Cognac: a framework for documenting and verifying the design of Cobol systems

Andy Kellens
Vrije Universiteit Brussel
akellens@vub.ac.be

Kris De Schutter
Vrije Universiteit Brussel
kdeschut@vub.ac.be

Theo D'Hondt
Vrije Universiteit Brussel
tjdhondt@vub.ac.be

Luc Jorissen
inno.com
Luc.Jorissen@inno.com

Bart Van Passel
inno.com
Bart.VanPassel@inno.com

Abstract

For any non-trivial software project, architectural drift is a well-known problem. Over time, the design rules and guidelines governing the software project are no longer obeyed, resulting in that the software becomes more difficult to maintain. While there exist numerous tools — such as code checkers, architecture and design checkers, and source code query languages — that aid in alleviating this problem none of these approaches are tailored towards supporting one of the main languages still in use today in industry, namely Cobol. In this paper we present Cognac, an extension of the IntensiVE tool that allows for documenting and verifying design rules in Cobol systems. Next to discussing the architecture of Cognac, we present a validation of our tool on an industrial, large-scale Cobol system.

1. Introduction

An intrinsic property of large-scale software systems is that they are intended to be used and maintained over a long period of time. One of the key factors that influences the maintainability of such systems is whether or not developers respect the rules and guidelines that are imposed by the system's design and architecture. Such design rules, which can range from simple naming conventions over the use of particular idioms to a description of how the different modules of the system work together, aid in ensuring the correct functioning of the system as well as making the system's implementation consistent and easier to understand.

Unfortunately, these design rules are often only described in an informal, implicit way by means of, for example, UML diagrams, architectural description languages or natural language. Moreover, the task of manually ensuring that the design rules are obeyed in the source code is tedious and time consuming. Due to the lack of of automated support, it is not guaranteed that during development and maintenance of the system the design rules in the source code are not violated. Over time this can lead to source code that becomes more difficult to comprehend and that gradually deviates from the original design and architecture, thus hampering the maintainability of the system.

Testimony to this problem is the large amount of effort (both in industry and academia) that has been invested in creating tools that aid developers in verifying the source code of a system either to a set of common rules and bad smells (e.g. Lint [9], CheckStyle [1]) or that allow for the verification of a description of (part of) the design of a system with respect to the implementation (e.g. Reflexion Models [19], Ptidej [6], FEAT [20]). While these tools alleviate the above problem, they are mostly targeted to modern-day object-oriented languages such as Java. As such, they neglect a large segment of systems written in languages such as Cobol. Especially since a large portion of the systems in use and maintained today in a business setting are still written in Cobol (and even new systems are developed in this language), support for verifying and maintaining the design of such systems with respect to the source code is necessary.

In this paper we present *Cognac*, a framework for documenting design rules in Cobol programs and for verifying the validity of these design rules with respect to the actual source code of the Cobol programs. Cognac is based on our previous work on declarative meta programming and is implemented as an extension to the Soul language [23] and our IntensiVE tool suite [13]. Cognac provides more than a mere extension of our tools for the Cobol language, taking some peculiarities of the language and the kinds of systems that are reasoned over into account as well. As a validation, we demonstrate the applicability of our approach by documenting a number of design rules taken from a real-life, industrial Cobol application.

2. IntensiVE

We start this paper by describing IntensiVE, which Cognac is an extension of. IntensiVE[1] is a tool suite, written in VisualWorks Smalltalk, that offers software developers a general framework for documenting and verifying design rules. To this end, IntensiVE proposes to group together source-code entities (such as classes, methods, functions, programs, and so on) that, according to a particular property, belong together in a so-called *intensional view*. For example, if we document particular design rules in an OO system concerning "getter" methods, we would create an intensional view that contains all getter methods in the system. Rather than specifying this set by means of enumerating its elements, an intensional view is defined by means of an *intension*. An intension is an executable meta program that, upon evaluation, yields a set of *tuples* representing the source-code entities belonging to the intensional view. For example, one possible intension for the intensional view that groups all getter methods in a Java program is:

```
1       ?class isClassDeclaration,
        ?class definesMethod: ?method,
3       ?method methodDeclarationHasName: {get*}
```

In order to express intensions, the IntensiVE tool uses the Soul language. Soul is a logic programming language inspired by Prolog, using a Smalltalk-like keyword-based syntax. Soul is complemented with a number of libraries offering predicates that make it possible to reason over Smalltalk and Java programs. The above intension consists of three logic conditions. The first condition binds the logic variable `?class`[2] to all classes in the system. In condition 2, the variable `?method` is bound to all methods implemented by the class bound to the variable `class`. The third condition restricts the bindings of the `?method` variable to those methods whose name starts with the prefix "get". This intension will thus group all getter methods in the system into an intensional view based on the naming convention that all such methods must start with "get".

Aside from the concept of intensional views, IntensiVE makes it possible to impose verifiable constraints over intensional views. In its current incarnation the model offers two types of constraints: multiple alternative intensions for a particular intensional view, and *intensional constraints*. The idea of multiple alternatives is based upon the observation that the entities belonging to an intensional view can often be described by multiple, equivalent intensions. The concept of a getter method is not only — as illustrated above — defined as all the methods starting with the prefix "get", but can also be defined as the set of all methods that return the value for a field. We could write this down using the following intension:

```
1       ?class isClassDeclaration,
        ?class definesMethod: ?method,
3       ?field isFieldInClass: ?class,
        ?method methodReturns: ?field
```

When specifying multiple, alternative intensions for the same intensional view, the constraint lies in the fact that, upon evaluation, all intensions should yield the exact same set of source-code entities. If in the above example for instance a method would exist that returns the value of a field, but that does not start with the prefix "get", it will be considered a violation of the constraint. Conversely, any method named "get" that does not return the value of a field will also be considered a violation of the constraint.

Furthermore, *intensional relations* can be used for imposing a constraint over the elements belonging to one intensional view (so-called *unary relations*), or between the elements of two intensional views (*binary* intensional relations). For example, suppose we want to document the relation that all methods that change the state in the system should contain an invocation to the persistence mechanism. After having created two intensional views respectively grouping all state changing methods and all methods implementing the persistence mechanism, the above design dependency can be expressed using the following binary intensional relation:

```
∀ ?statechange ∈ State Changing :
  ∃ ?persistence ∈ Persistence :
    ?statechange.method methodCalls:
        ?persistence.method
```

A developer can specify an intensional relation by specifying the intensional views involved in the relation, the relation quantifiers (in this case ∀ and ∃), and a Soul predicate (in the example above: `methodCalls:`) that indicates how the entities in the intensional views are related. Note that the elements belonging to an intensional view are tuples. In the above intensional relation we specified that the value of the `?method` variable from the intension of the *State Changing* intensional view (`?statechange.method`) must contain a call to the method implementing the persistence mechanism (`?persistence.method`).

IntensiVE also offers support to deal with explicit exceptions to a constraint. A user of the tool suite can declare that particular source-code entities should be explicitly included or excluded from an intensional view, or that a relation explicitly holds for a particular source-code entity.

Although IntensiVE offers a simple model for documenting design rules, in the past it has proven to provide ample facilities for documenting a wide range of different kinds of rules in object-oriented systems [10, 13, 14]. Moreover, in addition to specifying intensional views and constraints over these views, the tool suite offers a number of sub-tools that offer developers detailed feedback concerning violations of rules and that make it possible to integrate

[1] http://www.intensive.be

[2] Note that variables in Soul are indicated with a question mark.

the verification of the design rules with the standard unit testing process. The integration with the Mondrian [15] visualisation framework also makes it possible to create visualisations based on intensional views.

3. Cognac

Cognac is our extension to the IntensiVE tool suite that makes it possible to document design rules in Cobol programs. Although IntensiVE was conceived with language-independence in mind, providing support for Cobol did pose a number of challenges:

- Due to the complexity of the Cobol language, our approach needs to cope with a large variety of language constructs and possible Cobol dialects;
- Since it is our goal to apply Cognac to large systems, it needs to be scalable both in terms of memory consumption as well as a reasonable execution time;
- In order to express some interesting design rules, the information obtained from parsing the Cobol code does not suffice. Therefore, a number of additional static analyses are provided.

In the next sections, we take a look at the mechanics behind Cognac and how they tackle the above challenges. More particularly, we discuss the island-based parsing technique that lies at the root of Cognac and that makes it possible for us to deal with the various language constructs of Cobol on a by-need basis. Furthermore, this island-based parsing aids in managing the memory consumption of our tool as only the information that is needed for expressing the desired design rules is extracted from the source code. Furthermore, we take a look at the library of logic predicates that we provide the Soul language with such that intensional views and relations can be written over Cobol programs. We finish this section by discussing the static analyses that we have incorporated in our tool.

3.1. Island-based parsing

Cognac takes as input the source code of a software system. While this source code contains all relevant data, extracting this data from it poses a major challenge.

Cobol is, by now, a 50 year old language which predates the structured programming approach. It knows of no functions[3], has no support for local data, and allows for some very peculiar control flow. Extensions to functionality (e.g. XML processing) are not offered by means of APIs, but are provided through extensions of the Cobol language itself. Its syntax, which is as close to natural language (English) as its designers could get it, cannot be described by means of a context-free grammar. It requires pre-processing (which is a smaller annoyance), and in real-life applications mixes in for example SQL and CICS code which have grammars of their own.

While there exist some publicly available Cobol parsers (VS-Cobol-II [12], OpenCobol), they can rarely be applied to real industrial code as-is. They do not cover enough of the Cobol specification to parse the entire structure or gloss over SQL and CICS extensions, and they require pre-processing passes which are not always possible due to missing (e.g. proprietary, infrastructural) files. For the purpose of Cognac we have therefore opted for a new parser with two specific features:

- **Incomplete data:** The parser should be able to handle incomplete input. As external partners to a company, one is often not granted access to the entire system nor to the development environment. Consequently, the parser should be able to extract a maximum of information from the given source code, whether it is complete or not;
- **Customisability:** The documentation of different design rules requires knowledge about different language constructs. For example, if we want to reason about program calls, we are mostly interested in `CALL` statements. Due to the extent of the language, it is unfeasible to provide a sufficiently detailed parser that is able to deal with all of Cobol's language features up front. Rather, we need a parser that can be adapted (in a reasonable amount of time) to deal with the language features that are necessary for the documentation of a specific set of design rules.

We cover these features by setting up a Cobol parser by means of *island grammars*. Moonen [17] defines these as follows:

> An island grammar is a grammar that consists of detailed productions describing certain constructs of interest (the islands) and liberal productions that catch the remainder (the water).

Figure 1 illustrates how this works in practice. Figure 1 (a) presents a simple Cobol program (taken from [11]) with all reserved keywords displayed in bold. It should be clear even from this simple program that there are many keywords to cover for parsing it in full. (A full Cobol parser typically has to deal with over 500 such keywords, many of which are context dependent.) In contrast, Figure 1 (b) shows the view taken by an island-based parser where we only care about reconstructing the control flow. As you can see, most of the code is now ignored (the *water*, shown in gray). The parser mostly scans for the `PROCEDURE DIVISION` marker, and then looks out for verbs (e.g. `IF`, `OPEN`, `SET`, `MOVE`, etc.) and dots

[3] Functions have been added to later revisions, but their use is not mandatory nor common.

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. TOOLS/LOGFILE.
ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
  FILE-CONTROL.
   SELECT LOGFILE ASSIGN TO "FILES/LOGFILE.TXT",
   ORGANIZATION IS SEQUENTIAL.
DATA DIVISION.
 FILE SECTION.
   FD LOGFILE DATA RECORD IS LOGFILE-RECORD.
   01 LOGFILE-RECORD PIC X(2048).
 WORKING-STORAGE SECTION.
   01 LOGFILE-STATUS     PIC 9 VALUE ZERO.
      88 LOGFILE-IS-OPEN       VALUE 1.
 LINKAGE SECTION.
   01 LOGFILE-ENTRY.
      05 LOGFILE-VERB  PIC X(12).
      05 LOGFILE-NAME  PIC X(32).
      05 LOGFILE-DATA  PIC X(1024).
PROCEDURE DIVISION USING LOGFILE-ENTRY.
 IF NOT LOGFILE-IS-OPEN
    OPEN EXTEND LOGFILE
    SET  LOGFILE-IS-OPEN TO TRUE.
 MOVE  LOGFILE-ENTRY TO LOGFILE-RECORD.
 WRITE LOGFILE-RECORD.
 GOBACK.
```

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. TOOLS/LOGFILE.
ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
  FILE-CONTROL.
   SELECT LOGFILE ASSIGN TO "FILES/LOGFILE.TXT",
   ORGANIZATION IS SEQUENTIAL.
DATA DIVISION.
 FILE SECTION.
   FD LOGFILE DATA RECORD IS LOGFILE-RECORD.
   01 LOGFILE-RECORD PIC X(2048).
 WORKING-STORAGE SECTION.
   01 LOGFILE-STATUS     PIC 9 VALUE ZERO.
      88 LOGFILE-IS-OPEN       VALUE 1.
 LINKAGE SECTION.
   01 LOGFILE-ENTRY.
      05 LOGFILE-VERB  PIC X(12).
      05 LOGFILE-NAME  PIC X(32).
      05 LOGFILE-DATA  PIC X(1024).
PROCEDURE DIVISION USING LOGFILE-ENTRY.
 IF NOT LOGFILE-IS-OPEN
    OPEN EXTEND LOGFILE
    SET  LOGFILE-IS-OPEN TO TRUE.
 MOVE  LOGFILE-ENTRY TO LOGFILE-RECORD.
 WRITE LOGFILE-RECORD.
 GOBACK.
```

Figure 1. (a) Cobol program (taken from [11]) on the left showing all keywords in bold. (b) Same Cobol program on the right seen through the eyes of an island-based parser with a focus on control flow. Everything in grey gets ignored. The bold parts are used to reconstruct the control flow.

(shown underlined) from which it can deduce the control flow (the *islands*).

The island-based parser forms the front part of the importer for Cognac. It generates an abstract syntax tree, albeit a partial one due to the setup of the parser. (This counts as an additional advantage though, as we do not need to store the entire parse tree of a Cobol program, but only the parts that are of interest, thereby minimising the memory overhead of Cognac.) This AST is finally transformed into the Cognac structure which forms the basis for being queried.

3.2. Logic reification

After the Cobol source code has been loaded into Cognac, a developer can start documenting design rules by means of intensional views and constraints over these views. In order to write down the intension of an intensional view and the predicate of intensional relations, we have extended the Soul logic query language with a library of logic predicates that allow for reasoning over the imported Cobol code. Table 1 shows an excerpt of the library that we have provided for Cognac. Roughly, these predicates can be divided into two groups: predicates that reason about the structural reification of a Cobol program, and predicates that express more semantic relationships between source-code entities.

In the first category, we find predicates such as `?program isProgramWithIdentifier: ?id`, `?section isSectionInProgram: ?program`, and so on that reify the various program entities in a Cobol program and that allow for retrieving the general hierarchical composition of these entities (e.g. all calls in a program, all paragraphs in a section, ...). These predicates reify the parse trees obtained by the island-based parser. In other words, for each Cobol entity that is retrieved by the parser, we provide a corresponding set of predicates that allow for querying that kind of Cobol entity and its basic relationships. Note that, if in the configuration of the parser particular entities are not parsed, the usage of the corresponding predicates in logic conditions will result in that the logic conditions fail.

The second category of predicates make it possible to retrieve relationships between such source-code entities, such as calling relationships between programs, the usage of fields, embedded SQL statements, and so on. Some of this information can be extracted directly by analysing the parse tree of the Cobol program. For example, the predicate `?section sectionPerformsSection: ?callee` binds all sections that are called from within section `?section` to the logic variable `?callee` by analysing `PERFORM` statements. While the callee of a `PERFORM` statement is purely a string, this predicate will traverse the parse tree in order to identify the actual section that is being called. Other predicates (such as `?exec execStatementUsesTable: ?table`) make use of a separate parser in order to extract the required information from the SQL code. Finally, predicates such as `?field mayAliasWith: ?aliasField` require a light-weight static analysis of the program in order to obtain the necessary information. In the next section we will take a more in-depth look at these predicates.

Structural reification	Source code relationships
Programs	*Calling relationships*
`?program isProgram`	`?call callWithTarget: ?string`
`?program isProgramWithIdentifier: ?identifier`	`?call callsProgram: ?program`
`?program programIncludesCopybook: ?copybook`	`?call transitivelyCallsProgram: ?program`
Sections	`?call callUsingField: ?field`
`?section isSectionWithName: ?name`	`?program programUsingField: ?field`
`?section isSectionInProgram: ?program`	`?section sectionPerformsSection: ?callee`
`?section isSectionWithName: ?name inProgram: ?program`	`?section sectionPerformsParagraph: ?par`
Paragraphs	*Embedded SQL*
`?paragraph isParagraph`	`?exec isExecStatementInProgram: ?program`
`?paragraph isParagraphInProgram: ?program`	`?exec execStatementUsesTable: ?table`
`?paragraph isParagraphInSection: ?section`	`?exec execStatementWritesToTable: ?table`
Statements	*Move information*
`?move isMoveStatementInProgram: ?program`	`?field fieldIsSenderOfMove: ?move`
`?call isCallStatementInSection: ?section`	`?field fieldIsReceiverOfMove: ?move`
`?perform isPerformStatementInParagraph: ?par`	*Field aliasing*
Fields	`?field mayAliasWith: ?aliasField`
`?field isFieldInProgram: ?program`	`?field mayTransitivelyAliasWith: ?aliasField`
`?field isFieldInLinkageSection: ?linkage`	
`?linkage isLinkageSectionInProgram: ?program`	

Table 1. Excerpt of the library of predicates that is offered by Cognac.

Since we want to maximise the efficiency of the library of predicates that we defined, our implementation rigourously makes use of caching. For example, rather than having to traverse the parse tree multiple times when retrieving the various kinds of statements in a Cobol entity, this information is cached at different levels in the parse tree. Similarly, relatively expensive computations, such as finding the transitive call chain of a program, are also computed only once and cached afterwards.

3.3. Extracting static information

Although a Cobol parse tree offers a wealth of information, certain kinds of information are not directly accessible from such parse trees. We implemented the following static analyses in order to complement the information retrieved from the parse tree:

Call resolution One interesting source of information in Cobol programs are the various calling relations between Cobol programs. In order to retrieve this information from the source code, we need to analyse the `CALL` statements. For example, the statement `CALL 'Example' USING CALL-PARAM` indicates a call to the program named `Example` using the data field `CALL-PARAM` as an argument. While the first argument of the `CALL` in the simplest case is a string indicating the program name that gets called, it can also be a data field (e.g. `CALL PROG-SUB USING CALL-PARAM`). In this case, it is not certain which program will get called, since the value of `PROG-SUB` can be altered at runtime. Cognac implements a simple static analysis that, for `CALL` statements where the callee is stored in a data field, identifies possible programs by looking at data field initialisers (e.g. the field `PROG-SUB` might be initialised to the value `'Example'`) and the allocation of string literals to fields.

Field aliasing While the call resolution we discussed above allows us to give a coarse-grained approximation of the control flow in a Cobol system, Cognac also implements a field aliasing algorithm that offers a light-weight analysis of the data fields within the application. This analysis will collect for each data field in the system a set of other data fields which may possible alias with that particular field. For example, the usage of a `MOVE THIS TO THAT` statement, which moves the contents of one data field (`THIS`) to another data field (`THAT`) introduces an alias between the two involved fields. Similarly, the arguments of a call of a program result in that two different data fields are possibly pointing to the same piece of memory. Note that we take a conservative approach to calculating the aliases of a particular field: if a field is in the alias set of another field, this does not necessarily mean that at runtime they will get used for the same data.

4. Validation

4.1. Case

In order to demonstrate the applicability of Cognac, we will use our tool to document a selection of design rules that originate from a large-scale industrial case study. In particular, we analyse a banking system consisting of roughly 500KLoc of Cobol code obtained from our industrial partner, the Flemish company inno.com. The system has been in development since 2005 and is expected to continue to remain in production for a long period of time (20+ years). From the point of view of inno.com, the verification of their design rules can play an important role in the development process for three reasons:

- From the start, the actual development of the system has been outsourced to an external partner. Consequently, our industrial partner is interested in verifying to what extent the developers have respected the design and development guidelines that they were provided with;
- As the system has to be maintained over a long period of time, it is important to keep the source code as consistent as possible with respect to the design. In order to prevent the source code quality from deteriorating and thereby increasing the effort needed for maintaining the system, automated support for verifying the design with the implementation is needed;
- Throughout the various iterations of the system — spread over multiple years — the functionality of the system will be extended. Our industrial partner wants to assure that the intended design is respected in the first iteration and wants to assess violations of this design during subsequent iterations.

4.2. Design rules

In the subsequent sections we take a look at a number of design rules that originate from the above case study, relate their importance to the system, and discuss how we were able to document them and how well they were respected in the source code. We obtained these design rules by interviewing the original designers of the system. Although there exist a number of documents specifying the functional design of the system, for this first experiment we opted to express a number of more general design rules that ought to be respected in the case study.

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.

Although this design rule boils down to a fairly simple naming convention, due to its importance and since it serves as a nice illustration of our approach, we documented it using Cognac. To this end, we created an intensional view named *Sections with callees* that groups all sections in the Cobol system, together with the sections that are called by each section. The intension for this simple intensional view is the following:

```
?section sectionPerformsSection: ?callee
```

Afterwards, we declare a unary intensional relation on this intensional view, that states that for all callees, the condition must hold that the callee's first letter must come later in the alphabet (or be the same) than the first letter of the caller.

We express this intensional relation as the following:

```
∀ ?entity ∈ Sections with callees :
   ?entity.section isSectionWithName: ?callerName,
   ?entity.callee isSectionWithName: ?calleeName,
   [(?callerName at: 1)  <=   (?calleeName at: 1)]
```

The relation states that for all entities belonging to the *Sections with callees* intensional view, three conditions must hold. The first two conditions extract the name of the caller and the callee sections from the entities in the view (note that this view contains tuples of sections and callees). The third condition is a Smalltalk block that extracts the first letter from both the caller and the callee and that verifies if the first letter of the caller comes before the first letter of the callee. When verifying the validity of this relation with respect to the source code, our tool suite provides developers with feedback concerning the discrepancies between the documented design rule and the implementation (see Figure 2). The top-most pane shows the entities for which the relation holds; all inconsistencies are shown in the bottom-left pane. Out of a total of 21653 calls between sections, 98.8362 % respected this design rule. We were able to identify a total number of 252 violations of this rule.

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

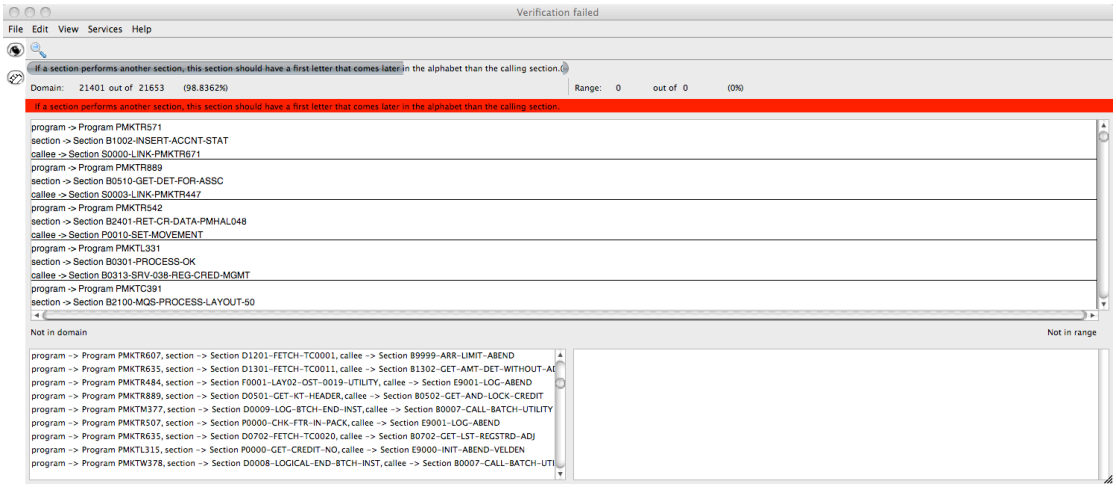

Figure 2. Feedback provided by our tool suite

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. We express the above design rule by means of a binary intensional relation. First, we create an intensional view named *Called programs* with as intension:

```
?program programCalls: ?call,
?call callCallsProgram: ?calledProgram
```

The above query retrieves all pairs of `?program` and `?calledProgram` between which there exists a possible calling relationship `?call`. Note that the above predicates make use of the call resolution analysis that was discussed in Section 3.3.

Next, we create a second intensional view *Program with copybook* that groups all programs together with the copybook that defines their linkage section data definitions. The intension for this view is:

```
?program programWithCopyStatement: ?copy,
?copy copyStatementInLinkageSection,
?copy copyStatementIncludesCopybook: ?copybook
```

This intension consists of three parts. The first condition retrieves all the copy statements in Cobol programs (copy statements are used to include a particular copybook). In the second condition, this set of copy statements is limited to those that are contained within the linkage section of the program. Finally, the third condition binds the logic variable `?copybook` to the actual copybook that was included in the linkage section.

Using the above two intensional views, we can now express the design rule as the following binary intensional relation:

∀ `?caller` ∈ *Called programs* :
  ∃ `?corresponding` ∈ *Program with copybook* :
    `?caller.calledProgram` *equals:* `?corresponding.program`,
    `?caller.program` *programIncludesCopybook:* `?corresponding.copybook`

The above relation verifies that for all called programs, the corresponding copybook in the linkage section of the callee is included by the caller. When verifying the above intensional relation, our tool reported on 42 violations of documented design rule. By manually inspecting each of these violations, we saw that they resulted from the usage of a utility library which the above design rule is not applicable to. In this library, one big copybook is used that includes the data definitions of all fields that are used in the library. Rather than including the specific corresponding copybook for a called program from the library, all clients of the library included this big, more general copybook. Since these invocations are exceptions rather than violations, we documented them as such by declaring them an exception to the intensional relation.

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.

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 cur-

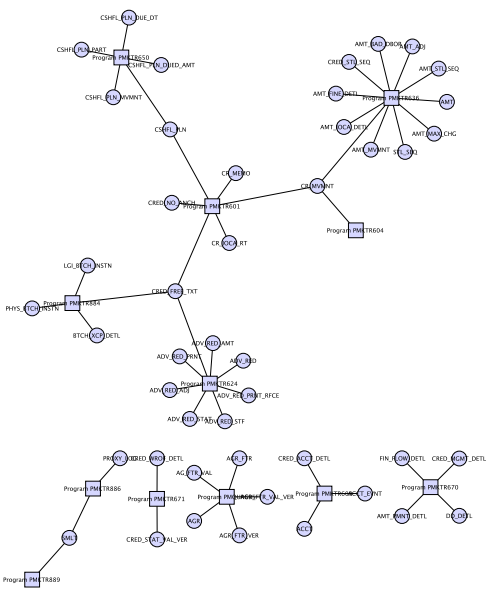

Figure 3. Visualisation of table usage

rent implementation. First, we created an intensional view named *Programs writing to tables* with as intension:

```
?stat isExecStatementInProgram: ?program,
?stat isExecSQLStatement,
?stat execSQLStatementWritesToTable:?table
```

Line 1 of this intension retrieves all Exec statements (this is the kind of statement in Cobol used for embedding SQL code). In line 2, these Exec statements are limited to those that actually contain an SQL query. Finally, line 3 extracts the names of the tables which the Exec statement writes to.

Rather than imposing constraints over this intensional view, we created a visualisation using IntensiVE in order to provide feedback concerning this design rule. An example of this feedback can be found in Figure 3. In this graph, the various programs are represented as rectangles; tables are represented as circles. An arrow between a table and a program indicates that the program possibly writes to the table via an embedded SQL query.

In the bottom part of the figure, we can see a number of "islands": isolated groups of programs and a number of tables that are written to by these programs. These islands illustrate the modularity of the database access: the tables that are local to a particular module are only written to from within the modules, as is also prescribed by the design rule. The top part of the figure on the other hand illustrates a possible violation of the intended design rule: in this part we see a larger cluster of tables and programs. Moreover, we see that there are a number of tables that are being written to from within various programs, possibly indicating a violation of the design rule. Notice that this visualisation only gives us an indication of possible violations; in order to fully assess whether the identified cluster is a violation of the design rule, an analysis by a developer is necessary.

Online programs In the case study, a distinction is made between offline programs that are executed in batch, and online programs. All of these online programs, and *only* these programs, should be called from somewhere in the system (if not they are redundant) and should include a linkage section (specifying that they take a particular input). In other words, according to this design rule, if a program is called and it is not an offline program, it should include a linkage section. Likewise, if a program includes a linkage section, then it should be an online program and it should be called from within the system.

We document this design rule by creating an intensional view with two alternative intensions:

1. `?program programCalls: ?call,`
   `?call callCallsProgram: ?calledProgram,`
   `?calledProgram isProgramWithIdentifier: ?identifier,`
   `not([’####M*’ match: ?identifier]),`
   `not([’####D*’ match: ?identifier])`
2. `?calledProgram isProgram,`
   `?linkage isLinkageSectionInProgram: ?calledProgram`

The first intension collects all programs that are called somewhere from within the case study and that do not contain an M or a D as the fifth character of the program name (these two letters are used to indicate offline programs). In the second intension, all programs are gathered that include a linkage section. Note that the use of multiple alternatives implies that both sets of source-code entities resulting from evaluating the intensions should be identical. In other words, if an online program is called but does not include a linkage section, it is marked as a violation. Conversely, any program that includes a linkage section should be called at least once from within the system and should not be an offline program (according to the naming convention). When verifying the validity of this design rule with respect to the implementation, we did not encounter any violations.

4.3. Performance

One of the goals for Cognac that we decided upon from the start was that it should be able to reason about large-scale, industrial systems. To this end, not only the runtime efficiency of the verification of the documented business rules is important, but also the memory consumption. As input for the case study, we took the entire source code of the system that we received from our industrial partner, amounting to 43 megabytes of Cobol code. We configured the island-based parser to extract the maximum of Cobol language features that is supported by the current version of Cognac. After the source code was parsed and loaded into Cognac, this resulted in a memory usage of 73 megabytes.

Cognac operation:	Time (in seconds):
Parsing	377s
Running the analyses	155s
Design rule verification	
Section layering	20s
Copybook - linkage correspondence	23s
Database modularity	1s
Online programs	3s

Table 2. Runtimes of Cognac operations and design rule verification.

An overview of the runtime efficiency of Cognac can be found in Table 2. For the entire case study, parsing and executing the analyses took just under 10 minutes[4]. For the verification of the first two design rules we discussed, about 20 seconds was needed. The last two design rules were computed in a couple of seconds. Although these numbers do not provide an empirical validation of the scalability of Cognac, they do provide some initial insights into the extent to which our tool can be applied on real-life case studies. While, in general, we do not believe that our tool can be used to provide developers feedback during the actual development process itself, it seems feasible to verify a large set of design rules overnight.

5. Related work

Parsing Cobol. Van Deursen et al. [21] present a number of analyses performed on legacy Cobol systems. They do this by extracting information by means of a Perl script which outputs comma-separated-value files of the relevant data. The analyses built on this data include program call graph reconstruction and database usage. Van Geet et al. [22] use a similar approach to enable reverse engineering by means of lightweight visualisation techniques. Again the necessary data is extracted with the help of a Perl script which scans the source code for the relevant statements. In essence, both are applying an island-based parsing technique much like the one presented in this paper, though they do so without the help of parser/lexer generators. In contrast, Moonen [18] applies an island grammar for enabling impact analysis in a Cobol banking system, making use of a parser generator. Similarly, the parser feeding data in Cognac also makes use of such tools.

Logic queries and Cobol. Lämmel et al. [11] define a possible aspect-oriented extension to Cobol, where they make use of logic based queries in the pointcut language. The pointcuts themselves are written down in a Cobol-like syntax, but translate into Prolog queries on the full abstract syntax tree of whatever Cobol program is being treated. The focus here is on aspect technology and weaver creation, so the logic queries are not available for other purposes.

Tools supporting design rules Although they do not support the Cobol language, there exist numerous tools that provide support for design rules. We can classify these tools in three categories:

- **Code checkers:** These tools, such as Lint [9], CheckStyle [1], P^3 [2], FindBugs [8] and many others provide a generally fixed set of rules expressing common mistakes, good coding style, platform-specific constraints, and so on, and allow for the verification of these rules with respect to the implementation of a system. While their strength lies in that they provide highly optimised and dedicated tools for supporting such regularities, often they can not be customised to support particular design rules that are specific to the design of one particular system;
- **Architectural and design checkers:** Tools such as RevJava [5], Ptidej [6] and Reflexion Models [19] are dedicated to verifying a high-level description of a system (e.g. design patterns, dependencies between components, ...) with respect to the source code. Similar to code checkers, these tools provide a dedicated tool for such kinds of design rules making it possible to for example automatically correct inconsistencies between source code and a design pattern implementation. Note that the advantage of our use of IntensiVE is that our tool is sufficiently versatile to support both low-level rules such as naming conventions and bad smells as well as the verification of more high-level rules with respect to the source code;
- **Meta-programming systems:** A final group of tools such as Law-governed Architecture [16], SCL [7], CCEL [3] and IRC [4]. Similar to our use of the Soul language in order to specify the intension of an intensional view, these tools offer developers a meta-programming language that makes it possible to write programs that verify design rules.

6. Future work and Conclusions

In this paper we have presented Cognac, a framework for documenting and verifying design rules in Cobol systems. It was our goal to obtain a tool that is versatile enough to document different kinds of design rules (as we demonstrated: from low level naming conventions, over database accesses to inter-module dependencies) in large, real-life case studies. To this end, we have opted to provide an

[4] On an Apple MacBook Dual-Core 2.16Ghz with 2G of RAM.

island-based parsing scheme in order to deal with the intricate complexity of the Cobol language. Furthermore, our tool provides a number of relatively simple static analyses that make it possible to extract more detailed call information and field aliasing information from the source code. The contributions of our work are the following:

- A customisable, island-based parser that supports a subset of the Cobol language and that can easily be extended;
- A library of Soul predicates that make it possible to reason about Cobol programs and that make use of the provided static analyses;
- An integration with the IntensiVE tool suite for documenting and verifying design rules.

As a validation, we presented the verification of a number of general design rules we obtained from the original designers of an industrial case study. A first direction of future work consists of the translation of the complete design of the system into intensional views and constraints over these views. To this end, we plan to (semi-)automatically convert the existing design documents (Rational Rose files) that describe the various modules in the system and the expected interactions into intensionsal views and constraints.

Furthermore, we plan to extend the functionality of our tool by enlarging the set of Cobol language constructs that is supported by our logic predicates, and by extending the precision of the analyses that we have implemented.

Acknowledgements

The authors would like to thank Kim Mens for his valuable feedback earlier drafts of this paper. 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 AspectLab research project, sponsored by the IWT Vlaanderen.

References

[1] Checkstyle, December 2006. http://checkstyle.sourceforge.net.

[2] C. Depradine and P. Chaudhuri. P^3: a code and design conventions preprocessor for java. *Software - Practice and Experience*, 33(1):61–76, 2003.

[3] C. Duby, S. Meyers, and S. Reiss. CCEL: A metalanguage for C++. In *USENIX C++ Technical Conference Proceedings*, pages 99–115. USENIX Assoc., 10-13 1992.

[4] M. Eichberg, M. Mezini, K. Ostermann, and T. Schäfer. Xirc: A kernel for cross-artifact information engineering in software development environments. In *Working Conference on Reverse Engineering (WCRE)*, pages 182–191. IEEE Computer Society Press, 2004.

[5] G. Florijn. Revjava - design critiques and architectural conformance checking for java software. Technical report, Software Engineering Research Centre (SERC), 2002.

[6] Y. Guéhéneuc and H. Albin-Amiot. Using design patterns and constraints to automate the detection and correction of inter-class design defects. In *Technology of Object-Oriented Languages and Systems (TOOLS)*, pages 296–305, 2001.

[7] D. Hou and J. Hoover. Using SCL to specify and check design intent in source code. *IEEE Transcactions on Software Engineering*, 32(6):404–423, 2006.

[8] D. Hovemeyer and W. Pugh. Finding bugs is easy. *ACM SIGPLAN Notices*, 39(12):92–106, 2004.

[9] S. Johnson. Lint, a c program checker. In M. McIlroy and B. Kemighan, editors, *Unix Programmer's Manual*, volume 2A. AT&T Bell Laboratories, seventh edition, 1979.

[10] A. Kellens. *Maintaining causality between design regularities and source code.* PhD thesis, VUB, 2007.

[11] R. Lämmel and K. De Schutter. What does aspect-oriented programming mean to Cobol? In *Aspect-Oriented Software Development (AOSD)*, pages 99–110, 2005.

[12] R. Lämmel and C. Verhoef. Semi-automatic Grammar Recovery. *Software—Practice & Experience*, 31(15):1395–1438, December 2001.

[13] K. Mens and A. Kellens. Intensive, a toolsuite for documenting and checking structural source-code regularities. In *Conference on Software Maintenance and Reengineering (CSMR)*, pages 239–248, 2006.

[14] 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.

[15] M. Meyer, T. Girba, and M. Lungu. Mondrian: An agile visualization framework. In *ACM Symposium on Software Visualization*, pages 135–144, 2006.

[16] N. Minsky. Law-governed systems. *Software Engineering Journal*, 6(5):285–302, 1991.

[17] L. Moonen. Generating robust parsers using island grammars. In *WCRE*, page 13, 2001.

[18] L. Moonen. Lightweight impact analysis using island grammars. In *Proceedings of the 10th International Workshop on Program Comprehension (IWPC 2002)*. IEEE Computer Society Press, June 2002.

[19] 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.

[20] M. Robillard and G. Murphy. Feat: a tool for locating, describing, and analyzing concerns in source code. In *25th International Conference on Software Engineering*, pages 822–823, 2003.

[21] A. van Deursen and T. Kuipers. Rapid system understanding: Two COBOL case studies. In *IWPC*, pages 90–97. IEEE Computer Society, 1998.

[22] J. Van Geet and S. Demeyer. Lightweight visualisations of COBOL code for supporting migration to SOA. In *3rd International ERCIM Symposium on Software Evolution*, oct 2007.

[23] 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.