Automated Source-Level Instrumentation for Dynamic Dependency Analysis of COBOL Systems

Holger Knoche[1], André van Hoorn[2], Wolfgang Goerigk[1], and Wilhelm Hasselbring[2]

[1] b+m Informatik AG, 24109 Melsdorf

[2] Software Engineering Group, University of Kiel, 24098 Kiel

Abstract

Dynamic analysis, e.g. dynamic dependency analysis, requires the injection of monitoring code into an existing application. This paper presents an approach to automatically locate and process the required injection sites in the source code of a COBOL system and discusses issues arising from source-level instrumentation when applied to an industrial case study.

1 Introduction

The analysis of dynamic dependencies of software systems is a valuable source of information, especially in modernization scenarios. The concept of aspect-oriented programming (AOP) [2] has proven to be a helpful tool for inserting the required instrumentation. However, although approaches have been proposed [3], most legacy runtime environments still lack AOP capabilities.

In this paper, we present a pragmatic approach to add instrumentation automatically to an application's source code [1] in an AOP-like way. Furthermore, we present selected results and challenges which arose when we used this approach to perform a dynamic calling dependency analysis of an industrial COBOL case study system.

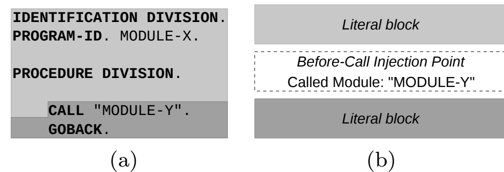
The remainder of this paper is structured as follows. Section 2 provides an overview of our source code instrumentation approach. Section 3 presents selected challenges that we encountered when we applied the approach to an industrial case study. Section 4 discusses the issue of incomplete traces which is likely to arise from source-level instrumentation. Finally, concluding remarks are presented in Section 5.

2 Source-Level Instrumentation

The process of automatically augmenting source code with instrumentation instructions is carried out in two subsequent steps. First, a static analysis component processes the source code using common static analysis techniques. This component produces an object model that represents the input file as a non-empty sequence of *blocks*, which may be either *literal blocks* or *injection points*. The latter are similar to AOP's *join points*, and like these, they may carry information about their static source context.

This work is partly funded by the German Federal Ministry of Education and Research (BMBF) under grant number 01IS10051.

This approach can be illustrated using the small COBOL code example presented in Figure 1(a).



(a) (b)

Figure 1: A simple COBOL example

Suppose we want to add instrumentation code before every `CALL` statement. In this case, the static analysis component could produce the following block sequence: one literal block from the beginning of the module to the beginning of the `CALL` keyword, one *before-call* injection point carrying the called module name as additional context information, and one literal block from the beginning of the `CALL` statement to the end of the module, as shown in Figure 1(b).

In a second step, the block stream is fed into a code-generation component which produces the instrumented code. When encountering an injection point, the code generator selects and expands an appropriate, user-provided code template depending on the type of the point. Thus, it is possible to exploit the provided specific context information during template expansion. The templates can therefore be seen as *advices* with simple *pointcut designators* that only depend on the point type.

Although this approach provides a flexible way of instrumenting source code and works well in practice, it has one major drawback: it can only be applied to source code written in a supported programming language. In heterogenous environments with third-party components, it is therefore likely that some components remain uninstrumented. Section 4 presents a way to handle this issue for dynamic calling dependency analysis.

3 Application to a Case Study

In order to evaluate the described approach, we applied it to perform a dynamic calling dependency analysis of a substantial industrial application consisting of more than 1,000 COBOL modules. Selected challenges that were encountered during this process are described below.

Static analysis of COBOL programs, which is required to locate the injection points, is challenging due to various reasons. First of all, many different dialects of and vendor-specific extensions to the language exist. In addition, many COBOL modules contain embedded transaction control or SQL statements that are replaced by specific preprocessors during the build process. However, particular transaction control statements are important for dependency analysis because they may affect the control flow.

In order to avoid implementing a complete COBOL grammar, we employed the concept of *island grammars* [4], which works particularly well for COBOL because of the language's many keywords. Since the code is only partially parsed in detail, such grammars tend to be more robust with respect to dialects and unknown constructs.

Albeit being helpful, island grammars have some subtle pitfalls. While most injections had to be inserted at the beginning of the respective statement, which was easy to find, some had to be added at the end. The location of these points proved to be challenging. The most notable example was the `PERFORM` statement, which had to be analyzed in detail not only to locate its end, but also to distinguish its procedural variant, which should be instrumented, from the inline variant.

Another challenge for both static analysis and code generation was COBOL's fixed source code format. Since common parser infrastructures are designed for free-format languages, it took some non-trivial adjustments to introduce format sensitivity. The main challenge for the code generator was to maintain the correct formatting after injecting code. Since the COBOL file format allows arbitrary data in certain columns, the code generator must preserve the exact start column of literal blocks when injections are made to prevent such data from slipping into the code area.

During our evaluation, we successfully managed to track module calls via both native COBOL mechanisms (`CALL`) and the employed transaction manager (`EXEC CICS LINK` and `EXEC CICS XCTL`) as well as section invocations (procedural `PERFORM`). A total of 140,351 injection points of 14 different types were inserted for the analysis.

4 Incomplete Event Traces

In order to collect trace information, we employ an event-oriented logging mechanism which logs call, entry, and exit events for modules and sections, respectively, into a database. These events are then transformed into a format compatible with our Kieker analysis and visualization framework [6].

The reason for monitoring both call and entry events is that this seemingly redundant information allows to partially reconstruct message traces involving uninstrumented components, such as third-party libraries for which no source code is available. We distinguish the following cases: (i) definite call: call $A \rightarrow B$, entry into B; (ii) immediate return from uninstrumented component: call $A \rightarrow C$, next logged event happens in A. In this case, we assume that C is successfully called and returns control to A. Note that there will always be a next event in A due to exit events; (iii) call through an uninstrumented component: call $A \rightarrow D$, entry into E. In this case, we assume that A has successfully called D and D (transitively) called E in the process; (iv) uncalled section: entry into a section S without a preceding call to S. In this case, we assume that the control flow has run through the section and do not interpret the situation as a call.

In order to visually distinguish assumed from definite call dependencies and components, we extended the Kieker framework. An example illustrating the aforementioned cases is shown in Figure 2. Assumed dependencies are shown using dashed lines, and assumed components are shaded.

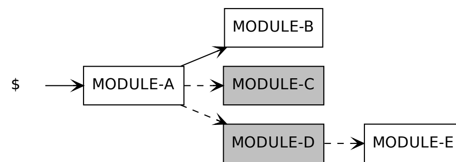


Figure 2: Module-level call dependency graph with assumed dependencies

5 Conclusions

Although we only implemented a small subset of the common AOP features, we were able to successfully perform a dynamic dependency analysis of an industrial case study application. The instrumentation approach integrates seamlessly with our earlier work [5], and its modular design allows to apply the approach to other analysis scenarios as well as to other languages and environments, what we plan for our future work.

References

[1] T. Chen, H. Kao, M. Luk, and W. Ying. COD — A dynamic data flow analysis system for Cobol. *Information & Management*, 12(2):65 – 72, 1987.

[2] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP '97*, pages 220–242. 1997.

[3] R. Lämmel and K. De Schutter. What does aspect-oriented programming mean to Cobol? In *AOSD '05*, pages 99–110, 2005.

[4] L. Moonen. Generating robust parsers using island grammars. In *WCRE '01*, pages 13–22, 2001.

[5] A. van Hoorn, H. Knoche, W. Goerigk, and W. Hasselbring. Model-driven instrumentation for dynamic analysis of legacy software systems. In *WSR '11*, pages 26–27, 2011.

[6] A. van Hoorn, J. Waller, and W. Hasselbring. Kieker: A framework for application performance monitoring and dynamic software analysis. In *ICPE '12*, 2012. To appear.