FeaVer 1.0 User Guide

Gerard J. Holzmann and Margaret H. Smith

Preface

FeaVer is a tool that can be used for the verification of distributed systems code written in ANSI-standard C. This guide is meant to be read as a first introduction to the tool and to the methodology on which it is based.

We begin with a brief explanation of the need for a tool that can trap concurrency related errors, followed by an overview of FeaVer's basic mode of operation. The main theme introduced here, and that will return throughout this document, is the definition of a *verification test harness*. The working of FeaVer is always determined by the test harness definition. Learning to use FeaVer, therefore, means learning to design and debug test harness descriptions.

FeaVer can be used either with a basic command-line interface or with the help of a more refined graphical user interface. In some cases the command-line interface is preferable, giving slightly more insight into the details of FeaVer's operation. But undeniably, especially when working with a fully developed test harness description, the graphical user interface can be more convenient. For first-time use, it is recommended to begin by developing a good basic understanding of the main processing steps that FeaVer performs when it runs. In the first few chapters of this guide, therefore, we will focus primarily on the command-line interface and a textual description of the main elements of a test harness. In a later chapter the graphical user interface is covered. A comprehensive review section at the end of this guide, is meant to be a point of reference for the more experienced user of FeaVer when constructing test harness definitions.

At the time of writing the FeaVer tool is being developed as a research prototype, and it continues to evolve. If you detect any information that is confusing or just plain wrong, please let us know and we will correct it.

Contents

Preface 1

Introducing FeaVer

What About Testing? 5
Limited Observability 5
Limited Controllability 5
The Alternative 6
The FeaVer Test 6
Properties 7
Limitations 8
Overview 8

How FeaVer Works

Model Extraction 10
Choices 11

A Simple Test Harness 11

%F 12
%X 12
%L 12
%P 12

Defining a Test Harness 12

Process Control 13
Control States 14
Code Encapsulation 16
Second Attempt 17
Mapping Rules 18
Second Example: Process Communication 20
Tracking Data 23

Checking Assertions 23

Causality 24
Timeline Properties 24
Temporal Logic 24

Checking Up on FeaVer 24

Preprocessing 24
Parsing 25
The Run Script 25
Spin 25
C-compiler 25
Execution 25

Graphical User Interface

Menus 27

Accept Cycles 27
Non-Progress Cycles 28
Property Violations 28
Invalid Endstates 28
Creating Mapping Tables 30
Control State Information 31
View Menu 31
The Check Button 31

..... 34

Defining TimeLine Properties

Overview 34
The Timeline (Vip) Tab 35
..... 35
File Operations 35
..... 36
Operations on Events and Constraints 36
Defining Symbols 38
Linking Events and Constraints to Code 40
Operations on Time Marks 44
Generating automata 44
Credits 45

Reference Guide

Overview of Test-Harness Commands 46
%F 47
%G 47
%L 47
%H 51
%D 51
%C 52
%P 52
%O 52
%Q 52

Recommended Code Structure

Event Driven Code 53
Example 53

Trouble-Shooting Guide

Small Example 54
Exercise 56

Step 1 57
Step 2 57
Step 3 57
Step 4 58
Step 5 58
Step 6 59
The Final Run 59
Vacuity Checking 60
Tracking Down What Happens 60
%H 61
The .M file 62
%D 62
%H vs %D 62
%D vs %C 62
%P 62

1. Introducing FeaVer

It is no secret that writing reliable code for concurrent systems applications can be quite difficult. Without tools, not even the best among us can produce such code flawlessly. It is certainly harder to write reliable multi-threaded code than it is to write sequential code, and even in sequential, standalone applications a few bugs always manage to escape scrutiny, no matter how talented or careful a programmer is. Euphemistically we call such bugs *residual software defects*. They are the bugs left for the end-user to find. Often these are harmless annoyances. But the potential for more dramatic failures is also there.

The residual defect ratio is typically smaller for more experienced programmers. Curiously, though, the smaller number of bugs of the experienced programmer may well be harder to find and fix than the larger number of simpler bugs of the less experienced programmer. Experience alone does not guarantee software quality.

What About Testing?

There are many techniques for evaluating code quality and for reducing the expected number of residual bugs: manual code walkthroughs, peer review, and of course plain old unit and integration testing. There is no question that these methods work: they catch bugs, but they are not perfect. Conventional testing techniques have some restrictions that limit the number of bugs that can be caught in this way. This is sometimes called the *quality ceiling* that is imposed by standard testing techniques.

To test a piece of code, either in isolation or in a given context, a tester usually creates a *test harness*: special instrumentation that is used to efficiently and reproducibly adminster a series of tests and to evaluate the results. At least, this is the intent. The keyword here is *reproducibility*, and this is precisely what makes it hard to set up a reliable test harness for a concurrent, or multi-threaded, system with conventional techniques.

Limited Observability

How can one get a logically or physically distributed system to perform reproducibly, even in the presence of errors? A distributed system typically consists of a number of asynchronously executing processes or threads. The details of the process executions are determined by schedulers. Typically there is one independent process scheduler per host computer in the system. A tester can try to setup a test harness to verify that for any given input the system as a whole will generate the required output, but there is not always a deterministic relation between inputs and outputs in a system like this. The behavior can depend on subtle timings and non-reprodicible interleavings of events. It is hard to tell, at a sufficiently low level of granularity, precisely how process executions are interleaved in time. Yet this knowledge can be of critical importance in tracking down race conditions and timing errors. This is the fundamental problem of *limited observability* in conventional testing.

Limited Controllability

A related problem is that even if the tester could know exactly how the process executions had to be interleaved in time to reproduce an error scenario, it is in general not possible to enforce such a schedule in a direct execution. This is the second fundamental problem in conventional testing: *limited controllability.* In a standard test environment, limited controllability and observability restrict the tester’s ability to thoroughly exercise a system. As a result, some of the most difficult to diagnose bugs can slip through the testing phase and hide as residual

```
int shared = 0; int *ptr;

void
thread1(void)
{       int tmp;

        ptr = &shared;
        tmp = shared;
        tmp++;
        shared = tmp;
}

void
thread2(void)
{       int tmp;

        if (ptr)
        {       tmp = shared;
                tmp++;
                shared = tmp;
                assert(shared == 1);
        }
}
```

FIGURE 1. Simple Threads Example

defects in production code, with at least the potential of striking at the least opportune moment.

In this guide we'll explore an alternative way of checking distributed systems software that tries to avoid the limitations of conventional testing. We'll start with a simple example.

The Alternative

Consider the C program fragment shown in Figure 1. The fragment defines the code for two concurrent threads of execution, named `thread1` and `thread2`. The two threads share access to an integer data object named `shared`, and each thread attempts to increment the value of this shared object by one. If the second thread reaches its `assert` statement before the other thread performs its increment, then the assertion will be true, but if the first process is faster, the assertion will be false. In a normal testing environment, the test could be executed many times, and pass each test. The standard metrics of code coverage and branch coverage fail when applied in a situation like this one. There is no good way to make sure that sufficient tests were done to rule out the possibility of a race condition in the code that could jeopardize its validity. It is easy to see in this case that there is at least one execution that will make the assertion fail. Is there a way to guarantee that such executions will be detected during system testing?

The FeaVer Test

If we submit this piece of code to the FeaVer tool, with a small test harness definition that we'll discuss shortly, the output is as shown in Figure 2 . For simplicity we use the command line interface, named **`feaver_cmd`**. We will discuss the graphical user interface later, in the chapter "Graphical User Interface" on page 26.

FeaVer generates an execution trace through the code, with a columnated interleaving of the statements that are executed by the two threads, leading up to an assertion violation: clear proof that such a violation is possible. The scenario that is

```
$ feaver_cmd threads.c
[...some preprocessing is done...]
thread1(0):[ now.ptr=&(now.shared); ]
                thread2(1):[ now.ptr ]
thread1(0):[ Pthread1->tmp=now.shared; ]
thread1(0):[ Pthread1->tmp++; ]
thread1(0):[ now.shared=Pthread1->tmp; ]
                thread2(1):[ Pthread1->tmp=now.shared; ]
                thread2(1):[ Pthread1->tmp++; ]
                thread2(1):[ now.shared=Pthread1->tmp; ]
pan: precondition false: (now.shared==1)
[...more details on the final state follow...]
$
```

FIGURE 2. Output for the Threads Example

generated here begins with the execution of a step in **thread1**, followed by a step in **thread2**, three consecutive steps in **thread1** and then finally three steps in **thread2**. At the last step the value of the variable **shared** is equal to two, not one, and the assertion fails. This is of course not the only possible execution that would lead to the assertion failure, but the one example will suffice to show that the problem exists.

A few other points are worth observing. First note that the statements that appear in the trace look different than in the original code. Local data in **thread1**, for instance, is prefixed with a structure reference "**Pthread1->**," or "**Pthread2->**", and global data is prefixed with "**now.**" The names of the threads also have a number appended to them in parentheses. The number is the process number (the *pid*) that FeaVer assigns to the concurrent process threads. The offending sequence is printed by a program **pan** that is generated, compiled, and invoked by FeaVer to perform its test for this code. The reasons for all these peculiarities will become clear shortly.

Properties

FeaVer can determine reliably if there is any system execution that could cause the failure of assertions that are part of the program text. There are three basic types of assertions that can be checked, of which the one from the threads program is the most basic one (and the one that most programmers are familiar with already). More details on these and the other types of assertions can be found in the section on "Checking Assertions" on page 23.

There are also other types of properties that FeaVer can check for all code that it instruments for a check; these include:

- Indexing errors for statically allocated arrays.
- Pointer dereferencing errors (null pointers).
- The existence of non-progress cycles (sometimes called *livelocks*).
- The existence of reachable system deadlocks (invalid terminations).
- Violations of time-line properties (a subset of temporal logic).

Builtin to FeaVer's graphical user interface is a timeline property editor that can capture a broad range of simple requjirements on causal and logic relations between events. Beyond this, FeaVer can also check any ω-regular property

(including properties formalized in the logic LTL, linear temporal logic), but such power is fortunately only rarely needed in performing software verifications.

Limitations

The FeaVer system is designed to check *logical* or *functional* system properties as efficiently as possible. This means that FeaVer cannot be used to check *real-time* or *performance* related issues. There is no notion of absolute time in FeaVer itself or in the logic that is used to define the checks.

The FeaVer system primarily targets the verification of *concurrent* software systems, written in ANSI-C. It is not much good for checking basic computational properties of sequential code. For instance, it would not be very good at checking that a square-root routine always functions as advertised, or that a sorting algorithm is implemented correctly. It can diagnose a broad class of concurrency related problems though, including race conditions and system deadlocks. It is possible to link seperately compiled code segments written in C++ into a FeaVer test harness. These code segments can define system routines and test drivers that can be employed as atomic modules in the test system. The most thorough types of checks that FeaVer can do, however, rely on its capability to convert C code into verification models, with the help of a model extractor. There is as yet no support for model extraction from code written in languages other than ANSI-C.

FeaVer uses a few background tools to perform its work. A first is a model extractor called **`modex`** that parses the C code and guided by FeaVer can extract verification models from that code. The second tool is an extended version of the logic model checker Spin. Spin generates test-code from either hand-written models or from the models produced by **`modex`**. FeaVer then compiles this code with a regular C-compiler and executes it, reading back the results.

Although detailed knowledge of model-checking techniques is not required to use FeaVer, some knowledge of model checking tools can be helpful in setting up more complex tests or in diagnosing problems that are encountered along the way. The user who is already familiar with the use of Spin will be at a definite advantage when using FeaVer. The principles of model checking, and the standard use of Spin is described elsewhere (see for instance [H97] in the reference section on page 63).

Overview

We will begin with an overview of the basic working of FeaVer, followed by a tutorial example that shows in a little more detail what steps are generally needed to perform a thorough systems check. As noted, the definition of a test harness takes a central place in all this. We will first explain some of the default assumptions that FeaVer will make about test harness code. Then we will review the role of the test harness in defining reproducible tests in more detail, expecially with respect to:

- process control,
- inter-process communications, and
- the tracking of state information.

After this we will review the types of properties that FeaVer can check for concurrent systems code in somewhat more detail. For the examples we will use the basic command line interface to FeaVer. More support for defining tests and for maintaining test harnesses for evolving code is provided by FeaVer's graphical user interface. We will cover it in a separate chapter, once the main functionality of FeaVer has been discussed.

Next we include a reference section, reviewing the main components of a test harness definition, including all commands and controls that can be used for defining a test system.

For complex applications, the setup of a FeaVer test can sometimes be simplified if the source code is given a preferred structure. We discuss that structure in the final part of this document and present some examples.

Finally, we present some guidance in troubleshooting test harness definitions, explaining in more detail what FeaVer does and how all the pieces fit together.

2. How FeaVer Works

FeaVer can build a test system that consists of three main elements:

- user-defined test-drivers,
- unmodified source code fragments (typically procedures), and
- models extracted from portions of the source code.

Not all these elements need to be present in every test harness definition though. A test harness, for instance, could be defined to link user-defined test drivers to unmodified source code fragments, to drive the code through a series of tests. Generally, though, greater control over a test can be obtained by using FeaVer to extract models from critical portions of the code. We will often refer to this as *test instrumentation*. The instrumentation serves two purposes:

- It allows FeaVer to insert extra assertions into the code that can trap common types of errors, such as out-of-bound array indexing and null-pointer dereferencing. The model checker that is used to perform the checks can detect if there are any system executions that can violate the assertions.
- It allows the model checker to control the interleaving of asynchronous process executions so that relevant interleaving patterns can systematically be enforced and checked. The instrumentation can secure both observability and controllability of critical pieces of code.

The level of granularity at which test instrumentation is performed is always under user control, giving the user considerable freedom in determining the scope and thoroughness of a system test.

Model Extraction

FeaVer can optionally extract detailed models from user-selected portions of the source code. The models are largely functional replicas of the original code, rendered in the language of the model checker that is used in the background. FeaVer, however, allows the definition of optional filters that can be used to simplify the version of the code that is going to be tested, for instance by systematically suppressing details that are not strictly relevant to the test at hand. By suppressing detail, the complexity of a check can be reduced.

By defining abstractions as filters, the version of the code that is verified can be tuned without the need for modification of the original source. (The modifications are made during the model extraction process, leaving the original source code untouched.) Almost all changes that might be needed to perform a test can be defined through user-defined filters, which can be defined in the test harness and automatically applied during model extractions.

As noted, the use of a filter is mostly optional. It can be used to speed up a software test, or to increase its coverage. There is one case, though where the use of a filter is required, and that is to *lift* operations that concern *process-control*, *thread control*, and *inter-process communications*. In these cases, the filters can be used to convert C-versions of these operations into equivalents that can be executed under the control of the model checker. (Below we sometimes also refer to the filters as *mapping tables*, *lookup tables*, or *abstraction tables*. These terms all refer to the same mechanism that can be used to control model extractions.)

Process control operations include, for instance, process *instantiation*, *suspension*, and *termination*. In the threads example that we saw before there were no explicit

```
$ cat threads.prx
%F threads.c
%X -x
%L
Import _all_ _all_
%%
%P
#include “_modex_all.spn”
%%
$
```

FIGURE 3. Prx Verification Test Harness for Threads Example, with Defaults FIlled in

thread control operations. Through definitions in the test harness the user can determine how many threads there are, which code fragments they execute, when they are suspended and released, how they interact, and when they terminate. We will see examples of these types of definitions shortly.

Choices

FeaVer’s strength lies in its support for model extraction and for defining a reproducible test environment for concurrent code, including all the required process control operations. As noted, though, not all of the source code of an application will generally need to be instrumented explicitly. In typical cases, verification models are extracted from only a small core piece of an application. The remainder of the code is often linked to the generated test system and executed *as is*.

By chosing which parts of an application are instrumented and which parts are run without instrumentation, the user can tradeoff the scope and cost of checks. As a rule of thumb: purely deterministic, and sequentially executing code, that contains no access to shared data, no interprocess communications, and no process or thread control actions, need not be instrumented and can be run as is. Other code should be instrumented through model extraction. Fortunately, the latter is often a relatively small portion of a total application, and it can often be made even smaller by structuring the code in a canonical way. We will return to the recommended (though not required) structure of the source code towards the end of this guide (see page 53).

A Simple Test Harness

In the threads example, we invoked the FeaVer command with a test harness definition consisting of just a few commands. A single test harness specification is fully defined, in a file with the extension **`.prx`**. All components in a **`.prx`** file have defaults, which means that not all components need to be present in all test harness specifications. The definition of the test harness for the threads example is shown in Figure 3 (where the last three lines match a default and are optional in this case).

The specification contains a set of commands to the model extractor, each identified by a different command prefix. A command prefix always starts with a percent sign (%) in the left margin, and is followed by a unique letter or phrase. There are about ten different commands that can be used. Only four appear in this example.

The first two commands (**%F** and **%X**) are single-line commands, with the parameters to the command placed on the same line as the command name itself. The other two commands (**%L** and **%P**) are multi-line commands, with the details following on separate lines. A multi-line command, such as **%L** or **%P**, is terminated with a double percent sign **%%** in the left margin.

We'll briefly discuss the meaning of the four commands that appear in the sample test harness, in the order in which they would normally appear in the **`.prx`** file.

%F An %F command identifies a single C-source file that will be the target of subsequent model extraction commands (test instrumentation). Model extraction commands are labeled **%X**, so the **%F** command applies primarily to the **%X** commands that follow. There can be multiple **%F** commands, each followed by one or more **%X** commands. In the absence of an **%F** command, the default target C-source file is assumed to be the name of the test harness file with the extension **`.prx`** replaced with **`.c`**. So the default target for **`threads.prx`** is **`threads.c`** and similarly, the default test-harness file for the source file **`threads.c`** is **`threads.prx`**. Through the use of explicit **%F** commands, we can override or extend this default.

%X An **%X** command identifies a procedure in the target C-source file from which FeaVer will extract a model to be used as part of the test system. The **%X** command can take a number of different parameters that determine the precise way in which model extraction will be performed. The **`-x`** parameter tells FeaVer to extract a model from *each* procedure that appears in the target C-source file.

%L An **%L** command is used to define an optional filter (i.e., *mapping table*) to be used in the model extraction. (The L stands for Lookup table.) It can be used to define simplifications, abstractions, modifications, or extensions of the code. The mapping table can also contain commands that determine whether local or global data objects from the target source file are included into the extracted models. One such special command appears in the default test-harness for the threads example: the **`Import`** command. The version of the command that is listed states that *all* data objects from the source are to be included in the generated verification model. The first argument of the **`Import`** command normally specifies the name of a data object, and the second argument specifies the scope in which it appears. The keyword **`_all_`** can be used to define that *all* data objects and/or *all* scopes are intended to be matched. In the example the filter itself is unnamed and therefore by default is assumed to apply to *all* model extractions performed through this test harness. A finer level of control is available by introducing a series of individually named filters (e.g., **`%L`** `name`), and by referring to one of these filters by name within a model extraction command (e.g., **`%X -L`** `name` **`-x`**).

%P A **%P** command defines how the various pieces of a model are to be combined and executed. In the default case, illustrated in the example, There can be only one **%P** command per test harness. FeaVer generates a file with with the extracted models that is included here. The contents of this segment matches the default that FeaVer would use if no **%P** command was present. In the example we've included the default file that is generated by FeaVer with all extracted models.

The above first description of a test harness is no more than a brief snapshot. We will fill in some of the details in the remainder of this section.

Defining a Test Harness

The FeaVer test harness is used to specify the details of a system test. All the test harness specifications are stored in a single file, with the suffix **`.prx,`** grouped into different commands, each dealing with a different aspect of the test setup. The test harness allows the user considerable freedom in defining the details of a test. In this introductory section we will use some examples to discuss the most common controls. A more detailed reference section explaining all available options follows later in this guide.

The test harness defines which processes are part of the test system, how they interact, and what code they execute. In a typical work-flow, the user starts with

some fairly basic assumptions about the test system, and refines it step by step into a more sophisticated test harness, guided by the responses from the FeaVer tool, the model extractor, the model checker, and the C compiler. Each of the background tools is invoked at various points in the testing process to verify the validity of the code that is being generated.

A test harness generally defines:

- Which processes are part of the test system, and where their definition is to be obtained: from source code via FeaVer's model extraction engine, from user-defined test drivers expressed in the Process Meta Language (Promela) of the model checker, or from uninstrumented source code.
- How these processes communicate. Typically we will replace the generic process communication mechanism from the source code with a special purpose mechanism that is used within the test harness. It would, for instance, needlessly introduce processing overhead if the processes were to communicate via socket connections over a real network. A more efficient test can be created by replacing that mechanism with an internal mechanism for passing data through shared buffers. The user can chose here to define the send and receive operations directly in C, or to use builtin message passing primitives from the model checker.
- Which data objects together define the *control state* of the system under test. State information typically includes any data object that must persist across different calls of the event processing loop (or its equivalent). Temporary data, or unchanging configuration data, typically does not contain persistent state information. When model extraction is used, FeaVer can determine which data objects hold state information, and include them in the test system. For code that is outside the pervue of the model extractor, though, it is the user's responsibility to identify the relevant data objects and to setup tracking for these in the test harness.

We will take a closer look at each of these issues with the help of some examples. We will start with a more detailed treatment of *process control*: a central issue in any test system.

Process Control

The determination of the number of concurrent processes in the test system, the code they execute, and their point of instantiation, is ultimately decided in the %P segment of a test harness.

In the threads example, two concurrent processes were generated by FeaVer, one for each routine that appeared in the source code, and both are included in the test. In general, we will need a little more control in determining which processes will be part of the test system and what code they execute. In many cases, the code to be executed will be found in the source code of the application, but rarely will *all* of the source code of an application be required for a meaningful test. Some code can be stubbed, for instance if it interacts with remote systems or processes, or relies on hardware interfaces. We can replace such code with our own test drivers that simulate the possible responses in a controlled manner. Other code fragments may perform routine computational functions that do not involve concurrency or externally visible operations. Such code need not be under the control of FeaVer, but can be invoked from external code via procedure calls. FeaVer needs to control only the main threads of concurrency and to monitor possible process interactions.

The **%P** segment of the test harness will generally include declarations of processes that are *not* defined in the source code itself, e.g. as test drivers that simulate parts of the environment in which the system is to be tested. Such test drivers can simulate, for instance, user input, file system operations, hardware responses. Through the systematic use of user-defined test drivers we can decouple the test system from components over which a tester would otherwise have no direct control. In this way we can make all system tests we perform fully reproducible. This process is often called *closing* the system to its environment, to build a standalone test system.

The **%P** segment in the test harness defines the full context in which the test system operates: no *process* (i.e., independent thread of execution) is part of the system unless it is declared in this segment. A small number of other declarations may also be placed here. For instance, we can declare channels for process communications here, if we decide to represent the process interactions not in C code but in Promela, the specification language of the model checker. To effect this we'd have to set up a mapping from C routine calls to Promela primitives for message passing, through the use of mapping tables. We'll return to this when we discuss the **%L** command, with which mapping tables are defined.

> The **%P** segment does *not* include definitions for parts of the source code that are to be used in their original form. If, for instance, a call on procedure that is defined in the original source appears within a test driver, then the assumption is that this fragment of source code will be linked with the test system. We will describe later how this linkage is accomplished.

As a simple example of the development of a test harness, let us look at a small, and flawed, algorithm for setting and releasing mutual exclusion locks, shown in Figure 4 on page 15. There is no code shown for user processes that might try to use the lock and unlock routine from this example. Fortunately, we don't need such code: we can define our own test drivers in the FeaVer test harness when we *close* the system.

A first attempt to define a test harness might avoid model extraction alltogether. We can do so by defining two user level processes that invoke the lock and unlock routines as C procedure calls. What happens inside these procedures is now beyond the control of the test system. In particular, the test system now cannot control the interleaving of the statement executions *within* each of these procedures during a test. The procedure calls performed in this way have the effect of indivisible, atomic actions. (Often this is quite acceptable, for instance if the procedure call has only local effect on the state of the system. In this case the choice is indeed dubious, since the procedures access shared data. We will return to this shortly.) Thetest harness for this first attempt is shown in Figure 5 on page 15. We will explain the commands and parameters used here shortly.Inter-process communication in this case occurs through access to the shared data objects x and y. So in this case we need not worry about handling inter-process communication via send and receive operations just yet.

Control States

In the checking system, we do have to define what precisely will constitute the control state of the system at each point of time. Part of the control state of a process (e.g, the value of the program counter) can be determined automatically by FeaVer, but data can also contribute state information. FeaVer is conservative in determining which data objects hold additional state information: the user must explicitly declare all data objects that hold state information in the test harness.

```
/* Bad Mutex Algorithm */

int x, y;

int
lock(int pid)
{
   x = pid;
   if (y != 0 && y != pid)
          return 0;    /* fail */

   y = pid;
   if (x != pid)
          return 0;    /* fail */

   return 1;           /* success */
}

void
unlock()
{
   x = 0;
   y = 0;              /* reset */
}
```

FIGURE 4. Flawed Mutual Exclusion Algorithm

```
%D
extern int x, y;/* persistent state information */
%%
%L
Track     &x     sizeof(int)
Track     &y     sizeof(int)
%%
%P
int count = 0;         /* global counter */

active [2] proctype user()
{
   do
   :: c_expr { lock(Puser->_pid + 1) } ->
          count++;
          assert(count == 1);
          count--;
          c_code { unlock(); }
   od
}
%%
%O mutex_attempt.c
```

FIGURE 5. First Attempt to Capture Process Control

For the mutual exclusion code, the value of the two integer variables x and y hold all essential state information. We define them as state variables in the **%L** segment of the test harness with a **`Track`** statement. A **`Track`** statement takes two

arguments. The first argument defines a pointer to the data object to be tracked, and the second argument specifies its size. There can be only one data object per **`Track`** statement.

Since the two data objects appear only in external code (in the file **`mutex_attempt.c`**), they must be declared as external objects before they can be referenced. This is done in C syntax with a **`%D`** command. The **`%D`** command lists all external data objects that the model checker must be able to refer to, and it can include any necessary structure or type definitions that are used. We can also introduce new, local, data objects in this segment, for instance for use within the test system itself. None are needed in this case, so the **`%D`** segment contains just one single line of text.

Next we come to the central part of the test harness where process control is defined. This is done with a **`%P`** command. The syntax used in a **`%P`** segment is the language of the model checker Spin, Promela. For the reader unfamiliar with Spin we will briefly explain the code shown in this example.

At the start of the **`%P`** segment we placed the declaration for an integer variable named **`count`**. This variable is used to check the accuracy of the locking code. Alternatively, this data object could also have been declared in C syntax in the **`%D`** segment. The method of declaration determines whether the data is automatically considered state information.

- A data object declared in a **`%D`** command is not automatically tracked as control state information. For this we need to include a **`Track`** statement in a **`%L`** segment, as shown for variables x, y, and z in Figure 5 on page 15. A Promela data object that is declared in a **`%P`** command is always tracked automatically. Clearly, in a **`%P`** segment we can only use Promela data types; in a **`%D`** segment any C data type can be used.
- A data object declared in a **`%D`** command can be used in C code using C syntax rules. The data object can be used in Promela code only in **`c_code`** and **`c_expr`** statements (see below). A data object declared in a **`%P`** segment, on the other hand, can be used in Promela code using Promela syntax, and only with extended syntax in **`c_code`** and **`c_expr`** statements. The extended syntax adds a special prefix that locates each data object within the model checker's state vector. For global data objects (e.g., the integer **`count`** in Figure 5 on page 15) the prefix is "**`now.`**", for data objects declared and used locally in proctype **`pname`** the prefix is "**`Ppname->`**". We have, for instance, used a reference to a predefined local variable **`_pid`** with this syntax in the Figure 5 on page 15.

The **`%P`** segment in the example test harness next defines two instantiations of a single process-type that we have named **`user`**. We have arranged for these two instantiations to execute a short sequence of statements, iteratively. Each process first waits until it can set the lock, it then increments the variable **`count`** and checks its value in an assertion, next it decrements the variable, and releases the lock. If all is well, the value of **`count`** will always be exactly one when the assertion is reached.

Code Encapsulation

Arbitrary C syntax can be used in any **`c_code`** and **`c_expr`** statement within a Promela proctype declaration. The difference between these two types of statements is that a **`c_code`** statement will always be executed unconditionally and atomically, while a **`c_expr`** statement can only be executed (passed) if it returns non-zero when its body is evaluated as a C expression. The evaluation of a

```
$ feaver_cmd attempt1.prx
....no errors are found....
$
```

FIGURE 6. A Run with the First Test Harness

c_expr is indivisible (i.e., atomic). For obvious reasons (**c_expr** statements may be evaluated many times to determine their executability) a **c_expr** is required to be side-effect free: it may only evaluate data, not modify it.

We have not used **%X** or **%F** commands in this test harness, because no models are extracted from the code just yet: the routines are simply called as external procedures from within our manually defined test-drivers. To obtain an executable test system, all we have to do is to link the model checking code that is generated by FeaVer to the source code of the `lock` and **unlock** routines themselves. This is done with the **%O** command that concludes the test harness description. If multiple external files must be linked they can be specified on the same line, and similarly if specific compiler directives must be used, they are specified on the same line as well. (There can be only one **%O** command per test harness file.)

The result of a run with the command-line version of FeaVer is illustrated in Figure 6 on page 17. No problems are reported. This means that if the lock and unlock code can be used atomically, without any possibility of interference, the code is secure and will behave as expected. This is of course not a very realistic assumption for this type of code. (Much simpler versions of this code will also have that property.) In a real system there may well be concurrent calls on the lock routine by multiple processes, or overlapping calls on the lock and unlock routine. To test if the code can stand up under those conditions must move the level of atomicity down to the statement level *within* the lock and unlock routines, and for that we have to extract models from these two routines.

Second Attempt

There are several ways to instrument the code for the right level of atomicity. One way would be to define two user processes as C test drivers, that call the lock and unlock routines, and to extract models from both the user processes and the lock and unlock routines, relying on a mechanism in FeaVer to replicate simple procedure call mechanisms (using **%X** parameter **-e**). We will discuss a slightly different version where the test drivers are written in Promela and the extracted models for lock and unlock are both imported into the Promela test driver as inlined code fragments.

This version of the revised test harness is shown in Figure 7 on page 18. It is set up to extract models from the **lock** and the **unlock** routines (using **%X** parameter **-n**). The test harness begins by identifying the source file in which these routines are defined with an **%F** command. After the **%F** command we see the first of two invocations of the %X command. The first invocation tells FeaVer to extract a model for the procedure named **unlock**. The use of the argument **-n** says that we want to generate a model fragment without the default process encapsulation (since we want to inline this into a manually defined test driver process). So in this case we will define the `proctype` header ourselves, and we use the freedom this gives us to outline the general structure of the user process as well. (More details on the types of parameters that can be used with an **%X** command, see page 47.)

The second **%X** command targets procedure `lock`. Also here the argument `-n` is used, so that the code can be included into an existing proctype. This **%X** com-

```
%F mutex_attempt.c
%X -n unlock
%X -L lock.lut -n lock
%L lock
Import    x      Global
Import    y      Global

return    0      goto Wait
return    1      goto Locked
%%
%P
int count;/* global counter */

active [2] proctype user()
{  int pid = _pid + 1;

Wait:do ::
#define PlockPuser
#include “_modex_lock.spn”

Locked:   count++;
          assert(count == 1);
          count--;
#define PunlockPuser
#include “_modex_unlock.spn”
   od
}
%%
%G
shortest: 1
%%
```

FIGURE 7. Second Test Harness

mand begins with an argument **`-L`** with parameter **`lock`**. This specifies that we want to define a specific, named, lookup table for this particular extraction. The lookup table is defined in the **`%L`** segment with the matching name. The named **`%L`** segment, **`%L lock`**, begins with two **`Import`** commands, each with two arguments (arguments are always tab-separated). The first argument gives the name of a data object that we would like to import into the model that is generated. The second argument gives the scope in which that object is declared in the source code. In this case, the scope is global and the data objects in question are the integers x and y. It would be less convenient to setup *tracking* for these state variables in this case, because we are not including any native C code in this version of the test harness. We could have declared x and y in a **`%D`** segment in the test harness and then define the tracking information as before, but it is simpler to use the Import mechanism in this case.

Mapping Rules

The next two entries in the **`%L`** segment of the test harness are especially interesting. Notice that the original lock routine returns a value of 0 or 1, depending on whether the lock could be obtained or not. We want to inline the code for both the lock and the unlock routine into a single Promela proctype here, so the return statements need to be converted into semantically equivalent Promela syntax (the specification language Promela has no procedure concept and therefore no return statement). There are several ways of handling this. Here, we have translate the

```
$ feaver_cmd attempt2.prx
                user(1):[ now.x=Puser->pid; ]
                user(1):[else]
user(0):[ now.x=Puser->pid; ]
user(0):[else]
                user(1):[ now.y=Puser->pid; ]
                user(1):[ (now.x!=Puser->pid) ]
user(0):[ now.y=Puser->pid; ]
user(0):[else]
                user(1):[ now.x=now.y=0; ]
                user(1):[ now.x=Puser->pid; ]
                user(1):[else]
                user(1):[ now.y=Puser->pid; ]
                user(1):[else]
                user(1):[count = (count+1)]
user(0):[count = (count+1)]
pan: error: assertion violated (count==1) (at depth 16)
....some more statistics....
$
```

FIGURE 8. A Run with the Second Test Harness

returns into jumps to the included code fragments. The translation is specified as a rule in the lookup table. An **%L** command can contain an arbitrary number of such conversion rules, which are specified as a table of two columns. The left hand column lists statements from the source code (in a canonical form that we'll return to later), the right hand column gives the replacement text in Promela syntax.

After the **%L** command, the test harness contains the definition of the user processes in the **%P** segment. The generated code for routine `X` is always placed in a file named **`_modex_X.spn`**. We place a label **`Wait`** in front of the included code for the **`lock`** procedure and a label **`Locked`** after it. Failure to obtain the lock now results in a jump backwards, so that the attempt can be repeated. Success results in a jump forwards. There's one final thing to watch for here and that is that the local variables in the source file refer to the routine named **`lock`**, while in the extracted model they should refer to our test-driver process that we named `user`. Two macro definitions make sure that all local variable references inside the two model fragments generated by FeaVer will be interpreted correctly. We'll explain the **%G** command shortly.

This time if we run the test, the model checker reports that the assertion can be violated, meaning that the `lock` and **`unlock`** routines are indeed unreliable. The model checker generates an example execution, with one possible interleaving of the process executions that leads to this error. The sequence is not necessarily the shortest possible such sequence, but it is usually more than adequate to diagnose the problem. If it is nonetheless important to find the shortest possible example sequence that breaks a given property (for instance for a description like this one), we can use a somewhat more costly option of the model checker to do so. We can do this by adding a **%G** segment to the code, where run and compile time parameters can appear. In this case it suffices to define that we are interested in the shortest counter-example by adding one line to the test harness. For the flawed mutual exclusion algorithm, this yields the execution sequence shown in Figure 8 on page 19.

```
typedef struct Msg {
    short  seq;         /* sequence number  */
    char   *cont;       /* message contents */
} Msg;

int
abp_snd(Msg in)
{   Msg out;

    if (in.seq == out.seq)
    {
          if (!get_data(out.cont))   /* more data  */
                return 0;            /* we're done */
          out.seq = 1 - out.seq;/* flip sequence number */
    }
    send(out);          /* send message */
    return 1;
}

void
abp_rcv(Msg in)
{   Msg out;
    int expect = 0;

    out.seq  = in.seq;
    out.cont = (char *) 0;
    send(out);          /* always acknowledge */

    if (in.seq == expect)
    {
          put_data(in.cont);/* accept data recvd */
          expect = 1 - expect;/* flip sequence number */
    }
}
```

FIGURE 9. Alternating Bit Protocol

Second Example: Process Communication

To take a closer look at the various ways in which we can capture inter-process communications, we'll consider another example. Figure 9 on page 20 shows the state update routines for a sender and receiver process in the classic alternating bit protocol. The routines are to be called whenever a new message arrives at, respectively, the sender or the receiver, and they will then generate the right responses. The send routine returns zero when no more data is available to be sent and one otherwise.

The receiver performs a simple check on the validity of the incoming messages, and if the check is passed it stores the data with a call on routine **`put_data`**. The receiver also acknowledges all messages received, valid or invalid, by returning an empty message with a copy of the sequence number from the incoming message back to the sender.

A first version of a test harness for this application is shown in Figure 10 on page 21. The initial part is now familiar. We start with an %F command to identify the source file used. If the code for the sender and the receiver would be located in different files, we'd simply identify each one with an **%F** command that immediately precedes the corresponding **%X** commands. We have two **%X** commands here, one

```
%F abp.c
%X -L abp_snd.lut -n abp_snd
%X -L abp_rcv.lut -n abp_rcv
%L
Import_all_      _all_
%%
%L abp_snd
get_data(...     true
send(out)        c_code { \
                        Pabp_snd->s = Pabp_snd->out.seq; \
                 }; \
                 qr!s
%%
%L abp_rcv
put_data(...     skip
send(out)        c_code { \
                        Pabp_rcv->s = Pabp_rcv->out.seq; \
                 }; \
                 qs!s
%%
%D
typedef struct Msg {
   short  seq;
   char   *cont;
} Msg;
%%
%P
chan qs = [0] of { bit };
chan qr = [0] of { bit };

active proctype abp_snd()
{  bit s;

   do
   :: qs?s -> c_code { Pabp_snd->in.seq = Pabp_snd->s; };
#include "_modex_abp_snd.spn"
   od
}
active proctype abp_rcv()
{  bit s;

   qs!0;  /* start sender */
   do
   :: qr?s -> c_code { Pabp_rcv->in.seq = Pabp_rcv->s; };
#include "_modex_abp_rcv.spn"
   od
}
%%
```

FIGURE 10. Test Harness for Sender and Receiver

for the sender and one for the receiver, and we specify a dedicated lookup table for each one. The **%X** commands are followed with a specification of the lookup tables. There are three **%L** segments here. The first one is nameless, and contains commands that must apply to all model extractions operations. It states that all data objects encountered, local or global, must be imported as part of the generated

model. Next there is a separate **%L** segment for each of the two routines. Note, for instance, that the send routine is called in both processes, but clearly will have a different destination in each case.

The first entry in the lookup table for the sender is a definition for the semantics of the routine **get_data**. For simplicity we will assume in the test that there will be an infinite stream of data to be transmitted, meaning that the **get_data** routine always returns a non-zero result (**true**). The working of the protocol itself is independent of the actual data that is being transmitted as payload in the messages, so we can conveniently ignore the generation and consumption of this data in the test framework. In the lookup table for the receiver, therefore, we replace the call on **put_data** with a **skip**. The use of sequence numbers, however, is critical, and must be represented.

The more interesting part of this test harness is the interpretation of the send operations in the two communicating processes. Long entries in the lookup table can be split across multiple lines, like macro definitions in C, by using a backslash as a continuation character at the end of a line. The send operations can be represented in C, in Promela, or some combination of both. We're using a combination of both here, to illustrate the point.

Two Promela channels are declared in the **%P** segment of the test harness. The channels are declared with a zero capacity, meaning that they define synchronous, rendezvous ports, rather than asynchronous, buffered channels. Using synchronous channel is often attractive in model checking applications, because it generally reduces the complexity of a system test. Since we can abstract from data contents, each message is given just one field, of type **bit**, to store the sequence number. We can send and receive via these channels with the Promela primitives **q!s** and **q?s**, respectively, where q is the channel and **s** the sequence number sent or received. Now we have the problem that the data to be sent is in a C data structure that may not be representable in Promela (actually, in this case it can be represented with a structure definition in Promela, but in general this will not always be the case). So we have to copy from a C data object into a Promela data object. This is done with a **c_code** statement. Within c_code statements, the references to state variables have to be prefixed with either "**now.**" or "**Pprocname->**". In the case of the sender the prefix becomes "**Pabp_snd->**" for both the Promela object and the C structure reference.

We need to import not just variable declarations but also all C type declarations that are used by these. This is what the **%D** segment of the test harness is for. Its function is to make sure that the types are known to the model checker. In this case we only need a type declaration for a **Msg** structure. The type declaration need not be identical to the one used for the C implementation. If the abstractions allow it, it can often be made quite a bit simpler by omitting all unused fields. In this case, we could consider omitting the contents field entirely, but since there is at least one explicit reference to this field in the receiver, we should then also put an entry into the lookup table for the receiver to map that reference to **skip**. We have not done so here.

The **%P** segment contains the definition of the sender and receiver, in the by now familiar format. The send process is triggered to send the first message with a dummy acknowledgement at the start of the receive process model. (If we would accidentily forget to add this initialization, the model checker would warn us of a deadlock in the initial state of this system.) We have a choice here to send either a zero as the sequence number or a one. If we pick the wrong number, the first mes-

sage transmitted by the sender would be garbage (since the sender will assume that the 'last' transmitted message had to be repeated). The first sequence number used by the sender defaults to zero, and thus the 'wrong' first acknowledgement sequence number to send would be a one.

There are several things we could check fort this implementation of the alternating bit protocol at this point. We could formally demonstrate the above observation about the required initial trigger for the transmission. We could check if it is possible for this system to reach a state from which no data will ever be accepted again, wether the system still has hidden deadlock or hangup states, etc. These features are best understood in the domain of model checking, so we will not discuss them in more detail here.

Tracking Data

By default, no data objects is tracked or imported by FeaVer: this must always be defined explicitly. There are some simple shortcuts to tell FeaVer to do so for all data objects encountered, as illustrated by the `Import` command in the first lookup table in Figure 10 on page 21 In general, though, it is wise to be frugal in the selection of data objects that is imported or tracked. In a first attempt to develop a working test harness it is reasonable to use a broad net in importing all data for the code fragments on which model extraction is used, and to setup tracking for persistent data objects used in externally linked code. To increase the scope, coverage, and efficiency of a check, though, it is good to revisit these initial choices later and carefully inspect which data objects were included and whether or not they serve a useful purpose. A broad "**`Import _all_ _all_`**" statement can then fruitfully be refined into a more specific and selective set of imports. The graphical user interface to FeaVer can help identify all data used. Support for this is admittedly crude if the command line interface to FeaVer is used instead, though the information available is more detailed. To see the global data objects that are imported by default, for instance, look at the definition of the structure `State` in the generated file pan.h. For local data, look for this information in a series of structure declarations (one for each proctype declaration that appears in the test harness) named **`P0 .. Pn`** in the same file **`pan.h`**.

The penalty for omitting to capture relevant state information can be that the check performed by FeaVer becomes incomplete, and that errors reported can be invalid. The penalty for capturing too much information is a loss of efficiency in the checking process. If too much irrelevant information is captured (say megabytes), it can also render the check itself infeasible, so some care in the selection of state information will be warranted.

Checking Assertions

All code that is part of a generated model will automatically be checked for a number of basic properties. There is support for three types of assertions: basic assertions, response assertions, and precedence assertions.

- A basic assertion has the form **`assert(expression)`**. FeaVer will check that the expression evaluates to true when the assertion statement is reached, in all feasible system executions.
- A response assertion has the form **`assert_r(expression)`**. For a response assertion FeaVer checks that the expression evaluates to true *within a finite number of steps after* the assertion statement is reached, again in all feasible system executions. The expression is required to become true in all system executions.
- A precedence assertion has the form **`assert_p(expression1, expression2)`**. For this type of assertion FeaVer checks that

`expression1` is true when this assertion is reached and remains true at least until **`expression2`** becomes true. The second expression, however, is not required to become true.

First among these is the basic assertions. For example, if the source code contains a statement

```
assert(x > y);
```

then the test will include a check for the existence of any system execution for which this assertion might fail. The model extractor will also insert a small number of basic assertions into the code. This happens for all array indexing operations (asserting that the index to be used will always be within the declared bounds of the array) and with pointer derefencing operations (asserting that the pointer used cannot be nill).

Causality

There are two additional types of assertions, not commonly found in regular C code. These two types of assertions can be used to verify some basic causality requirements on the system executions. A statement of the type

```
assert_r(x > y);
```

states the requirement that within a finite number of steps after this statement is executed the condition specified in the argument should always become true.

An example of the second type of causality assertion is:

```
assert_p(x < 10, x > y);
```

This assertion specifies that the first expression is true at the moment this assertion is executed, and necessarily will remain true at least until the second expression also becomes true (which itself is not required to also happen).

Timeline Properties

Longer patterns of events with consistency checks can be defined with the timeline editing tool that is part of the FeaVer graphical user interface. The use of this tool is described in detail in a separate section of this guide, see “Defining TimeLine Properties” on page 34.

Temporal Logic

FeaVer generates a model from the C code of an application that is checked for its logic properties by the model checker Spin. This means that any ω-regular property can be checked. Assertions and timeline property specifications cover only the most frequently needed types of requirements. Spin offers a range of other options for specifying correctness checking and for managing the complexity of large verification runs. See the references on page 63 for more details.

Checking Up on FeaVer

Especially when first starting to develop a test harness, it is good to double-check how FeaVer is interpreting the commands given. The simplest way of doing this is probably to run the commands from a shell-command line, one by one, and to look at the results that are generated. The first command called is always the model extractor `modex` (this is the program that first interprets the commands in the test harness file). You can call the model extractor manually for a test harness file `t.prx`, simply with the command: `modex t.prx`.

Preprocessing

The model extractor, in turn, starts by preprocessing each C-source file that is mentioned in the test harness in a `%F` command. It calls the standard C-preprocessor to do this. Since this is the first action that is executed, it is also the first thing that can fail. It can fail, for instance, if there are missing compiler directives, missing header files, syntax-errors, etc. The errors are often reported as appearing in a file with the extension `.I`, an intermediate version of the source file that is gener-

ated by the model extractor. As a rule, first make sure that you can compile the C-sources with the regular C compiler, before trying to extract models from them. The model extractor is a little more forgiving for certain types of omissions or errors, but it too has its limits of course.

The result of a successful preprocessing step is placed in a file with the extension `.M`, where future calls on the model extractor will look for it. (The intermediate version with extension `.I` disappears.)

Parsing Next, the model extractor will parse the preprocessed file to prepare for model extraction. Errors can be reported in this phase as well. The most common causes will be missing type declarations (caused by missing header files), syntax errors, etc. Missing information can be provided in `%D` segments. The extractor by default will try to ignore system header files, to avoid trapping lots of unnecessary data and declarations. In most cases this works well, but in some cases there are critical type declarations in the system header files that must be picked up. In those cases the model extractor can be forced to read and interpret all header files if it is called with option `-z`.

The Run Script If all is well, the model extractor will generate lots of files with portions of the extracted code. The main file among these is the file with extension `.drv` (driver). Another interesting files is the one with extension `.cln`, which contains a script to cleanup (remove) all files that were generated. The third interesting file has extension `.run`, and it contains a script that can be executed to perform the actual verification. Executing the run script will first cause another call on the preprocessor, this time to process the generated Promela code. This will expand the contents of the .drv file, and everything it refers to, into a single file named `model`. This is the best place to look to figure out what the model extractor actually generated from the test harness and the source files. This step is unlikely to generate errors, but in principle, if bad code is generated, it is possible.

Spin Next, the model checker `spin` is called with file `model` as input. Clearly, `spin` too can generate error reports if the code it finds is not to its liking. Causes could be bad entries in the lookup tables, missing declarations, missing `Import` commands in the test harness, etc. etc. Spin now generates the model checking code in a set of C source files named `pan.[chmbt]`.

C-compiler The pan.c file is now compiled with the standard C compiler. Again, errors can now reveal themselves if, for instance, the encapsulated C code is improper, if there are missing declarations etc. etc. If you are familiar with the code in the pan files, looking at the data structures that are generated for each process and for the global state vector is sometimes helpful in tracking down the cause of errors. Rest assured that in virtually all cases it comes down to omissions or mis-statements in the test harness, so ultimately this is where the errors have to be repaired.

Execution If compilation succeeds, we are not quite out of the woods yet. Also the execution of the model checking code can fail, with memory errors or core dumps. Bad code from the original source files, or bad replacement code from the lookup tables, will be the cause, which can be tracked down with standard debugging techniques.

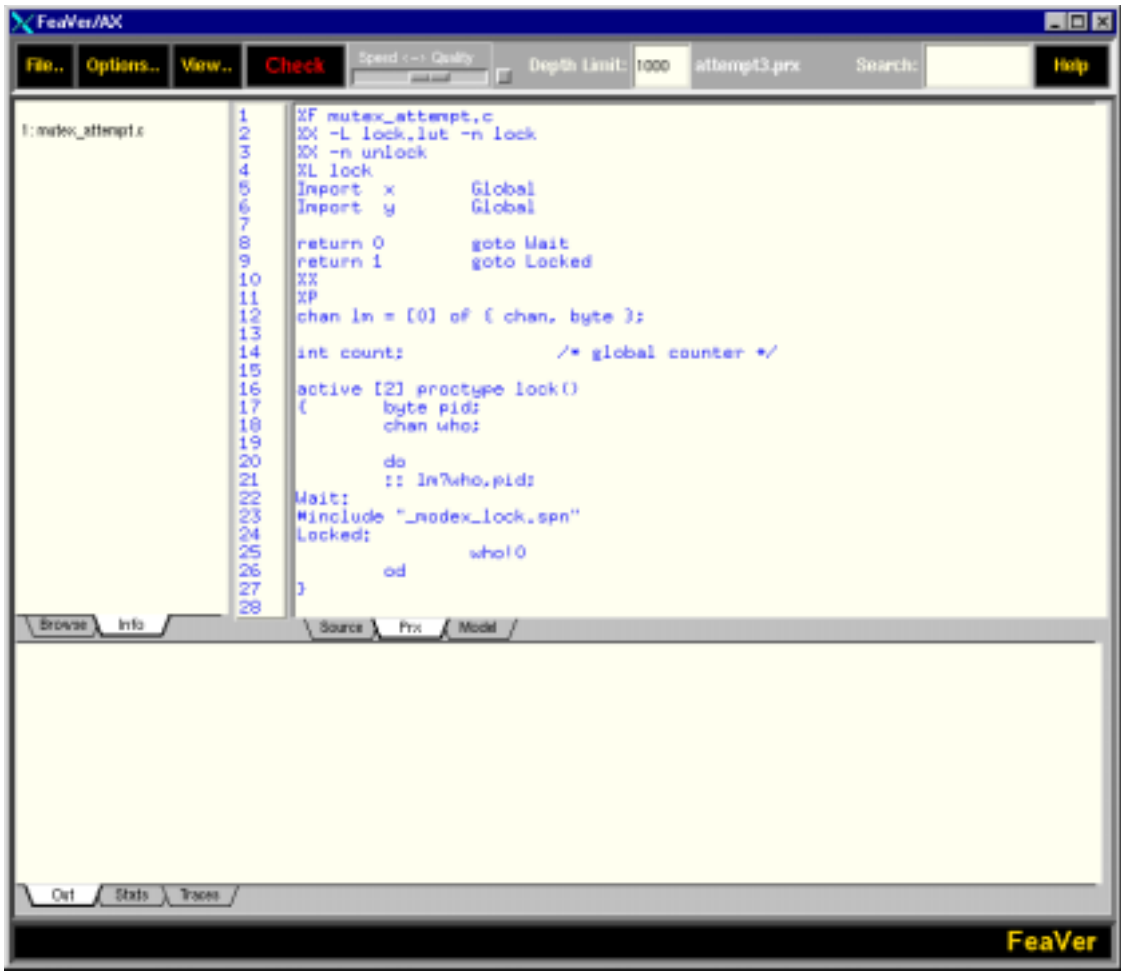


FIGURE 11. FeaVer Graphical User Interface

3. Graphical User Interface

So far we have only used the command line interface to FeaVer in the examples. This establishes basic access to the functionality of the tool, but not a very sophisticated one. A more attractive alternative is to use FeaVer's graphical user interface, with the shorter name: **`feaver`**. The graphical user interface is written in Tcl/Tk and it runs on both Unix and Windows platforms. To function properly, the tool requires access to a small number of background commands, that must be installed first. These commands are:

- An ANSI compatible standard C compiler.
- FeaVer's model extraction engine tool, named modex.
- The model checker Spin.

The basic view of the graphical user interface, displaying the last test harness from the mutual exclusion example, is shown in Figure 11 on page 26.

The tool displays three main panels: two large panels side by side at the top of the screen, and one smaller panel along the bottom of the screen. Each of these panels has tabs that allows a selection of different views.

The panel at the upper-left has two tabs. The first tab, named **`Browse`**, brings up a directory browser, for loading a test harness file (file extension **`.prx`**), an associated timeline property specification (file extension **`.vip`**), or a plain Promela model (file extension **`.pml`**). The second tab, named **`Info`**, displays general information with a listing of the target source files mentioned in a test harness, and where the progress of checks is indicated.

When a test harness file is first opened, the display is as shown in Figure 11 on page 26. The **`Info`** tab on the left hand side shows all C source files that are mentioned in **`%F`** commands. By clicking on one of those files, they can be displayed in the `Source` view on the right hand side.

There are three main tabs in the upper right-hand side panel, named **`Source`**, **`Prx`**, and **`Model`**. Additional tabs can appear on special selections (e.g., when a timeline property specification is loaded). The **`Source`** tab can show the C source text of a target C source file from an **`%F`** command in the test harness. The **`Prx`** tab shows the test harness, and the **`Model`** tab shows the exact verification model that FeaVer constructs based on the instructions in the test harness. FeaVer invokes background tools to generate these views. Any error messages that might be generated in this process will appear in one of the three displays at the bottom of the screen.

Menus

The **`File menu`** allows changes made in the test harness or in the target source files to be saved before a check is initiated. It has four entries:

- **`Open`**
- **`Save Src`**
- **`Save Prx`**
- **`Quit`**

Remember that source and test harness (**`.prx`**) files must always be saved before any changes in these files is picked up in a new run of the check.

The **`Options menu`** contains four main groups of options. The first group is:

- Search for accept cycles
- Search for non-progress cycles
- Search for Property violations
- Search for invalid endstates

By default, a check with a test harness will only check for assertion violations, array indexing errors, and nill pointer referencing errors. If any other properties need checking, this can be specified in the test harness in the `%G` segment (see Table 3 on page 48), or the settings can be selected manually from the `Options` menu without being recorded in the test harness file.

Accept Cycles

A search for accept cycles only makes sense if at least one labelname in a source file, or in one of the test drivers, starts with the prefix “`accept`”. The model checker

will search for any potentially infinite execution that passes through such a label. Normally, the accept-labels, however, are not user defined, but generated by the FeaVer system as part of the definition of a timeline property. So, if no timeline properties are used, this first option is not of much interest.

Non-Progress Cycles

The second option tells FeaVer that we're interested in finding infinite cycles that do not pass through any label with the prefix "`progress`". Again, if no progress labels appear in any of the source files or test driver specifications, this search will be of limited use.

Property Violations

A search for property violations can be initiated if first an automaton was generated with the timeline editor. The last automaton generated will be included in the test system and verified if this check is enabled.

Invalid Endstates

The last option from this group enables a search for invalid endstates. This means that the model checker is asked to report states of the system where no further processing is possible, but where not all processes have either reached the end of their code, or are stopped at a statement labeled with a labelname that begins with the three characters "**end**". An example of an invalid endstate is a system deadlock: an unforeseen termination of the system of concurrent processes.

The next two options can modify how error traces are reported by the system in message sequence charts. The options are:

- **`Show all statements in MSCs`**
- **`Show real i/o values in MSCs`**

The default is that the message sequence charts contain only message passing operations, provided that they are defined with Promela primitives. By enabling the display of *all* statements the MSCs are made more detailed, including all statements executed. Secondly, in message passing operations normally the symbolic names for message parameters are displayed. By enabling the display of *real* values, the numeric equivalents will be used instead.

The third group of options supports the creation and the maintenance of mapping tables.

- **`Edit LUT..`**
- **`Populate LUT..`**
- **`Upgrade LUT..`**
- **`Get Pan Info..`**

The first option in this group allows one to view the mapping tables that are defined in a test harness and to modify them if needed. The same can also be done by directly editing the corresponding entries in the test harness file. The separate display for a mapping table (called LUT for lookup table here) is sometimes a more convenient alternative. When the option **`Edit LUT..`** is selected, an extra button will appear on the top menu bar, as illustrated in Figure 12.

By double-clicking one of the entries from the LUT menu, the LUT editor for that table is displayed, as illustrated in Figure 13. The table displays two columns. On the left hand side are the basic source text statements defined in the mapping table, and on the right hand side are the corresponding mappings that have been defined. The right hand side entries are color coded: a yellow color means that the entry was defined manually by the user, a white color means that the entry is the



FIGURE 12. **LUT Menu with Mapping Tables from Test Harness**

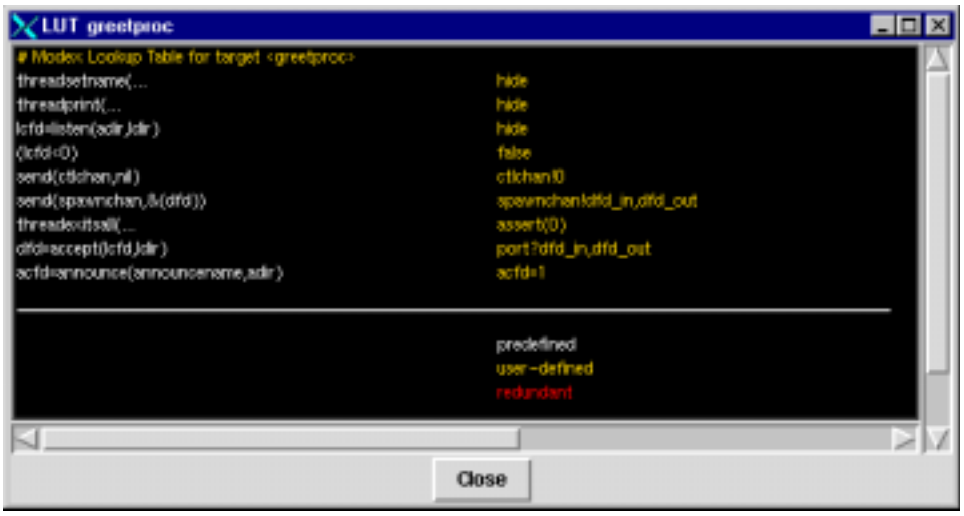


FIGURE 13. **Mapping Table Editor**

default conversion for the source statement, and a red color means that the entry is redundant (i.e., the source statement cannot be found in the target procedure). Any entry in the table can be modified by double-clicking the right hand side entry. This will bring up an edit box for the entry selected. If the change is then saved (by selecting the **`OK`** instead of the **`Cancel`** button) the mapping table will be updated both in the mapping table editor and in the displayed vesion of the prx file.

The **`Populate LUT..`** option from the Options Menu prompts the user for the name of a mapping table (remember that there can be multiple mapping tables in any given test harness definition). When selected, the table as specified in the test harness is completed automatically with all default entries. To use this option when no mapping table yet exists, first create an empty table with the syntax:

```
%L newtable
%%
```

and select its name from the menu prompted by the **`Populate LUT`** option. The table will now be filled in automatically. The same procedure can be used if the mapping table has partially been defined. It is, for instance, useful to double-check that the left hand side entries are accurately spelled (meaning that no alternative entries would be added to the table when this option is invoked).

The **`Upgrade LUT..`** option prompts the user with a menu choice of the source files that has been declared inside the test harness with **`%F`** commands. Select the source file that was modified. Define a new name for the test harness in which the modified version will be saved. It is wise to maintain a separate version of the test harness for each major version of the source code, rather than to maintain a single version that is changed along with the sources. When the **`OK`** button is now pressed, the new test harness file is created and a menu button will be generated for each mapping table that incurred changes because of the source file modification. The buttons will bring up the mapping table editor, which can now be used to study the changes and to adjust the mappings where necessary. When all changes have been made, save the new version of the test harness file with the main File Menu. If multiple source files were modified, select one at a time and repeat the above update procedure one file at a time, creating multiple intermediate test harness files: one for each file that changed.

The **`Get PanInfo`**.. option extracts information about all global and local data objects that are part of the model from the model checking code that is generated. This option requires that all steps in the model extraction process can be completed that are necessary for the generation of this information. The option serves as an overview of all data that will be considered state-information during the model checking process. Check if no data objects are accidentily omitted, and if no irrelevant data objects are accidentily included. If an "**`Import _all_`** ..." command is used in one or more of the mapping tables, the number of data objects can easily become too large. The overview generated here can help the user to replace the indiscriminate Import command with a series of more selective ones, leaving out the irrelevant data objects, to reduce memory consumption in the model checking process.

Creating Mapping Tables

Once a mapping table is defined, the use of FeaVer is fairly straightforward. But the initial creation of a set of mapping tables for a new application can be somewhat daunting, especially on a first attempt. One relatively mild problem is to find out which entries may be used to populate the left-hand side of the mapping table. FeaVer uses a canonical form for these entries, which may be hard to guess. To simplify this process the model extractor can generate the default mapping table for an application, showing precisely all the entries that can appear on the left-hand side of the table, with the default translation on the right. From the graphical user interface this is done by chosing the option **`Populate LUT`**. The fully populated table is inserted into the test harness, for the user to peruse. The recommended action at this step is to omit everything from the table that will not need any overrides from the user, and to keep only those few entries that must be handled differently. Typically this is only a handful of entries that are mapped to **`hide`**, **`true`**, **`false`**, **`skip`** (see Table 5 on page 51), or that are converted into Promela syntax.

When the source text of the application changes, the mapping tables may have to be adjusted. To facilitate this process, the **`Upgrade LUT`** option will prompt for the names of the old and new versions of a C source file and show which entries in the existing mapping table have become redundant (indicating code that disappeared) and which new default entries were added (indicating code that was revised or

added). With this feedback it is usually simple to adjust the mapping table for the new version of the code.

Control State Information

The last option in this group, **`Get Pan Info`**, can be used to retrieve information from the generated test system about all state information. It can be used to determine if any control information is missing, or if any redundant information was accidentily included.

View Menu

The **`View`** menu allows various types of additional information related to the test harness to be displayed. The entries in this menu are:

- Scrollbars
- Parsed Code
- Functions Used
- Data Types Used
- Test Driver Code

The first option from this group can be used to add scrollbars to all the display panels. Without the scrollbars (the default configuration) one can pan and scroll in every window with the *middle mouse button* down. It leaves more room for the display itself, compared to the display with scrollbars enabled.

The second option can enable an additional tab in the upper right-hand side display to show the source code as it appears after the preprocessing phase has been completed. The third option similarly can add a tab with a listing of all function calls that appear in the target source code, and the fourth option does the same for all data types that are used. The fifth and last option in this group allows the display of the test driver code as it is generated from the test harness by the background model checker. From this driver code the standard C preprocessor will generate the actual verification model for the test system. The latter is displayed in the permanent tab labeled Model. For debugging, it can sometimes be useful to see also the intermediate step. This option should only rarely be needed.

The Check Button

Clicking the **`Check`** button will start the checking process. In the beginning, when a new test harness is being developed, the checking process will likely halt with error messages from the **`Modex`** model extractor, from **`Spin`**, or from the C compiler. All errors are displayed in one of the three output displays at the bottom of the screen.

Once the check goes through, the display looks as illustrated in Figure 14 on page 32. The first ten errors found in the code are placed in the top menu bar, in a menu labeled **`Traces`**. The first of these errors is also generated as a message sequence chart of the type shown in Figure 15 on page 33. The same error sequence is also shown in more verbose form in the **`Stats`** display at the bottom of the screen. Line numbers in the `Stats` display are linked to line numbers under the **`Model`** tab. By clicking these lines with the mouse, the corresponding line in the **`Model`** tab will be displayed.

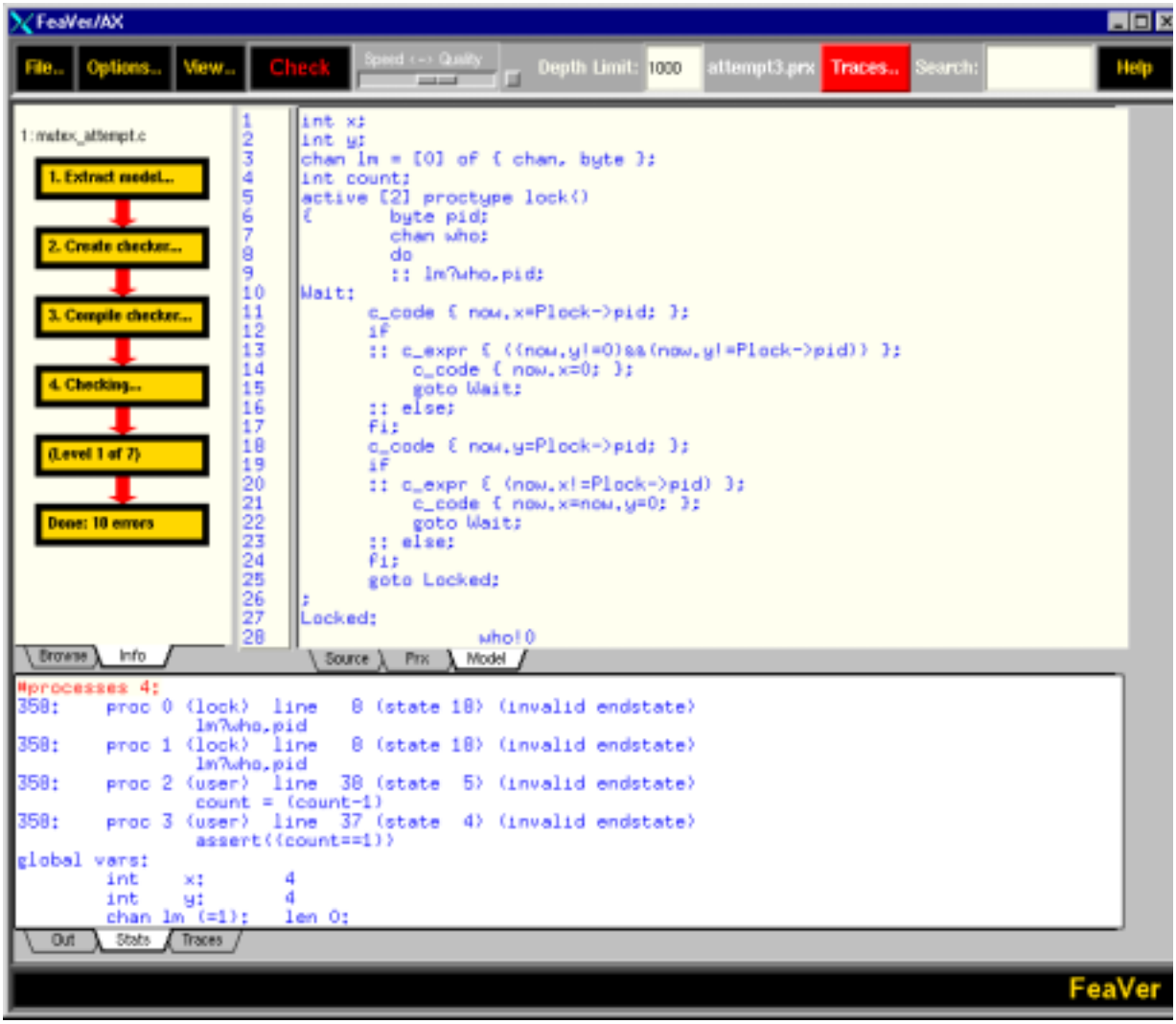


FIGURE 14. FeaVer Display at the End of a Successful Search for Errors

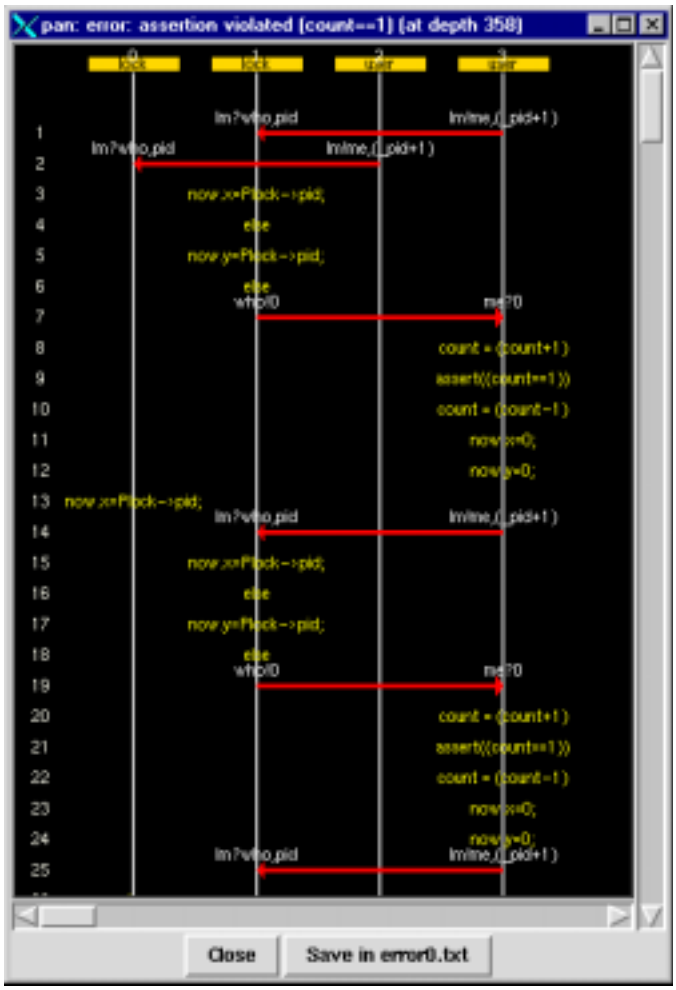


FIGURE 15. **Message Sequence Chart for an Error**

4. Defining TimeLine Properties

Many properties of interest can be embedded as assertions within the source text of an application. For more complex properties FeaVer offers an intuitive visual tool for defining properties as timelines. The model checker that is used by FeaVer to perform system checks can check more complex types of conditions than can be expressed with assertions or timelines alone. For advanced use, for instance, properties may be expressed in linear temporal logic formulae, or still more generally as omega automata. This much expressive power, though, is rarely needed in practice. We restrict ourselves here to the more common cases.

Overview

The timeline editor is a visual tool for expressing properties of concurrent systems. A property is defined as a relation on a set of potentially interesting system events, and zero or more constraints. To check if the system satisfies a timeline property, a timeline specification is first translated into a *test automaton* and then checked against the system code. The checker uses the test automaton to uncover any feasible system execution that can violate the stated property.

In the FeaVer system the system model is extracted directly from the ANSI C source code of the system. Error traces produced by FeaVer/Spin, are execution traces through the source code that lead to the error.

A timeline specification, like the one shown in Figure 16, has a wide horizontal bar close to the top of the drawing canvas. Time progresses along the bar from left to right. Along the timeline bar there are vertical bars. These are *marks* which are used to record the relevant system execution steps. The marks do not represent clock ticks, since there is no reliable notion of a global clock in a system of asynchronously executing concurrent processes. Marks are the places where events of interest to a system property can be attached.

There are three types of **events**:

- **regular events**, denoted by a down arrow labeled with an **e**. These are *possible* events, typically input from users or external entities, that need not appear in every feasible system execution, but that make the property relevant if they do occur. Examples, for telephony features: onhook, offhook, and the arrival of an incoming call.
- **required events**, denoted by a down arrow labeled with an **r**. These are events that are *required* to occur in response to previous regular events, provided that all applicable constraints are also satisfied. It is an error if required events are not detected.
- **fail events**, denoted by a down arrow labeled with an **f**. These are events that should not occur in response to previous regular and required events. It is an error if fail events are detected.

A timeline can also specify **constraints**.

Constraints persist over certain intervals of the timeline. For instance, if there is a telephone switch requirement that the system must respond to an offhook by providing dialtone, a constraint: ‘in the absence of onhook events’ can be defined for the interval between when the offhook occurs and when dialtone is provided, otherwise, many executions of no interest would be included in the check. A constraint, denoted as a horizontal line placed below the main timeline, starts and ends at a timeline mark. The constraint can optionally include or exclude the begin

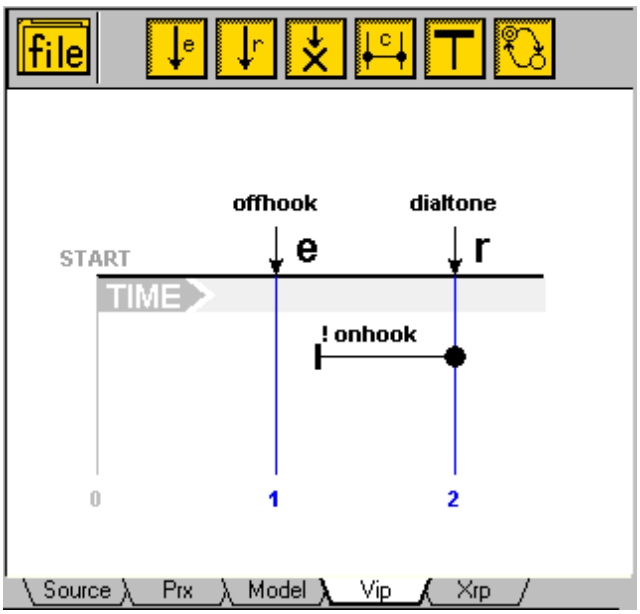


FIGURE 16. **A sample timeline specification**

or end mark to which it is attached. The consequences of including or excluding endpoints will be discussed in more detail.

Marks are shown as small blue vertical lines to which event names are attached. Each mark can only represent one single event. Nothing is implied about the amount of time that can pass between two subsequent marks on a timeline. If event **e2** immediately follows event **e1** on the timeline, for instance, we cannot conclude anything about the amount of time that would pass between these two event occurrences. We can only conclude that there are no other events of interest to the property that should occur between **e1** and **e2**. There may of course well be other event occurrences in this interval that are not of interest to the property that is specified. This is depicted in Figure 17.

Timeedit can display a test automaton for the property either in graphical or textual form. The graphical version is useful for visual inspection only; the textual version can be used by the model checker the performs the actual system test.

In the remainder of this chapter we discuss the main options that are available on the timeline editor graphical user interface.

The Timeline (Vip) Tab

Select the VIP tab in the upper right-hand panel on the FeaVer interface. The VIP toolbox and a blank timeline will appear.

File Operations

Creating a new vip file. From the Vip tab, select File → New Vip. A blank timeline will appear.

Opening a file. From the VIP tab, select File → Open Vip.

Saving a file. From the Vip tab, select File → Save Vip to overwrite the current file.

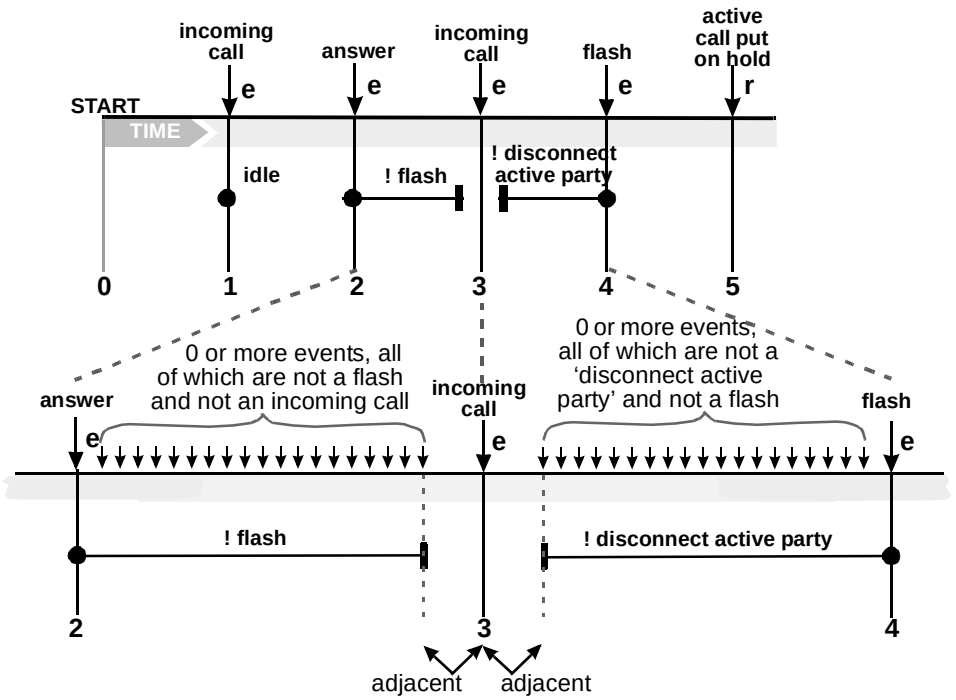


FIGURE 17. Timeline Details

Saving under a new name. From the Vip tab, select File → Save Vip As. A file browser will appear. Type or navigate to the desired directory. Select an existing file, or type a new filename, then press the *Save* button.

Creating an image of the timeline. The timeline tool can save a graphical version of the timeline in a file, either in postscript or GIF format. To print the graphical version to a printer, you can import the '.gif' file into a graphical editor (i.e. PowerPoint FrameMaker) or open the file using a graphical editor (i.e. PhotoShop) and use the device printing facility of one of those applications. To save a GIF or postscript version of the timeline, select:

File → Save Vip As

To save a postscript version of the timeline, select postscript from the 'Save As Type' drop down menu. Select an existing postscript file or specify a new filename.

To save a GIF version of the timeline, select GIF from the "Save As Type" drop down menu. Select an existing GIF file or specify a new filename.

Operations on Events and Constraints

Adding a new event. Select 

for a regular event or 

for a required event. Select the mark to which the event should be attached, and the event window, shown in Figure 21, will open. To name the event, select a predefined event from the list and press the *Apply* button, or to define a new event, press the *Define Symbols* button to open the Symbol editor, then enter the new event name in the



FIGURE 18. **Edit Event window**

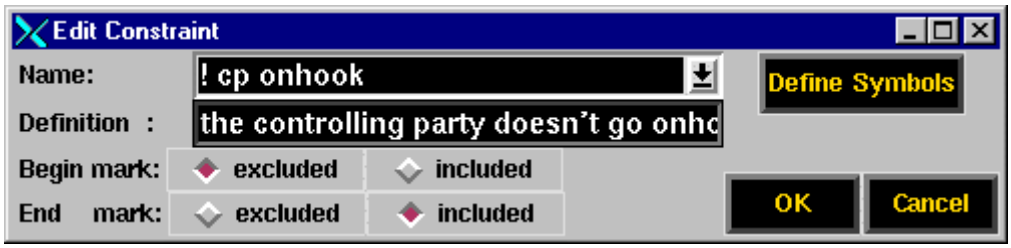


FIGURE 19. **Edit Constraint window**

Event field, and a definition in the **Definition** field, then press the *Apply* button. There must be an empty mark available in order to add a new event.

Adding a new constraint. Select the *constraint* button . Using the select button of your mouse, select and hold down the button over the initial mark. Move the mouse to the end mark and release, and the constraint window will open. To name the constraint, select a predefined constraint from the list and press the *Apply* button, or to define a new constraint, press the *Define Symbols* button to open the Symbol editor, then enter the new constraint name in the **Constraint** field, and a definition in the **Definition** field, then press the *Apply* button.

Moving an event or constraint. Select the event or constraint to be moved. With the select mouse button depressed, move the event to an empty mark and release. An event may not be attached to the Start mark. A constraint can be moved left and right or up and down.

Deleting an event or constraint. Select the event or constraint to be deleted. Click the menu mouse button. Select *delete* from the popup menu then release the menu mouse button. Note: there is no undo.

Editing event or constraint attributes. Select the item to be modified. With the cursor positioned over the selected item, depress the menu mouse button. Select *edit* from the popup menu. The *Edit* window will appear (Figures 18 and 19). Change the name by scrolling the drop down list (for more detailed information see under 'Linking Events and Constraints to Code" below). To define a new name, select *Define Symbols*. For events, change the type of event (**e**, **r** or **f**) by selecting the radio box associated with the desired type. For constraints, include or exclude the begin or end mark by selecting the appropriate radio box. Save changes by selecting *OK*.

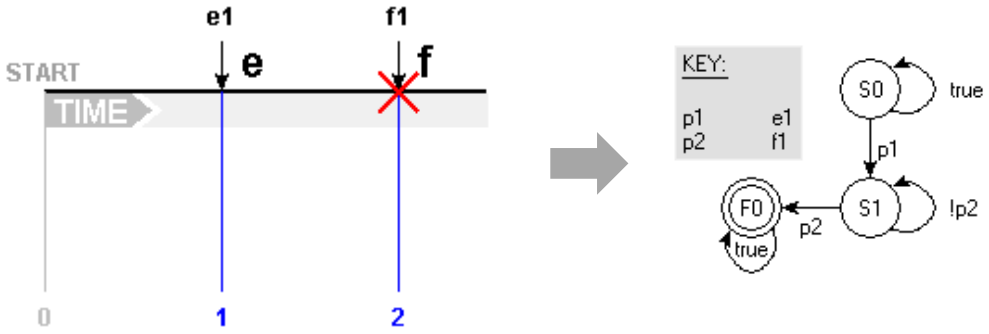


FIGURE 20. Timeline with fail event and associated automaton

Creating a fail event. Select the fail event button . Select the regular or required event to be converted to a fail event. Alternatively, convert a regular or required event to a fail event by selecting the *fail* radiobutton on the *Edit Event* window.

An example timeline with a fail event and its associated automaton are shown in Figure 20.

Defining Symbols

The instructions in this section apply to the Constraint Symbol Editor and the Event Symbol Editor.

Invoking the Symbol Definition Window: The Symbol Definition Window, shown in Figure 21, can be invoked using the *Define Symbols* button (see Figures 18 and 19) of the Edit Event/Constraint Window. Alternatively, the Symbol Definition Window can be invoked from the FeaVer Edit menu, via Edit ⟶ Edit Event Symbols or Edit ⟶ Edit Constraint Symbols.

The Symbol Definition Window displays the Symbols on the left and the Symbol definitions on the right.

Populating Symbols: The Symbols and Symbol Definitions in the Symbol Editor are populated automatically when a test harness file (ends with extension ".prx") is opened. All symbols previously loaded in the Symbol Editor will be deleted when a test harness file is opened.

Symbols may be added to the table when a new Vip file is opened. Any unique symbols in the Vip specification will be added to the Symbol Editor. Since symbol definitions are not stored in Vip files, any symbols added to the symbol Editor when a Vip file is opened will lack a definition.

Edit Operations:

Modifying a symbol: Selecting a symbol in the left list will cause the symbol name and definition to be displayed in the edit windows at the bottom of the Symbol Editor. The symbol and it's definition can be modified. If the symbol is modified, and the Set button is pressed, a new symbol will be added to the list. If only the definition is modified, and the Set button is pressed, the definition of the original symbol will be modified.

Deleting a symbol: Select the symbol to be deleted, then enter Ctrl-x.

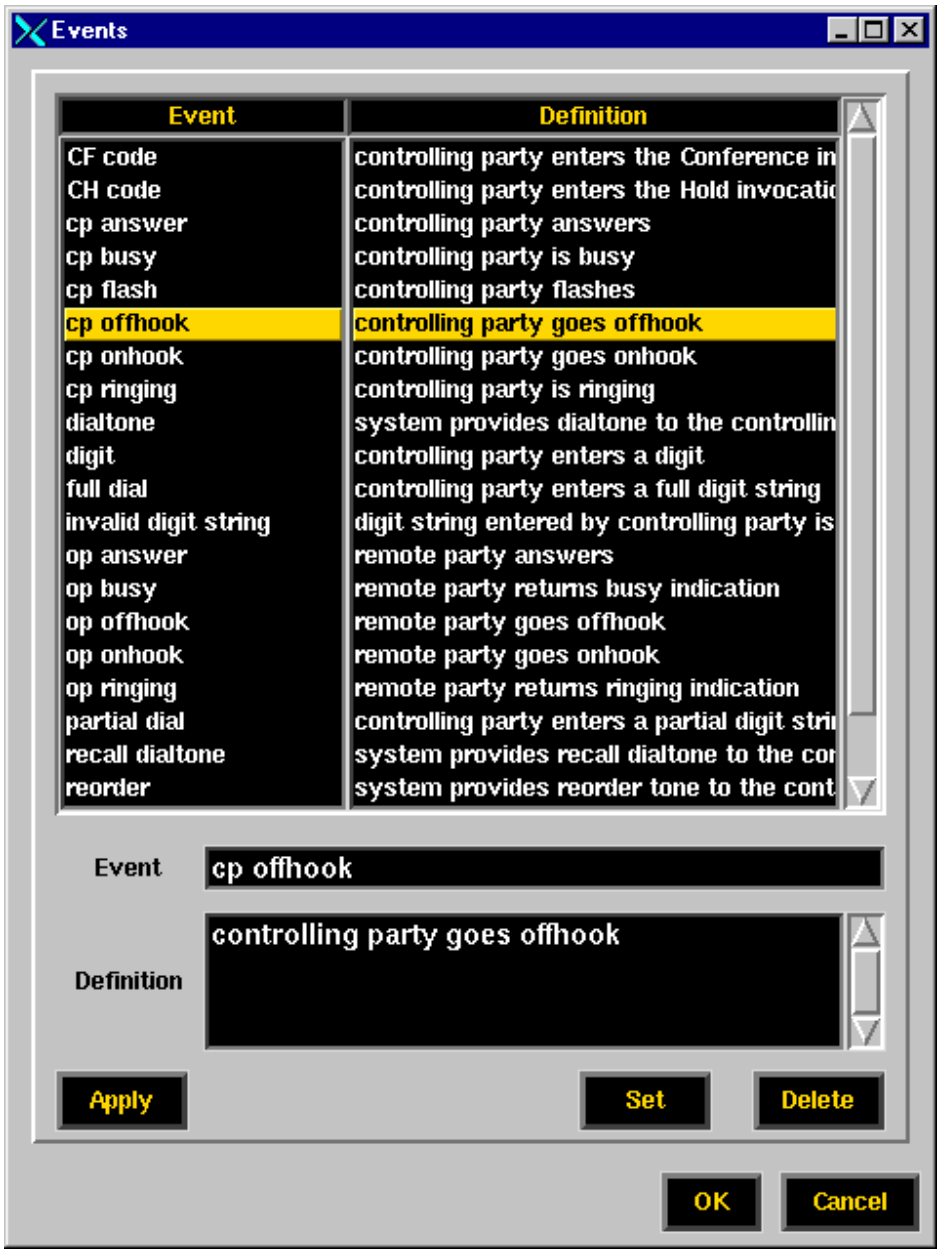


FIGURE 21. Adding an Event to a Timeline mark

Copying a symbol: Select the symbol to be deleted, enter Ctrl-c, then enter Ctrl-v.

Saving Changes: Changes made to the Symbols are temporary until the OK button is pressed. Pressing OK, causes the modifications to be displayed in the current test harness window. Pressing Cancel discards all modifications. Changes to the test harness window (such as modifications to symbols) must be explicitly saved to the test harness file (via the OK button) in order to persist between FeaVer sessions.

Linking Events and Constraints to Code

In the Edit window for events and constraints, you can enter a precise definition for each named event and constraint. In the remainder of this section we will talk primarily about event definitions. Everything we say applies equally to constraint definitions.

The Definition provided in the Edit window establishes the link between the informal notion of an ‘event’ and the precise condition that has to be fulfilled in the executing system to declare that the event took place. The condition can, for instance, be defined as a variable having a particular value, a message having been appended to a message queue, a specific program location being reached, etc. The precise definition can depend on the details of the event occurrence. We will consider three cases. The first case is likely to be used most frequently. The second case offers the largest form of expressiveness., and the third the smallest.

- The event occurrence can be derived from the values of data objects that are declared within the *C implementation* itself. In this case, first make sure that all data objects referenced in the expression are *tracked* as state variables in the test harness in %L segments. Next, add a special purpose C function in the %C segment of the test harness to test for the occurrence of the event. The function should return 1 when the event is detected, and 0 otherwise. The function should *not* modify *any* data objects when called: it must be side-effect free. If the function is called ev1(), then add the following entry into the definitions field of the edit box for this event. (Note carefully that there are no semi-colons, and no arguments to the function.)

  ```
  c_expr { ev1() }
  ```

  We cannot easily refer to program locations in this case.

- The event occurrence can be defined over *Promela data objects* that are defined in the %P segment of the test harness. For instance, if the event is defined as the sending of the message Msg to the Promela message channel q while a global Promela variable a has a value greater than 10, then it could be detected with the side-effect free expression q?[Msg] && a > 10. This expression can be typed directly into the definitions field of the edit box for the event. (Note that no c_expr wrapper is used in this case.)

  ```
  q?[Msg] && a > 10
  ```

  If variable a is declared as a local data object in proctype y, and there is only one process instantiation for proctype y (the most common case), then the definition is written:

  ```
  q?[Msg] && y:a > 10
  ```

  If, finally, there can be multiple process instantiations of proctype y, and we want to refer to the instantiation with process number 4, the definition would be written:

  ```
  q?[Msg] && y[4]:a > 10
  ```

  A reference to a program location can also be included in expressions when they are defined in this way. If proctype y includes a model that is extracted from a C function, and a label named L appears anywhere within that C function, the expression

  `y@L` respectively `y[4]@L`

  will be true if and only if the process reaches the program location indicated by the label L.

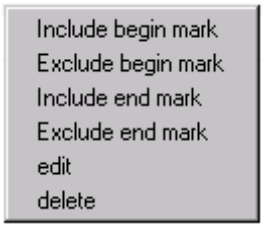


FIGURE 22. Setting Constraint Options

- A third, slightly more difficult, case is to handle expressions that contain both references to data objects declared in the C program and data objects declared in a Promela segment of the test harness. In this case we will be restricted to referring to only globally declared Promela data objects, and we also cannot easily refer to program locations. If an important data object was declared locally, then we can promote that object to a global, to make it visible as an event that must be defined in this way. Next, we can define a C function (as in the first example) and refer to the global Promela objects with the variable prefix “now.”. A reference to the global Promela variable x, for instance, would be written as “now.x” in the C function.

In principle it would be possible to refer to locally declared Promela objects also with the first and third method, but the required syntax is obscure and therefore not discussed here. (Note carefully that a possible reference to the variable a declared locally in proctype y *cannot* be written as `Py->a` in this context.

Including or excluding constraint begin and end marks. Including and excluding end points of a constraint determines whether the constraint applies at those points in the timeline. These settings can be changed via the Edit Constraint window as shown above.

To include the begin mark: select the constraint, open the *Edit* window, and select the *Begin mark - included* radiobutton, then select *Ok.* Alternatively, select the constraint then invoke the popup menu using the mouse menu button. Select *include begin mark* from the popup menu. The left side of the constraint will have a filled black circle ● and will overlap the begin mark, to indicate that the constraint includes the begin mark. An included begin mark means that the constraint specified must hold at this execution step of the begin mark. If the event offhook is assigned to the begin mark, then the constraint must hold at the execution step where offhook occurs.

To exclude the begin mark: select the constraint, open the *Edit* window, and select the *Begin mark - excluded* radiobutton, then select *Ok.* Alternatively, select the constraint then invoke the popup menu using the mouse menu button. Select *exclude begin mark* from the popup menu. The left side of the constraint will have a vertical black line ▌ that will be offset to the right of the begin mark, to indicate that the constraint does not include the begin mark. An excluded begin mark means that the constraint does not have to hold at this execution step. However, the constraint must hold at the very next execution step.

To include the end mark: select the constraint, open the *Edit* window, and select the *End mark - included* radiobutton, then select *Ok.* Alternatively, select the constraint then invoke the popup menu using the mouse menu button. Select *include end mark* from the popup menu. The right side of the constraint will have a filled

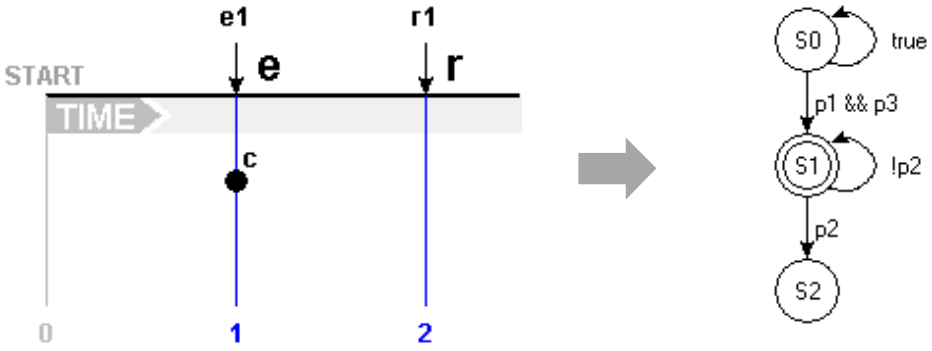


FIGURE 23. constraint confined to first mark, and its resulting automaton

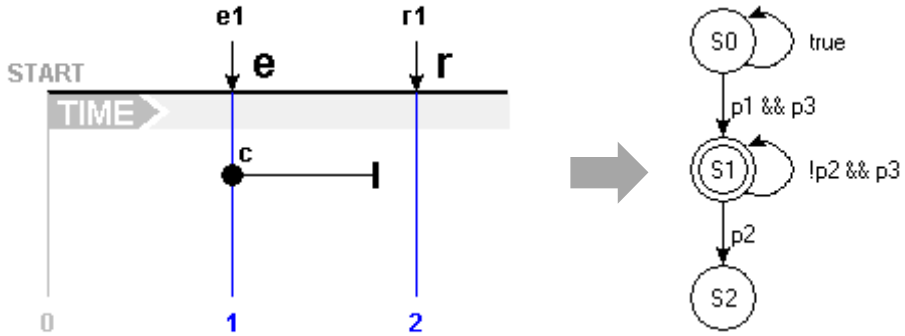


FIGURE 24. timeline with constraint with included begin mark and excluded end mark

black circle ● and will overlap the end mark, to indicate that the constraint includes the end mark. An included end mark means that the constraint must hold at this execution step.

To exclude the end mark: select the constraint, open the *Edit* window, and select the *End mark - excluded* radiobutton, then select *Ok.* Alternatively, select the constraint then invoke the popup menu using the mouse menu button. Select *exclude end mark* from the popup menu. The right side of the constraint will have a vertical black line ▌ that will be offset to the left of the end mark, to indicate that the constraint does not include the end mark. This means that the constraint does not have to hold at this execution step. However, the constraint must hold until the step just before this execution step.

The effect of including and excluding begin and end points of constraints on the automaton generated by timeedit is shown in the examples that follow.

Constraint applies at first mark only: In Figure 23 constraint **c1** applies at **e1** only.

Begin mark included and end mark excluded: In Figure 24, constraint **c1** applies at **e1** and **c1** applies from the step after **e1** is detected up until the step preceding **r1**.

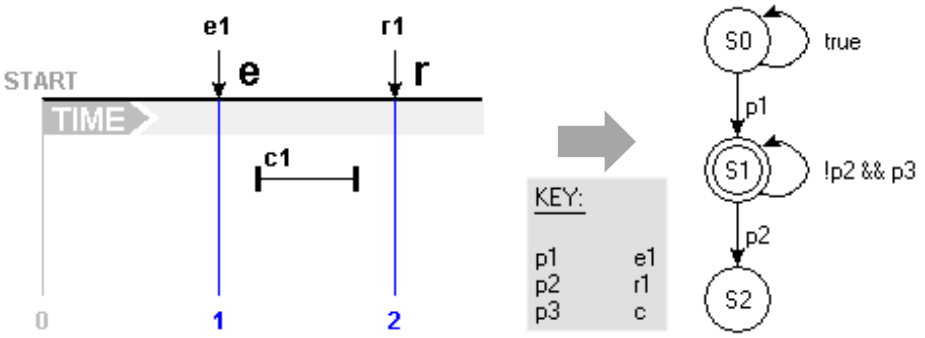


FIGURE 25. Timeline including constraint with excluded begin and end marks, and its resulting automaton

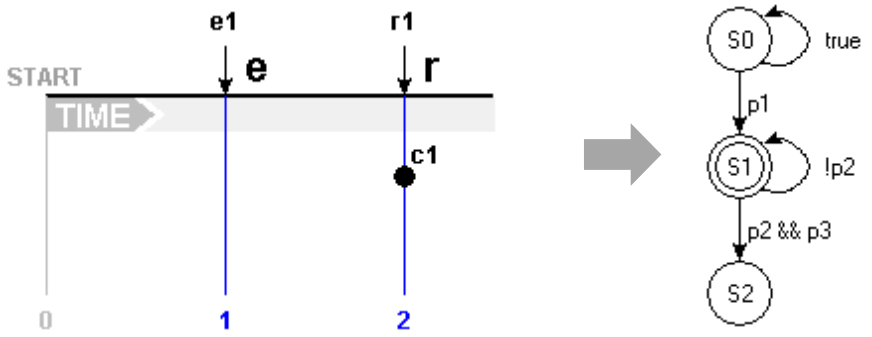


FIGURE 26. Timeline with constraint that applies only at the last mark

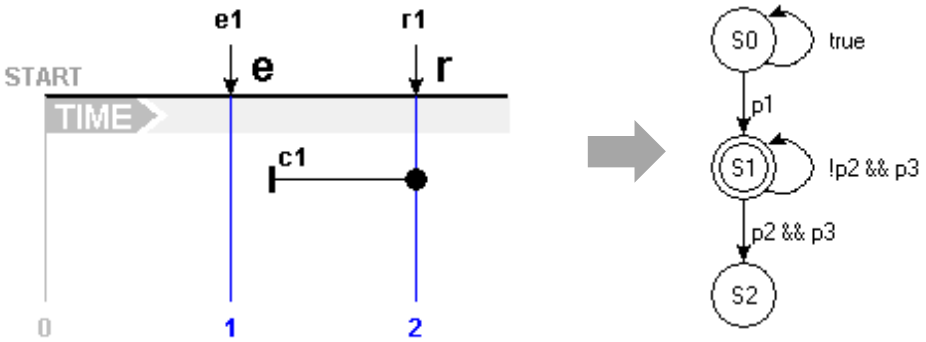


FIGURE 27. Timeline with constraint that excludes the begin mark and includes the end mark

Both begin and end mark excluded: In Figure 25, constraint **c1** does not apply at the step **e1** and it does not apply at the step **r1,** but it does apply from the step after **e1** is detected up until the step preceding **r1**.

Constraint applies at last mark only: In Figure 26, constraint **c1** applies only at **r1,** which is the end mark for the timeline.

Begin mark excluded and end mark included: In the Figure 27, constraint **c1** does not apply at **e1**, it does apply from the event following **e1** until and including detection of **r1.**

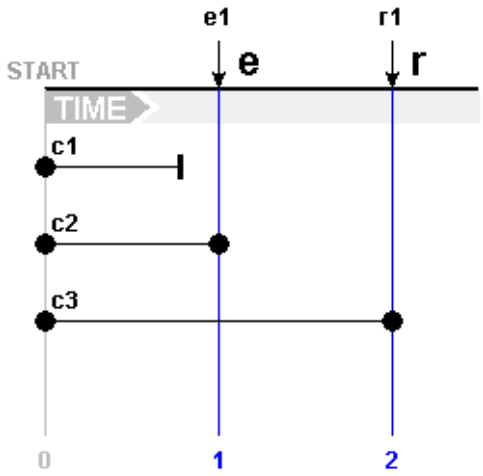


FIGURE 28. **Three forms of constraints that may precede the first event**

Constraints that precede the first event Three forms of constraints that may precede the first event on a timeline are shown in Figure 28. A constraint that precedes the first event specifies a condition that must hold from the beginning of the execution. For instance, if constraint **c1** is defined as **!e1,** then the check starts at the first **e1** in the execution sequence. Without this constraint the check may start at subsequent occurrences of **e1**, that is, one or more occurrences of **e1** may be ignored before the check starts.

Operations on Time Marks

Adding a mark. To add a new mark on the timeline, select . Position the cursor anywhere along the horizontal bar of the timeline where the new mark is desired. Depress the select mouse button to insert the new mark. Existing marks to the right of the inserted mark will be moved to the right to make room.

Deleting a mark. To delete a mark, position the cursor over the vertical line of the mark to be deleted. Depress the mouse menu button and select *delete mark* from the popup menu. Any event assigned to the mark will be deleted. Constraints assigned to or spanning the deleted mark will be adjusted.

Generating automata

To view the equivalent correctness claim (a test automaton) for a timeline specification, select .

A view-only graphical version of the automaton will be displayed in the graphical automaton window. To view the never claim version of the Vip specification, select the *See Text* button on the graphical automaton window. This button does not save the automaton to a file, but is simply for viewing the graphical automata and the never claim version.

To save the never claim version of the Vip specification to a file, select:

File ⟶ Generate Automata

The file save dialog will appear. Select or type a filename, using the file extension “aut”, then press the Save button. Another way to save the never claim version of the Vip specification to a file is via File ⟶ SaveAs. Select “Automata Files (*.aut)” from the Save as type pull-down menu.

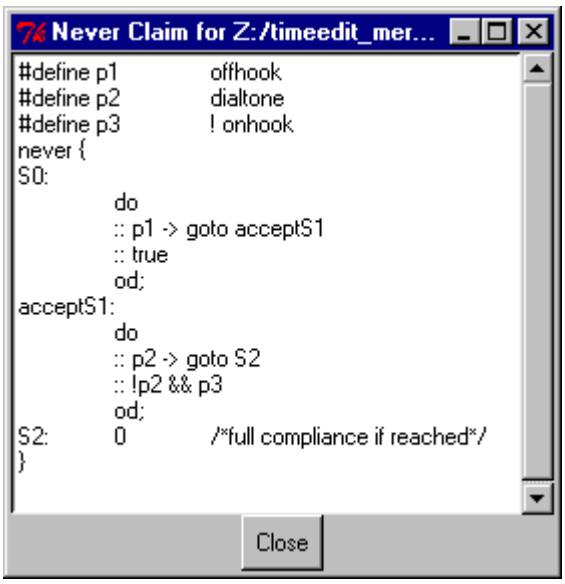


FIGURE 29. Textual Version of Test Automaton

The graphical automaton cannot be saved to a file. A generated test automaton always reflects the current contents of the timeline display, which may differ from the last saved version of the timeline.

Credits

Gif generation is provided by *fly*. *fly* is copyright 1995-1999, Martin Gleeson, and The University of Melbourne, and the Quest Protein Database Center, Cold Spring Harbor Labs

Executable versions of timeedit are created using TclPro software which is copyrighted by the Regents of the University of California, Sun Microsystems, Inc., Ajuba Solutions, and other parties.

Table 1. List of Test-Harness Commands

Symbol	Single- or Multi- Line	Meaning
%F	single	Set target filename for ANSI C-source.
%X	single	Defines C source procedures to be extracted.
%G	multi	Sets parameters used by the FeaVer GUI, feaver_gui.
%L	multi	Defines a filter (i.e., mapping table) to be used in model extractions via one or more %X commands.
%H	single/ multi	Defines optional header information for generated code.
%D	single/ multi	Declares C data types used in the generated code.
%C	single/ multi	Defines C data declarations used in the generated code.
%P	single/ multi	Defines process infrastructure in Spin modeling language.
%O	single	Defines extra directives and linked external C-files for final compilation of the generated model checker
%Q	single	Defines extra directives for preprocessing source files from which models are to be extracted.

5. Reference Guide

Overview of Test-Harness Commands

There are ten types of commands that can appear in a test harness definition, as summarized in Table 1. The **%F, %X, %O,** and **%Q** commands always appear on a single line, with parameters following the command symbol. The commands **%H, %D, %C, %L, and %P** can be used either in single line or in multi-line mode. When used as single-line commands, the command prefix is followed by a parameter that gives the name of a file in which the details for the command are stored. The command **%G** can only be used as a multi-line command. The appearance of the symbol **%G**, for instance, introduces a separate segment of the test harness that must be concluded with a double percent sign **%%**. A rule for interpreting a named **%L** command is that the model extractor will first try to open a file wiith the name that follows the **%L** prefix, if this fails then the name is used to identify the lookup table contents that follows, up to the first **%%** prefix. A single-line **%L** command, refering to a named file, is also considered to be a named lookup table, where the prefix of the file name serves as the reference.

We have already seen some examples of many of the test harness commands, but not all. We will review all commands below and give a synopsis of their intended use, possible parameters, etc. We will do so in the order in which the commands are listed in Table 1, matching the most likely order in which they would be used in a test harness. The order can help structure a test harness specification, but is otherwise not strictly important. There is just one exception to this rule where order does matter: an **%X** command always refers to the last preceding **%F** command. A test harness file therefore usually begins with an **%F** command.

Table 2. Valid Parameters to an %X Command

Parameter	Meaning
-x	Extract all procedures from the target C-source file.
-a pname	Extract procedure pname as an active proctype.
-i pname	Extract procedure pname as an inline definition.
-p pname	Extract procedure pname as a passive proctype.
-e pname	Extract procedure as an extended active proctype (instrumented to support a procedure call mechanism).
-E pname	Extract procedure as an extended passive proctype.
-n pname	Extract procedure pname without proctype encapsulation.
-L mapname	Define a map to be used in subsequent model extractions.

%F Sets the filename for the source of subsequent **%X** commands. There can be multiple **%X** commands in a test harness, one for each procedure that is instrumented by model extraction. The last **%F** command before each **%X** command will be the one that determines which source file will be scanned for the procedure. If no **%F** command is given, the source file defaults to the name of the test harness with the suffix **`.prx`** replaced with **`.c`**. A warning is generated if that file does not exist, and naturally any subsequent **%X** command would fail in that case.**%X**

Defines one or more procedures that are the target of a model extraction attempt. An option flag preceding the procedure name specifies in what form the model is to be encapsulated into the model. Five possible option flags are summarized in Table 2. The **`-x`** option does not take an argument, since it targets all procedures found in the current source file. The four remaining option flags take exactly one procedure name as an argument. Each option flag can be preceded by an **`-L`** option that specifies a named mapfile that is to be used for this extraction.

The recommended format is to group options that relate to the same target of a model extraction command on the same command line. For instance:

```
%X -L map1 -a p1
%X -L map2 -n p2
```

Any other grouping of white-space separated options, however, is also valid. If the test harness contains an unnamed **%L** command, then the map defined by that command is applied to all model extractions performed, in addition to possibly named maps specified with **`-L`** options. In case of conflicts, a named **`-L`** option always takes precedence over an unnamed mapfile.

A missing %X command by default implies that model extraction is applied only to the *main* procedure of a target C source file.

%G The **%G** command is used for setting persistent parameter values for use in the graphical user interface to FeaVer. Currently only the options listed in Table 3 on page 48 are recognized. Some of the parameters determine the type of search that will be performed, others determine particulars of the way that message sequence charts for error sequences are generated.

%L The **%L** command is used to define filters, i.e., mapping tables, either with an unnamed command (i.e., when the **%L** is not followed by an identifying name) that

Table 3. Valid %G Options

Option Prefix	Default Value	Meaning
maxdepth:	1000	Limit search depth for model checking to 1000 steps.
memlim:	350	Limit memory use for model checking runs to 350 Mbytes
loops:	0	Do no search for accept cycles (values 0 or 1)
np_loops:	0	Do not search for non-progress cycles (values 0 or 1)
noend:	1	Do not search for invalid endstates (0 or 1)
asserts:	1	Search for assertion violations (0 or 1)
shortest:	0	Not necessarily the shortest possible error (values 0 or 1)
quality:	3	Set a medium search coverage (range 0..6); 0 gives maximal speed, 6 gives maximal coverage
exhaustive:	0	Do not force an exhaustive search coverage (values 0 or 1)
msc:	0 1 2 3 4 5 ...	Place processes in MSC in order of creation
hidden:		Do not hide any processes in the MSC display
io_only:	0	List all operations in MSCs, not just i/o (values 0 or 1)
use_vals:	0	Do not display parameter values in MSCs (values 0 or 1)
zflag:	0	Do not process system header files when preprocessing source files for model extraction (values 0 or 1).

applies to all models extracted through this test harness, or with named mapping tables (an **%L** command followed by a unique name that can be refered to in an **%X** command with the **-L** option). There can be any number of mapping tables, but only one unnamed one, and only one for each unique name used.The main purpose of a mapping table is to define conversions from statements or conditions that appear in the source text to abstracted versions that are to be used in the model. There is a predefined conversion that simply encapsulates each statement in a Promela **c_code** fragment and each condition in a Promela **c_expr** fragment,

while adjusting all variable references for the imported data. A user-defined conversion is therefore only needed when the default isn't adequate. A conversion rule can be used to suppress pieces of code that are outside the check, to simplify or abstract pieces of code, or to lift certain operations up to the level of the test harness, overriding the builtin functionality of the application itself. This would, for instance, be done if the real application environment is replaced with a set of test drivers that simulate all the behaviors of interest of that environment in a controlled way. Send and receive operations are typically replaced with test artifacts in this way, process control and thread control operations similarly are usually replaced with model specific equivalents.

There are also a small number of special commands that can be used within a mapping table definition. These special commands are summarized in Table 4 on page 50.

A Constant command is used to introduce names that are known to have a constant value (which means that they need not be tracked dynamically as state information). For instance:

```
Constant  zero   one
Constant  True
```

The Include command specifies the name of a secondary mapping table that will be read in and inserted into the current table in place if the Include command. For instance:

```
Include shared.lut
```

The **`Declare`** command takes three arguments: first the name of a data object, then the type of that object and finally the scope of that object. A scope definition consists of either the predefined word **`Global`**, or the name of a procedure within the source file, for a local variable. The effect of the command is that a declaration of the given type will be included as part of the state descriptors in the generated test code. For instance,

```
Declare  struct rlV5IFDB   ifdb_11     Global
Declare  int               number      main
```

The Import commands takes two arguments: the name of a data object that appears in the source file, and the scope of that object, as before. The data object must be referenced in at least one of the procedures that is the target of a model extraction, or else it has no effect. For instance:

```
Import   ifdb_11     Global
Import   number      main
```

A NonState commands takes two arguments: first is either the keyword "**`hidden`**" or the type of a data object. The second argument is the name of the data object. The scope of a NonState variable is always assumed to be global. If the first argument is "**`hidden`**" then the declaration will not appear in the test code that is generated. (There may be cases where the data object must be known during the model extraction phase, even though it may not needed in the generated code). For instance:

```
NonState V_State     *tossinfo
NonState char        dummy[64]
NonState hidden      a[]
```

Table 4. Mapping Table Special Commands

Name	Nr. Params	Description
Constant	>=1	Declare one or more names as symbolic names for constant values.
Declare	3	Add a data declaration that does not appear in the source.
Import	2	Identify data objects in the source that must be part of the model.
Include	1	Recursively include a subsidiary mapping table.
NonState	2	Declare data objects that carry no state information.
Substitute	2	Define a simple rewrite rule for mappings
Track	2	Declare data objects in encapsulated code (not part of the extracted models) that carry state information that must be tracked by the model checker.
Icall	1	Identify the name of a procedure for which all calls are to be instrumented as calls on other extracted proctypes (instrumented calls).

A Substitute command takes two arguments. The first argument is a text-string that is to be replaced with the second argument after all other mapping table commands have first been applied. (Somewhat obsolete.)

The Track command takes two arguments. The first argument is a pointer to a data object that is declared externally to the code managed by the model checker (i.e., code that will be linked with the model checker to perform an executable for the test system). The second argument defines the size of that object.

All entries into a mapping table that do not match any of the special commands are interpreted as conversion rules. A conversion rules always consists of two parts, separated by one or more tabs. The left hand side of the rule specifies the source text to be replaced, and the right hand side defines the replacement text of that piece of source code in the extracted model.

There are a few short-hands that can be used on the right-hand side of a conversion rule. The available short-hands are summarized in Table 5.

There is one type of pattern that can be used on the left-hand side of a conversion rule. Any text-string that ends in three dots is understood to be a prefix of a code fragment, and it will match any statement of expression that begins with the string before the three dots. For instance

```
printf(...     warn
```

Table 5. Predefined Mapping Keywords in Lookup Tables

Keyword	Meaning
comment	Includes the statement as a comment in the model
hide	Omits the statement from the test model.
skip	Like hide, but replaces statement with a null-step.
keep	Copies the statement without syntax conversions.
print	Includes the statement text in a print statement.
warn	Issues a warning if the left-hand side is encountered.
true	define an expression to always evaluate to true.
false	define an expression to always evaluate to false.
C_code	Encapsulates the statement as is inside the model.
C_expr	Encapsulates and treats as an expression.

could be used to issue a warning for every **`printf`** statement that appears in the source text for a target procedure.

If multiple matches are possible, the first match encountered in the scanning of the mapping tables will take effect. A named mapping table is always scanned first. A default unnamed table is scanned last.

%H The **%H** command is rarely used. It provides a capability to prepend a set of definitions or declarations at the front of a source file, so that these definitions are read by the model extractor before the parsing of the source file itself begins. This capability is useful when, for instance, a source file is imported from another platform and some data types must be redefined for portability. The required information can be placed in a file, which can be named as the argument of a single-line version of the **%H** command; it can also be specified inline.

%D The **%D** command is used to declare data types for imported data objects in the test harness. Import commands in the mapping tables only affect the variable objects themselves, not the corresponding type declarations. The **%D** command gives the opportunity to define specialized versions of the data types used in an application. It can also simply include a header file with the original type declarations from the application. If an include file is used in the test harness, it is recommended to prefix the # character in the left margin with a backslash, to defer interpretation to the final compilation step. This bypasses possible bounds that may exist in the model checker. As with the **%H** command, the specifics for the **%D** command can be placed in a file which is named as a single argument to a single line version of the **%D** command, or it can be defined inline.

%C The **%C** command can be used to introduce arbitrary C code into the model. It is useful, for instance, when some small additional procedures must be defined. An alternative is to place such code in an external file and to link it with the generated code.

%P The **%P** command is the main vehicle for defining the architecture of a test model, the test drivers and possible channel declarations and Promela data objects, as illustrated in the examples.

%O The **%O** command is used to list compile-time directives and external C-files that are to be linked with the test executable that is generated by FeaVer. For instance, it could say:

```
%O ipc_stubs.o Downlink.o
```

to define that the two precompiled modules are to be linked in with the test executable, presumably because these contain the code for procedures that are called from within test-drivers or from within instrumented code. The command is rarely needed.

%Q The %Q command is used to define additional preprocessing directives that may be needed to properly parse the source code before model extraction takes place. An example is:

```
%Q -DCPU=SIMSPARCSOLARIS -DFSW_UNIT_TEST -IWind
```

which defines two compile-time constants and sets the path for a directory to be searched for include files. It is rarely needed.

6. Recommended Code Structure

There are no absolute requirements on the structure of the code that is subjected to a FeaVer check. Nonetheless, the construction of a test-harness can be simplified if the code to be tested is well-structured. The intended structure can best be illustrated with a small example.

The processes of a distributed software system are sometimes described as *reactive*: in response to external events a process can perform some computations and generate new events in response. These new events can trigger subsequent behavior in other processes, and so on. Generally, the events of interest are message passing operations, with the arrival of a message at a process the main type of event to be responded to.

Event Driven Code

To implement this type of *event-driven* system, each process will generally have an event-queue where pending events are stored, and an event handler that executes a basic loop, retrieving events from the event-queue and executing a predefined set of instructions that is determined by the process's current *control state* and the *type of event* being processed.

It simplifies the construction of a test harness if the process communication and control operations are clearly identified and appear at the same level of functionality in the code. There are two parts to every interface definition: input and output. The best case is if all inputs are handled through the event handling mechanism, and all outputs are visible within the event-handling loop itself, instead of appearing inside subroutines invoked from within the event-handler. The same holds for process control operations. If the application can invoke multiple threads, fork asynchronous processes, or set or release mutual exclusion locks, the best case is if these process control operations can be made to appear only within the main event-handler.

A structure like this not only facilitates the construction of a test harness, it also increases the readability and maintainability of the code. All operations that can impact process control are presented in a clear and structured manner; all subordinate computations are pushed down into lower-level routines or libraries.

Example

Following these guidelines, a device handler can, for instance, be written in ANSI C as shown in Figure 30 on page 54. The event handler is to be invoked once for each new event that arrives, from the main event loop of the component. The handler first checks its current control state, and then checks the type of event that was received. It then performs the corresponding actions, which may involve the generation of new events. Computations that do not involve message passing or top-level process control are delegated to lower level procedures. Accounting operations are of that type. Checking whether a hardware device is in-service or broken, idle or busy, would similarly be relegated to lower-level procedures.

The general rule-of-thumb to be used here is that if an operation has a top-level effect that can affect process execution elsewhere in the system, it should be visible at the top-level in the event-handler. If an operation has only a local effect, it can be hidden inside lower-level code.

```
void
event_handler(Device *dev, Event *ev)
{
         switch (dev->state) {
         case Idle:  if (ev->type == Request) {
                          dev->state = Busy;
                          respond(ev->sender, ACK);
                     } else if (ev->type == Release)
                          respond(ev->sender, NAK);
                     break;
          case Busy:  if (ev->type == Release) {
                          dev->state = Idle;
                          respond(ev->sender, ACK);
                     } else if (dev->type == Request)
                          respond(ev->sender, NAK);
                     break;
         case Error:
                     if (ev->type == Reset)
                          dev->state = Idle;
                     break;
         default:
                     error_report(dev, ev);
                     dev->state = error;
                     respond(ev->sender, ERR);
                     break;
}         }
```

FIGURE 30. Recommended Code Structure

7. Trouble-Shooting Guide

The most challenging part of working with FeaVer is often in the initial definition of a verification test harness. Once the test harness is set up, there is relatively little maintenance to be done and things will progress very fast.

When starting from scratch for a new software application, there are some guidelines that can be followed about the best way to proceed from step to step. The easiest way to get started is to use FeaVer's graphical user interface. It provides guidance on the initial definition of mapping tables (%L segments) etc. When a problem is encountered that cannot be fully investigated with the builtin options of the graphical user interface, though, it can be invaluable to be able to reproduce the steps that are performed one by one, to inspect the results of each step carefully to home in on the root-cause of a problem. Below we will therefore walk you through a series of manual steps that can be made from the command line to reproduce the main processing steps that the FeaVer GUI normally performs automatically. This is just for trouble-shooting purposes, or perhaps to explain in a little more detail all that happens behind the screen in a session with FeaVer.

Small Example

To begin, we will look at two very small examples, and see how FeaVer can guide us towards the definition of the basic elements of the test-harness by looking at the types of error messages that are generated.

```
int shared = 0;
int *ptr;

void
thread1(void)
{  int tmp;

   ptr = &shared;
   tmp = shared;
   tmp++;
   shared = tmp;
}

void
thread2(void)
{  int tmp;

   if (ptr)
   {     tmp = shared;
         tmp++;
         shared = tmp;
         assert(shared == 1);
   }
}
```

FIGURE 31. Threads.c Test-Case

In Figure 31 the simple threads example is shown, listed earlier in Figure 1 on page 6. Suppose we want to outline an explicit test harness for this piece of code, not relying on the default version generated by the model extractor when no **`.prx`** file is specified.

The **`Browse`** window in the FeaVer GUI only displays files with the extension **`.prx`**, **`.vip`**, or **`.pml`**, identifying respectively test harness files, time-line properties, and Promela files. In particular, C source files are not shown, unless they are identified through %F commands within a test harness. To start a new test harness definition, simply click the **`Prx`** tab, located approximately in the middle of the FeaVer display when the tool is invoked. (Invoke the tool in the directory that contains the file `threads.c`.) The GUI will warn “no .prx file”. Click the OK button and type the following command in the empty Prx window:

```
%F threads.c
```

Save this minimal **`prx`** file under the name **`threads.prx`**, by using the corresponding **`File Menu`** entry.

Invoking the Check button on the FeaVer GUI makes FeaVer perform two steps: it extracts the model and attempts to create a checking program from it. This second step, however, will fail because the test harness description is still incomplete. The error message is displayed in the `Out` panel, at the bottom of the FeaVer display. The error report comes from `Spin` and says, among other things:

```
spin: line 2 ,"_modex_out", Error: no runnable process.
```

At the same time, the model tab remains empty: it shows that no Spin model could be generated. (Checking in the files system, you will see that a file was created

```
#include <stdio.h>

#define N5
#define M4

int
main(void)
{  int i, q[M], *p[N][M];

   for (i = 0; i < N; i++)
         p[i][3] = &q[i];
}
```

FIGURE 32. A Similar Test Case: Bounds.c

named "**_modex_out**", but it is empty. Intermediate files such as these are created automatically by FeaVer, and deleted when they are no longer needed.)

The problem that we run into here is that the default behavior of FeaVer in the presence of a test harness file is to extract a model for the procedure named **main** in the source file that was indentified with a **%F** command. There is no such procedure in our example, so no process behavior was defined, hence no runnable process generated, hence the complaint from **Spin**. What we really want is to extract a model from each of the procedures in the file **threads.c**, and we can do so with an **%X** command. We now extend test harness definition with an extra line, and save it:

```
%F threads.c
%X -x
```

If we click the `Check` button again, the second step of the model extraction can now be completed, and a model is displayed in the **Model** tab. But this time the third step fails (the compilation of the generated code), triggering errors from the compiler. Because of these errors, the generated model checking code is displayed in extra tabs labeled **Pan.c** and **Pan.h**.

To find out what is wrong in this case it suffices to look at the error message in the **Out** tab, at the bottom of the FeaVer display. The compiler complains that a reference to variable **now.ptr** is undefined. The default behavior of FeaVer in the presence of a test harness file is not to import any data object unless explicitly requested with `Import` commands within `%L` segments. We can chose to import all data objects referenced in all procedures, by adding a single line global import command discussed earlier in this guide. The resulting test harness now looks as follows:

```
%F threads.c
%X -x
%L
Import   _all_       _all_
%%
```

Now if we save the file and click the Check button, the check will complete all steps in the model successfully and display the example execution that shows that the assertion that was placed in the code can indeed be violated.

Exercise Try the same experiment with the file `bounds.c`, illustrated in Figure 32. Create the file **bounds.prx** with the final version of your test harness.

As suggested by the above two examples, to successfully complete a software test with FeaVer, we have to setup the test harness to take us passed a series of hurdles. Each step in the process can in principle fail, and generate it's own error messages. By eliminating the causes of error one by one, we can converge relatively quickly on a usable test harness description. We will discuss the main steps taken by FeaVer one by one in a little more detail below.

Step 1 The test harness file should minimally define the C source file(s) that is (are) the target of the test (using one or more **%F** commands). It should also define the target procedures that will be modeled as asynchronous process threats (using **%X** commands). Often some test drivers will need to be added, either in **%C** or in **%P** segments to close the system, together with a minimal infrastructure that is needed to expose the communication between the processes to the checking system. To be able to setup the checking system for this, take some time up front to decide on the main mechanism for process communication, especially whether it should be represented in C or in Promela message passing primitives.

Step 2 Next we should look at the definition of initial versions of the mapping tables for the **%L** segments in the test harness. To find out what the syntactic requirements for the entries in the mapping table are, we can let the model extractor generate all the required information for us. We can do so manually by calling the model extractor with the command:

```
modex -Y -k check.prx
```

If this is the very first time that the model extractor is invoked for a new application, this may produce a series of parsing errors for the C source code that was specified as the target of the model extraction exercise. Don't worry too much at this stage. Errors could appear, for instance, if the source file presented isn't fully ANSI-C compatible, or if some of the header-files used are missing on the system where the check is being done. We can make changes in the source by using the compiler directive "**MODEX**" to hide new segments from a real compilation for the executable version of the code. Do so by bracketing any additions to the code that are just meant to satisfy the model extractor with:

```
#ifdef MODEX
      some declarations to satisfy the modex parser
#endif
```

Once the parse errors disappear (or start looking harmless), look carefully at the files with the extensions `*.nlut` that were generated. These files contain the fully populated mapping tables for each target that was defined in an %X command in the test harness, with all default conversion rules filled in.

We generally need to select and redefine only a small fraction of the entries in the **%L** segments to setup a check, so do not be discouraged if this initial listing is very long. The main purpose of the listing is to provide us with a precise spelling of the left-hand side entries from the source statements for which we may want to introduce a conversion rule in an **%L** segment. (When using the graphical user interface, there is an entry in the Options menu to populate the **%L** definitions in the same way.)

Step 3 Determine the desired selection for the **%L** segments in the test harness. Minimally you should redefine thread operations (mapping them to `hide`), any process instantiations (using Promela operations to redefine them) and all send and receive operations (using the chosen conversions to represent them). Pay special

attention to the handling of any parameter values that may be passed to a procedure upon thread initializations. Those values will have to be passed to the model version of the process in some other way, for instance via global variables, or by passing them as parameters in a message that is sent to the process over an internal channel.

Once all this is done, make a first attempt to generate the full verification model. From the command line interface this is done by invoking modex without parameters:

```
modex check.prx
```

Inspect the various parts of the model that are generated. To get a good overview of the complete model, use the C preprocessor to assemble it from the pieces that were generated. For instance on a Unix system you would use:

```
/lib/cpp *.drv > model
```

The model extractor generates a number of files. A script that will cleanup all files that are generated is available in a little helper file with the extension `*.cln`. The FeaVer graphical user interface uses that file to cleanup, but you can do the same manually of course.

Step 4 Next, we can attempt to generate the model checking code itself from the model. We can do this manually with the command:

```
spin -a model     # using spin version 4.0 or later
```

On the graphical user interface this call is invoked automatically in the background once errors in the previous steps have been fixed.

At this point, we may get some parse errors from `spin`. This can happen if the mapping tables are incomplete or contain errors, if not all declarations are in order, etc. It is usually not hard to find out where the complaints come from. Adjust the variable declarations, using `%D` segments, check the declarations and tracking statements in the `%L` segments until all errors reported in this step are fixed.

Step 5 Compile the code that `spin` generated. This is the model checking code that can perform the check of the generated system. From the command line this can be done by invoking a standard C compiler, for instance gcc:

```
gcc -o pan pan.c
```

This can also be done by invoking the script from the command-line interface to FeaVer itself, e.g.:

```
feaver_cmd file.prx
```

Check this script to see which commands are executed. We may get some more errors from the C compiler at this point. This can happen if the C code contained within `c_expr` or `c_code` statements contains errors. Repair the errors by revisiting data declarations, checking conversion syntax, etc. until a clean model is obtained. The error messages from the C compiler will pin down exactly where the problems come from.

Note that in all this we are editing the generator for the model code, not the model code itself, so our changes can more easily survive modifications of the original source code of the application later.

Step 6 Run the check by executing the executable `pan` program that was created in the last step. There may be some runtime errors, but these should be rare. The model checker may, for instance, ungraciously dump core and abort when it is asked to execute a faulty piece of C code that was encapsulated into the model checker. Locate the error in the same way that you would do this for a regular C program. It is usually sufficient to locate only the statement that caused the runtime error to happen (using your favorite C debugger to inspect the core file).

To get a clear picture of the **`pan.c`** source, it is best to manually inline the contents of the **`pan.m`** file into the main pan.c source-file before compiling the program. In this way you can get accurate source line references from the debugger also if the offending code appears in an included file (such as **`pan.m`** or **`pan.b`**). To do the inlining, find the *two* places in the **`pan.c`** file that say:

```
#include FORWARD_MOVES
```

The definition for **`FORWARD_MOVES`** appears in the **`pan.h`** file. In this application it is always defined as **`pan.m`**. Replace these two lines with the contents of the pan.m file. (Only one of the two pieces of code can actually be enabled at a time, but it is a little harder to determine precisely which one that is in each case.)

The Final Run At some point, when errors in the test definition are all taken care of you will start seeing real error reports from the model checker. To run the model checking code, you can simply invoke the executable and rely on the default parameter settings:

```
pan
```

By default the search it limited to a depth of 10,000 steps. To set a different depth-limit, say a shorter one of only 100 steps, use

```
pan -m100
```

If the model checker reports that it runs out of memory, it may be better to compile the pan.c source for a high-coverage approximate check, for instance as

```
gcc -DBITSTATE -o pan pan.c
```

at this point you can do runs of increasing coverage (and increasing runtime) by trying a sequence like this:

```
pan -w20
pan -w22
pan -w24
```

and so on, until an error is reported, or you again run out of memory or time. Spin has many more options for handling complexity that cannot be covered here. Refer to the online manpages and tutorials for the details:

```
http://netlib.bell-labs.com/netlib/spin/whatispin.html
```

Once you get to this point, though, you have successfully created a test harness. If error sequences are reported, inspect each such scenarios and report or fix what needs fixing in the code of the application itself! From the command line you can reproduce an error scenario discovered in a `pan` run with the command:

```
pan -C
```

Note that with included C code you cannot do this by invoking `spin` in guided simulation mode. Only model checking code that is generated by **`spin`** Version 4.0 or later will recognize the **`-C`** runtime option. The **`FeaVer`** graphical user interface will

```
void
main(void)
{
   printf(“hello world\n”);/* shows up in pan.m */
}
```

FIGURE 33. The Infamous Hello World Example

```
%F dummy.c
%X -x
%L
printf(...       keep
%%
%H
typedef int mutex_t;/* shows up at the start of dummy.M */
%%
%D
typedef d_part {
   int d_1;
   char d_2;
} d_part;        /* shows up in pan.c before pan.h */
%%
%C
State c_part;    /* shows up in pan.c after pan.h */
%%
%P
int p_part;/* shows up in pan.h inside state vector */
#include “_modex_all.spn”
```

FIGURE 34. Experimental Test Harness for Hello World Example

automatically generate appealing graphical message flow diagrams for all errors detected.

Vacuity Checking

If no errors are reported, it is good to carefully check the code coverage that is reported by the model checker at the end of each run. (By invoking pan with the -v verbose runtime option code coverage is reported not just in the main model but also in any test automata derived from timeline properties, if used during the check.) If certain parts of the code appear to be systematically missed, it is possible that the test drivers, or other parts of the test harness definition are to blame for that. If not, the code could be truly unreachable, which is likely to be an error in and of itself.

Tracking Down What Happens

The precise effect of four of the test harness commands in practicular, **%H %D %C** and **%P**, can be tricky. (The effect of the other four, **%F**, **%X**, **%G**, and **%L**, can be observed more easily, with the explanations given in this guide.) To see the precise difference between these four commands, we’ll discuss a small experiment. We will use the standard ‘hello world’ C program as our test input, illustrated in Figure 33, and setup a small test-harness, shown in Figure 34, with some simple declarations that will allow us to track down how elements of the test harness carry over into pieces of the model checking code that is generated.

```
open(...          comment
close(...         comment
fflush(...        comment
exit(...          comment

fprintf(...       comment
printf(...        comment

write(...         comment

fgets(...         warn
fscanf(...        warn
read(...          warn

connect(...       warn
socket(...        warn
gethostbyname(... warn
getprotobyname(...warn
```

FIGURE 35. Default Conversion Rules

The `%X -x` command in the test harness preserves the single procedure `main` as the target for model extraction. The `%L` segment contains just one command, which is meant to override the default treatment of `printf` statements.

> Unless otherwise defined, the model extractor will use the default conversions shown in Figure 35. Most file operations, except those that read data, are converted into comments. Operations that retrieve data from a file are treated with some care, since their effect is not necessarily reversible, and modifies potentially relevant state information. The use of these operations within a procedure that is to be instrumented by the model checker will generate a warning.

In this case we want the `printf` statement preserved so that we can see where it shows up in the model checking code.

The model checking code is generated by `Spin` from the model that is produced by the model extractor `Modex`, following the instructions recorded in the test harness. The Spin generated model checking code traditionally consists of the following X files:

`pan.h`, which contains all header information, and data declarations,
`pan.c`, which contains the model checking procedures,
`pan.m`, which contains the code generated for each transition in the system,
`pan.b`, which records how the effect of each transition can be undone,
`pan.t`, which records the control flow structure for all generated process types.

As noted in Figure 33, we should expect the `printf` statement to show up as an entry in the large case-switch that is defined in the `pan.m` file.

%H The declaration of the `typedef` for `mutex_t`, given in the %H segment of the test harness, would at first appear to have disappeared, since it does not show up literally in any of the `pan` source files. It does appear in another file that is automatically created in the directory where the file with the ‘hello world’ example is stored. This intermediate file has the extension `.M`, and it contains the output of the preprocessing phase for the target source file.

If there are multiple target source files, as defined in a series of %F commands in the test harness, then the results of preprocessing each will be stored in a separate .M file.

The .M file

The model extractor will not extract its models from the original .c file but from an intermediate .M file that is created after all relevant include files are inlined and all macros expanded. Before the model extractor starts, it always checks if there is an .M file in the directory that is more recent than the target .c file. If so, the model extractor will skip the preprocessing step and move straight on to the processing of the .M file.

This can sometimes also have undesired consequences, for instance if a file that is *included* into the .c file is updated. The update may then be invisible to the model extractor, which will continue to work with the previously produced .M file, missing the update. If this happens, you can force an update of the .M file by removing it manually, or by re-writing the .c file (to make it newer than the .M file).

The declaration for mutex_t appears at the top of the file dummy.M, together with any other entries that could be specified in a %H segment. No processing is done on these entries: they are copied literally into the .M file. In this case, the typedef allows us to process data objects of type mutex_t, even in the absence of a library header file where this type is declared. For the purposes of the test system, we now use such data objects as integers. In a similar way we can use entries in the %H segment to rename data objects with macro definitions, etc. We can in this way make small changes to the original source code of an application without having to directly modify that code. The %H segment gives us a way to add small overrides into the code through definitions that are recorded directly in the test harness.

%D

The declaration of the data structure d_part from the %D segment of the trial test harness shows up in the main file pan.c, *before* the point where the pan.h header file is included. This gives us a way to introduce data types that can be refered to from within the state vector. A Declare statement in an %L segment, for instance, can introduce a data type that must be made known to the model checker. The %D segment is the proper place to declare such a data type.

%H vs %D

Declarations that are placed in the %H segment modify the source code of the application, *before model extraction* takes place. Declarations placed in the %D, %C, and %P segments modify the target code of the model checker, which is compiled *after model extraction* takes place.

%D vs %C

The declaration of c_part from the %C segment will show up in the pan.c file *after* the point where pan.h is included. This gives us a way to refer to the state vector (State) itself, or to any other data structures that appear inside the pan.h file, both in declarations and in C procedure definitions. From their relative placement in the code, the %D segment of the test harness is the logical place for data **D**eclarations, while the %C segment is the logical place for defining the body of **C** procedures that are to become part of the test system.

%P

The declaration of the integer p_part, finally, will show up inside the state vector itself, which is defined in file pan.h. Note that the %P segment introduces Promela code, and therefore the declaration of p_part is treated as a global Promela object that is automatically declared as such, and that could be refered to from within C procedures under the extended name now.p_part.

Technically, there is yet another way to introduce C code into the model checking code by declaring a standalone global `c_code { ... }` statement inside the %P segment of the test harness. The effect of this would be the same as if the contents of the `c_code` statement (without the c_code wrapper) are included in a regular `%c` segment

References

[H97] G.J. Holzmann, “The model checker Spin,” *IEEE Trans. on Softw. Eng.*, Vol. 23, No. 5, May 1997, pp. 279-295.

More details on Spin can also be found online at:

- http://spinroot.com/spin/whatispin.html

[HS99a] G.J. Holzmann and M.H. Smith, “A practical method for the verification of event-driven software,” *Proc. Intern. Conf. on Software Engineering*, May 1999, Los Angeles, CA., pp. 597.607.

[HS99b] G.J. Holzmann, and M.H. Smith, “Software model checking: extracting verification models from source code,” *Software Testing, Verification and Reliability*, Vol. 11, No. 2, June 2001, pp. 65-79.

[HS00] G.J. Holzmann, and M.H. Smith, “Automating software feature verification,” *Bell Labs Technical Journal*, Vol. 5, No. 2, April-June 2000, pp. 72-87.

[H00] G.J. Holzmann, “Logic Verification of ANSI-C Code with Spin,” *Proc. SPIN Workshop*, Stanford University, CA., Aug-Sept. 2000, Springer Verlag, *Lecture Notes in Computer Science*, Vol. 1885, pp. 131-147.

[H01] G.J. Holzmann, “From Code to Models,” *Proc. 2nd International Conference on Concurrency to System Design*, Newcastle, U.K., IEEE Computer Society Press, June 2001, pp. 3-10.

Table 6. Summary of Test-Harness Commands and Arguments

Command	Argument	Default	Meaning
%F	*fname*		Set target C source filename for model extractions.
%X Defines the procedures that will be extracted.	-x		Extract all procedures from the target C-source file.
	-a pname		Extract procedure pname as an active proctype.
	-i pname		Extract procedure pname as an inline definition.
	-p pname		Extract procedure pname as a passive proctype.
	-e pname		Extract procedure as an extended active proctype (instrumented to support a procedure call mechanism).
	-E pname		Extract procedure as an extended passive proctype.
	-n pname		Extract procedure pname without proctype encapsulation.
	-L mapname		Define a map to be used in subsequent model extractions.
%G Sets parameters used by FeaVer GUI, feaver_gui	maxdepth:	1000	Limit search depth for model checking to 1000 steps.
	memlim:	350	Limit memory use for model checking runs to 350 Mbytes
	loops:	0	Do no search for accept cycles (values 0 or 1)
	np_loops:	0	Do not search for non-progress cycles (values 0 or 1)
	noend:	1	Do not search for invalid endstates (0 or 1)
	asserts:	1	Search for assertion violations (0 or 1)
	shortest:	0	Not necessarily the shortest possible error (values 0 or 1)
	quality:	3	Set a medium search coverage (range 0..6); 0 gives maximal speed, 6 gives maximal coverage
	exhaustive:	0	Do not force an exhaustive search coverage (values 0 or 1)
	msc:	0 1 2 3 4 5 ...	Place processes in MSC in order of creation
	hidden:		Do not hide any processes in the MSC display
	io_only:	0	List all operations in MSCs, not just i/o (values 0 or 1)
	use_vals:	0	Do not display parameter values in MSCs (values 0 or 1)
	zflag:	0	Do not process system header files when preprocessing source files for model extraction (values 0 or 1).
%L	Defines a filter (i.e., a mapping table) to be used in model extraction..		
%H	Defines optional header information for generated code.		
%D	Declares C data types used in the generated code.		
%C	Introduces arbitrary C code into the model.		
%P	Defines process infrastructure in Spin modeling language.		
%O	Defines directives and external C-files for compilation of generated test system.		
%Q	Defines directives for preprocessing source from which models are to be extracted.		