# Fact Extraction from Bash in Support of Script Migration

I. J. Davis, R. C. Holt
School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
{ijdavis,holt}@uwaterloo.ca

R. Mraz
Owl Computing Technologies, Inc.
38A Grove Street, Suite 101
Ridgefield, Connecticut, USA
rmraz@owlcti.com

***Abstract*—Owl Computing Technologies provides software and hardware that facilitates secure unidirectional data transfer across the internet. Bash scripts are used to facilitate customer installation of Owl's client/server software, and to provide high level management, control, and monitoring of client/server interfaces. With the evolution of more robust scripting languages, Owl now wishes to convert their bash scripts to other scripting languages. As part of this conversion exercise the configuration and customization of their bash scripts will no longer involve direct end user modifications of the script logic. It will instead be achieved through appropriate modification of a supporting XML configuration file, which is read by each script. This avoids the risk that end users erroneously change scripts, and makes legitimate end user customization of their scripts simpler, more obvious, and easier to discern.**

**An open source fact extractor was implemented that determines the dynamic usage made of every variable within an arbitrary bash script. This tool reports errors in a script and generates an XML configuration file that describes variable usage. Those variables whose value may not be assigned by an end user are manually removed from this XML configuration file. A second program reads this configuration file, generates the appropriate bash variable assignment statements, and these are then applied within bash by using the bash *eval* command. Collectively this provides a simple mechanism for altering arbitrary bash scripts so that they use an external XML configuration file, as a first step in the larger exercise of migrating bash scripts to other scripting languages.**

***Keywords—bash; parameterization; customization; refactoring, autonomous re-engineering***

## I. Introduction

Owl Computing Technologies Inc. [9] designs and markets hardware enforced data-diode-based cross-domain solutions for government and military cyber security. They provide electronic perimeter diode defence systems for critical infrastructure such as power generation and water management. They support secure, reliable, one-way information sharing for all data types, including historian replication, streaming full-motion video, scanned files, and SMTP email systems.

Because end-user requirements vary, and evolve over time, the software distributed by Owl to a client must be configured to align with each client's stated needs, expectations, operational environment and purchased features. This customisation is currently achieved by manually modifying the Owl bash scripts [3, 8] that install, invoke and monitor the underlying network client/server protocols, implemented in C.

As the number of customers and products that Owl supports has grown, this manual approach to configuring Owl's software has become increasingly difficult. It is difficult to know what modifications can be safely performed to Owl's scripts and to recover from the scripts information about how client software has been configured. It is difficult and time consuming for Owl to troubleshoot problems that arise within scripts that have been erroneously modified by an end user. It is also difficult for Owl to incrementally improve or correct the underlying logic employed within the scripts they provide to customers, since each modification must be manually applied against all variants of a given script. This is a labour intensive exercise and an inherently risky process.

There is now a desire to refactor the existing scripts, so they read instructions as to how they are to configure themselves from an XML [13] configuration file [5]. Given that Owl is currently maintaining in the order of 1,000 bash scripts, and that the largest of these scripts is more than 2,000 lines long, this represents a considerable undertaking.

In the longer term this refactoring exercise is expected to prove beneficial in simplifying the migration of bash scripts to more robust scripting languages (such as Python) which have built in capabilities for parsing XML files [1, 2, 6, 7].

## II. Our Refactoring Approach

When we initially began examining Owl's scripts [10], it was assumed that refactoring would involve three steps: manual examination of each script to identify the configurable variables contained within it; manual construction of an XML configuration file; and manual modification of each script to assign configuration variables the values read from this XML file. These assigned values would then parameterize the behaviour of a script.

This approach proved to be problematic. Those doing the refactoring lacked a-priori knowledge about what the numerous scripts did; why they had been implemented as they had been; or what aspects of a script were to be deemed by Owl to be 'configurable' by an end user. There was no obvious way of changing all the scripts to assign certain variables within them the corresponding values specified in an XML file, and there was no mechanism for automating the construction of these XML configuration files.

We have expertise in extracting facts from source code [11], so we initially thought that we might benefit by developing a bash fact extractor. The resulting facts contained in a bash script could then be presented graphically, and navigated by discovery software, more easily than the original scripts. To this end we modified the bash 4.2 open source code [4] to parse and document the contents of bash scripts.

The resulting fact extractor proved to be of limited use in offering insights into the design and functionality of a bash script. The sequential flow within most scripts, the lack of deeply nested function calls or control structures, and the comparatively small size of these scripts, meant that 'facts' about these scripts could be more readily understood by visual examination of them, than by trying to interpret the graphical output produced by running our bash fact extractor on them.

None the less, our decision to implement a fact extractor proved serendipitous, because in working with the bash open source code it became obvious that this source code might be exploited in other ways. What we then implemented from this same bash interpreter's source code was a fact extractor that discovered facts, not about the static logic and control structures present within a bash script, but instead about the dynamic run-time usage of bash variables.

## III. EXTRACTING FACTS ABOUT VARIABLES

When a bash script is interpreted its variables are dynamically created, assigned values, modified, and used. Discovering how such variables are manipulated allows us to automatically classify variables, and to automatically generate an XML configuration file for each bash script.

To discover the usage of variables, the bash interpreter was modified. The resulting open source tool is named **prowl**, since it discovers 'parameterization within Owl scripts' [12].

Control flow tests are evaluated but the resulting value then ignored. *If* statements have both their *then* and *else* clause sequentially interpreted. *Switch* statements have every *case* clause sequentially interpreted. *For*, *while* and *until* statements have their body interpreted exactly once. *Break, continue*, *exit and return* statements are ignored. Function declarations are noted, and function calls invoked. Instructions to source external bash scripts are honored. All other built in bash operations are ignored, as are invocations of external programs.

*Prowl* keeps track of how each variable is created and modified anywhere within a bash script. The following binary flags are currently employed for this purpose:

1. ENVIRONMENT: Variable's initial value is obtained from the corresponding environment variable.
2. USED: Variable's value is used in interpreting the script.
3. CHANGED: Variable is assigned multiple distinct values.
4. RETURNED: Variable is used in computing the value associated with a return or exit statement.
5. SOURCED: Variable is created within a sourced script, and never observed to be manipulated or used outside of sourced scripts.
6. DERIVED: Variable is assigned a value that is itself derived by expanding other variables.
7. LOOP: Variable is created in a *for*, *while*, or *until* construct.
8. FUNCTION: Variable is created within a function.

Armed with the knowledge about how variables are created and used within a bash script, it is straightforward to emit this knowledge at end of run, in a suitably encode XML file.

For example, if a bash script file named *script1* contains

```
y=10; x=1; y=2; z=$x+$y; exit $(($z))
```

then the command *prowl script1 > script1.cfg* generates the XML file *script1.cfg* shown below:

```
<prowl>
  <script1 version="default">
    <default>
      <x>1</x>
      <y changed>2</y>
      <z derived returned>1+2</z>
    </default>
  </script1>
</prowl>
```

Figure 1. Example XML Configuration File

The XML configuration file has four levels of nesting:

1. The outmost root entity is labelled <prowl>.
2. The second entity level (e.g., *script1*) identifies by its entity name a named script. This entity can be repeated, thus permitting configurations associated with multiple scripts to be recorded in a single file. The optional version attribute specifies a default version to use if none is provided.
3. The third entity level (e.g., *default*) identifies a named version. Multiple configuration versions can be associated with a named script. If no version is specified the earliest is used. Distinct configurations for a named script typically define the same named variables but with different values.
4. The innermost entities (e.g., *x*) contain text. The name of each such entity identifies a bash variable, while the text corresponds to the last value assigned to this variable. Attributes associated with these entities describe variable usage of potential concern to the reader. To improve readability, and assist in file comparisons these entities are themselves sorted by their entity name.

Those variables most likely to be configuration variables are those that are assigned an initial value in the global scope of the shell script and that never change. The value assigned to these variables should be used, but not be derived from other variable values, or used within *return* or *exit* statements.

*Prowl* can optionally include only such variables in the output XML file. However, it is safer to emit details about all variables used within a bash script to the XML configuration file, and then have a reviewer manually remove from this XML file those variable assignments not deemed to be modifiable by an end user. The remaining variables are precisely those that (by having their value changed) permit an end user to configure their scripts appropriately.

Some scripting languages (such as Python) differ from bash in requiring stricter typing of variables. Future work will examine if it is possible to offer advice as to how variables should be typed, within the XML file, as part of the larger exercise of porting bash scripts to other scripting languages.

## IV. READING THE CONFIGURATION FILE

Having produced the desired configuration file, the assignments specified in it must be ported back into the original bash shell script, so that the script operates on these now external configuration settings.

This is achieved by implementing a second program (named **prowler)**. *Prowler* parses a specified configuration file, and is provided with the name of a script and optional configuration version, from which configuration information must be recovered. It then emits a composite bash statement that assigns all of the configuration variables the corresponding values specified in the appropriate version of the named script. By wrapping the *prowler* invocation within a bash *eval* statement at an appropriate point within the original script, these externally assigned values are imported back into the script.

Configuration variables can also be assigned values by specifying these assignments as input arguments to *prowler.* This permits configuration instructions to also be provided as command line inputs to a bash script, when this is considered appropriate.

For example, in Figure 1 eliminating the variable z because it is derived from other variables and changing the values of *x* and *y* produces the modified configuration file *script1.cfg*

```
<prowl>
  <script1 version="default">
    <default>
      <x>10</x>
      <y>20</y>
    </default>
  </script1>
</prowl>
```

The command

```
prowler script1.cfg y=25
```

then emits

```
x="10";y="25";
```

while the modified bash script

```
x=1; y=2
eval `prowler script1.cfg`
z=$x+$y
exit $(($z))
```

assigns *x*=10 and *y*=20 and so exits with a return code of 30. If the variable *x* is removed from *script1.cfg* it will then default to *x*=1. This value cannot then be changed by *prowler*.

## V. INDUSTRIAL EXPERIENCE

In automating the construction of XML configuration files a number of minor issues arose. Because we are interpreting all possible paths within a bash script, we discovered previously unobserved semantic errors in our bash scripts. For example, functions must be declared before being invoked, but we found cases where a bash function was being incorrectly invoked before been declared. The ability to thus validate the run time behavior of a bash script, and to correct discovered errors, proved an unexpected benefit of running *prowl* on all our bash scripts.

Because we do not execute external programs, there are cases where we fail to identify the values associated with variables. For example the bash statement *a=`date`* does not execute *date* and so assigns the variable *a* the empty string instead of the current date.

There are cases where the bash interpreter is instructed to *execute* a script, rather than *source* it. Since such scripts are not executed by *prowl*, their content is not interpreted by *prowl*, and so variable usage within them is never seen. This problem can be addressed by examining each program invocation, discovering when a bash shell script is being executed, and then recommending that such external scripts be reworked so that they are sourced as bash scripts, rather than executed as programs.

Sometimes there is complex nesting of *sourced* bash scripts, and in these circumstances it is difficult to manually determine where a discovered bash variable is actually being used. To address this problem *prowl* was augmented to list the locations of all source scripts which declared or used each reported variable, as well as indicating if each such variable was used in the main script.

Another problem is that sometimes constants within a script are consecutively assigned distinct values, with the prior redundant assignments being left unchanged, rather than being removed. This poor programming style makes it difficult to accurately report which script variables are intended to be constant. Cases where the same variable is assigned distinct values in consecutive statements should be reported, so that the offending scripts can then be improved.

A benefit of our approach is that differences between variants of the same script configured for different users can now be discovered not only by comparing the scripts, but also by performing '*diffs*' on the XML configuration files produced by our analysis of these scripts. This provides a good indication of how scripts have been initially cloned, and then subsequently modified on a per customer basis.

The final challenge in the refactoring exercise, having decided which constants may be assigned values by an end user, is in deciding where to place the introduced invocation to *prowler* within each script. To be effective, this placement must occur after all assignments of values to configuration variables (which would ideally be commented out or removed if not defaults), but before any of these configuration variables are used or tested within a script. This is a straightforward exercise when these configuration variables are all assigned values at the start of the main script, but challenging when the assignment of a value to a configuration variable is delayed until first use of that configuration variable, which may potentially occur not in the main script but in a sourced script.

When *prowl* is presented with a Bash script it therefore assists in validating *prowler's* placement. To do this it reports all XML configuration variables used before *prowler* is invoked, and all assignments to these variables afterwards.

## VI. CONCLUSIONS

Our group at the University of Waterloo has historically focused on static fact extraction from binary source code written in C, C++, and Java, as well as from binary executable programs. We have been hesitant to develop dynamic fact extraction tools, which document the observed runtime behavior of software, because it is difficult to ensure that all relevant runtime behavior is observed, and because representing runtime behavior is inherently challenging.

We solved the challenge of deciding how a bash shell script should best be executed in order to extract dynamic facts from its execution by forcing all possible paths through each bash script to be executed at least once. We solved the problem of how to concisely present factual information about the runtime behavior of a script, by limiting the facts presented to those that were both useful within the larger refactoring exercise, and relevant to a reviewer, irrespective of the actual runtime behavior of an observed script.

This represents a novel approach to dynamic fact extraction, and offers opportunities for recovering other important facets of run time behavior, not easily discovered by static examination of source code. We could for example discover what programs are potentially being invoked by the execution of a script, what files are being accessed, or what interfaces are being used.

Modifying the bash open source software as described is straightforward and involves only a limited number of changes to a small number of source files. The result is a useful tool, which is as robust in handling arbitrary bash shell scripts as the original bash interpreter. It is also easy to port to the multitude of existing platforms that support bash.

The challenge of manually examining approximately 1,000 bash scripts to identify configuration variables in these scripts and to then refactor each bash script so as to obtain actual configuration values from an external configuration file seemed a necessary, but daunting exercise. The tools we have presented here make this exercise much less painful.

Our industrial partners no longer have to laboriously examine each script to discover how each may be configured. Instead they are presented with the list of all variables used by an arbitrary bash script, and how each such variable is used. If sensible naming conventions are used, it is comparatively easy to decide which variable assignments are to be performed by the end user as part of their configuration process, and which are to remain within the script. The XML configuration file to be associated with each script is automatically generated, ensuring a consistent look and feel, and the decision as to which scripts ultimately share a single configuration file is left open.

End users benefit since it is easier for them to modify an XML file than to identify which scripts need to be changed and how. Reconfiguration of their software becomes easier, as does the process of installing routine updates to the scripts they are using. The modifications to the XML file concisely document how an end user has elected to configure the software they are using, which is important to both Owl and the end user.

In the longer time frame, changing the bash scripts to use external XML configuration files makes it easier to migrate all of the bash scripts to other scripting languages, without requiring end user familiarity with these new scripting languages.

## ACKNOWLEDGMENT

Owl CTI motivated this research, provided the funding for it, and assisted in the preparation of this paper. Their support and encouragement is much appreciated. Gabriel Silberman provided the initial vision, which made this project possible. The referees are thanked for their many helpful suggestions.

## REFERENCES

[1] R. Delaney, "Python scripts as a replacement for bash utility scripts" Linux Journal, November 2012 pp. 69-78
http://www.linuxjournal.com/content/python-scripts-replacement-bash-utility-scripts

[2] N. Gift, "Python for bash scripters: A well-kept secret" Red Hat Magazine, 7th February 2008.
http://magazine.redhat.com/2008/02/07/python-for-bash-scripters-a-well-kept-secret

[3] GNU, "Bash reference manual".
http://www.gnu.org/software/bash/manual/bashref.html

[4] GNU "Bash 4.2 project archive". http://ftp.gnu.org/gnu/bash

[5] H. Hall, "XML configuration files for your applications", Code Project, 6 February 2009. http://www.codeproject.com/Articles/33166/XML-Configuration-File-for-Your-Applications

[6] M. Lutz, "Programming Python" O'Reilly 2011.

[7] Maxwell, "Python vs. Bash Benchmark Test" 17 February 2009 www.murga-linux.com/puppy/viewtopic.php?mode=attach&id=16212

[8] C. Newham, "Learning the bash shell: Unix shell programming" O'Reilly 2005.

[9] Owl Computing Technologies, Inc. Securing your networks from cyber threats. http://www.owlcti.com

[10] Owl Computing Technologies, Inc. "Owl Computing Technologies partner with University of Waterloo" Press Release http://www.owlcti.com/news/pr/Owl_Partners_Waterloo.html

[11] University of Waterloo, "Javex Java Fact Extractor"
http://www.swag.uwaterloo.ca/javex/index.html

[12] University of Waterloo, "Converting bash scripts to use XML configuration files" http://www.swag.uwaterloo.ca/prowl/index.html

[13] W3C, "Extensible Markup Language (XML) 1.0 (Fifth Edition)
http://www.w3.org/TR/xml