Verifying the design of an outsourced COBOL system with IntensiVE

Andy Kellens, Carlos Noguera, Theo D'Hondt
Software Languages Lab
Vrije Universiteit Brussel
Pleinlaan 2
B-1050 Brussels, Belgium
Email: {akellens | cnoguera | tjdhondt }@vub.ac.be

Luc Jorissen, Bart Van Passel
inno.com
Heiststeenweg 131
B-2580 Beerzel, Belgium
Email: {luc.jorissen | bart.vanpassel }@inno.com

***Abstract*—Companies nowadays rely on outsourcing for the implementation of their software. While outsourcing can reduce the actual development costs for a piece of software, it can also reduce a company's control over the quality of the delivered software. In light of obtaining maintainable software, it is however important that the delivered software is well-structured and obeys the various design rules that were postulated by a company using an outsourcing partner. This paper reports on a collaboration between academia and industry, where the research tool IntensiVE has been applied to verify the design rules underlying a large-scale COBOL system. We describe how the IntensiVE tool was customized in order to support verification of the COBOL system, and how this academic tool aided in providing an initial quality assessment of the outsourced software system.**

I. Introduction

Outsourcing has become an important practice within the software development industry. Rather than developing software in-house, major companies rely more and more on external partners to implement their software systems. While this business model reduces the actual development cost of the software, it can reduce a company's control over the quality of the delivered software. To manage the problems associated with *external* quality, rigorous testing practices are often put in place that ensure that the delivered software exhibits the desired behavior. The *internal* quality of an outsourced software system however also plays an important role in the system's success. For an outsourced system to remain comprehensible and maintainable over an extended period of time, it is imperative that the system's source code is well-structured and obeys the design guidelines and coding practices imposed by the company that makes use of the outsourcing partner.

This paper reports on our experiences in applying the academic tool IntensiVE[1] ([1], [2], [3]) in the context of an industrial collaboration. The subject of this collaboration is a recently developed, large COBOL system owned by one of the major Belgian banks. While the actual design of the system was performed in-house, the implementation of the system was outsourced to an external partner. Due to the size of the investment, considerable effort was spent in creating a modular design for the system. Since the bank intends to use and maintain the system over an extended period of time, interest was expressed in assessing whether the outsourced code obeys the design, and in acquiring tool support for keeping source code and design consistent throughout the life cycle of the system.

IntensiVE is a mature academic tool suite that offers a dedicated means for documenting structural design rules (such as naming conventions, language idioms, design patterns, module dependencies), and verifying the validity of these design rules in the source code of a system. It consists of a number of tools that provide both textual and visual feedback regarding violations of design rules, and offers a tight integration with the surrounding VisualWorks Smalltalk development environment. In the context of the collaboration reported in this paper, the tool suite has been extended and customized in order to support a light-weight verification of the design rules underlying the industrial system.

The contributions of this paper are two-fold. First, we discuss the approach we took in verifying the various design rules that underly the implementation of the industrial system. Building upon our previous work [4] — that introduced our infrastructure for reasoning over COBOL systems — we discuss the customizations to IntensiVE that enable us to automatically verify the design documentation of the industrial case study with respect to the source code. Second, we describe the lessons learned from this case study, both from an academic as from an industrial point of view.

This paper is structured as follows. Section II introduces the COBOL system we investigate in this paper. We discuss the motivations for the industrial partner to verify the design information of this system. Section III provides a detailed of the different kinds of design information our industrial partner deemed interesting to verify. In Section IV we briefly discuss the IntensiVE tool suite, how it was customized to support reasoning over COBOL systems, and how IntensiVE supports the verification of the design information underlying the case study. The concrete results of this verification are discussed in Section V-A. Before concluding this paper in Section VII, we discuss the lessons learned from performing this case study in Section VI.

[1] see also http://www.intensional.be

II. Context and problem statement

The system under investigation in this paper is a large-sized (> 1 MLOC) industrial application that supports the core activities (*e.g.,* insurances and mortgages) of a major, Belgian bank. Despite that work on the system started as recent as 2005, the bank opted to develop the back-end of this new system in COBOL in order to ease integration with existing infrastructure. Assisted by the Flemish consulting company inno.com, a large effort was spent from the start in designing the system in a modular and extensible way using a component-based methodology. The system is decomposed in a number of components, each one implementing a number of use cases. Each component consists of a well-defined interface, and a set of associated database tables containing data local to that component.

While the system was designed in-house, the implementation of the system was outsourced to an off-shore company. As such, the Belgian bank and inno.com were interested in assessing whether the source code of the novel system obeyed the specified design of the system. In particular, they were interested in evaluating the adherence to the design during the development process, and in preventing design erosion.

The motivations for this are two-fold:

- Work on the system was planned over multiple iterations. During each of these iterations novel functionality was added to the system, or changes were made to the already present functionality. To prevent these iterations from having a negative impact on the internal quality of the system, and to acquire an initial assessment of this internal quality, the bank was eager to verify how well the delivered source code of each iteration follows the specified design.
- The system poses a considerable investment for the bank, and is planned to be used over an extended period of time (20+ years). Over this period, the bank expects to keep the software maintainable — either in-house or using an external partner. As such, it is imperative that subsequent evolutions of the system remain consistent with respect to the design of the system. If not, this can lead to design erosion, resulting in a system that becomes harder to understand and maintain.

III. Case study

In order to evaluate the internal quality of the system, our industrial partner identified two kinds of design information that must be verified: implementation guidelines and patterns and the validation of sequence diagrams that represent use cases.

A. Mapping object-oriented concepts onto COBOL programs

While the case study was designed in a component-based fashion, the COBOL language provides little or no support for this design paradigm. In a nutshell, a COBOL system consists of a set of programs, each composed out of a number of sections. These sections are then further broken down into paragraphs. In COBOL, programs can call their own internal sections/paragraphs, or can call the functionality of another program. To reflect the modular decomposition of the system in the actual source code, a set of implementation guidelines and patterns were specified. These guidelines and patterns describe how a component in the system is mapped onto the language concepts offered by COBOL.

For each component, a layering scheme is used in which one COBOL program serves as the interface of the component, a number of programs implement the various operations offered by the components, and a set of programs are dedicated towards manipulating the database tables local to that component. In order to preserve component boundaries, programs belonging to one component should use the interface of the other components instead of directly invoking the programs belonging to that component. Similarly, database manipulation within a particular component should be restricted to those database tables that belong to that component; other operations that wish to manipulate data should use the interface of the component responsible for that data.

Within a COBOL program, a similar layering scheme is used. One program can implement multiple operations, which are each mapped onto a top-level section in that program. A concrete implementation of the operation is then provided by further delegating to other sections within the program. These sections, in turn, can invoke the functionality offered by other components, or can decompose the implementation by calling auxiliary sections. The layering scheme however implies that from within the implementation of one operation, no other top-level (or higher-level) sections are invoked.

B. Implementation guidelines and patterns

To make the above implementation scheme explicit in the source code, a number of naming conventions, implementation guidelines and idioms are used. For example, the layering of sections *inside* a particular COBOL program is reflected by a naming convention that identifies the layer to which a section belongs by prefixing the name of the section. Top-level sections are prefixed with an 'A', second level with a 'B' and so on. In order for a developer not to violate the layering scheme, an implementation guideline was put in place that expresses that sections should only call sections starting with the same prefix, or a prefix that follows that prefix in the alphabet. Similarly, an implementation idiom is proposed to ensure that — when calling one program from within another — both caller and callee use the same (correct) data definitions.

C. Sequence diagrams

Next to the set of guidelines describing how the source code of the system should be structured, the design documentation of the case study contains a technical specification of the interactions between the various components and layers of the system. This specification consists of a description of the use cases of the system, that are expressed using UML sequence diagrams.

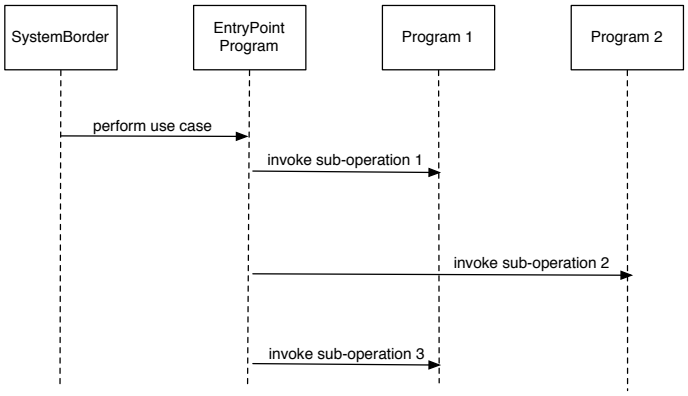

Fig. 1. Example of the description of a use case using a sequence diagram.

Figure 1 illustrates[2] this kind of documentation. For a particular use case, the sequence diagram identifies the COBOL program that is the point of entry for that use case, and describes the interactions with other COBOL programs (representing an interface to a component) that are necessary to handle the use case.

For example, figure 1 describes a use case that is initiated by calling a program `EntryPoint` from the `SystemBorder` (*e.g.,* a user request). In order to handle the use case, program `EntryPoint` relies on the functionality offered by two other programs, namely `Program 1` and `Program 2`. More precisely, the implementation of the use case imposes a specific order in which the other programs in the use case are called. Our example shows that first operation `sub-operation 1` on `Program 1` should be called, then `sub-operation 2` on `Program 2` and finally `sub-operation 1` on `Program 1` in order to correctly handle the use case. Note that this ordering describes the relative ordering of the calls within the sequence diagram. Intermixed with these calls, other programs are allowed to be invoked (*e.g.,* to deal with data retrieval, error handling, logging, ...). While not shown in our example, the sequence diagrams can contain additional information concerning the control-flow of the use case (such as branching statements and loops). Given that this information is expressed in a textual, informal way, we omitted it since it is of no use to us while verifying the design information.

Despite the fact that design documentation of the use cases is informal, it offers developers a wealth of information during the execution of maintenance tasks. Whenever the system needs to be altered, this documentation aids developers in identifying the entry point in the source code where the changes need to be made. Furthermore, the description of how the various components interact provides developers with an initial understanding of how the particular use case is implemented and thus can be adapted. Therefore, it is of interest to our industrial partner to know how well these sequence diagrams can be linked to actual source-code entities, and how well the source code obeys the description of the interactions between COBOL programs that is contained within them.

If tool support is available to verify whether design documentation and source code are consistent, such tool support can then also be used to co-evolve both artifacts upon evolution of the system.

[2]The figure shows an abstract representation of how the system is documented. Due to a non-disclosure agreement, we are not allowed to show concrete sequence diagrams from the case study.

IV. Verifying the design of the case study using IntensiVE

Although a manual assessment of the internal quality is feasible, due to the system's size such a process would be impractical and tool support is needed. During the maintenance phase of the system, development teams will strive for short release cycles. These short cycles would require the design of the system to be frequently verified, which — in case of manual verification — would be time-consuming and not cost effective.

In what follows, we discuss how our academic tool IntensiVE was applied to support the verification of the design knowledge underlying the industrial COBOL system. We start this section by providing a short overview of the IntensiVE tool suite and how it can be used to document and verify design rules. In particular, we take a look at how our tool was customized for reasoning over COBOL systems.

For a description of how our customization of IntensiVE was applied to verify the general implementation guidelines and patterns that express the layering scheme within individual components and the correct usage of data structures (cf. § III-B), we refer to [4]. In the remainder of the section, we focus on describing the process of verifying the sequence diagrams that document the use cases of the industrial system.

A. IntensiVE for COBOL

IntensiVE is a tool suite that is closely integrated with the VisualWorks Smalltalk development environment, and that offers developers a formalism and associated set of tools for documenting and verifying structural design rules. Using IntensiVE, design rules can be documented by means of *intensional views* and by imposing *constraints* over these intensional views.

An intensional view is a set of source-code entities (e.g. classes, methods, functions, ...) that conceptually belong together. Rather than specifying this set of source-code entities by enumeration, this set is defined by an *intension*, an executable query that — upon evaluation — yields the set of entities belonging to the intensional view. IntensiVE uses the logic program query language SOUL [5], a dialect of Prolog [6] that is tailored towards reasoning over source code, to define intensional views. SOUL is language independent, providing libraries of predicates for reasoning over OO program written in Smalltalk [7], [8], Java [8], [9] and C(++) [10]

An important step towards applying IntensiVE to the COBOL system consisted of extending the SOUL with support for querying COBOL programs.

In order to implement the design verification required by the case study, SOUL was extended with:

- A parser for the COBOL dialect that was used at the bank in order to obtain a queryable representation of COBOL programs. To this end, we opted for an island-based parser [11]. Such a parser does not extract a complete reification of the source code, but will only extract a structural representation of a subset of the language constructs present in a language. Since our analyses do not require access to all source-code elements, such an island-based parser proved to be a good choice for dealing with the inherent complexity of the COBOL language;
- A simple static analysis on top of these partial parse trees, that extracts a call-graph and a data-flow graph from the structural representation;
- A library of SOUL predicates that offers a declarative means to query the information contained in partial parse trees. This library also provides predicates to access the information in the call-graph and data-flow graph.

To illustrate how IntensiVE is used to provide the functionality required by the industrial partner, we document a design rule that expresses that all sections in a program can only call sections in a lower layer (i.e. sections of which the name prefix comes later in the alphabet). To document this design rule, we create an intensional view *Sections with callees* that consists of pairs of sections in the COBOL system in between of which there is an invocation relationship:

```
if ?caller sectionPerformsSection: ?callee
```

While a full explanation of the logic program query language SOUL lies outside of the scope of this paper, we briefly describe the logic query that defines this intensional view and, upon evaluation, calculates the set of entities belonging to that view. The query consists of a single logic condition that expresses a relationship between two logic variables *?caller* and *?callee*. To this end, the logic condition expresses that there should exist an invocation relationship between *?caller* and *?callee* that is verified using the predicate `sectionPerformsSection:`. Note that in contrast to Prolog, SOUL indicates logic variables by means of a question mark, and that a keyword-based syntax (similar to Smalltalk) is used to write down the predicates. Evaluation of the above logic query by the SOUL logic solver will then result in that the COBOL system is queried for all possible bindings of the logic variables *?caller* and *?callee* for which the above condition holds.

Actual design rules are documented by imposing intensional constraints over intensional views. An intensional constraint is a quantified condition that is applicable to the entities belonging to one (or more) intensional views. Similar to intensional views, this quantified condition is expressed using the SOUL language.

For example, we document the above design rule by creating an intensional constraint:

$\forall$ *?invocation* $\in$ *Sections with callees* :
?invocation.caller `isSectionWithName:` *?callerName*,
?invocation.callee `isSectionWithName:` *?calleeName*,
[*?callerName* $<=$ *?calleeName*]

The above constraint expresses that for all elements (*?invocation*) belonging to the intensional view *Sections with callees*, the calling section is only allowed to invoke sections that come later in the alphabet. This design rule is verified by querying both caller and callee for their name (which will be bound to *?callerName* and *?calleeName* respectively), and then verifying in the last logic condition whether the *?callerName* is smaller (or equal) to the *?calleeName*.

B. Sequence diagram verification

To verify the sequence diagrams that describe the use cases of the system, IntensiVE was further customized. The process for performing this verification consists of four steps:

- Exporting the sequence diagrams into a textual format that can be read by IntensiVE;
- Importing the sequence diagrams into IntensiVE and mapping them to the source code;
- Documenting the sequence diagrams in terms of intensional views and constraints, and verifying the diagrams;
- Customized reporting regarding inconsistencies between diagrams and source code.

In what follows, we provide a more detailed explanation of each of these four steps.

1) Exporting the sequence diagrams: As a tool to describe the sequence diagrams, our industrial partner used *Rational Rose*. A first step in our process consists of automatically exporting the documented sequence diagrams into a format that can be read by our own tools. Using the internal VisualBasic scripting engine that is offered by *Rational Rose*, we exported all sequence diagrams in a textual format. For each of the sequence diagrams, the name of the diagram was exported, the program that serves as the entry point of the scenario, the list of programs that contribute to the implementation of the use case, along with the order in which these programs are expected to be invoked.

2) Importing the diagrams into IntensiVE: IntensiVE was extended with an importer such that the sequence diagrams can be imported into our tool. This importer transforms the textual format of the sequence diagrams into an internal representation, that will be used during the actual verification process. The importer is also responsible for mapping the names of the COBOL programs that occur in the sequence diagrams onto concrete COBOL programs in the source code. By convention, the names of COBOL programs always consist of eight characters. In the most trivial of cases, the program names that occur in the sequence diagrams exactly match the names of actual COBOL programs in the source code. A

detailed study of the documented diagrams revealed however that slight variations on the naming scheme of the programs occurred in practice:

- In the sequence diagrams, punctuation marks (hyphenation, underscore) and spaces were sometimes present in the program names;
- The system consists of both an online as well as a batch-processing mode. For both modes, separate COBOL programs were used, where the batch programs were automatically generated from the online versions. Both kinds of programs are identified by means of a different naming convention. In the sequence diagrams, both variants of these program names are used;
- Program names were occasionally prefixed or postfixed.

Based on this analysis of the naming schemes used in the sequence diagram, our importer was implemented such that the variations in the naming scheme were used when mapping sequence diagrams onto concrete COBOL entities.

3) Documenting and verifying the diagrams: In order to verify the imported diagrams,they are translated into the concepts offered by the IntensiVE tool suite, namely intensional views and constraints. The set of all sequence diagrams that could be completely mapped onto source code entities (*e.g.,* for which all program names in the diagram were found to be corresponding to a concrete COBOL program), is represented by the intensional view *Sequence Diagrams*. This intensional view — defined using the SOUL query language — contains, for each sequence diagram, a tuple (*?diagram*) consisting of the name of the diagram (*?diagram.name*), the diagram's entry point (*?diagram.entryPoint*) and the sequence of programs that should be called to implement the use case (*?diagram.sequence*).

Over this intensional view, an intensional constraint is imposed that performs the verification of the sequence diagram. This constraint is specified using the set of predicates that are offered by IntensiVE to reason over COBOL programs, and is described as:

$\forall$ *?diagram* $\in$ *Sequence Diagrams* :
?mainline `isSectionInProgram:` *?diagram.entryPoint*,
?mainline `isSectionWithName:` $\{*MAINLINE*\}$,
?mainline `sectionPerforms:` *?section*,
?section
`containsOrderedCallsTo:` *?diagram.sequence*

This query consists of four logic conditions. The first two conditions query the entry point *?diagram.entryPoint* of the sequence diagram for a section *?mainline*, that is the main entry point of the program. To identify this section, we rely on the naming convention that this section contains the string `MAINLINE` in the section name. The third logic condition retrieves the sections *?section* that get invoked from within the *?mainline* section. Finally, the fourth condition checks whether the section *?section* correctly calls the sequence *?diagram.sequence*. If such a section that implements the sequence does not exist, IntensiVE will mark the diagram *?diagram* as a violation of the design documentation.

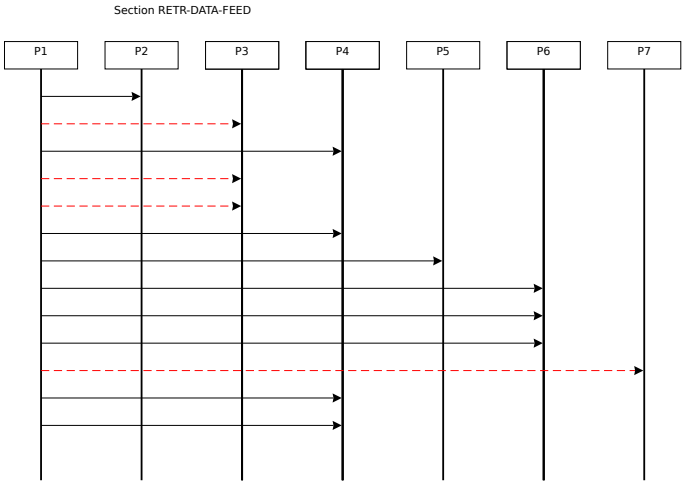

Fig. 2. Generated sequence diagram of a violation.

The definition of the intensional constraint makes extensive use of the dedicated logic library for reasoning over COBOL programs. Key to the process of verifying the sequence diagrams is the predicate `containsOrderedCallsTo:` that verifies whether a particular section contains a sequence of calls. Internally, this predicate makes use of the call-graph analysis that is part of our framework to reason over COBOL. The predicate verifies that, within the transitive control flow of a particular section, a path exists that contains the sequence of calls. Note that, since we cannot extract information regarding branching and loops from the sequence diagrams, the `containsOrderedCallsTo:` predicate flattens the branching information that was obtained from the call-graph analysis and compares the documented sequence of programs with this flattened call-graph.

4) Customized feedback regarding violations: By default, IntensiVE provides textual and visual feedback regarding identified violations. Within the context of this industrial collaboration, our tool suite was further customized to provide dedicated feedback regarding violations between documented sequence diagrams and the source code. In particular, a custom reporting tool was implemented that offers the user a generated sequence diagram for each of the violations identified by our tool, with the violating edges indicated in red. Figure 2 demonstrates such a generated diagram, with the names of the concrete programs obfuscated. In the figure, we see the sequence diagram of a use case named `retrieveDataForFeeding`. Verification of this use case with respect to the source code revealed that the implementation does not contain a section that implements this sequence diagram correctly, and thus flags it as a violation.

In case of a violation, our customization of IntensiVE will identify the section in the entry point program that best matches the sequence diagram. To this end, it uses the information in the call graph to identify the section in which the control flow exhibits the most overlap with the sequence diagram, and generate a sequence diagram for this section (such

Version	LOC	#diagrams	parsing (sec)	analysis (sec)	verification (sec)	total (sec)
Version 1	548 560	895	105	150	192	447
Version 2	665 220	917	134	228	204	566
Version 3	1 053 381	1366	413	479	288	1180

TABLE I
OVERVIEW OF THE THREE VERSIONS USED TO EVALUATE OUR APPROACH

Version	#diagrams	mappable	unmappable	% mappable	consistent	inconsistent	% inconsistencies
Version 1	895	408	487	45.59%	326	82	25.15%
Version 2	917	476	441	51.90%	386	90	23.33%
Version 3	1366	763	603	55.85%	637	126	19.78%

TABLE II
RESULTS OF THE VERIFICATION ON THE THREE VERSIONS

as is the case in figure 2) with all expected — but missing — calls indicated by a red, dashed line. In our example, we see that our tool reports that section `RETR-DATA-FEED` is the best match for the sequence diagram, but that in this diagram four different calls to two distinct programs were missing or were not called in the correct order. In case multiple sections match the sequence diagram equally well, our feedback tool will generate a sequence diagram for each of these matching sections.

V. EXPERIMENT

As a validation of our approach, we have applied it *a posteriori* to three different versions of the industrial COBOL system. These three versions align with three major iterations of the system in which novel behaviour was added. Next to receiving the source code of these three versions, we also received the Rational Rose documentation that was present at the time that the version was delivered.

A. Results of the experiment

Table I provides more information regarding the size of the system. The first to versions of the system consisted of approximately half a million lines of COBOL code. In the last version of the system, this amount of couple was almost doubled to over one million lines. As for the number of documented sequence diagrams, these vary from around 900 in the first two versions to almost 1400 in the last version.

Within the Table, we can also find the amount of time[3] that was needed to parse the COBOL code using our island-based parser, perform the static analyses (call-graph and data-flow), and verify the sequence diagrams with respect to the source code. As the table indicates, our customized support for reasoning over COBOL is quite fast: the entire process took from around 10 minutes for versions 1 and 2 up to 19 minutes for the last version. This seems to indicate that our approach scales well with respect to the verification of a real-life system.

The results of the verification can be found in Table II. First, we see that over the three versions, about half of the delivered sequence diagrams could be automatically mapped onto actual source-code entities in the implementation. Surprisingly, as the system grew larger (both in terms of lines of code as in the number of sequence diagrams), the number of sequence diagrams that could be automatically mapped increased with ten percent (from 45% to 55%). As for the sequence diagrams that could be mapped onto the source code, the vast majority (75%) was found by our tool to be correctly implemented. Similar to the evolution of the number of mappable diagrams, the amount of inconsistencies dropped over time from 25% to 20%.

[3]Measured on an Apple MacBook Pro 2.8Ghz Core Duo 2, with 4GB of RAM.

B. Analysis

1) Mapping the diagrams onto the source code: Despite the fact that our approach takes variations into account when mapping sequence diagrams onto concrete COBOL programs, and that this set of variations was obtained by studying the available sequence diagrams, our approach is only able to map about half of the sequence diagrams in the system. Analysis of the unmappable diagrams resulted in the following observations:

1) In a small number of cases, the sequence diagrams did not refer to concrete source-code entities (names of COBOL programs), but referred to the names of the components that were invoked (e.g. referring to a component named "CostManager" instead of the name of the program that implements the cost manager). These cases represent an opportunity to further refine the technical design by including the concrete program names instead of the names of the components;
2) A number of use cases could not be mapped since their corresponding implementation did not exist in the version of the software that we analyzed. The design documentation was not intended to be kept strictly synchronized with the implementation, resulting in that use cases were present which did not belong to the behavior that was supposed to be implemented in that particular version of the system. As development of the system progressed, the amount of implemented use cases should also increase, something which is supported by our findings: in version 3 we were able to map 55%

of the diagrams to source code, as opposed to 45% in version 1;

3) The majority of the sequence diagrams that could not be mapped onto the source code referred to concrete source code entities that were not present in the actual source code of the system, but that were parts of other (external) components of which we did not have the source code.

While our mapping to the source code might appear to behave poorly on first sight, our above analysis showed that this was caused by the fact that a large part of the diagrams in the documentation were not applicable to the source code we were investigating.

2) Analysis of inconsistencies: Our verification of the mappable sequence diagrams revealed that the vast majority of the use cases could be found in the concrete implementation. Given the relative small amount of inconsistent diagrams, we manually analyzed the reported inconsistencies:

- Approximately a quarter of the identified inconsistencies (version 1) down to 15% (version 3) were caused by the fact that the documentation contained diagrams that were not implemented in that particular version of the system. While all programs in the sequence diagram were found in the implementation, the concrete implementation of these diagrams was not present yet and thus, could not be found;
- Half of the inconsistencies occurred due to the order of the calls in the source code not corresponding to the specified order in the documentation. While all required calls were present in the implementation, they occurred in an order other than the one specified in the design. These inconsistencies indicate possible improvements where either the source code needed to be adapted in order to implement the correct order, or where the documentation needed to be updated in order to match to the actual situation in the source code;
- Other inconsistencies were caused by sequence diagrams of which one or more calls were not present in the source code. While our tool was able to identify in these cases the concrete section that implemented the use case, particular calls that were present in the documentation were omitted in the actual implementation.

C. Discussion

Our analysis of the industrial COBOL system using IntensiVE was able to provide the Belgian bank with an initial quality assessment of the outsourced source code. This initial assessment confirmed the bank's impression that the delivered source code in general respected the design information that was provided to the outsourcing partner, and that this conformance with this design was upheld during subsequent iterations of the system. We did find a number of violations of the documented design rules that pointed out possible improvements to source code and design documentation. Particular improvements to the source code include correct adherence to the imposed implementation guidelines and naming conventions, and corrections to the source code to more closely obey the specification of the use cases as detailed in the sequence diagrams. With respect to the design documentation, our tool identified a number of locations where the design documentation could be updated to reflect the current situation in the source code.

VI. Lessons learned

A. From an academic perspective

From an academic perspective, this case study quickly made it clear that, in order to successfully apply an academic tool to an industrial problem, a certain amount of pragmatism was required.

As the IntensiVE tool suite was tailored towards reasoning over object-oriented programs, written in languages such as Smalltalk, Java and C++, one of our primary concerns in performing this case study was extending our tool suite to support COBOL. While IntensiVE was designed from the start to be independent of the actual programming language used, this support for COBOL still required a considerable investment in effort and time. Given the specific nature of the kinds of design rules that the bank requested to verify in the system, we opted not to provide a full-fledged extension to IntensiVE to reason over COBOL systems in general, but rather provide a lightweight solution that was tailored towards the case study and that could be extended on a by-need basis. This resulted in a number of design choices such as the fact that we opted for an island-based parser to extract only the information necessary for our analysis from the source code of the system, instead of investing a large effort in providing a detailed parser that retrieves a complete representation of the source code.

Similarly, the IntensiVE tool suite provides a tight integration with the surrounding development environment. One of the goals of this integration was to support a methodology in which the verification of the design information occurs frequently (cfr. unit testing), and where the verifiable design documentation created with IntensiVE becomes an active part of the coding process. For this case study, such an integrated approach was not required nor preferred by our industrial partner. In order to monitor the progression of the quality of the outsourced system, a tool that checks conformance of the design with respect to the source code overnight suffices. As such, we opted for the development of a non-intrusive tool that generates a simple HTML report on the violations of the documented design rules.

B. From an industrial perspective

This case study illustrated the use of dedicated tool support to verify whether outsourced source code respected the design rules that are imposed by a company that makes use of an outsourcing partner. While for this case study the tool was able to confirm the impressions of the industrial partner that, in general, the outsourced system upheld the quality standards put forward by the company, the case study also identified the role that (academic) tools can play in providing a means to assess this quality.

Although in this case the quality analysis happened *a posteriori*, tools that allow for verifying the design rules imposed by a company on its outsourcing partners could play an important part in maintaining quality of the delivered source code. One particular methodology to ensure that the internal quality of an outsourced system is upheld is for a company that relies on the services of an outsourcing partner to formalize the design rules underlying the outsourced system, and to share these formalized rules with the outsourcing partner. This provides advantages for both parties involved. The outsourcing company can incorporate the verification of the internal quality of the software into their development process and can catch violations of the intended design early on. Furthermore, they can provide their client with a number of guarantees that the design of the software has been obeyed. The company using the outsourcing partner can use the documented design rules to verify that the delivered source code obeys the required design.

VII. Summary

This paper reported on a case study where the academic tool IntensiVE was used to document the design rules underlying the implementation of an outsourced COBOL system, and verify the validity of these design rules with respect to the delivered source code. Key to this system was the unique design methodology, where a component-based design was put forward that is reflected in the source code by relying on coding guidelines and naming conventions. Furthermore, the various use cases for this system were documented using sequence diagrams that describe how the different COBOL programs in the system interact. The paper discussed how IntensiVE was customized to support the verification of this design documentation, and how the use of this academic tool was able to provide an initial quality assessment of how well the modular structure of the system is reflected in the source code.

Acknowledgment

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). This research is supported by the IAP Programme of the Belgian State.

References

[1] J. Brichau, A. Kellens, S. Castro, and T. D'Hondt, "Enforcing structural regularities in software using intensive," *Science of Computer Programming: Experimental Software and Toolkits (EST 3)*, vol. 75, no. 4, pp. 232–246, April 2010.

[2] A. Kellens, "Maintaining causality between design regularities and source code." Ph.D. dissertation, Vrije Universiteit Brussel, June 2007.

[3] K. Mens and A. Kellens, "Towards a framework for testing structural source-code regularities," in *International Conference On Software Maintenance (ICSM)*, 2005, pp. 679 – 682.

[4] A. Kellens, K. D. Schutter, T. D'Hondt, L. Jorissen, and B. V. Passel, "Cognac: A framework for documenting and verifying the design of cobol systems," in *Proceedsings of the 13th European Conference on Software Maintenance and Reengineering (CSMR09)*, 2009, pp. 199–208.

[5] R. Wuyts, "A logic meta-programming approach to support the co-evolution of object-oriented design and implementation," Ph.D. dissertation, Vrije Universiteit Brussel, January 2001.

[6] P. Deransart, A. Ed-Dbali, and L. Cervoni, *Prolog: The Standard Reference Manual*. Springer-Verlag, 1996.

[7] K. Mens, I. Michiels, and R. Wuyts, "Supporting software development through declaratively codified programming patterns," in *SEKE 2001 Proceedings*. Knowledge Systems Institute, 2001, pp. 236–243, international conference on Software Engineering and Knowledge Engineering, Buenos Aires, Argentina, June 13-15, 2001.

[8] J. Brichau, C. De Roover, and K. Mens, "Open unification for program query languages," in *Proceedings of the XXVI International Conference of the Chilean Computer Science Society (SCCC 2007)*, 2007.

[9] C. De Roover, "A logic meta programming foundation for example-driven pattern detection in object-oriented programs," Ph.D. dissertation, Vrije Universiteit Brussel, August 2009.

[10] C. De Roover, I. Michiels, K. Gybels, K. Gybels, and T. D'Hondt, "An approach to high-level behavioral program documentation allowing lightweight verification," in *Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC06)*. IEEE Computer Society, 2006, pp. 202–211. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/ICPC.2006.10

[11] L. Moonen, "Generating robust parsers using island grammars," in *WCRE*, 2001, p. 13. [Online]. Available: http://computer.org/proceedings/wcre/1303/13030013abs.htm