



---

## **Spotlight Support for Model Driven Test (MDT)**

Version 1.9  
April 15, 2005

---

### *PathMATE Technical Notes*

Pathfinder Solutions LLC  
33 Commercial Drive, Suite 2  
Foxboro, MA 02035 USA  
[www.PathfinderMDA.com](http://www.PathfinderMDA.com)  
888-662-7284

# Table Of Contents

1. Introduction..... 1

## 1. Introduction

This Technical Note describes extensions to PathMATE Spotlight to facilitate Model Driven Test (MDT) through the following capabilities:

- Tailored and automated tracing configuration specifically to support automated testing
- Specification of test suite executive control, test result analysis, and individual test stub/driving via interpreted PAL
- Conversion of captured execution traces to driver files
- Tools to diff trace logs and instance dumps
- Test Coverage instrumentation

The PathMATE Console will add extensions for test suite management and sharing of setup, configuration, and stimulus between test cases.

Section 2 describes the features that will be implemented for PathMATE MDT support. Section 3 describes how these features affect existing and planned PathMATE tools and components. Section 4 captures some good ideas for future developments. Section 5 captures the breakdown of features into 2 phases of development, with incremental deliverables.

### Configurability of Mechanism and Generated Instrumentation

Note that we will continue to support disabling instrumentation at translation, compile, and run-time. At translation time, the property "SpotlightEnabled" can be set to "F" to turn off instrumentation. At compile time for C and C++ the compile switch NO\_PATH\_IE when defined will disable instrumentation if it has been generated. For Java, the mechanisms class, PfdDefine, defines a public static final int NO\_PATH\_IE, which disables instrumentation at compile time.

At run-time, the instrumentation currently requires Spotlight to connect to it. An enhancement has been logged to allow the application to check for Spotlight on the local machine, and auto connect if it is found or continue without Spotlight if it is not.

## 2. Phase 1 Feature Overview

The primary goals of Phase 1 are to:

- Provide features focused on the automation of basic test case set up and execution
- Deliver the capabilities rapidly and start gaining feedback from field use

Features selected are based on ease of integration into the existing PathMATE toolset and benefit to the MDA developer and tester. Additionally, some foundation activities described in the Phase 2 section will be completed in the Phase 1 release timeframes – see section 8 Functionality Release Schedule.

## Test Coverage

Spotlight and instrumentation will assist in determining the coverage of the test suite. Templates will generate the breakpoint Spotlight driver settings for all possible locations. These include:

- Events
- Operations
- Action language statements
- Transitions
- Entry and exit actions

In coverage mode, Spotlight will be notified of and record the breakpoint, clear the breakpoint, and continue. When the test case finishes, the untriggered breakpoints are stored to be used by the next test case, and the coverage data is also stored in an ActualCoverage file. Coverage data is accumulated across test case executions. In this way, the coverage of the entire test suite can be evaluated.

Reports of test suite coverage from the ActualCoverage file will include the following data in textual format:

- Number of test cases executed
- Percentage of all model elements covered
- Percentage coverage of action language statements
- Percentage coverage of statechart actions and transitions
- Percentage coverage of operations
- Uncovered model elements (PAL statements, transitions, actions, and operations)

Uncovered model elements, operations, actions, and transitions, will be displayed in the model editor in red. Eclipse integration will include a PAL editor that will display the uncovered PAL statements in red. States with event ignored and event deferred non-event transitions out that have not been exercised will also be displayed in red.

Note that this approach to coverage requires full instrumentation to be available, and may affect execution performance and code space on the target environment. The usage scenarios of coverage tools is that they are run infrequently, so execution time is not generally a priority. The Tech Note on Code Coverage describes a design side set of lighter-weight instrumentation that can collect coverage information in a more recourse-constrained setting.

### Effort Estimate:

1. Breakpoint Driver File Generation: 1 day
  2. Spotlight Coverage Mode: GUI 3 days, Back end 2 days
  3. Coverage Data Storage: 1 days
  4. Report Generation: 2 days
  5. Editor Coverage Display (depends on Eclipse integration) : 3 days
- Instrumentation Side Coverage: 21 days (See Tech Note Code Coverage) – This is not part of the MDT deliverables, but included only as a reference to a related tech note supporting more efficient collection of coverage data.

## Test Execution

The test manager will receive indications from Spotlight on the verdict of tests and display the results.

Test results, consisting of

- trace files
- instance dumps
- verdict indicator
- application specific output

will be stored with data describing when and where the test was run. Instance dumps are already dumped into an XML format. Trace files will also be changed to be stored as XML. The verdict is a pass/fail status reported to Spotlight from the test case driver file, and reported back to the test manager to be displayed. Verdict status is set using the setVerdict script command. By default, the status is set to Unknown.

### Effort Estimate:

1. Test Execution/Spotlight Integration: 3 days
2. Result display: 1 day

## Replay Traces

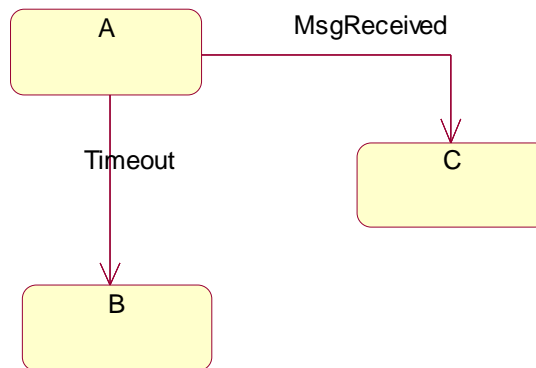
This feature adds a capability to take a captured trace output from a test and run Spotlight to replay the results. This is done by converting the trace file to driver file format. In any case this would require a discernment of external stimuli from internal operation invocations and event traffic.

For example, if the models are running on the target system, with trace instrumentation, then the actual inputs from the target environment can be captured and replayed in the development environment. Inputs from user actions within Spotlight are captured the same way.

When in capture for replay mode, Spotlight will control the trace settings used to capture information for the replay. This will ensure that the replay information is complete.

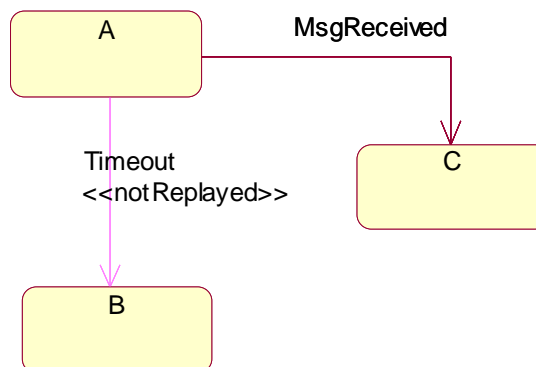
All events will be captured and logged into a file. Events generated by the application in replay mode will be removed and replaced with the events from the original execution. The main reason for this is race conditions, which are common in multi-thread/process deployments, but also single thread deployments with timing considerations. To achieve repeatability, the models must take the same path, which includes the sequence of events. For example, a timeout event may have a race condition with a response event, and this scenario must be replayed identically in order to debug it.

For an example, let's say that an object Class1 is in state A, with transitions to state B on a Timeout event and to state C on a MessageReceived event.



So the object is waiting for a message but will time out if it does not show up. So there is a race condition between the message and the timer. Let's say the initial capture has the message arriving before the timeout.

The test script and instrumentation involved in the execution of the system may affect the timing of the system, so the timeout may arrive before the message. In order to replay the message sequence identically, if the timeout event is generated by the application, it cannot be processed (since it was not processed in the original captured event sequence). Instead, we delete the events from the event queue and supply events from the captured event stream.



With the exception of interfaces, all inputs to a domain come from domain operations, inter-task communication, or from Spotlight. These sources can be instrumented to identify internal from external event sources.

**Effort Estimate:**

1. Spotlight Capture Mode : 2 days
2. Spotlight Replay Mode: 2 days
3. Instrumentation changes: 4 days
4. Trace file XML format: 1 day

## Test Output Support Tooling

To facilitate the easy use of these capabilities to run large suites of tests, we need to have high-level tools that provide concise and meaningful summary info.

These tools are:

- Spotlight Trace Output Differencing
- Spotlight Instance Dump Differencing
- Test Program Output Capture (stdout, stderr)

### Trace Data Differencing

Proper regression test techniques require initial validation of test results to create a "gold" file – test execution output that is deemed "good". For example, if the application is expected to print out something to a file, then the output from the test case must be compared with the expected data in the gold file. This comparison requires the ordering of the trace points to be exactly the same in both files (new results and gold results). It is possible that non-deterministic event ordering may result from timing variations. Race conditions could be reflected if some events/incidents are traced, even though both paths may result in the same output and final state of the system. To ensure accurate results in the automated differencing of the user can control which events are traced to ensure only significant verification information is traced and differenced, and non-essential information is not traced that may vary from test to test. The marking **TestTrace** ("EXCLUSIVE", "AUTO", "SUPPRESS", default "AUTO") can be used to select a subset of classes, events and services to trace. By default, all classes, events and services with **TestTrace** set to AUTO will be traced. If any class, event or service has **TestTrace** set to EXCLUSIVE, then only those events with **TestTrace** set to EXCLUSIVE will be traced. If an event, class, or service has **TestTrace** set to SUPPRESS, then it will not be traced in any case. In Phase 2 the management of different sets of TestTrace markings for different test suites is managed by attaching different property sets to the test suite in the TestManager application.

These markings are fed into the trace difference tool, which will provide a report based on the tracing selections, and provide a higher level assessment of the differences from the current test results and the expected results, including timestamp handling, and order comprehension.

Example of a message from a trace difference report:

"Difference from Base: The operation VH.Car.backOut() was not called at trace step 15."

### Instance Data Differencing

Differencing of an instance dump required special processing beyond a simple textual comparison, especially under rename conditions. Instances with unique identifiers will be validated first. Associated instances will be checked next, based on matching attributes across the set of current vs. gold values. Attributes not in both dumps will be ignored.

Instance dumps will have the same format, and Spotlight understands instances. Spotlight Trace files also have a consistent format. Application specific output formats will vary widely, and can only be managed at the file level, with a generic "diff" utility to determine if the regression and actual versions are the same.

Like Trace differencing, instance data differencing can be scoped by domain or class using an **InstanceDiff** marking with values of AUTO, EXCLUSIVE, SUPPRESS, with AUTO as the default. By default, all domains and classes are AUTO and are included in an instance comparison. If any class or domain is set to EXCLUSIVE, then only the classes with EXCLUSIVE set will be included. SUPPRESS can be used to explicitly remove classes and domains from the comparison. This feature is useful from a performance perspective to constrain the comparison, and from an incremental development perspective, where instances in another domain or subsystem may not be relevant to the test.

Attributes also can carry the **InstanceDiff** marking, with its scope only to the comparison of instances of the class. If SUPPRESSED is set on an attribute, then the attribute values of instances will not be included in evaluation equivalence.

The currentState is also included in the comparison and can be suppressed as the attributes.

This depends on the instance editor capability in the console for display of differences in instance dumps.

### **Program Output Differencing**

For data that is sent to stdout or stderr like printf(), this data will be collected by the test manager as part of the test results. Test commands to support post processing of the data will be available to write the captured data to a file and execute system commands such as perl and diff on them.

Data sent directly by the component to files will not be visible to Spotlight or the Test Manager. The user must write the test script to manage these through the operating system escape feature. The user may use the escape to the system shell to invoke utilities such as perl or diff.

For Phase 1, instance and trace file differences will be reported in a textual format similar to the Unix diff utility. Phase 2 will display differences in the instance editor, as a side by side comparison, with differences highlighted and indexed to go to next and previous difference, in a manner similar to the MS Visual Studio Windiff utility.

#### **Effort Estimate:**

1. Instance file diff algorithm: 4 days
2. Instance diff GUI display: 4 days
3. Trace file diff algorithm: 2 days
4. Trace file diff display: 1 day
5. Stdout and stderr data collection: 1 day



## Test Instrumentation Interface

For more test scenarios not covered by the Spotlight test script capability, the Model-Level Test and Debug interface will be provided. This is a C++ interface built on top of the same mechanisms Spotlight uses to connect and control the application via its instrumentation. The interface conforms to the OMG standard interface to UML based instrumentation.

The interface can be used to build custom test and debug capabilities with the instrumentation.

This is a wrapper around the back end that Spotlight uses to communicate with the application instrumentation. With it, you could build a simply test driver, debug monitor, or a complete custom test/debug facility. By using this standard interface no instrumentation or other code changes are required on the application or instrumentation side. It will work with applications in all language maps.

**Effort Estimate:** 6 days

In development now as part of internal development. But may be deferred given MDT tech note priorities. This has been allocated to phase 1, but is not part of the effort estimate since it is already committed.

## Test Driver Domains

Test cases can also be modeled in UML as elements that drive the system in test-specific ways. This approach provides added flexibility when combined with Spotlight. For example, in an air traffic control application, the test cases would need to simulate the movements of planes within the airspace, with periodic updates to each planes' position.

A generic model of a plane's movement could easily be created. Variation points for test cases, such as number of initial plane objects with speed and position data, update rate, new plane entry rate, would then be included in the model. Thus, test cases would vary by the population of the domain under test and the test driver domains.

Test driver domains will be marked as such and only be generated and built for the test configurations that require them.

The ability to develop test drivers using models is a current capability, and no further support of this feature is planned. It is discussed here as an option for developing tests.

**Effort Estimate:** 0 Days

# 3. Phase 1 Feature Implementation

## Spotlight

### *Test coverage mode*

Spotlight will add a test coverage mode, When in this mode, when break and trace points are triggered, Spotlight will record the break location and model

element, then clear the breakpoint for faster execution. This mode is continued across test cases, and the cumulative data is stored in a file.

### **Test coverage report**

Spotlight produce a textual report showing the covered and uncovered model elements of the model.

### **Capture/Replay**

Spotlight will be extended to include Capture/replay

### **Instrumentation**

No changes to instrumentation have been identified. The Tech Note on Code Coverage offers a light weight instrumentation approach to collecting coverage data.

Currently, if Spotlight Instrumentation is enabled, optimization of generated code is off. This keeps all model elements, operations, classes, etc., accessible for model testing, but may not accurately reflect the desired deployed code. This limitation may be addressed in future releases.

However, this does affect how interpreted PAL operates for test cases, in that some of the model elements used in interpreted PAL may not be available. In this case, the test case, upon detecting the unavailable element, will cease executing and the Verdict status will be set to Unknown, to indicate the test case could not be run.

### **Maps**

The Maps will support a domain marking **TestDriver** ("FALSE", "TRUE", default "FALSE") indicating the domain is a test driver. If building a release mode version of the application, this domain will be excluded (suppressed).

## **4. Phase 2 Feature Overview**

The primary goals of Phase 2 are to:

- Improve that manageability of large amounts of test cases and related data through test suite management
- Increase the controllability and flexibility of test execution control, driving and custom test response through interpreted PAL.

### **Test Suite Management**

A test suite is the set of tests cases applied to a system or a component to determine correct operation of the component with respect to requirements, both functional and non-functional (performance). PathMATE proposes to extend the Console to include a Test Manager to manage the test suite.

A test suite manager will group the test cases for easier management. The groups will be managed hierarchically. Test cases may appear in more than one group across association A2. For example, functionality targeted by a test case may overlap with other test groups (within association A1).

Test execution is broken up into phases: initialization, stimulus, data collection, and verification. The test manager and Spotlight will support configuration settings and automation for all of these test phases. Initialization settings including the initial instance population and the stubs which emulate the component's environment. Stimulus is the set of inputs to the component under test and can also include breakpoint settings for applying stimulus at a specific point during execution. Data collection includes the tracing of component execution through trace points, examination of specific instances and attribute values at break points, instance dumps, and collection of application text outputs. Verification includes the comparison of instance dumps and expected outputs to actual test results.

A *test configuration*, as defined in the OMG Standard for the UML Testing Profile, is the collection of test component objects and of connections between the test component objects and to the SUT. In terms of this tech note, a test configuration consists of the various test settings that are applied to a test case. Test cases may share configuration settings, including instance population, break/trace point settings, stubs, and stimulus pieces of the Spotlight driver files.

A test case uses a collection of test settings, defining the system under test, the inputs, data collection instructions, and expected outputs. A test setting defines the sets of stubs, Spotlight test driver, initial instance population, and expected outputs.

The test case also includes test outputs and expected results. Outputs of the form of traces and instance dumps are recognized and handled as described in sections below. Other, non-Spotlight related test results, such as custom printed traces can be managed and differenced on as text files. Test cases may invoke other test cases using a `<callscript>` test driver command. A test case may have one and only one verdict. Test cases are identified by the *name* attribute of the `<script>` driver section.

A test session is the period Spotlight is connected to the system under test. Multiple test cases may be applied during a test session. Test cases may be chained together manually using test driver commands or controlled through the test suite manager. The test suite manager will manage sessions and sequence test cases for Spotlight to execute. The verdict for the test session is computed from the test cases executed. If all test case verdicts are set to either PASS or UNKNOWN, then the test session has a value of PASS. Otherwise, if any test case has a verdict of FAIL, the session's verdict is FAIL, otherwise, if any test case session is set to ERROR, then the session's verdict is ERROR.

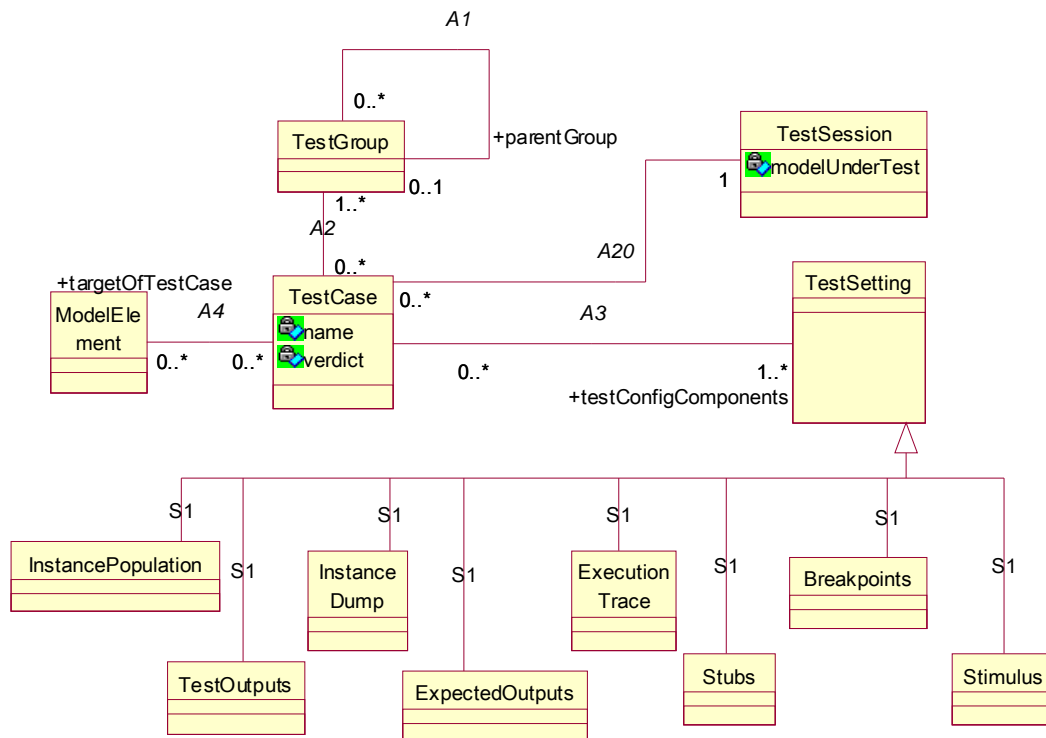


Figure 1

### Test Setting Management

Breakpoint settings may be grouped to be reused across test cases. If each test case uses one Spotlight driver file, then those batch files can refer to common trace settings in other files. A test manager component of the console will be developed to manage named break/trace setting groupings.

The GUI for managing the test settings for a test case will allow individual test settings to be selected and configured. Test cases will appear in a browser tree control. Settings can then be added to the test case by choosing them from lists. For example, zero or more initial instance population files may be added to the test case to perform the initial setup. Break and trace point files and IPAL stimulus settings can also be added. By default, one stimulus “GO” command is attached to each test case to start the system.

Timestamps will be taken on trace points, but not used to compare traces, as it will vary from one run to the next.

**Effort estimate:** Included under in management of test setting data and file design and implementation.

### Effort Estimate:

1. Rename of test case files: 6 days
2. Test Manage Console GUI: 5 days
3. TestSetting Management Design: 4 days
4. Test Setting data management implementation: 2 days
5. Test Setting Editor GUI – Spotlight : 3 days
6. Test Management Groups: 2 days
7. Stub management: 2 days

## Executive Control

Interpreted PAL can manage individual test execution and examine test results. This lets the test developer write tests in the same language as the model. PAL statements are added to the test driver file in places where the simple stimulus and data collection features of the scripting language is not enough. IPAL is syntactically the same as PAL, written for a domain operation. IPAL can invoke operations, generate events, access attribute data and traverse associations. IPAL will also has additional capabilities not provided in standard PAL:

- Switch Domain Context with `SELECT_DOMAIN <domain prefix>`
- Inspect Active Object Current State with `<instance ref>.currentState`
- Invoke command line programs with `<string> = SystemInvoke (<string command line>)`
- Access events and event parameters
- Define test script operations
- Sleep command
- Echo – print a character string to the standard out stream
- ON BREAK – used to pause test script execution until a condition in the application occurs.

Interpreted PAL can be used to do data collection and test stimulus. By allowing interpreted PAL to be executed on initialization, test setup can be performed. Executing IPAL at breakpoints can be used to control the application of stimulus with respect to the internal execution points within the application. For verification, IPAL can extract the resulting internal state and compare against predicted or computed results.

An additional capability will allow the script to invoke an operating system shell and invoke commands, such as a perl script. The script will capture the shell command output, much like the FILTER command of the Engine template Language.

IPAL will require some additional constructs not available in PAL. For example, IPAL needs a construct to access the state of an object. Since a test may be applied to many modeled domains at the same time, IPAL will require a statement to set the domain scope of the IPAL statements. Thus, all IPAL statements will be considered to be part of some domain scope.

In addition, the UML editor provides a way to create operations and parameters for domains and classes. For reuse purposes, IPAL will need to be able to define operations with parameters. These test operations are considered to be at system scope, and may be called anywhere from IPAL statement blocks.

IPAL scripts may require some services only available in the test driver command syntax. As these features are identified, these will be packaged as operations on a TestExecution domain, available only to IPAL scripts. The current test script commands to be supported in IPAL are:

- go
- pause
- setVerdict
- getVerdict
- writeLog

Test scripts will need to have access to operate a timer on the test script side, eg. to wait 5 seconds before applying the next stimulus. In a sequential test script, a Sleep(miliseconds) function will provide this functionality.

Spotlight4 format driver files format can be integrated with and referenced from scripts for backward compatibility. This build on the batch command capability in the Spotlight Tech Note. **Please see Appendix A for an example XML file containing interpreted PAL.**

### **Extended Breakpoint Information**

Extended breakpoint control to access information on the type and location of the breakpoint trigger within the test driver script. Different information is available for query based on different breakpoint contexts. For example, within the context of an Object Create breakpoint, no Association information is available because an Object Create does not involve associations.

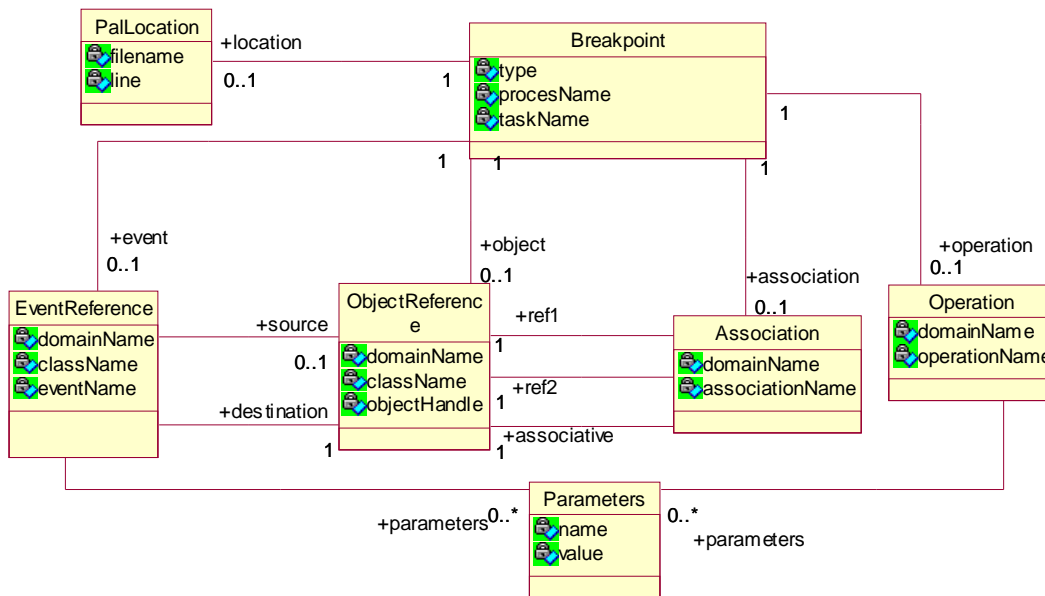
Breakpoint information available to the IPAL is shown in the following table. Fields are shown across the top and breakpoint types are shown on the left. X denotes that the field applies to the breakpoint type.

	<b>Fields available in IPAL for each type of breakpoint</b>						
<b>Breakpoint type</b>	Type	PAL Location	Object Reference	Event Reference	Association	OperationName	Parameters
Object Create	X	X					
Object Delete	X	X	X				
Event Send	X	X	X	X			X
Event Receive	X		X	X			X
Object Transition	X		X				
Link	X	X	X		X		
Unlink	X	X	X		X		
PAL Statement	X	X	X				
Operation	X		X			X	X
Transition	X		X	X			X

**Table 1. Breakpoint Data**

The class diagram below defines the fields available to IPAL on breakpoints. A singleton instance of the Breakpoint class, referred to as *break*, holds the last triggered breakpoint information for the current task. Additional information is contained in the fields of the Breakpoint and associated classes.

The IPAL syntax for accessing this information is obtained through the attributes and role phrases within the diagram. For example, *break.location.line* accesses the line number of the current PAL breakpoint. The *type* defines what information is available according to table 1.



**Figure 2. Breakpoint Class Diagram**

Break and idle conditions are notifications of an occurrence of an incident within the application. In a test script, these incidents are used to synchronize the application with some test stimulus to control the execution flow. As such, within a test script, breakpoints and idle conditions are expected, and expected at a particular point. Execution of the test script pauses until the prescribed condition is met, then continues.

Within an ON BREAK condition, an IPAL statement block is executed. The IPAL may gather information about the breakpoint and the current state of the application, and induce stimulus.

Here is an example of how more breakpoint information is accessed to allow the script to evaluate the breakpoint information.

For example:

```

ON BREAK
{
  IF (break.type == TRANSITION)
  {
    IF (break.object.className == "Printer")
    {
      // ... IPAL...
    }
  }
}
  
```

```
}
}
```

### Using IPAL to Achieve Conditional Breakpoints

Conditional breakpoint capabilities can be approximated with the breakpoint type and logic capabilities in IPAL. .

```
ON BREAK
{
IF (break.type == Operation && break.operation.operationName == "runningSum" &&
break.operation.parameters["param"] == 50)
{
// extract information
DUMP INSTANCES;
}
ELSE
{
GO;
}
}
```

In this case, the GO command instructs the application to continue processing, and resets the break so that upon the next breakpoint notification, the same ON BREAK IPAL statement block is executed. If the GO command is not encountered, then execution continues after the IPAL block.

#### Effort Estimate:

1. IPAL Parser (leverages current PAL parser): 2 days
2. IPAL Execution Engine: 11 days
- IPAL PAL Extensions:
3. Current State Access: 1 day
4. IPAL Operations: 4 days
5. Set Domain Context: 3 days
6. Sleep script command: 1 day
7. Extended Breakpoint Information: 5 day

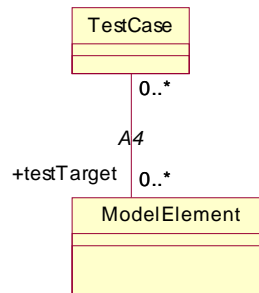
### Test Coverage - Test Target Elements

As part of this specification select parts of the model are identified as a target of a test case during a test suite, either to drive or to validate proper operation - usually domain operations or classes, but can refer to any model element. Any model element can be specified as a target of the test. If the target model element is a class, the changes to the class attributes, operations, statechart, or action language would be considered a change.

Target elements are identified explicitly by test case designers, or implicitly through the stimulus they use. For example, if a test case uses a domain operation in its stimulus, then the operation is added to the set of target elements for the test case.

The markings will provide traceability of the tests to the model elements they target. Test cases and test groups can be attached to visible element markings.



**Figure 2**

Sample report:

Class FP.RecipeSpec changed on 3/14/05 06:32, and required the following test cases to be rerun:

Xx\_yy

Cc\_oo\_pp

#### Effort Estimate:

1. Console GUI: 2 days
2. Map Test Cases to Model Elements: 2 days

## 5. Phase 2 Feature Implementation

### Console

The Console is extended to include test management capabilities.

- Test case creation and documentation
- Management and configuration of test case components (instance populations, break/trace settings, etc.)
- Grouping of test cases
- Mapping of test cases and groups to model elements
- Mapping of test cases to requirements
- Detection of changed elements and execution of related test cases
- Build management for test environment, stubbed domain set or real domain implementations
- Test result and coverage reporting
- Difference instance dumps and trace logs

The console will be extended to include these test management capabilities. The test manager will coordinate Spotlight's instructions per test case. The driver file will be broken up into multiple modules to enable reuse of pieces such as trace settings and instance populations. The actual file structure is left to a design decision. The individual test settings may be stored as individual files using external links and references supported by XML, or

managed in one file with internal links and references. The overall requirement is to make it easy for the test developer to reuse test settings.

The method of communication between the Test Manager and Spotlight is still subject to design decisions, but possibilities are:

- Test manager manages the separate driver files for each test case and invokes Spotlight with the correct file per test case.
- Test manager manages the data internally, generates the driver files to invoke Spotlight.
- Test manager communicates with Spotlight dynamically at run-time, though a COM-like interface.

If a model fails to translate and compile cleanly, the console should be able to detect the build error. Otherwise, Spotlight will be invoked to perform the test, but fail to successfully start the application.

## Spotlight

Spotlight will be extended to include these features:

- Edit and manage Interpreted PAL scripts
- Driver file partitioning : Driver files are made up of distinct parts, such as break and trace point settings, stimulus, instance populations, etc. Each of these parts will be individually managed (but file design is not final), so they may be mixed and matched to be reused across test cases. See the TestSetting class in the diagram in section 2.1

## Editor Integration

Editor integration will be extended to display test coverage results as a view of the models. Just like state animation does now, PathMATE will highlight states, transitions, and operations that are not covered by any test cases. This feature is part of the Eclipse integration and is enabled when using the Test Manager perspective when there is a model coverage file in the project.

## Rename of Test Cases

Test settings that are inputs to a test case, namely instance populations, stimulus, stubs, and breakpoints, must handle rename as current PAL rename works. Rename or deletion of attributes, classes, operations, associations, events or parameters will check the PAL files but also the test cases for the model.

Sample use case: An attribute is removed from the model. Rename capability built into the test manager would update the test settings including instance population, expected instance dumps, break and trace points. IPAL would also be examined, with warnings given recommending manual intervention. Rename of an attribute or class would update the IPAL directly.

## 6. Beyond Phase 2 - Good Ideas for Related Development

This section describes ideas for future development related to model driven testing that are *not* scheduled for Phase 1 or 2 delivery. These items are not in the estimates or delivery schedules, but are here as place holders and discussion points so that these concepts do not get lost.

## Other Integration

Test case management should include mappings of test cases to the requirements they are meant to test. Integration with existing tools, like DOORs, and evolving standards like SysML, the extension to UML for systems engineering will be examined.

When used with tools that support mapping of model elements back to requirements, mapping test cases back to requirements can then be followed forward to the dependent model elements.

## PathMATE Extensions

Plug-ins are found throughout applications that integrate with Eclipse. Plug-ins allow extension points to be defined for applications, so sophisticated users like Canon could extend functionality of Spotlight, for example. Much like the way Rose provides Visual Basic for extending behavior, Eclipse tools define plug-ins written in Java. We have not yet evaluated how we will use Eclipse-style plug-ins within PathMATE.

## Detailed Test Coverage

If the breakpoint were not cleared using trace points instead), Spotlight could count the number of times each breakpoint is triggered and provide more detailed coverage information for each breakpoint. This is a variation of coverage mode which records how many times a breakpoint is triggered. For example, a PAL statement may be triggered 37 times within the test suite, but further analysis reveals that 35 of those times is from test case 12.

## Test Dependency and Auto-Run

When incremental update is supported in the console, the test manager can detect the changed model elements and automatically run the test cases that target the changed model elements. This will be dependent on the capabilities built on the Eclipse framework.

## Test Driver Editor

Test driver files are stored as XML files. They include combinations of setup, stimulus, breakpoint, and data collection test settings. There exists an editor in Spotlight to set breakpoints, but only a text editor or XML editor is available to support all test script features. A future enhancement would add a GUI editor to make it easier to edit all parts of the test settings.

# 7. Test Script Syntax

This section describes the syntax for the new commands in the XML script driver. These commands are built on the commands described in the Spotlight User Interface tech note and the syntax described in the Test Driver tech note.

## Halt Script Processing

This command terminates test script execution, but leaves the application running. If Spotlight is in batch mode, it exits. If Spotlight is in interactive mode, it waits for user input.

```
<cmd cmd="halt" />
```

## Shutdown Application

This command will shutdown the application. If the application is a distributed deployment consisting of multiple processes, all processes will be shut down.

```
<cmd cmd="applicationShutdown" />
```

## Exit Batch Processing

This command will stop processing for the current test case and force Spotlight to exit, stopping all further test processing.

```
<cmd cmd="exit" />
```

## IPAL

Interpreted PAL expressions within a test driver appear within `<ipal>` `</ipal>` XML tags. Any local variables defined within the IPAL tags are scoped to that block of IPAL statements. IPAL expression blocks are standalone blocks. They accept no inputs and provide no outputs.

```
<ipal>
```

```
    // IPAL Expressions here
```

```
</ipal>
```

IPAL blocks can be contained only within `<script>`, `<onBreak>`, `<onIdle>`, or `<testOperation>` tags.

## Test Script Operation Definition

Test operations are common functions that may be used by multiple test cases. Test operations are scoped initially to the system level, and can be called from any IPAL statement block. Test operations contain one and only one IPAL statement block.

Test operations can accept input parameters and provide output parameters using the parameter definition tags within the testOperation block. Parameters must appear before the operation's IPAL statement block. All XML attributes are required for parameters.

- name – the name of the parameter
- type – the data type of the parameter. Can be a user defined type from the model.
- Mode – input or output

```
<testOperation name="opName">
  <parameter name="argName" type="dataType" mode="output"/>
  <ipal>
    // IPAL statements
  </ipal>
</testOperation>
```

## Verdict

Values for the test status are Pass and Fail. If the status is not set, the value will be Unknown. If Spotlight detects an error while executing the test case, the verdict is set to Error.

```
<cmd cmd="setVerdict" value="pass" />
```

The verdict can also be set through an IPAL instruction, `setVerdict(PASS);`

The verdict can also be read using `getVerdict`.

Since a the verdict may be set many times within a script, Spotlight manages the verdict according to the following algorithm, taken from the OMG's UML Testing Profile. The enumerated values for the verdict are set in order.

- UNKNOWN
- PASS
- FAIL
- ERROR

The verdict is only updated to an equal or lower value in the above order. For example, the verdict starts out UNKNOWN and is then set to PASS. If it is set to FAIL, subsequent attempts to set it to PASS will leave the verdict unchanged at FAIL.

The verdict is reset when a new script is started.

## Output Comparison

Compare files. There are 3 types of differencing supported, trace, instance dumps, and text. File1 and file2 are the files to compare. Resulting differences are stored in the output file. If differences are found and `updateVerdict` is set to true, the test case is flagged as Failed. If `updateVerdict` is false, then the test status remains unchanged. Maintenance of the verdict is described in section 7.6.

```
<cmd cmd="diff" type="trace | instances | text" updateVerdict="true"
file1="" file2="" output="" />
```

## 8. Functionality Release Schedule

The first release of functionality will be available in Feb 05, with releases every three months to lessen the overhead on the Pathfinder side to prepare and test each release, and on Canon's side to receive and upgrade. Of course, urgent fixes and patches will continue to have highest priority and turned around quickly.

Many of the features are dependent on the console and/or Eclipse integration features, which is tentatively slated for January availability.

Below is an estimate of feature availability.

## Phase 1 Development and Release Plan

**Jan 05** (total 19 days)

Test Coverage:

1. Breakpoint Driver File Generation: 1 day
2. Spotlight Coverage Mode: GUI 3 days, Back end 2 days
3. Coverage Data Storage: 1 days
4. Report Generation: 2 days

Test Execution:

1. Test Execution/Spotlight Integration: 3 days
2. Result display: 1 day

Test Suite Management: (foundation activities for Phase 2)

3. Test Setting Management Design: 4 days
4. Test Setting data management implementation: 2 days

**March 05** (total 27 days, excluding 6 days for Test Interface free - not to be charged in any PO)

Replay Traces:

1. Spotlight Capture Mode : 2 days
2. Spotlight Replay Mode: 2 days
3. Instrumentation changes: 4 days
4. Trace file XML format: 1 day

Test Coverage:

5. Editor Coverage Display (depends on Eclipse integration) : 3 days

Test Output Support Tooling:

1. Instance file diff algorithm: 4 days
2. Instance diff GUI display: 4 days
3. Trace file diff algorithm: 2 days
4. Trace file diff display: 1 day
5. Stdout and stderr data collection: 1 day

Test Interface: 6 days

Test Suite Management: (foundation activities for Phase 2)

5. Test Setting Editor GUI – Spotlight : 3 days

## Phase 2

Estimates, features and development plan for phase 2. The IPAL development effort is bundled together tightly for the initial parser and interpretation engine for the support of current PAL actions. IPAL extensions will be developed afterward, but are dependent upon the core IPAL parser and execution engine.

The test suite management will be developed and released incrementally.

Items in *Section □*, *Driver* file partitioning : Driver files are made up of distinct parts, such as break and trace point settings, stimulus, instance populations, etc. Each of these parts will be individually managed (but file design is not final), so they may be mixed and matched to be reused across test cases. See the TestSetting class in the diagram in section 2.1

*Editor Integration*

*Editor integration* will be extended to display test coverage results as a view of the models. Just like state animation does now, PathMATE will highlight states, transitions, and operations that are not covered by any test cases. This feature is part of the Eclipse integration and is enabled when using the Test Manager perspective when there is a model coverage file in the project.

#### *Rename of Test Cases*

*Test settings* that are inputs to a test case, namely instance populations, stimulus, stubs, and breakpoints, must handle rename as current PAL rename works. Rename or deletion of attributes, classes, operations, associations, events or parameters will check the PAL files but also the test cases for the model.

Sample use case: An attribute is removed from the model. Rename capability built into the test manager would update the test settings including instance population, expected instance dumps, break and trace points. IPAL would also be examined, with warnings given recommending manual intervention. Rename of an attribute or class would update the IPAL directly.

Beyond Phase 2 - Good Ideas for Related Development are not included here.

#### **June 05** (total 27 days)

##### Executive Control - IPAL Extensions:

1. IPAL Parser (leverages current PAL parser): 2 days
2. IPAL Execution Engine: 11 days
3. Current State Access: 1 day
4. IPAL Operations: 4 days
5. Set Domain Context: 3 days
6. Sleep script command: 1 day
7. Extended Breakpoint Information: 5 day

#### **Sept 05** (total 19 days)

##### Test Suite Management:

1. Rename of test case files: 6 days
2. Test Manage Console GUI: 5 days
6. Test Management Groups: 2 days
7. Stub management: 2 days

##### Executive Control - Test Target Elements:

- 1 Console GUI: 2 days
- 2 Map Test Cases to Model Elements: 2 days

## Appendix A – Examples

The following example shows a Spotlight driver file in XML format.

### Sample Driver File as XML

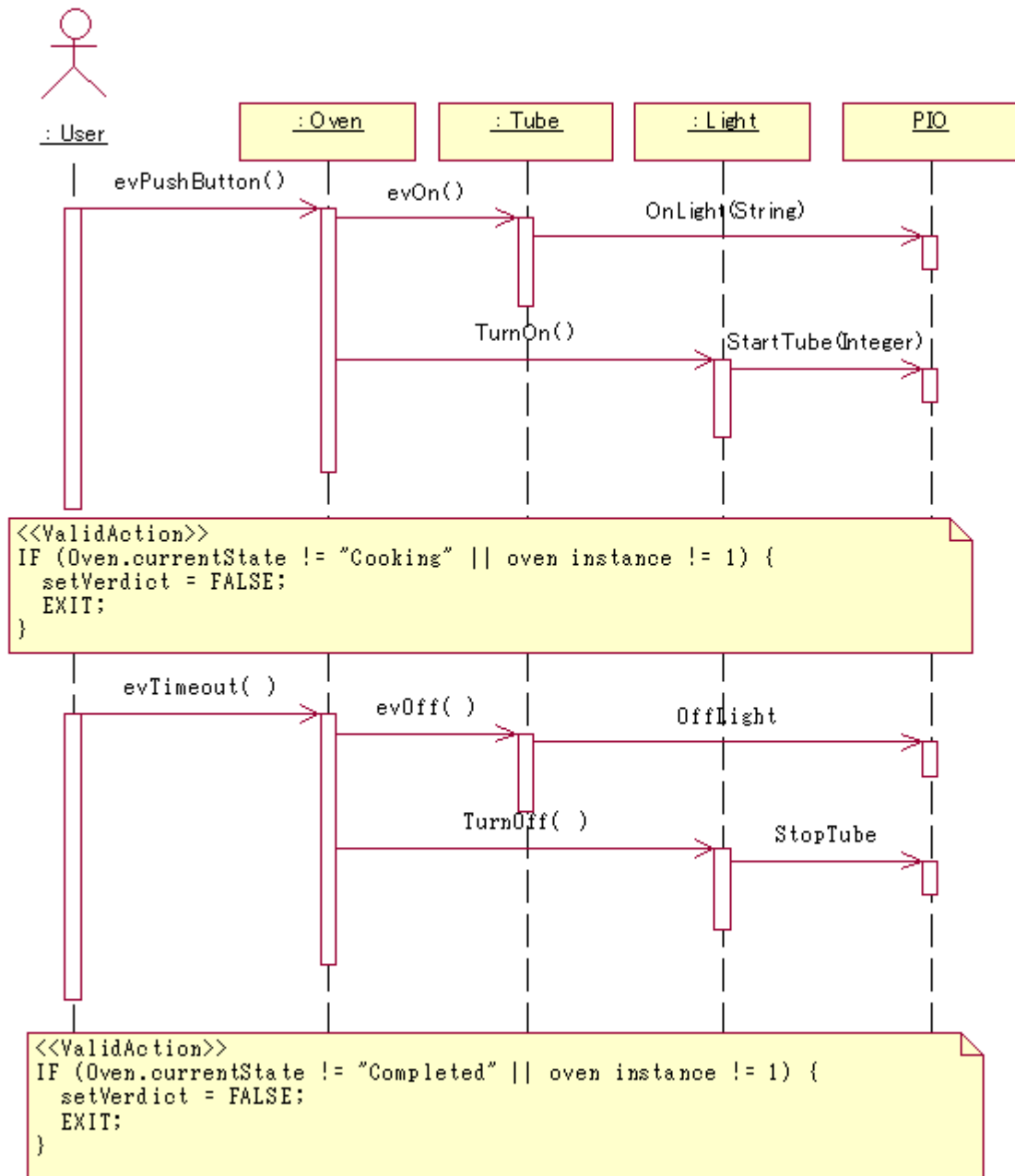
```
<comment>
This is a sample test script. The actual format, structure, XML
schema, are not finalized and subject to change.
</comment>
<process filename="recipe_ove.exe" id="localOven">
  <task name="SYS_TASK_ID_MAIN" port="5150">
    <breakpoints>
      <point break="true" trace="false" category="class"
type="transition" subject="Oven"/>
      <point break="true" trace="false" category="class" type="create"
subject="Oven"/>
    </breakpoints>
  </task>
</process>
<script name="sampleTest">
  <ipal>
    startOven(5000); // call to locally defined test operation
    SELECT_DOMAIN Oven; // set context to Oven domain
    Ref\<Oven\> o = FIND CLASS Oven;
    ECHO o.name; // printf to stdout
  </ipal>
  <onbreak type="transition" subject="Oven">
    SELECT_DOMAIN Oven;
    Ref\<Oven\> o = FIND CLASS Oven;
    GENERATE Oven:DoorOpen() TO (o); // local variables may not carry
over ipal scope
  </onbreak>
  <onIdle>
    <!-- test complete when app goes idle →
    <cmd cmd="dumpPopulation" scope="domain" filename="instances.xml"
name="RecipeOven.Oven" />
  </onIdle>
  <cmd cmd="diff" type="instance" updateVerdict="true" file1="
instances.xml" file2="expected.xml" output="ovenInstanceDiff.xml"/>
</script>

<testOperation name="startOven">
  <parameter name="delayTime" type="Integer" mode="input"/>
  <parameter name="ready" type="Boolean" mode="output"/>
  <ipal>
    SELECT_DOMAIN Oven;
    Sleep(delayTime);
    Oven:StartOven();
  </ipal>
</testOperation>
```



## Example 2- Recipe Oven Test

The scenario shown in the sequence diagram is achieved with the test script below. The <<ValidAction>> text is reflected in the test script, with the exception of the EXIT.



```

<process location="localhost" executable="recipe_oven.exe"/>
<task name="MAIN" port="5150">
    <point break="true" trace="false" category="operation"
type="invoke" element="PIO.StartTube" uuid="" />
    <point break="true" trace="false" category="operation"
type="invoke" element="PIO.StopTube" uuid="" />
</task>
</process>

<script name="MasterScript">
    <callScript name="Start"/>
    <callScript name="Finish"/>
    <cmd cmd="applicationShudown" />
    <cmd cmd="exit" />
</script>

<script name="Start">
    <ipal>
        SELECT_DOMAIN Oven;
        Ref\<Oven\> o = FIND CLASS Oven;
        GENERATE evPushButton() TO (o);
        ON BREAK
        {
            Integer ovenCount = 0;
            countOvens(ovenCount);
            IF (break.type == Operation && break.operation.operationName ==
"StartTube")
            {
                IF (o.currentState == "Cooking" && ovenCount == 1)
                {
                    TestExecution:setVerdict(PASS);
                }
                ELSE
                {
                    TestExecution:setVerdict(FAIL);
                }
            }
        }
    </ipal>
</script>

<script name="Finish">
    <cmd cmd="go" />
    <ipal>
        SELECT_DOMAIN Oven;
        Ref\<Oven\> o = FIND CLASS Oven;
        GENERATE evTimeout() TO (o);
        ON BREAK
        {
            Integer ovenCount = 0;
            countOvens(ovenCount);
            IF (break.type == Operation && break.operation.operationName ==
"StopTube")
            {
                IF (o.currentState == "Completed" && ovenCount == 1)

```

```
{
    TestExecution:setVerdict(PASS);
}
ELSE
{
    TestExecution:setVerdict(FAIL);
}
}
ELSE
{
    GO;
}
}
</ipal>
</script>

<testOperation name="countOvens">
    <parameter name="count" type="integer" mode="output"/>
    <ipal>
        SELECT_DOMAIN Oven;
        Integer count = 0;
        FOREACH o = CLASS Oven
        {
            count = count + 1;
        }
    </ipal>
</testOperation>
```

### Example 3 - Robochef

This scenario shows a test script to execute the Robochef sample model. It startup up the model by invoking a service, selecting a recipe to make and monitors the OpFailed event and sets the verdict to FAIL if the event is received.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Driver file for PathMATE Spotlight (http://www.pathmate.com)-->
<spotlightDriver xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://www.PathfinderMDA/spotlightDriver
.xsd" name="Robochef" version="5.01.002"
modelFile="../../analysis/Robochef.mdl" >

  <process filename="../../rose/project/cpp/debug/robochef.exe"
id="Robochef">
    <task name="SYS_TASK_ID_MAIN" port="5150" >
      <script>
        <point break="true" trace="false" category="event"
type="receive" element="Robochef.FP.APPL.OpFailed" uuid="" />
      </script>
    </task>
  </process>

  <script>
    <cmd cmd="openfile" type="log" filename="Robochef.log" />
    <ipal>
      SELECT_DOMAIN FoodPrep;
      FoodPrep:StartRecipe("Flour and Water");

      // loop, watching for the end condition and monitoring failures
      Boolean atEnd = FALSE;
      Integer failCount = 0;
      WHILE (atEnd != TRUE)
      {
        ON BREAK
        {
          IF (break.type == Event && break.event.eventName ==
"OpFailed")
          {
            TestExecution:setVerdict(FAIL);
            failCount = failCount + 1;
          }
          ELSE IF (break.type == Idle)
          {
            // event loop done, and no more to do
            atEnd == TRUE;
          }
          ELSE
          {
            GO;
          }
        }
      }
      IF (failCount > 0)
```

```
        {
            String failMsg = "Fail event detected " + failCount + "
times";
            TestExecution:writeLog(failMsg);
        }
    </ipal>

    <cmd cmd="dumpPopulation" scope="system"
filename="robochefInstances.xml"/>
    <cmd cmd="closefile" filename="Robochef.log"/>
    <cmd cmd="diff" type="instance" updateVerdict="true"
file1="robochefInstances.xml" file2="expected.xml" output="diffs.xml"
/>

    <ipal>
        // compare the logs with external utility
        // diff results using windiff and save in diff.out, then read
in the file
        String result = TestExecution:SystemInvoke("windiff -Ssx
diff.out Robochef.log RobochefExpected.log");
        result = TestExecution:systemInvoke("type diff.out");
        IF (result != "")
        {
            // fail
            TestExecution:setVerdict(FAIL);
        }
        ELSE
        {
            TestExecution:setVerdict(PASS);
        }
    </ipal>
</script>
</spotlightDriver>
```