Users Guide

GOTCHA

Release 4.0.0

IBM Haifa Research Laboratory

Contents

1 Introduction

This is the user's guide for the GOTCHA software test suite generator. GOTCHA is designed for use with the Spider test execution engine. We begin with an overview of the entire process. More details on the Spider test execution engine may be found in the Spider Test Execution Engine Users Guide and the Model Driven Test Overview document.

1.1 Overview

GOTCHA-Spider (previously known as GOTCHA-TCBeans) is a set of tools for:

1. Creating and editing models of software components
2. Generating test suites for software components, using a variety of methods.
3. Debugging models of software components.
4. Creating scripts for test drivers.
5. Running test suites against the software components.
6. Comparing the observed results with the results predicted by a model of the component under test.
7. Viewing and analyzing both the test suites and their execution trace.

GOTCHA (an acronym for **G**eneration **o**f **T**est **C**ases for **H**ardware **A**rchitectures) is the part of the tool that generates an abstract test suite from a behavioural model of the software and a set of testing directives.

Spider (which used to be known as TCBeans – when it focused more on Java components) is the part of the tool that interprets the abstract test suite. It either translates the abstract test suite into runnable test scripts or runs the test suite against the unit and automatically compares the results with the model predictions.

This document is an overview on how to use both GOTCHA and Spider together. For more details on either of the individual componets see their respective user manuals.

GOTCHA is the property of the IBM Corporation and is copyright © 2000. Spider is the joint property of the AGEDIS consortium (www.agedis.de).

GOTCHA-Spider Release 4.0 is an extension of GOTCHA-TCBeans Release 3.0. See What's New in this Release.

GOTCHA is derived from the Murphi model checker - a public domain program available on the Murphi Home Page, and the ESPRIT Genevieve Project.

This release of GOTCHA-Spider is tailored to work with the Cygnus g++ compiler available from Redhat. However, this should not be interpreted to mean that GOTCHA-Spider is Open Source code.

GOTCHA-Spider is currently available only to internal IBM users and members of the AGEDIS consortium.

1.2 Requirements

To install and run GOTCHA-Spider, you need the following:

- A computer running Windows 95, 98, 2000, XP Windows NT, AIX, or Linux.
- Java Developer's Kit JDK 1.2 or higher.
- The GOTCHA-Spider software package is available (for IBM users) from http://w3.haifa.ibm.com/softwaretesting/gtcb/ and directly from Alan Hartman (hartman@il.ibm.com for members of the AGEDIS consortium).
- A C++ compiler, preferably g++ from Redhat, which can also be downloaded from our website. (AGEDIS members will receive the whole package on a CD-ROM)

1.3 What's New in this Release

This release (4.0) of GOTCHA-Spider provides substantially new functions in addition to the existing functions available in GOTCHA-TCBeans Release 3.0.

GOTCHA-Spider 4.0. contains the following new and enhanced features:

- A convenient editor for GOTCHA models using the Eclipse framework
- A new set of test generation methods especially tailored for models which suffer from state space explosion see Section 3.3 Generating test suites quickly.
- The ability to use the #include macro in GOTCHA models
- New forms of ruleset for conrolling state explosion in data intensive GOTCHA models see Section 2.4.6.7 Rulesets with Explicit Input Tables and Section 2.4.6.8 Rulesets with Input and Input Pair Coverage.
- New coverage criteria enabling reference to the rules and their parameters see Sections 4.2.2.3 Some Explicit Transition and 4.2.2.6 Explicit Transition Projection.
- Abstract test suite output conforming to the new ATS standard defined in the AGEDIS project

1.4 The GOTCHA-Spider Methodology

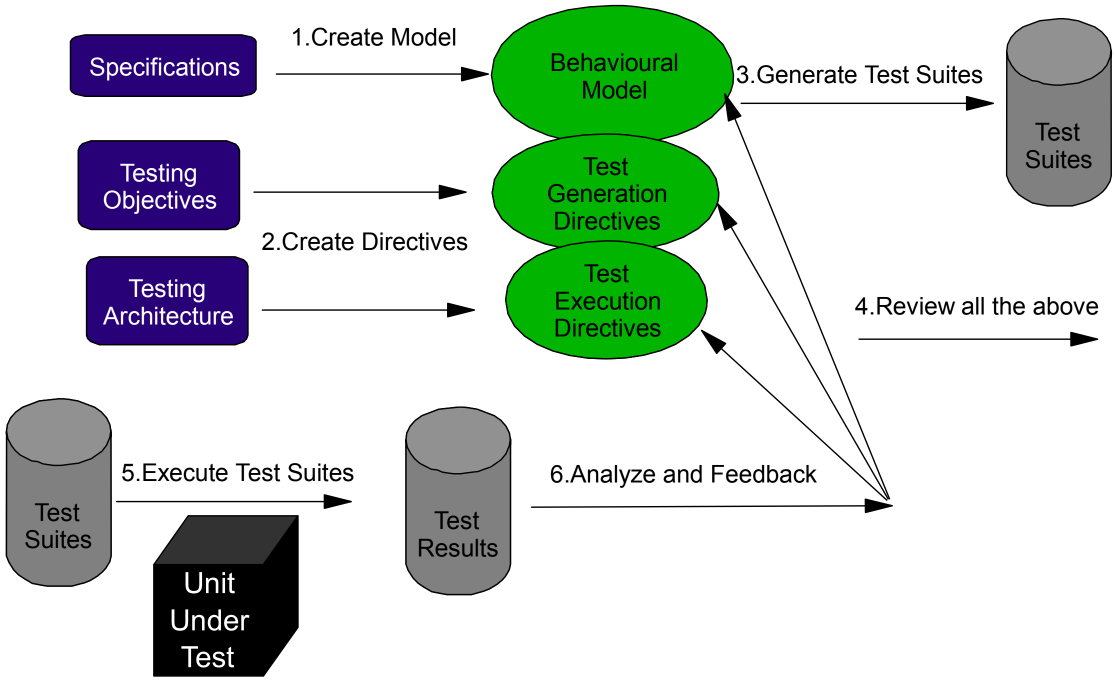


Figure 1. Methodology Diagram

The GOTCHA-Spider methodology for software testing is the process supported by the GOTCHA-Spider software testing toolkit. To use the tool effectively, you must follow the methodology. The methodology consists of the following phases:

1. Write and debug a behavioural model of the software with testing directives. Specification defects are discovered at this stage.
2. Write a testing interface between the model and the application under test. You can write the testing interface in a language understood by an existing test execution engine or by the Spider test driver. Interface defects are discovered at this stage.
3. Generate one or more abstract test suites (ATS) derived from the behavioral model using one or more of the GOTCHA test generators. Review the abstract test suite using the Test Suite Browser (TSBrowser tool). If necessary, modify the model, test constraints, or coverage criteria.
4. Review the model and testing interface with the development team and test team. Further specification defects are discovered at this review.
5. Execute the test suites against the software unit under test using an existing test execution engine or using the Spider test driver. Spider executes the abstract test suite and writes the results to a standardized Suite Execution Trace (SET).
6. Review the Suite Execution Trace with the TSBrowser. Coding defects are discovered at this phase. Observe the test results and, if necessary, augment or restrict the model or interface and repeat steps 4-6.

1.4.1 Write a Behavioral Model for Test

The first step in the methodology is to write and debug a behavioral model of the software application under test in the GOTCHA Definition Language (GDL) – a dialect of the Murphi Definition Language. The model is written on the basis of the software specifications, and in conjunction with the code architects and developers. The model may also contain testing directives, including descriptions of the coverage goals and test constraints required by the test suite. If no coverage goals are explicitly stated, then a set of default coverage goals are available, or a set of random test cases may be generated. GOTCHA provides an interactive test generation and debugging facility for exploration of the model and manual test generation if necessary.

1.4.2 Write a Testing Interface

The second step in the methodology is to create a testing interface between the model and the application under test. The purpose of the interface is to provide the connection between concepts used in the behavioral model and those of the software unit and/or the test execution engine.

Prepare the testing interface by coding an abstract to concrete (A2C) test translation table. The initial A2C translation table is in an html document generated by the interface wizard.

Test suites can be executed using the Spider test driver. The Spider testing interface is a Java class appropriate for testing software applications written in Java, C, or C++. Defects in the software interface specifications are often exposed when writing and compiling the interface class.

1.4.3 Generate Abstract Test Suites

The GOTCHA tool has several test generators for creating abstract test suites for the behavioural model.

The simplest of these test generators generates a set of random input sequences and their expected behaviour.

Another test generator is interactive and can be used to construct an abstract test suite by hand.

The remaining test generators generate test suites based on some form of coverage model.

The simplest coverage model is a guarantee of covering all inputs to all methods described in the model. More complex coverage models are available, and at the highest level of sophistication, the user may define an abstract set of coverage tasks, each of which will be covered by the test suite generated by the GOTCHA comprehensive test suite generator.

The tool reports any coverage tasks that cannot be covered by a test satisfying the constraints. Often these indicate bugs in the model or overly constraining test requirements. However, they may also expose defects in the specification.

You can use the TSBrowser tool to view the abstract test suite.

1.4.4 Review Behavioural Model, Abstract Tests, and Testing Interface

A vital part of the GOTCHA-Spider process is the review of the behavioural model, abstract tests, and testing interface. Testers, architects, and developers of the software conduct the review. The model review reveals inaccuracies, omissions, and contradictions in the specifications. The abstract test review often unearths issues that may be have been misinterpreted by the testers. It also serves to focus the discussion of the behavioural model. The testing interface review reveals problems related to the interface design and specification. These interface defects are similar to those faced by a customer writing an application or component that interacts with the software unit under test.

In addition to revealing specification and interface defects, the review process also reveals defects caused by imperfect communication between members of the development and test teams. Catching these bugs early in the process saves time and expense later on.

The tools that may be used during the review include the interactive test generator, the test suite browser, and the interactive test execution engine.

1.4.5 Execute the Test Suite

The Spider Test Driver can execute a GOTCHA abstract test suite directly against the application under test. This produces a suite execution trace that not only records the test execution, but also compares the outcome of each step in the test with the outcome predicted by the model.

You can use the TSBrowser tool to view the suite execution trace.

The test execution phase usually reveals most of the coding and design bugs.

1.4.6 Observe and Iterate

You should compare the results of executing the test with the coverage goals of the test plan. If necessary, make further modifications to the model, the testing directives, and/or the test generator's runtime parameters. You can then generate further abstract test suites to improve the effectiveness of the test.

1.5 The contents of this document

This document is a manual of the GOTCHA tool for generating test cases from a software model, and thus we concentrate on the first three steps of the methodology:

1. Writing a model
2. Writing test generation directives
3. Generating a test suite

The GOTCHA tool includes several algorithms with built-in testing objectives, so that one may use the tool initially without explicitly writing test generation directives.

2 Writing a Behavioural Model

2.1 What is a Behavioural Model?

A behavioural model of a software unit is an abstract description of the software that includes abstract descriptions of its data types (classes), data structures (objects), and transitions. The transitions of a model are the events that cause its data structures to change their values, and the procedures and functions invoked in order to compute the new values. If the software under test is an API (Application Programming Interface), the calls to the interface are usually modelled by transitions. Transitions are the external events that trigger changes in the state of the software. Transitions may also occur as a result of the passage of time.

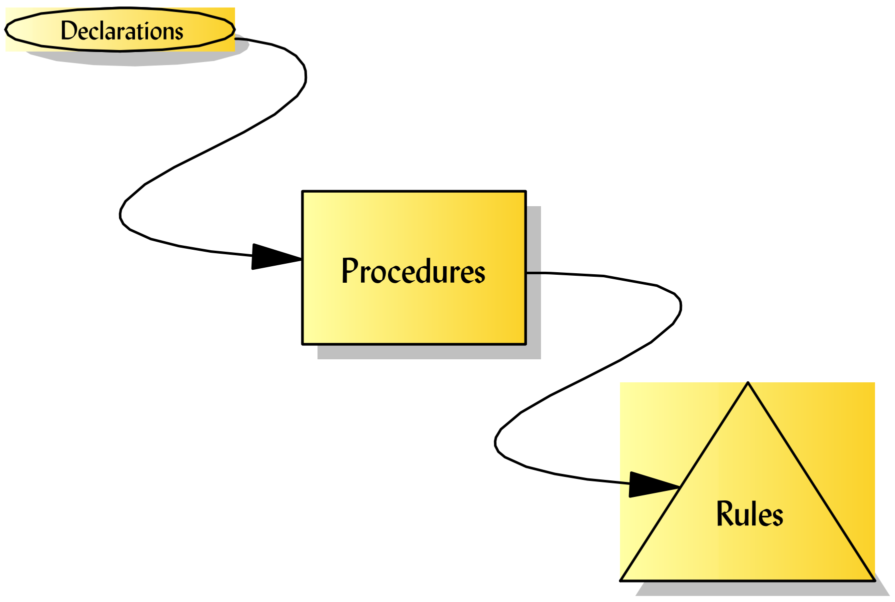
A GOTCHA behavioural model also includes directives to guide the test generator in its choice of test cases to generate. These directives are called coverage criteria and test constraints. Other directives may also be given at the time of running the model, but most coverage criteria and test constraints must be coded into the model itself. It is advisable to keep the coverage criteria and test constraints in separate files to separate out the behavioural aspects of the model from the testing strategy to be used.

The software model is written in a special purpose language for the description of models. This language is based on the Murphi Description Language; large portions of the text in this chapter are taken from the Murphi User's Manual.

2.2 Writing a Behavioural Model

A GOTCHA Definition Language (GDL) model is divided into three parts, which must occur one after the other. The model itself may be spread over a number of files (each of file type .g). We recommend that you keep the file(s) containing test generation directives separate from the file(s) that describe the behavioural model itself.

The three parts of a GDL model are:



1. Declarations
2. Functions and Procedures
3. Rules and Directives

A GDL description consists of:

- Declarations of constants, types, and global variables.
- Declarations and descriptions of functions and procedures (optional).
- A collection of transition rules.
- A description of the states where test cases may start.
- A description of the states where test cases may end (optional).
- A set of coverage criteria (optional).
- A set of state invariants and test constraints (optional).

The behavioural part of GDL is a collection of transition rules. Each transition rule is a command with a pre-condition (a Boolean expression on the global variables) and an action (a block of statements that modify the values of the variables). In GDL Release 3.0 and higher, a transition rule may have several actions associated with it, each of which represents a possible outcome of the transition rule. A transition with more than one possible action is called a *pluRule* and is used to model non-deterministic behaviour of software.

The condition and the action(s) are both written in a Pascal-like language. The action can be an arbitrarily complex statement block containing loops and conditionals. No matter how complex it is, the action is executed *ATOMICALLY*; no other rule can change the variables or otherwise interfere with it while it is being executed.

A GDL Model implicitly determines a state graph (see A Sample Model – Hello World). A **state** is a node in the graph that is labeled by an assignment of a value to each global variable. The start states of the graph are defined by TC_StartTestCase clauses in the <rules> section of the program. The **transitions** or directed arcs of the graph are defined by rules within the <rules> section of the program. Rule name strings and ruleset parameters label the arcs.

2.3 Basic GDL Concepts

2.3.1 Backus-Naur Form (BNF)

The syntax is specified in this guide in a Backus-Naur Form:

<> denote nonterminals.

[] denote optional sections.

{} denote repetition zero or more times.

a | b denotes either a or b.

() denote grouping.

When any of these symbols are required within the language, they are escaped with backslashes.

2.3.2 Lexical Conventions

2.3.2.1 Reserved Words

The following are **reserved words** in GDL:

alias	array	assert
begin	boolean	break
by	case	CC_All_State
CC_All_Transition	CC_Some_State	CC_Some_Transition
CC_State_Projection	CC_Transition_Projection	clear
const	coverage_var	do
else	elsif	end
endalias	endexists	endfor
endforall	endfunction	endif
endpluRule	endprocedure	endrecord
endrule	endruleset	endstartstate
endswitch	endwhile	enum
error	exists	false
finalstate	for	forall
from	from_condition	function

if	in	includes
interleaved	invariant	length
of	on	ordenum
pluRule	procedure	process
program	put	record
return	rule	ruleset
startstate	switch	TC_EndTestCase
TC_Forbidden_State	TC_Forbidden_Transition	TC_Forbidden_Path
TC_StartTestCase	TC_Within	then
to	to_condition	traceuntil
true	type	var
while		#include
ruleID	CC_Some_Explicit_Transition	CC_Projected_Explicit_Transition
via	via_condition	inputcoverage
inputtable	endinputtable	inputpaircoverage

Reserved words are written out in the BNF. Some of these reserved words do not yet have defined meanings; these are reserved for future expansion. Those words are **in**, **interleaved**, **process**, **program** and **traceuntil**.

The new reserved words in Release 4.0 appear at the end of the table in blue.

2.3.2.2 *Case Sensitivity*

GDL is case-sensitive, except for the reserved words. For example, 'foo' and 'Foo' represent different identifiers. 'Begin' and 'BeGiN' represent the same reserved word.

2.3.2.3 *Include Files*

A GDL file may have another GDL file included in it by a pre-processor, just as in the case of C.

The line

```
#include filename.g
```

will result in the entire contents of filename.g being included in the file which contains it.

2.3.2.4 *Synonyms for End*

The reserved word 'end' is a synonym for every specific type of end: 'end' may be used freely in place of 'endrule', 'endfor', etc.

2.3.2.5 *Identifiers*

An identifier is any sequence of letters, underscores, and digits beginning with a letter. All identifiers beginning with underscore are reserved for use by the system. Identifiers are referred to in the BNF below as <ID>.

2.3.2.6 *Strings*

A string, referred to in the BNF as <string>, is a sequence of characters other than double quote (\") enclosed in double quotes.

2.3.2.7 *Integer Constants*

Integer constants, <integer-constant> in the BNF, are specified in base 10.

2.3.2.8 *Comments*

There are two types of comments in GDL: Ada-style comments that begin with -- and end with a newline. C-style comments that begin with /* and end with */. C-style comments do not nest.

2.4 GDL Syntax

2.4.1 Model Syntax

A Model has the following BNF structure:

```
<Model> ::= { <decl> }          -- Constant, type, and global variable declarations
            { <procdecl> }      -- Procedure and function declarations
            { <rules> }         -- Rules, directives, and invariants
```

2.4.2 Declaration Syntax

Declarations have the following syntax:

```
<decl> ::=    const { <constdecl> ; }
         |    type { <typedecl> ; }
         |    var { <vardecl> ; }
         |    coverage_var { <vardecl> ; }
```

2.4.2.1 *Constant declarations*

```
<constdecl> ::= <ID> : <expr>
```

The <expr> of a constant declaration must have a value that can be evaluated at compilation time.

2.4.2.2 Type declarations

<typedecl> ::= <ID> : <typeExpr>

The special enumerated type "boolean" is predefined, along with the constants "true" and "false". The type "integer" is not predefined, because using general integers without restricting them to subranges would consume too much memory.

The **simple types** are Boolean, enumerations, finite subranges of integers.

The **compound types** are arrays of compound or simple types, records of compound or simple types. The index types of arrays must be simple types.

<typeExpr> ::= <ID> -- a previously defined type.

<typeExpr> ::= <expr> .. <expr> -- Integer subrange.

<typeExpr> ::= enum \{ <ID> {, <ID> } \} -- enumeration.

<typeExpr> ::= record { <vardecl> } end

<typeExpr> ::= array \[<typeExpr> \] of <typeExpr>

2.4.2.3 Union Types

In addition to the type expressions defined in the previous section on Type declarations, GDL also contains another simple data type. The syntax for a union type is as follows:

<typeExpr> ::= union \{ <list> \}

<list> ::= <list>, <listelt> | <listelt>, <listelt> /*at least two members in the list*/

<listelt> ::= ID /* an enum type that has already been declared */

The union type provides a shorthand for referring to several enumerated types at once, while retaining the separation between parts of the larger enumerated union type. For example:

```
Type  vowel_t : enum {A, E, I, O, U};
      consonant_t : enum {B, C, D, F, G, H, J, K, L, M, N, P, Q,
R, S, T, V, W, X, Y, Z};
      letter_t : union {vowel_t, consonant_t};
```

The minimum element of a union type (used by `clear`) is the minimum element of the first enumerated type in the list. Therefore, the minimum element of `letter_t` would be `A`.

2.4.2.4 Variable declarations

<vardecl> ::= <ID> { , <ID> } : <typeExpr>

Variable declarations are used to define the global or state variables and also for the definition of local variables.

The use of the reserved word **`Coverage_var`** for coverage variables is no longer necessary in Release 3.0, since the coverage criteria **CC_State_Projection** and **CC_Transition_Projection** provide more expressibility of the important variables to be covered. See Coverage Criteria.

Example: The following example illustrates declarations.

```
Const
      NUSERS: 3;
      NFILES: 3;
      FOO : NFILES /4;
      BAR : true;
Type
      userid_t: enum {a, b, c};
      fileid_t: 1..NFILES;
      cmd_t: enum { Open_for_Read, Open_for_Write, Close };
      file_status_t : enum { Closed, Reading, Writing };
      response_code_t: enum{ AcceptCommand, RejectCommand };
      status_resp_t: enum { OpenR, OpenW, Closed, Locked };
      file_control_block_t : array [ userid_t ] of file_status_t ;
Var
      system_status: array [fileid_t] of file_control_block_t;
      response : response_code_t;
      result   : Boolean;
      reason : status_resp_t;
      filenum : fileid_t;
```

In the example, constant declarations precede type declarations, which precede variable declarations. In a GDL model in general, this need not be the case. Constant, type, and variable declarations may be interspersed in any order, provided that any instance of an identifier is declared before being used. (This is because GOTCHA uses a one-pass compiler).

The array `system_status` is a **two-dimensional array** – since it is an array of `file_control_block_t`, which is itself an array. The second index of the array is an enumerated type (`userid_t`), so that one would access a member of the array as `system_status[1][a]`.

The division in GDL is integer division, so that the constant `FOO` evaluates to zero.

2.4.3 Procedure and Function Syntax

All procedures and functions must be declared at the top level of the program, with the following syntax:

```
<procdecl> ::=  <procedure>
          |     <function>
```

<procedure> ::= procedure <ID> \([<formal> { ; <formal> }] \) ;
[{ <decl> } begin] [<stmts>] end;

<function> ::= function <ID> \([<formal> { ; <formal> }] \)
: <typeExpr>;
[{ <decl> } begin] [<stmts>] end;

Unlike Pascal procedures, procedures and functions with no arguments still need the parentheses surrounding the empty parameter list.

Functions must return a value with a return statement at some point in the function. Functions can have side effects; however, there are restrictions on the use of functions with side effects.

The format of the parameter list in a procedure or a function is:

<formal> ::= [var] <ID> { , <ID> } : <typeExpr>

Formal parameters declared "var" are passed by reference. Formals that are not declared "var" are also passed by reference, but the function or procedure is not allowed to modify them.

Formal parameter declarations and local declarations shadow declarations outside their scope.

Example: the following example illustrates procedures and functions.

```
procedure Swap(var i, j: val_t);
var temp: val_t;
begin
  temp := i;
  i := j;
  j := temp;
end;

function plustwo(input: val_t): val_t;
const two : 2;
begin
  return (input + two);
end;
```

2.4.4 Expression Syntax

GDL is a strongly typed language, so expressions that mix variables of different types are syntactically illegal. Type equivalence is by name only, so two variables that are declared as being of type 1..4 will not be considered as the same type unless a type declaration has named the type 1..4.

Expressions of any integer subrange type are legal wherever an integer expression is legal (although they may generate a run-time error when running GOTCHA; see assignments below). Booleans are not type-compatible with integer expressions.

It is an error to use an out-of-bounds index for an array. This too is detected only when GOTCHA is run.

Ordered enumerated types are now supported in GDL. These are enumerated types with an order defined between members of the type. This allows the user to use the comparison operators, <, >, <=, >= in addition to the = and != operators which are defined for Murphi enumerated types.

2.4.4.1 Designators

```
<designator> := <ID> { . <ID> | \[ <expr> \] }
```

As usual, the form <designator>.<ID> refers to selecting a field of a record. The form <designator> [<expr>] refers to selecting an element of an array.

2.4.4.2 Expressions

```
<expr> :=  \( expr \)
        | <designator>
        | <integer-constant>
        | <ID> \( <actuals> \)           -- a function call.
        | forall <quantifier>
          do <expr> endforall            -- universal quantification, see below.
        | exists <quantifier>
          do <expr> endexists            -- existential quantification, see below.
        | <expr> + <expr>
        | <expr> - <expr>
        | <expr> * <expr>                -- multiplication.
        | <expr> / <expr>                -- integer division.
        | <expr> % <expr>                -- remainder.
        | ! <expr>                       -- logical negation (not).
        | <expr> | <expr>                -- logical disjunction (or).
        | <expr> & <expr>                -- logical conjunction (and).
        | <expr> -> <expr>               -- logical implication. (implies)
        | <expr> < <expr>
        | <expr> <= <expr>
        | <expr> > <expr>
        | <expr> >= <expr>
        | <expr> = <expr>
        | <expr> != <expr>               -- not equals
        | <expr> ? <expr> : <expr>       -- C-style conditional expression.
```

2.4.4.3 Operators

The priority of operators is as follows, with lowest-priority operators listed first and operators on the same line having equal priority:

```
?:
->
|
&
!
< <= = != >= >
+ -
* / %
```

- '+', '-', '*', '/', '%', '<', '<=', '>=', and '>' are only defined on integer operands.
- '=' and '!=' are only defined on simple operands (not on records or arrays).
- '!', '&', '!', and '->' are only defined on Boolean operands.
- For the '?:' operator, the test must be a Boolean expression, and the two alternatives must be of compatible type.
- '+', '-', '%','/', and '*' return an integer; the rest return Booleans, except for '?:' and function calls.
- a->b "a implies b" or "if a then b" is equivalent to bV!a (either b is true or a is false) . The truth table for "a implies b" is given below:

a	b	a->b (a implies b)
T	T	T
T	F	F
F	T	T
F	F	T

2.4.4.4 'Forall' and 'Exists' Operators

The <quantifier> used in "exists" and "forall" has the following syntax:

```
<quantifier> ::= <ID> : <typeExpr>
               | <ID> := <expr> to <expr> [ by <expr> ]
```

Recall that the syntax for quantified expressions ("exists" or "forall") is:

```
<quantified_expr> :=
            | forall <quantifier>
              do < boolexpr > endforall
            | exists <quantifier>
              do < boolexpr > endexists
```

For a quantified expression, the boolexpr must be a Boolean expression; it is evaluated once for each value of the quantifier. A "forall" is true if and only if its expression is true for every value of the quantifier; an "exists" is true if its expression is true for some value of the quantifier.

Consider the following examples:

```
forall f:fileid_t do array[f]>0 endforall;
```

is equivalent to

```
array[1]>0 & array[2]>0 & array[3]>0
```

```
exists f:fileid_t do array[f]>0 endexists;
```

is equivalent to

```
array[1]>0 | array[2]>0 | array[3] >0
```

2.4.5 Statement Syntax

The followings are the statements in GDL:

```
<stmts> ::= <stmt> {; [<stmt>] }


<stmt> ::= <assignment>        /* assignment */
        | <ifstmt>             /* if statement */
           | <switchstmt>            /* switch statement */
           | <forstmt>               /* for statement */
           | <whilestmt>             /* while statement */
           | <proccall>              /* procedure call */
           | <clearstmt>             /* clear statement */
           | <errorstmt>             /* error assertion */
           | <assertstmt>            /* assertion */
           | <putstmt>               /* output statement */
           | <returnstmt>            /* function return */
           | <breakstmt>             /* debugger break statement */
```

2.4.5.1 Assignment

```
<assignment> ::= <designator> := <expression>
```

The target and the expression must have compatible types, and the target must not be declared const. One of the most common GDL syntax errors occurs when the tester writes = instead of **:=** for assignment.

Assigning a value to a variable that is outside its range will generate an error. This error is detected at run time.

2.4.5.2 If statement

```
<ifstmt> ::= if < boolexpr > then [ <stmts> ]
                 { elsif < boolexpr > then [ <stmts> ] }
                   [ else [ <stmts> ] ]
                endif
```

Each of the <expr>'s must be of Boolean type.

A very common syntax error is the omission of the "endif" from an "if" statement. C programmers are also likely to forget the "then". The quirky spelling of "elsif" is also a problem for some.

Example:

```
if (response1 != response2)
then result := OneAcceptOneReject;
else
      if (response1 = AcceptCommand)
      then result := BothAccepted;
      else result := BothRejected;
      endif;
endif;
```

2.4.5.3 Switch statement

```
<switchstmt> ::= switch <expr>
                   { case <expr> {, <expr>} : [ <stmts> ] }
                   [ else [ <stmts> ] ]
                  endswitch
```

Each of the expressions in the case must be a constant of a compatible type with the switch expression. If no case expression is matched, the code labeled 'else' is executed.

There is no fall through on cases (unlike in C).

Example:

```
switch system_status[f][u]
          case Reading , Writing:
                response := Unchanged;
          case Closed:
                  if File_Available_For_Read(f,u)
                  then
                        response := Accept_Command;
                        system_status[f][u] := Reading;
                  else
                        response := Reject_Command;
                  endif;
            else
                  Error "Illegal value for system_status";
endswitch;
```

2.4.5.4 *For statement*

<forstmt> ::= for <quantifier> do [stmts] endfor

Quantifiers apply to "for" statements, to quantified expressions, and to rulesets (see below).

<quantifier> ::= <ID> : <typeExpr> {; <ID : <typeExpr> }

| <ID> := <expr> to <expr> [by <expr>]

The first form executes the body of the "for" statement for each value in the <typeExpr> (which must be a simple type), from least to greatest value. The second form corresponds to the Modula-2 FOR statement. The two expressions must be of integer type, and the "by" expression must be a constant expression.

Examples:

```
for u: userid_t do
      array[u] := 0;
endfor;
for f:= 1 to 3 by 2 do
      array[f] := 0;
endfor;
For f : fileid_t; u: userid_t Do
     system_status[f][u] := Closed;
End;
```

Note: Using a quantifier, in a "for" statement or a quantified expression, declares the <ID> of the quantifier local to the "for" statement, shadowing any external declarations. (C programmers beware – you are used to declaring your loop variables before using them.)

It is illegal to modify the quantifier variable from within the body of the "for" loop.

2.4.5.5 While statement

<whilestmt> ::= while < boolexpr > do [stmts] end

Example:

```
f:=1;
while f<=3 do
    array[f] := 0;
    f:=f+1;
endwhile;
```

An infinite loop is a runtime error. Infinite loops pose a practical problem for GOTCHA. Right now, GOTCHA stops with an error message after 10,000 iterations as a default option. The user may change this limit by a command-line runtime argument to GOTCHA.

2.4.5.6 Procedure call

<proccall> ::= <ID> \(<expr> {, <expr> } \)

This obeys all the standard rules of procedures. Const formal parameters can be passed an actual of any compatible type; var parameters must be passed an lvalue of the same type; a var parameter of a subrange type must be passed an lvalue of the same subrange type.

2.4.5.7 Clear statement

<clearstmt> ::= clear <designator>

This sets all components of an lvalue to the minimum values of their type. The minimum value of an enumerated type is the first value declared in the list of names. The minimum value of the type boolean is false.

NOTE: "Clear" is frequently used to set "uninteresting" variables to a fixed value; otherwise, many states would be created during test generation with random values in these variables. We do not recommend the use of "clear" for other purposes.

2.4.5.8 Error assertion

<errorstmt> ::= error <string>

An "error" statement generates a run-time error. If an "error" statement is executed, test generation terminates and the specified string is printed to the console. See example above.

2.4.5.9 Assertion

<assertstmt> ::= assert < boolexpr > [<string>]

"assert < boolexpr > <string>" is completely equivalent to

"if !< boolexpr > then error <string> end"

2.4.5.10 *Output statement*

<putstmt> ::= put (<expr> | <string>)

Prints out the indicated value, each time the statement is executed. This is handed straight to printf, so be careful to include a

```
put "\n"
```

after each line you want to print.

Generally, this will cause a large quantity of material to be printed during test generation (with much duplication). We have used it for debugging and for generating a file of all possible values of certain variables (which we then process to eliminate duplicates).

2.4.5.11 *Function return:*

<returnstmt> ::= return [<expr>]

Exits the current procedure, function, rule, or startstate. If exiting a function, the <expr> must be provided and must match the return type of the function; otherwise, there must be no return value.

2.4.5.12 *Debug break statement:*

<breakstmt> ::= break

A "break" statement will cause the test generator to halt execution at this point in the code when GOTCHA is run with the –dbg flag. For more details about the model debugging interface see Debugging a Model. If the tool is run without the flag –dbg, than a "break" statement has no effect.

2.4.6 Rules and Directives Syntax

This section deals with behavioural rules. For details about coverage criteria and test constraints see Writing Testing Directives.

The syntax for behavioural rules and directives is as follows:

<rules> ::= <rule> {; <rule> } [;]

```
<rule> ::= <simplerule>
         | <pluRule>
         | <starttestcase>
         | <plustarttestcase>
         | <invariant>
         | <ruleset>
         | <aliasrule>
         | <coveragecriterion>
         | <testconstraint>
```

2.4.6.1 Simple rule

```
<block> ::=    begin
                  [<stmts> ]
                 end


<simplerule> ::= rule <Rule_Identification_string>
                  [<boolexpr > ==> ]  {<decl> } <block> [;]
```

A simple rule determines a transition from one state of the finite state graph to another.

A simple rule defines a transition between states. Logically, it consists of a block, which is a set of statements to be executed, and a condition, a Boolean expression characterizing the states under which the block may be executed. If the condition is true in a state, then the body of the rule may be executed to provide a transition to another state.

The condition of a rule is optional. When no condition is specified, the rule is enabled in all states.

The Rule_Identification_string that labels the rule is output to the abstract test suite as a method pattern, whenever the transition defined by the rule is included in a test case. For more details of the use of the method pattern, see section 3.1 The Abstract Test Suite – GOTCHA's output file.

It is an error to use an expression with side effects in a rule condition. An example of such an illegal action would be to call a (Boolean) function that changes the value of a state variable.

NOTE: The GOTCHA compiler does not usually pick up these errors – so be extra careful!

The rule may declare local variables, constants, and types, which are not part of the state.

It is an error if the model does not have at least one simple rule or pluRule.

Example:

```
Rule "Close(int f, char u)"
  TRUE
```

```
==>
begin
  response := AcceptCommand;
  if ( system_status[f][u] = Closed) then
    reason := FileStatusUnchanged;
  else
    reason := FileClosed;
    system_status[f][u] := Closed;
  endif;
end;
```

2.4.6.2 PluRule

<blocks> ::= <block> { <block> }

```
<pluRule> ::= rule < Rule_Identification_string >
                    [<boolexpr > ==> ]
                    [{<decl> } pluRule <blocks> ]
                         endpluRule [;]
```

A pluRule determines a transition from one state of the finite state graph to several equally valid alternative outcomes. It is used to model a situation where there are several possible outcomes to applying a particular stimulus to the system under test.

An example of such a situation is when the stimulus (rule) consists of two processes simultaneously requesting the same resource. It is possible that both the first process and second process get the resource. In this case you would use a pluRule, with the first block of statements allocating the resource to process one and putting process two into a wait, and the second block of statements reversing the roles.

A simple rule defines a transition between states. A pluRule defines the sets of possible outcomes of applying a stimulus to a non-deterministic software module. Logically, it consists of a set of blocks (each of which is a set of statements to be executed) and a condition (a Boolean expression characterizing the states under which the stimulus may be applied). If the condition is true in a state, then any one of the blocks of the pluRule may be executed to provide a transition to another state.

The condition of a pluRule is optional. When no condition is specified, the pluRule is enabled in all states.

It is an error to use an expression with side effects in a rule condition. An example of such an illegal action would be to call a (Boolean) function that changes the value of a state variable.

NOTE: The GOTCHA compiler does not usually pick up these errors – so be extra careful!

The Rule_Identification_string that labels the rule is output to the abstract test suite as a method pattern, whenever the transition defined by the rule is included in a test case. For more details on the method pattern, see section 3.1 The Abstract Test Suite – GOTCHA's output file.

The pluRule may declare local variables, constants, and types, which are not part of the state. The blocks of statements are enclosed within the keywords "pluRule" and "endpluRule."

It is an error if the model does not have at least one simple rule or pluRule.

NOTE: There is no semicolon between the blocks in a pluRule. See the example below.

PluRules should be used sparingly, since they create the possibility of executing a test case with an undecidable outcome (i.e., one may not know if a particular execution of a test case was successful or not, which could cause a lengthy retry cycle). Example:

```
Rule "GetResource(int r, char u1) || GetResource(int r, char u2)"
  u1 < u2   -- different users
==>
pluRule
      begin -- First possibility, user 1 gets in first
            AssignResource(r, u1);
      end -- first possibility

      begin -- Second possibility, user 2 gets in first
            AssignResource(r, u2);
      end -- second possibility
endpluRule;
```

2.4.6.3 Start Test Case

<starttestcase> ::= TC_StartTestCase < Rule_Identification_string >

{ <decl> } <block> [;]

A TC_StartTestCase is a special type of rule. It is executed only at the beginning of a test generation. In other words, every test in the test suite consists of one TC_StartTestCase and then one or more simple rules. A TC_StartTestCase rule has no preconditions.

If a TC_StartTestCase does not assign a value to a global variable, that global variable is "cleared" (i.e., the compiler inserts a "clear" statement for every global variable at the beginning of each TC_StartTestCase block).

If the model does not have one TC_StartTestCase or non-deterministic TC_StartTestCase, then the GOTCHA translator creates a default TC_StartTestCase rule, which clears all state variables.

The Rule_Identification_string that labels the rule is output to the abstract test suite as a method pattern, whenever the transition defined by the rule is included in a test case. For more details on the method pattern, see section 3.1 The Abstract Test Suite – GOTCHA's output file.

Example:

```
TC_StartTestCase "Initialize()"
Begin
  clear system_status;
  response := RejectCommand;
  reason := FileClosed;
End;
```

2.4.6.4 Non-deterministic Start Test Case

```
<plustarttestcase> ::= TC_StartTestCase  < Rule_Identification_string >
                          [ { <decl> } pluRule <blocks> ]
                                endpluRule [;]
```

A non-deterministic TC_StartTestCase is the analog of a pluRule for starting a test case. You should use it if the system under test cannot be initialized in a predictable state. The blocks of a non-deterministic StartTestCase describe each of the possible starting configurations.

If a block in a non-deterministic StartTestCase does not assign a value to a global variable, that global variable is "cleared" (i.e., the compiler inserts a "clear" statement for every global variable at the beginning of each StartTestCase block.)

The Rule_Identification_string that labels the rule is output to the abstract test suite as a method pattern, whenever the transition defined by the rule is included in a test case. For more details on the method pattern, see section 3.1 The Abstract Test Suite – GOTCHA's output file.

Example:

```
TC_StartTestCase "Initialize()"
PluRule
      begin
        clear system_status;
        response := RejectCommand;
        clear reason;
      end
      begin
        clear system_status;
        response := AcceptCommand;
        clear reason;
      end
```

```
EndpluRule;
```

2.4.6.5 *Invariant*

<invariant> ::= invariant [<string>] < boolexpr >

The form

```
  invariant "foo"
    <expr>
```

is syntactic sugar for

```
 rule
   !<expr>
 ==>
   Error "Invariant violated: foo"
 end
```

Many modelers find it more natural to use an embedded specification style with assert and error statements than to use invariants for some conditions. However, for properties that are conveniently expressed as invariants, it is generally more efficient to express them as invariants, because the compiler can then take advantage of the restricted properties of that invariant.

Invariants are typically used in the debugging phase of writing a model; they have no function in test generation.

It is an error to use an expression with side effects in an invariant.

Example:

```
Invariant "At most one writer on the file"
      NumWriters(f) <= 1;
```

2.4.6.6 *Ruleset*

<ruleset> ::= ruleset <quantifier>
    {; <quantifier> } do [<rules>] end

A ruleset can be thought of as syntactic sugar for creating a copy of its component rules for every possible value of its quantifiers. In GOTCHA 4.0 we also introduce ways of limiting the number of rules generated by a ruleset.

When a rule within the ruleset is included in a test, the values of the quantifying variables are output to the abstract test suite as a data pattern. A data pattern looks like a sequence of assignment statements.

Rulesets may be thought of as passing parameters to their constituent rules. In the following example, the ruleset passes a fileid and a userid to the rule, which opens a file for reading.

The standard ruleset uses all possible combinations of the input values. The following example creates an Open_for_Read() rule for each possible combination of files and users. If there are three users and two files, the number of rules generated will be six.

```
Ruleset f: fileid_t; u: userid_t
 Do
      Rule "Open_for_Read(int f,int u)"
            reset_status != hung
      ==>
      Begin
      switch system_status[f][u]
          case Reading , Writing:
                response := Unchanged;
          case Closed:
                  if File_Available_For_Read(f,u)
                  then
                        response := Accept_Command;
                        system_status[f][u] := Reading;
                  else
                        response := Reject_Command;
                  endif;
      endswitch;
      End;
End;  -- of ruleset
```

Rulesets may also pass parameters to directives including TC_StartTestCase and TCEndTestCase; not only to simple rules. For more details on the use of rulesets to pass parameters to the unit under test, see section 3.1 The Abstract Test Suite – GOTCHA's output file.

2.4.6.7 Rulesets with Explicit Input Tables

```
<ruleset> ::= ruleset <quantifier>
                {; <quantifier> }
            InputTable <identifier> <Input_Table_Identification_string>
            <TableRow> { <TableRow>}
            EndInputTable
            do [<rules>] end
       <TableRow> ::= <constant_expression> {<constant_expression> };
```

A ruleset with an explicit input table allows the user to define precisely which combinations of values for the parameters will be input to the rules within the ruleset. In a standard ruleset, all possible combinations of values are used. The user should use an explicit input table or an implicit input table when the number of possible combinations of input variables is too large.

The following is an example of an explicit input table:

```
type
      Result_t : enum {OK, MyError   };
      int_range_t : 0..3;
      Type2: enum { a, b, c, d, e };


ruleset  r: Result_t; aa: int_range_t; bb:int_range_t; cc:Type2
InputTable  TableId  "A Sample Input Table"
    OK         0  1  a;
    OK         3  1 a;
    MyError     1  3 d;
    MyError     2  0 e;
endInputTable


do
      rule "F2(r,aa,bb,cc)"
      :
endruleset;
```

The rule `F2(r,aa,bb,cc)` is only instantiated in the model with the four combinations of values given in the input table rather than the $2 \times 4 \times 4 \times 5 = 160$ possible combinations of the four input parameters r, aa, bb, and cc.

Compilation errors will occur if the constant expressions in the table rows are not of the correct type required by the quantifier of the ruleset, or if the number of constant expressions in a row does not equal the number of quantifiers in the ruleset.

2.4.6.8 Rulesets with Input and Input Pair Coverage

```
      <ruleset> ::= ruleset <quantifier>
                  {; <quantifier> }
```

inputcoverage {<integer_expression>} |

inputpaircoverage {<integer_expression>}

do [<rules>] end

There are two types of rulesets with implicit input tables (other than the standard ruleset which implicitly defines all possible combinations).

Input Coverage rulesets create a table of values where each input value occurs a fixed minimum number *n* of times in the table. If no <integer_expression> is given for the coverage factor *n*, then *n* is assumed to be 1.

The following is an example of an input coverage ruleset:

```
type
       Result_t : enum {OK, MyError   };
       int_range_t : 0..3;
       Type2: enum { a, b, c, d, e };

ruleset  r: Result_t; aa: int_range_t; bb:int_range_t; cc:Type2
InputCoverage
do
       rule "F2(r,aa,bb,cc)"
       :
endruleset;
```

The rule `F2(r,aa,bb,cc)` will be instantiated in the model with an input table which is guaranteed to include each value of each of the four input parameters r, aa, bb, and cc at least once. For example GOTCHA may instantiate this as a set of five rules with the following values of the input parameters:

r	**aa**	**bb**	**cc**
OK	0	1	a
MyError	1	3	b
OK	2	0	c
MyError	3	2	d
OK	0	3	e

Specifying

```
InputCoverage 2
```

would produce a table with ten entries, guaranteed to cover each input value at least twice, for example:

r	**aa**	**bb**	**cc**
OK	0	1	a

MyError	1	3	b
OK	2	0	c
MyError	3	2	d
OK	1	3	e
MyError	2	1	a
OK	2	2	b
MyError	0	0	c
OK	3	0	d
MyError	1	2	e

Input Pair Coverage rulesets create a table of values where each pair of input values occurs a fixed minimum number *n* of times in the table. If no <integer_expression> is given for the coverage factor *n*, then *n* is assumed to be 1.

The following is an example of an input pair coverage ruleset:

```
type
      int_3_t : 0..2;

ruleset  aa: int_3_t; bb:int_3_t; cc: int_3_t; dd : int_3_t
InputPairCoverage
do
      rule "F4(aa,bb,cc,dd)"
      :
endruleset;
```

The rule `F4(aa,bb,cc,dd)` will be instantiated in the model with an input table which is guaranteed to include each pair of values of each of the four input parameters aa, bb, cc, and dd, at least once. For example GOTCHA may instantiate this as a set of nine rules with the following values of the input parameters:

aa	**bb**	**cc**	**dd**
0	0	0	0
0	1	1	1
0	2	2	2
1	0	1	2
1	1	2	0
1	2	0	1
2	0	2	1
2	1	0	2
2	2	1	0

The input table constructed by GOTCHA for input pair coverage is close to the minimum possible size (number of rows) that covers each pair of inputs. To estimate how large this minimum size is multiply the sizes of the two largest quantifier ranges (in the example above this is $3 \times 3 = 9$) to get a lower bound. If the number of quantifiers, q, in the ruleset ($q = 4$ in the example above) is large, one should multiply the lower bound by a factor roughly equal to $\log_2 q$.

2.5 A Sample Model – Hello World

This section contains a simple model of a piece of software whose specifications are given below. All the source code for the application and its models are contained in the samples directory (see Model Driven Test Overview).

2.5.1 Hello World Application Specification

The application is a class whose purpose is to output the letters of the string " HELLO WORLD ". The class has three methods:

- initialize() – which sets a pointer at the blank character before the H, and enables the object to receive forward() calls.
- forward() – which moves the pointer forward in the string and outputs the next character. After reaching the blank at the end of the string, forward() returns a Null character.
- quit() – which returns a Null character and prevents any further calls to the forward() method.

2.5.2 The GDL Model of Hello World

The model is given below:

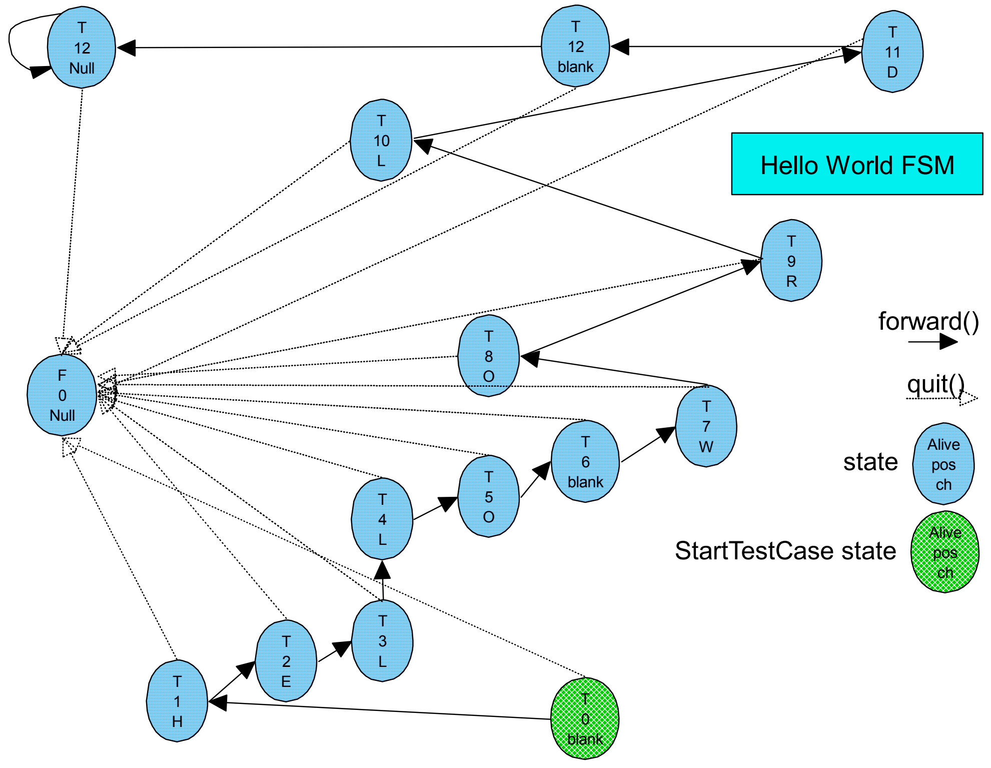
```
-- The Declarations
Const MAXPOS : 12;
Type  position_t : 0..MAXPOS;
Type char_t : enum{Null, H, E, L, O, blank, W, R, D};
Var   ch: char_t;
Var   pos : position_t;
Var   alive : boolean;

-- The functions
function stringchar(p : position_t) : char_t;
var result : char_t;
begin
      switch pos
            case 0,6,12:  result := blank;
            case 1:       result := H;
            case 2:       result:=E;
            case 3,4,10:  result:=L;
            case 5,8:     result:=O;
            case 7:       result:=W;
            case 9:       result:=R;
            case 11:      result:=D;
      endswitch;
      return result;
end;

-- The transition Rules
Rule "forward()"
      alive
==>
Begin
      if pos < MAXPOS
      then
            pos := pos + 1;
            ch := stringchar(pos);
      else
            ch := Null;
      endif;
End;

Rule "quit()"
      alive
==>
Begin
      alive := false;
      ch := Null;
      pos := 0;
End;

-- The Test Constraint for starting a test case
TC_StartTestCase "initialize()"
Begin
      alive := true;
```

```
        ch := blank;
        pos := 0;
End;
```

2.5.3 The Hello World Finite State Machine

The finite state machine graph described in this model is shown pictorially below. Note that the values of the three state variables `alive`, ch, and pos describe each state. We have chosen to illustrate the nature of projection onto variables by drawing the values taken by the projection variable, `ch`, on the X-axis, and the values taken by the non-projection variables (`alive` and `pos`) on the Y-axis. A state will be denoted by the ordered pair `(ch, alive and pos)` to emphasize the use of the `ch` variable as the x-coordinate and the combination of `alive` and `pos` as the y-coordinate in the diagrams below.

In theory, any combination of values is possible, so any position in the diagram below could represent a state. For example the state `(Null, True 0)` is represented in the bottom left hand corner. The state `(Null, True 12)` is represented in the top lefthand corner. All states with `ch = Null` are in the leftmost column. All states with `ch = D` are in the rightmost column.



The start test case state is coloured green, and all other intermediate states are blue.

The rule, `forward()` enables us to pass from one state to another using the solid arrows, while the dotted arrows represents the `quit()` transitions. In general when drawing state graphs, the arrows are labelled with the name of the rule that is activated to make the transition between states. Note that a loop represents a transition from a state to itself, as in the `forward()` transition from the state `(Null, True 12)`.

3 Generating Test Suites – First Steps

3.1 The Abstract Test Suite – GOTCHA's output file

GOTCHA generates a set of test cases and outputs the resulting test suite to a file using the XML Schema for Abstract Test Suites defined by the AGEDIS project. The Abstract Test Suite (ATS), and its extension the Suite Execution Trace (SET) are formally defined and largely explained in the TestSuite User's Guide. Not all the features described in that document are used by the GOTCHA generated test cases. In this section we describe the features of an ATS that are produced by GOTCHA.

Note that the ATS can be most conveniently viewed using the TSBrowser which is a separate component in the Model Driven Test Tools. Any XML viewer or flat file editor may be used to view the test suites.

The first **filename** of the GDL model is used as the **class** name of the unit under test in the ATS.

The **constants** declared in the model are translated as constants in the ATS file.

The **types** declared in the model are translated as types in the ATS file. Some types not explicitly declared in the model may appear as types in the ATS file. A type expression used in the model will be declared with a new GOTCHA generated name, and some new types may be defined by GOTCHA when the use is made of certain test constraints and coverage criteria.

Each stimulus of the unit under test corresponds to a Rule in the GDL model. These rules are listed as **members** of the class, and their signature is derived from the rulesets which enclose the rule in the GDL model. Each rule will generate a member of the class. Some additional members will also be declared when GOTCHA generates a default end test case condition or start test case rule.

Each state variable in the model also generates a member of the class.

For example if the model contained the following lines in the file named **filesystem.g**:

```
Const   NFILES: 3;
Type    userid_t:  enum {a, b, c};
      fileid_t: 1..NFILES;
      response_code_t: enum { AcceptCommand, RejectCommand } ;
Var   response:    response_code_t;
Ruleset f : fileid_t; u: userid_t
Do
Rule "OpenForRead(int f, char u)"
..
EndRuleSet;
TC_StartTestCase "Initialize()"
```

The ATS file would contain the following XML fragments describing the model:

```
<testSuite>
<abstractTestSuite generator="GOTCHA">
<model>
<class name="filesystem">
      <constants>
            <constant name="NFILES" type="int">
                  <value>3</value>
            </constant>
      </constants>
<types>
            <type name="fileid_t">
                  <range type = "int">
                        <interval from="1" to="3"/>
                  </range>
            </type>
            <type name="userid_t">
                  <enum>
                        <element name="c"/>
                        <element name="b"/>
                        <element name="a"/>
                  </enum>
            </type>
            <type name="response_code_t">
```

```
                    <enum>
                          <element name="RejectCommand"/>
                          <element name="AcceptCommand"/>
                    </enum>
              </type>
</types>
      <members>
            <member signature="Initialize()"/>
            <member signature="OpenForRead(fileid_t, userid_t)"/>
            <member signature="response:response_code_t"/>
      </members>
</class>
<object name="filesystem_1" class="filesystem"/>
</model>
```

The test cases then follow the model in the ATS file. Each test case consists of a series of steps, each step being either an invocation of a rule or checking the values of the state variables.

For example:

```
<testCase id="TestCase0">
<step id="S0_S000000000031_0">
<interaction object="filesystem_1" signature="Initialize()"
type="call_return">
</interaction>
</step>
<step id="S1_S000000000031_1">
<interaction object="filesystem_1"
signature="response:response_code_t" type="check">
<value>RejectCommand</value>
</interaction>
</step>
<step id="S4_00000000040R500000100000_0">
<interaction object="filesystem_1"
signature="OpenForRead(fileid_t, userid_t)" type="call_return">
<value>2</value>
<value>c</value>
</interaction>
</step>
<step id="S5_00000000040R500000100000_1">
```

```
<interaction object="filesystem_1"
signature="response:response_code_t" type="check">
<value>AcceptCommand</value>
</interaction>
</step>
</testCase>
</testSuite>
```

Spider test execution directives are used to map GOTCHA state variables, rules and types to their implementation counterparts. The class representing the implementation defaults to the model's name. State variables and rules default to implementation data members and methods with the same name. Spider invokes the rule implementation method when the rule is fired in the abstract test suite. It then compares each state variable with its implementation data member to determine whether the actual results are equal to the results predicted by the model. Table 1 describes the implementation defaults for the GOTCHA types:

Table 1 GOTCHA to Spider Defaults

GOTCHA type	**default implementation type**
boolean	boolean
Integer range	integer
enumeration	string
array	array
record	Class (inside model's class)

The default value of the enumeration members is the members' name.

Changes to the above defaults are specified in Spider test execution directives. For more details see the Model Directed Test Overview and the Spider Test Execution User Guide.

3.2 Interactive Test Generation

The GOTCHA tool provides methods for generating a test suite both automatically and interactively. It is often helpful to use the interactive test generator as an aid in understanding the model that has been built before starting to generate automatically. The interactive generation acts as a sanity check on the model, and provides a convenient method for debugging the model, and generating a few tests for the review process.

3.2.1 Introduction to the Interactive Generator

The GOTCHA interactive test generator serves two main purposes:

- Enables you to walk through the finite state machine and observe the effects of the rules, preconditions, StartTestCase clauses, EndTestCase clauses, and other testing directives on the state variables.

- Enables the creation of manually constructed test cases and small test suites in the abstract test suite format.

The interactive generator uses a command line interface with short letter or number commands, which you input. The interactive session maintains a record of the steps taken so far in exploring the model. This is retained in memory as the current test case. The current test case can be stored in the current test suite. After storing a test case, a new test case can be started from a StartTestCase rule, or a new test can be created, carrying on from where the previous test case left off.

After saving one or more test cases, the resulting test suite can be printed to an abstract test suite file.

The interactive session may be in one of two conditions:

4. Waiting for a choice of rule (RuleWait), or
5. Waiting for a choice of state (StateWait).

The second condition – StateWait – is only encountered if the previous rule chosen was a pluRule, or if the test case is started by a non-deterministic StartTestCase rule.

3.2.2 Starting An Interactive GOTCHA Session

In order to invoke the interactive generator, use the following command in a DOS window:

```
GOTCHA beh_model.g test_dir.g -itg
```

This assumes that the behavioural model and test directives are in the files *beh_model.g* and *test_dir.g* respectively.

At the start of an interactive GOTCHA session, the following actions take place:

- If there is more than one StartTestCase rule, the session displays all the StartTestCase rules and prompts the user to choose one. The session is now in the RuleWait condition.
- If the only StartTestCase rule is deterministic, then the session displays the set of rules enabled in the state reached by applying the StartTestCase rule. The session is now in the RuleWait condition, and the current test case already contains the first rule and state specified by the unique StartTestCase rule and its outcome.
- If the only StartTestCase rule is non-deterministic (a pluRule), the session enters the StateWait condition and displays the set of possible states reachable.

3.2.3 What to do in RuleWait

When the session is in RuleWait, it lists all the enabled rules at the current state. The number of rules displayed is governed by the Rules Per Screen parameter (default 5) – but all enabled rules can be displayed using the (m) more command.

The format of a list of enabled rules is as follows:

***Rule** 1 (E0V1F0):*

MethodPattern1 DataInputPattern1

Rule *2 (**E1V2F1**):*

MethodPattern2 DataInputPattern2

(**E**n**V**m**F**k) indicates that the states reachable from this rule include n valid EndTestCase states, m valid non-EndTestCase states, and k forbidden states (forbidden by a test constraint).

The RuleWait user options are:

- Choosing a rule (enter the number of the rule selected). This action appends the new rule to the current test case and displays the set of alternative states. The choice may result in three possible outcomes:
    a. If the rule chosen is not a pluRule and it generates a single permissible state (E1V0F0 or E0V1F0), then the new state is entered, followed by the list of enabled rules. The session remains in RuleWait.
    b. If the rule chosen generates only forbidden states (E0V0Fn), then all the forbidden states are displayed. The session remains in RuleWait and prompts you to choose another rule to continue.
    c. If the rule is a pluRule that generates both permissible and forbidden states, then all states are displayed. The session moves to StateWait, but only permissible states may be chosen.
- Show Current State (c): displays the values of all the state variables in the current state. The session remains in RuleWait.
- Back (b): retracts the previous state or rule choice. The session transitions to either StateWait or RuleWait, depending on the previous choice, and displays the appropriate list of states or rules.
- Restart (r): prompts for save/delete of the partial test case and performs the action requested. The session returns to its initial state, while retaining the current test suite in memory.
- Save (partial) test (s): saves the current partial test case to the current test suite in memory. The session remains in RuleWait.
- Print (partial) test (p): prints the current partial test to an ATS file and prompts for a file name, and whether to continue from the current state or return to a StartTestCase. The session remains in RuleWait.
- Print test suite (ps): prints the current partial test and all the saved tests to an ATS file and asks whether to start a new test case, start a new test suite, or continue from the current position. The session remains in RuleWait.
- Discard test suite (d): prompts for confirmation, then discards all tests stored so far. The session returns to its initial state.
- Information (i): displays the number of test cases in the current test suite, the number of states and rules in the current test case, and whether or not the current state is an EndTestCase state.
- Commands Help(h): lists the commands available and their interpretation (1-NR, c, b, r, s, p, ps, d, i, h, m, o, q). The session remains in RuleWait.

- More (m): continues the display of the current enabled rules list.
- User Interface Options (o): allows you to change the user interface options. There are three user interface options currently available:

  a. Show the states in a long or shortened form. The long form of the states shows each of the variable names and their respective values. The short form shows only the values in a compressed string.

  b. Rules Per Screen – limits the number of rules displayed in a group, use (m) more to see the next set of rules in a group.

  c. States Per Screen – limits the number of states displayed in a group, use (m) more to see the next set of states in the group. This parameter applies to the alternative states list, the EndTestCase states list, and the Forbidden states list.
- Quit (q): prompts for save/delete partial test, prompts for print/discard current test suite. The interactive GOTCHA session is terminated.

3.2.4 What to do in StateWait

Upon entering the StateWait condition, the interactive GOTCHA session prints the list of possible alternative states resulting from the application of a pluRule. The number of states displayed is governed by the States Per Screen parameter (default 5) – but all alternatives can be displayed using the (m) more command.

The format of a list of possible alternative states is as follows:

***State** 1 **(E)**:*

var1 = val11

var2 = val12

…

varn = val1n

***State** 2 **(V)**:*

var1 = val21

var2 = val2,

…

varn = val2n

***State** 3 **(F)**:*

var1 = val31

var2 = val32

…

varn = val3n

(E) indicates that the state is a valid EndTestCase state.

(V) indicates that the state is a valid non-EndTestCase state.

(F) indicates a forbidden state (forbidden by a test constraint).

Valid EndTestCase states will be displayed before valid non-EndTestCase states, and forbidden states will be displayed last.

StateWait user options are:

- Choose a state (input a state number): appends the new state to the current test case and displays the set of enabled rules in the new state. The session moves to RuleWait.
- Show EndTestCase constraints (e): displays which EndTestCase constraints are satisfied by the listed EndTestCase states. The session remains in StateWait.

  Sample Display:

  The above states are EndTestCase states because:

  ***State** 1:*

  EndTestCasePattern1

  ParameterPattern1

  ***State** 1:*

  EndTestCasePattern2

  ParameterPattern2

  ***State** 2:*

  EndTestCaseStatePattern3

  ParameterPattern3

- Show Forbidden (f): displays the constraints violated by the forbidden states that are reachable by applying the rule just chosen. The session remains in StateWait.

  Sample Display:

  The above states are forbidden because:

  ***State** 3:*

  TestConstraintPattern1 ParameterPattern1

  ***State** 3:*

  TestConstraintPattern2 ParameterPattern2

  ***State** 4:*

  TestConstraintPattern3 ParameterPattern3

- Show Current State (c): displays the current state. The session remains in StateWait.
- Back (b): retracts the previous rule choice and displays the set of enabled rules again. The session moves to RuleWait.
- Restart (r): prompts for save/delete of the partial test suite and performs the request. The session reinitializes the current test case.

- Information (i): displays the number of test cases in the current test suite, the number of states and rules in the current test case, and whether or not the current state is an EndTestCase state.
- Commands Help(h): lists the commands available and their interpretation (1-V, c, b, r, e, f, i, h, m, o, q) where V is the total number of valid states.
- More (m): continues the current state list under display. If all alternatives are being displayed, it displays the next set of alternatives; if EndTestCase states are being displayed, it displays the next set of EndTestCase states; and if forbidden states are being displayed, it displays the next set of forbidden states.
- User Interface Options (o): allows you to change the user interface options. There are three user interface options currently available:

    a. Show the states in a long or shortened form. The long form of the states shows each of the variable names and their respective values. The short form shows only the values in a compressed string.

    b. Rules Per Screen – limits the number of rules displayed in a group, use (m) more to see the next set of rules in a group.

    c. States Per Screen – limits the number of states displayed in a group, use (m) more to see the next set of states in the group. This parameter applies to the alternative states list, the final states list, and the illegal states list.
- Quit (q): prompts for save/delete partial test, prompts for print/discard current test suite. Exits the program

3.2.5 A Sample Interactive Session

In this section we describe a session using the interactive test generator on the sample filesystem.g in the samples/filesystem/models folder.

To start the session, issue the following command:

```
>> gotcha samples/fs_plurule/models/filesystem_p.g -itg
```

The screen then shows the following text:

```
Input Model: filesystem_p
Algorithm:
~~~~~~~~~~
         GOTCHA Test Generation in interactive mode
         Random Seed is 1556053837.
         Setting up interactive GOTCHA test generation
         Computing test start states and rules
         Firing unique start rule
Initialize()
         Entering unique start state
         Rules enabled in current state:
```

```
Rule 1(E0V2F0):
OpenForWrite(int f, char u1) || OpenForWrite(int f, char u2)
f = 1
u1 = a
u2 = b
Rule 2(E0V2F0):
OpenForRead(int f, char u1) || OpenForRead(int f, char u2)
f = 1
u1 = a
u2 = b
Rule 3(E1V0F0):
Close(int f, char u)
f = 1
u = a
Rule 4(E1V0F0):
Close(int f, char u)
f = 1
u = b
Rule 5(E0V1F0):
OpenForWrite(int f, char u)
f = 1
u = a
Enter m to see more enabled rules
Enter a Rule Number(1-8), c, b, r, s, p, ps, d, i, h, m, o, or q
```

The model has been compiled, and the interactive test generator has started running. When the message "Computing test start states and rules" appears, the system computes all the StartTestCase rules.

Since there is only one StartTestCase rule, it fires this rule to start the test case. The generator moves to the unique start state, and computes the rules that are currently enabled in this new state.

We see that eight rules are enabled, the first five are displayed, and the remainder can be displayed by typing m (more).

```
>> m
         Rules enabled in current state:
Rule 6(E0V1F0):
OpenForWrite(int f, char u)
f = 1
u = b
```

```
Rule 7(E0V1F0):
OpenForRead(int f, char u)
f = 1
u = a
Rule 8(E0V1F0):
OpenForRead(int f, char u)
f = 1
u = b
Enter a Rule Number(1-8), c, b, r, s, p, ps, d, i, h, m, o, or q
```

Typing the command h (Show Command Help) will give the full text of the commands available:

```
>> h
Choose a Rule Number(1-8) or one of the following:
c - show Current state,         b  - Back up one step,
r - Restart the current test,   s  - Save current test,
p - print current test,         ps - Print the whole test suite
d - Discard the test suite,     i  - show current Information,
h - show command Help,          o  - User interface options,
m - show More rules,            q - Quit GOTCHA.
Enter a Rule Number(1-8), c, b, r, s, p, ps, d, i, h, m, o, or q
```

Typing the commands i (information) and c (current state) will give the following text:

```
>> i
The current test suite has 0 test cases.
The current test case has 1 state and 1 rule.
The current state is an EndTestCase state.
Enter a Rule Number(1-8), c, b, r, s, p, ps, d, i, h, m, o, or q
>> c
         The state is:
second_reason = FileOpenForRead
main_reason = FileClosed
response = RejectCommand
system_status[1][a] = Closed
system_status[1][b] = Closed
Enter a Rule Number(1-8), c, b, r, s, p, ps, d, i, h, m, o, or q
```

Choosing Rule 1, which is a pluRule, will lead to one of two valid states – neither of which is an EndTestCase state or a forbidden state, since the text displayed after Rule1 is E0V2F0. On the other hand choosing Rule 3 (E1V0F0) will lead to a valid EndTestCase state. Choosing Rule 1 will cause the interactive GOTCHA session to enter into the StateWait condition, since there are two possible outcomes. Choosing Rule 3 will lead back into the RuleWait condition, since there is only one possible outcome to Rule 3.

First try Rule 1:

```
>> 1
         You have chosen Rule Number 1:
         Possible outcome states of rule chosen:
State 1(V):
second_reason = FileLocked
main_reason = FileOpenForWrite
response = OneAcceptOneReject
system_status[1][a] = Closed
system_status[1][b] = Writing
State 2(V):
second_reason = FileLocked
main_reason = FileOpenForWrite
response = OneAcceptOneReject
system_status[1][a] = Writing
system_status[1][b] = Closed
Enter a State Number (1-2), c, b, r, e, f, i, h, m, o, or q
```

You are now in the StateWait condition, and the options that you can choose from have changed accordingly. If you type h (Show Command Help), you will get a full list of the options:

```
>> h
Choose a State Number (1-2), or one of the following:
c - show Current state,           b - Back up one step,
r - Restart the current test,     e - show EndTestCase state
details,
f - show Forbidden state details, i - show current Information,
h - show command Help,            o - User interface Options,
m - show More states,             q - Quit GOTCHA.
Enter a State Number (1-2), c, b, r, e, f, i, h, m, o, or q
```

Choose State 1 to see what the next set of options will be:

```
>> 1
         You have chosen State Number 1:
         Rules enabled in current state:
```

```
Rule 1(E0V2F0):
OpenForWrite(int f, char u1) || OpenForWrite(int f, char u2)
f = 1
u1 = a
u2 = b
Rule 2(E0V2F0):
OpenForRead(int f, char u1) || OpenForRead(int f, char u2)
f = 1
u1 = a
u2 = b
Rule 3(E0V1F0):
Close(int f, char u)
f = 1
u = a
Rule 4(E1V0F0):
Close(int f, char u)
f = 1
u = b
Rule 5(E0V1F0):
OpenForWrite(int f, char u)
f = 1
u = a
Enter m to see more enabled rules
Enter a Rule Number(1-8), c, b, r, s, p, ps, d, i, h, m, o, or q
```

As before, eight rules are enabled. Choose Rule 3 to obtain the following outcome:

```
>> 3
         You have chosen Rule Number 3:
         Rules enabled in current state:
Rule 1(E0V2F0):
OpenForWrite(int f, char u1) || OpenForWrite(int f, char u2)
f = 1
u1 = a
u2 = b
Rule 2(E0V2F0):
OpenForRead(int f, char u1) || OpenForRead(int f, char u2)
f = 1
u1 = a
u2 = b
```

```
Rule 3(E0V1F0):
Close(int f, char u)
f = 1
u = a
Rule 4(E1V0F0):
Close(int f, char u)
f = 1
u = b
Rule 5(E0V1F0):
OpenForWrite(int f, char u)
f = 1
u = a
Enter m to see more enabled rules
Enter a Rule Number(1-8), c, b, r, s, p, ps, d, i, h, m, o, or q
```

Now choose Rule 4, which should lead to an EndTestCase state. This will produce a test that satisfies all the constraints, which you will then save to the current test suite, and print to an abstract test suite file.

Applying Rule 4 gives the following:

```
>> 4
         You have chosen Rule Number 4:
         Rules enabled in current state:
Rule 1(E0V2F0):
OpenForWrite(int f, char u1) || OpenForWrite(int f, char u2)
f = 1
u1 = a
u2 = b
Rule 2(E0V2F0):
OpenForRead(int f, char u1) || OpenForRead(int f, char u2)
f = 1
u1 = a
u2 = b
Rule 3(E1V0F0):
Close(int f, char u)
f = 1
u = a
Rule 4(E1V0F0):
Close(int f, char u)
f = 1
```

```
u = b
Rule 5(E0V1F0):
OpenForWrite(int f, char u)
f = 1
u = a
Enter m to see more enabled rules
         Current state is an EndTestCase state.
Enter a Rule Number(1-8), c, b, r, s, p, ps, d, i, h, m, o, or q
```

Entering s (Save) to save the current test case to the current test suite leads to the following dialog:

```
>> s
Do you wish to start a new test case? (y/n)
>> n
Enter a Rule Number(1-8), c, b, r, s, p, ps, d, i, h, m, o, or q
```

Answering no to the question allows you to continue a second test case from the end of the first test case. Had you answered yes, the current test case would start again from a starttestcase state. To verify, type i (for Information):

```
>> i
The current test suite has 1 test case.
The current test case has 4 states and 4 rules.
The current state is an EndTestCase state.
Enter a Rule Number(1-8), c, b, r, s, p, ps, d, i, h, m, o, or q
```

Now enter ps (Print the test suite). This adds the current test case to the current test suite and outputs the resulting suite to an abstract test suite file:

```
>> ps
Enter a file name for the abstract test suite:
>> pxx
Printing Current Test Suite to file pxx.ats
Do you wish to:
         start a new test case? (r)
         start a new test suite? (d)
         or continue? (c)
>> c
Continue interactive test generation with current suite and test
Enter a Rule Number(1-8), c, b, r, s, p, ps, d, i, h, m, o, or q
```

Choosing the option to continue retains the current test case and current test suite in memory. Choosing to start a new test case retains only the current test suite

Request information to clarify the situation after c before quitting:

```
>> i

The current test suite has 2 test cases.

The current test case has 4 states and 4 rules.

The current state is an EndTestCase state.

Enter a Rule Number(1-8), c, b, r, s, p, ps, d, i, h, m, o, or q
```

Now enter q to quit the interactive GOTCHA session:

```
>> q

You have chosen to quit the interactive GOTCHA session

Do you wish to save the current test in the current test suite?
(y/n)

>> n

Discarding Current Test Case

Do you wish to print the test suite before quitting the session?
(y/n)

>> n

Task completed.
```

3.3 Generating test suites quickly

The next step, after writing a behavioural model and testing directives, is to produce an abstract test suite. The GOTCHA components include several test generators. In this section we discuss four automated robust test generators.

These four test generators automatically generate an abstract test suite using implicitly defined coverage criteria. The fifth test generator is interactive and is discussed in the previous section on Interactive Test Generation. The most powerful test generators use full state space traversal algorithms with explicitly defined coverage criteria and they are discussed in Section 55.

The advantage of using the test generators discussed in this section is that they do not suffer from the state explosion problems encountered with full state space traversal of large models.

The test generators discussed here all have implicitly defined coverage models. This means that there is no need for the user to explicitly specify a coverage model, and there is no need for the user to be actively involved in the test selection.

The default test generation strategy used by GOTCHA is input coverage, which can be invoked by the following command:

GOTCHA *modelname.g* **-o** *modelname*

This will create a test suite *modelname.ats*, using the input coverage test generator with default values of its parameters. This assumes that the installation directory has been set as an environment variable named `GOTCHA_HOME`, and that the behavioural model is in the file *modelname.g*. The abstract test suite is written to a file entitled *modelname.ats*.

There are four speedy test generators supplied by GOTCHA:

- Random
- Input coverage
- Input transition coverage
- Input transition pair coverage

Each of these test generators generates a set of test cases for any model, irrespective of the state explosion problem.

They all ignore any coverage criteria and TC_EndTestCase constraints in the input model (see Section 4.1 Explicitly specifying coverage models and Section 4.2 Writing Testing Directives).

The **random test generator** generates a set of randomly chosen transitions in each test case.

The **input coverage test generator** choses test cases in such a way that all input values of all rules appear somewhere in the test suite.

The **input transition coverage test generator** choses test cases in such a way that all input values of all rules appear in every transition position in the test suite.

The **input transition pair coverage test generator** choses test cases in such a way that all pairs of input values of all rules appear in every pair of transition positions in the test suite.

The latter three test generators are ordered by increasing thoroughness of the test suite generated. This means that the coverage tasks implicitly defined for the input coverage are contained in the tasks defined for the input transition coverage, which are in turn contained in the tasks defined for the input transition pair coverage.

If a model has two rules: Rule1() which is not contained in a ruleset, and Rule2(bool b) which is contained in a ruleset with a Boolean variable, b, as input, and furthermore, each of the rules is enabled at all states, then:

The input coverage generator would generate only one test case:

```
Rule2(true) Rule1, Rule2(false).
```

The input transition coverage generator would generate three test cases:

```
Rule2(true), Rule1, Rule2(false).

Rule2(false), Rule2(true), Rule1.

Rule1, Rule2(false), Rule2(true).
```

The input transition pair coverage generator would generate nine test cases:

```
Rule2(true), Rule1, Rule2(false).

Rule2(false), Rule2(true), Rule1.

Rule1, Rule2(false), Rule2(true).

Rule2(true), Rule2(true), Rule2(true).

Rule2(false), Rule2(false), Rule2(false).
```

```
Rule1, Rule1, Rule1.
Rule2(true), Rule2(false), Rule1.
Rule2(false), Rule1, Rule2(true).
Rule1, Rule2(true), Rule2(false).
```

In the rest of this section we give more details on each of the speedy test generators.

3.3.1 Random test generation

The random test generator provided by GOTCHA can be invoked from the command line as follows:

> **GOTCHA** *modelname.g* **-o** *modelname* **-rtg**

This will create a test suite *modelname.ats*, with ten test cases each containing at most ten randomly chosen transitions. To control the number of test cases generated, and the number of transitions in each test case, use the command:

> **GOTCHA** *modelname.g* **-o** *modelname* **-rtg -st5 -gtlub15**

This will create a test suite with 5 test cases each containing 15 randomly chosen transitions.

Invoking the random test generator without specifying a random seed will produce different results on each invocation. To specify the random seed, and thus guarantee repeatable results use the flag –rs as shown below:

> **GOTCHA** *modelname.g* **-o** *modelname* **-rtg -rs1234**

The output of the random test generator looks like the following:

```
      ============================================================
      GOTCHA Release 4.0
      Generation Of Test Cases from Haifa.
      Copyright (C) 1999,2002
      by the IBM Corporation.
      ============================================================
      Input Model: filesystem.g
      Test Generator:
               GOTCHA Random Test Generator
      WARNING: The Random Test Generator ignores all
TC_EndTestCase constraints
      WARNING: The Random Test Generator ignores all CC_ coverage
criteria
               Number of test cases 10.
               Test Case Length 10.
               Random Seed is 1234.
```

```
        Random Test Case generation completed
        Generated 10 random test cases
```

3.3.2 Input coverage test generation

The input coverage test generator can be invoked from the command line as follows:

GOTCHA *modelname.g* **-o** *modelname* **-rctg**

This will create a test suite *modelname.ats*, using the input coverage test generator with default values of its parameters. The default configuration for this test generator generates at most ten test cases each containing at most ten transitions, and attempts to cover each rule and start state with each possible input (ruleset values) at least once.

The input coverage generator has the following input parameters:

- Test case length, controlled by the flag **-gtlub** (default value 10).
- Limit on the number of test cases generated, controlled by the flag **-st** (default value 10).
- The coverage factor, which controls the number of times each rule and startstate is to be covered. Controlled by the flag **-gtcf** (default value 1).
- The random seed, controlled by the flag **-rs** (default value, time of day).

The input coverage test generator attempts to use every rule and every start state with every possible combination of inputs at least c times in the test suite, where c is the coverage factor. It will stop without achieving 100% coverage if it reaches its limit on the number of test cases. It reports on the coverage achieved as follows:

```
Number of Coverage Tasks:
        Start Rules: 1
        Transition Rules: 27

Number of start rules completely covered 1
Start rule coverage: 100%
Number of transition rules completely covered 27
Transition rule coverage: 100%

Number of test cases generated 3
```

The *number of start rules completely covered* indicates the number of start rules that appear at least c times in the test suite, where c is the coverage factor. Likewise the number of transition rules completely covered is the number of transition rules that occur at least c times in the test suite.

3.3.3 Input transition coverage test generation

The input transition coverage test generator can be invoked from the command line as follows:

```
GOTCHA modelname.g -o modelname -rtctg
```

This will create a test suite *modelname.ats*, using the input transition coverage test generator with default values of its parameters. The default configuration for this test generator generates at most ten test cases each containing at most ten transitions, and attempts to cover each rule and its inputs in each transition position at least once.

The input transition coverage generator has the following input parameters:

- Test case length, controlled by the flag **`-gtlub`** (default value 10).
- Limit on the number of test cases generated, controlled by the flag **`-st`** (default value 10).
- The coverage factor, which controls the number of times each rule is to be covered in each transition position. Controlled by the flag **`-gtcf`** (default value 1).
- The random seed, controlled by the flag **`-rs`** (default value, time of day).

The input transition coverage test generator attempts to use every rule with every possible combination of inputs at each transition position at least c times in the test suite, where c is the coverage factor. It may stop after reaching its limit on the number of test cases. It reports on the coverage achieved as follows:

```
Estimated number of Coverage Tasks: Upper Bound 244 Lower Bound
244

Number of tasks completely covered 91

Coverage estimate: Between 37% and 37%

Number of test cases generated 10
```

The *number of input transition coverage tasks* is estimated by the number of enabled rules observed at each transition position. At different times in the generation process, different numbers of rules may be enabled at a particular transition in the test case. At the first step of a test case, the number of start rules enabled is always the same. However, the number of transition rules enabled may vary according to the state variables which determine whether or not the guards on the transition rules are enabled.

Note that the lower bounds and upper bounds count the minimum and maximum numbers of rules enabled at a particular transition – they do not count the number of distinct rules enabled at that position. So if at some stage only Rules 1 and 2 are enabled at position 1, and at another stage Rules 1 and 3 are enabled, the minimum and maximum number of rules enabled will still be 2, and the task will be considered as covered if two distinct rules at that position are exercised. Thus the upper bound on the number of tasks may be underestimated.

The input transition coverage generator stops generating test cases either when the maximum number of test cases has been generated, or when the lower bound on the number of coverage tasks has been achieved.

3.3.4 Input transition pair coverage test generation

The input coverage test generator can be invoked from the command line as follows:

```
GOTCHA modelname.g -o modelname -rpctg
```

This will create a test suite *modelname.ats*, using the input transition pair coverage test generator with default values of its parameters. The default configuration for this test generator generates at most 100 test cases each containing at most five transitions, and attempts to cover each pair of rules and their inputs in each pair of transition positions at least once.

The input transition pair coverage generator has the following input parameters:

- Test case length, controlled by the flag **`-gtlub`** (default value 5).
- Limit on the number of test cases generated, controlled by the flag **`-st`** (default value 100).
- The coverage factor, which controls the number of times each rule is to be covered in each transition position. Controlled by the flag **`-gtcf`** (default value 1).
- The random seed, controlled by the flag **`-rs`** (default value, time of day).

The input transition pair coverage test generator attempts to use every pair of rules with every possible combination of inputs at each pair of transition positions at least c times in the test suite, where c is the coverage factor. It may stop after reaching its limit on the number of test cases. It reports on the coverage achieved as follows:

```
Estimated number of Coverage Tasks: Upper Bound 4482 Lower Bound
4482

Number of tasks completely covered 708

Coverage estimate: Between 15% and 15%

Number of test cases generated 100
```

The *number of input transition pair coverage tasks* is estimated by the product of pairs of the number of enabled rules observed at each pair of transition positions. At different times in the generation process, different numbers of rules may be enabled at a particular transition in the test case. At the first step of a test case, the number of start rules enabled is always the same. However, the number of transition rules enabled may vary according to the state variables which determine whether or not the guards on the transition rules are enabled.

Note that the lower bounds and upper bounds count the minimum and maximum numbers of rules enabled at a particular transition – they do not count the number of distinct rules enabled at that position. So if at some stage only Rules 1 and 2 are enabled at position 1, and at another stage Rules 1 and 3 are enabled, the minimum and maximum number of rules enabled will still be 2. If the same thing happens at position 2 of the test case, then the minimum number of pairs will be estimated as 4, even though there are potentially 9 possible combinations. Thus the upper bound on the number of tasks may be underestimated.

The input transition pair coverage generator stops generating test cases either when the maximum number of test cases has been generated, or when the lower bound on the number of coverage tasks has been achieved.

3.3.5 The parameters for quick test generation

The runtime parameters for quick test generation fall into four categories:

1. The input file names
2. The output file name
3. The test generation algorithm
4. The optional parameters

A typical command line invocation of GOTCHA is as follows:

GOTCHA *modelname.g* **–o** atsname **<algflag> <optflags>**

Where the input file is modelname.g, the test cases will be output to the file atsname.ats, the algflag is an algorithm flag – one of the five quick algorithms (-rtg, -rctg (the default), -rtctg, -rpctg, or –itg), and the optflags sets values for the optional parameters of the algorithm selected. The four algorithms defined by the flags -rtg, -rctg, -rtctg, and -rpctg, are discussed in the previous four subsections. The interactive test generator (-itg) is discussed in the next section.

3.3.5.1 Specifying the input files

Any argument without a – is taken to be an input filename and must end in .g. Up to ten input files can be specified, and each input file may include any number of #include macros to include other .g files. The input files are processed in the order that they are specified on the command line.

3.3.5.2 Specifying the output files

The runtime argument **–o** followed by a space and a string, say **atsname**, specifies that the abstract test suite will be output to a set of files whose root is the file **atsname.ats**.

3.3.5.3 Specifying the quick generation algorithm

The quick generation algorithms are specified by selecting one of the following runtime parameters:

-rtg	Random test generator
-rctg	Input coverage test generator
-rtctg	Input transition coverage test generator
-rpctg	Input transition pair coverage test generator
-itg	Interactive test generator

3.3.5.4 Specifying the optional runtime parameters

The optional runtime parameters for the four main test generation algorithms are used to control the number of tests cases generated, the length of the test cases, and the number of times that each coverage task is covered.

-st**<n>**	Stop after n test cases generated
-gtlub<n>	Test cases of length n
-gtcf<n>	Coverage factor (generate n tests for each task)

```
-rs<n>       Use n as the seed for randomization
```

The **–st** flag puts an upper bound on the number of test cases to be generated by any of the test generation algorithms. Do not leave a space between the flag and its parameter.

Invoking GOTCHA with the flag **–st15** will ensure that no more than 15 test cases are generated by the test generator. If the coverage goals are reached before the number of test cases is exhausted, the test generator will stop before generating 15 test cases. The default number of test cases depends on the test generator being used, see the following table of default values:

Test Generator	**Default maximum number of test cases**
Random	10
Input Coverage	10
Input Transition Coverage	10
Input Transition Pair Coverage	100

The **gtlub** parameter specifies the number of transitions to be generated in each test case. All test cases will be of the length specified unless the test case reaches a state where no more transitions are possible. Invoking GOTCHA with the flag **–gtlub15** will ensure that all test cases have at most 15 transitions (counting the start transition). All test cases will have precisely 15 transitions unless the model itself has limits on the number of transitions.

The model and the algorithm determine a number of coverage tasks which must be covered by the test cases generated. Increasing the coverage factor requires that the generator cover each of these tasks more times. Invoking GOTCHA with the flag –**gtcf3** and input coverage will ensure that each input occurs three times in the test suite (provided this can be done within the number of test cases specified by the –st parameter). The flag has no impact on the random test generator.

There is an additional parameter to control the random number generator used. Specifying the same random seed will guarantee that the same test cases are output. If the random seed is not specified, the random number generator is seeded by the time of day, and different test cases will result from each invocation of the algorithm.

3.3.6 Coverage Criteria and Test Constraints

When using the "quick" test generation algorithms (random, input coverage, input transition coverage, and input pair transition coverage) or the interactive test generator, all coverage criteria in the model are ignored. However all test constraints, with the exception of TC_EndTestCase, are respected.

The quick test generators create test cases that may end at a state that satisfies an EndTestCase constraint or not. If the test case does end at an EndTestCase state, the EndTestCase condition will be noted in the Abstract Test Case.

4 Writing Test Generation Directives

4.1 Explicitly specifying coverage models

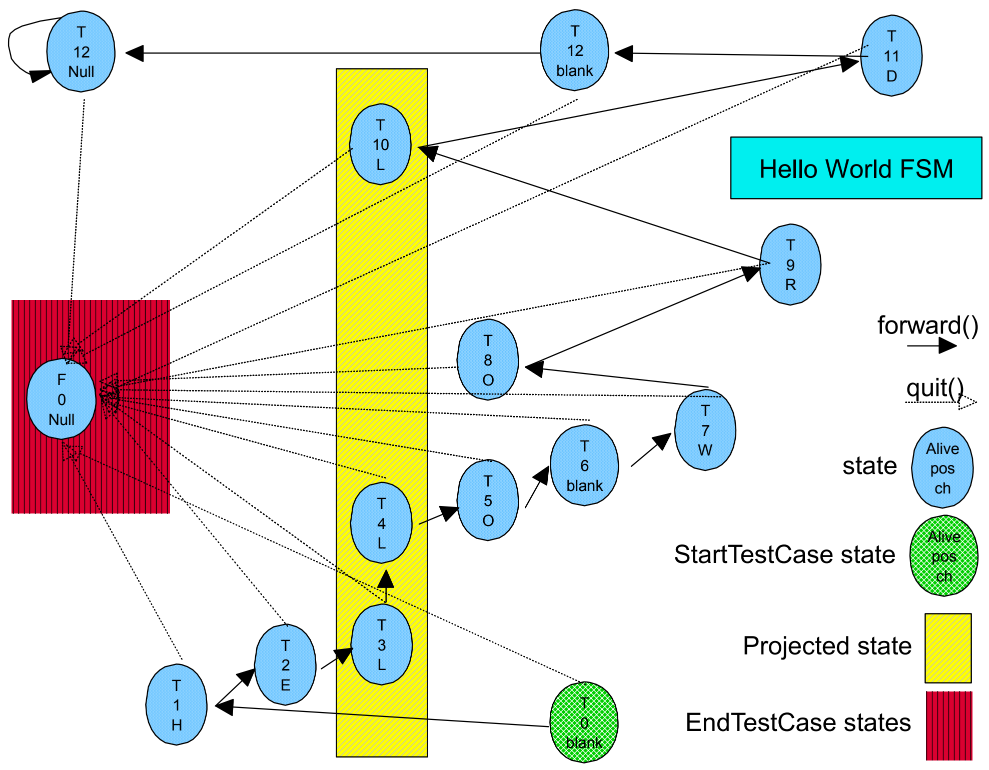
The purpose of this section is to clarify the concepts of:

- Projected State
- Reachable State
- Coverable State
- Coverage Task

These concepts are defined in the Glossary – but they may be easier to assimilate through this example.

In order to explicitly specify a set of coverage tasks, we add the following code to our Hello World model:

```
-- The Test Constraint for ending a test case
TC_EndTestCase "afterquit()"
      !alive;

-- A Coverage Criterion
CC_State_Projection
      TRUE
      On
      ch;
```

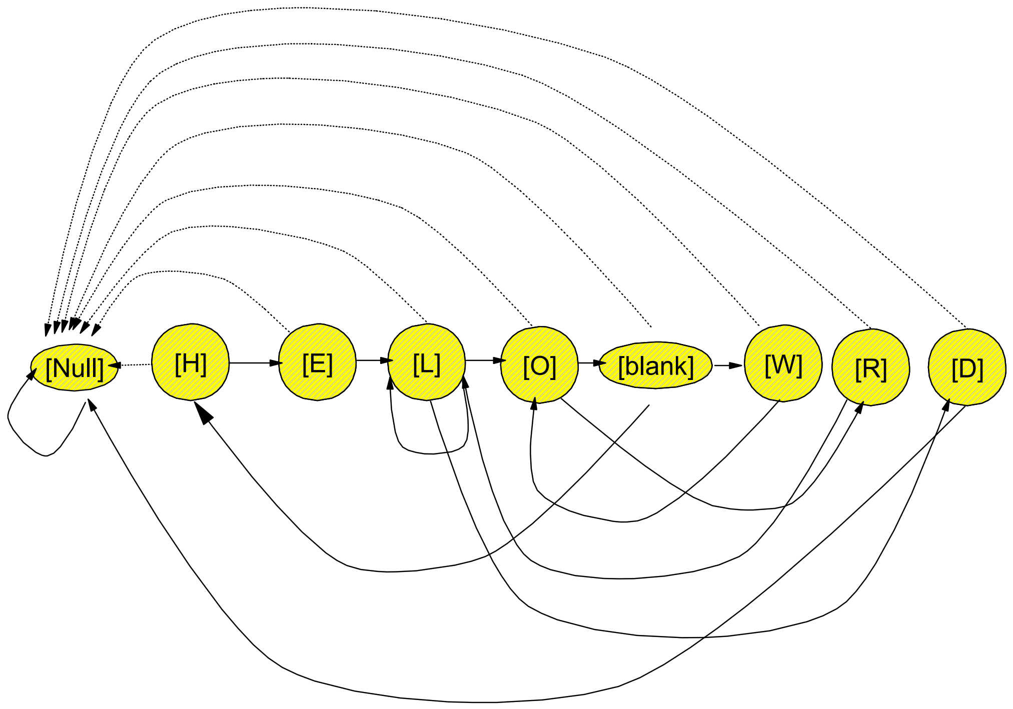


The start test case state is coloured green, intermediate states are blue and the set of all end test case states (all states with alive = FALSE) is coloured red. The set of all states with `ch = L` are included in the yellow shaded column. Each column of the diagram represents a "projected state".

The **`TC_EndTestCase`** clause above is a Boolean expression and not an assignment statement. It is shorthand for: "Any state where the value of the expression is TRUE is a legitimate place to finish a test." In this example we can finish a test case whenever the `alive` variable is FALSE.

The **`CC_State_Projection`** clause describes the fact that we want to project all of the states in our state machine onto the values of the state variable `ch`. Thus each column is collapsed into a single "projected state". The arc from (L, 3, true) to (L, 4, true) becomes a loop in the projected space from [L] to itself. The symbol [L] represents all three states with first coordinate equal to L.

The projected state graph is illustrated below:



This projected state machine graph is derived from the full state machine graph by projecting all the states in a single column onto a single projected state.

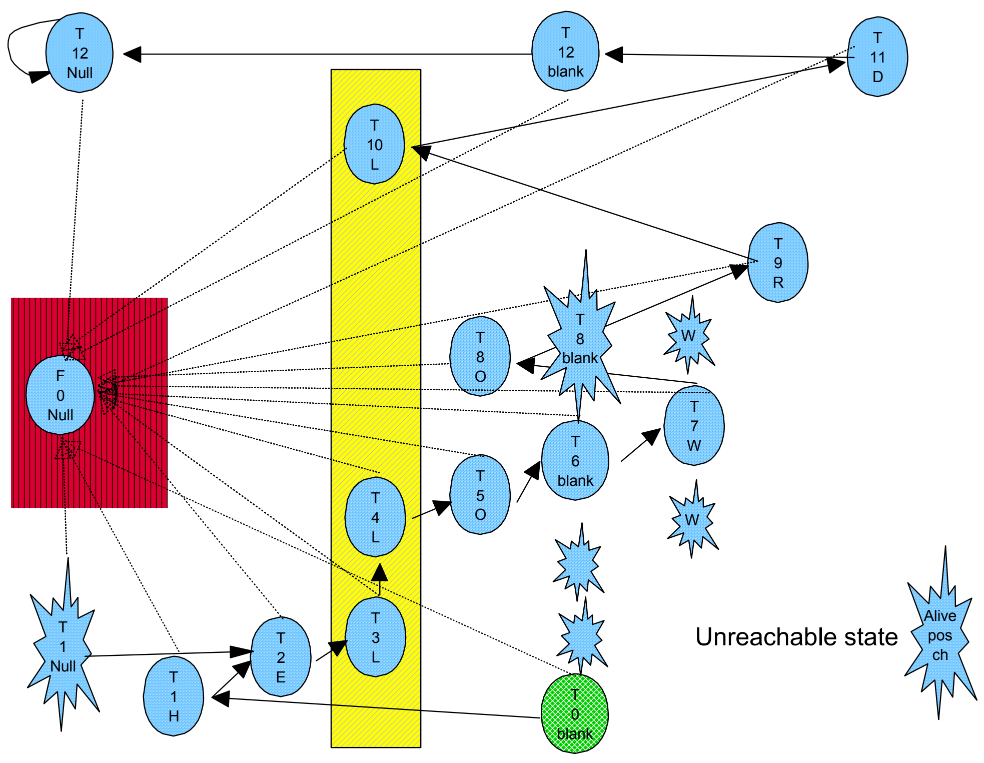
4.1.1 Reachability

The concepts of reachable and unreachable states are illustrated in the next diagram.

A state is **reachable** if there is a sequence of transitions (rules) that produces the state from a start test case state. So for example the state `(L, True 3)` is reachable since the sequence:

```
1. initialize() (blank, True 0)
2. forward() (H, True 1)
3. forward() (E, True 2)
4. forward() (L true 3)
```

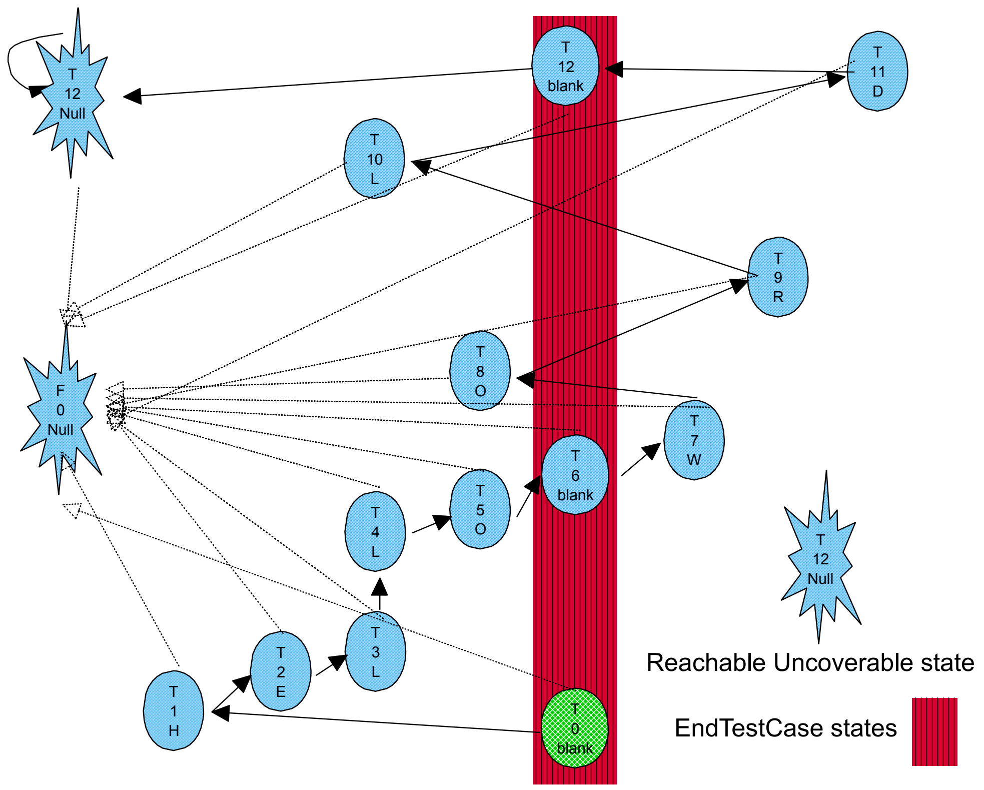
starts from a StartTestCase state `(blank, True 0)` and reaches `(L, True 3)` by applying the `forward()` rule three times.



The states `(Null, True 1)` and `(blank, True 8)` are unreachable, despite the fact that they contain legitimate values of all three state variables. If the application ever entered one of these states, it would indicate a bug, since these states can never be reached in the model.

4.1.2 Coverability and Coverage Tasks

We now consider a slightly modified Hello World example in order to illustrate the concepts of uncoverable and coverable states and coverage tasks. Let us assume that instead of `TC_EndTestCase !alive;`, we have `TC_EndTestCase ch=blank;`. This situation is illustrated in the figure below.



A state is **coverable** if there exists a path from a StartTestCase state to the state in question (i.e. it is reachable), and then on to an EndTestCase state.

In the above situation, `(Null, True 12)` and `(Null, False 0)` are reachable states – since they can be reached from the StartTestCase state. However, they are **uncoverable**, since there is no path from either of these states to an EndTestCase state (i.e. one where `ch = blank`).

All the other reachable states are coverable, since a sequence of twelve `forward()` transitions passes through each of the other reachable states from the StartTestCase state. This sequence ends at the state `(blank, True 12)`, which is an EndTestCase state since it satisfies `ch = blank`, and thus all the states in this sequence are coverable.

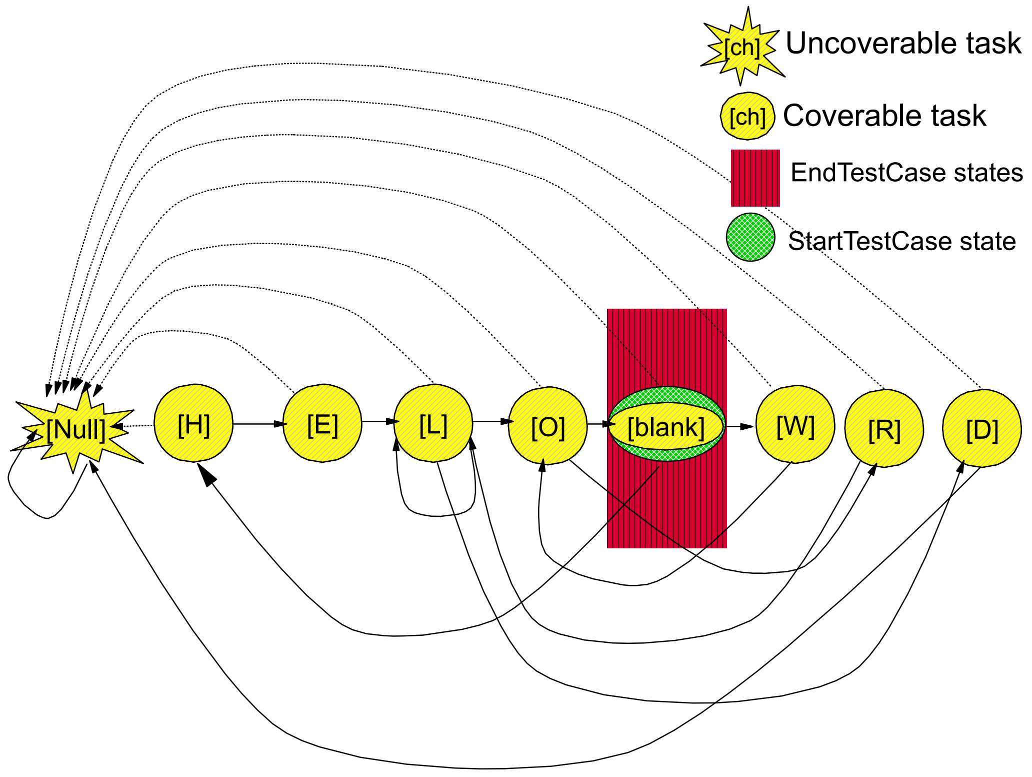
A **coverage task** is a set of states or transitions all of which satisfy some instance of a coverage criterion.

In the case of State Projection coverage (defined below), a coverage task is a projected state – or, equivalently, the set of all states in the full state machine that map onto a state in the projected graph. Each member of the set is called a **representative** of the coverage task.

In our example there are nine coverage tasks, one for each reachable value of the projection variable ch. The coverage task **[H]** has only one representative `(H, True 1)`, but the coverage task [blank] has three representatives `(blank, True 0) (blank, True 6)` and `(blank, True 12)`.

A coverage task is **uncoverable** if all its representatives are uncoverable. Or equivalently a coverage task is **coverable** if any one of its representatives is coverable.

In our example (with the changed EndTestCase condition), the coverage task [Null] is uncoverable while the other eight coverage tasks are all coverable.



The test case consisting of six `forward()` transitions starts at the StartTestCase state, ends at an EndTestCase state and covers the tasks [blank], [H], [E], [L], and [O]. The tasks [blank] and [L] are each covered twice by this test case. The test case consisting of twelve `forward()` transitions covers all eight coverable tasks. There is no test case satisfying the test constraints that includes any representative of [Null], and so this task is uncoverable.

4.2 Writing Testing Directives

Testing directives have the same syntax as rules, but they are used specifically to drive the test generation process – not to describe the finite state machine or state graph, which is the behavioural model of the software under test. We recommend that you keep the testing directives in a .g file separate from the behavioural model .g file(s).

Testing directives come in two types: coverage criteria and test constraints. Note that TC_StartTestCase is a very special case of test constraint.

4.2.1 Test Constraints

<testconstraint> ::= TC_EndTestCase < string > <boolexpr>

TC_Forbidden_State [<string>] < boolexpr > |

```
TC_Forbidden_Transition [<string>] < boolexpr >; < boolexpr > |
TC_Forbidden_Transition [<string>]
      [From] < boolexpr > To < boolexpr > |
TC_Forbidden_Path [<string>] < boolexpr >; < boolexpr >; <expr> |
TC_Forbidden_Path [<string>]
      [From] < boolexpr > To < boolexpr > Length <expr>|
TC_Within [<string>] < boolexpr >; < boolexpr >; < boolexpr > |
TC_Within [<string>]
      [From] < boolexpr > To < boolexpr > Includes < boolexpr >
```

The test constraints all have an optional string that is used in the abstract test suite as a name for the test constraint. This is especially important in the case of a TC_EndTestCase constraint, which appears at the end of every test case.

The expressions must all be Boolean expressions in the variables within the scope of the constraint. The only exception is the length expression in the forbidden path constraint – which must evaluate to an integer in the range 2–7.

4.2.1.1 End Test Case

TC_EndTestCase < string > <expr>

EndTestCase states are the last state in any test generated by the full traversal test generator, thus they are effectively a test constraint. Every test generated with the full traversal algorithm must finish in a state where some End Test Case expression evaluates to True.

If the model contains no TC_EndTestCase constraint, then the GOTCHA compiler inserts a default constraint that makes every state an EndTestCase state.

```
TC_EndTestCase "Default" TRUE;
```

It is an error to use an expression with side effects in a TC_EndTestCase expression. An example of such an illegal action would be to call a (Boolean) function that changes the value of a state variable.

NOTE: The GOTCHA compiler gives a warning only when a function call has such side effects – so be extra careful!

The string is output to the test case as a method pattern, and any parameters to the TC_EndTestCase (should it be enclosed in a ruleset) are output as data patterns in the abstract test suite.

4.2.1.2 Forbidden State

The semantics of a **Forbidden State** test constraint are that: No test will be generated that passes through a state where the expression is TRUE.

NOTE: This constraint is applied throughout the test generation, and it can have a dramatic effect in reducing the size of the reachable state graph. It is a key element in controlling the size of the test suite and in dealing with the state explosion problem.

Specifying `TC_Forbidden_State var1=3;` removes all states where `var1=3` from the state space and thus from all the tests generated. Any state that can only be reached via states with `var1=3` is also eliminated.

Specifying `TC_Forbidden_State TRUE`; causes GOTCHA to generate no tests since all paths pass through a forbidden state.

4.2.1.3 Forbidden Transition

The semantics of a **Forbidden Transition** test constraint are that: No test will be generated that passes through a transition where the first expression is true before the transition, and the second is true after the transition.

NOTE: This constraint is applied throughout the test generation, and it can decrease the size of the reachable state graph. It is a method for controlling the size of the test suite and dealing with the state explosion problem.

Specifying `TC_Forbidden_Transition var1=3; var1!=3` will remove all transitions from the state space when var1 is altered from 3 to some other value by the transition. Any state that can only be reached via such a transition will also be eliminated.

4.2.1.4 Forbidden Path

The semantics of a **Forbidden Path** test constraint are that: No test will be generated that passes from a state where the first expression is true to a state where the second is true in k or fewer steps, where k is the value of the third expression.

Note that when k=1 this is just a forbidden transition and the forbidden transition construct should be used.

NOTE: This constraint is applied throughout the test generation. It can have a positive or negative effect on the size of the reachable state space, since an additional state variable is introduced to keep track of the constraint. This additional variable can be seen in the abstract test suite and has a name that contains the user's name for the constraint. The longer the path, the more likely this constraint is to increase the size of the state space.

Specifying `TC_Forbidden_Path var1=3; var1!=3; 4;` will remove all execution sequences which alter var1 from 3 to another value in four or fewer steps. Any state that can only be reached via such a sequence will also be eliminated.

4.2.1.5 Within

The semantics of a **Within** test constraint are that: No test will be generated that passes from a state where the first expression is true to a state where the second is true, without passing through a state where the third expression is true.

NOTE: This constraint is applied throughout the test generation. It can have a positive or negative effect on the size of the reachable state space, since an additional state variable is introduced to keep track of the constraint. This additional variable can be seen in the abstract test suite and has a name that contains the user's name for the constraint.

Specifying

```
TC_Within "Interesting" From Cmd=FileOpen To Cmd=FileClose
Includes Cmd=Write & nbytes > 0;
```

will cause all test sequences, which contain a Close command following an Open command to include a Write command with a non-zero nbytes.

4.2.2 Coverage Criteria

<coveragecriterion> ::=

CC_Some_State [<string>] <boolexpr> |

CC_Some_Transition [<string>] <boolexpr>; <boolexpr> |

CC_Some_Transition [<string>]

[From] <boolexpr> To <boolexpr> |

CC_Some_Explicit_Transition [<string>]

[From] <boolexpr>; [Via] <boolexpr>; [To] <boolexpr>; |

CC_State_Projection [<string>]

<boolexpr> On <type_exprs_pairs> ; |

CC_Transition_Projection [<string>]

From_condition <boolexpr> From <type_exprs_pairs> ;

To_condition <boolexpr> To <type_exprs_pairs> ; |

CC_Projected_Explicit_Transition [<string>]

From_condition <boolexpr> From <type_exprs_pairs> ;

Via_condition <boolexpr> Via <type_exprs_pairs> ;

To_condition <boolexpr> To <type_exprs_pairs> ; |

CC_All_State [<string>] <boolexpr> |

CC_All_Transition [<string>] <boolexpr>; <boolexpr> |

CC_All_Transition [<string>] [From] <boolexpr> To <boolexpr>

<type_exprs_pairs> ::= <type_expr_pair> {;[<type_expr_pair>]}

<type_expr_pair> ::= expr : typeid | nonint_type_expr

Coverage criteria are a way to direct the full traversal test generator. They supply criteria for what tests to generate (as opposed to test constraints, which tell the generator which tests not to generate).

The coverage criteria all have an optional string that is used only in the printout as a name for the criterion. The expressions denoted by boolexpr must all be Boolean expressions in the variables within the scope of the criterion.

The <type_exprs_pairs> token stands for a semicolon delimited string of tokens, each of which is either a simple expression or a simple expression followed by its type identifier. The type identifiers are only necessary when the expression is of integer subrange type, otherwise, the type identifier can be omitted since the compiler can deduce the type of the expression. Note also that a Boolean expression is a special case of an enumerated type expression.

Some_Explicit_Transition and Projected_Expilict_Transition criteria permit the use of a special reserved keyword RuleId in their "via" condition and projection parts. They also contain some special additional options and limitations as opposed to other coverage criteria. Please, see more information in the corresponding sections.

4.2.2.1 Some State

Some State coverage is a criterion that describes a single coverage task. The task is specified by the Boolean expression, which is evaluated at each state to decide whether or not it is a representative of the coverage task. Any state where the Boolean expression evaluates to TRUE is a representative of this coverage task, and GOTCHA will generate a test case that passes through a randomly chosen representative state.

Specifying `CC_Some_State TRUE;` will cause GOTCHA to generate a single test through a randomly chosen state. If no coverage criteria are given in the model, this coverage criterion is the default that is added to the model.

Specifying `CC_Some_State "Interesting" var1=3 & var2=4;` will cause GOTCHA to generate a single test, which includes a randomly chosen state where var1=3 and var2=4 (if such a state, which is both reachable and coverable, exists).

4.2.2.2 Some Transition

Some Transition coverage also describes a single coverage task. The task is specified by two Boolean expressions, which are evaluated at the beginning and end (respectively) of each transition, to decide if it is a representative of the coverage task or not. Any transition from *s* to *t* where the FROM expression evaluates to TRUE on *s* and the TO expression evaluates to TRUE on *t* is a representative of the task.. GOTCHA will generate a test case that passes through a randomly chosen representative transition.

Specifying `CC_Some_Transition TRUE; TRUE;` - will cause GOTCHA to generate a single test through a randomly chosen transition, since every transition is a representative of this task.

Specifying `CC_Some_Transition "Interesting" var1=3; var2=4;` will cause GOTCHA to generate a single test which includes some transition from a state where var1=3 to a state where var2=4 (if such a transition, which is both reachable and coverable, exists).

4.2.2.3 Some Explicit Transition

Some Explicit Transition coverage also describes a single coverage task. The task is specified by three Boolean expressions. A transition from *s* to *t*, performed by the application of rule *r*, may be regarded as a representative of the coverage task if all of the following conditions are satisfied:

- The FROM expression evaluates to TRUE on *s*
- The TO expression evaluates to TRUE on *t*.
- The VIA expression also evaluates to TRUE on *r*.

The VIA expression has some special properties – which make it somewhat different from the TO and FROM expressions.

- The VIA expression can utilize the keyword ruleID, which can be regarded as a variable, containing the id of the applied rule. It can also use names of model rules as part of the expression (specifically to compare ruleID with the specific rule names). Integer values are assigned to all the rule names inside the GOTCHA model. The later the rule is defined in the model code – the higher the value.

  Example: VIA (ruleID > (“some_rulename”))

  In this case the rule *r* must satisfy the condition (*r* > (“some rulename”)) in order to be a representative of the coverage task. The string “some rulename” must be the name of some rule, defined within the GOTCHA model. All rules defined in the model body *after* the rule “some rulename” will satisfy this condition.
- The VIA expression can reference values of state variables, defined in the GOTCHA model. These values are calculated in the FROM state. (Of course, their value may change in the TO state of the transition).
- The VIA expression can reference values of rule parameters. Caution must be used, so as not to reference a parameter, that is not applicable to the corresponding rule. The VIA expressions are calculated in the C- style manner. This means, that if partial calculation of the expression (according to the precedence rules) is enough to deduce its value – the computation is terminated and the remaining part of the expression is ignored. For example, a | b will calculate only the “a” part of the expression if a evaluates to TRUE – as this is enough to deduce the value of a | b - and both parts if “a” evaluates to FALSE

  Example: (part of GOTCHA model)

  ```
  Ruleset i : 1 .. 3
  Do
    Rule “abc”
       :
   End;
   Rule “cde”
  ```

:

Example 1: VIA (i > 1) --incorrect

This is an incorrect use of a rule parameter within a coverage criterion, since parameter i is not defined for all the model rules. An attempt to evaluate it for rule “cde” to see if it satisfies the coverage model will fail.

Example 2: VIA ((ruleID = (“abc”)) & (i > 1)) --correct

In this case only if the rule *r* is “abc” the term (i > 1) is evaluated. Parameter i is defined for rule “abc”.

Example 3: VIA ((i > 1) & (ruleID = (“abc”))) --incorrect

In this case the (i > 1) expression is evaluated for all the rules, before the ruleID is checked to be “abc”. Hence, it is also evaluated for “cde” – which results in failure.

Note that rule parameters with the same name must be of the same type. In case of integer subranges, these types may have different names, but must have the same range.

GOTCHA will generate a test case that passes through a randomly chosen representative transition that satisfies all the three Boolean conditions (TO, FROM and VIA).

Specifying `CC_Some_Explicit_Transition TRUE; TRUE; TRUE;` - will cause GOTCHA to generate a single test through a randomly chosen transition, created by applying a randomly choosen rule, since every transition + corresponding rule combination is a representative of this task.

Specifying `CC_Some_ Explicit_Transition "Interesting" var1=3; VIA ruleID = “abc”; var2=4;` will cause GOTCHA to generate a single test which includes some transition from a state where var1=3 to a state where var2=4 (if such a transition, which is both reachable and coverable, exists), applying a rule, which has a name “abc” in the Gotcha model with any of its possible input parameters.

4.2.2.4 State Projection

State Projection coverage describes a set of coverage tasks and not a single task. Each task is an equivalence class of states of the finite state machine. The projection variables are the expressions given in the list of expressions. The condition provides a method for further subsetting the set of tasks and their representatives.

The concept of projected state coverage is explained more fully in the section entitled: A Sample Model – Hello World.

For example to specify coverage tasks in the Hello World model projected onto the values of the `ch` variable, as in the example above, we would use the following coverage criterion:

```
CC_State_Projection TRUE On ch;
```

This means that each coverage task is specified by the different values taken by the variable `ch`. So one coverage task is represented by any state with `ch =A`, another by all states with `ch =B`, up to a maximum of 26 coverage tasks. In the example, the coverage task with `ch =L` has three representatives.

If we had specified:

```
CC_State_Projection NumOhs=1 On ch;
```

Then only two reachable states would satisfy the condition `(O,2,1)` and `(W,2,1)`, where each state is denoted by the triple `(ch, numEls, numOhs)`. And since each of these is in a different coverage task (has different values for `currenChar`), GOTCHA would attempt to generate two test cases – one through each of these states. (In practice it would only generate the first test case, since this test covers both coverage tasks in this simple model.)

With this semantics for State Projection, you can designate several projections in the same model. You are not limited to state variables, and can define implicit variables for projection.

For example if the state contains two integer variables x and y, but you are interested in test cases where the values of the sum x+y are distinct, then you would use the coverage criterion:

```
CC_State_Projection TRUE On x+y:sum_range_t;
```

Specifying `CC_State_Projection TRUE On var1; var2; var3`; - is equivalent to the projected state coverage available in previous releases of GOTCHA, where var1, var2, and var3 are all Boolean coverage variables.

An upper limit on the number of coverage tasks generated by a State Projection criterion of the form:

```
CC_State_Projection boolexpr On var1; var2; var3;
```

is the product of the ranges of the three variables. If all three variables were Boolean, then up to 2*2*2 = 8 tasks would be generated by this coverage criterion. The actual number of tasks, and hence test cases, generated could be smaller, for one of the following reasons:

- The subsetting effect of the Boolean expression may reduce the number of representatives of a task to zero.
- Not all eight tasks may be reachable from a StartTestCase state.
- More than one of the tasks may be covered by a single test case.
- No reachable representative of a task is actually coverable.

4.2.2.5 Transition Projection

Transition Projection also describes a set of coverage tasks in a similar way to State Projection. It is as though we are projecting the state space onto two different sets of variables whose values are given by the expressions in the lists. We consider the transitions of interest to be those whose projections are distinct, when their first state is projected onto the first list, and their second state is projected onto the second list. Moreover, we are able to specify different subsetting conditions on each of the two projections.

The criterion below is equivalent to the projected transition coverage available in previous releases of GOTCHA, where var1, var2, and var3 are all Boolean coverage variables.

```
CC_Transition_Projection

From_Condition TRUE From var1; var2; var3;
```

```
To_Condition TRUE To var1; var2; var3;
```

Let us assume that BoolArray is an array of Boolean variables, and that enumvar1 is of an enumerated type with three possible values, v1, v2, and v3. Then specifying

```
CC_Transition_Projection "Interesting"
From_Condition system=stable From BoolArray[0];
To_Condition system=unstable To enumvar1;
```

will cause GOTCHA to generate up to 2*3=6 test cases:

- one with a transition from a state with system=stable and BoolArray[0]=TRUE to a state with system=unstable and enumvar1=v1.
- one with a transition from a state with system=stable and BoolArray[0]=TRUE to a state with system=unstable and enumvar1=v2.
- one with a transition from a state with system=stable and BoolArray[0]=TRUE to a state with system=unstable and enumvar1=v3.
- one with a transition from a state with system=stable and BoolArray[0]= FALSE to a state with system=unstable and enumvar1=v1.
- one with a transition from a state with system=stable and BoolArray[0]= FALSE to a state with system=unstable and enumvar1=v2.
- one with a transition from a state with system=stable and BoolArray[0]= FALSE to a state with system=unstable and enumvar1=v3.

There may be fewer than six test cases generated if there are no reachable and coverable transitions where one of the above situations occurs. Fewer than six test cases may also be generated if a single test case can be constructed with more than one of the required transitions occurring in the same test case.

4.2.2.6 Explicit Transition Projection

Explicit Transition Projection describes a set of coverage tasks in a similar way to Transition Projection. The conditions FROM_CONDITION and TO_CONDITION and the FROM and TO projection expressions are defined exactly as in the Transition_Projection coverage criteria. Additionally, the Projected_Explicit_Transition defines a Boolean condition that must be satisfied and a projection expression, that must be calculated. These correspond to the rule that causes the transfer between the states, *s* and *t,* of the transition. As in the Some_Explicit_Transition criteria, the Projected_Explicit_Transition corresponds to a set {*s,t,r*} – where *s,t* are states of a transition and *r* – the rule that causes this transition.

The VIA_CONDITION Boolean expression is defined exactly as in the Some_Explicit_Transition and can reference the RuleId value, rule names and rule parameters.

The VIA projection expression adds another set of variables (in addition to the ones for the FROM and TO sets) that are projected on from the state space. The only difference is that the expressions that define the values of these variables are governed by the rules similar to the ones for the Boolean expression of the VIA_CONDITION.

Overall, this coverage criteria may be seen as the projection of all the possible combinations {state, rule, state} onto the values of the expressions defined for each of these three components. The VIA expression defines a set of expressions to be projected on from variable values, the rule name and parameters.

Here is the summary of the rules for the VIA projection expression.

- The VIA expression may contain the RuleID keyword. This may be seen as the name of the rule that causes the transition in question. This expression may also contain the names of rules in the GDL model.

  Example 1:

  `VIA (ruleID > ("abc"));`

  All the {s, r, t} combinations will be projected on a boolean variable for ths expression. In case r is defined in the model after "abc" – this variables value is TRUE. Otherwise – FALSE.

  Example 2:

  `VIA ruleID;`

  All the {s, r, t} combinations will be projected on a variable which can have as many values as the number of different rules in the model. The rule parameters do not have any influence on the value of the projection expression. {s1, r1, t1} and {s2, r2, t2} will represent the same coverage task if and only if r1 is the same rule as r2.

- The VIA expression may contain global variables. The values of these variables are calculated as they are in the state, from which the transition is made (s).

- The VIA expression may contain the names of the rules input parameters. It is an error to use a parameter that is not defined for all the rules that satisfy the VIA_CONDITION expression.

  Example: (part of GOTCHA model)

```
Ruleset i : 1 .. 3
Do
    Rule "abc"
         :
 End;
 Rule "cde"
        :
```

  Example 1: `VIA_CONDITION TRUE VIA i;   --incorrect`

  This is an incorrect use of a rule parameter within a coverage criterion, since parameter i is not defined for the model rule "cde" (which satisfies the VIA_CONDITION expression). An attempt to evaluate the projection expression for rule "cde"will fail.

  Example 2: `VIA_CONDITION ruleID = ("abc") VIA i;   --correct`

  In this case the projection expression i is evaluated only if rule *r* is "abc". Parameter i is defined for rule "abc". There will be three representatives for the coverage criteria in this case – one for every possible value of parameter i for rule "abc".

Note that rule parameters with the same name must be of the same type. In the case of integer subranges, these types may have different names, but must have the same range.

Consider a model, with a Boolean variable var1 and two rules “abc” and “cde”. Now consider the following coverage criterion:

```
CC_Projected_Explicit_Transition FROM_CONDITION TRUE FROM var1;
                                 VIA_CONDITION TRUE VIA ruleID;
                                 TO_CONDITION TRUE TO TRUE;
```

This will cause GOTCHA to generate up to 4 test cases. These test cases will cover the following combinations of variable values / rules:

- one with a transition from a state with var1=false through activation of rule “abc”.
- one with a transition from a state with var1=true through activation of rule “abc”.
- one with a transition from a state with var1=false through activation of rule “cde”.
- one with a transition from a state with var1=true through activation of rule “cde”.

There may be fewer than four test cases generated if there are no reachable and coverable transitions where one of the above situations occurs. Fewer than four test cases may also be generated if a single test case can be constructed with more than one of the required transition/rule combinations occurring in the same test case.

4.2.2.7 All State

All State coverage describes a subset of all projected states that are projected onto the coverage variables.

NOTE: We no longer recommend the use of this coverage criterion because State Projection enables you to define an equivalent coverage criterion without using coverage variables in the declarations section of the model.

Specifying `CC_All_State Boolexpr`; is equivalent to the projected state coverage achieved by the criterion `CC_State_Projection Boolexpr On var1; var2; var3`, where `var1, var2`, and `var3` are defined as `Coverage_var` in the declarations section of the model.

We no longer recommend the use of this criterion and its companion `CC_All_Transition`, since they require the coverage notions to be a part of the behavioural model, rather than a separate entity in a possibly different file.

4.2.2.8 All Transition

All Transition coverage describes a subset of all projected transitions projected onto the coverage variables. The subset is specified by two Boolean expressions, which are evaluated at the start and end (respectively) of each transition to determine whether or not it is a representative of a coverage task.

NOTE: We no longer recommend the use of this coverage criterion becauseTransition Projection allows you to define an equivalent coverage criterion without using coverage variables in the declarations section of the model.

If var1, var2, and var3 are defined as coverage_var in the declarations section of the model, then `CC_All_Transition Boolexpr1; Boolexpr2;` - is equivalent to the transition projection criterion:

```
CC_Transition_Projection
From_Condition Boolexpr1 From var1; var2; var3;
To_Condition Boolexpr2 To var1; var2; var3;
```

5 Generating Test Suites – With Explicit Directives

5.1 Automatically Creating an Abstract Test Suite Using an Explicit Coverage Model

You can create an abstract test suite using an explicit coverage model with full traversal of the state space using the GOTCHA command as follows:

```
GOTCHA beh_model.g test_dir.g –o model –fulltg <GOTCHA_options>
```

This assumes that the installation directory has been set as an environment variable named `Gotcha_TCBeans`, and that the behavioural model and test directives are in the files `beh_model.g` and `test_dir.g` respectively. The abstract test suite is written to a file entitled `model.ats`. The progress messages are written to `stdout`, which defaults to the screen.

You can use runtime options to customize the resulting test suite. These runtime options serve several purposes, including:

- Customizing the output.
- Varying the algorithms used to generate the test suite.
- Giving soft constraints on the test cases.
- Allocating memory.

- Initialising the random number generator.

In order to understand some of the runtime options fully, refer Section 5.3 on the The Full Traversal Test Generation Algorithm.

5.2 Complete GOTCHA Runtime Options

5.2.1 Help (-h)

To display the basic GOTCHA runtime options, use the command line:

GOTCHA *modelname.g* **-h**

The results are displayed below:

```
Basic GOTCHA options <default -rctg -st10 -gtlub10>:
       -o <filename>      Output the test suite to <filename>.ats,
                          progress messages to standard output
       -rctg              Input coverage test generation
       -st<n>             Stop after n test cases generated
       -gtlub<n>          Test case length recommendation
       -morehelp          Full set of Gotcha options
       -rs<v>             Seed for randomization. Default TOD
       -rtg               Random test generator
       -itg               Run Interactive Test Generator
       -l                 Print license
       -morehelp          Print full details of all GOTCHA runtime options
       An argument without a – is taken to be an input filename and must end in .g
```

5.2.2 More Help (-morehelp)

To display all GOTCHA runtime options, use the command line:

GOTCHA *modelname.g* **-morehelp**

The results are displayed below:

```
1) Basic <default -rctg -st10 -gtlub10>:
       -o <filename>      Output the test suite to <filename>.ats,
                          progress messages to standard output
       -rctg              Input coverage test generation
       -st<n>             Stop after n test cases generated
       -gtlub<n>          Test case length recommendation
       -morehelp          Full set of Gotcha options
```

-rs<v>	Seed for randomization. Default TOD
-rtg	Random test generator
-itg	Run Interactive Test Generator
-l	Print license
-morehelp	Print full details of all GOTCHA runtime options

An argument without a – is taken to be an input filename and must end in .g

2) Test Generation Strategy: (default: -rctg -st10 -gtlub10 -gtcf1 -m8)

-rctg	Input coverage test generation.
-st<n>	Stop after n tests generated.
-gtlub<n>	Test case length recommendation.
-gtcf<n>	Coverage factor (all coverage tasks).
-itg	Interactive test generation.
-rtg	Random test generation.
-rtctg	Input transition coverage test generation.
-rpctg	Input transition pair coverage test generation.
-dbg	Run model debugger.
-fulltg	Full traversal with coverage criteria test generation
-m<n>	Memory allocation in Mb. Default: 8Mb
-loop<n>	Maximum number of cycles in any while loop limited to 10^n, n in 1..8.

3) Full Traversal Parameters: (default: -gtb -gftb)

-gtd, -gtb, -gtc Initial state traversal by Depth First Search, Breadth First Search, Task Coverage Search.

-gftd, -gftb, -gftc Further traversal by Depth First Search, Breadth First Search, Task Coverage Search.

-gtllb<n> Test case length lower bound recommendation.

-ues Test extension by unbounded search on the state space.

-efct Test extension attempt should be performed from the covered task, not the final state of the extended test.

-rr Randomly reorder lists of next states, reachable from the current one

-gotfp<n> Activate On The Fly test generation every 10^n states explored.

4) GOTCHA Output Options: (default: -gus0, -fst1, -gtp2, -gdp2, -p3)

-ps	Print out full states in test cases, as opposed to state changes.

-gts	`Don't print out the generated test cases.`
-p<n>	`Report progress every 10^n space exploration events, n in 1..5.`
-pn	`Print no space exploration progress reports.`
-gtp<n>	`Report progress every 10^n test cases generated, n in 1..5.`
-gtpn	`Print no test case generation progress reports.`
-gdp<n>	`Report progress every 10^n tasks covered, n in 1..5.`
-gpnc	`Print the number of new covered tasks before each test case.`
-gde	`Print details of uncoverable tasks at the end.`
-gtur	`Print details on uncoverable task representatives.`
-gus<n>	`Report the first n uncoverable states.`
-fst<n>	`Report the first n forbidden states/transitions found.`
-fp	`Print a path from an initial state to a forbidden state/transition.`
-ip	`Print a path from an initial state to an invariant.`

5.2.3 Input Coverage Test Generation (-rctg)

If you invoke GOTCHA with the command line:

`GOTCHA` *modelname.g* **`-rctg`**

this starts the input coverage test generator on the behavioural model contained in *modelname.g*. This generates a test case covering each rule with all possible inputs. For details about the interactive test generation process see Section 3.3.2 Input coverage test generation.

5.2.4 Random Test Generation (-rtg)

If you invoke GOTCHA with the command line:

`GOTCHA` *modelname.g* **`-rtg`**

this starts the random test generator on the behavioural model contained in *modelname.g*. For details about the random test generation process see Section 3.3.1 Random test generation.

5.2.5 Input Transition Coverage Test Generation (-rtctg)

If you invoke GOTCHA with the command line:

```
GOTCHA modelname.g -rtctg
```

this starts the input transition coverage test generator on the behavioural model contained in *modelname.g.* This generates a test case covering each rule with all possible inputs in each possible transition position of a test case. For details about the input transition coverage test generation process see Section 3.3.3 Input transition coverage test generation.

5.2.6 Input Transition Pair Coverage Test Generation (-rpctg)

If you invoke GOTCHA with the command line:

```
GOTCHA modelname.g -rpctg
```

this starts the input transition pair coverage test generator on the behavioural model contained in *modelname.g.* This generates a test case covering each pair of rules with all possible inputs in each possible pair of transition positions of a test case. For details about the input transition pair coverage test generation process see Section 3.3.4 Input transition pair coverage test generation.

5.2.7 Full Traversal Test Generation

If you invoke GOTCHA with the command line:

```
GOTCHA modelname.g -fulltg
```

this starts the full traversal test generator on the behavioural model contained in *modelname.g.* This generates test cases which satisfy all the test constraints and cover every task specified in the coverage criteria specified in *modelname.g*.

The process involves first traversing the entire state space of the model and then generating test cases which satisfy all the constraints in order to cover all tasks specified in the coverage criteria. If no test constraints are in the model, the only constraint added is that every state may satisfy the EndTestCase constraint. If no coverage criteria are specified, the default coverage criterion is to cover some example of an arbitrary state.

For details about the full traversal test generation process see Section 5.3.

5.2.8 Full Traversal Test Generation On The Fly (-fulltg –gotfp<n>)

When invoking the full traversal test generator, GOTCHA completes the initial state space exploration before beginning to generate test cases. If the state space is very large this can take a long time and may even fail to complete due to memory limitations. When this occurs it is advisable to use on-the-fly test case generation, which generates test cases as soon as possible after a certain number of states have been added to ReachableStates.

The drawbacks of on-the-fly test case generation are:

- Length extension algorithms cannot be invoked before the entire state space has been explored.
- Randomization of test cases is less accurate.

On-the-fly test case generation can be invoked after every 10, 100, 1000, 10,000, or 100,000 states are added to ReachableStates with the flag `-gotfp1, -gotfp2,…, -gotfp5`, respectively.

5.2.9 Number of test cases (-st<n>)

The **–st** flag puts an upper bound on the number of test cases to be generated by any of the automated test generation algorithms. Do not leave a space between the flag and its parameter.

Invoking GOTCHA with the flag **–st15** will ensure that no more than 15 test cases are generated by the test generator. If the coverage goals are reached before the number of test cases is exhausted, the test generator will stop before generating 15 test cases. The default number of test cases depends on the test generator being used, see the following table of default values:

Test Generator	**Default maximum number of test cases**
Random	10
Input Coverage	10
Input Transition Coverage	10
Input Transition Pair Coverage	100
Full Traversal Coverage	1000
Full Traversal Coverage On The Fly	100

5.2.10 Test case length recommendation (-gtlub<n>)

The **–gtlub** flag is the recommended test length to be used by the test generator. Different test generators use different defaults for the test length recommendation. The test length includes the start rule as one of its transitions. The default used by each test generator is given in the following table:

Test Generator	**Default test length recommendation**
Random	10
Input Coverage	10
Input Transition Coverage	10
Input Transition Pair Coverage	5
Full Traversal Coverage	50

5.2.11 Coverage Factor (-gtcf<n>)

The coverage factor controls the number of representatives generated for each coverage task in the whole test suite. This parameter is used by all test generators except random. The default is 1. Specifying **`-gtcf3`** causes GOTCHA to generate the test suite so that it will have at least 3 occurrences of each coverage task in the abstract test suite. The test cases pass through different representatives chosen at random from among the representatives of a given coverage task. GOTCHA may repeat a test case in order to reach the coverage factor.

5.2.12 Interactive Test Generation (-itg)

If you invoke GOTCHA with the command line:

```
GOTCHA modelname.g -itg
```

this starts an interactive test generation session on the behavioural model contained in *modelname.g*. For details about the interactive test generation process see Section 3.1 Interactive Test Generation.

5.2.13 Code Debugger (-dbg)

If you invoke GOTCHA with the command line:

```
GOTCHA modelname.g -dbg
```

This starts a code debug session on the behavioural model contained in *modelname.g*. The model can be debugged in conjunction with the GOTCHA test generation algorithm (the default) or in conjunction with the interactive test generator. For details about the code debugging process see Debugging a Model.

NOTE: The use of this option degrades GOTCHA's performance considerably, so use it only when you are actually debugging.

5.2.14 Input and Output (-o <filename>, >)

The standard command line invocation for GOTCHA is:

```
GOTCHA infile1.g infile2.g infile3.g -o outfile
```

This assumes that the behavioural model and testing directives are spread across three files, and that the abstract test suite is written to *outfile.ats.* GOTCHA accepts up to 10 input files.

The output of GOTCHA may be directed to two separate files by using the **–o** flag and the redirection symbol >. The abstract test suite is directed to the file following the **–o** flag, and the progress indicators and other messages (including compilation and error messages) are directed to the file following the redirection symbol >. If no file names are given on the command line, the output is directed to the screen (*stdout*).

```
GOTCHA helloworld.g -o helloworld > helloworld.msgs
```

The command above directs the abstract test suite to *helloworld.ats*, and the messages to the file *helloworld.msgs*.

5.2.15 Compact State Representation (-c)

The **–c** option produces a test generator that encodes the states of the model very compactly. It is rumored that this option saves space and impacts negatively on runtime of the test generator. It definitely saves space – and we have yet to see a case where it had a negative effect on the runtime. We recommend using this option with the full traversal test generator.

5.2.16 Disable Runtime Checking (-d)

The **–d** option produces a test generator that does not conduct runtime checking of the ranges of variables. This is reputed to make the generator faster, but it may also cause ungraceful crashes if your model causes a variable to go out of range. Use this option only after the model has been completely debugged.

5.2.17 Memory Allocation (-m<n>)

This is a critical runtime parameter for the full traversal test generator. It determines the memory size (in MB) allocated for the **ReachableStates** data structure. The default is 8 MB.

5.2.18 Random Seed (-rs<n>)

This is a critical runtime parameter for all GOTCHA test generators. It determines the seed used by the random number generator. The random number generator is called at several points in all the test generation algorithms. If no random seed is specified, the test generator takes a random number from the time-of-day clock on the computer. In order to repeat a test generation run in all its details, you must use the same random seed.

5.2.19 Model Traversal Library (-lib)

The -lib parameter causes the creation of the model traversal library, which can be used to implement a user-programmed test generator. Invoking GOTCHA with –lib causes GOTCHA to produce a library, not a test suite. When omitted, GOTCHA produces an abstract test suite, using one of the test generators.

5.2.20 Initial State Space Exploration Strategy (-gtb, -gtd, -gtc)

These flags control the strategy for full traversal initial state space exploration.

The default is breadth first search, which produces short tests by taking a shortest path tree as the exploration tree.

Depth first search produces very long tests, and is recommended only if the test suite contains a single, long test. Using depth first search for more than one test usually generates tests with high redundancy. If you use the coverage task strategy, the test suite should be shorter, but this effect is not guaranteed in all cases.

Coverage directed space exploration directs the test generator to explore states that contain new coverage tasks before states that represent tasks that have already been seen.

It sometimes produces tests with many different tasks in each test, and thus reduces the overall length of the test suite.

In terms of the algorithm, the initial state exploration strategy controls which algorithm is used to ChooseAndRemove a state from the frontier.

5.2.21 Further Exploration Strategy (-gftb, -gftd, -gftc)

These flags control the full traversal strategy that determines whether a state is or is not covered during the `AfterExplorationGeneration` algorithm.

Breadth first search yields shorter tests than depth first search.

Coverage first search may not yield shorter individual tests, but tends to yield shorter test suites overall.

The default strategy is a breadth first search.

5.2.22 Test Case Length Recommendation (-gtlub<n>, -gtllb<n>), Test Case Extension Strategy (-ues -eftc)

The length recommendations are soft constraints on the abstract test suite. If possible, the test cases will have more transitions than the lower bound and fewer transitions than the upper bound. However GOTCHA may be unsuccessful in its attempts to increase or decrease the length of a test. The default test extension algorithm in full traversal is breadth first.

Do not leave a space between the flag and its parameter. The usage is:

```
GOTCHA helloworld.g –gtllb25 –ues –o helloworld.ats
```

The test case extension strategy is invoked by the full traversal generator during the output of a test. The strategy that has the lowest computation time is the default (without either **ues** or **eftc**). Unbounded extension search and coverage task extension have longer computation times, but may result in more successful extensions to the test case. The two switches are independent of each other, so there are four possible test extension strategies.

The test case reduction strategy is unique, and always gives the shortest possible test passing through a given task representative.

5.2.23 Random reorder (-rr)

The rules enabled for execution at each state are ordered in a fixed order based on their order in the input file. This can lead to test suites that appear to be very predictable in some models. The **–rr** flag eliminates the fixed ordering of rule invocation and thus randomizes the test suite better. This flag may have a negative impact on the performance of the test generator.

5.2.24 Loop Exit (-loop<n>)

All “while” loops in a GOTCHA model are limited in the number of times they are allowed to execute. The default is 10,000 iterations. This parameter may be set to any power of 10 from 10 to 10^8.

5.2.25 Output Format (-ps, -gts)

The option **–ps** alters the format of the abstract test suite by printing the values of all state variables after every transition, and not just the values that have changed. The default is to print only changed values of state variables. This option may be desirable both when debugging the model, and when using A2C translation.

The option **–gts** suppresses all test case output, and prints only the progress reports and messages.

5.2.26 Progress Reports (-p<n>, -pn, -gtp<n>, -gtpn, -gdp<n>)

The options **–p**<n> and **–pn** control the printing of messages concerning the number of states currently in **ReachableStates** and **FrontierStates**. Reports are printed every 1000 new states in **ReachableStates** by default, but you can changes this to 0, 10, 100, 1000, 10,000, or 100,000.

The options **–gtp**<n> and **–gtpn** control the printing of messages concerning the number of test cases generated so far. Reports are printed every 100 test cases by default, but you can changes this frequency to 0, 10, 100, 1000, 10,000, or 100,000.

The option **–gdp**<n> controls the printing of messages concerning the number of coverage tasks covered so far. Reports are printed every 100 tasks by default, but you can changes this frequency to 10, 100, 1000, 10,000, or 100,000.

5.2.27 Coverage Task Details (-gpnc)

The flag **–gpnc** causes a comment to be inserted in the abstract test suite at the beginning of each test indicating how many new coverage tasks are contained in this test. A coverage task is “new” if no representative of that task has appeared in a previous test.

5.2.28 Uncoverable States, Transitions, Tasks Details (-gde, -gtur, -gus<n>)

These flags control output to the console – not to the abstract test suite. They describe uncoverable tasks, uncoverable representatives, and uncoverable states respectively.

The flag **–gus**<n> produces a printout the first n times that an uncoverable state is encountered during either exploration phase. The same state may be output several times. The default produces no printout.

The flag **–gtur** produces a printout when a task representative (state or transition) is proved uncoverable.

The flag **–gde** produces output when a whole task is proved uncoverable.

5.2.29 Forbidden State, Forbidden Transition, Invariant Details (-fst<n>, -fp, -ip)

The flag **–fst**<n> produces a printout to the console the first n times that each test constraint violation is encountered. The same state or transition may be output several times. The default is **–fst1**, which produces a printout when the first instance of each constraint violation is encountered.

The flag **–fp** extends the printout produced by **–fst** by giving an execution path from a start state to the forbidden state just encountered. This is useful for debugging the model.

The flag **–ip** causes an execution path to be printed whenever a state that violates an invariant is encountered. After encountering a state that violates an invariant GOTCHA stops.

5.3 The Full Traversal Test Generation Algorithm

The test generation algorithm uses three principal data structures:

- A hash table, called **ReachableStates**, which contains a list of all the states reached, and an indication of whether that state is coverable, uncoverable, or undecided. Each state also keeps a record of its immediate ancestor in the exploration tree.
- A stack or queue of states, called **FrontierStates**, which contains all those states that have been reached, but whose neighbors have not yet been examined.
- A set of coverage tasks, called **TaskTree**, which contains a list of reachable coverage tasks, together with accounting details of the task's representative states. A task is marked as covered when a test through one of its representatives is output. A task is marked as uncoverable after all its representatives have been marked as uncoverable.

The algorithm implemented in the test generator is approximated by the following pseudo-code (many details have been omitted or abstracted out):

```
main();
            Set ReachableStates, FrontierStates, TaskTree all
            empty (ϕ);
            for all s ∊ StartStates do StateCheck(s);
            while FrontierStates ≠ ϕ do
                  s = ChooseAndRemoveStateFrom(FrontierStates)
                  for all r ∊ Rules do
                        If r is enabled in s then
                        Begin
                              d = Apply r to s
                              If TransitionCheck(s,d) then
            StateCheck(d)
                        End
                  Endfor
                  Every once in a while do OnTheFlyGeneration();
            Endwhile;
            AfterExplorationGeneration();
End Main
```

```
Procedure StateCheck(s)
    If s is Forbidden then return;
    If s is not in ReachableStates then insert it in both ReachableStates and FrontierStates
    If s belongs to any coverage task, update the TaskTree
    If s is a final state mark it and its ancestors in the exploration tree as coverable
End Procedure StateCheck


Function TransitionCheck(s,d)
    If (s,d) is a forbidden transition return FALSE
    If (s,d) belongs to a coverage task, update TaskTree
    return TRUE
End Function TransitionCheck


Procedure OnTheFlyGeneration()
    For all s ∈ ReachableStates
        If s is coverable and belongs to an uncovered task in TaskTree, then Output a test through s and update TaskTree.
    For all s ∈ ReachableStates and s not in FrontierStates
        If (s,d) is not a forbidden transition and belongs to an uncovered task in TaskTree, then Output a test through (s,d) and update TaskTree.
    Return to Main
End Procedure OnTheFlyGeneration


Procedure AfterExplorationGeneration()
    For all t ∈ TaskTree
        If t is neither covered nor uncoverable
        Then for all r representatives of t
            If r is undecided then decide if r is coverable or not by a further search starting from r
            If r is coverable Output a test, update t, break
            Else just update t.
End Procedure AfterExplorationGeneration
```

6 The Programming Interface to the Test Generator

GOTCHA provides an API that can be used to implement a user-programmed test generator, using the state space description extracted from the GOTCHA model. The API functionality is similar to the one provided by the Interactive Test Generator, but geared towards convenience of use by a programmer.

It is possible to use the API in conjunction with a test driver that interacts with the unit under test. This possibility allows the creation of test cases that respond to the non-deterministic actions of the application. It also allows an on-the-fly coverage improvement algorithm by allowing test generation and test execution to take place concurrently.

In order to access the API a user must create the model traversal library, using GOTCHA –lib option.

6.1 Creating the Model Traversal Library

The Model Traversal Library is a static C++ library, which has a permanent interface, but a model – specific implementation. In particular, its functionality depends not only on the model, but also on the model's test constraints.

Entering the command:

```
        GOTCHA beh_model.g test_dir.g –lib –o beh_model_api
```

will create the Model Traversal Library named *beh_model_api.dll*. Note, that using –lib will prevent GOTCHA from running the traversal engine on the model and thus from creating any tests. This also means that test generation directives and test printing instructions to GOTCHA become irrelevant and are ignored.

The include files are under the GOTCHA_TCBeans root (wherever you have installed GOTCHA-TCBeans), in the include/ directory. In order to use the library, you need to include the itg_api.h header file. The namespace of the library is GOTCHA_HRL. The functionality of the library can be accessed through public methods of GOTCHA_HRL::ITG_API class.

6.2 Test Generator API

This section covers the Model Traversal Library API. The model library contains the definition of the ITG_API class. Objects of this class can be used to:

- Access the state space of the model.
- Activate model rules.
- Inspect the values of the state variables.
- Create test suites.

It is important to understand the basic behavior of the ITG_API class object to use the API. The object can be in one of the four states:

- START
- RULE_WAIT
- STATE_WAIT
- FINAL.

These states mirror the state of the current test case that is being created by user actions. In other words: the application of the main methods of the object, in addition to traversing the model's state space, also update the current test case. This test case depicts the path taken by the user so far.

The ITG_API object is created in START state. In this state, the values of the system variables are not yet defined. The values are defined by choosing a StartTestCase rule. The user may change the state of the object by alternatively choosing the rule to be applied in the current state, and then choosing a state from the list of alternatives states, reachable by the last applied rule. The choice is made using the <rule_id> and <state_id> parameters in the method calls. These ids are nothing more than the sequential number of rule/state in the current list of choices. To make a reasonable choice, the user can use information calls, to get the values of state variables in different states or the names of the rules to be applied.

The include files are under the GOTCHA_TCBeans root (wherever you have installed GOTCHA-TCBeans), in the include/ directory. In order to use the library, you need to include the itg_api.h header file. The namespace of the library is GOTCHA_HRL. The functionality of the library can be accessed through public methods of GOTCHA_HRL::ITG_API class.

The return values of all the methods follow the principles:

1. In case of success return true (if boolean) or actual value (int or std::string pointer)
2. In case of failure return false (if boolean), 0(if int) or pointer to empty string (if std::string)
3. std::string is defined in the STL (Standard Template Library).
4. All methods that return pointers to std::string, allocate memory for these strings. The user is responsible for cleaning up the space allocated for such strings.

The following table lists the public methods of the ITG_API class objects.

Method	Description

ITG_API();	Constructor. Initialization.
~ITG_API();	Destructor. Clean up.
/* test suite controls */	
bool StartTestSuite(void);	Starts a new test suite. Discards the old one. Doesn't alter the current test case.
bool DiscardTestCase(void);	Discards the current test case (together with the undo information).
bool SaveTestCase(void);	Adds the current test case to the test suite. Does not discard the current test case. The current test case can be updated (including undo option) from the same point.
bool NewTestCase(void);	Discards the current test case.
bool PrintTestSuite(char* filename);	Prints the test suite in the files filename.ats and filename_0.ats (master and slave)
bool PrintTestCase(char* filename);	Creates a temporary test suite, consisting of a single test case (the current test case) and prints it out in two files (master and slave), like PrintTestSuite.
/*info calls */	
std::string *GetVal(const std::string& var_name, int state_id);	Allocates a new std::string (as defined in STL) and assigns to it the value of state variable <var_name> in state <state_id>. This method can be called in STATE_WAIT state. In case of problems (e.g. called in RULE_WAIT, no variable with <var_name> in the model, or erroneous state_id, etc.) the method returns a pointer to an empty string.
std::string *GetVal(const std::string& var_name);	The same as the previous method, but used only in RULE_WAIT, START or FINAL state.
int CountEnabledRules(void);	Returns the number of enabled rules in the current state. Can only be used in RULE_WAIT, START or FINAL state. In STATE_WAIT returns 0.
int CountAlternativeStates(void);	Returns the number of alternative states that can be reached by activating the last chosen rule. Can only be used in STATE_WAIT state. In other states returns 0.
bool IsFinalState(int state_id);	Returns true if the state <state_id>, reached by the last rule activation, satisfies some TC_EndTestCase condition. Can only be used in STATE_WAIT state. In other states returns false.
bool IsFinalState(void);	The same as the previous method, can only be used in non STATE_WAIT state. Otherwise, returns false.
bool IsLegalState(int state_id);	Returns true if the state <state_id>, reached by the last rule activation, does not violate some test constraint. Can only be used in STATE_WAIT state. In

	RULE_WAIT, START or FINAL state, returns false.
std::string *RuleName(int rule_id);	Allocates std::string and returns the name of the rule with <rule_id> in the current state. Can only be used in non STATE_WAIT state. In STATE_WAIT state returns a pointer to an empty string.
int NumStartRules()	Returns the number of start rules in the model
int NumRules()	Returns the number of transition rules in the model
int RuleNumber(int rule_id)	Can only be used in RULE_WAIT. Behave differently at the start of a test case. Returns the absolute rule number in the range 0..NumStartRules-1 if the itg is waiting for a startrule. Returns the absolute rule number in the range 0..NumRules-1 if the itg is waiting for a transition rule.
/*main controls */	
bool ApplyRule(int rule_id);	Makes rule with <rule_id> the next rule of the current test case. Returns true in case of success. Used only in RULE_WAIT, START states. Changes the state of the system to STATE_WAIT. In other states or in case of erroneous rule_id returns false and does nothing.
bool Finalize(void);	Adds the final clause to the current test case, returns true, switches the system state to FINAL. Can only be applied in RULE_WAIT state. In case of failure (current state doesn't satisfy any TC_EndTestCase condition or wrong state of system) returns false and does nothing.
bool SetCurrentState(int state_id);	Makes state with <state_id> the next state of the current test case. Is used only in STATE_WAIT state. In case of success returns true and switches the state to RULE_WAIT. In case of failure does nothing and returns false. Please, note, that it is impossible to set a state in case it violates some test constraint.
bool Undo(void);	Undo the last test case operation (one of the following 3 methods can be undone: ApplyRule, Finalize, SetCurrentState. The operation restores the state of the system before the method that is undone and restores the current test case to the one before the operation. Undo() can be applied a number of times (returning true), until the state of the system becomes START (start of test suite reached). An application of Undo() in START does nothing and returns false. Other operations can not be undone.

7 Debugging a Model

The Code Debugger has two main purposes:

- Debugging the code of a rule, procedure or function.
- Using breakpoints to reach a specific rule. This can be difficult with the Interactive Test Generator, since reaching a state where the particular rule is enabled may be complicated.

To activate the Code Debugger run GOTCHA with the -dbg option.

```
GOTCHA modelname.g –dbg
```

or

```
GOTCHA modelname.g –dbg -itg
```

The first command runs the standard GOTCHA traversal and test generation algorithm and stops at the first invocation of a rule or at the first `break;` statement encountered in this algorithm.

The second command enters the interactive test generation framework. When a rule, procedure, or function with a `break` statement is invoked, the Code Debugger takes control beneath the interactive test generator control mechanism.

You can insert breakpoints in the model by using the `break` statement. You can also insert or remove them during debugging. The debugger enables you to inspect the values of global (state) variables at run time, while stepping through the code of a rule, function, or procedure.

We do not recommend using breakpoints in the code of functions that are called by the guard of a rule or in an expression used in a test constraint or coverage criterion. This is because the GOTCHA or Interactive Test Generators call these functions frequently in the course of their computations.

Running in debug mode degrades performance. We recommend its use only for debugging.

7.1 Debugger Commands

The following features are implemented in the Code Debugger:

1. **Code navigation:** Enables you to move through simple GOTCHA model statements (i.e., one-line GOTCHA statements, such as assignment, clear, procedure call, and so forth). Use the following debugger instructions to navigate:

- **next (n):** Goes to the next simple statement in the model. When there are no more simple statements in the current rule, the debugger goes to the first simple statement of the next rule fired, or to the first call to a function or procedure that is invoked before the next rule is fired. If the current statement is a procedure call or contains a function call, the procedure or function will not be entered.
- **step into (si)**: Same as **next**, but procedures or functions called in the current statement are entered.
- **step out (so)**: Same as **next**, but the next stop is outside the scope of the current function, procedure, or rule. Execution continues until the current scope is exited.
- **continue (c)**: Runs until the next breakpoint.

2. **Dynamic breakpoints**: You may insert, remove, enable, and disable breakpoints at any simple statement during debugging. The specific breakpoint handling commands are:
   - **insert breakpoint (ib):** Inserts a breakpoint at a simple statement.
   - **remove breakpoint (rb):** Removes a breakpoint at a simple statement.
   - **disable all breakpoints (rab)**: Disables all breakpoints currently enabled (including static breakpoints), but does not delete them.
   - **enable all breakpoints (eab)**: Reverses the disabling of breakpoints.
3. **Code Context Display:** When the debugger stops at a statement, the statement is displayed in the context of the two preceding and following lines of code. The current statement is marked by the ">>" sign. If the current statement is an enabled breakpoint (static or dynamic), it is marked "*>>".If the current statement is a disabled breakpoint, it is marked "@>>". The command to show the current code position again is:
   - **print code (pc)**
4. **Variable Display**: You can display the values of all state variables. The command for state variable display is:
   - **print (p)**
5. **General Control**: The basic help and quit operations:
   - **quit (q)** When running the Code Debugger under the Interactive Test Generator mode, you can exit GOTCHA, using the **q** option from the Code Debugger. In this case, no "save current test case?" questions will be asked.
   - **help (h)**: Lists the available commands.

7.2 Sample Debug Session

This section describes a session using the Code Debugger in conjunction with the interactive test generator on the sample *SimpleTicketMachine.g* in the samples/ticketmachine/models folder.

Issue the following command to start the session:

gotcha *samples/ticketmachine/models/SimpleTicketMachine.g* –**itg -dbg**

The following text is displayed:

```
Input Model: simpleticketmachine.g


Algorithm:
~~~~~~~~~~
         GOTCHA Test Generation in interactive mode
         Random Seed is 847463941.
         Setting up interactive GOTCHA test generation
         Computing test start states and rules
         Firing unique start rule
Initialize the machine


    Code line 33 file simpleticketmachine.g:

     TC_StartTestCase "Initialize the machine"
     begin
 >>     coinBuffer := 0;
        coinStore := 0;
        coinTray := 0;


   Please, enter a command from the list
   p, pc, c, n, si, so, rab, eab, q, ib, rb, h – help
```

Note that the code debugger takes control before the interactive test generator, since the interactive test generator invokes the unique start test case rule before giving you control. Input the command `h` to see the list of Code Debugger commands:

```
h
        Code Debugger Commands:

   p   -   print out local and global variables
   pc  -   print out code location
   c   -    continue
   n   -    next
   si  -    step into
   so  -    step out
   ib  -    insert breakpoint
   rb  -    remove breakpoint
   rab -    disable all breakpoints
   eab -    enable all breakpoints
   q   -    quit
   h   -    print commands list



   Please, enter a command from the list
   p, pc, c, n, si, so, rab, eab, q, ib, rb, h – help
```

Enter `c` to resume running the interactive test generator:

```
c
         Entering unique start state
         Rules enabled in current state:
Rule 1(E1V0F0):
serviceMachine - service the machine
Rule 2(E1V0F0):
collectTickets - empty the ticket tray
Rule 3(E1V0F0):
collectCoins - empty the coin return tray
Rule 4(E1V0F0):
returnCoins - press the coin return button
Rule 5(E1V0F0):
printTicket - press the Green button
Enter m to see more enabled rules
Enter a Rule Number(1-6), c, b, r, s, p, ps, d, i, h, m, o,
eab, rab, st or q
```

The interactive test generator is now running. Choose the rule returnCoins, where there is a breakpoint:

```
4
         You have chosen Rule Number 4:


    Code line 79 file simpleticketmachine.g:

        if coinTray + coinBuffer <= coinTrayCapacity
        then
*>>             break;
                coinTray := coinTray + coinBuffer;
                coinBuffer := 0;


   Please, enter a command from the list
   p, pc, c, n, si, so, rab, eab, q, ib, rb, h – help
```

You have returned to the Code Debugger menu.

This section demonstrates the `p` command to print all the values of the global variables, and the n and s commands to step through the code:

```
p
    State Variables:

result = Reset
ticketStore = 10
ticketTray = 0
coinTray = 0
coinStore = 0
coinBuffer = 0


   Please, enter a command from the list
   p, pc, c, n, si, so, rab, eab, q, ib, rb, h - help
n


    Code line 80 file simpleticketmachine.g:

        then
                break;
```

```
 >>             coinTray := coinTray + coinBuffer;
                coinBuffer := 0;
                result := coinReturnOK;


   Please, enter a command from the list
   p, pc, c, n, si, so, rab, eab, q, ib, rb, h - help
n


    Code line 81 file simpleticketmachine.g:

                break;
                coinTray := coinTray + coinBuffer;
 >>             coinBuffer := 0;
                result := coinReturnOK;
        else


   Please, enter a command from the list
   p, pc, c, n, si, so, rab, eab, q, ib, rb, h - help
si
```

The difference between n (next) and si (step into) is only observable if the current line contains a call to a function or procedure. If the current line contains a call to a function or procedure, then si moves the user to the first line of the called procedure, whereas n simply moves to the next line of code in the current scope.

```
    Code line 82 file simpleticketmachine.g:

                coinTray := coinTray + coinBuffer;
                coinBuffer := 0;
 >>             result := coinReturnOK;
        else
                coinTray := coinTrayCapacity;


   Please, enter a command from the list
   p, pc, c, n, si, so, rab, eab, q, ib, rb, h – help
```

Now invoke the `so` (step out) command, which moves to the next highest scope. In this case, the upper scope is the interactive test generator. However, had you been inside the code of a procedure called by a rule, you would have returned to the rule at the line of code immediately after the procedure call:

```
so
         Rules enabled in current state:
Rule 1(E1V0F0):
serviceMachine - service the machine
Rule 2(E1V0F0):
collectTickets - empty the ticket tray
Rule 3(E1V0F0):
collectCoins - empty the coin return tray
Rule 4(E1V0F0):
returnCoins - press the coin return button
Rule 5(E1V0F0):
printTicket - press the Green button
Enter m to see more enabled rules
```

```
         Current state is an EndTestCase state.
Enter a Rule Number(1-6), c, b, r, s, p, ps, d, i, h, m, o,
eab, rab, st or q
```

Choose Rule 1 – serviceMachine, where there is another breakpoint in the source code of the model:

```
1
         You have chosen Rule Number 1:


    Code line 110 file simpleticketmachine.g:

     ==>
     begin
*>>     break;
        coinBuffer := 0;
        coinStore := 0;


   Please, enter a command from the list
   p, pc, c, n, si, so, rab, eab, q, ib, rb, h – help
```

The next section of the sample session demonstrates the use of the commands to insert, enable, and disable breakpoints:

```
rb
    Breakpoint removed



    Code line 110 file simpleticketmachine.g:

     ==>
     begin
 >>     break;
        coinBuffer := 0;
        coinStore := 0;


   Please, enter a command from the list
   p, pc, c, n, si, so, rab, eab, q, ib, rb, h - help
si


    Code line 111 file simpleticketmachine.g:

     begin
        break;
 >>     coinBuffer := 0;
        coinStore := 0;
        coinTray := 0;


   Please, enter a command from the list
   p, pc, c, n, si, so, rab, eab, q, ib, rb, h - help
ib
    Breakpoint created
    Code line 111 file simpleticketmachine.g:

     begin
        break;
*>>     coinBuffer := 0;
        coinStore := 0;
        coinTray := 0;


   Please, enter a command from the list
   p, pc, c, n, si, so, rab, eab, q, ib, rb, h - help
rab
  All breakpoints disabled



    Code line 111 file simpleticketmachine.g:

     begin
        break;
@>>     coinBuffer := 0;
        coinStore := 0;
        coinTray := 0;


   Please, enter a command from the list
   p, pc, c, n, si, so, rab, eab, q, ib, rb, h - help
eab
  All breakpoints enabled



    Code line 111 file simpleticketmachine.g:

     begin
        break;
*>>     coinBuffer := 0;
        coinStore := 0;
        coinTray := 0;


   Please, enter a command from the list
   p, pc, c, n, si, so, rab, eab, q, ib, rb, h – help
```

Now quit the debug session:

```
q
Task completed.
```

8 Browsing Test Suites

The test suite browser, **TSBrowser** allows you to view an ATS (abstract test suite) or SET (suite execution trace) file. The test suite is displayed in a tree structure with test suites hierarchically above test cases and test cases above interactions.

The Test Suite Browser will be documented elsewhere.

9 Glossary of Terms

API

Application Programming Interface.

ATS

Abstract Test Suite: an XML markup language for the test suites generated by the GOTCHA test generator.

Behavioural Model

A behavioural model of a software unit is an abstract description of the software that includes abstract descriptions of its data types (classes), data structures (objects), and transitions (events which cause the data structures to change their values).

Coverable State

A state is coverable if there exists a test case which begins at a start state, passes through the state in question, and continues on to a final state while satisfying all the test constraints.

Coverable Task

A coverage task may be either a set of states or a set of transitions. A state or transition that belongs to a coverage task is said to be a representative of that task. A coverage task is **coverable** if there exists a representative of that task which is coverable. That is, if there is a test satisfying all the test constraints that includes a state or transition, which belongs to the task.

A task is **uncoverable** if **all** its representatives are uncoverable.

Coverable Transition

A transition is coverable if there exists a test case which begins at a start state, passes through the transition in question, and continues on to a final state while satisfying all the test constraints.

Coverage Criterion

A coverage criterion is a description of a set of tasks that the test suite should cover. The coverage criteria expressible in GOTCHA Definition Language are sets of states, sets of transitions, and projected states and transitions.

Coverage Task

A coverage task may be either a set of states or a set of transitions. A state or transition that belongs to a coverage task is said to be a **representative** of that task. An example of a coverage task might be a projected state.

Failed

A test case outcome that indicates that one of the test case's transitions completed with a state not predicted by the transition's associated rule.

GDL

The GOTCHA Definition Language. A language based on the Murphi description language for describing behavioural models of software units.

ITG

The GOTCHA Interactive Test Generator: An interface for generating test cases for a GDL model either manually or through an application programming interface (the model traversal library).

PluRule

A pluRule in a behavioural model is the description of a stimulus to the unit under test that can have more than one possible valid effect on the system. The various possible outcomes are described by sets of GDL statements enclosed in a **Begin End** block. The set of all outcomes is enclosed in a **pluRule EndpluRule** block.

Projected State

The projected state of a state in a behavioural model is defined by the values of all its coverage variables. The coverage variables are a subset of the global or state variables of the model. Two different states with the same projection are said to be **representative**s of the projected state. A state may also be projected onto a sequence of expressions, this projection computes the values of the expressions and ignores all other state variables. This form of projection is used by the coverage criterion CC_State_Projection.

Projected Explicit Transition

A projected explicit transition consists of a pair of projected states and a ruleID condition with the property that there exists a transition from some representative of the first projected state to some representative of the second projected state using a rule which satisfies the ruleID condition.

Projected Transition

A projected transition consists of a pair of projected states with the property that there exists a transition from some representative of the first projected state to some representative of the second projected state.

Reachable State

A state is reachable if there exists a path (or sequence of transitions) in the model from a start state to the state in question.

Rule

A rule is a description of the behaviour of the unit under test under a given stimulus. A rule consists of a Boolean precondition, which determines if the stimulus may be applied in a given state, and a set of GDL statements that assign new values to the state variables.

SET

Suite execution trace. The SET is an XML markup language that describes a trace of a test suite execution.

State

The state of a model is its current situation including its responses to prior stimuli. The state of a behavioural model is specified by the values of all its global variables (or state variables).

Suite

A test suite is a sequence of tests.

Successful

A test case outcome that indicates that all of the test case's transitions completed with states predicted by the transitions' associated rules.

Test

A test (or test case) is a sequence of transitions starting at a start state and finishing at a final state.

Test Constraint

A test constraint is a limitation on the form of test cases that the GOTCHA test generator can produce.

Test Directive

A test directive is either a test constraint, coverage criterion, or run-time parameter of the GOTCHA tool that influences the test suite produced.

Test Suite

A test suite is a set of test cases.

Transition

A transition is an event that causes a behavioural model to change its state. A transition is specified by a state of the model, a rule that is enabled in that state, and the new state that results from activating the rule. The first state is called the source of the transition, and the second state is called the destination of the transition. In an abstract test, a transition is given by the rule and its destination state only, since the source of the transition is always the destination of the preceding transition.

Undecided

A test case outcome that indicates that one of the test case's transitions completed with an alternate state predicted by the transition's associated pluRule.

UUT

Unit Under Test: the software application being tested.

10 Index