Writing Triggers in Perforce

S. Vance
J. Bowles

April 26, 2005

Abstract

Perforce has introduced new trigger types with the 2004.2 release. The triggers allow a new range of capabilities to maintain the development environment, enforce development policies, and implement development support services. This paper discusses several **techniques** for implementing trigger scripts and examines a several **purposes** for triggers ranging from form modification and validation to automatic integration and build scripts. This paper elaborates upon the Scripting with Perforce paper for the Perforce 2005 User Conference.

1 Introduction

Perforce introduced the first server-side trigger in release 99.1 with the pre-submit trigger. This trigger satisfied a long-standing desire in the user community, but demand continued for more hooks. In release 2004.2, Perforce squarely hit the need with the addition of five new trigger types. Release 2005.1 adds yet one more trigger type to this list rounding out one of the categories of triggers to completeness.

This paper discusses triggers, **techniques** for implementing them and **purposes** for using them. It presumes a general knowledge of scripting. The examples follow in several programming languages. They should be easy to follow with knowledge of general programming, and any more arcane constructs will be explained.

The paper also presumes a reasonable knowledge of Perforce scripting alternatives, such as that presented in [Bowles2005]. Although this paper will address the scripting of triggers comprehensively, it will refer to other Perforce scripting **contexts** and to Perforce commands with an assumption of familiarity.

1.1 What is a trigger?

Triggers are programs that run on the server immediately in response to some well-defined event in Perforce. Therefore, the **context** for a trigger is running on the server using the trigger mechanism to start.

Triggers are typically written in a shell script such as Perl, Python or Ruby due to the flexibility and facilities they provide. However, triggers can be written in any programming language that can interface with Perforce, including UNIX shell (sh, ksh, csh and work-alikes) and compiled languages like C/C++.

1.2 Types of triggers

Triggers fall into two categories. Pre-submit triggers enable actions in response to the submission of changelists. Form triggers allow actions in response to various stages of the life cycle of a form, regardless of the form type. This section provides a brief overview of the trigger types in preparation for the more detailed discussion.

1.2.1 Pre-submit triggers

There are three types of pre-submit triggers corresponding to different points in the life cycle of a submission.

- "Submit" triggers execute after the changelist has been created but before the files have been transferred, allowing inspection of the changelist details but disallowing file inspection.

- "Content" triggers execute after file transfer but before commit, allowing for inspection of the files.

- "Commit" triggers execute after the commit, allowing inspection of the changelist and file contents, but disallowing canceling of the submission.

1.2.2 Form triggers

Form triggers come in four types depending on the point in the form's life cycle in which they are invoked.

- "Out" triggers execute when the form is generated and can modify the form before it is presented to the user.

- "In" triggers execute when the form is sent back to Perforce but before it is parsed, also allowing modification of the form on its way in.

- "Save" triggers execute after the form has been parsed but before it is saved, allowing reaction to the form but not modification.

- "Delete" triggers execute before a form is deleted, allowing failure of the deletion.

1.3 Why use a trigger?

Knowing why to use a trigger is partially a matter of knowing what are the competing alternatives. The alternatives naturally come from other **contexts**, since triggers define a **context** of their own. This section details the salient operational characteristics of triggers and contrasts them with Perforce alternatives. The three primary alternatives to triggers are wrapper scripts, such as p4wrapper, [1] review daemons, and journal tailers. A variation on the wrapper script would be a script available from the Tools menu in P4Win.

---

[1] p4wrapper can be found in the Public Depot at //public/perforce/utils/p4wrapper.

1.3.1 Synchronous execution

Triggers execute synchronously in response to their associated event. This provides an immediacy of response that is sometimes required or at least highly desirable. One option that provides synchronous execution could be an action invoked from a wrapper script such as p4wrapper. Another option would be to forego synchronous execution and rely on frequently running review daemons. Journal tailers would also perform asynchronously, although with very rapid and event-driven response.

1.3.2 Immediate user feedback on error

Triggers can provide messages back to the user, but only on error. Messages are not delivered on successful execution. A wrapper script can deliver messages to the user regardless of whether an error occurs or not. Review daemons and journal tailers can only provide feedback through indirect mechanisms.

1.3.3 Enforceability

Enforceability refers to the ability of an administrator to ensure that the script will run regardless of the client program used to initiate the operation. Because triggers are installed on and executed by the server in response to server events, they will execute regardless of the client program. Wrapper scripts will only be invoked when the wrapper is used, whereas direct use of p4 or use of a different client program will circumvent the desired action. Review daemons and journal tailers are also enforceable due to their **context** on the server.

1.3.4 Modify a form

Form triggers can modify a form as it is delivered to the user or as it is sent back to Perforce. Wrapper scripts share this characteristic. Review daemons and journal tailers can not modify forms except to the extent that any user or administrator can after the operation has finished.

1.3.5 Customize any action

Form triggers have a limited ability to customize actions that involve forms, but they do not have the ability to react to any arbitrary command. Similarly, pre-submit triggers can only react to submits. Review daemons can only react to commands whose side effects can be reliably observed, something that is not readily available from the command line in many cases or from review mechanisms. Journal tailers have the ability to react to any action that affects database entries, which includes almost all.

1.3.6 Optimization for bulk processing

The review mechanism gives the ability to process a large number of actions in an orderly and efficient manner as long as those changes impact a counter. This optimization is not readily if at all available to triggers, wrappers or journal tailers due to their association with individual commands or journal entries.

1.3.7 Deterministic execution

Triggers and wrappers provide an exact and deterministic understanding of when they will execute relative to the command that initiates them. Review daemons are generally driven in a time-based manner and therefore do not execute deterministically relative to the Perforce command. Journal tailers are closer to deterministic than review daemons, but can conceivably execute prior to completion of a command.

1.3.8 Summary

The following table summarizes the characteristics of the scripting **contexts** that compete with triggers for Perforce scripting.

Characteristic	*Trigger*	*Wrapper*	*Review Daemon*	*Journal Tailer*
Synchronous	✓	✓		
Success feedback		✓		
Error feedback	✓	✓		
Enforceable	✓		✓	✓
Modify form	✓	✓		
Customize any action	partial	✓		most
Bulk optimization			✓	
Deterministic	✓	✓		

2 Trigger Techniques

This section discusses **techniques** used to create triggers of the different types. Some **techniques** have applicability to **contexts** other than triggers, but all are relevant to triggers. The following **techniques** build upon the **techniques** presented in [Bowles2005], but are commonly or exclusively used in the trigger **context**.

2.1 Exit codes

This is an almost trivially simple **technique** that applies solely to triggers. The trigger communicates whether the associated action is successful or not through its exit status. An exit status of 0 indicates success. Any other exit status indicates failure. The exact value of a non-zero exit status can not be meaningfully used. Some languages provide implicit non-zero exit status with common error handling facilities, such as `die()` in Perl.

```
if( $success )
{
     exit 0;
}
else
{
     die "Operation failed. Fix problem.";
}
```

Figure 1. Perl example of trigger exit status

2.2 Error messages

Ordinary scripting delivers all output messages to the caller. Triggers can only provide custom feedback to the user through messages that accompany error exit status. Messages output during script execution will be discarded when the trigger is successful. Figure 1 shows a message that will be sent back to the user on failure.

2.3 Multiple different actions on same files

With triggers you have the ability to invoke multiple triggers on the same object until one of the triggers fails. An object is a particular file or file path for pre-submit triggers, or a particular form type for form triggers. This is accomplished by defining multiple triggers with different names on the same object definition. Triggers fire in the order presented in the triggers definition.

```
success1 submit //... "ruby submit_trigger.rb"
failure1 submit //... "ruby always_fails.rb"
success2 submit //... "ruby another_trigger.rb"
```

Figure 2. Example of multiple submit triggers on the same path.

In the above example, the third trigger will never fire because the second trigger always fails.

2.4 Same action on multiple file patterns

A single trigger also has the ability to operate on multiple file patterns. Multiple lines with the same name as shown in Figure 3 are considered to be a single trigger.

```
doit submit //....h "python trigger1.py"
doit submit //....c "python trigger2.py"
```

Figure 3. Example of a single multi-line trigger.

Note that only the first command is significant. If a file matches the pattern for the second line, the command from the first line will be executed.

2.5 Tagged output

A **technique** that is applicable to all scripting is the use of tagged output in general or marshaled output from Python or, if comfortable with undocumented functionality, Ruby. With the appropriate global option flag, p4 will output the results of commands in marshaled object format for the chosen language. Figure 4 and Figure 5 show examples using Python and Ruby, respectively.

```
$ p4 –G label –o mylabel | processlabel.py
```

Figure 4. Example of Python marshaled object output.

```
$ p4 –R label –o mylabel | processlabel.rb
```

Figure 5. Example of the undocumented Ruby marshaled object output.

An example of handling marshaled output in Python is given in [Goldstone2005]. More information on tagged output is presented in [Bowles2005].

Tagged output eliminates the need to write stateful parsers to process commands.

2.6 Form modification

Another generic **technique** that is commonly used in triggers is form parsing and modification. Tagged output, discussed in section 2.5, simplifies the task. The essence of the **technique** is to use the -o flag to output the form and the -i flag to input the form. A coarse example using Python is shown in .

```
$ p4 –G client -o | transformit.py | \
     p4 -G client -i
```

Figure 6. High-level example of form modification.

2.7 Initiating follow-on processing

Although this **technique** is applicable to many **contexts**, it is particularly useful with "save" and "commit" triggers. Neither of these trigger types have the ability to modify forms or fail an operation. They faithfully execute after the completion of the operation, providing a perfect opportunity to initiate further processing. This processing must not be time-consuming, as they still prevent the operation from returning until they finish.

This approach is also useful when the follow-on processing would lock Perforce databases that would already be locked as a natural consequence of the triggering operation. Dissociating the processing from the initial operation avoids deadlock conditions.

There are two primary **techniques** to accomplish this with numerous implementations. The first **technique** starts the processing immediately but does not wait for completion as with launching a sub-process. The second **technique** requires a means of sending a signal or "opening the gate" by setting some state that another process will recognize as a cue to

process. Note that the latter **technique** includes review daemons, which involves another **context** and does not require cooperation from a trigger.

Specific implementations are left to the experience and ingenuity of the reader.

2.8 Coordinated trigger actions

Triggers execute in the following order: out, in, save, submit, content, commit. Obviously not all trigger types execute in all situations. Pre-submit triggers only execute on submissions. Out triggers do not execute when submitting a numbered changelist because the form has already been generated. The order of execution allows triggers to coordinate actions in a manner appropriate to their individual strengths and weaknesses.

As an optimization, a form trigger that was already processing a form could cache state information that would be used by a submit trigger. For example, certain reviewers may be required in certain areas of the repository or for certain users. The parsing of reviewer information could occur once during the in trigger and be used during the submit trigger instead of being re-parsed.

As another example, whether a build should occur may depend on the nature of the change recognized by a content trigger, but the policy for the decision to build or not could be encapsulated and acted upon in a commit trigger from where the build should be launched.

3 Trigger Purposes

This section presents several representative **purposes** for writing triggers. Providing a complete survey of trigger **purposes** would be impossible. This paper attempts to stimulate thought on the range of possibilities by showing canonical or interesting applications of each trigger type. Code for several of the examples will be presented at the conference, but are not included in the paper for brevity.

3.1 Ensure related files are submitted together

Since submit triggers have been available longer than the other trigger types, their use has been explored more fully. One of the canonical uses is to ensure that related files are submitted together rather than being allowed to track at different rates.

The key enabler is that the relationship between the files predictably or observably exists and that it can be easily discerned by patterns in the file naming or location. The naming requirement stems from the submit trigger's inability to examine the content of files.

Two common uses for this are

- To ensure that the implementation file is updated whenever the header file is modified in a language like C++. Every time a .h, .hxx or .hpp file is modified, its corresponding .c, .cpp, .cxx, .CC, .c++ file is also in the changelist.

- To ensure that test harnesses are updated whenever their corresponding source code is modified. In Java with JUnit, a class named MyClass.java should have a test case named MyClassTestCase.java. The test case class will generally be in a different directory tree to keep it separate from the production code, but the location and naming should be predictable, and the presence of the file is most likely required.

3.2 Verify all header files have company copyright banner

Some **purposes** require inspection of the files being submitted, which is accomplished with a content trigger. A common use for content triggers checks header files to ensure that company copyright information has been included and preserved in the file. This is particularly important when the headers are being distributed as part of an API for a product.

In a language like Java, headers usually take the form of interface classes, leading to a file pattern for the trigger of "//.../I*.java". This may catch more than just interface classes, leading to an additional use for the content inspection to check whether the class is truly an interface or just a class whose name starts with "I."

In a language like C/C++, headers may have multiple extensions. This may be to distinguish between C and C++ files or to distinguish between header files for different platform compiler conventions. Directory locations may also be used to identify the relevant files. In any case, application of the **technique** from section 2.4 applies the same action to each of the different file patterns.

3.3 Reject submissions with zero-content deltas

Another application of content triggers would be to reject changelists with empty deltas. This functionality was requested on the perforce-user mailing list in February 2005[2]. The ability of a content trigger to inspect the contents and reject the submission is ideal for this situation.

3.4 Run continuous integration build after each submission

A canonical example of a commit trigger initiates a build based on a submission. Since the build would be too time-consuming to complete during trigger execution, the **techniques** from section 2.7 would be used to initiate the build.

Starting the processing immediately could easily run into race conditions, so signaling the processing would be a better option. A review daemon implementation should be considered at that point, since it provides the signaling mechanism and the implementation will also use scheduled polling.

---

[2] http://maillist.perforce.com/pipermail/perforce-user/2005-February/032851.html

3.5 Integrate changes on a branch forward

A common need in a shrink-wrap style release scheme is to propagate fixes to a release codeline forward to later releases or back to main. Ignoring issues of process approval, this kind of integration is generally highly automatable. Conflicts generally only occur when there have been significant overhauls to the affected section of code. Since integration is also typically a very fast operation, this **purpose** can be fulfilled during the submission. The brevity of the operation, along with the sanctity of release branches and the locality of the change, minimizes the possibility of a race condition. Failures due to conflicts would provide notification to a branch owner for manual resolution. Safer automatic resolve options could be used for greater confidence in the result. Naturally, a follow-on build and test cycle should be signaled, suggesting that this trigger should be declared prior to the build trigger.

3.6 Configurable number of binary revisions

This novel and daring example of a commit trigger helps to maintain disk space by extending the concept behind the +S type modifier to an arbitrary number of revisions. The details of this example are discussed extensively in [Baum2005].

3.7 Add company-specific content to change form

In the absence of the hopefully upcoming fully customizable forms[3] many installations have adopted the convention of putting well-known tags in the changelist Description field. Until out triggers became available in 2004.2, this had to be done manually when filling in the form, leaving it up to the attention to detail of the submittor.

Out form triggers allow this policy to be proactively implemented by inserting the required information when the form is initially created. A typical example of this usage puts a ReviewedBy field in the description to be filled in by the developer to indicate who reviewed the code being submitted.

3.8 Enforce jobs status life cycle

An example of an “in” form trigger presented in [Bowles2005] enforces defect tracking state transition in jobs fixed by a changelist.

3.9 Enforce finer grained authorization

The prime example of a “save” form trigger is shown in [Perforce2004.2] on page 98. This trigger provides or denies authorization to modify client forms to particular groups. It compares the group of the person submitting the form to a list of groups authorized to modify client specs and rejects the modification when not allowed. This type of finer grained authorization can operate on any type of form and could also take into account

- User name: similar to group

[3] See ‘p4 help undoc’ and look at the ‘p4 spec’ command, but don’t use it yet.

- View contents: Disallow modification if views contain particular paths
- Form name: Disallow modifications to branches or labels that contain certain naming components, such as “rel”
- Option changes: Disallow use of particular options, such as “compress” or “normdir”
- Combinations: For example, only administrators are allowed to turn on compression on clients that do not involve already compressed binary content.

3.10 Protect template specs from deletion

The new “delete” form trigger added in 2005.1 also allows finer grained access control. An immediately apparent use is to protect crucial specs from deletion, just as you would protect them from modification in section 3.9. Prime examples of the types of forms you would want to protect are

- Release or test marker labels
- Release branch specs
- Special build client specs

4 Conclusion

Triggers provide a mechanism by which you can implement a wide range of policies for your development environment. Understanding the **context** and the available **techniques** allow you to choose the right implementation for your **purpose**. This paper attempts to give you a head start by cataloging several **techniques** and **purposes** in the trigger **context** as the basis for your own library.

References

[Baum2005] Baum, Richard, *Commit Trigger Example: Configurable Number of Revisions*, Proceedings of the 2005 Perforce User's Conference, Las Vegas.

[Bowles2005] Bowles, Jeff, *Scripting with Perforce*, Proceedings of the 2005 Perforce User's Conference, Las Vegas.

[Goldstone2005] Goldstone, John, *Using P4G.py From The Command Line*, Proceedings of the 2005 Perforce User's Conference, Las Vegas.

[Perforce2004.2] Perforce Software, *Perforce System Administrator's Guide 2004.2*, Perforce Software.