic configuration $d$ of a statechart is of type

$$S \times \mathcal{F}(V \times Z) \times \mathcal{F}(P \times S) \times Y^*$$

where $Y^*$ is the set of strings consisting of elements of $Y$, (including the empty sequence). So this could be considered to be the actual type of the range of the transition function $h$. However, the most efficient mode of operation is to clear traces and merge worlds between processing events; if this is not done, old and new traces are concatenated. Traces do not impinge on the transition algorithm. With this understanding, we discount the traces in a dynamic state; in this way we more closely map to the description given by Hierons.

Unfortunately, the term state is overloaded, since it can mean either of

- a part of a statechart: a set, cluster or leafstate. We may also call this a state-machine or just a machine.
- an occupancy configuration of a state-machine.

However, the word state is so much more natural than, say, machine and occupancy that it is often retained, with clarification where needed.
7.2 Statechart properties

The following definitions are available in expressing various properties of a statechart:

source(t): the source state of a transition t
orbit(t): the orbital state of a transition t
targets(t): the set of target states of a transition t
cond(t): the condition on transition t, (dynamically true or false)
actions(t): the sequence of actions attached to transition t

sources(T): the set of source states of a set of transitions T

sources(T) = \{ source(t) \mid t \in T \}

parent(s): the set of parent states of state s (or ∅ for a top level state)
ancestors(s): the set of ancestor states (superstates) of state s (or ∅ for top level states)
children(s): the set of child states of state s (or ∅ for leaf states)
descendants(s): the set of descendant states (substates) of state s
enter_actions(s): the set of on-enter actions attached to state s
exit_actions(s): the set of on-exit actions attached to state s

Machine states S are partitioned into state-types \{clusters, sets, leafstates\}.
We also define \( \text{nonleafs} = \text{clusters} \cup \text{sets} = S \setminus \text{leafstates} \)

For convenience, we write “s is a cluster” to mean “s ∈ clusters” etc.

Furthermore, the arrangement of states is a tree-like hierarchy:

- The set of top-level states is the set of states which are no state's descendant:
  \( \text{toplevels} = \{ s \in S \mid \text{parents}(s) = \emptyset \} \)

- There is only one top-level state\(^1\).
  \( | \text{toplevels} | = 1 \)

- Clusters and sets must have at least one member (=child)\(^2\):
  \( \forall s \in \text{nonleafs} \cdot |\text{children}(s)| \geq 1 \)

\(^1\) One could imagine allowing more than one top-level state, e.g. so as to have two totally independent machines in one source or object file. However, this has little value, and would complicate the description.

\(^2\) One could imagine allowing sets and clusters that contain no children. This would introduce a partition of cluster and sets into \{empties, nonempties\}. However, an empty set or cluster has little benefit (a leaf state will serve as a replacement). To allow empty sets and clusters would only complicate the properties of a statechart.
• Leaf states do not have children
  \[ \forall s \in \text{leafstates} \cdot |\text{children}(s)| = 0 \]

• States have at most one parent
  \[ \forall s \in S \cdot |\text{parent}(s)| \leq 1 \]

• If a state has children, then the parent of those children is the original state
  \[ \forall s \in \text{children}(p) \cdot \text{parent}(s) = p \]

• Ancestors are parents, or parents of ancestors; to express this nonrecursively:
  \[ a \in \text{ancestor}(s) \iff \exists \text{ some sequence } (p_1, p_2, \ldots, p_n) \text{ where } p_1 = a, p_n = s \]
  such that \[ \forall i \in [1, n-1] \cdot p_i = \text{parent}(p_{i+1}) \]

• Descendants are child states or descendants of child states:
  \[ d \in \text{descendant}(s) \iff \exists \text{ some sequence } (p_1, p_2, \ldots, p_n) \text{ where } p_1 = d, p_n = s \]
  such that \[ \forall i \in [1, n-1] \cdot p_i \in \text{children}(p_{i+1}) \]

7.2.1 Dynamic aspects of a statechart

Each state has occupancy; it can be occupied or vacant. States also have a history indication, although it is only relevant to clusters.

States have a history attribute, but for sets and leaf states it is none. For clusters it is either none or the child state that was last occupied.

A third dynamic aspect of a statechart is the value of the variables. We assume the range of values is finite – it is determined by practical limitations.

A full configuration of a statechart contains the occupancies of all states, all state history, and all variable values. An occupancy configuration \( F \) comprises a tuple

\[ \{ \text{occs}, \text{vacs} \} \]

where \( \text{occs} \) is the set of state that are occupied, and \( \text{vacs} \) is the set of states that are vacant.

---

1 For an in-flight algorithm, this would be \( \{ \text{occs}, \text{vacs}, \text{shadow_occs}, \text{shadow_vacs}, \text{blockeds} \} \)
The function $H$ maps a machine state in a configuration to its history.

$$H : F \times S \rightarrow \{\text{none}\} \cup S$$

if for any $H(f,s) = x \in S$

then

$s$ is a cluster

and

$x \in \text{children}(s)$

The function $\mathcal{F}$ maps a variable in a configuration to its (integral) value. $Z$ is the set of integers (within some practical limits)$^1$

$$\mathcal{F} : F \times V \rightarrow Z$$

**Configuration sets$^2$**

As discussed in section 4.9, nondeterminism is handled by creating *worlds* to represent the various alternative outcomes when an event is processed. Worlds contain the dynamic data associated with a statechart (state occupancy, state history and variable values). In other words, each world corresponds to a configuration.

A *configuration set* is a set of worlds $W$ containing state data of a particular statechart. At specific intermediate phases of the transition algorithm, the configuration-set-to-be will in general be a *bag* of worlds rather than a set, though this will be converted to a set on completion of a transition.

**Properties of a valid configuration of a statechart**

1. The statechart as a whole is occupied. This means that all top-level states (although we only allow one) are occupied:

   $$\text{toplevels} \subseteq \text{occs}$$

2. Every state is occupied or vacant but not both

   $$\{\text{occs}, \text{vacs}\} \text{ becomes a partition}$$

3. For every *occupied* cluster, the number of occupied children is 1

   $$\forall s \in \text{clusters} \cap \text{occs} \cdot |\text{children}(s) \cap \text{occs}| = 1$$

4. For every *vacant* cluster, no children are occupied

   $$\forall s \in \text{clusters} \cap \text{vacs} \cdot \text{children}(s) \subseteq \text{vacs}$$

---

$^1$ Later additions are strings and arrays. Array elements can be counted as scalar variables, and the concatenated ASCII values in strings can be considered as integers.

$^2$ Another term that was considered to express this, but which is too imprecise, is *state vectors.*
5. For every occupied set, all children are occupied
\[ \forall s \in \text{sets} \cap \text{occs} \cdot \text{children}(s) \subseteq \text{occs} \]

6. For every vacant set, no children are occupied
\[ \forall s \in \text{sets} \cap \text{vacs} \cdot \text{children}(s) \subseteq \text{vacs} \]

### 7.3 Transition selection

We consider a statechart in configuration \( f \) under some event \( a \)

\( T_a \) is the set of all transitions on event \( a \), (whatever their condition and whatever the configuration-state of the statechart).

\( T_{fa,\text{true}} \) is the set of all transitions where the associated source/orbit/target pre-requisites and transition conditions are true. The default condition is true. The source pre-requisite is that the source state is occupied. No orbit or target pre-requisite is needed in *after-landing* semantics. (Otherwise, these states must not be blocked in any way).

\[
T_{fa,\text{true}} = \{ t : t \in T_a, \ 
\begin{align*}
\text{cond}(t) &= \text{true} \\
\land \text{source}(t) &\in \text{occs} \\
\land \text{targets}(t) &\subseteq \text{occs} \cup \text{vacs} \\
\land (\text{orbit}(t) &\in \text{occs} \cup \text{vacs} \lor \text{orbit}(t) = \emptyset) \}
\end{align*}
\]

\( T_{fa,\text{false}} \) is the set of all transitions where the associated pre-requisites and conditions are false.

\[
T_{fa,\text{false}} = T_a \setminus T_{fa,\text{true}}
\]

\( S_{fa,\text{true}} \) is the set of *source states* of transitions on the event under consideration for which at least one associated transition condition is true:

\[
S_{fa,\text{true}} = \text{sources}(T_{fa,\text{true}})
\]

\( T_{sfa,\text{true}} \) is the set of transitions from source state \( s \) where the associated pre-requisites and conditions are true:

\[
T_{sfa,\text{true}} = \{ t \in T_{fa,\text{true}} | \text{source}(t) = s \}
\]

\( S_{fa,\text{qual}} \) is the set of source states of transitions on the event under consideration for which at least one associated transition condition is true, and for which the source state qualifies under the hierarchy prioritisation algorithm. A state qualifies if there is no transition with a true condition (on the same event) having a source state hierarchically below\(^\dagger\) it.

\( ^\dagger \) In an alternative prioritisation: *above*
\( S_{f,a,\text{qual}} = \{ s \in S_{f,a,\text{true}} \mid \text{descendants}(s) \cap S_{f,a,\text{true}} = \emptyset \} \)

\( T_{f,a,\text{qual}} \) is the set of all transitions where the associated pre-requisites and conditions are true and which qualify under the hierarchy prioritisation algorithm

\[ T_{f,a,\text{qual}} = \{ t \in T_{f,a,\text{true}} \mid \text{source}(t) \in S_{f,a,\text{qual}} \} \]

\( T_{f,a,\text{disq}} \) is the set of all transitions where the associated pre-requisites and conditions are true but which are disqualified by the hierarchy prioritisation algorithm

\[ T_{f,a,\text{disq}} = T_{f,a,\text{true}} \setminus T_{f,a,\text{qual}} \]

Qualifying transitions come from the outermost statechart layer(s) containing true transitions. This could be regarded as an exercise to

- Find the innermost\(^1\) layer of the hierarchy that has at least one true transition
- All true transitions from this layer are qualifying
- All true transitions above\(^3\) this layer are disqualified

\( T_{f,a,\text{qual}}^{s} \) is the set of all transitions from source state \( s \) where the associated pre-requisites and conditions are true and which qualify under the hierarchy prioritisation algorithm

\[ T_{f,a,\text{qual}}^{s} = \{ t \in T_{f,a,\text{qual}} \mid \text{source}(t)=s \} \]

\( T_{f,a,\text{qual}}^{s} \) is the set of sets \( T_{f,a,\text{qual}}^{s} \) for all states \( s \) in \( S_{f,a,\text{qual}} \). Each member set contains all qualifying transitions from the same qualifying source state.

\[ T_{f,a,\text{qual}}^{s} = \{ T_{f,a,\text{qual}}^{s} \mid s \in S_{f,a,\text{qual}} \} \]

Since different elements of \( T_{f,a,\text{qual}}^{s} \) contain transitions from different source states, they are disjoint:

\[ \forall T_1, T_2 \in T_{f,a,\text{qual}}^{s} \cdot T_1 \cap T_2 = \emptyset \]

\( T_{f,a,\text{qual}}^{s} \) is the set of sets where each element of \( T_{f,a,\text{qual}}^{s} \) is formed by taking one element from each element of \( T_{f,a,\text{qual}}^{s} \). (It is rather like a distributed cartesian product, but it is a set of sets, not a set of tuples). Each element of \( T_{f,a,\text{qual}}^{s} \) contains a qualifying transition from each qualifying source-state. There is as yet no notion of orderings of transitions. These elements represent fork nondeterminism.

\[ T_{f,a,\text{qual}}^{s} = \{ T \in \mathcal{P}T_{f,a,\text{qual}} \mid (\forall T_1 \in T, \forall T_2 \in T_{f,a,\text{qual}}^{s} \cdot \#(T_1 \cap T_2) = 1) \} \]

Here, \( \# \) is used to denote the size of a set.

\( T_{f,a,\text{exec}} \) is the set of sequences formed by replacing each set in \( T_{f,a,\text{qual}}^{s} \) by sequences covering every permutation (i.e. ordering) of the replaced set. So each sequence

\[ \text{In the alternative prioritisation: ancestors} \]

\[ \text{In the alternative prioritisation: outermost} \]

\[ \text{In the alternative prioritisation: below} \]
contains an ordering of a qualifying transition from each qualifying source-state. These sequences represent *fork and race* nondeterminism.

\[ T_{f,a,exec} = \{ \text{seq} \in \text{Perm}(\text{tup}_p) \mid \text{tup}_p \in T'_{f,a,qual} \} \]

Set transit nondeterminism is not part of transition *selection*; it is handled within the transition *processing* algorithm.

![Figure 113. Transition derivatives example (similar to test model t6240)](image)

**Notes:**
1. There is just one *event* \( \alpha \) - the superscript identifies *transitions* on \( \alpha \).
2. \([t]\) stands for a true condition, \([f]\) for a false one.
3. To illustrate the alternative prioritisation scheme, we would have \( \alpha^9[t], \alpha^2[f] \).

As an example, given the statechart in Figure 113, assuming leafstates a5b and baa are occupied, event \( \alpha \) leads to the following quantities:
STATECRUNCHER's transition selection algorithm is to select all transition sequences in $T_{f_0,exec}$.

For each sequence, a new world (or more than one) can result after execution of the sequence. As will be seen, worlds are not created in advance of processing each sequence, but rather are created deeper in the algorithm where each individual transition is processed when it needs to change the configuration.

### 7.4 Discussion of Hierarchical Fork Nondeterminism

As mentioned, when there are transitions on the same event at different hierarchical levels, STATECRUNCHER applied the UML-conformant policy of specialisation, whereby inner transitions take precedence over outer ones. We consider here what procedure would best be followed if hierarchical prioritisation is replaced by fork nondeterminism across different hierarchical levels, which we call hierarchical fork nondeterminism.

It must first be decided what is meant by event $a$ in the figure below.
The nondeterminism lies in choosing either (\(\alpha^1\) and \(\alpha^2\)) or just \(\alpha^3\) (with set-transit consequences). We do not allow combinations such as \(\alpha^1\) and \(\alpha^{3,2}\) in Figure 115 below (where \(\alpha^{3,2}\) is considered a logical component of \(\alpha^3\)).

Bold font in a leaf state name indicates an occupied state.

For more general state structures containing many nested clusters and sets, we organize transitions into groups originating from source states which are members of the same cluster or set and groups which stand in hierarchical relationship to one another. We wrap and mark sibling source states in a set with a from-each tag, indicating that a transition must be taken from each source state. We wrap and mark source states in a hierarchical relationship with a from-one tag. Sometimes there will only be one state in a from-each or from-one package, making the issue irrelevant, but in the examples, we show the tag anyway.

We create a quantity \(S_{\text{ext,tree}}\) to represent this; for Figure 114 above, this would be

\[
\left[\text{from-one},\quad \left[\text{from-each},\ baaa,\ bab\right],\quad \left[\text{from-one},\ ba\right]\right].
\]

As a more extensive example, we take the following model, which is similar to the previous one, but with more depth of hierarchy.
Hierarchical fork nondeterminism, as a variation on hierarchical prioritisation, will generate a new $S_{fa,qual}$ set, from which a new $T_{fa,qual}$ and $T_{fa,exec}$ set can be constructed analogously to the previous algorithm.

$S_{fa,true} =$

$$[	ext{from-each},$$
$$[	ext{from-one}, a2, a4],$$
$$[	ext{from-one},$$
$$[	ext{from-one}, ba],$$
$$[	ext{from-each}, baaa, bab)]]$$
Next, per source state, we substitute all transitions from it, with a *from-one* tag, giving $T^*_{fa,true} =$

```
[from-each,
    [from-one, [from-one, $\alpha^2$], [from-one, $\alpha^5$, $\alpha^7$]],
    [from-one,
        [from-one, $\alpha^{10}$, $\alpha^{11}$],
        [from-each, [from-one, $\alpha^{12}$], [from-one, $\alpha^{13}$]]],
]```

This tree can be walked according to the tagged instruction, with example PROLOG code shown following.

**PROLOG code for an each-one walker**

```prolog
/* Each/One-walker data */
eodatal(X) :-
    \[\text{from_each},
        \[\text{from_one},
            \[\text{from_one, a2}],
            \[\text{from_one, a5, a7}]],
        \[\text{from_one},
            \[\text{from_one, a10, a11}],
            \[\text{from_eeach},
                \[\text{from_one, a12}],
                \[\text{from_one, a13}]]\]]].

/* Each/One Walker */
eowalk(X,X) :-
    atom(X).
eowalk([\text{from_one}|T], X) :-
    gn_member(N,T),
    eowalk(M, X).
eowalk([\text{from_each}|T], X) :-
    eowalks(T, X).
eowalks([], []).
eowalks([\text{H}|T], [\text{LH}|LT]) :-
    eowalk(H, LH),
    eowalks(T, LT).

/* Walk the example data */
go_eo:-
    eodatal(X),
    eowalk(X, Y),
    gn_flatten(Y, W),
    write(W), nl,
    fail.
```

© Graham G. Thomason 2003-2004
The output of running go_eo is:

\[
\begin{align*}
[a2,a10] \\
[a2,a11] \\
[a2,a12,a13] \\
[a5,a10] \\
[a5,a11] \\
[a5,a12,a13] \\
[a7,a10] \\
[a7,a11] \\
[a7,a12,a13]
\end{align*}
\]

This corresponds to:

\[ T^{\text{fa,qual}}_{\text{fo,qual}} = \]

\[
\begin{align*}
\{ & \{a2,a10\}, \\
& \{a2,a11\}, \\
& \{a2,a12,a13\}, \\
& \{a5,a10\}, \\
& \{a5,a11\}, \\
& \{a5,a12,a13\}, \\
& \{a7,a10\}, \\
& \{a7,a11\}, \\
& \{a7,a12,a13\} \}
\end{align*}
\]

Permuting the transitions, we obtain the sequences we wish to execute:

\[ T^{\text{fa,exec}}_{\text{fo,exec}} = \]

\[
\begin{align*}
\{ & (t2,t10), (t10,t2), \\
& (t2,t11),(t11,t2), \\
& (t2,t12,t13),(t2,t13,t12),(t12,t2,t13),(t12,t13,t2),(t13,t2,t13),(t13,t12,t2), \\
& (t5,t10),(t10,t5), \\
& (t7,t11),(t11,t7), \\
& (t5,t12,t13),(t5,t13,t12),(t12,t5,t13),(t12,t13,t5),(t13,t2,t13),(t13,t12,t5), \\
& (t7,t10),(t10,t7), \\
& (t7,t11),(t11,t7), \\
& (t7,t12,t13),(t7,t13,t12),(t12,t7,t13),(t12,t13,t7),(t13,t7,t12),(t13,t12,t7) \}
\end{align*}
\]
7.5 Transition course

7.5.1 Effective transitions

A transition arc (including bifurcations) indicates one source state and one or more target states. In general, the transition arc does not indicate leaf states at either end, and these must be determined by some algorithm. The transition course is the actual sequence of states exited and entered, and can be indicated by an effective transition arc, which we show by a dotted line in the figures below. A requirement is that, in the absence of orbital transitions, the transition should be as "low flying" as possible, i.e. it should not exit and enter any states unnecessarily.

The algorithm to find the effective transition arc basically involves:

- Determining the enter tree scope and exit tree scope. These are sometimes (but not always), identical, and might be the common ancestor of the source and target states of the transition. The reason these scopes are needed is given below.
- Constructing an intermediate exit tree to the exit tree scope level.
- Constructing an intermediate enter tree to the enter tree scope level.
- Removing common states between the enter and exit trees, (but not necessarily so when the transition is orbital). The reason for this operation is that effective transitions are as low flying as possible, which means that the exit and enter trees must not take the transition to an unnecessary height. An example of a low-flying transition with higher level intermediate exit and enter trees is the transition on α₁ in Figure 120, (given the occupancy configuration shown in the figure). After common state removal, the highest level remaining is called the altitude of the transition. The residual exit and enter trees are called the definitive exit tree and the definitive entry tree.

The algorithm is explained in more detail in this section.

The scopes are needed, because without them, we would have to exit to statechart level, and we could then be exiting set members that are not involved in the transition. Constructing the enter tree would then be more difficult, because we would have to re-enter states that really never should have been considered for exit, when we want to concentrate on entering states because the transition demands it. Moreover, we would have to ensure that such states are never actually exited and re-entered. It would also be inefficient to work with exit and entry trees to statechart level if this is not necessary.

Figure 117 below indicates how a transition on event α might effectively correspond to the transition marked by α². The tail and tip of the transition arc explicitly give states to be exited and entered.
Deep History

The effect of deep history is to ensure that when a decision must be taken as to which member to enter of a cluster that is under deep history, the member that was last occupied is entered. We call this the historical member. If no member has ever been entered, or the record of the history has been cleared, the default member is taken. If the cluster is already occupied, the currently occupied member is regarded as the target member.

A cluster member is liable to be "under deep history" if there is an ancestor set or cluster that has a deep history marker. We shall see that there are nevertheless circumstances when we do not regard a transition entry step as being under the dominion of an ancestral deep history marker.

Deep history ensures that after an 'excursion' from a cluster, such as the excursion from $t$ to $c$ and back again in Figure 118 below, defined by the transitions on $\alpha$ and $\beta$, the original cluster is back in its original state.

However, a transition such as the one on event $\gamma$ below does not 'see' the deep history, since no member of the cluster with the deep history marker, $t$, undergoes an enter operation. This is the behaviour we want; the local behaviour in clusters $a$, $c$ and $d$ should not be altered by an outer wrapper such as $t$.

The orbital transition on $\delta$, however, does see deep history, because it actually enters cluster $t$, which is marked with deep history, since cluster $t$ is below its orbital level.
Deep history illustration

Deep history is only "seen" if the effective transition arc actually enters the deep history cluster.

In STATECRUNCHER, history is recorded on cluster exit. So it is still present on subsequent cluster entry.

We therefore take the current cluster to act as a more recent equivalent to history than the formal historical cluster, when dealing with a target cluster that is already occupied.
Now the issue in finding the transition course in a statechart with deep history appears at first sight to be a chicken and egg problem:

- to find the transition course, we need, amongst other things, the enter tree
- constructing the enter tree depends on knowing when to apply deep history
- knowing when to apply deep history depends on knowing whether a particular cluster will actually be entered in the effective transition
- knowing whether a particular cluster will actually be entered depends on knowing the transition course.

Despite the apparent circular reasoning, it is possible to find a satisfactory algorithm.

The algorithm parameters available to control the transition course are as follows:

- Exit/enter tree scope logic is based on transition source and transition target states, and orbital state
- Enter tree construction logic is based on explicit target states, history markers, target occupancy, and the orbital state.

The intermediate exit tree is created by (recursively) exiting the highest level in the exit tree scope then the child of each state exited. When a member of a set is exited, sibling set members are also exited.

The intermediate entry tree is more difficult to construct; details follow in Section 7.5.3.

### 7.5.2 Logic for exit and enter tree scopes

**Terminology for the decision logic**

Given a transition, with its source state, orbit and target states, we may refer to:

- the transition common ancestor, \( TnCA \) (the common ancestor of the source state and all target states)
- the target common ancestor, \( TgCA \) (the common ancestor of all target states)
- the source-side child of transition common ancestor
- the target-side child of transition common ancestor
- the source-side child of orbit
- the target-side child of orbit

Not all conceivable algorithms require all these terms. Examples of the terms are given with reference to Figure 122.
For the transition on $\alpha$:

- the **transition common ancestor** is the common ancestor of cluster $c$, cluster $e$ and leafstate $f_1$, which is **cluster s**
- the **target common ancestor** is the common ancestor of cluster $e$ and leafstate $f_1$, which is **set b**

For the transition on $\beta$:

- the **transition common ancestor** is the common ancestor of cluster $c$ and cluster $a$, which is **cluster a**

For the transition on $\gamma$:

- the **transition common ancestor** is the common ancestor of leafstate $c_1$ and leafstate $d_1$, which is **cluster a**
- the source side child of transition common ancestor is cluster $c$
- the target side child of transition common ancestor is cluster $d$
- the orbit is cluster $a$
- the source side child of orbit is cluster $c$
- the target side child of orbit is cluster $d$

The logic for the scope of the intermediate exit and enter trees is given in Figure 123. Legend for that figure (not all terms necessarily used in the current algorithm):

- $Sor$ = Source state of transition
- $Tar$ = Target state of transition, or common ancestor of target states if there are several
- $TnCA$ = Transition common ancestor
- $orb$ = orbital state
- $A > B$ reads "A is a strict ancestor of B." [A is greater in age, as it were].
- $A < B$ reads "A is a strict descendant of B."
- $A / B$ reads "A and B are not in a direct ancestral line."
- $A - 1$ reads "a child of A".
- $A - 1'$ reads "the child of A on the source side of the transition, i.e. the active child of A.
- $A - 1'$ reads "the child of A on the target side". 
Figure 123. Decision logic for scopes

Notes
1. By target scope, we mean the common ancestor of all targets.
2. The above logic could be exhibited in a more condensed form, but as it stands, it brings out separate cases more explicitly, making it easier to review the logic. In particular, cases 1, 3, 5, 7 (orbital cases) condense, as do cases 4, 6, 8 (line-of-descent cases).
3. For all orbital cases, the enter and exit trees have exit and enter scope of orbit-1 and orbit-1 respectively. Where the source and target are in the same line of descent, the enter and exit scope given will necessarily be the same for each.
4. In case 2 we have TnCA-1 and TnCA-1', and that we do not attempt to remove any common tree, because we know there is no common tree.
5. Where there is a low orbit, it will have the effect of limiting the amount of common tree removal. Where there is no orbit, the maximum amount possible of common tree removal will take place.
Notes
1. All the above transitions terminate on state c.
2. The $\alpha_1/\beta_1/\gamma_1/\delta_1$ events are high orbital, the $\alpha_2/\beta_2/\gamma_2/\delta_2$ events are lower orbital, and the $\alpha_3/\beta_3/\gamma_3/\delta_3$ events are non-orbital.
### Table 11. Exit and enter tree scope examples

<table>
<thead>
<tr>
<th>Case</th>
<th>Transition</th>
<th>Orbit</th>
<th>TnCA</th>
<th>Key Properties</th>
<th>Intermed. exit scope</th>
<th>Intermed. enter scope</th>
<th>Attempt common tree removal?</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 ta1</td>
<td>x y</td>
<td>Sor/Tar orbit&gt;TnCA (high orbit)</td>
<td>orbit-1(^s) = x-1(^s) = y = y</td>
<td>No</td>
<td>Orbit <em>above</em> common ancestor Orbit-child scopes No common tree removal.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 ta2</td>
<td>y y</td>
<td>Sor/Tar orbit=TnCA (high orbit)</td>
<td>TnCA-1(^s) y-1(^s) = p = a</td>
<td>No</td>
<td>Orbit <em>at</em> common ancestor Common tree removal would fail if attempted</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 ta3</td>
<td>- y</td>
<td>Sor/Tar no orbit</td>
<td>TnCA-1(^s) y-1(^s) = p = a</td>
<td>No</td>
<td>Common tree removal would fail if attempted</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 tβ1</td>
<td>y a</td>
<td>Sor/Tar orbit&gt;TnCA (high orbit)</td>
<td>orbit-1(^s) = y-1(^s) = a = a</td>
<td>No</td>
<td>Orbit <em>above</em> common ancestor Orbit-child scopes No common tree removal.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 tβ2</td>
<td>a a</td>
<td>Sor&gt;Tar orbit&lt;TnCA (low orbit)</td>
<td>TnCA = a</td>
<td>Yes,</td>
<td>Orbit <em>at</em> common ancestor Transition common ancestor scope <strong>but...</strong> ORBIT will restrict common tree removal</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 tβ3</td>
<td>- a</td>
<td>Sor&gt;Tar no orbit</td>
<td>TnCA = a</td>
<td>Yes</td>
<td>Transition common ancestor scope Common tree will remove c (at least) Note that c's history marker will be seen.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 ty1</td>
<td>b c</td>
<td>Sor=Tar orbit&gt;TnCA (high orbit)</td>
<td>orbit-1(^s) = b-1(^s) = c = c</td>
<td>No</td>
<td>Orbit <em>above</em> common ancestor Orbit-child scopes No common tree removal</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 ty2</td>
<td>c c</td>
<td>Sor&gt;Tar orbit&lt;TnCA (low orbit)</td>
<td>TnCA = c</td>
<td>Yes,</td>
<td>Orbit <em>at</em> common ancestor Transition common ancestor scope <strong>but...</strong> ORBIT will restrict common tree removal</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 ty3</td>
<td>- c</td>
<td>Sor&gt;Tar no orbit</td>
<td>TnCA = c</td>
<td>Yes</td>
<td>Transition common ancestor scope Common tree will remove c (at least) Note that c's history marker will be seen.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 tδ1</td>
<td>b c</td>
<td>Sor&lt;Tar orbit&gt;TnCA high orbit</td>
<td>orbit-1(^s) = b-1(^s) = c = c</td>
<td>No</td>
<td>Orbit <em>above</em> common ancestor Orbit-child scopes No common tree removal</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 tδ2</td>
<td>c c</td>
<td>Sor&lt;Tar orbit=TnCA low orbit</td>
<td>TnCA = c</td>
<td>Yes</td>
<td>Orbit <em>at</em> common ancestor Transition common ancestor scope <strong>but...</strong> ORBIT will restrict common tree removal</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 tδ3</td>
<td>- c</td>
<td>Sor&lt;Tar no orbit</td>
<td>TnCA = c</td>
<td>Yes</td>
<td>Transition common ancestor scope Common tree will remove c (at least) Note that c's history marker will be seen.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
7.5.3 Entry tree construction

Entry of a cluster

We first introduce the terminology guide-mode and orbitality.

![Diagram of guide mode and orbitality](image)

Figure 125. Guide mode and orbitality

The transition on $\alpha$ in Figure 125 is specified as coming from cluster $b$, but this is a non-leafstate; the effective transition source could be various leafstates within cluster $b$: $j$, $k$, $c2$, or $b2$ (but not $a2$, which is not within cluster $b$). The target state is also specified at non-leafstate level, (cluster $d$), the effective target state always being in fact $j$. The transition arrow is not entirely an explicit guide for determining the effective transition. If the transition actually comes from state $b2$, (because state $b2$ is the occupied leafstate) it is clear from the transition arrow that clusters $c$ and $d$ must be entered. This is guided entry. If this transition comes from state $c2$, then cluster $d$ will be entered as guided entry. But the final part of determining the actual transition target (leafstate $j$) is not explicit in the transition arrow, and will be performed as unguided entry.

The transition on $\beta$ illustrates orbitality:

- Cluster $b$ is at-orbit
- Cluster $a$ is above the orbit, i.e. super-orbital
- Clusters $c$ and $d$ and leafstate $k$ are below the orbit, i.e. sub-orbital
- Note that for the transition on $\alpha$ all states in the hierarchy are qualified as no-orbit

The dependency factors for entering a cluster are:

- whether the cluster is entered in guide-mode = guided or unguided
- whether the cluster history-attribute = deep history or history or no history
- whether the cluster history-availability = available or unavailable
- whether the cluster is entered under a dho = deep-history-obligation (on statechart entry set to false) or not. This means that the historical member must be (recursively) entered if possible, due to a deep history marker having set this up in a preceding part of the transition course.
• whether the cluster orbitality = suborbital (i.e. at a level at-or-below orbital level) or superorbital or no-orbit.
• whether the target state occupancy = occupied or vacant

Entry of a set
This is basically as for a cluster, except that
• all members are entered
• there are typically several guides, prescribing entry into various set members.
An illustrative example is given at Figure 128.

The following figures show flow diagrams that specify which member of a cluster is entered using the above factors.

Figure 126. Entry tree for clusters (1)

Rationale for the above
As these are all the guided entry mode cases, the cluster member entered will certainly be the one specified by the guide. The remaining issue is whether or not to impose a deep history obligation (dho) on the member state that is entered, for its (or its descendants’) use when the
guide ceases. The dho is imposed when a cluster is entered with a deep history marker, but it can be cancelled. Cancellation takes place (cases 1 and 4 in Figure 126) when a cluster is entered which has the property that both source and target state belong to it (the transition being local to the cluster, and the deep history being inapplicable) – providing there is no orbit that takes the transition above the cluster being entered. An example of cancellation of the dho taking place is the transition on α1 in Figure 120.

If a cluster is entered which is vacant, or which is occupied but sub-orbital, then it is known that this entry step will form part of the effective transition, as the transition cannot be more local. In these cases, a deep history marker will set up a dho, and an existing dho will be imposed on the member cluster.

Unguided entry:

![Diagram](image)

**Figure 127. Entry tree for clusters (2)**

**Rationale for the above**

The first issue is to determine which member state is to be entered. We first determine whether history is applicable: this is the case if there is a (D) or (H) history marker, or if a
deep history obligation (dho) has been imposed. Having established applicability of history, we regard a currently occupied member state as the intended target, i.e. the present as overriding history, or to put it another way, the currently occupied state is the last entered state and so is the historical state.

This is the correct choice of member state whether or not the effective transition passes through this hierarchical level. If it does, then the entry step will be reflected in the effective transition. If it does not, then the transition is at a local level in the cluster we are entering, so the enter-tree must reflect this; it will be eliminated when the intersection of the entry tree and exit tree is taken to form the common tree.

To apply history to a vacant cluster requires that history data is also available. If it is, we take the historical state; if not, we take the default state. If history is not even applicable, we ignore historical state information and take the default state (even if another member is currently occupied). In this last situation, there is no dho, and this is maintained that way. The fact that the cluster is vacant implies it will be entered in the effective transition too.

The second issue is the deep history obligation (dho) setting. The dho is imposed, maintained or cancelled employing the same considerations as those given under guided entry relating to Figure 126.
The rules as given for a cluster apply, but with the following extra provisions:

- If any member of a set is exited, the entire set must be exited. So if the transition altitude would otherwise be a member of a set, the whole set must be exited recursively upwards in the hierarchy until a non-set state is found (cluster or statechart).

- There can be multiple target states. Construction of the entry tree involves following all multiple target states (as far as they go), then relying on history and default settings. If any one member of a set is entered, all members must of course be entered, be it guided by a target state or relying on history and default states.

When members of a set (or a member of a cluster) are entered under guided entry, all elements of the guide-list that can be consumed, must be consumed as entry takes place. The guide list will be supplied to each member entered, and irrelevant elements in it for each particular member can be discarded. Sometimes (when we are about to commence unguided entry) an entry may be reduced to the empty list; such entries can also be discarded. An example of this is the guide-path as far as $y_3$ in the figure above. The paths of the target of the transition on $\xi$ above, as the entry progresses, would be

Initial guide-paths, in set $x$, as we are about to enter members $x_1$ and $y_1$
After entering member $x_1$ the guide-paths are:

$[ [y_3, x_2, x_1], [q_5, y_4, x_3, x_2, x_1], [p_2, y_2, x_1] ]$

In parallel, member $y_1$ is entered, but the guide paths do not apply, and will be ignored or effaced.

From set $x_1$, members $x_2$ (a set), and $y_2$ (a cluster) must be entered.

After entering set $x_2$, this set retains guide paths as follows (irrelevant ones struck through)

$[ [y_3], [q_5, y_4, x_3], [p_2, y_2] ]$

From $x_2$ we enter $x_3$ and $y_3$.

In $x_3$, we retain one guide path in the list which is as follows.

$[ [q_5, y_4] ]$

In $y_3$, which we must enter anyway, we retain an empty path, which can be effaced from the list

$[ [] ]$

leaving no guide paths, represented by the empty list:

$[ ]$

The remaining guide-path in to $y_4$ and $q_5$ is followed through, being consumed as entry steps are taken.

### 7.5.4 Common tree removal

An exit or enter tree is a nested structure of states, e.g. (simplified) $[a, [b, [c]]]$, with the outermost layer representing the highest part of the statechart hierarchy in the tree. The process of removing common states is to examine the top of each tree for a match and if found, to peel off the outer layer from each structure (giving in our example $[b, [c]]$) and to repeat the operation until a difference is found. An intermediate enter tree of $[a, [b, [c, [d]]]]$ and an intermediate exit tree of $[a, [b, [e, [f]]]]$ would yield definitive trees of $[c, [d]]$ and $[e, [f]]$ respectively.

If the trees are identical, all states are removed, and the transition is effectively a self-transition.

Some nesting layers may represent sets, giving e.g. $[a, [b, [c, d, e]]]$. Where the two trees contain several top-level elements, representing set members, the set members are subjected to common tree removal by a recursive call for each member. This can only occur when their parents have just been removed, so guaranteeing that the enter and exit trees contain the same set members at this stage. The exit-tree and enter-tree orderings of set
members correspond to enable this (set members are ordered in their declaration order). As soon as the exit and enter trees differ, the removal process is complete and they become the definitive exit and enter trees.

Throughout the process, an orbital level (if present) is used in a check so that the process can terminate prematurely, as it were, if the enter and exit trees have reached the level at which no more common tree removal is permitted.

The enter and exit trees actually contain permutation markers on set members so as to support set nondeterminism (section 7.6.5), but this does not affect the test for commonality or the removal of common states.
7.6 Task processing

7.6.1 Introduction to event processing and generalisation to tasks

The algorithm presented here for processing an event, taking account of all forms of nondeterminism (as discussed) involves extensive mutual recursion at many processing levels. We take a top-down approach to event processing, generalising to a task, and leading to a highly general top-level call, which effectively abstracts away many details which are best considered at a lower level.

The main function of the machine engine is to process a single event. The *first* event is normally processed on the initial state, which is unique and so is represented by just one world. However, apart from special situations, an event is typically processed in *each of several worlds*. This is because the previous event (which may entail broadcast events, i.e. fired events and internally generated meta-events) will in general produce many worlds.

When one event has been processed, the resulting worlds will be needed for subsequent event processing. Any worlds representing an *earlier* situation can be destroyed, unless a record of them is required for some reason, in which case they can be retained, but they will not participate in any world merging during event processing.

When an event is processed, the transition *selection* algorithm produces a *set of transition sequences*. This is the input to the transition *execution* algorithm. We generalize a transition into a task. The most general internal STATECRUNCHER call is a call to *process a set of sequences of tasks in a set of worlds*. We also generalize *events* and *actions* into tasks. The outer layer of our algorithm for processing sets of sequences of tasks in many worlds will be applicable to any kind of task. When we come to process one task in one world, we will identify the task and handle it with a specific handler, (a *client handler* to the more general routines).

*The world merging symbol*

In the diagrams, the following symbol is used to indicate world merging:

![World Merging Diagram](image)

Figure 129. World merging diagram
**Processing a sequence**

We represent sequences of items, and sets of items, as lists. As is conventional in PROLOG and elsewhere, we call the first element the head, and the remainder, which is a list, the tail. The tail may contain many elements, or just one, or none at all (in which case it is a null list).

Typical recursive processing of a given list is as follows:
- Process the head using a different, lower level routine, which knows how to process the one item
- Process the tail by a recursive call to the same routine that is handling the given list
- Combine results of the processed head, and processed tail
- The termination condition of the recursion is to process the null list, and return a null list as the processed output.

The example code for processing a task sequence in worlds, given later in this section, illustrates this.

**An issue for any processing routine:**
- In typical parallel world head/tail processing, where we are not concerned with a specific sequence, the processing order (of head and tail) may be reversed. As long as there are no side effects in the two calls, these are equivalent.

![Figure 130. Process task in worlds (i)](image1)

![Figure 131. Process task in worlds (ii)](image2)
• Totally different is the **serial** (or *feed-forward*) case, used for *sequences*, where the output worlds of head processing feed into the tail processing.\(^1\)

![Figure 132. Process task sequence in world](image)

### 7.6.2 The specific routines

We now consider what routines are needed when an event is to be processed in many worlds. In the section following this one, we will generalize these routines to tasks.

**Process event in worlds**

![Figure 133. Process event in worlds](image)

**Process event in world**

![Figure 134. Process event in world](image)

\(^1\) If the sequence ordering convention is reversed, then the output of tail processing feeds into the head processing.
Process transition sequences in world

process transition sequences in world

\[ \text{process each sequence in an identical clone of the world given recursion} \]

process (head) transition sequence in world see below worlds

process (tail) transition sequences in (same) world recursive worlds

Figure 135. Process transition sequences in world

Process transition sequence in world

process transition sequence in world

\[ \text{each transition will produce many worlds - through set transit nondeterminism or broadcast event nondeterminism} \]

process (head) transition in world see below worlds

process (tail) transition sequence in (above) worlds see below worlds

indirect recursion (this is new)

Figure 136. Process transition sequence in world

Process transition sequence in worlds: Algorithm A - outer loop on worlds

process transition sequence in worlds

\[ \text{Algorithm A} \]

\[ \text{outer loop on worlds} \]

process transition sequence in (head) world see above worlds

process transition sequence in (tail) worlds recursive worlds

Figure 137. Process transition sequence in worlds (A)
Process transition sequence in worlds: Algorithm B - outer loop on transitions

![Algorithm B diagram]

Figure 138. Process transition sequence in worlds (B)

Process transition in worlds

This is required by the algorithm B approach to process transition sequence in worlds

![Process transition in worlds diagram]

Figure 139. Process transition in worlds
There are many different exit and enter orderings due to set-transit nondeterminism.

These exit and enter sequences, as housekeeping (simple exit/enter occupancy setting) exercises, will all produce the same effect, since ordering has no consequence. So we may as well just do any one sequence (the first).

Only when actions and meta-events are executed do differences appear.

To correspond precisely to the sequencing described in section 6, meta-event exit sequences and upon-exit sequences should alternate hierarchical level by hierarchical level but in Release 1.06 are simply as shown. Similar ordering applies to enter sequences.

Figure 140. Process transition in world

Housekeeping exit and enter tasks

These are the simple state occupancy changes without execution of any actions.

Figure 141. Process housekeeping task sequence in world
Process meta-event sequences in worlds: Algorithm A - outer loop on worlds

Figure 142. Process meta-event sequences in worlds (A)

Process meta-event sequences in worlds: Algorithm B - outer loop on meta-event sequences

Figure 143. Process meta-event sequences in worlds (B)

We opt for algorithm B (see the dependency analysis below in this section).

Process meta-event sequences in world
[Required by the algorithm A approach to process meta-event sequences in worlds]

Figure 144. Process meta-event sequences in world
**Process meta-event sequence in worlds - Algorithm A - outer loop on worlds**

![Algorithm A](image1)

**Figure 145. Process meta-event sequence in worlds (A)**

**Process meta-event sequence in worlds - Algorithm B outer loop on meta-events**

![Algorithm B](image2)

**Figure 146. Process meta-event sequence in worlds (B)**

**Process meta-event sequence in world**

![Layered Diagram](image3)

**Figure 147. Process meta-event sequence in world**
The clone world action type applies to assignments (including function calls).

Delegated action types are:
- fire event
- conditional action
These action types do not clone directly.

7.6.3 Task generalisation

Generalisation is possible across different kinds of task as long as such tasks are wrapped up in a similar way, with a tag to identify the actual task when it comes to be processed at a low level.

We have the following routines and their generalisation, with the following classification of world mode:
- **serial mode**, as previously explained
- **parallel mode**, as previously explained
- **specific mode**, where one task is to be processed in one given world, i.e. we are at a *client handler* level for processing the task. The task will be identified (as an *event, transition, action* etc.), and handled accordingly. Responsibility is taken for cloning if any changes are to be made to the world, and the changed world (or indirectly generated worlds) are the output. If cloning responsibility has been taken care of by the caller, the routine is free to make alterations in the world given. This mode is used for making direct state occupancy changes.

<table>
<thead>
<tr>
<th>Specific</th>
<th>Generalized</th>
<th>Direct Cloning</th>
<th>World mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>process event in world</td>
<td>specific client handler</td>
<td>no</td>
<td>specific</td>
</tr>
<tr>
<td>process event in worlds</td>
<td>process task in worlds</td>
<td>no</td>
<td>parallel</td>
</tr>
<tr>
<td>process transition <em>seq</em> in <em>world</em></td>
<td>process task <em>seq</em> in <em>world</em></td>
<td>no</td>
<td>serial</td>
</tr>
<tr>
<td>process transition <em>seqs</em> in <em>world</em></td>
<td>process task <em>seqs</em> in <em>world</em></td>
<td>no</td>
<td>parallel</td>
</tr>
<tr>
<td>process transition <em>seq</em> in <em>worlds</em></td>
<td>process task <em>seq</em> in <em>worlds</em> alg A/B</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Alg. A - outer loop over <em>worlds</em></td>
<td>no</td>
<td>parallel</td>
</tr>
<tr>
<td></td>
<td>• Alg. B - outer loop over <em>tasks</em></td>
<td>no</td>
<td>serial</td>
</tr>
<tr>
<td>process transition in <em>world</em></td>
<td>specific client handler</td>
<td>yes</td>
<td>specific</td>
</tr>
<tr>
<td>process hkeep exit <em>seqs</em> in <em>world</em></td>
<td>process task <em>seqs</em> in <em>world</em></td>
<td>no</td>
<td>parallel</td>
</tr>
<tr>
<td>process hkeep exit <em>task</em> in <em>world</em></td>
<td>specific client handler</td>
<td>no</td>
<td>specific</td>
</tr>
<tr>
<td>process hkeep exit task in <em>world</em></td>
<td>process task <em>seqs</em> in <em>world</em></td>
<td>no</td>
<td>parallel</td>
</tr>
<tr>
<td>process meta-event <em>seqs</em> in <em>worlds</em></td>
<td>process task <em>seqs</em> in <em>worlds</em> alg A/B</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Alg. A - outer loop over <em>worlds</em></td>
<td>no</td>
<td>parallel</td>
</tr>
<tr>
<td></td>
<td>• Alg. B - outer loop over <em>seqs</em></td>
<td>no</td>
<td>parallel</td>
</tr>
<tr>
<td>process meta-event in <em>world</em></td>
<td>process task <em>seqs</em> in <em>world</em></td>
<td>no</td>
<td>parallel</td>
</tr>
<tr>
<td>process meta-event <em>seq</em> in <em>worlds</em></td>
<td>process task <em>seq</em> in <em>worlds</em> alg A/B</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Alg. A - outer loop over <em>worlds</em></td>
<td>no</td>
<td>parallel</td>
</tr>
<tr>
<td></td>
<td>• Alg. B - outer loop over <em>tasks</em></td>
<td>no</td>
<td>serial</td>
</tr>
<tr>
<td>process meta-event in <em>world</em></td>
<td>process task <em>seq</em> in <em>world</em></td>
<td>no</td>
<td>serial</td>
</tr>
<tr>
<td>process meta-event in <em>world</em></td>
<td>specific client handler</td>
<td>no</td>
<td>specific</td>
</tr>
<tr>
<td>process action <em>seqs</em> in <em>world</em></td>
<td>process task <em>seqs</em> in <em>world</em></td>
<td>no</td>
<td>parallel</td>
</tr>
<tr>
<td>process action <em>seq</em> in <em>worlds</em></td>
<td>process task <em>seqs</em> in <em>worlds</em></td>
<td>no</td>
<td>parallel</td>
</tr>
<tr>
<td>process action <em>seq</em> in <em>world</em></td>
<td>process task <em>seq</em> in <em>world</em></td>
<td>no</td>
<td>serial</td>
</tr>
<tr>
<td>process action in <em>world</em></td>
<td>specific client handler</td>
<td>some</td>
<td>specific</td>
</tr>
</tbody>
</table>

Table 12. Task generalisation
The actions mentioned in the above table could be upon-exit actions, on-transition actions, or upon-enter actions. Some actions clone a world directly (e.g. an assignment); others may cause world generation indirectly (e.g. firing an event).

Having generalized, we regard the general routines as a task-processing server, serving client handlers that handle single tasks in a single world.

A dependency analysis shows that if we select the algorithm-B options, a minimal set of processing routines will suffice. Shaded routines are not required.

![Figure 150. Task processing dependency diagram](image)

© Graham G. Thomason 2003-2004
7.6.4 Further descriptions of task processing routines

These descriptions complement those of the previous section, including some additional world generation diagrams and actual STATECRUNCHER PROLOG-code (which is remarkably compact for the functionality it gives).

Various equivalent names are used in the descriptions that precede and follow, e.g.

- Process task sequences in worlds  
  a descriptive name
- process_task_seqs_in_worlds  
  a pseudo-code name
- me_process_task_seqs_in_worlds_algB  
  a specific actual code example

The prefix “me_” is the machine engine module naming prefix.

The hierarchy of routines derived from the dependency diagram (see previous section) can be represented as follows.

```
| process_task_seqs_in_worlds  
  (many task sequences in many worlds) |
|----------------------------------------|
| process_task_seq_in_worlds  
  (one task sequence in many worlds)  |
|----------------------------------------|
| process_task_in_worlds  
  (one task in many worlds)  |
|----------------------------------------|
| process_task_in_world  
  (client handler)  
  (one task in one world)  |
```

potential recursion, e.g. when a transition task fires a new event

**Figure 151. Hierarchy of transition-processing routines**

**Process task sequences in worlds**

In Algorithm A we turn the Process task sequences in worlds call into Process task sequences in world calls. This algorithm was not chosen.

In Algorithm B we turn the Process task sequences in worlds call into Process task sequence in worlds calls. This algorithm was chosen.
Process task sequences in worlds

There are two possible approaches, which we discuss and illustrate in figures following:
1. Algorithm A: Outer loop over worlds, inner loop over tasks, requiring an intermediate routine `process_task_seq_in_world`
2. Algorithm B: Outer loop over tasks, inner loop over worlds, requiring an intermediate routine `process_task_in_worlds`

The second of these options is probably better in general, as it probably involves merging of one small worldbag with one large worldbag. (We use the term `worldbag` for consistency with the STATETRUNCHEER code - during processing it is often a bag, but to the user it is always a set, because the user is never confronted with intermediate results). The smaller worldbag is the result of processing just one task since the previous world merge.

We have the option of processing the worlds in head first or tail first order, though in Algorithm B this is determined at a lower level, in `process_task_in_worlds`. The diagrams following illustrate world generation:
- according to algorithm A, with head world first
- according to algorithm B, with head world first
- according to algorithm B, with tail worlds first.
Figure 152. Process task sequence in worlds - Alg. A with head world first

Figure 153. Process task sequence in worlds - Alg. B with head world first
Note tail first - world 1 is merged in last

Figure 154. Process task sequence in worlds - Alg. B with tail worlds first

N.B. The example diagrams taken do not correspond to the same state behaviour as this would not be practical in the limited diagram width.
The **Process task sequence in worlds** Algorithm B routine processes the sequence of tasks in each world in the worldbag. The first task is processed in the worldbag which obtains on calling the routine. Subsequent tasks are processed in the subsequent worldbags resulting from processing the previous task.

**PROLOG code for process task sequence in worlds**

```prolog
/*---------------------------------*/ /* no tasks, OUTWORLDS = INWORLDS */ /*---------------------------------*/
me_process_task_seq_in_worlds_algB([], INWORLDS, INWORLDS):-
    me_set_world_and_bag(INWORLDS), !.

/*---------------------------------*/ /* one task, many worlds */ /*---------------------------------*/
me_process_task_seq_in_worlds_algB([TASK], INWORLDS, OUTWORLDS):-
    me_process_task_in_worlds(TASK, INWORLDS, OUTWORLDS),
    me_set_world_and_bag(OUTWORLDS), !.

/*---------------------------------*/ /* many tasks, many worlds */ /*---------------------------------*/
me_process_task_seq_in_worlds_algB([H_TASK | T_TASKS], INWORLDS, OUTWORLDS):-
    me_process_task_seq_in_worlds_algB(T_TASKS, INWORLDS, OUTWORLDS1),
    me_process_task_in_worlds(H_TASK, OUTWORLDS1, OUTWORLDS2),
    me_merge_worlds(OUTWORLDS2, OUTWORLDS),
    me_set_world_and_bag(OUTWORLDS), !.
```

---

1 By convention in the implementation, the tasks are in tail first order (head of list is the last task).
**Process task in worlds**

This routine calls *Process task in world*, which is regarded as a client routine to the task processing service. Client *Process task in world* routines will be written for event processing, transition processing, action processing etc.

It is seen that all client handlers are of signature

```
me_process_task_in_world(TASK, WORLD, OUTWORLDS)
```

and that by conforming to this, routines which really are hardly aware of nondeterminism are embeddable in a scheme for handling highly nondeterministic tasks.

**PROLOG code for process task in worlds**

```prolog
/* -------------------------- */
/* no worlds */
/* -------------------------- */
me_process_task_in_worlds(_, [], []):- !.

/* -------------------------- */
/* one world */
/* -------------------------- */
me_process_task_in_worlds(TASK, [WORLD], OUTWORLDS):- !, /* this must be the ONLY way to handle one task, one world */
da_write_world(WORLD),
me_process_task_in_world(TASK, WORLD, OUTWORLDS), /* calls client handler */
me_set_world_and_bag(OUTWORLDS),
!.

/* -------------------------- */
/* many worlds */
/* -------------------------- */
me_process_task_in_worlds(TASK, [H_INWORLD|T_INWORLDS], OUTWORLDS):-
me_process_task_in_worlds(TASK, T_INWORLDS, OUTWORLDS1),
da_write_world(H_INWORLD),
me_process_task_in_world(TASK, H_INWORLD, OUTWORLDS2),/*calls client handler*/
me_merge_worlds(OUTWORLDS1, OUTWORLDS2, OUTWORLDS),
me_set_world_and_bag(OUTWORLDS), !.
```

---

7.6.5 Set-transit nondeterminism; permutable sequences and trees

We are nearly ready to review the individual task *client handlers*. But first we illustrate exit and enter trees, and task permutations derived from them, as needed in *Process transition in world*, where we process set nondeterminism. In this section, we will denote sequences using square brackets, for compatibility with illustrative PROLOG examples.

The following model gives rise to set-transit nondeterminism:
various states and transitions: $\alpha^1, \alpha^2, \alpha^3, \alpha^4, \alpha^5, \alpha^6, \alpha^7, \alpha^8, \alpha^9$

Figure 155. Effect of set-transit nondeterminism (cf. Figure 113)

We ignore the fork on transitions $\alpha^{10}$ and $\alpha^{11}$, and any races with any others ($\alpha^1 - \alpha^9$), since these will have been abstracted away by the time one transition is to be processed in one world. We take the exiting part of $\alpha^{10}$ as our example. In general an exit tree is produced, in our example as follows:

```
exit ba
  /   \
/     \
exit baa exit bab
  /   \
/     \
exit baaa exit baab
  /   \
/     \
exit baaaa exit baaab
```

Figure 156. Exit tree

Each node will give rise to a permutation of its branches. The exit sequences are equivalent to the paths (from right to left) through the diagram below.
Notice that we do not permute *all* lowest-level exit tasks in one big permutation. We permute on a level by level basis, retaining orderings imposed by a previous level. So we do not permute the lower level exit tasks from one set member with any exit tasks of a different member. The total number of paths through the above figure is $2^7 = 128$ (being less than the number of permutations of all 8 leaf states to be exited, which is $8! = 40320$).

The above tree happens to be a binary one, because our sets have just two members, but that is of course not the case in general. If there had been an intervening *cluster* in the exit tree, it would not give rise to any extra permutations, as it would be at a node with one branch, and would not be marked for permutation. Such a cluster is shown in dotted outline in Figure 155.

**Permutation handling**

Permutable sequences need to be able to represent parts of the sequence being permuted and parts not. This must apply across different nesting levels. Two elements, $[A, B]$ may form a permutable subsequence, so requiring expansion into $[A, B]$ and $[B, A]$.

One of these elements, say $A$, may itself be a subsequence, say $[a_1, a_2]$, that is to be permuted. The other may be a subsequence $[b_1, b_2]$ that is *not* to be permuted. The required generated subsequences from $[A, B] = [[a_1, a_2], [b_1, b_2]]$

if flattened are then $[a_1, a_2, b_1, b_2], [a_2, a_1, b_1, b_2], [b_1, b_2, a_1, a_2], [b_1, b_2, a_2, a_1]$.

It may be that $[b_1, b_2]$ should be treated as a single element, so that we must generate $[a_1, a_2, [b_1, b_2]], [a_2, a_1, [b_1, b_2]], [[b_1, b_2], a_1, a_2], [[b_1, b_2], a_2, a_1]$.  

© Graham G. Thomason 2003-2004

183
In general, we will need control over what is to be permuted, and what is not, and what is to be flattened and what is not. Certain elements of a sequence are likely to be nested lists in themselves, and as such they must neither be permuted nor flattened.

We can represent all our requirements in a PROLOG-compatible way using the following indicators in a list

- leading element $pm_{y}$(‘permute-yes’) means generate all permutations of this list. The $ is to avoid clashes with user symbols, and in PROLOG code this needs quoting, '$pm_{y}$'. Each permutation generated will substitute $pm_{d}$ (‘permute-done’) for $pm_{y}$. Also, all sublists will be walked for further permutation indications. The $pm_{d}$'s can be removed later. A nonleading $pm_{y}$ element is not recognized as an indicator.
- for any other leading element, the list will be not be permuted at this level, but it will be walked looking for permutations at lower levels.

When a permuted list is flattened, that all sublists starting with $pm_{d}$ are raised up a level.

If the user wishes to effect a permutation on certain chunks of a list monolithically (but with possible sublist permutations as well), then the user will need to wrap the chunks as sublists. Automatic unwrapping of such chunks can be performed if the user supplies an extra $pm_{d}$ element at the head of such chunks.

The following examples show this in action. They show sequences wrapped as permutations.

<table>
<thead>
<tr>
<th>Wrapped sequence</th>
<th>Equivalent straight sequences after flattening</th>
</tr>
</thead>
<tbody>
<tr>
<td>[a, [b, c]]</td>
<td>[a, [b, c]]</td>
</tr>
<tr>
<td></td>
<td>no permutation because no indicator</td>
</tr>
<tr>
<td>['$pm_{y}$', a, b, c]</td>
<td>[a, b, c]</td>
</tr>
<tr>
<td></td>
<td>[a, c, b]</td>
</tr>
<tr>
<td></td>
<td>[b, a, c]</td>
</tr>
<tr>
<td></td>
<td>[b, c, a]</td>
</tr>
<tr>
<td></td>
<td>[c, a, b]</td>
</tr>
<tr>
<td></td>
<td>[c, b, a]</td>
</tr>
<tr>
<td>[a, [b, c], [$pm_{y}$, d, ['$pm_{y}$', e1, e2]], f]</td>
<td>[a, [b, c], d, e1, e2, f]</td>
</tr>
<tr>
<td></td>
<td>[a, [b, c], d, e2, e1, f]</td>
</tr>
<tr>
<td></td>
<td>[a, [b, c], e1, e2, d, f]</td>
</tr>
<tr>
<td></td>
<td>[a, [b, c], e2, e1, d, f]</td>
</tr>
</tbody>
</table>

\[ba, ['$pm_{y}$', BAA, BAB]],
where
BAA= ['$pm_{d}$', baa, ['$pm_{y}$', BAAA, BAAB]],
BAB='BAB',
BAAA= ['$pm_{d}$', baaa, ['$pm_{y}$', baaaa, baaab]],
BAAB= ['$pm_{d}$', baab, ['$pm_{y}$', baaba, baabb]].

Note the user of $pm_{d}$ in the input.
This is the set-transit example, but simplified by condensing all bab... items into one symbol, BAB.
X = [ba, ['$pm_y$', BAA, BAB]],
   BAA = ['$pm_d', baa, ['$pm_y', BAAA, BAAB]],
   BAB = ['$pm_d', bab, ['$pm_y', BABA, BABB]],
   BAAA = ['$pm_d', baaa, ['$pm_y', baaa, baaa]],
   BAAB = ['$pm_d', baab, ['$pm_y', baab, baaa]],
   BABA = ['$pm_d', bab, ['$pm_y', baba, baba]],
   BABB = ['$pm_d', babb, ['$pm_y', babb, babb]].

This is the set-transit example above.

Table 13. Permutation generation
7.6.6 Review of tasks

This section is a review of the various tasks. The following figure shows what tasks exist.

![Figure 158. Breakdown of tasks](image)

7.6.6.1 Process task in world

Here we come to the innermost part of the hierarchy of sets and sequences and worlds, where we must process according to the specific kind of task involved. In C terms, this is just a switch statement to route control to the right lower-level routine. In C++ terms this might be a
question of matching a prototype function according to a parameter type. In PROLOG terms, it is a question of matching a call with a predicate using parameter unification to obtain the right predicate for the task in question.

<table>
<thead>
<tr>
<th>Process_task_in_world</th>
</tr>
</thead>
<tbody>
<tr>
<td>Given an input world and kind of task, switch on kind of task</td>
</tr>
<tr>
<td>Case transition: call process_transition</td>
</tr>
<tr>
<td>Case fired event: call process_event</td>
</tr>
<tr>
<td>Case expression(incl. assignment): call process_expression</td>
</tr>
<tr>
<td>Case conditional: call process_conditional_action</td>
</tr>
<tr>
<td>Case enter-state housekeeping: call process_enter_state_housekeeping</td>
</tr>
<tr>
<td>Case exit-state housekeeping: call process_exit_state_housekeeping</td>
</tr>
</tbody>
</table>

Figure 159. Process_task_in_world

We have seen the general nature of these (Figure 140, Figure 141, Figure 148 and Figure 149). In this section, we consider the tasks in more detail, especially the processing of a transition, where set-nondeterminism is handled.

7.6.6.2 Process event

An outline was given in Figure 134. Transitions for the supplied world only are selected according to the algorithm given in section 7.1. This gives rise to a set of transition sequences. Note that this is the case whether we opt for hierarchical prioritisation or hierarchical fork nondeterminism, except that the latter case generally produces larger sets and longer sequences. The resulting set of transition sequences can be processed by process_task_seq_in_worlds.

<table>
<thead>
<tr>
<th>process_event</th>
</tr>
</thead>
<tbody>
<tr>
<td>In world supplied $w_i$</td>
</tr>
<tr>
<td>Generate the set of transition sequences on this event, $T_{exec}$</td>
</tr>
<tr>
<td>Wrap the world as a list (with this one element)</td>
</tr>
<tr>
<td>Process set of transition sequences using process_task_seq_in_worlds</td>
</tr>
</tbody>
</table>

Figure 160. Review of process event

We do not pre-clone for any of these transition sequences. This routine will perform all processing needed departing from a given world (which will be left intact, eventually being cloned at a lower level). We do not need to think about intermediate processing, such as process a transition sequence, as all has been taken care of in our hierarchy of task processing routines.
7.6.6.3 Process transition

The structure of this task was shown in Figure 140. Transitions are processed according to the 'after-landing' principles and sequencing as already discussed.

A transition can alter state occupancies, and so clones the supplied world. Through the multiple calls to this routine as a result of event processing, many new worlds will be created (and merged). Even if there is no nondeterminism, one new world will be generated, because this routine does not know how much nondeterminism is involved, if any.

We considered transition processing in more detail, covering especially set nondeterminism. Set nondeterminism involves entering (or exiting) the members of a set in different orderings. Unlike fork nondeterminism and race nondeterminism, set-nondeterminism permutations are generated during transition processing, not transition selection. From the transition, an enter tree and an exit tree are derived. From these trees, all forms of set nondeterminism are derived (set transit/action and set meta-event nondeterminism).

7.6.6.4 Process expression

A clone takes place (Figure 148). Expressions are evaluated in a specified scope by a standard call to the evaluator. An assignment is regarded as an expression including the assignment operator ‘=’.

7.6.6.5 Process conditional action

This simply consists of evaluating the condition and recursively calling the relevant nested action (Figure 149).

7.6.6.6 Process enter state housekeeping and Process exit state housekeeping

An outline was given in Figure 141. The routines changes state occupancies and cluster history settings in the current world. No clone of the world is needed as the transition processing routine takes responsibility for it. Since all consequences of state changes have been separated out (on-exit actions etc.), the order in which housekeeping changes are made is irrelevant. Only one of the many orderings generated by permutation of the enter/exit trees is used by the transition processing routine when it calls this routine.
7.6.7 Summary by example of event processing

We process event $\alpha$ in 3 worlds. The transition sequences per world to process are:

- $w_1$: $\langle \alpha^5, \alpha \rangle \langle \alpha^{10}, \alpha^5 \rangle$
- $w_2$: $\langle \alpha^6, \alpha \rangle \langle \alpha^{10}, 
\alpha^6 \rangle$
- $w_3$: $\langle \alpha^6, \alpha \rangle \langle \alpha^{10}, \alpha^6 \rangle \langle \alpha^7, \alpha^6 \rangle$

Figure 161. Example of event processing
8. The STATECRUNCHER command language

This topic is discussed in detail in [StCrPrimer]. Here, we give the inventory of all STATECRUNCHER commands. The most important point about the commands is that they enable a primer (a neighbouring program) to communicate with STATECRUNCHER in various ways, the combination potentially providing very sophisticated test generation algorithms.

The table below shows abbreviated commands as well as unabbreviated ones. Where abbreviated ones are not available, the arrow (→) refers the reader to the unabbreviated one.

Syntax of the descriptions: An optional argument to a command is preceded by a question mark, (?). Normal courier indicates a literal item; italics indicate a non-literal or explanation. A choice is indicated by a vertical bar (|).

The important commands are those that allow setting of state occupancies and variables and traces. These make a state-space exploration algorithm possible. These are

- **WORLD STATEKIND STATENAME MPATH = OCCUPANCY HISTORY**
- **WORLD VAR VARKIND VARIABLENAME MPATH = VALUE**
- **WORLD TRACE = TRACE**

These commands are in STATECRUNCHER's own output format.

<table>
<thead>
<tr>
<th>Abbrev. Command</th>
<th>Command showing typical example and/or typical output</th>
</tr>
</thead>
</table>

### Main processing: high priority black box testing commands

<table>
<thead>
<tr>
<th>pe ...</th>
<th>process event EVENT ?p=PARAMETERS ?t=EXPECTEDTRACE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>pe gamma p=[4, xy] (statechart scope assumed)</td>
</tr>
<tr>
<td></td>
<td>pe [alpha,[sc]] p=1 t=[2,4]</td>
</tr>
<tr>
<td></td>
<td>pe [alpha,[sc]]</td>
</tr>
</tbody>
</table>

*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.*

<table>
<thead>
<tr>
<th>gt</th>
<th>get trace</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>7 TRACE = [1,2]</td>
</tr>
</tbody>
</table>
Main processing: medium priority commands

<table>
<thead>
<tr>
<th>gae</th>
<th>get all events</th>
<th>(whether transitionable or not; not world-related)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>EVENT [theta2, [z3,z,s,sc]] [pcol,[z,s,sc]]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>gate</th>
<th>get all transitionable events</th>
<th>(union from all worlds; no worlds shown)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TREV [[delta,[sc]],0,[],[]]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TREV [[gamma,[sc]],3,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[[r,0,100000],[r,0,100000],[r,0,100000]],[]}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TREV [[gamma,[sc]],1,[[r,0,100000]],[]]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TREV [[gamma,[sc]],2,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[[r,0,100000],[r,0,100000]],[]}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TREV [[alpha,[sc]],0,[],[]]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>gav</th>
<th>get all variables</th>
<th>Gets the value-ranges, not the current value per world</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>VAR INTEGER bool1 [sc] RANGE=[0, 1]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VAR INTEGER coll1 [sc] ENUM=[0, 7, 8, 4, 8]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VAR INTEGER p1 [b2, b, s, sc] RANGE=[0, 9]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VAR STRING str [sc]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>gaw</th>
<th>get all worlds</th>
<th>Gets the current worlds</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[2,7,8]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>gc</th>
<th>get config</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2 statechart sc</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 cluster a [s, sc] =OCC [] **</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 leafstate a1 [a, s, sc] =OCC [] **</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 cluster a2 [a, s, sc] =VAC []</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 VAR INTEGER bool1 [sc] =1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 VAR INTEGER coll1 [sc] =8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 VAR INTEGER p1 [b2, b, s, sc] =unknown</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 VAR STRING p5 [b2, b, s, sc] =unknown</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 VAR STRING str [sc] = [98] = b</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 TRACE = []</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 TREV [[zeta,[s,sc]],</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4,[[r,0,9],[e,0,7,8,4,8],[r,0,1],[&lt;string&gt;]],</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[pcol,[z3,z,s,sc]]]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>outworlds=[2,4]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>number of outworlds=2</td>
<td></td>
</tr>
<tr>
<td>gst</td>
<td>get symbol table</td>
<td></td>
</tr>
<tr>
<td>-----</td>
<td>------------------</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SYMB delta [sc]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>eventdecl []</td>
<td></td>
</tr>
<tr>
<td></td>
<td>XREF leafstate b1:[b, s, sc]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>XREF leafstate z1:[z, s, sc]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>kill ...</th>
<th>kill WORLD</th>
<th>WORLDS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>kill 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>kill [2,7,10]</td>
<td></td>
</tr>
</tbody>
</table>

$\rightarrow$ 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)

$\rightarrow$ 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)

$\rightarrow$ 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)

<table>
<thead>
<tr>
<th>cnw</th>
<th>create new world</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Creates a new world in its default state</td>
</tr>
<tr>
<td></td>
<td>- needed before writing variable/state/trace values to a new world</td>
</tr>
<tr>
<td></td>
<td>34 (the new world number is returned)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>mw</th>
<th>merge worlds</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(useful when all trace/state/variable changes have been made)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>gpt</th>
<th>get processing time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(timing data is set on processing an event)</td>
</tr>
<tr>
<td></td>
<td>exec time=00h 00m 00s 210ms</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>gd</th>
<th>get date</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(get date and time)</td>
</tr>
<tr>
<td></td>
<td>DATE: 24 Apr 2003 16:01:40/649</td>
</tr>
</tbody>
</table>

**Containment of combinatorial explosion: low priority commands**

These commands limit the number of permutations used in set transit nondeterminism and race nondeterminism.

<table>
<thead>
<tr>
<th>nst</th>
<th>no set tran</th>
</tr>
</thead>
<tbody>
<tr>
<td>lst</td>
<td>low set tran</td>
</tr>
<tr>
<td>mst</td>
<td>medium set tran</td>
</tr>
<tr>
<td>hst</td>
<td>high set tran</td>
</tr>
<tr>
<td>nr</td>
<td>no race</td>
</tr>
<tr>
<td>lr</td>
<td>low race</td>
</tr>
<tr>
<td>mr</td>
<td>medium race</td>
</tr>
<tr>
<td>hr</td>
<td>high race</td>
</tr>
</tbody>
</table>

© Graham G. Thomason 2003-2004
Compilation, loading, start-up, and finish: very low priority

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
</table>
| 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 |
| 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

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>help ...</td>
<td>help</td>
</tr>
</tbody>
</table>
| prolog | prolog  
Gives a PROLOG prompt; enter a PROLOG goal |

Table 14. STATECRUNCHER commands

Notes:
- By priority, we mean the priority given through the parse-attempt order, which will affect the response time.
- If anything is to be set in non-existent world, it is created (but a model must have been loaded)
A typical sequence of commands

1. `mm` set model mode
2. `run t5110` load model and enter machine
3. `pe alpha` process event alpha (in statechart scope)
4. `gc` get configuration
5. `pe gamma` process event gamma (in statechart scope)
6. `gc` get configuration
7. `rm` reset machine
8. `pe gamma` process event gamma (in statechart scope)
9. `quit` quit STATECRUNCHER

Error and warning messages are shown in the following table.

**Command parsing**

<table>
<thead>
<tr>
<th>Code</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>PR-E-020</td>
<td>COMMAND SYNTAX ERROR</td>
</tr>
</tbody>
</table>

**Preliminary checks**

<table>
<thead>
<tr>
<th>Code</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>PR-E-040</td>
<td>NO MODEL LOADED (compiler-produced part)</td>
</tr>
<tr>
<td>PR-E-041</td>
<td>NO MODEL LOADED (validator-produced part)</td>
</tr>
<tr>
<td>PR-E-042</td>
<td>MULTIPLE COMPILED FILES LOADED</td>
</tr>
<tr>
<td>PR-E-043</td>
<td>MULTIPLE VALIDATED FILES LOADED</td>
</tr>
<tr>
<td>PR-E-044</td>
<td>THERE WAS A COMPILATION ERROR</td>
</tr>
<tr>
<td>PR-E-045</td>
<td>THERE WAS A VALIDATION ERROR</td>
</tr>
<tr>
<td>PR-E-046</td>
<td>VERSION INCOMPATIBILITY</td>
</tr>
</tbody>
</table>

**Command execution**

<table>
<thead>
<tr>
<th>Code</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>PR-E-060</td>
<td>COMMAND EXECUTION ERROR</td>
</tr>
<tr>
<td>PR-E-061</td>
<td>WORLD IS NEITHER EXTANT NOR EXTINCT</td>
</tr>
</tbody>
</table>

**Internal errors**

<table>
<thead>
<tr>
<th>Code</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>PR-E-900</td>
<td>INTERNAL ERROR - NO COMMAND HANDLER</td>
</tr>
</tbody>
</table>

Table 15. Error and warning messages
9. Using STATECRUNCHER

In this section, we briefly describe what STATECRUNCHER does from an input/output perspective, and how a user prepares a model. Full details of operation are given in the user manual, [StCrUser], which is designed also as a training manual.

This section also gives an indication of how STATECRUNCHER was tested.

In order to illustrate STATECRUNCHER in action in a concrete way, models of the well-known "dining philosophers" problem are developed in this section, without and with the use of a semaphore. These models are deterministic (though care must be taken to ensure that); we discuss a nondeterministic model of a television component as developed in Philips Research in section 10.

9.1 Data flow

The following figure shows the data flow in model compilation and event processing. *Primitive compilation* and *validation* are regarded as one compilation process by the user, as the cp command invokes the validator automatically, (unless the previous phase gives errors).
9.2 Running STATECRUNCHER

STATECRUNCHER runs under [WinProlog] and [SWI-Prolog], and is also available as an MS-DOS executable (using the WinProlog kernel, but the user need not know that the implementation language is PROLOG). Details of how to install and run STATECRUNCHER are given in [StCrManual].

As an executable, STATECRUNCHER will read commands from standard input and direct its output to standard output. The protocol between STATECRUNCHER and the primer program is the subject of a separate report [StCrPrimer].
The development cycle of a STATECRUNCHER model is basically to:

1. Load or run STATECRUNCHER.
2. Prepare a model using a text editor.
3. Compile the model with the cp command. This includes validation and loading and entering the initial configuration of the model.
4. If there are no errors, the model is ready to be driven with pe (process event) commands. Otherwise, edit and re-compile.

A previously compiled model is loaded and made ready for use with the run command.

The user manual [StCrManual] serves as a detailed set of demonstration models, with model source code supplied, and compilation and running instructions given, and output explained.

9.3 Testing of STATECRUNCHER

STATECRUNCHER has been tested throughout its course of development with module tests, where test cases are defined by a PROLOG predicate as follows:

\[
\text{tc}(\text{test\_name, description, predicate\_under\_test, pass\_criterion}).
\]

The test_name is hierarchically defined, e.g. [sc, sy, decl, evns, 2], so that any subtree of all tests can be run, e.g. [sc, sy]. A test harness, described in [StCrGP4], picks up all test cases specified and runs them, producing a test report. An example of an actual test, testing the parse of a list of event-expressions in an event declaration such as

\[
\text{event ev1 , $ev2 , ev3 ;}
\]

is:

\[
\text{tc}([\text{sc, sy, decl, evns, 2}], \text{syzc}(\text{sy\_event\_names, A, SP, R}), SP=E):- \]
\[
\text{A=' ev1 , $ev2 , ev3 '},
\]
\[
\text{E= [g\_ok, [eventnames, l\_ok,}
\]
\[
[ \text{ex\_evt\_expr, [ex\_id, ev1]},
\]
\[
\text{[ex\_evt\_expr, [[ex\_monadic, mback], [ex\_id, ev2]]},
\]
\[
\text{[ex\_evt\_expr, [ex\_id, ev3]] ] ]] }.
\]

The syzc predicate is a testing auxiliary to apply a parsing predicate under test (here sy_event_names) to an ASCII string (the second argument, A, which is ' ev1 , $ev2 , ev3 ') and to produce a status-and-parse (the third argument, SP), and a rest-string (fourth argument, R). The SP=E (E for Expected) term tests that the parse is as expected.

There are also, as system tests, 23 models to test the compiler, 31 models to test the validator, 80 models for machine engine tests, 9 models for stress testing, and many models of practical examples. Tests using these models are run using the [StCrGP4] test harness in the same way as the above parsing example. In all there are well over 10,000 tests, covering general utilities (such as permutation generation), parsing, expression evaluation, machine engine operations etc. System testing of STATECRUNCHER is described in detail in [StCrTest], where diagrams of the main models are given. Users report that STATECRUNCHER is reliable.
9.4 The dining philosophers

In this subsection, we show how a system taken from the CSP literature can be modelled in
STATECRUNCHER. We take a fairly easy example that nevertheless illustrates the essence of
CSP and which is discussed in [Hoare] and [Schneider] (and many other books): the Dining
Philosophers. A first STATECRUNCHER model is shown, with output from a session driving it
to deadlock. A refined model shows how a semaphore can be used to prevent deadlock.

9.4.1 The dining philosophers in CSP

The description of the exercise is given in [Hoare, p77]:

In ancient times, a wealthy philanthropist endowed a College to accommodate five
eminent philosophers. Each philosopher had a room in which he could engage in his
professional activity of thinking; there was also a common dining room, furnished with a
circular table, surrounded by five chairs, each labelled by the name of the philosopher
who was to sit in it. The names of the philosophers were PHIL0, PHIL1, PHIL2, PHIL3,
PHIL4, and they were disposed in this order anticlockwise round the table. To the left of
each philosopher there was laid a golden fork, and in the centre stood a large bowl of
spaghetti, which was constantly replenished.

A philosopher was expected to spend most of his time thinking; but when he felt hungry,
he went to the dining room, sat down in his own chair, picked up his own fork on his left,
and plunged it into the spaghetti. But such is the tangled nature of spaghetti that a second
fork is required to carry it to his mouth. The philosopher therefore has also to pick up the
fork on his right. When he has finished, he would put down both his forks, get up from his
chair, and continue thinking. Of course, a fork can be used by only one philosopher at a
time. If another philosopher wants it, he just has to wait until the fork is available again.

![Figure 163. The dining philosophers](image-url)
Schneider [Schneider, p79] also describes the problem, (but with chopsticks, not forks). Beveridge [Beveridge, p93] describes the problem, and shows a Win32 solution to the deadlock problem using mutexes.

The description of the behaviour in CSP is as follows, where the symbol \( \oplus \) means addition modulo 5 and \( \ominus \) means subtraction modulo 5.

\[
\begin{align*}
PHIL_i &= (i.SitsDown \rightarrow i.PickUpFork.i \rightarrow i.PicksUpFork.(i \oplus 1) \rightarrow i.PutsDownFork.i \rightarrow \nonumber \\
& \quad i.PutsDownFork.(i \oplus 1) \rightarrow i.GetsUp \rightarrow PHIL_i) \nonumber \\
FORK_i &= (i.PicksUpFork.i \rightarrow i.PutsDownFork.i \rightarrow FORK_i \nonumber \\
& \quad |(i \ominus 1).PicksUpFork.i \rightarrow (i \oplus 1).PutsDownFork.i \rightarrow FORK_i) \nonumber \\
PHILOS &= (PHIL_0 + PHIL_1 + PHIL_2 + PHIL_3 + PHIL_4) \nonumber \\
FORKS &= (FORK_0 + FORK_1 + FORK_2 + FORK_3 + FORK_4) \nonumber \\
COLLEGE &= (PHILOS \parallel FORKS) \nonumber 
\end{align*}
\]

9.4.2 The dining philosophers in STATECRUNCHER

9.4.2.1 The model of the dining philosophers in STATECRUNCHER

Figure 164 shows how the dining philosophers can be modelled in STATECRUNCHER.

Following the figure, a description of the model is given, then a session running the model is reproduced.

The source code of the model given later in this section. It corresponds to the figure in almost every detail.
Figure 164. The dining philosophers [model t4330]
A description of the STATECRUNCHER model, with the relationship to the CSP specification

It would have been sufficient to represent the forks as in Figure 165, but we more closely follow the CSP model as implemented in Figure 164. In the Figure 165 model, if a fork is being held, examination of the philosopher states will reveal who is holding it.

![Figure 165. Simpler Fork Model](image)

Now there is a fundamental difference in approach between CSP and STATECRUNCHER, described in the following paragraphs.

[Hoare p.65-66]

When two processes are brought together, the usual intention is that they will interact with each other. These interactions may be regarded as events that require simultaneous participation of both the processes involved.

CSP has an AND condition on combined processes: they must both be able to respond to the common event. STATECRUNCHER transitionable events are transitionable if they trigger a transition in ANY (OR) set members.

The CSP model for composition is not applicable in STATECRUNCHER. The standard model for communication in STATECRUNCHER is the fired event, and a returned fired event, with "after landing" semantics, so that in Figure 166, event alpha is sufficient to bring client to state3 and server to state9.
If the intermediate \textit{state2} is never observed by the user, and the server is regarded as completing instantly, the following simplification can be used:

\textit{Why we need the STATECRUNCHER composition paradigm}

If we allow server and client to respond to the same event, we get a race problem as follows: (we take the \textit{Phil0} situation, but it applies to all the philosophers).
We see that we need a condition on $P_0\_PickFork0$, $[\text{in}($Forks.Fork0.Lying$)]$, because without it, Phil0 can pick up a fork that is in use by Phil4.

We also put a condition on $P_0\_PickFork0$, $[\text{in}($Philosophers.Phil0.SittingHungry$)]$, because it causes the event to show as transitionable only when it really is. When finding transitionable events, STATECRUNCHER evaluates the condition on the transition.

We do not want a race between two transitions on

$P_0\_PickFork0[\text{in}($Philosophers.Phil0.SittingHungry$)]$

and

$P_0\_PickFork0[\text{in}($Forks.Fork0.Lying$)]$

This is because the transitions invalidate each other. But the current semantics will not allow both transitions, because the condition is re-evaluated at execution time.

Without a condition on the second transition, two race orderings will be run, and one will not execute the first transition. Two worlds will be produced, one of which is unwanted.

The solution adopted is the fired event system between client (philosopher) and server (fork), as shown in Figure 164. That is why the fork transitions are called $L_0\_PickFork0$ etc., where $L$ stands for local, as opposed to the external one initiated by the philosopher.

### 9.4.2.2 Session with the dining philosophers [model t4330]

We allow all the philosophers to sit down, then we have them each pick a fork. The events to do this are shown in **bold font**.

```prolog
?- cruncher.
sc: |  mm
sc: |  run t4330
...```
TREV [[L4_PickFork0, [sc]], 0, [], [internal, [sc]]]
TREV [[L1_PickFork1, [sc]], 0, [], [internal, [sc]]]
TREV [[L0_PickFork1, [sc]], 0, [], [internal, [sc]]]
TREV [[L2_PickFork2, [sc]], 0, [], [internal, [sc]]]
TREV [[L1_PickFork2, [sc]], 0, [], [internal, [sc]]]
TREV [[L3_PickFork3, [sc]], 0, [], [internal, [sc]]]
TREV [[L2_PickFork3, [sc]], 0, [], [internal, [sc]]]
TREV [[L4_PickFork4, [sc]], 0, [], [internal, [sc]]]
TREV [[L3_PickFork4, [sc]], 0, [], [internal, [sc]]]

outworlds=[2]
number of outworlds=1

SC: : pe P0_Sit
SC: : pe P1_Sit
SC: : pe P2_Sit
SC: : pe P3_Sit
SC: : pe P4_Sit

SC: : gc (occupied leaf states and external transitionable events only)
7 leafstate SittingHungry [Phil0, Philosophers, College, sc] = OCC [] **
7 leafstate SittingHungry [Phil1, Philosophers, College, sc] = OCC [] **
7 leafstate SittingHungry [Phil2, Philosophers, College, sc] = OCC [] **
7 leafstate SittingHungry [Phil3, Philosophers, College, sc] = OCC [] **
7 leafstate SittingHungry [Phil4, Philosophers, College, sc] = OCC [] **
7 leafstate SittingHungry [Phil0, Philosophers, College, sc] = OCC [] **
7 leafstate Lying [Fork0, Forks, College, sc] = OCC [] **
7 leafstate Lying [Fork1, Forks, College, sc] = OCC [] **
7 leafstate Lying [Fork2, Forks, College, sc] = OCC [] **
7 leafstate Lying [Fork3, Forks, College, sc] = OCC [] **
7 leafstate Lying [Fork4, Forks, College, sc] = OCC [] **
7 TRACE =[]
7 TREV [[P0_PickFork0, [sc]], 0, [], [external, [sc]]]
7 TREV [[P1_PickFork1, [sc]], 0, [], [external, [sc]]]
7 TREV [[P2_PickFork2, [sc]], 0, [], [external, [sc]]]
7 TREV [[P3_PickFork3, [sc]], 0, [], [external, [sc]]]
7 TREV [[P4_PickFork4, [sc]], 0, [], [external, [sc]]]

outworlds=[7]
number of outworlds=1

SC: : gc (unabridged)
statechart sc

set College [sc] = OCC [ ] **
set Philosophers [College,sc] = OCC [ ] **

cluster Phi10 [Philosophers,College,sc] = OCC [ ] **
leafstate SittingHungry [Phi10,Philosophers,College,sc] = VAC [ ]
leafstate OneForkHungry [Phi10,Philosophers,College,sc] = VAC [ ]
leafstate Eating [Phi10,Philosophers,College,sc] = VAC [ ]
leafstate OneForkSatiated [Phi10,Philosophers,College,sc] = VAC [ ]
leafstate SittingSatiated [Phi10,Philosophers,College,sc] = VAC [ ]

cluster Phi11 [Philosophers,College,sc] = OCC [ ] **
leafstate SittingHungry [Phi11,Philosophers,College,sc] = VAC [ ]
leafstate OneForkHungry [Phi11,Philosophers,College,sc] = VAC [ ]
leafstate Eating [Phi11,Philosophers,College,sc] = VAC [ ]
leafstate OneForkSatiated [Phi11,Philosophers,College,sc] = VAC [ ]
leafstate SittingSatiated [Phi11,Philosophers,College,sc] = VAC [ ]

cluster Phi12 [Philosophers,College,sc] = OCC [ ] **
leafstate SittingHungry [Phi12,Philosophers,College,sc] = VAC [ ]
leafstate OneForkHungry [Phi12,Philosophers,College,sc] = VAC [ ]
leafstate Eating [Phi12,Philosophers,College,sc] = VAC [ ]
leafstate OneForkSatiated [Phi12,Philosophers,College,sc] = VAC [ ]
leafstate SittingSatiated [Phi12,Philosophers,College,sc] = VAC [ ]

cluster Phi13 [Philosophers,College,sc] = OCC [ ] **
leafstate SittingHungry [Phi13,Philosophers,College,sc] = VAC [ ]
leafstate OneForkHungry [Phi13,Philosophers,College,sc] = VAC [ ]
leafstate Eating [Phi13,Philosophers,College,sc] = VAC [ ]
leafstate OneForkSatiated [Phi13,Philosophers,College,sc] = VAC [ ]
leafstate SittingSatiated [Phi13,Philosophers,College,sc] = VAC [ ]

cluster Phi14 [Philosophers,College,sc] = OCC [ ] **
leafstate SittingHungry [Phi14,Philosophers,College,sc] = VAC [ ]
leafstate OneForkHungry [Phi14,Philosophers,College,sc] = VAC [ ]
leafstate Eating [Phi14,Philosophers,College,sc] = VAC [ ]
leafstate OneForkSatiated [Phi14,Philosophers,College,sc] = VAC [ ]
leafstate SittingSatiated [Phi14,Philosophers,College,sc] = VAC [ ]

set Forks [College,sc] = OCC [ ] **
cluster ForkO [Forks,College,sc] = OCC [ ] **
leafstate Lying [ForkO,Forks,College,sc] = VAC [ ]
leafstate HeldByPhilO [ForkO,Forks,College,sc] = VAC [ ]
leafstate HeldByPhil4 [ForkO,Forks,College,sc] = VAC [ ]

cluster Forkl [Forks,College,sc] = OCC [ ] **
leafstate Lying [Forkl,Forks,College,sc] = VAC [ ]
leafstate HeldByPhilO [Forkl,Forks,College,sc] = VAC [ ]
leafstate HeldByPhil4 [Forkl,Forks,College,sc] = VAC [ ]

cluster Fork2 [Forks,College,sc] = OCC [ ] **
leafstate Lying [Fork2,Forks,College,sc] = VAC [ ]
leafstate HeldByPhil2 [Fork2,Forks,College,sc] = VAC [ ]
leafstate HeldByPhilO [Fork2,Forks,College,sc] = VAC [ ]

cluster Fork3 [Forks,College,sc] = OCC [ ] **
leafstate Lying [Fork3,Forks,College,sc] = VAC [ ]
leafstate HeldByPhil3 [Fork3,Forks,College,sc] = VAC [ ]
leafstate HeldByPhil2 [Fork3,Forks,College,sc] = VAC [ ]

cluster Fork4 [Forks,College,sc] = OCC [ ] **
leafstate Lying [Fork4,Forks,College,sc] = VAC [ ]
leafstate HeldByPhil4 [Fork4,Forks,College,sc] = VAC [ ]
leafstate HeldByPhil3 [Fork4,Forks,College,sc] = VAC [ ]

TRACE =()
TREV [L0_PutForkO,sc],0,[ ]
TREV [L1_PutFork1,sc],0,[ ]
TREV [L2_PutFork2,sc],0,[ ]
TREV [L3_PutFork3,sc],0,[ ]
TREV [L4_PutFork4,sc],0,[ ]

outworlds=17
There are no transitionable events at the external PCO: deadlock!

9.4.3 Introduction of a semaphore on picking up forks

9.4.3.1 The model with semaphores

Hoare discusses the following solutions to the deadlock:
- Agree that one philosopher should always pick up the wrong fork first.
- Buy more forks.
- Employ a footman to restrict the number of seated philosophers to a maximum of 4.

Schneider adds
- Allowing a philosopher to release a fork if he holds only one.

Neither considers the use of a semaphore, which is the obvious software-technical choice. Beveridge shows how to use Win32 mutexes (mutual exclusions, which are essentially semaphores with a maximum count of one), to solve the problem. The mutexes enable the philosophers to wait for two forks atomically.

In order to reduce unnecessary elements of the model, we make the following simplifications:
- We eliminate the Standing state, and we call the sitting-with-no-forks-held-or-requested the Thinking state. The philosophers now do their thinking at the table.
- We restrict fork states to Lying and Held. The forks respond to events PickFork0 etc., in which no account is taken of who is interacting with the fork.
- We add STATECRUNCHER traces, which are not the same as CSP traces – they are a record of specific selected outputs, generated by the trace(...) function. They are used in black-box testing, representing observable outputs. We record traces on entering and exiting the Eating state: \^trace(P4Eat) and \^trace(P4Stp).

We also shorten the names of some items for convenience. We also distinguish between various categories of event:

External events
- A philosopher has a Pang of hunger
- A philosopher has eaten enough and becomes Full

Events for communication with the semaphore
- Request, Acquire and Release a pair of forks

Internal events
- Fork status administration
In the model, the different events have different PCOs (Points of Control and Observation). The model should be driven by external events only.

The pairwise fork operations work broadly as follows. The Reset state is for when there is no outstanding request. Whenever in a fork-pair cluster a Request for a pair of forks is made, it is either satisfied, broadcasting the Acquisition, with no change of state here, or the cluster goes into the Requested State. Whenever, elsewhere, one of the participating forks is Released, a broadcast event causes a new Try in this cluster to be made to satisfy the request. By the same token, when in the present cluster the forks are Released, two Try events are broadcast so that other clusters can respond to them, each involving one of the forks just released.

The self-transitions on Try01, Try12 etc. are unnecessary, are not present in the implemented model. However, such transitions could be used to trace what has happened, and could be useful in debugging a model.
Figure 169. Model with semaphores [model c4335]
9.4.3.2 A session with the model with semaphores [model t4335]

```plaintext
SC: gc
   statechart sc
   set College [sc] = OCC [] **
   set Philosophers [College,sc] = OCC [] **
   cluster Phi10 [Philosophers,College,sc] = OCC [] **
   leafstate Thinking [Phi10,Philosophers,College,sc] = OCC [] **
   leafstate Waiting [Phi10,Philosophers,College,sc] = VAC []
   leafstate Eating [Phi10,Philosophers,College,sc] = VAC []
   cluster Phi11 [Philosophers,College,sc] = OCC [] **
   leafstate Thinking [Phi11,Philosophers,College,sc] = OCC [] **
   leafstate Waiting [Phi11,Philosophers,College,sc] = VAC []
   leafstate Eating [Phi11,Philosophers,College,sc] = VAC []
   cluster Phi12 [Philosophers,College,sc] = OCC [] **
   leafstate Waiting [Phi12,Philosophers,College,sc] = VAC []
   leafstate Eating [Phi12,Philosophers,College,sc] = VAC []
   cluster Phi13 [Philosophers,College,sc] = OCC [] **
   leafstate Waiting [Phi13,Philosophers,College,sc] = VAC []
   leafstate Eating [Phi13,Philosophers,College,sc] = VAC []
   cluster Phi14 [Philosophers,College,sc] = OCC [] **
   leafstate Waiting [Phi14,Philosophers,College,sc] = VAC []
   leafstate Eating [Phi14,Philosophers,College,sc] = VAC []
   set Forks [College,sc] = OCC [] **
   cluster Fork0 [Forks,College,sc] = OCC [] **
   leafstate Lying [Fork0,Forks,College,sc] = OCC [] **
   leafstate Held [Fork0,Forks,College,sc] = VAC []
   cluster Fork1 [Forks,College,sc] = OCC [] **
   leafstate Lying [Fork1,Forks,College,sc] = OCC [] **
   leafstate Held [Fork1,Forks,College,sc] = VAC []
   cluster Fork2 [Forks,College,sc] = OCC [] **
   leafstate Lying [Fork2,Forks,College,sc] = OCC [] **
   leafstate Held [Fork2,Forks,College,sc] = VAC []
   cluster Fork3 [Forks,College,sc] = OCC [] **
   leafstate Lying [Fork3,Forks,College,sc] = OCC [] **
   leafstate Held [Fork3,Forks,College,sc] = VAC []
   cluster Fork4 [Forks,College,sc] = OCC [] **
   leafstate Lying [Fork4,Forks,College,sc] = OCC [] **
   leafstate Held [Fork4,Forks,College,sc] = VAC []
   cluster Pair01 [Forks,College,sc] = OCC [] **
   leafstate Reset [Pair01,Forks,College,sc] = OCC []
   leafstate Requested [Pair01,Forks,College,sc] = VAC []
   cluster Pair12 [Forks,College,sc] = OCC [] **
   leafstate Reset [Pair12,Forks,College,sc] = OCC []
   leafstate Requested [Pair12,Forks,College,sc] = VAC []
   cluster Pair23 [Forks,College,sc] = OCC [] **
   leafstate Reset [Pair23,Forks,College,sc] = OCC []
   leafstate Requested [Pair23,Forks,College,sc] = VAC []
   cluster Pair34 [Forks,College,sc] = OCC [] **
   leafstate Reset [Pair34,Forks,College,sc] = OCC []
   leafstate Requested [Pair34,Forks,College,sc] = VAC []
   cluster Pair40 [Forks,College,sc] = OCC [] **
   leafstate Reset [Pair40,Forks,College,sc] = OCC []
   leafstate Requested [Pair40,Forks,College,sc] = VAC []
```

TRACE =

TREV [[Pang0,[sc]],0,{},[external,[sc]]]
TREV [[Pang1,[sc]],0,{},[external,[sc]]]
TREV [[Pang2,[sc]],0,{},[external,[sc]]]
TREV [[Pang3,[sc]],0,{},[external,[sc]]]
TREV [[Pang4,[sc]],0,{},[external,[sc]]]
TREV [[Pick0,[sc]],0,{},[internal,[sc]]]

© Graham G. Thomason 2003-2004
outworlds=2
number of outworlds=1

SC I: pe Pang0
SC I: pe Pang1
SC I: pe Pang2
SC I: pe Pang3
SC I: pe Pang4
SC I: gt
20 TRACE = [P2Eat, P0Eat]
SC I: pe Full10
SC I: gt
30 TRACE = [P4Eat, P0Stp, P2Eat, P0Eat]
SC I: pe Full14
SC I: gt
35 TRACE = [P4Stp, P4Eat, P0Stp, P2Eat, P0Eat]
SC I: pe Full2
SC I: gt
50 TRACE = [P3Eat, P1Eat, P2Stp, P4Stp, P4Eat, P0Stp, P2Eat, P0Eat]
SC I: gc
50 statechart sc
50 set College [sc] = OCC [] **
50 set Philosophers [College,sc] = OCC [] **
50 cluster Phil10 [Philosophers,College,sc] = OCC [] **
50 leafstate Thinking [Phil10, Philosophers,College,sc] = OCC [] **
50 leafstate Waiting [Phil10, Philosophers,College,sc] = VAC []
50 leafstate Eating [Phil10, Philosophers,College,sc] = VAC []
50 cluster Phil11 [Philosophers,College,sc] = OCC [] **
50 leafstate Thinking [Phil11, Philosophers,College,sc] = VAC []
50 leafstate Waiting [Phil11, Philosophers,College,sc] = VAC []
50 leafstate Eating [Phil11, Philosophers,College,sc] = OCC [] **
50 cluster Phil12 [Philosophers,College,sc] = OCC [] **
50 leafstate Thinking [Phil12, Philosophers,College,sc] = OCC [] **
50 leafstate Waiting [Phil12, Philosophers,College,sc] = VAC []
50 leafstate Eating [Phil12, Philosophers,College,sc] = VAC []
50 cluster Phil13 [Philosophers,College,sc] = OCC [] **
50 leafstate Thinking [Phil13, Philosophers,College,sc] = VAC []
50 leafstate Waiting [Phil13, Philosophers,College,sc] = VAC []
50 leafstate Eating [Phil13, Philosophers,College,sc] = OCC [] **
50 cluster Phil14 [Philosophers,College,sc] = OCC [] **
50 leafstate Thinking [Phil14, Philosophers,College,sc] = OCC [] **
50 leafstate Waiting [Phil14, Philosophers,College,sc] = VAC []
50 leafstate Eating [Phil14, Philosophers,College,sc] = VAC []
50 set Forks [College,sc] = OCC [] **
50 cluster Fork0 [Forks,College,sc] = OCC [] **
50 leafstate Lying [Fork0, Forks,College,sc] = OCC [] **
50 leafstate Held [Fork0, Forks,College,sc] = VAC []
50 cluster Fork1 [Forks,College,sc] = OCC [] **
50 leafstate Lying [Fork1, Forks,College,sc] = VAC []
50 leafstate Held [Fork1, Forks,College,sc] = OCC [] **
50 cluster Fork2 [Forks,College,sc] = OCC [] **
50 leafstate Lying [Fork2, Forks,College,sc] = VAC []

© Graham G. Thomason 2003-2004
leafstate Held [Fork2,Forks,College,sc] = OCC [] **
cluster Fork3 [Forks,College,sc] = OCC [] **
leafstate Lying [Fork3,Forks,College,sc] = VAC []
leafstate Held [Fork3,Forks,College,sc] = OCC [] **
cluster Fork4 [Forks,College,sc] = OCC [] **
leafstate Lying [Fork4,Forks,College,sc] = VAC []
leafstate Held [Fork4,Forks,College,sc] = OCC [] **
cluster Pair01 [Forks,College,sc] = OCC [] **
leafstate Reset [Pair01,Forks,College,sc] = OCC [] **
leafstate Requested [Pair01,Forks,College,sc] = VAC []
cluster Pair12 [Forks,College,sc] = OCC [] **
leafstate Reset [Pair12,Forks,College,sc] = OCC [] **
leafstate Requested [Pair12,Forks,College,sc] = VAC []
cluster Pair23 [Forks,College,sc] = OCC [] **
leafstate Reset [Pair23,Forks,College,sc] = OCC [] **
leafstate Requested [Pair23,Forks,College,sc] = VAC []
cluster Pair34 [Forks,College,sc] = OCC [] **
leafstate Reset [Pair34,Forks,College,sc] = OCC [] **
leafstate Requested [Pair34,Forks,College,sc] = VAC []
cluster Pair40 [Forks,College,sc] = OCC [] **
leafstate Requested [Pair40,Forks,College,sc] = OCC [] **
leafstate Requested [Pair40,Forks,College,sc] = VAC []

TRACE=[P3Eat,P1Eat,P2Stp,P4Stp,P4Eat,P0Stp,P2Eat,P0Eat]

TREV [[Pang0,[sc]],0,[],[external,[sc]]]
TREV [[Full1,[sc]],0,[],[external,[sc]]]
TREV [[Pang2,[sc]],0,[],[external,[sc]]]
TREV [[Full3,[sc]],0,[],[external,[sc]]]
TREV [[Pang4,[sc]],0,[],[external,[sc]]]
TREV [[Pick0,[sc]],0,[],[internal,[sc]]]
TREV [[Put1,[sc]],0,[],[internal,[sc]]]
TREV [[Put2,[sc]],0,[],[internal,[sc]]]
TREV [[Put3,[sc]],0,[],[internal,[sc]]]
TREV [[Put4,[sc]],0,[],[internal,[sc]]]
TREV [[Req01,[sc]],0,[],[composition,[sc]]]
TREV [[Req112,[sc]],0,[],[composition,[sc]]]
TREV [[Req12,[sc]],0,[],[composition,[sc]]]
TREV [[Req123,[sc]],0,[],[composition,[sc]]]
TREV [[Req23,[sc]],0,[],[composition,[sc]]]
TREV [[Req34,[sc]],0,[],[composition,[sc]]]
TREV [[Req40,[sc]],0,[],[composition,[sc]]]
TREV [[Req40,[sc]],0,[],[composition,[sc]]]

outworlds=[50]
number of outworlds=1
SC: | :
9.4.3.3 Diagram of the events

- Shading shows the fork in use
- **Bold font** shows the change(s) due to the last event

<table>
<thead>
<tr>
<th>Event</th>
<th>PHIL0</th>
<th>PHIL1</th>
<th>PHIL2</th>
<th>PHIL3</th>
<th>PHIL4</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial state</td>
<td>Thinking</td>
<td>Thinking</td>
<td>Thinking</td>
<td>Thinking</td>
<td>Thinking</td>
</tr>
<tr>
<td>Pang0</td>
<td><strong>Eating</strong></td>
<td>Thinking</td>
<td>Thinking</td>
<td>Thinking</td>
<td>Thinking</td>
</tr>
<tr>
<td>Pang1</td>
<td>Eating</td>
<td><strong>Waiting</strong></td>
<td>Thinking</td>
<td>Thinking</td>
<td>Thinking</td>
</tr>
<tr>
<td>Pang2</td>
<td>Eating</td>
<td>Waiting</td>
<td><strong>Eating</strong></td>
<td>Thinking</td>
<td>Thinking</td>
</tr>
<tr>
<td>Pang3</td>
<td>Eating</td>
<td>Waiting</td>
<td>Eating</td>
<td><strong>Waiting</strong></td>
<td>Thinking</td>
</tr>
<tr>
<td>Pang4</td>
<td>Eating</td>
<td>Waiting</td>
<td>Eating</td>
<td>Waiting</td>
<td><strong>Waiting</strong></td>
</tr>
<tr>
<td>Full0</td>
<td><strong>Thinking</strong></td>
<td>Waiting</td>
<td>Eating</td>
<td>Waiting</td>
<td><strong>Eating</strong></td>
</tr>
<tr>
<td>Full1</td>
<td>Thinking</td>
<td>Waiting</td>
<td>Eating</td>
<td>Waiting</td>
<td><strong>Thinking</strong></td>
</tr>
<tr>
<td>Full2</td>
<td>Thinking</td>
<td><strong>Eating</strong></td>
<td>Thinking</td>
<td><strong>Eating</strong></td>
<td>Thinking</td>
</tr>
</tbody>
</table>

Table 16. Diagram of the events

9.4.4 Conclusion on the dining philosophers

This section has shown how a typical client-server application is modelled in STATECRUNCHER, providing a direct comparison with a well-known example in the literature. Both STATECRUNCHER and CSP are amenable to the problem, but the emphasis is different: STATECRUNCHER is a state machine engine providing the white box or black box oracle to tests and does not support calculus manipulations; CSP is a calculus which is used to prove properties of composed systems.

9.4.5 Source listings of models

9.4.5.1 Source listing of the dining philosophers without semaphores [model t4330]

```plaintext
// Module: Philosophers.scs.txt
// Author: Graham Thomason, Philips Digital Systems Laboratories, Redhill
// Date: 18 July, 2003
// Purpose: StateCruncher model: The Dining philosophers [Hoare, p.75]
//
// Copyright (C) 2003 Philips Electronics N.V.

statechart sc(College)

PCO external;
PCO internal;

event P0_Sit, P0_Sit @external;
event P1_Sit, P1_Sit @external;
event P2_Sit, P2_Sit @external;
event P3_Sit, P3_Sit @external;
event P4_Sit, P4_Sit @external;
event P0_PickFork0, P0_PickFork1, P0_PutFork0, P0_PutFork1 @external;
event P1_PickFork0, P1_PickFork2, P1_PutFork1, P1_PutFork2 @external;
```

© Graham G. Thomason 2003-2004

213
event P2_PickFork2, P2_PickFork3, P2_PutFork2, P2_PutFork3 @external;
event P3_PickFork3, P3_PickFork4, P3_PutFork3, P3_PutFork4 @external;
event P4_PickFork4, P4_PickFork0, P4_PutFork4, P4_PutFork0 @external;
event L0_PickFork0, L1_PickFork1, L2_PickFork2, L3_PickFork3, L4_PickFork4 @internal;
event L0_PutFork0, L1_PutFork1, L2_PutFork2, L3_PutFork3, L4_PutFork4 @internal;
event L0_PickFork1, L1_PickFork2, L2_PickFork3, L3_PickFork4, L4_PickFork0 @internal;
event L0_PutFork1, L1_PutFork2, L2_PutFork3, L3_PutFork4, L4_PutFork0 @internal;

set College(Philosophers, Forks)
set Philosophers(PhilO, Phil1, Phil2, Phil3, Phil4)

cluster PhilO (Standing, SittingHungry, OneForkHungry, Eating, OneForkSatiated, SittingSatiated)
  state Standing { P0_Sit->SittingHungry; }
  state SittingHungry { P0_PickFork0 in $$Forks.Fork0.Lying} -> OneForkHungry \ {fire L0_PickFork0;};
  state OneForkHungry { P0_PickFork1 in $$Forks.Fork1.Lying} -> Eating \ {fire L0_PickFork1;};
  state Eating { P0_PutFork0 -> OneForkSatiated \ {fire L0_PutFork0;};}
  state OneForkSatiated { P0_PutFork1 -> SittingSatiated \ {fire L0_PutFork1;};}
  state SittingSatiated { P0_Stand -> Standing;}

cluster Phil1 (Standing, SittingHungry, OneForkHungry, Eating, OneForkSatiated, SittingSatiated)
  state Standing { P1_Sit -> SittingHungry; }
  state SittingHungry { P1_PickFork1 in $$Forks.Fork1.Lying} -> OneForkHungry \ {fire L1_PickFork1;};
  state OneForkHungry { P1_PickFork2 in $$Forks.Fork2.Lying} -> Eating \ {fire L1_PickFork2;};
  state Eating { P1_PutFork1 -> OneForkSatiated \ {fire L1_PutFork1;};}
  state OneForkSatiated { P1_PutFork2 -> SittingSatiated \ {fire L1_PutFork2;};}
  state SittingSatiated { P1_Stand -> Standing;}

cluster Phil2 (Standing, SittingHungry, OneForkHungry, Eating, OneForkSatiated, SittingSatiated)
  state Standing { P2_Sit -> SittingHungry; }
  state SittingHungry { P2_PickFork2 in $$Forks.Fork2.Lying} -> OneForkHungry \ {fire L2_PickFork2;};
  state OneForkHungry { P2_PickFork3 in $$Forks.Fork3.Lying} -> Eating \ {fire L2_PickFork3;};
  state Eating { P2_PutFork2 -> OneForkSatiated \ {fire L2_PutFork2;};}
  state OneForkSatiated { P2_PutFork3 -> SittingSatiated \ {fire L2_PutFork3;};}
  state SittingSatiated { P2_Stand -> Standing;}

cluster Phil3 (Standing, SittingHungry, OneForkHungry, Eating, OneForkSatiated, SittingSatiated)
  state Standing { P3_Sit -> SittingHungry; }
  state SittingHungry { P3_PickFork3 in $$Forks.Fork3.Lying} -> OneForkHungry \ {fire L3_PickFork3;};
  state OneForkHungry { P3_PickFork4 in $$Forks.Fork4.Lying} -> Eating \ {fire L3_PickFork4;};
  state Eating { P3_PutFork3 -> OneForkSatiated \ {fire L3_PutFork3;};}
  state OneForkSatiated { P3_PutFork4 -> SittingSatiated \ {fire L3_PutFork4;};}
  state SittingSatiated { P3_Stand -> Standing;}

cluster Phil4 (Standing, SittingHungry, OneForkHungry, Eating, OneForkSatiated, SittingSatiated)
  state Standing { P4_Sit -> SittingHungry; }
  state SittingHungry { P4_PickFork4 in $$Forks.Fork4.Lying} -> OneForkHungry \ {fire L4_PickFork4;};
  state OneForkHungry { P4_PickFork0 in $$Forks.Fork0.Lying} -> Eating \ {fire L4_PickFork0;};
  state Eating { P4_PutFork4 -> OneForkSatiated \}
9.4.5.2 Source listing of the dining philosophers with semaphores [model t4335]

statechart sc(College)

PCO external;  // For philosopher actions
PCO composition; // For communication from semaphore to philosopher
PCO internal;  // Internal events

event Pang0, Pang1, Pang2, Pang3, Pang4 @external;
event Full0, Full1, Full2, Full3, Full4 @external;
event Req01, Req12, Req23, Req34, Req40 @composition;
event Rel01, Rel12, Rel13, Rel34, Rel40 @composition;
event Acq01, Acq12, Acq23, Acq34, Acq40 @composition;
event Try01, Try12, Try23, Try34, Try40 @internal;
event Pick0, Pick1, Pick2, Pick3, Pick4 @internal;
event Put0, Put1, Put2, Put3, Put4 @internal;

© Graham G. Thomason 2003-2004
set College(Philosophers,Forks)

set Philosophers(PhilO,Phil1,Phi12,Phi13,Phil4)

cluster PhilO(Thinking,Waiting,Eating)
state Thinking {PangO->Waiting {fire ReqO1;};)
state Waiting {AcqO1->Eating;} 
state Eating {upon enter {trace("POEat");} 
upon exit {trace("POStp");} 
FullO->Thinking {fire RelO1;};}

cluster Phil1(Thinking,Waiting,Eating)
state Thinking {Pangl->Waiting {fire Reql2;};}
state Waiting {Acql2->Eating;} 
state Eating {upon enter {trace("PlEat");} 
upon exit {trace("PIStp");} 
Fulll->Thinking {fire Rell2;};}

cluster Phi12(Thinking,Waiting,Eating)
state Thinking {Pang2->Waiting {fire Req23;};}
state Waiting {Acq23->Eating;} 
state Eating {upon enter {trace("P2Eat");} 
upon exit {trace("P2Stp");} 
Full2->Thinking {fire Rel23;};}

cluster Phi13(Thinking,Waiting,Eating)
state Thinking {Pang3->Waiting {fire Req34;};}
state Waiting {Acq34->Eating;} 
state Eating {upon enter {trace("P3Eat");} 
upon exit {trace("P3Stp");} 
Full3->Thinking {fire Rel34;};}

cluster Phi14(Thinking,Waiting,Eating)
state Thinking {Pang4->Waiting {fire Req40;};}
state Waiting {Acq40->Eating;} 
state Eating {upon enter {trace("P4Eat");} 
upon exit {trace("P4Stp");) 
Full4->Thinking {fire Rel40;};}

set Forks(ForkO,Forkl,Fork2,Fork3,Fork4, PairO,Pai12,Pai23,Pai34,Pai40)

cluster ForkO(Lying,Held)
state Lying {PickO->Held;} 
state Held {PutO--Lying;}

cluster Forkl(Lying,Held)
state Lying {Pickl->Held;} 
state Held {Putl--Lying;}

cluster Fork2(Lying,Held)
state Lying {Pick2->Held;} 
state Held {Put2--Lying;}

cluster Fork3(Lying,Held)
state Lying {Pick3->Held;} 
state Held {Put3--Lying;}

cluster Fork4(Lying,Held)
state Lying {Pick4->Held;} 
state Held {Put4--Lying;}

/****[Fork Pair Control]****/ 
cluster PairO1(Reset,Requested)
state Reset {ReqO1[in($Fork0.Lying) & in($Fork1.Lying)] 
{fire Pick0; fire Pick1; fire AcqO1;} 
RelO1 
{fire Put0; fire Put1; fire Try40; fire Try12;} 
ReqO1 [in($Fork0.Lying) || in($Fork1.Lying)] 
->Requested; }

state Requested {TryO1[in($Fork0.Lying) & in($Fork1.Lying)]
-> Reset
(fire Pick0; fire Pick1; fire Acq01;)
}

cluster Pair23(Reset, Requested)

state Reset
{Req23[in($Fork2.Lying) && in($Fork3.Lying)]
(fire Pick2; fire Pick3; fire Acq23;)
Rel23
(fire Put2; fire Put3; fire Try12; fire Try34;)
Req23[ !in($Fork2.Lying) || !in($Fork3.Lying)]
->Requested;
}

state Requested
{Try23[in($Fork2.Lying) && in($Fork3.Lying)]
-> Reset
(fire Pick2; fire Pick3; fire Acq23;)
}

cluster Pair34(Reset, Requested)

state Reset
{Req34[in($Fork3.Lying) && in($Fork4.Lying)]
(fire Pick3; fire Pick4; fire Acq34;)
Rel34
(fire Put3; fire Put4; fire Try23; fire Try40;)
Req34[ !in($Fork3.Lying) || !in($Fork4.Lying)]
->Requested;
}

state Requested
{Try34[in($Fork3.Lying) && in($Fork4.Lying)]
-> Reset
(fire Pick3; fire Pick4; fire Acq34;)
}

cluster Pair40(Reset, Requested)

state Reset
{Req40[in($Fork4.Lying) && in($Fork0.Lying)]
(fire Pick4; fire Pick0; fire Acq40;)
Rel40
(fire Put4; fire Put0; fire Try34; fire Try01;)
Req40[ !in($Fork4.Lying) || !in($Fork0.Lying)]
->Requested;
}

state Requested
{Try40[in($Fork4.Lying) && in($Fork0.Lying)]
-> Reset
(fire Pick4; fire Pick0; fire Acq40;)
}

// ------------------------------[end of module]----------------------------------
10. Experience with STATECRUNCHER and conclusions

The project set out with two experimental goals: (1) to investigate whether an approach to automatic generation of state-based tests of nondeterministic systems using a nondeterministic oracle would offer an improved testing technique, and (2) to see whether PROLOG is a feasible implementation language for such a tool, both from an ease-of-coding viewpoint and from a run-time performance perspective. This section reports on how the testing approach is being pursued within Philips. We illustrate how STATECRUNCHER has been successfully transferred to an end-user within Philips Electronics, with a real example of an embedded software component being tested in a tool chain using STATECRUNCHER as the test oracle. We also review the implementation approach taken. Lastly, we draw a final conclusion.

10.1 Experience at Philips

Software testing as a Research activity was formally transferred from PRL (Philips Research Laboratories - Redhill) to PRI-B (Philips Research India - Bangalore) at the start of 2002. The development of STATECRUNCHER at Redhill, and support to PRI-B continued in 2002 and part of 2003, carried out in the PDSL-R organisation (Philips Digital Systems Laboratories - Redhill).

Philips Research India - Bangalore (PRI-B) has successfully worked with STATECRUNCHER, having integrated it into the TorX tool chain, testing [Koala] components for television systems.

The following figure is by Nitin Koppalkar at PRI-B, who did the integration.

![Figure 170. STATECRUNCHER integrated in the TorX tool chain (Nitin Koppalkar)]
Two components that have been modelled and tested are *TV Program Installation* and the *Last Status Manager*.

**TV Program Installation (modelled by Tim Trew)**

A STATECRUNCHER model has been produced for a component that installs a program in a TV. The sequence of operations is to:

1. Find the carrier
2. Analyse the modulation to find out the TV system (PAL / NTSC / SECAM)
3. Analyse the VBI (vertical blanking interval) data to deduce the station name.

The issues are:

- To use a generic model of the program installation component in any testing configuration or composed-system configuration.
- To obtain all nondeterministic outcomes in the STATECRUNCHER model due to a failure to proceed at any stage.
- To obtain all nondeterministic outcomes in the STATECRUNCHER model due to interleavings of external and internal events.

The Philips report [Trew 03] covers this model, and discusses challenging generic issues in component modelling, such as how to generate interleavings of external and internal events in STATECRUNCHER.

The following model is a simplification of what has been produced. A more extensive model contains details of the tuner.
Figure 171. Program Installation, simplified, (Tim Trew) [model t4410]
Points to note:

- The clock generates tick events, to which the programinstallation states respond, forking on alternatives where they exist.
- The programinstallation area fires a tock after any response to a tick, in order to keep the clock going.
- This clock does not need to limit the number of ticks fired, as the programinstallation is not capable of infinite cycling.
- Race nondeterminism is used to generate interleavings of external events (PCO_...) and internal events (tick). This covers situations where an external event is given, but is preempted by an internal event.
- Fork nondeterminism is used to continue or terminate the clock at every step.

Performance is acceptable: on a 300MHz machine, it takes about 2 seconds to process

```
PCO_pgins_startmanualinstallation
```

...giving 9 worlds.

Output after event `PCO_pgins_startmanualinstallation` (9 worlds generated).

<table>
<thead>
<tr>
<th>Wld</th>
<th>program instaln.</th>
<th>Clock</th>
<th>Trace (read in reverse order)</th>
<th>Remarks on program installation</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>idle</td>
<td>idle</td>
<td>[tick, tick/pgins_onstationnotfound, tick in searching, firing tick, PCO_pgins_startmanualinstallation clock, pginsN_onmanualinstallationstarted, pgins_startmanualinstallation in idle, PCO_pgins_startmanualinstallation executed]</td>
<td>Searching did not find a station.</td>
</tr>
<tr>
<td>20</td>
<td>idle</td>
<td>active</td>
<td>[firing tick, tick, tick/pgins_onstationnotfound, tick in searching, firing tick, PCO_pgins_startmanualinstallation clock, pginsN_onmanualinstallationstarted, pgins_startmanualinstallation in idle, PCO_pgins_startmanualinstallation executed]</td>
<td>Searching did not find a station. There was an extra tick, with no response.</td>
</tr>
<tr>
<td>24</td>
<td>tuned</td>
<td>idle</td>
<td>[tick, tick/pginsN_onstationfound, tick in searching, firing tick, PCO_pgins_startmanualinstallation clock, pginsN_onmanualinstallationstarted, pgins_startmanualinstallation in idle, PCO_pgins_startmanualinstallation executed]</td>
<td>Searched and found a station. Did not proceed to detect the TV system.</td>
</tr>
<tr>
<td>32</td>
<td>Tv System Detected</td>
<td>idle</td>
<td>[tick, tick/pginsN_onTvSystemDetected, tick in tuned, firing tick, tock, tick/pginsN_onstationfound, tick in searching, firing tick, PCO_pgins_startmanualinstallation clock, pginsN_onmanualinstallationstarted, pgins_startmanualinstallation in idle, PCO_pgins_startmanualinstallation executed]</td>
<td>Searched, found a station and detected the TV system. Did not proceed to find station name.</td>
</tr>
<tr>
<td>40</td>
<td>idle</td>
<td>idle</td>
<td>[tock, tick/pginsN_onStationNameFound, tick in TvSystemDetected, firing tick, tock, tick/pginsN_onTvSystemDetected, tick in tuned, firing tick, tock, tick/pginsN_onStationNameFound, tick in searching, firing tick, PCO_pgins_startmanualinstallation clock, pginsN_onmanualinstallationstarted, pgins_startmanualinstallation in idle, PCO_pgins_startmanualinstallation executed]</td>
<td>A complete cycle through searching, finding a station, detecting the TV system and finding the station name.</td>
</tr>
<tr>
<td>43</td>
<td>idle</td>
<td>active</td>
<td>[firing tick, tock, tick/pginsN_onStationNameFound, tick in TvSystemDetected, firing tick, tock, tick/pginsN_onTvSystemDetected, tick in tuned, firing tick, tock, tick/pginsN_onstationfound, tick in searching, firing tick, PCO_pgins_startmanualinstallation clock, pginsN_onmanualinstallationstarted, pgins_startmanualinstallation in idle, PCO_pgins_startmanualinstallation executed]</td>
<td>A complete cycle with an extra tick, to which there was no response.</td>
</tr>
<tr>
<td>44</td>
<td>searching</td>
<td>idle</td>
<td>[PCO_pgins_startmanualinstallation clock, pginsN_onmanualinstallationstarted, pgins_startmanualinstallation in idle, PCO_pgins_startmanualinstallation executed]</td>
<td>Searching, with no further progress.</td>
</tr>
<tr>
<td>55</td>
<td>searching</td>
<td>idle</td>
<td>[pginsN_onmanualinstallationstarted, pgins_startmanualinstallation in idle, PCO_pgins_startmanualinstallation executed, PGG^pgins_startmanualinstallation clock]</td>
<td>Searching, with no further progress. Differs from world 44 because of the race (clock wins).</td>
</tr>
<tr>
<td>61</td>
<td>searching</td>
<td>active</td>
<td>[pginsN_onmanualinstallationstarted, pgins_startmanualinstallation in idle, PCO_pgins_startmanualinstallation executed, firing tick, PCO_pgins_startmanualinstallation clock]</td>
<td>Searching, with clock winning a race and doing nothing.</td>
</tr>
</tbody>
</table>

Table 17. Program Installation results

After the traces have been cleared, there are 6 residual worlds. Then event PCO_pgins_stopmanualinstallation can be given, generating 24 worlds (in about 15 seconds on a 300 MHz machine). Space does not permit us to tabulate the results, but we remark that on stopping the installation, a race is run on two transitions on PCO_stopmanualinstallation, generating interleavings of events pgins_stopmanualinstallation and tick. The tick first situation could represent a user stopping the installation, but just before the command is seen, the installation completes.

Model listing

```
// Author: Tim Trew
// Test of transition algorithm for clock ticking - can we interleave
// all "wait" events with external events?

// User enters
// SC: pe [PCO_pgins_startmanualinstallation, [composition, sc]]
```
// SC: ct
// SC: pe [PCO_pgins_stopmanualinstallation, [composition, sc]]

statechart sc(composition)

set composition (programinstallation, 
    controllable_function_handler, 
    notification_handler)

cluster programinstallation (idle, searching, tuned, TvSystemDetected)

/* Program Installation Provided functions */
event composition%%pgins_startmanualinstallation;
event composition%%pgins_stopmanualinstallation;

/* Program Installation notifications */
event composition%%pginsN_onmanualinstallationstarted;
event composition%%pginsN_onmanualinstallationcompleted;
event composition%%pginsN_onmanualinstallationstopped;
event composition%%pginsN_onsearchinprogress;
event composition%%pginsN_onstationfound;
event composition%%pginsN_onstationnotfound;
event composition%%pginsN_onTvSystemDetected;
event composition%%pginsN_onStationNameFound;

state idle {
    pgins_startmanualinstallation -> searching
    {trace("pgins_startmanualinstallation in idle");
    fire pginsN_onmanualinstallationstarted ; ;
    pgins_stopmanualinstallation
    {trace("pgins_stopmanualinstallation in idle - ignored") ; ;

state searching {
    pgins_startmanualinstallation
    {trace("pgins_startmanualinstallation in searching - ignored") ; ;
    pgins_stopmanualinstallation -> idle
    {trace("pgins_stopmanualinstallation in searching");
    fire pginsN_onmanualinstallationstopped ; ;
    tick -> tuned {trace("tick/pginsN_onstationfound");
    fire pginsN_onstationfound; fire tock; } ;
    tick -> idle {trace("tick/pgins_onstationnotfound");
    fire pginsN_onstationnotfound; fire tock; } ;

state tuned {
    pgins_startmanualinstallation
    {trace("pgins_startmanualinstallation in tuned - ignored") ; ;
    pgins_stopmanualinstallation -> idle
    {trace("pgins_stopmanualinstallation in tuned");
    fire pginsN_onmanualinstallationstopped ; ;
    tick -> TvSystemDetected{trace("tick/pginsN_onTvSystemDetected");
    fire pginsN_onTvSystemDetected;
    fire tock ; ;

state TvSystemDetected {
    pgins_startmanualinstallation
    {trace("pgins_startmanualinstallation in TvSystemDetected - ignored") ; ;
    pgins_stopmanualinstallation -> idle
    {trace("pgins_stopmanualinstallation in TvSystemDetected");
    fire pginsN_onmanualinstallationstopped ; ;
    tick -> idle{trace("tick/pginsN_onStationNameFound");
    fire pginsN_onStationNameFound; fire tock ; ;

// provides functions

© Graham G. Thomason 2003-2004 223
set controllable_function_handler (prov_fun, clock)
event composition%%PCO_pgins_startmanualinstallation;
event composition%%PCO_pgins_stopmanualinstallation;

event composition%%tock;
event composition%%tick, StartClock;

state prov_fun {
  PCO_pgins_startmanualinstallation
  {trace("PCO_pgins_startmanualinstallation executed") ;
   fire pgins_startmanualinstallation ;};
  PCO_pgins_stopmanualinstallation
  {trace("PCO_pgins_stopmanualinstallation executed") ;
   fire pgins_stopmanualinstallation ;};
}

cluster clock (clockidle, starting, clockactive) {
  PCO_pgins_startmanualinstallation -> clock -> clock.starting
  {trace("PCO_pgins_startmanualinstallation clock") ;};
  PCO_pgins_stopmanualinstallation -> clock -> clock.starting
  {trace("PCO_pgins_stopmanualinstallation clock") ;};
}

state clockidle;

state starting {
  upon enter { fire StartClock; }
  StartClock -> clockidle;
  StartClock -> clockactive;
}

state clockactive {
  upon enter {
    trace("firing tick");
    if (in (: : composition.programinstallation.searching))
      {trace("tick in searching"); }
    if (in (: : composition.programinstallation.tuned))
      {trace("tick in tuned"); }
    if (in (: : composition.programinstallation.TvSystemDetected))
      {trace("tick in TvSystemDetected"); }
    fire tick;
  }
  /* Fork non-determinism to terminate at every possible step. */
  tock -> clock -> clockactive {trace("tock");};
  tock -> clockidle {trace("tock");};
}

cluster notification_handler (notif_handler)
/* Turned fired notifications in to traces */
event composition%%pginsN_onchannelfound;

state notif_handler {
  pginsN_onchannelfound -> notif_handler
  {trace ("pginsN_onchannelfound"); }
  pginsN_onmanualinstallationstarted -> notif_handler
  {trace ("pginsN_onmanualinstallationstarted"); }
  pginsN_onmanualinstallationcompleted -> notif_handler
  {trace ("pginsN_onmanualinstallationcompleted"); }
  pginsN_onmanualinstallationstopped -> notif_handler
  {trace ("pginsN_onmanualinstallationstopped"); }
  pginsN_onsearchinprogress -> notif_handler
  {trace ("pginsN_onsearchinprogress"); }
}
The more extensive model (including the tuner) has been integrated into the TorX tool chain by Nitin Koppalkar at PRI-B. The following diagram, by Nitin Koppalkar, shows the tool chain in action:

Figure 172. STATECRUNCHER and TorX in action (Nitin Koppalkar)
**Last Status Manager:** Currently (November 2003), PRI-B is working on testing this module, which manages status information, writing it at intervals to non-volatile memory (NVM). At any time, the cache can contain messages that have been written to NVM and messages that still have to be written to NVM, under the constraint that if a message has been written to NVM, all older messages must have also been written to NVM. Later messages may or may not be in NVM, hence nondeterminism. It was considered useful to have an array facility to handle the messages in chronological order. It was to meet the needs of this system that arrays were implemented in STATECRUNCHER (in Release 1.04).

**Outcomes of the trials of STATECRUNCHER**

We have shown that STATECRUNCHER has been successfully deployed in a live project. The experience of this trial clearly demonstrated STATECRUNCHER's ability to handle all the forms of nondeterminism that were inherently present in the system under test. The successful outcome of these trials has led to a number of reports and continued work using STATECRUNCHER. The following reports have been written or are nearing completion:

On integrating STATECRUNCHER into the TorX tool chain [Koppalkar 02, 03]:
- Nitin Koppalkar and Animesh Bhowmick
  Integration of Generic Explorer with the TorX Tool Chain

- Nitin Koppalkar
  Interfacing STATECRUNCHER with TorX for demonstrating the state-based testing technique taking MG-R components for a case study

On modelling software components in STATECRUNCHER [Trew 03]:
- Tim Trew
  State-based modelling of software components for integration testing
  A practical guide to the creation of STATECRUNCHER models

We indicate some future directions at the end of this section.

**10.2 PROLOG as the implementation language**

There is of course a subjective element in stating whether PROLOG is a feasible implementation language for any given purpose. Different people show affinity to different programming languages, and few can claim competence in a really wide range of them. The present author's view is that to build the same STATECRUNCHER system in C would require a significant multiple of the effort taken, although such an undertaking by a team, given the present implementation as a precise specification, would not be pointless, as it would lead to
improved performance and greater maintainability in an organisation, because one could then
tap into a wider pool of programmers than is the case with a PROLOG implementation. To
use an object oriented language could help in many ways, but the hard parts of the transition
algorithm are not clearly amenable to an object-oriented approach.

**Strengths of PROLOG as a programming language**

In the author's estimation, the power of PROLOG (for readers not entirely unfamiliar with
PROLOG), lies in the following features:

- **Compact notation.** Although this is arguably a very superficial aspect, it does make for
readable programs. They can be overseen with more ease because there is less syntactic
overhead (compare the abundant use of brackets in LISP). Examples:
  - Variables have no declaration and their scope is just the one clause they are used in.
    Symbols beginning with capitals or underscore are variables, and are distinct from
    those beginning with lower case letters which are *atoms*, i.e. constants. The *and*
    operator is a comma, and the *or* operator is a semicolon. The result approaches the
    compactness of the notation for predicates and specifications in discrete mathematics.
  - The notation \([H \mid T]\) denotes the head and tail of a list. The head is one element of a
    list and the tail is conventionally zero or more elements of the list. The term \([H \mid T]\)
    will construct a list from a head and a tail, or split a list into head and tail, or it can be
    used to check whether an item is a list with at least one element and some tail, (which
    may be the empty list).

- **Typelessness.** The fact that PROLOG is untyped makes many routines very general,
where in C many versions of a function might be needed, one for each type of argument,
though this is less of a problem in C++, where a `template` construction can be used.

- **The interpretative nature.** Programs, whether large or very small (e.g. just one *clause*)
can be experimented with at the command prompt. PROLOG programs have no header
files and compile so fast there is no need for a developer to *build* them, as in non-
interpretative languages. The whole of STATECRUNCHER compiles in little more than a
second on a modern computer.

- **Unification.** This allows a partially grounded structure to be matched against another one,
e.g. \([a, [B, C]]\) against \([D, [e \mid T]]\). A variable matched against a grounded item is
*instantiated* to that item and becomes grounded. The above match succeeds with
  
  \[
  \begin{align*}
  B &= e \\
  C &= _G163 \\
  D &= a \\
  T &= [_G163]
  \end{align*}
  \]

  This sort of match is useful e.g. in extracting parts of compiled statements, such as the
condition of a transition, where the parse contains structures partly labelled by fixed
atoms, with the remaining parse body representing the real parse content to be extracted.
The result of the unification may still contain non-ground terms, as variable T is in the above example, though it is constrained to be a list of one element.

- **Backtracking.** This is a search mechanism that will look for a structural match, and satisfaction of further constraints. An example of use might be to find a parsed statement satisfying a certain constraint, such as finding a declaration of a variable of a certain name, or finding a potential transition, then requiring that it satisfy various conditions. An extension to backtracking is to ‘find all’ items satisfying some constraint. Backtracking is also a good mechanism for generating many solutions to some requirement, such as permutations.

- **Reversibility.** PROLOG clauses can be written to work in two directions - indeed they will do automatically in many cases, perhaps without the program author realising it. The same simple PROLOG clauses defining how to append two lists \( L1 \) and \( L2 \) making \( L3 \), can also break up a given list \( L3 \) into sublists \( L1 \) and \( L2 \) which when appended, make the given list. It will do this in all possible ways, e.g. \([a, b, c]\) can be split this way into:
  - \( L1 = [], \quad L2 = [a, b, c] \)
  - \( L1 = [a], \quad L2 = [b, c] \)
  - \( L1 = [a, b], \quad L2 = [c] \)
  - \( L1 = [a, b, c], \quad L2 = [] \)

In fact the append clause can work with three instantiated parameters to verify that \( L1 \) and \( L2 \) append into \( L3 \), and even with only \( L1 \) instantiated or only \( L2 \) instantiated or even more unusually with all three parameters uninstantiated.

- **The Definite Clause Grammar (DCG).** This is very convenient way of expressing Backus-Naur grammar rules and recording a parse for them. It is based on processing list structures by specifying what part of a list is used up in the parse, and what part is returned as unused, available for the next term in a grammar rule. It is described very lucidly in [Clocksin]. The implementation of STATECRUNCHER's expression parser shows that use of DCGs is feasible on a large scale (about 20 operator precedences), provided care is taken to maintain efficiency.

**PROLOG's run-time performance**

There are two parts to PROLOG's execution performance: compilation and the run-time engine. Although PROLOG's Definite Clause Grammar is well-known for its parsing capability, it is probably for performance reasons that it is not more widely used for full domain-specific-language systems. However, the compilation speed of a STATECRUNCHER model is very acceptable, good even, on a modern (3GHz) machine, where typical illustrative models (as in [StCrManual]), compile in a second or so. Compilation, especially of expressions is certainly felt to be an area where, with more analysis and profiling, the performance could be improved further.
The stress tests in [StCrTest] show that performance is generally good, but with nondeterminism, there are, and always will be, cases of combinatorial explosion. In deterministic situations, STATECRUNCHER is fast, by human standards, in all models investigated, including automatically generated large ones.

There are differences between different PROLOG implementations, but the author has been very satisfied with the two chosen for the investigation: [SWI Prolog], which is in the public domain, and [WinProlog], a commercial system. There are not great differences in execution speed, although it can be remarked that the difference between running the WinProlog system as an MS-DOS executable and running in the development environment gives a factor of 2 or 3 difference in performance.

10.3 Future directions

Future directions can be seen in tooling and in testing.

10.3.1 The tooling side

Possible enhancements to STATECRUNCHER

Philips Research has expressed interest in extending STATECRUNCHER with machine implantation, whereby state machine templates can be dynamically implanted into a statechart, as described in [StCrFunMod]. This makes whole statecharts recursive, and would solve the problem of how to model (indirectly) synchronous and asynchronous recursive function calls.

A less drastic enhancement to STATECRUNCHER is to implement UML pseudo-states, though these can be simulated with the existing features. Ideally, STATECRUNCHER would keep pace with all developments in UML, as this is becoming the industry standard.

Other possible enhancements are: to support forward chaining of data and lambda transitions (i.e. transitions that take place when some boolean expression becomes true) and to combine cause effect graphing with statecharts.

STATECRUNCHER's performance

STATECRUNCHER has been subjected to some stress tests, described in detail in [StCrTest]. Some models of regular structure but arbitrary size can be generated by PROLOG programs. Examples are: broad clusters, deep clusters, broad sets, deep sets, intensive nondeterminism, and long chains of fired events. Response times for processing an event as given below are for STATECRUNCHER running under [SWI-Prolog] on a 300 MHz machine. More modern machines can give a factor 10 improvement.
STATECRUNCHER almost always performs well with deterministic models (i.e. no forks in the model, and with race and set transit nondeterminism disabled). Examples:

- Test model t7110, with 25 clusters of 25 leaf-states (625 leaf-states in total), executes a leafstate-to-leafstate transition in 1 second and a cluster-to-cluster one in 2.5 seconds.
- Test model t7120, containing a set of 5 sets each with 5 member clusters of 2 leaf-states, executes an event causing transitioning in all 25 clusters in 1.8 seconds.
- Test model t7180 executes a chain of 25 fired events across 25 members of a set in 1.7 seconds.

In nondeterministic situations, models with a few tens of worlds generally perform adequately. The Program Installation example (Figure 171) performs well. With larger numbers of worlds (say 100), performance can become a bottleneck, though models have been run leading to world numbers in the thousands after very few events. Set nondeterminism with nested sets appears to degrade performance considerably.

Approaches to increasing STATECRUNCHER's performance

What options are there for performance improvements? We consider some:

- **Re-write the program in ‘C’.** ‘C’ is a compile-to-executable (non-interpretative) language which facilitates very precise control over all algorithms, including memory allocation. A disadvantage of this approach is that it would probably be a very time-consuming exercise, though the existence of the PROLOG implementation would provide an unambiguous specification, and would give much guidance on implementation strategy.

- **Write critical inner loops in ‘C’**. One would profile the execution of the PROLOG version to find the critical inner loops. Profiling utilities and an interface mechanism to external code exist for most PROLOG systems. This approach could be very effective, but it is PROLOG-implementation specific. It could be that what is critical to one PROLOG engine is not critical to another. Also, the external interface mechanisms are liable to be specific to the PROLOG system used.

- **Write one's own subset of PROLOG in ‘C’**. By implementing some PROLOG operations as ‘C’ routines, especially list operations, one might be able to produce a system that generally makes use of the existing PROLOG structure, whilst benefiting from the efficiency and controllability of ‘C’.

- **Investigate other PROLOG engines**. There are many suppliers of PROLOG systems. STATECRUNCHER already runs under two PROLOG systems, [SWI-Prolog] and [WinProlog]. This means that a framework for further porting is already in place, with many system-dependent predicates already implemented in a compatibility library. The test suite, (mentioned in section 9.3) would help drive the porting process: once all tests run, the serious porting work is likely to be complete.
• **Tweak PROLOG garbage collection.** A weakness of PROLOG as a programming language could be that the user does not have adequate control over memory management. The garbage collection algorithm used may not be known. However, most PROLOG systems offer the possibility to make extra garbage collection calls. A few experiments have been done with this, but so far no significant improvements have been observed.

• **Tailor the coding style to a particular PROLOG engine.** Some PROLOG suppliers offer guidance on how to write efficient code, though what is good for one system may be bad for another. A case in point is whether to be liberal or sparing with the use of the PROLOG cut. One might think that putting in a redundant cut at the end of a deterministic predicate helps a PROLOG engine, enabling it to recover many stack frames, but it may impede it. This may be because it interferes with tail recursion optimisation, where a recursive call at the end of a predicate is executed at the caller's level, rather than by creating a new calling level. A few experiments with removing cuts in the process set of task sequences in worlds routine shows that memory requirements become very different (e.g. stack space is traded for heap space), but that there is no drastic performance or capacity change. Another aspect to tailoring code is to make use of supplied library functions rather than one's own generic implementations.

• **Algorithmic experimentation.** The transition algorithm was described with various algorithmic alternatives, such as the algorithm A / algorithm B options in the main process set of task sequences in worlds routine. It could be that a better choice can be found.

• **Write a front-end cache to STATECRUNCHER** that pre-explores the state space when the IUT is not executing under real-time constraints, so that when the IUT is executing under real-time constraints, a rapid-response test oracle can be given.

• **Make use of parallel processing** (e.g. a processor per world). This would be easier at a macroscopic level (allocating each extant world visible at user-event processing time to a processor) than at a microscopic level (allocating each extant world visible at internal-event processing time to a processor). As the number of worlds may be larger than the number of processors, some form of dynamic allocation of tasks would be required.

The above list gives many options, but it must be remembered that performance optimisation is in competition with pressure for new features (e.g. as mentioned in this subsection). Moreover, STATECRUNCHER is in competition for resources with the other elements of the tool chain. Should more effort be spent on test generation? Priorities are often determined by the customer.

**Perspectives for on-the-fly testing and test generation**

There is scope for research into advanced primers (test generators), performing intelligent transition tours and disambiguating IUT states under nondeterminism. STATECRUNCHER at
least enables flattening of nondeterministic UML statecharts, and may be useful for other transformations, e.g. finding an observable NFSM (Nondeterministic Finite State Machine) that is equivalent to an unobservable one (observable means that outputs on transitions reveal the new state). STATECRUNCHER could have a role to play as an experimental vehicle for advanced on-the-fly testing (Lee's adaptive testing) algorithms, which can be more efficient than off-line generated batch tests (Lee's preset testing). For example, the homing problem (see [Lee], p.1095) consists of determining the final state of a machine by giving it a sequence of events and observing the outputs. With on-the-fly testing the homing sequence can be shorter than in the batch case. However, homing (which drives the machine into a known state following on from a test) is weaker than distinguishing or verifying or identifying the state after the test, but on-the-fly testing helps here too [Lee, p.1097, p.1105], [Hierons 98]. STATECRUNCHER's command language offers efficient hooks needed by the test generation or other programs.

10.3.2 The testing side

Practical problems being tackled

STATECRUNCHER has been the test oracle tool on which state-based testing at Philips has been focussed for well over a year. The strength of STATECRUNCHER is seen as being in its UML-friendly and intuitive syntax, and its ability to handle nondeterminism, which was the motivation for its development. Other strengths are its support for scoping operators and its after-landing transition semantics, both of which facilitate component composition.

PRI-B has shown itself able to use STATECRUNCHER in an advanced testing environment. STATECRUNCHER has been integrated into an end-to-end tool chain, based on TorX, using EXPECT scripts to adapt STATECRUNCHER's interface to that required by TorX. Various components have been selected for modelling and testing.

It has been found that creating some dynamic models from a conventional specification is a particularly skilled task. Part of the difficulty is that this needs to be done in a way that enables component model composition to follow the mechanism of component composition.

The challenges have been successfully met, and as they have revealed additional needs in STATECRUNCHER, (a socket interface, pruning of worlds on invalid traces, arrays) these have been supplied. The testing activities have also exposed some new problems, in particular the issue of how to handle large numbers of notifications (asynchronous messages) without creating a STATECRUNCHER world for each potential number of notifications.

It is intended to complete this phase of trialling with STATECRUNCHER in 2004. There are plans to make a comparison with another product, Conformiq, of Finnish manufacture. The results of the comparison should be available in the course of 2004.
10.4 Final conclusion

We have presented a state machine system that handles nondeterminism for the purpose of providing a test oracle in a tool chain. It has successfully been transferred to Philips Research India - Bangalore for use on live projects, where it has been deployed for testing of embedded software components with inherent nondeterminism. The successful outcome of these trials to date has led to ongoing use of STATECRUNCHER in testing research within the Philips Electronics organization. We believe that one of the main contributions of this thesis has been to take a research concept from inception through to deployment in an industrial setting.
11. Glossary and abbreviations etc.

11.1 Greek letters

For compactness, and as in [CHSM], we will often use Greek letters for event names; in the STATECRUNCHER source, these would be spelled out in Roman letters. The English names of the letters are as follows:

<table>
<thead>
<tr>
<th>Greek Letter</th>
<th>English Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>α alpha</td>
<td>β beta</td>
</tr>
<tr>
<td>ε epsilon</td>
<td>ζ zeta</td>
</tr>
<tr>
<td>γ gamma</td>
<td>δ delta</td>
</tr>
<tr>
<td>ι iota</td>
<td>κ kappa</td>
</tr>
<tr>
<td>ν nu</td>
<td>ξ xi</td>
</tr>
<tr>
<td>ρ rho</td>
<td>σ sigma</td>
</tr>
<tr>
<td>φ phi</td>
<td>χ chi</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Greek Letter</th>
<th>English Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>θ theta</td>
<td>ι iota</td>
</tr>
<tr>
<td>μ mu</td>
<td>ν upsilon</td>
</tr>
<tr>
<td>π pi</td>
<td>ω omega</td>
</tr>
</tbody>
</table>

Table 18. Greek letters

11.2 Glossary and abbreviations

**Action:** A STATECRUNCHER term for processing that is associated with a transition (or the entering/exiting of a state). An action can be e.g.

- a ‘C’-like assignment to a variable
- the firing of an event
- the generation of output (a trace).

**Black-box testing:** Testing where system outputs can be observed, but not system internals. In the case of state-based testing, the state (more precisely, configuration) of the system will not be directly observable, and must be deduced from traces (outputs generated when events are processed).

**Broadcast-event:** An event that is generated within a statechart which can be responded to by the model (transitions can be triggered by it). The STATECRUNCHER keyword to generate a broadcast event is fire event.
**Broadcast-event nondeterminism:** Also known as fired-event nondeterminism, this is the form of nondeterminism that arises when an action associated with a transition fires an event, which in turn gives rise (directly or indirectly) to one of the other forms of nondeterminism (e.g. fork, race, set-transit).

**CCS:** The Calculus of Communicating Systems. A process calculus defined by Robin Milner.

**CHSM:** Concurrent Hierarchical finite State Machine. A language implemented by Paul J Lucas [CHSM].

**Cluster:** A hierarchical state and component of a statechart with the understanding that if the cluster is occupied, exactly one of its members must be occupied. It is the XOR-state of Harel.

**Configuration:** The dynamic state of a statechart in a broad sense, comprising: occupancy (occupied/vacant) of the states in the statechart, variable values, cluster history, and trace values.

**CSP:** Communicating Sequential Processes. A process calculus defined by C.A.R. Hoare.

**DCG:** Definite Clause Grammar. This is the standard PROLOG grammar notation, which enables grammar rules to be written in Backus-Naur form.

**Event:** A signal (that has no time duration) which may be responded to in a statechart model by the triggering of transitions.

**Fire:** The act of generating an event in an action associated with a transition: “the action fires the event”. [Compare “triggering a transition”, which may take place when the fired event is processed].

**Fired-event nondeterminism:** Also known as broadcast-event nondeterminism, this is the form of nondeterminism that arises when an action associated with a transition fires an event, which in turn gives rise (directly or indirectly) to one of the other forms of nondeterminism (e.g. fork, race, set-transit).

**Fork nondeterminism:** The form of nondeterminism that arises when an event triggers mutually exclusive transitions in the statechart, and which produce a different outcome.
FSM: Finite state machine. We normally mean a flattened state machine of the Mealy type that produces observable outputs on transitions.

GP4: Generic Prolog Parsing and Prototyping Package. An underlying layer of PROLOG programs to provide parsing support (especially tokenization and expression parsing).

GUI: Graphical User Interface.

Harness: A test harness is a tool that contains or accesses a test script so as to obtain tests and their oracle, and communicates with an implementation under test to run the tests. It compares actual with expected output, and logs the results as pass or fail.

IUT: Implementation Under Test.

Leafstate: A state and a component of a statechart at the lowest hierarchical level.

LHS: Left Hand Side.

Machine engine: A program that holds a representation of a statechart and a configuration of that statechart, and which can process an event and in so doing calculate and assume the new configuration.

Meta-event: An event that is internally generated when a state is exited or entered, and which can be used to trigger transitions in other parts of the statechart.

NFSM: Nondeterministic Finite State Machine.

Nondeterminism: Dynamic behaviour of a system whereby there is more than one outcome of processing an event. Distinguishing aspects of an outcome are: state occupancy, cluster history, variable values, and traces. For a formal definition of a nondeterministic finite state machine, see section 7.1.

ONFSM: Observable Nondeterministic Finite State Machine. For ONFSMs, a unique target state on a transition can be deduced from the output generated by the transition.
Oracle: The pre-determined output of the system on a successful test, for comparison purposes with the actual output.

PCO: Point of Control and Observation. These are used for systems such as networked and client-server systems where inputs and outputs must be partitioned according to which separate testing point can provide and observe them.

Primer: The TorX terminology for the part of the tool chain that decides what events (or transitions) are to be given to the explorer and indirectly to the implementation under test to be processed.

Race nondeterminism: The form of nondeterminism that arises when an event triggers transitions in parallel parts of the statechart, and when the order in which these events are processed will affect the outcome.

RHS: Right Hand Side.

Set: A state and a component of a statechart with the understanding that if the set is occupied, all its members must be occupied. This represents the parallelism of a model. It is the AND-state of Harel.

Set-action nondeterminism: The form of nondeterminism that arises when actions (such as variable assignments) in different members of a set are executed, when the order in which this happens affects the outcome.

Set nondeterminism: A generic term for set-transit nondeterminism, set-action nondeterminism and set meta-event nondeterminism.

Set-meta-event nondeterminism: The form of nondeterminism that arises when elements of a set are exited or entered, (generating enter and exit meta-events), when the order in which this happens affects the outcome.

Set-transit nondeterminism: The form of nondeterminism that arises when a set is exited or entered, when the order in which the members are exited or entered affects the outcome.

SRT: State Relation Table. A table relating input states to output states via events.
State: This word is used in two senses according to the context
- a statechart consists of a hierarchy of states, which may be sets, clusters, or leaf-states
- a state is the occupancy (occupied/vacant) of a state in the above sense.

Statechart: A concurrent, hierarchical representation of a dynamic behaviour model consisting of states, events, transitions, and optionally variables and statements for processing them.

STATECRUNCHER: A provisional name for a program that compiles statecharts, process events, and provide state or trace information.

SUT: System Under Test.

Trace: The output generated on processing an event (or transition), corresponding to the expected observable output of the Implementation Under Test.

Transition: The relation between the state of a system before and after that system has processed any event that triggers that transition.

Trigger: The act of responding to an event by processing an associated transition: "the event triggers the transition". [Compare "firing an event", which may take place as an action on the transition].

UML: Universal Modelling Language, as set out by the Object Modelling Group. UML is the industry standard for various modelling views on a system. The dynamic modelling view uses statecharts.

White-box testing: Testing where system internals can be observed. In the case of state-based testing, the state (more precisely, configuration) of the system can be observed directly.
12. References

STATECRUNCHER documentation and papers by the present author

Main Thesis   [StCrMain] The Design and Construction of a State Machine System that Handles Nondeterminism

Appendices
Appendix 1   [StCrContext] Software Testing in Context
Appendix 2   [StCrSemComp] A Semantic Comparison of STATECRUNCHER and Process Algebras
Appendix 3   [StCrOutput] A Quick Reference of STATECRUNCHER's Output Format
Appendix 4   [StCrDistArb] Distributed Arbiter Modelling in CCS and STATECRUNCHER - A Comparison
Appendix 5   [StCrNim] The Game of Nim in Z and STATECRUNCHER
Appendix 6   [StCrBiblRef] Bibliography and References

Related reports
Related report 1   [StCrPrimer] STATECRUNCHER-to-Primer Protocol
Related report 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

Note: For a separate annotated bibliography, where references have been structured into categories, see [StCrBiblRef], listed above. The references below are those specifically referred to in this thesis.

[BCS-SIGIST] Standard for Software Component Testing
*British Computer Society* - Special Interest Group in Software Testing

[Beizer] B. Beizer
*Software Testing Techniques*

[Bérard] B. Bérard
*Systems and Software Verification*

[Beveridge] Jim Beveridge and Robert Wiener
*Multithreading Applications in Win32. The Complete Guide to Threads*

[Callahan] John R. Callahan
http://www.cs.wvu.edu/~callahan/interests.html
http://www.ivv.nasa.gov

[CdR] Côte de Resyste (delivers the TorX tool)
http://fmt.cs.utwente.nl/CdR

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

[Chow] Tsun S. Chow
Testing Software Design Modeled by Finite-State Machines
Paul J. Lucas
An Object-Oriented System for Implementing Concurrent, Hierarchical, Finite State Machines.
MSc. Thesis, University of Illinois at Urbana-Champaign, 1993

W.F. Clocksin & C.S. Mellish
Programming in Prolog, 2nd Edition
Springer Verlag, 1984.

The Edinburgh Concurrency Workbench
http://www.dcs.ed.ac.uk/home/cwb/

Anton T. Dahbura, Krishnan K. Sabnani, and M. Ümit Uyar
Formal Methods for Generating Protocol Conformance Test Sequences
Proceedings of the IEEE, Vol. 78, No. 8, August, 1990

CMMI-SE/SW, Version 0.2b, Sept 1999
Capability Maturity Model - Integrated Systems/Software Engineering
CMMI website: http://www.sei.cmu.edu/cmmi/cmmi.html

http://www.conformiq.com

P.A. Darnell and P.E. Margolis
C: A Software Engineering Approach

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

Lydie Du Bousquet, Solofo Ramangalahy, Séverine Simon, César Viho
Formal Test Automation: The Conference Protocol with TGV/TorX
Available on the web at the [TorX] site.

M.J. Hollenberg
Extended Hierarchical Concurrent State Machines, Syntax and Semantics
[Farchi] E. Farchi, A. Hartman, S.S. Pinter
Using a model-based test generator to test for standard conformance

[Friedman] Galit Friedman, Alan Hartman, Kenneth Nagin, Tomer Shiran
Projected State Machine Coverage
IBM Haifa Research Laboratory
Presentation ISSTA 2002

[Harel] D. Harel et al.
On the Formal Semantics of Statecharts
*Logic in Computer Science*, 2nd Annual Conference, 1987, pp.54-64

[Hennie] F.C. Hennie
Fault Detecting for Sequential Circuits

[Hierons 98] Rob M. Hierons
Adaptive testing of a deterministic implementation against a nondeterministic finite state machine

[Hierons 03] Rob M. Hierons
Generating candidates when testing a deterministic implementation against a non-deterministic finite state machine

[Hoare] C.A.R. Hoare
*Communicating Sequential Processes*

[Hollenberg] M.J. Hollenberg
Test Templates for PHACT
*Philips Nat. Lab. Technical Note 152/99*
[Hong] Hyoung Seok Hong, Jeong Hyun Kim, Sung Deok Cha and Yong Rae Kwon
Static Semantics and Priority Schemes for Statecharts
Proceedings of COMPSAC '95, IEEE Computer Society Press.


[ISO 9646-1] International Organization for Standardization
Information technology -- Open Systems Interconnection -- Conformance
testing methodology and framework -- Part 1: General concepts

[Jagadeesan] Lalita Jategaonkar Jagadeesan et al.
Specification-Based Testing of Reactive Software:
Tools and Experiments. Experience report,
Proceedings of the International Conference on Software Engineering,
May 1997

[Koala] R. van Ommering, F. van der Linden, J. Kramer, J. Magee
The Koala Component model for Consumer Electronics Software

[Koppalkar 02] Nitin Koppalkar and Animesh Bhowmick
Integration of Generic Explorer with the TorX Tool Chain
Philips Nat. Lab. Technical Note 2002/387

[Koppalkar 03] Nitin Koppalkar
Interfacing STATECRUNCHER with TorX for demonstrating the state-based
testing technique taking MG-R components for a case study

[Koymans] Ron Koymans
An Overview of Automatic Test Generation Techniques for
Communication Protocols
[Kwan] Kwan Mei-Ko
Graphic Programming Using Odd or Even Points

[Lee] David Lee and Mihalis Yannakakis
Principles and Methods of Testing Finite State Machines
Proceedings of the IEEE, Vol. 84, No 8, August, 1996

[Myers] G.J. Myers
The Art of Software Testing

[Petrenko] Alexandre Petrenko, Nina Yevtushenko, Alexandre Lebedev,
Anindya Das
Nondeterministic State Machines in Protocol Conformance Testing
Protocol Test Systems VI (C-19), pp. 363-378, 1994

[PHACT] L. Heerink and M. Hollenberg
Conformance Testing Using PHACT

[Phadke] Madhav S. Phadke
Planning efficient software tests

[Sabnani] Krishnan Sabnani and Anton T. Dahbura
A Protocol Test Generation Procedure

[Schneider] Steve Schneider
Concurrent and Real-time Systems, The CSP Approach

[Sloane] N.J.A. Sloane
A library of orthogonal arrays

[StCr...] For STATECRUNCHER appendices/references, please see the start of this section.


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

[Trew 01] Tim Trew
Software Component Composition - Still "Plug and Pray?"

[Trew 03] Tim Trew and Seshaiah Uppala
State-based modelling of software components for integration testing
A practical guide to the creation of STATECRUNCHER models

UML specifications are available from this website.

[VnV] Eileen Hollenberg and Erik Mallens
CvnvTestframe User Manual

[von der Beeck] Michael von der Beeck
A Comparison of Statechart Variants
Aachen University of Technology, Aachen, Germany

Logic Programming and Compiler Writing

http://www.lpa.co.uk
[WinRun] WinRunner v4.0/v5.01, Mercury Interactive
http://www.merc-int.com/products/winrunguide.html

[Yule] D.C. Yule
Automatic State-Based Testing

[Zhang] Fan Zhang and To-yat Cheung
Optimal Transfer Trees and Distinguishing Trees for testing Observable nondeterministic Finite-State Machines