ARVE: Aspect-oriented Runtime Verification Environment

Hiromasa Shin, Yusuke Endoh, and Yoshio Kataoka
System Engineering Laboratory, Corporate R&D Center, Toshiba Corporation
{hiromasa.shin, yusuke.endoh, yoshio.kataoka}@toshiba.co.jp

Categories and Subject Descriptors D.2 [*Software Engineering*]: Testing and Debugging

General Terms Aspect-oriented programming, Debugger

Keywords Runtime verification, Debugger, Optimization

1. Introduction

Software testing, including runtime verification, is essential for developing reliable software products. As is often the case with the development of industrial software products, the scale of a test program tends to be larger than that of a product program. As the test program is inherent in the product program, it is often structured in an ad hoc manner. We consider that improving modularity, reusability and portability of a test program constitutes an important issue. Since a test program is concerned with its target program in crosscutting manner, the paradigm of aspect-oriented programming [1] is useful for improving the composition of a test program.

Even today many industrial software products, such as embedded software, are written in C/C++, and popular software development tools, which are commonly available in various platforms, are classic tools, such as symbolic debuggers and performance profilers. Meanwhile, most of aspect-oriented language are designed for Java language, or are not available in embedded platforms. Considering these circumstances in industrial software development, we have designed a practical tool named aspect-oriented runtime verification environment (ARVE). ARVE enhances the capability of a symbolic debugger by employing aspect-oriented technology. In ARVE, a test program is written in aspect-oriented script language and can be dynamically woven into a target program.

We briefly present the features, structure and evaluation results of ARVE and explain simple applications. We also explain work-in-progress and speculative future work concerning ARVE.

2. Recent work

For illustration of ARVE's functionality, we firstly present an application of ARVE to an event sequence checker. The following script *RegexpChecker* gathers events of file handling operations in target program execution, and checks whether or not the operation sequence satisfies the pattern specified in regular language. Having found the operation deviating from the specified sequence, this checker breaks the execution of the target program and dumps the execution stack.

```
import "RegexpChecker.pl";
aspect FileRegexpChecker extends RegexpChecker {
  pointcut mark() : call(^fopen$) || call(^fread$)
                  || call(^fwrite$) || call(^fclose$);
  sub new () {
    my $class = shift;
    my $self = RegexpChecker->new(
       "A-fopen[-1] (B-fread[3]|B-fwrite[3])* B-fclose[0]");
    return bless $self, $class;
  }
}
```

This ARVE script is written in the Perl-based language equipped with aspect-related syntax similar to AspectJ [2]. The concrete aspect *FileRegexpChecker* inherits the abstract aspect *RegexpChecker*, and describes the event and the pattern by overriding pointcut *mark()* and constructor *new(..)* argument. The parent aspect *RegexpChecker* is a reusable aspect, which contains the algorithm to generate DFA (Deterministic Finite Automaton) from the regular expression and to drive the DFA by invocation of advice related with pointcut *mark()*. The meaning of the terminal symbol in the regular expression is *"(after or before)-(name of joinpoint)[argument index of file handler]"*. We applied this aspect to monitor the API usage of socket handling in the server process, such as Apache and Squid, and conformed that it worked properly. In this example, the basic mechanism of aspect-oriented programming improves the modularity of the checker, and especially inheritance mechanism improves the reusability of the checker.

Figure 1 is an architecture diagram of the ARVE system consisting of ARVE script, ARVE kernel, symbolic debugger, script interpreter and target program. The symbolic debugger works as a peripheral system for the ARVE kernel, and provides the functionality of breakpoint management, the target's symbol table management, and the target's memory access for the ARVE kernel. The ARVE system utilizes the debugger's function via a clear-cut debugger interface. This interface layer ensures the independence of the ARVE kernel from a specific debugger. Any debugger satisfying this interface can work in the ARVE system. In this implementation, we adopted the debugger GDB [3], which is a popular debugger and supports many embedded platforms.

We evaluated runtime performance of the prototyped ARVE system. In a laptop PC (Dynabook TECRA 9000, Pentium-III 1.2GHz, Linux 2.4.20 and gdb 6.4), the elapsed time for a subroutine call is about 10 nanoseconds, and the elapsed time for the same call with an empty advice is about 6 microseconds. Installing an advice at each subroutine call, the program execution will take 600 times longer time. However, this situation will be extreme; many applications will use fewer advice calls than in this case. Typical overhead time is 6 microseconds, and this value will be acceptable for monitoring appropriately selected places in network communication or user interactive application.

We briefly summarize the difference from related work. ARVE uses a symbolic debugger to weave aspect into the target, and this

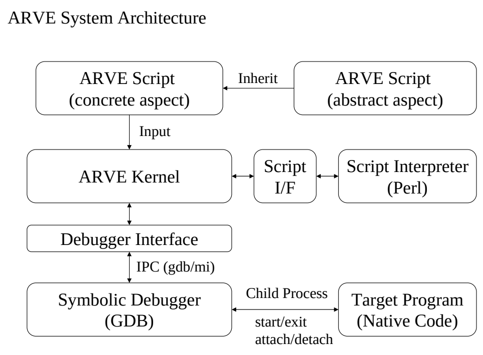

Figure 1. ARVE System Architecture

feature differs from usual dynamic weaving techniques based on JVM (Java Virtual Machine) reported in articles [4, 5]. ARVE uses script language to describe aspect, and this feature differs from the usual dynamic instrumentation techniques reported in articles [6, 7].

Compared to the usual dynamic instrumentation techniques, the ARVE approach has a disadvantage in terms of runtime efficiency; however, the ARVE approach has an advantage in terms of platform portability. The aspect of ARVE is written in script language. ARVE has a definite interface with the symbolic debugger, and can use a different symbolic debugger for each platform. Thus the script of ARVE remains independent of the platform. This feature will improve portability of a test program.

3. Work in progress

We are working on two plans. The first plan is to extend the ARVE kernel to support multi-process environments. In our experiment on an Apache server, we had difficulty in tracing many processes forked by the server. If ARVE supports an aspect among multi-processes and automatically attaches to multiple processes, the runtime verification aspect concerning IPC (Inter Process Communication) can be naturally described in a single aspect.

The second plan is to design ARVE script to check an event sequence specification written in LTL (Linear Temporal Logic). We have already prototyped the event sequence checker based on regular expression. Analogous to the case of the regular expression, the reusable abstract aspect implements the automaton, which is constructed by existing tableau construction techniques and is driven by execution event.

4. Highly speculative work

We have two plans concerning speculative future work. The first plan is to apply ARVE to an execution environment for model-based testing [8]. In order to utilize the capability of ARVE, we feel a strong need to connect ARVE usage and upstream design. In model-based testing, we can extract a test suite from the formal specification of a target. Converting the test suite to ARVE script, ARVE can execute the conformance test. Since ARVE has a debugger's capability, it can not only monitor the relation between input and output in testing, but also monitor the internal state of IUT (Implementation Under Test).

The second plan is to make ARVE and static analysis complement each other. The static analysis, such as ESP [9], has an advantage in terms of full path coverage without execution, but has a disadvantage in lack of information due to abstract interpretation. Since the advantage and disadvantage of dynamic analysis are opposite of those, the complementary combination of each analysis is expected to constitute a powerful approach. The specification language, such as ESP's OPAL, describing the finite state machine, can be viewed as an aspect-oriented automaton description language. Unifying the specification language between dynamic analysis and static analysis, will be a good starting point to make the two analysis methods complement each other.

5. Summary

We have presented a practical tool, ARVE, which enables development of a test program in script language and in aspect-oriented paradigm, and achieves independence from an underling symbolic debugger. As work in progress, an extension for multi-process support and a development for LTL-based verifying aspect are presented. As a highly speculative work, combination with model-based testing or static analysis is presented. All these approaches are aimed at improving reusability, portability and modularity of a test program in industrial software development.

References

[1] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, 1997.

[2] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[3] Richard M. Stallman. *Debugging With GDB: The Gnu Source-Level Debugger (9th Edition)*. Free Software Foundation, 2002.

[4] Andrei Popovici, Thomas Gross, and Gustavo Alonso. Dynamic weaving for aspect-oriented programming. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 141–147, New York, NY, USA, 2002. ACM Press.

[5] Davy Suvée, Wim Vanderperren, and Viviane Jonckers. Jasco: an aspect-oriented approach tailored for component based software development. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 21–29, New York, NY, USA, 2003. ACM Press.

[6] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM Press.

[7] Bryan Buck and Jeffrey K. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

[8] Michael Barnett, Wolfgang Grieskamp, Lev Nachmanson, Wolfram Schulte, Nikolai Tillmann, and Margus Veanes. Towards a tool environment for model-based testing with asml. In *FATES*, pages 252–266, 2003.

[9] James R. Larus, Thomas Ball, Manuvir Das, Robert DeLine, Manuel Fahndrich, Jon Pincus, Sriram K. Rajamani, and Ramanathan Venkatapathy. Righting software. *IEEE Software*, 21(3):92–100, 2004.