Using Dynamic Aspects to Distill Business Rules from Legacy Code

Isabel Michiels
Theo D'Hondt
PROG, Vrije Universiteit Brussel (VUB)

Kris De Schutter
Ghislain Hoffman
INTEC, Universiteit Gent (UGent)

Abstract

Large organizations often rely on business rules to express their business constraints and very often these rules are scattered throughout different parts of the source code. Although call-stack context and other dynamic events provide a valuable view on (old) code, checking why the output of their system produces certain values remains a complex and time-consuming process. In this paper we advocate the use of dynamic aspects to facilitate and optimize the process of distilling business rules from legacy code. We demonstrate this use through a possible scenario of investigation of a small but real life case study and conclude with our envisioned practical implementation.

1 Introduction

When information systems are created for use within (large) organizations, many business rules are embedded into the software as some kind of constraints. On first implementation of these business rules they are usually fragmented and inserted into many different parts of the source code, which makes it hard to localize them at a later stage when, for example, the software is evolving.

Information systems that have been around for a long time typically suffer from this scattering of business rules; although some rules have to be continuously adapted and are thus kept in human memory, there are other business rules that, once they are implemented, are left unchanged and, through the passage of time, are somehow "lost" in the source code. This gets even more problematic when documentation is not being kept up to date.

Checking whether the output of an information system is correct, or why it produces a certain value, therefore becomes very difficult. The only plausible approach to tackle this scattering is to trace program execution, which can be a complex and time-consuming process.

As an illustration, consider being confronted with the following situation: our accountancy department reports that several of our employees were accredited an unexpected and unexplained bonus of €500. Accountancy rightfully requests to know the reason for this unforeseen expense. Not knowing the exact cause, we are left faced with having to comprehend an old and poorly documented system.

2 Research Context

The problems described here were encountered several times within the ARRIBA[1] project, a generic research project funded by the IWT, Flanders[2], which started out in October 2002 and will last four years [2].

[1] http://arriba.vub.ac.be/
[2] http://www.iwt.be/

The main goal is to provide a methodology and its associated lightweight tools in order to support the integration of disparate business applications that have not necessarily been designed to coexist.

Inspiration for this project comes from two driving forces: on the one hand we have a consortium of research groups[3] that have been active in the field of software engineering and, more particularly, in re(verse) engineering, software evolution and software architectures. On the other hand, we have the recently created forum of Belgian enterprises[4] (large and small) interested in a joint initiative to identify generic problems and likewise generic solutions plaguing their ICT base.

The object of this investigation is therefore the identification of mainstream ICT problems within this forum of enterprises that rely on information technology for their critical business activities.

Part of this is covered by the newly named discipline of Enterprise Application Integration (EAI), and re(verse) engineering; another part lies in the use of AOSD techniques for code instrumentation as an important tool to aid program comprehension.

3 Difficulties encountered

With an estimated 60\% to 80\% of all business applications still written in COBOL[4], it was no surprise to find exactly this in the code base of the companies involved in ARRIBA. COBOL therefore quickly gained much of our focus.

Working with COBOL has its difficulties:

- The applications concerned are no longer understood. Major mission-critical applications were developed in the 70's by programmers that are no longer working at the company, or have moved on to other projects.
- The code is badly structured and poorly documented. The amount of code is huge (millions of LOC) and has been adapted many times for several reasons (switching platforms, year 2000 conversions, transition to the Euro currency,…). So keeping the documentation synchronized with those evolutionary changes didn't always happen.
- Logic is spread out over the entire application. COBOL has only limited modularity mechanisms. Therefore complex logic had to be manually distributed over the programs.
- COBOL as a programming language is no longer understood. Languages like COBOL are no longer very popular with the new generation of programmers, nor are they being actively taught to students. We are more and more faced with a new generation of computer scientists with a different kind of background.
- Specific COBOL language constructs: COBOL language constructs such as REDEFINE, GO TO or ALTER make it extra difficult to trace the execution of a program.

4 Proposed approach

To tackle some of the above problems we propose to use dynamic aspects as an aid for semi-automating the process of tracing the execution of a (COBOL) program.

Dynamic aspects are aspects that are used when you want to invoke some behavior based on the dynamics of program execution[1]. This technology will allow us to inspect specific parts of the dynamic execution of an application.

---

[3] Vrije Universiteit Brussel (VUB), Universiteit Gent (UG), Universiteit Antwerpen (UA); supported by UCL in Louvain-La-Neuve, and SCG, Berne, Switzerland.

[4] Inno.com, KBC, LCM, Banksys, Toyota, KAVA and Pefa

To elaborate on this point, let us reconsider the situation presented in the introduction: being faced with an unexpected bonus of €500 for some of our employees. We now present a simplified but realistic scenario of how dynamic aspects can help us get a handle on this problem.

4.1 Possible scenario of investigation

First thing we have to figure out is which variable or record holds the value for the bonus. Accountancy has provided us with printed reports showing the questionable results. We use these to find the routine which has generated these results through a simple search on strings, and find that the data (the bonus in euros) is being held in a variable named (for instance) BNS-EUR. (Because we plan on using it later we also write down the variable holding the employee id number.)

Looking into the definition of BNS-EUR, it tells us that it is defined as an edited picture[5]. We conclude that this variable is only used for pretty printing the output, and not for performing actual calculations. At some time during execution the correct value for the bonus was moved to BNS-EUR, and consequently printed. We now have to find what variable that was.

Rather than looking at all MOVEs to BNS-EUR, we will cut down the list to those MOVEs which occurred while processing one of the 'lucky' employees (which we can deduce from the reports we received from accountancy). This is where our first dynamic aspect helps us out. It limits the data we have to look at by allowing us to apply previously gained knowledge.

We find the possibilities to be one of several string literals (which we can therefore immediately disregard) and a variable named BNS-EOY (whose name suggests it holds the full value for the end-of-year bonus).

Our next step is to try to figure out how the end value was calculated. Knowing that would allow us to check the figures and maybe spot an error. To achieve this we set up another aspect to trace all statements modifying the variable BNS-EOY.

Consider the following piece of (pseudo) COBOL code to demonstrate some of the things we (might) have to capture in this phase:

```
03000       READ EMPLOYEE.
03100*      ...

04800       ADD B31241 TO BNS-EOY.
04900*      or
05000       COMPUTE BNS-EOY = BNS-EOY + B31241.
05100*      or
05200       MOVE B31241 TO BNS-EOY.
05300*      or...
```

These include arithmetic statements and MOVEs. Dynamic aspects allow us to get this lifetime trace of a variable. And again we can limit these lifetime-traces to those which occur while processing specific employees.

In doing so we get lucky and find a variable (cryptically) named B31241, which is consistently valued €500, and is added to BNS-EUR in every trace.

Before moving on we'd like to make sure we're on the right track. We want to verify that this addition of B31241 is only triggered for our list of 'lucky' employees. Again, a dynamic aspect allows us to trace execution of exactly this addition and helps us verify that our basic assumption holds indeed.

---

[5] In COBOL you define a variable as being a picture of a number of characters or numeric values, like "A-VAR PIC 9(2)", which means that A-VAR can hold a numeric value of 0 up to 99. You can also edit these PICs by making them ready for displaying them the way you want, like a date for example: "A-DATE PIC 99/99/99". [6]

Knowing what is added leaves us with the question of why. Unfortunately, the logic behind this seems spread out over the entire application. So to try to figure out this mess we would like to have an execution trace of each *lucky* employee, including a report of all tests made and passed, up to and including the point where B31241 is added. Dynamic aspects allow us to get these specific traces. Comparing these will narrow down our search and help us find our way inside the original code.

This is where our story ends. We find that B31241 is part of a business rule: it is a bonus an employee receives when he or she has sold at least 100 items of the product with number 31241. Apparently this product code had been assigned to a new product the year before. It once was associated to another product which had been discontinued for several years. The associated bonus was left behind in the code, and never triggered until employees started selling the new product.

4.2 Future Implementation

The way we want to implement dynamic aspects in COBOL is to use a declarative language at the meta level as a pointcut language that will reason statically about dynamic execution traces. A declarative language is especially suited for expressing constructs like mentioned in our COBOL example in section 4.1: for capturing the modification of a variable one can write a predicate which expresses the different ways for changing a variable. This way you create an abstraction layer which makes it easy to adapt the predicate in case of any changes (for example when another COBOL dialect uses other statements to modify a variable).

Having the ability to transform COBOL code into XML, we have started to work on two similar declarative approaches to achieve this. In one, we have been experimenting with combining SOUL (Smalltalk Open Unification Language)[5] with this XML representation and then representing the structure of COBOL applications into the logic language SOUL. So far this has allowed for static reasoning only.

The second approach uses a Java-Prolog bridge (in the form of PrologCafe[6]) to enable a Prolog environment to reason about the intermediate XML representation, and even transform it. This has already made it possible to implement a very simple and generic logging aspect.

The above mentioned approaches will end up generating extra COBOL code into the original applications, thereby implementing the proposed aspects. Both approaches allow the exploration of a richer and easier-to-use language to help us express our concerns.

So far we can conclude that a declarative language at a meta level is a powerful medium and that it is certainly suited within this context.

5 Conclusions

In this position paper we presented our intentions of using dynamic aspects as a technique for semi-automating the process of distilling "lost" business rules out of large pieces of (COBOL) legacy software. These ideas came about within the context of the research project ARRIBA.

We first pointed out the difficulties we came across in this research since we are working on large real-life case studies of COBOL legacy systems. The size and complexity of these information systems and lack of expressiveness regarding modularity cannot be ignored.

We demonstrated how we see our approach being integrated in a simplified but real life problem scenario, frequently encountered in one of the companies involved in our research project. We then advocated the use of a declarative language at the meta-level as some sort of pointcut language to capture specific execution traces. This way we could simplify the process of distilling "lost" business rules out of information systems.

---

[6] http://kaminari.scitec.kobe-u.ac.jp/PrologCafe/

To conclude we would like to point out that, although we have demonstrated our ideas within the context of COBOL legacy applications, we firmly believe that they can be applied to similar cases implemented in other programming languages.

Acknowledgments

Our thanks to Wolfgang De Meuter, Thomas Cleenewerck and Herman Tromp for their ideas and for proofreading the paper.

References

[1] Johan Brichau, Wolfgang De Meuter, Kris De Volder. Jumping Aspects. In ECOOP Workshop on Aspects and Dimensions of Concerns", Cannes, 2000.

[2] Isabel Michiels, Dirk Deridder, Herman Tromp and Andy Zaidman. Identifying Problems in Legacy Software: Preliminary Findings of the ARRIBA Project. In ELISA workshop at ICSM, 2003.

[3] Mo Budlong. Sams Teach Yourself COBOL in 21 Days. Sams Publishing, 1999.

[4] Aberdeen Group. Legacy Applications: From Cost Management to Transformation. Executive White Paper from Aberdeen Group, March 2003. Can be found at http://www.aberdeen.com/2001/research/03038126.asp.

[5] Roel Wuyts. Declarative Reasoning about the Structure of Object-Oriented Systems. In Proceedings of TOOLS USA '98, 1998.

[6] Mike Murach, Anne Prince, Raul Menendez. Murach's Structured COBOL. Mike Murach & Associates, 2000.