A Generic Framework for the Analysis of Heterogeneous Legacy Software Systems

Amir M. Saeidi, Jurriaan Hage, Ravi Khadka, Slinger Jansen
Department of Information and Computing Sciences, Utrecht University, The Netherlands
{a.m.saeidi, j.hage, r.khadka, slinger.jansen}@uu.nl

***Abstract*—The reverse engineering of legacy systems is a process that involves analysis and understanding of the given systems. Some people believe in-depth knowledge of the system is a prerequisite for its analysis, whereas others, ourselves included, argue that only specific knowledge is required on a per-project basis. To give support for the latter approach, we propose a generic framework that employs the techniques of non-determinism and abstraction to enable us to build tooling for analyzing large systems. As part of the framework, we introduce an extensible imperative procedural language called KERNEL which can be used for constructing an abstract representation of the control flow and data flow of the system. To illustrate its use, we show how such framework can be instantiated to build a use-def graph for a large industrial legacy COBOL and JCL system. We have implemented our framework in a model-driven fashion to facilitate development of relevant tools. The resulting GELATO tool set can be used within the Eclipse environment.**

I. INTRODUCTION

Many companies operate systems which are developed over a period of many decades. These legacy systems are subject to continuous adaptation and evolution to deal with changing internal and external factors. Many of these systems do not meet the requirements of a maintainable system, mainly due to lack of documentation and programming structure. Reverse engineering can be employed to create a high level abstraction of the system and to identify its logical components [1].

There are many challenges that one needs to deal with when reverse engineering a large legacy system. First of all, finding a program understanding tool which can deal with the system of interest is almost impossible. On the other hand, implementing a high-quality tool from scratch that can handle the system is a tedious and time-consuming task. Furthermore, the old programming languages used to develop the legacy systems tend to suffer from a lack of “singularity”[2] and “elegance”[3], as viewed from the perspective of modern programming languages. We have investigated the use of automatic analysis techniques to provide tool support and help with understanding programs written in these languages.

Program analysis is an automatic analysis technique that can be used as part of reverse engineering [4]. Any deep program analysis starts with a syntactic analyzer parsing syntactic units into what is known as an abstract syntax tree. The tree produced must be annotated with the necessary semantic knowledge by means of a semantic analysis. Although syntactic analysis depends on the grammar of the language for which analysis needs to be performed, we argue that semantic analysis should be performed independent of the language to be processed (see Section II). This raises two questions that need to be addressed: 1) Is it possible to capture the semantics upfront for all dialects and implementations of the same programming language? 2) How much semantic information is ‘necessary’ to establish a sound foundation for conducting a particular program analysis?

For a language like COBOL which comes in various dialects, each of which may have different compiler products, establishing such semantic knowledge is impractical. In short, no single semantics exist! On the other hand, the semantic knowledge required strongly depends on the analysis one wants to perform. For example, a type-based program analysis needs to decorate the data definitions with the appropriate types, whereas a control-based analysis needs to know about control dependencies. Moreover, when dealing with large systems, abstraction is not a choice but a necessity. The analysis techniques need to be precise and scale at the same time.

Lämmel and Verhoef [2] propose a technique in which syntactic tools are constructed and later augmented with semantic knowledge on a per-project basis (*demand-driven* semantics). We build on this approach by introducing a generic framework that employs 1) nondeterminism to compute a sound abstraction of the control-flow of the program, and 2) abstraction by computing a particular program analysis with respect to enough amount of semantic information required. To realize the above features, the framework consists of an extensible intermediate language that helps achieve separation between abstraction of the problem and data flow analysis. This separation provides the context for an incremental approach to analyzing large software systems.

The paper makes the following contributions:

1) It presents a generic framework for performing program analysis on legacy systems that can be instantiated in a system-specific fashion.
2) It employs techniques from MDE to facilitate analysis of legacy systems and construct the required reverse engineering tools.

This paper is structured as follows. In Section II we outline the challenges we have faced in dealing with our industrial legacy system, and describe the generic framework to overcome the stated problems. We proceed by giving an empirical evaluation of our framework in Section III. Finally, in Section IV we conclude and outline future work.

II. A Generic Framework

We were involved in a legacy to SOA migration project in a large banking institution in the Netherlands, comprising of five distinct legacy systems. Like many business-critical systems, their systems are implemented in COBOL which runs on platforms such as IBM z/OS and HP Tandem nonstop. We have proposed a method [5] for migrating legacy systems to SOA which involves identifying candidate services followed by concept slicing to extract relevant pieces of code. To evaluate our methodology, we have been given access to one of their legacy systems, which from now on we will refer to as *InterestCalculation*. As it is the case with most legacy systems, the documentation of the *InterestCalculation* system is outdated and many of the people who were involved in its development are not around anymore. We want to apply techniques from the field of program analysis to help with both identification of services and slicing.

There are three important issues that need to be addressed when performing program analysis on legacy systems. First of all, many legacy systems are heterogeneous and constitute multi-language applications. For instance, the systems implemented for IBM mainframe usually employ JCL job units to describe different task routines that need to be performed within the legacy environment. Furthermore, COBOL has several extensions to provide support for embedded languages such as SQL and CICS. These are used to perform queries on tables and process customer transactions, respectively. This also holds for our *InterestCalculation* system, which comprises of COBOL and copybooks as well as JCL jobs, the former of which contains embedded SQL statements.

Second, programming languages used for legacy systems do not follow an explicitly defined language standard. In languages like COBOL and C, the semantics of many operations are left open and the implementation must choose how to implement these operations. Furthermore, instances of a given programming language may be home-brewed. It is estimated that there are about 300 COBOL dialects, each of which has its own compiler products with many patch levels [2]. Consequently, the only possible way to deal with inconsistencies is to rely on the compiler used to compile the system that is subject to analysis.

Finally, legacy systems contain code bases that run well into millions of lines of code, hence scalability of any program analysis technique is essential. The *InterestCalculation* system consists of almost half a MLoC of COBOL source files and copybooks. Developing analysis techniques that are simultaneously precise and scalable is not a simple task.

To overcome the aforementioned problems, we introduce a generic framework for analyzing large legacy systems which has the following three features:

1) *Language/Dialect-Independence*: We strongly believe standardization through conversion to a well-understood syntactic structure with semantic variation points is the key for analyzing different dialects and versions of the COBOL language, and naturally paves the way for heterogeneous systems comprising of COBOL and JCL.
2) *Abstraction and Nondeterminism*: Semantic analysis needs to be performed in a context-specific manner. We borrow concepts from programming language theory including non-determinism and abstraction to create an environment through which semantic knowledge can be added to the system of interest. Non-determinism guarantees the soundness of the analysis by exploring all the possible variations at the cost of performance, whereas, abstraction ensures that only a minimal amount of information is stored to perform a sound analysis.
3) *Incrementality*: Incrementality is key in building analysis tools that scale to large systems. Separation of problem specification (abstraction) and data flow analysis is the way forward for incremental analysis. In this approach, the framework can be re-instantiated with the new information obtained from the result of an analysis to perform more fine-grained analyses.

To realize the above properties, the framework consists of an extensible intermediate language called KERNEL. KERNEL employs non-determinism to capture semantics variation points at the control flow level. Furthermore, it provides extension points to extend the language to incorporate abstractions required to compute a particular data flow analysis.

Figure 1 depicts the step-by-step approach to instantiating the framework for performing a particular data-flow analysis. The first step involves syntactic abstraction (parsing) of the source program into an AST. In the next step, an abstract (static) semantics is created based on the concrete or abstract programming languages that the program has to conform to, irrespective of whether those are different dialects/implementations of the same programming language or different languages that the program is written in.

Depending on the data flow problem we are interested in, the deployed abstraction techniques ensure that enough information is stored to perform a sound analysis with respect to that problem. For instance, reaching definition analysis used to build the use-def graph is expressed as an abstract interpretation of the program which for each expression in the program infers whether a variable is definitely used or defined after the possible execution of the statement. To help with the formulation of this abstraction, extension points are provided in the kernel language to instantiate the abstract domain for a set of analysis problems.

Based on the extracted abstract semantics, a mapping is created from the syntactic structure of the source program into an instance of the KERNEL language. The use of non-determinism makes it possible to encode inconsistencies amongst different implementations, as well as points where the particular semantics cannot be derived, e.g. when no knowledge of the compiler is available.

In the next step, we specify the data flow analysis problem as a monotone framework instance [4] and solve the instance using an iterative work-list algorithm. The monotone framework consists of a set of monotonic transfer functions which

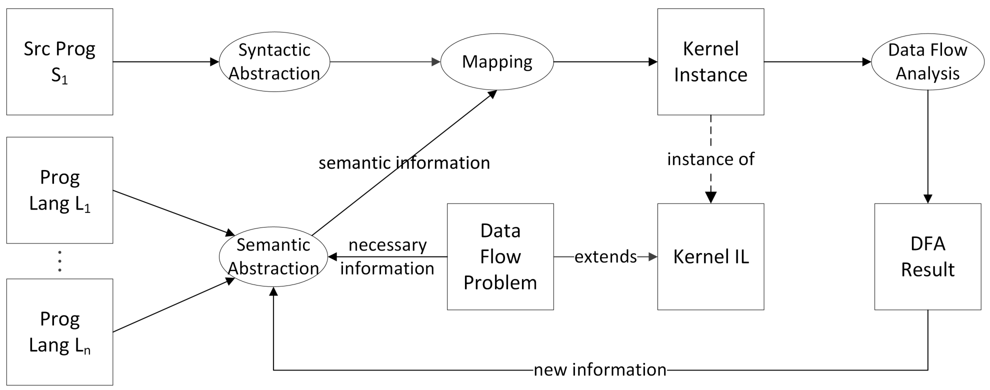

Fig. 1. The Generic Framework for Analyzing Legacy Systems

express the effect of statements on desired properties of the program with respect to a flow analysis problem. Many flow analysis problems such as reaching definition analysis (RDA) meet the monotonicity requirement and can be expressed in terms of the monotone framework. Based on the data flow analysis problem, here RDA, we give a set of transfer functions and data flow equations to instantiate the monotone framework.

The results of data flow analysis can be reused to incrementally analyze a legacy system. The result of one analysis serves as a foundation to conduct more fine-grained analyses. To demonstrate this, consider the inter-procedural dependencies derived from RDA analysis. Once we have constructed the information chain, we can interactively scope down our analysis to a smaller set of modules to perform much more detailed analyses, that because of their resource demands cannot be applied to the system as a whole. For more information about KERNEL language as well as details of the analysis, please refer to Saeidi et al. [6].

We have implemented the framework as part of the GELATO (Generic Language Tools) toolset [7]. GELATO is an integrated set of language-independent (generic) tools for legacy software system modernization, including parsers, analyzers, transformers, visualizers and pretty printers for different programming languages including COBOL and JCL.

III. EVALUATION

A. Example Case Study

Here we give an example COBOL program that is representative of our *InterestCalculation* system, depicted in Listing 1. The mapping from the set of referenceable elements in COBOL including datanames, recordnames and filenames to their corresponding set of variables in KERNEL is stored that can be used to interpret the KERNEL program. Furthermore, a mapping exists from the set of elements including procedures and statements to their corresponding labels in KERNEL that can be used to trace back its origin. Listing 2 depicts an example JCL batch job which is used to submit the INTERESTCALCULATION program to the operating system. The programs called through the EXEC statements are used as the entry point to the COBOL program. Listing 3 depicts the resulting KERNEL program from translation of COBOL program and JCL unit. Data flow analysis is then performed to build the use-def graph for a particular job unit.

```
IDENTIFICATION DIVISION.
PROGRAM-ID INTERESTCALCULATION
...
DATA DIVISION.
  FILE SECTION.
    FD IN-FILE.
      01 IN-REC.
        02 IN-NAME PIC A(20).
        02 IN-ACCOUNT  PIC 9(6)V99.
        02 IN-INTEREST  PIC 99V99.
    FD OUT-FILE.
      01    OUT-REC PIC X(80).
...
  WORKING-STORAGE SECTION.
    77 EOF PIC X VALUE "N".
    01 INTEREST1 PIC 99V99.
    01 INTEREST2 PIC 99V99.
...
PROCEDURE DIVISION.
  MAIN.
    OPEN INPUT IN-FILE OUTPUT OUT-FILE.
    READ IN-FILE END MOVE "Y" TO EOF.
    PERFORM INTEREST-CALC THRU PRINT.
    PERFORM END-PROGRAM.

  INTEREST-CALC.
    IF IN-ACCOUNT IS NOT < 150000
      SUBTRACT 50000 FROM IN-ACCOUNT
      MULTIPLY IN-INTEREST BY IN-ACCOUNT
        GIVING INTEREST1, INTEREST2.
...
   PRINT.
        MOVE "INCOME INTEREST SLIP " TO OUT-REC.
      WRITE OUT-REC.
...
   END-PROGRAM.
  STOP RUN.
```

Listing 1. A representative COBOL program

B. Empirical Findings of InterestCalculation System

In this section, we give our findings and observations with respect to the *InterestCalculation* system.

```
//EXJCL JOB 'CALC',CLASS=6,MSGCLASS=X,NOTIFY=&SYSUID
//*
//STEP001 EXEC PGM=INTERESTCALCULATION
```

Listing 2. An example JCL batch job for submitting COBOL program to OS

```
0:Procedure main(){
  35: call INTERESTCALCULATION();
}
1:Procedure INTERESTCALCULATION(){
  2:Procedure PROC1(){
    6:try {7:[uses(var1);uses(var2)];}
        with 8: exception {9: abort;}
    10:try {11:[uses(var1)];}
       with 12: exception {13: [defines(var3)]; }
    14: { 15: call PROC2(); 16: call PROC3();}
    17: { 18: call PROC4();}
  }
  3:Procedure PROC2(){
    19:if(20:[uses(var1)])then{
      21:try {22:[uses(var1);defines(var1)];}
        with 23: exception {24: abort;}
      25:try {26:[uses(var1);uses(var1);defines(var4
);defines(var5)];}
        with 27: exception {28: abort;}
      };
  }
  4:Procedure PROC3(){
    29:[defines(var2)];
    30:try {31:[uses(var2)];}
        with 32: exception {33: abort;}
  }
  5:Procedure PROC4(){
    34:{abort;}
  }
}
```

Listing 3. The representation of COBOL program in KERNEL

As is the case in many legacy systems, during our copybook inlining operation, we found out that there are 45 copybooks missing. Consequently, some of the identifiers used in the program could not be resolved to any data item. To overcome this hurdle, we create a set of proxy referenceable elements to resolve the unresolved identifiers. Moreover, to our surprise, we found out that just over a quarter of the 21085 copybooks handed to us were actually used. The entire set of copybooks comprised of almost 600 KLoC. Table I gives some metrics for the *InterestCalculation* system.

We use the classification of dependencies for COBOL as defined in [8]. They classify the dependencies in terms of functional dependencies and data dependencies. A functional dependency is created from a calling program to the callee through a CALL statement, whereas data dependency is created from a program to a copybook through a COPY statement. We extract the structural dependencies during the inlining operation.

TABLE I
METRICS FOR INTERESTCALCULATION SYSTEM

	#Files	KLoC	Highest (In/Out)degree	Lowest (In/Out)degree
COBOL Source Files	321	413.17	151	2
Used Copybooks	599	88.04	1055	1

In order to extract the functional dependencies, we needed to build the use-def graph for the *InterestCalculation* system. We have followed the approach as given in the previous section to instantiate the framework to construct the use-def graph. We have opted to make an exhaustive analysis by including a call to all the modules in the intial program entry. All the program calls in the system are dynamic, however upon completion of this analysis, we observe that all the values reaching call statements are uninitialized. That is, the value reached upon calling a program comes from the environment. A practice used in this system, as is common in many well-engineered systems, is to use a set of immutable variables and initialize them with the name of the programs to be called in the declaration section of the COBOL program. Following up on this finding, we extract the string literals of the initial values of the variables from the declaration section to create a functional dependency graph.

The experiment is performed on a 2.80 GHz Intel Core i7 quad-core machine with 16 GB RAM. The parsing of COBOL and JCL code takes about 20 minutes. The transformation and writing to a new file takes the least amount of time, that is 812 seconds. Loading the generated KERNEL code followed by instantiating the monotone framework and performing the data flow analysis takes the longest with 1676 seconds. Thus, the whole process from parsing COBOL and JCL units to constructing the use-def graph takes just over an hour. Further improvement on the performance is required to ensure the completion of the analysis in a reasonable amount of time.

IV. CONCLUSION AND FUTURE WORK

We have proposed a generic framework for analyzing legacy software systems. Based on our observations regarding the problems one may encounter when dealing with large legacy systems, our framework employs nondeterminism and abstraction to achieve language-independency and incrementality. Language independency is achieved through the specification of the source program in terms of an intermediate language which uses nondeterminism to capture semantic variations points at the control flow level. Moreover, the intermediate language provides extension points to give support to abstraction of data flow problem. This gives rise to incrementality which can be used to compute more precise as well as fine-granular analyses.

As part of future research direction, we want to go beyond COBOL, by both extending our tools to analyze programs in heterogeneous environment, as well as handle embedded languages. We want to further maturize the GELATO toolset by both conducting more experiments on real case studies and conduct more testing to validate it. Furthermore, we want to perform more analyses to assist with service identification using the framework.

References

[1] L. Moonen, "A generic architecture for data flow analysis to support reverse engineering," *Theory and Practice of Algebraic Specifications; ASF+ SDF*, vol. 97, 1997.

[2] R. Lammel and C. Verhoef, "Cracking the 500-language problem," *Software, IEEE*, vol. 18, no. 6, pp. 78–88, 2001.

[3] P. Baumann, J. Faessler, M. Kiser, Z. Oeztuerk, and L. Richter, "Semantics-based reverse engineering," 1994.

[4] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of program analysis*. Springer-Verlag New York Incorporated, 1999.

[5] R. Khadka, G. Reijnders, A. Saeidi, S. Jansen, and J. Hage, "A method engineering based legacy to SOA migration method," in *27th ICSM'11*. IEEE, 2011, pp. 163–172.

[6] A. Saeidi, J. Hage, R. Khadka, and S. Jansen, "A generic framework for model-driven analysis of heterogeneous legacy software systems," 2017. [Online]. Available: https://dspace.library.uu.nl/handle/1874/359542

[7] ——, "Gelato: GEneric LAnguage TOols for model-driven analysis of legacy software systems," in *Reverse Engineering (WCRE), 2013 20th Working Conference on*, Oct 2013, pp. 481–482.

[8] J. Van Geet and S. Demeyer, "Lightweight visualisations of Cobol code for supporting migration to SOA," *Electronic Communications of the EASST*, vol. 8, 2008.