Verifying the design of a Cobol system using Cognac

Andy Kellens, Kris De Schutter, and Theo D´Hondt

Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2
B-1050 Brussels
{akellens | kdeschut | tjdhondt}@vub.ac.be

1 Introduction

A property of large-scale, industrial systems is that they are intended to be used and maintained over a long period of time. In order to keep such large systems maintainable, it is important that developers respect the various rules that underlie the design of such systems during the subsequent evolutions of the system. These design rules can range from low-level naming conventions and coding guidelines, over the correct use of frameworks to the different constraints that are imposed by the architecture of the system.

Serving as a testimony to this problem is the amount of effort that has been devoted — both in academia and industry — to tools and approaches that aid in verifying design rules with respect to a system's source code. Examples of such tools are low-level code checkers such as Lint [3] and CheckStyle [1], tools such as Ptidej [2] that enforce design patterns, approaches like Reflexion Models [5] that verify a high-level specification (architecture) of a system with the source code and so on.

However, it seems that the vast majority of these tools neglects the Cobol language, which is still one of the most prevalent languages in industry. In this presentation we discuss *Cognac*, our approach that offers a general framework for documenting design rules in Cobol code and verifying their validity with respect to the implementation.

2 Context

The context of our work is a fairly large case study (500KLoc) we are conducting together with the Flemish company inno.com that has recently designed a new Cobol system for a Belgian bank. Their interest in verifying this design with respect to the implementation is three-fold:

- The implementation of the system has been out-sourced, resulting in that our industrial partner is interested in knowing whether the external partner respected the intended design and the provided coding guidelines/naming conventions;

- The system is expected to be in use for 20 to 25 years, resulting in that keeping the system maintainable is a valuable asset;
- The system is being implemented in various phases spread over multiple years, during which novel functionality is added. Our industrial partner wants to assure that during these phases the design is respected and wants to assess possible violations of the design.

3 Outline of our approach

Our tool — Cognac — offers developers a common framework to document and verify design rules in Cobol systems. Cognac is developed as an extension to our IntensiVE tool suite [4]. In a nutshell, the main idea of IntensiVE is to document design rules by grouping source-code entities in so-called *intensional views*: sets of source-code entities that belong conceptually together and that are defined by means of a logic program query (expressed in the SOUL language [6]). Either by specifying multiple, alternative definitions for one intensional view, or by imposing constraints over intensional views, design rules can be expressed using the tool. IntensiVE offers a number of subtools that allow for the verification of these design rules with respect to the source code and offer developers detailed feedback concerning possible violations.

Reasoning about Cobol posed a number of interesting challenges. Therefore, Cognac makes the following extensions to IntensiVE:

- There exist different variants of the Cobol language, each specifying a large amount of different language constructs. In order to deal with this problem, we have implemented a customisable island-based parser. Such an island-based parser allows us to extract only the information that is necessary for the analyses we wish to express from Cobol source code;
- We have implemented a set of SOUL predicates that reason about the Cobol parse tree, such that we can define intensional views (and constraints over these views) over such programs. This set of predicates consists of basic predicates that allow to query the structure of Cobol programs, predicates that retrieve relations between the various source-code entities, as well as predicates that e.g. extract information from embedded SQL statements;
- By reasoning purely over parse trees, we were restricted in the number of interesting design rules that can be expressed. Therefore, we complimented the set of SOUL predicates with two static analyses. One analysis is used to resolve *call* statements in the source code and link these statements to the actual Cobol programs that might get invoked. The second analysis — data field aliasing — conservatively computes aliases between different data fields in Cobol programs.

4 Design rules in the case study

In this section, we take a brief look at three of the design rules that we have documented in the case study. During the presentation, a more in-depth look at

these design rules will be given, along with details about how we documented them using Cognac.

Section layering In the case study under investigation, the designers of the system introduced a clear layered structure in the individual Cobol programs as a means to make the control flow more explicit. More specifically, the various sections in each program were divided into separate layers in which sections in one layer are only allowed to invoke sections in the same, or a lower layer. This design rule is reflected in the source code by means of a simple naming convention: each section's name is prefixed with a letter grouping sections at the same level using the same letter. From within each section, only sections may be invoked with the same starting letter, or with a letter that comes later in the alphabet. During the presentation, we show how to document this design rule by creating an intensional view that groups all callers and callees of sections, and by imposing a constraint over the elements of this intensional view specifying that for all pairs of callers and callees, the first letter of the callee should be the same or come later in the alphabet than the first letter of the caller.

Copybook - linkage correspondence A Cobol program that can be called from within another program needs to declare a linkage section that specifies the data definition of the arguments that it expects as input. In our case study, one design rule that needs to be obeyed is that, if a program calls another program, it uses the same data definition for the argument of both caller as well as callee. In order to ease this correspondence, a copybook is used that contains the data definition and that should be included in the linkage section of the called program as well as in the calling program. Since this pattern however is not enforced by the language itself, we have documented it using intensional views.

Database modularity The case study we investigated is designed in a component-oriented fashion. In the system, the various components consist of a top-level program that serves as the component's interface, along with a number of programs to which this top-level program delegates particular requests. Also associated with each component is a set of database tables that contain the persistent data which the module is responsible for. In order not to break this modularity, only programs from within one particular module are allowed writing access to the tables associated with that module. All other programs need to retrieve and manipulate data via the interface program of that module. Preferably also, the number of programs within a module that are allowed to write to the associated tables is limited. As we will show in the presentation, in order to verify this design rule, we opted to use a more pragmatic approach in which we use a visualisation as a means to provide the original designers of the system with feedback concerning the use of database tables in the current implementation.

Acknowledgements

Andy Kellens is funded by a research mandate provided by the "Institute for the Promotion of Innovation through Science and Technology in Flanders" (IWT Vlaanderen). Kris De Schutter received support from the Belgian research project AspectLab, sponsored by the IWT Vlaanderen.

References

1. Checkstyle, December 2006. `http://checkstyle.sourceforge.net`.
2. Y. Guéhéneuc. Three musketeers to the rescue – meta-modeling, logic programming, and explanation-based constraint programming for pattern description and detection. In *Workshop on Declarative Meta-Programming at ASE 2002*, 2002.
3. S.C. Johnson. Lint, a c program checker. In M.D. McIlroy and B.W. Kemighan, editors, *Unix Programmer's Manual*, volume 2A. AT&T Bell Laboratories, seventh edition, 1979.
4. K. Mens, A. Kellens, F. Pluquet, and R. Wuyts. Co-evolving code and design with intensional views: A case study. *Elsevier Journal on Computer Languages, Systems & Structures*, 32(2-3):140–156, 2006.
5. G. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *Symposium on the Foundations of Software Engineering (SIGSOFT)*, pages 18–28, 1995.
6. R. Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, January 2001.