A Generic Framework for Model-Driven Analysis of Heterogeneous Legacy Software Systems

Amir M. Saeidi

Jurriaan Hage

Ravi Khadka

Slinger Jansen

Technical Report UU-CS-2017-003
March 2017

Department of Information and Computing Sciences
Utrecht University, Utrecht, The Netherlands
www.cs.uu.nl

ISSN: 0924-3275

Department of Information and Computing Sciences
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands

A Generic Framework for Model-Driven Analysis of Heterogeneous Legacy Software Systems

Amir M. Saeidi, Jurriaan Hage, Ravi Khadka, and Slinger Jansen

Department of Information and Computing Sciences, Utrecht University
{a.m.saeidi, j.hage, r.khadka, slinger.jansen}@uu.nl

Abstract. Reverse engineering of legacy systems is a process that involves analysis and understanding of the system. 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.

1 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 Sections 2 and 3). 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 a word, 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.

Model-driven engineering (MDE) provides a practical solution to performing software analysis at different abstraction levels. In this setting, models present an abstract representation of the system, which using standard transformation languages can be transformed into other models, hence facilitating analysis of the system at different abstraction levels. Furthermore, using a standardized metamodel, it is possible to create language-independent techniques to analyze and visualize 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 2, we motivate our work by outlining the challenges we have faced in dealing with our industrial legacy system. In Section 3 we describe the generic framework, while in Section 4, we introduce the *Kernel* intermediate language used in the framework and give the control flow representation of a valid program in *Kernel*. As an example, in Section 5, we demonstrate how the framework can be instantiated to construct the use-def graph. In Section 6 we give an informal description of the current status of our tooling. We proceed by giving an empirical evaluation of the framework using the Gelato tooling in Section 7. Related works are presented in the Section 8. Finally, in Section 9 we conclude and outline future work.

2 Motivation

We are currently 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. Furthermore, many old programming languages used to develop legacy systems contain unstructured elements and provide only a minimum level of abstraction. One of the problems with

Cobol syntax is its verbosity, where there are many constructs with exactly the same semantics. For instance, there are five statements for computing various arithmetic operations, namely, ADD, SUBTRACT, MULTIPLY, DIVIDE and COMPUTE, each of which may take up to three formats.

Second, programming languages used for legacy systems do not follow an explicitly defined language standard. C programming language is famous for comprising of many operations which exhibit arbitrary behavior [6]. In C, using an uninitialized allocated variable on the heap, division by zero, or indexing an array outside of its defined bounds are situations where undefined behavior may occur. On the other hand, in these languages the semantics of many operations are left open and the implementation must choose how to implement these operations. An example for this situation is the pre- and post-incrementation in C, where the exact moment when incrementation occurs is an implementation-specific issue. 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.

If we do in fact have access to the compiler, there are two issues that need to be taken into consideration. Program executables may exhibit different behavior depending on the compiler flags they are compiled with. Moreover, the observable behavior of the program is unpredictable and depends on the context in which it is executed such as the platform and the peripheral storage devices used. On the other hand, it is not always possible to get access to the compiler product. The compiler may not exist anymore or companies may deny access to their compiler product, as is the case in our situation. We are fortunate to know that the system complies with IBM Enterprise Cobol, so that we can follow the implementation reference [7]. However, when this is not possible, one is faced with the difficult job of considering all semantic variations that may occur between compilers, compiler versions and compiler flags. To get a feeling of the kind of variations one may encounter when trying to gather different semantic variation points, let's consider the following two examples.

The following Cobol fragment corresponds to a compound ADD statement:

```
1 ADD A, B, C TO C, D(C), E.
```

Since we are using the Enterprise Cobol standard, we can rely on the implementation reference of language to figure out the order of evaluation, that is left-to-right. However, this is an implementation-specific property and may affect portability of the program, if the target compiler does not follow the same evaluation order.

The exception handling mechanism is another example which displays inconsistent behavior across different compilers. The exception handling in Cobol is based on a *syntax-driven specialization* mechanism, where for one exception and two possible handlers, the more specific handler is taken. For some statements, a statement-level handler can be defined. Besides statement-level handlers, Cobol provides more general handlers known as USE declaratives to specify procedures that are to be executed in the case an exception is raised. Although exception handling in Cobol is prioritized, depending on different dialects and implementations, additional handlers may still be taken. To clarify this point, let's have a look at the following Cobol fragment:

```
1 ADD a To b ON SIZE ERROR imperative-statements.
```

In Enterprise Cobol, if the size error condition is raised upon execution of the ADD statement, control is passed to the exception-specific handler. After completion of the handler statements, execution continues with the statement directly after the ADD statement. This does not hold in the case of ILE Cobol. In ILE Cobol, the statement-level specific-handler still has the highest priority, however the control is then passed to an ILE condition handler, provided it is registered with the run unit. Then the control is transferred to the USE declarative in Cobol. After execution of all the additional handlers, the execution continues with statement directly after the ADD statement.

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 1 MLOC of Cobol source files and copybooks. Developing analysis techniques that are simultaneously precise and scalable is not a simple task. To help resolve the above problems, we have developed a generic framework that supports language-independent specification of data-flow analyses.

3 A Generic Framework

In this section, we introduce a generic framework for analyzing large legacy systems. To overcome the problems stated in the previous section, our framework 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.

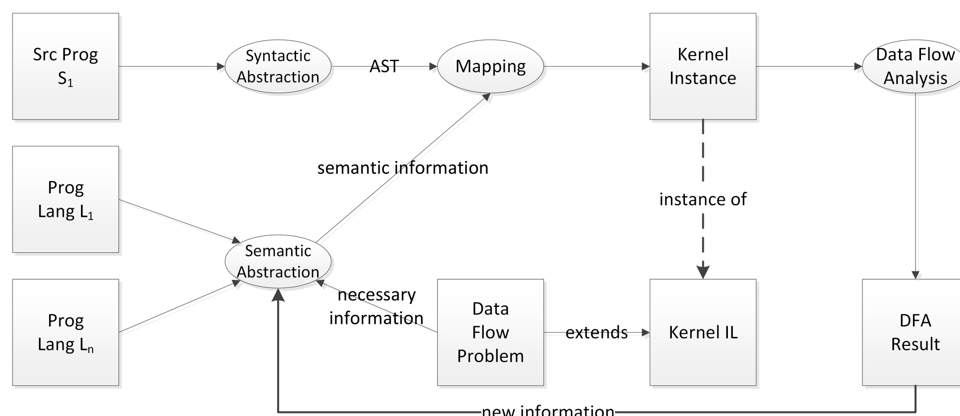

Fig. 1. The Generic Framework for Analyzing Legacy Systems

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 abstraction techniques deployed 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 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.

4 Kernel Language

In this section, we introduce an extensible imperative procedural language called *Kernel*. We proceed by giving the control flow representation of a valid program in *Kernel*.

4.1 Syntax of Kernel Language

We begin our formal semantic definition by giving the abstract syntax of *Kernel*. In *Kernel*, we distinguish between expressions and statements. Expressions and statements may both cause computational effects or diverge. The set of labels are denoted using Lab_*. Annotating statements with labels helps support the tracking of elements in *Kernel* back to their origin in the original program, by creating traceability relations between the constructs in the program and the *Kernel* code.

The *Kernel* language as defined in Figure 2 allows for defining parameterized procedures at both the program level as well as nested subprocedures. It is equipped with default assignments, conditionals, iteration and unstructured jumps. The calls to procedures are done through explicit call by name (label). The return statement transfers the control to the calling program. The try-exception block allows for defining statements that may throw an exception, with its set of handlers explicitly defined. Normal or abnormal termination of a run unit is defined using the **abort** statement. The **skip** statement is equivalent to a no-op. To reduce the complexity of control flow analysis, we apply a reduction rule to reduce consecutive skip statements into a single skip statement, while preserving their corresponding labels.

To give support for nondeterministic behavior, *Kernel* provides parallel blocks $*>$ as well as non-deterministic blocks $|>$. The parallel block comes in three variations: the $*>_{l2r}$ and $*>_{r2l}$ blocks are deterministic where the order of evaluation of its statements are from left-to-right and

$$
\begin{array}{lrl}
\text{program} & \mathbf{P} ::= & l_i : proc_i(e_i) \\
\text{procedures} & \mathbf{proc}(e) ::= & \overrightarrow{l_i : proc_i(e_i)} \\
 & \mid & s \\
\text{statements} & \mathbf{s} ::= & l : c \\
 & \mid & s_1; s_2 \\
 & \mid & |\!> \ \overrightarrow{s_i} \\
 & \mid & *\!>_{l2r} \ \overrightarrow{s_i} \\
 & \mid & *\!>_{r2l} \ \overrightarrow{s_i} \\
 & \mid & *\!>_{ilv} \ \overrightarrow{s_i} \\
\text{commands} & \mathbf{c} ::= & e \\
 & \mid & v := e \\
 & \mid & \mathbf{ret}\ l : e \\
 & \mid & \mathbf{goto}\ l \\
 & \mid & \mathbf{skip} \\
 & \mid & \mathbf{if}\ (l : e)\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2 \\
 & \mid & \mathbf{while}\ (l : e)\ \mathbf{do}\ s \\
 & \mid & \mathbf{call}\ l(e) \\
 & \mid & \mathbf{try}\ s_1\ \mathbf{with\ exception}\ s_2 \\
 & \mid & \mathbf{abort} \\
\text{expressions} & \mathbf{e} ::= & \mathbb{D}
\end{array}
$$

Fig. 2. The context-free grammar for the Kernel

right-to-left, respectively. On the other hand, $*\!>_{ilv}$ block uses interleaving semantics by exploring all the possible interleavings of its statements. Please, note that the parallel block does not give any extra computational power, but is simply syntactic sugar. The parallel block can be unfolded into a deterministic or non-deterministic block depending on the value of the order of evaluation.

The expression language is abstracted to some abstract domain $\mathbb{D}$, which acts as an extension point which can be instantiated based on the needs of a particular data flow analysis problem.

4.2 Interprocedural Nondeterministic Control Flow Graph

Before we give the control flow representation of a valid program in *Kernel*, we introduce the notion of interprocedural nondeterministic control flow graph. We combine the nondeterministic control flow representation, as given in [8] with the standard definition of interprocedurally valid program in [9] to yield an interprocedural nondeterministic control flow graph (INCFG, for short). INCFG is defined as a supergraph consisting of a collection of intraprocedural nondeterministic flow graphs (NCFG, for short).

An $NCFG$ is defined as follows:

$$NCFG = < V, V_P, V_N V_C, V_R, A_C, A_N, A_{CR}, s, t > \quad (1)$$

where

- V: a set of nodes in the graph
- $V_N \subset V$: a set of nondeterministic selection points
- $V_C \subset V$: a set of call nodes
- $V_R \subset V$: a set of return-site nodes
- $A_C \subseteq V \times V$: a set of intraprocedural sequential edges
- $A_N \subseteq V_N \times V$: a set of intraprocedural nondeterministic selection edges
- $A_{CR} \subseteq V_C \times V_R$: a set of intraprocedural call-to-return-site edges
- $s \in V$: the unique entry point
- $t \in V$: the unique exit point

An $INCFG$ is defined as follows:

$$INCFG = < NCFG^*, V_C^*, V_R^*, S, T, A_{CS}, A_{TR}, s, t > \quad (2)$$

where

- $NCFG^*$: a collection of NCFGs
- S: the set of all entry points in $NCFG^*$
- T: the set of all exit points in $NCFG^*$
- V_C^*: the set of all call nodes V_C in $NCFG^*$
- V_R^*: the set of all return nodes V_R in $NCFG^*$
- $A_{CS} \subseteq V_C^* \times S$: the set of interprocedural call-to-return-site edges from the the call-site to the entry point of the called procedure
- $A_{TR} \subseteq T \times V_R^*$: the set of interprocedural exit-to-return-site edges from the exit node to the return-site of the calling procedure
- $s \in S$: the unique entry point corresponding to one of the entry points in its containing $NCFG$
- $t \in T$: the unique exit point corresponding to one of the exit points in its containing $NCFG$

Please, note that all the set of S, T, V_C^* and V_R^* can be derived from the $NCFG^*$, but are included for convenience.

4.3 Control Flow Representation of Kernel

In the sequel, we show how to instantiate the $INCFG$ for a program in *Kernel*. We distinguish between control flow of the procedures at the program-level and those of nested subprocedures.

A program P in *Kernel* is represented using an $INCFG$, denoting a flattened representation of the control flow of the program. This flattened graph comprises a collection of intraprocedural $NCFG$s, separated into a set of subgraphs which are sequentially disjoint. Each of the subgraphs maps to a procedure at the program level and there is no sequential edge connecting any two subgraphs at this level. On the other hand, the subgraph for each procedure is a monolithic graph comprising a set of $NCFG$s corresponding to each subprocedure in the procedure, with a set of additional sequential edges. In this subgraph, for each $NCFG$, there is a successor (sequential) edge from its exit vertex to the entry vertex of its syntactically successive $NCFG$. Since the language supports arbitrary levels of nesting, its corresponding graph becomes a disjoint multi-level supergraphs of INCFGs.

We define a pair of functions, $labels(P)$ and $flow(P)$ to represent the control flow graph of program P, where the set of labels correspond to the set of nodes in the INCFG, and its flow maps to the set of edges in the graph. There is a pair of labels for each call statement, l^c and l_r. We define a pair of two auxiliary functions namely, $init_{s,p}()$ and $final_{s,p}()$ which returns the initial label and final labels of a statement and a procedure, respectively. The edges are subscripted by n, s, c, r to denote nondeterministic, sequential, call and return edges, respectively. Figure 3 gives the flow of a well-formed program in *Kernel*.

5 Data Flow Analysis

An important data flow analysis that is useful for program understanding is use-def chaining. The links derived from this analysis determine for each use of a variable in a program, all the assignments that may reach that use. To give support for this analysis, we show how the *Kernel* language can be extended with the necessary abstractions to perform the analysis.

5.1 Instantiating the Abstract Domain for Building Use-Def Chains

We introduce a base domain by instantiating the abstract domain for analyzing programs to build use-def chains. We are only interested in the flow of data from the definition site of a variable to

$$
\begin{aligned}
\mathbf{flow}(\overrightarrow{l_i : proc_i(e_i)}) &= \textstyle\bigcup_i \mathbf{flow}(proc_i(e_i)) \\
&\quad \textstyle\bigcup_i \{(final_p(proc_i(e_i)), init_p(proc_{i+1}(e_{i+1})))_s\} \\
\mathbf{flow}(l : e) &= \emptyset \\
\mathbf{flow}(l : \mathbf{skip}) &= \emptyset \\
\mathbf{flow}(s_1 ; s_2) &= \mathbf{flow}(s_1) \cup \mathbf{flow}(s_2) \\
&\quad \cup\{(l, init_s(s_1))_s \mid l \in final_s(s_1)\} \\
\mathbf{flow}(l : |> \ \overrightarrow{s_i}) &= \textstyle\bigcup_i \mathbf{flow}(s_i) \\
&\quad \textstyle\bigcup_i \{(l, init_s(s_i))_n\} \\
\mathbf{flow}(\mathbf{if}\ (l : e)\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2) &= \mathbf{flow}(s_1) \cup \mathbf{flow}(s_2) \\
&\quad \cup\{(l, init_s(s_1))_s, (l, init_s(s_2))_s\} \\
\mathbf{flow}(\mathbf{while}\ (l : e)\ \mathbf{do}\ s) &= \mathbf{flow}(s) \cup \{(l, init_s(s))_s\} \\
&\quad \cup\{(l', l)_s \mid l' \in final_s(s)\} \\
\mathbf{flow}(l : \mathbf{ret}\ l' : e) &= \{(l, final_p(p))_s \\
&\quad \mid \text{p is the procedure that contains the return statement}\} \\
\mathbf{flow}(l : \mathbf{goto}\ l') &= \{(l, l')_s\} \\
\mathbf{flow}(l : \mathbf{call}\ l'(e)) &= \{(l^c, l_r)_s\} \\
&\cup \{(l^c, init_p(p))_c \\
&\quad \mid \text{p is the procedure corresponding to label } l'\} \\
&\cup \{(final_p(p), l^r)_r \\
&\quad \mid \text{p is the procedure corresponding to label } l'\} \\
\mathbf{flow}(l : \mathbf{try}\ s_1\ \mathbf{with\ exception}\ s_2) &= \mathbf{flow}(s_1) \cup \mathbf{flow}(s_2) \\
&\cup \{(l', init_s(s_2))_s \\
&\quad \mid l' \in \text{final labels of all the elementary statments in } s_1\} \\
\mathbf{flow}(l : \mathbf{abort}) &= \{(l, final_p(p))_s \\
&\quad \mid \text{p is the outermost containing procedure of abort}\}
\end{aligned}
$$

Fig. 3. The control flow of a program in *Kernel*

its uses. The domain is a list of elements of the abstract domain where the order of occurrence of elements influences the flow of data. Figure 4 defines the elements of the domain suited for our purposes.

As a side note, we have decided to distinguish between visible and hidden side-effects. It is important to make such a distinction as to allow one to track the origin of the occurrence of a variable to a statement in the original program. Usually, hidden side-effects are not obvious from the syntax of the program.

There are two cases in Cobol where a hidden side-effect may occur: the file position indicator and the file-status key. If the file position indicator is set, I/O operations result in advancing the file offset. On the other hand, the file status key is used to provide an error-handling mechanism for I/O operations. If a file status key is associated with a file, any error resulting from execution of I/O on the file at runtime, results in altering the value of the key. We use the *affects*() subexpression to denote these hidden side-effects.

We define a pair of use and definition (hidden and visible) subexpressions. The first set of subexpressions are used to use (*uses*()) or overwrite the current value of the field (*defines*() and *affects*()). The second set of subexpressions use the value of the field before reaching the containing statement (*uses*′()) or overwrite the field after completion of the statement (*defines*′() and *affects*′()). To understand why we need the extra set, let's have a look at the Cobol fragment below.

```
1 ADD A, B, C TO C, D(C), E.
```

The execution of this compound ADD statement is equivalent to executing the following set of statements:

```
ADD A, B, C GIVING TEMP.
ADD TEMP TO C.
ADD TEMP TO D(C).
ADD TEMP TO E.
```

A compiler may generate a temporary result field, TEMP, to store the temporary result of addition of the operands on the LHS of the operator. In other words, the value of these operands in the subsequent statements depend on the value as defined before reaching the ADD statement.

$$
\begin{array}{llcll}
\text{abstract domain} & D ::= & [abs] & & \\
\text{domain elements } abs ::= & & uses(v) & \mid defines(v) \\
 & \mid & affects(v) & \mid uses'(v) \\
 & \mid & defines'(v) & \mid affects'(v)
\end{array}
$$

Fig. 4. The expression language for the Kernel

The order of definition and usage is from left to right in the list. Please note, $defines$ subexpression in the expression list uses all the variables specified as $uses$ starting from the head of the list until the position where the $defines$ subexpression occurs. To help improve the performance, we apply reduction rules to reduce consecutive abstract elements of the same variable to a single abstract element. For example, consecutive occurrences of $defines(v)$ is reduced to a single definition of v.

Assignment statements as defined in the syntax of *Kernel* are used to express single variable assignments, whereas the expression language is much more expressive allowing for specifying a sequence of arbitrary usage, define and affects subexpressions. Consequently, the expression language is powerful enough to express impure/effectful expressions. As part of normalization, we transform assignment statements into an expression statement.

5.2 Building the Use-Def Graph

Constructing the use-def graph requires establishing knowledge about reaching definitions of a variable in a statement where it is used. We follow the use-def chaining as described in [4] to construct the use-def graph. We apply *Reaching Definition Analysis* to construct links between statements that define a variable and those that use them.

The data flow equations for the reaching definition analysis used here are the same as the ones given in [4]. The only difference is that post-definitions will not redefine the variable until they reach the end of their immediate confined non-deterministic block. If no non-deterministic block exists, they are treated as if they were on-site definitions.

We use the algorithm for interprocedural data flow analysis as given in [9] to compute the set of reaching definitions for all program statements. Once the result is obtained, creating use-def chains simply becomes associating a link between the usage of a variable in a statement with all the possible set of reaching definitions of that variable. One simple difference is that the pre-usage is linked to its reached definitions prior to entering the innermost nondeterministic block.

6 Implementation

In this section, we introduce our Gelato (Generic Language Tools) toolset [10]. Gelato [1] is an integrated set of language-independent (generic) tools for legacy software system modernization,

[1] The latest builds of the plugins can be obtained at: `https://github.com/amirms/gelato`

including parsers, analyzers, transformers, visualizers and pretty printers for different programming languages. There are many workbenches that support generating generic tools for language engineering such as the ASF-SDF environment [11], and model-based workbenches such as EMFText[2] and Xtext[3]. We have opted for EMFText due to its support for a wide range of languages, in particular Java through the Jamopp project, to support software system analysis in a model-driven fashion.

We have adopted a model-driven approach to facilitate development. The Eclipse Modelling framework (EMF) is an Eclipse-based modeling framework which uses the standard Ecore metamodel for model-based language engineering. Although MDE deals with artifacts at the model level, EMFText and Xtext try to bridge this gap between modelling languages and programming languages by providing a text syntax for instances of Ecore metamodels. Defining a language with its text syntax as a Ecore metamodel comes with many benefits including 1) language reuse and extension using embedded DSLs, 2) use of standard transformation languages such as QVT [12] for model transformation, and 3) a wide availability of language-independent tools for analysis and visualization.

In Section 7, we demonstrate how the framework can be instantiated. In the first stage, a parser parses Cobol programs into an abstract tree which we call a Cobol model. We exploit embedded DSL techniques to extend the *Kernel* language to provide the abstractions necessary for the specification of the data flow problem. The transformation from the Cobol model to *Kernel* is performed in two stages: a pre-translation stage computes static information including aliases, hidden side-effects and exception handling techniques; this information is then fed into the translator to obtain its implementation in *Kernel* which is then used to perform control and data flow analyses. To determine the main entry points to the Cobol program, a JCL job unit is parsed and transformed as explicit calls to the corresponding procedures of the Cobol programs in *Kernel*. The pre-built monotone framework which is implemented in Java is instantiated for performing reaching definition analysis (RDA), the result of which establishes the information chain for building the use-def graph.

6.1 Parsing Cobol

The Cobol language, as part of the Gelato toolset defines a metamodel for Cobol that covers the whole language. The standard Ecore metamodels are used to specify Cobol language, which allows for future extension. The textual syntax is defined in EMFText which generates a parser as well as a pretty printer for the language. There has been a lot of work on how to deal with different dialects of Cobol at the syntactic level [13]. The Cobol language used in our legacy system is based on Cobol 85 as implemented in IBM Enterprise Cobol; no support is given to the new constructs introduced in Cobol II and IBM extensions (VS Cobol II, OS/VS Cobol, etc.) such as pointers. Moreover, we have decided to exclude deprecated ALTER and ENTRY statements, as they are not used in our system.

The grammar we are using is based on the ANSI Cobol 85 standard. We have exploited lake grammars [14] to fine-tune the syntax to allow us to correctly parse the Cobol legacy system. Contrary to island grammars, we start with a complete grammar for a given language and extend the grammar with a number of liberal production rules to allow for parsing slightly different dialects, for instance, inclusion of IBM extensions in our case. However, at times we have diverged from the specification by simplifying the rules, therefore allowing more programs to be accepted. In our approach, we ensure that the syntactic units that have been skipped do not interfere with the semantics of the program with respect to a particular analysis. Our primary goal here is to develop an environment for pure reverse engineering purposes, where we make the assumption that all the code we are dealing with are compilable (syntactically correct).

Our metamodel consists of 461 classes, 120 of which are abstract. The metamodel is divided into 28 packages. There is support for all the features of the language from abbreviated conditions to special names.

[2] `http://www.emftext.org`
[3] `http://www.eclipse.org/Xtext`

We follow the approach proposed in [13] by performing parsing in three-stages, while at each step preserving enough information to be able to recreate its printed version. The parsing process is as follows: 1) stripping Cobol code of the irrelevant columns to obtain the so called 666-code, followed by its lexical disambiguation; 2) a preprocessor parses the disambiguated 666-code to extract the COPY statements followed by inlining the referenced copybooks after application of the replacements, if necessary; 3) the inlined 666-code is then fed into the Cobol parser to obtain its abstract representation (Cobol model).

6.2 Testing

We have partially tested the Cobol parser and pretty printer, but further evaluation is required to assess their correctness and completeness. Since our parser can also accept invalid programs, the main objective of the testing phase is to ensure it can handle industrial-sized applications. We have defined a test suite to check whether 1) valid programs are accepted; 2) name and cross-reference resolution is performed correctly; 3) the pretty printing does generate a program which correctly maps to the original one; and 4) it does indeed generate a correct parse tree. Although we can claim with a certain level of confidence that the parser and printer meet the (1-3) criteria, we are yet to perform a thorough testing on whether the parser indeed creates a correct parse tree.

6.3 Preprocessing Cobol Model

Before we can perform the translation, we need to infer the static information from the Cobol program that is required to create a sound abstraction of its control and data flow. This information includes:

- **Propagating Alias Information**: The alias information has to be extracted and propagated. An alias occurs when two or more identifiers point to the same memory location. In Cobol 85, there are 3 cases where an alias arises: 1) structured fields (group data items), 2) redefining/renaming fields and 3) condition-names.
- **Extracting Exception Handling Mechanisms**: As stated in Section 2, Cobol provides various exception handling mechanisms. Based on the abstract semantics of the language we are working with, we need to infer the set of handlers a statement may throw, which may include handlers with the USE declarative. We also need to know if a statement may cause an abnormal termination of the run unit for a particular type of exception.
- **Hidden Side-effects**: The hidden side-effects for different statements need to be identified. As stated in Section 5, file status key in Cobol is an example where execution of an I/O statement may alter its value.

6.4 Entry Points in Batch Mode

In a mainframe environment, programs can be executed in batch and online mode. In batch mode, programs are submitted to the operating system through a batch job. The Job Control Language (JCL) is used to submit a batch job to the mainframe through the Job Entry System (JES). Each JCL job consists of one or more job steps; each can execute a program through the EXEC statement or call another procedure which in turn comprises of some job steps. For identifying all programs belonging to an application, all the programs submitted to the Operating System through the JCL procedures need to be taken into account [15]. It is only in the context of these programs, that together with indirect calls, a system dependency graph can be derived.

The current implementation of JCL language in the Gelato toolset is a partial implementation of IBM MVS JCL. We have restricted ourselves to the EXEC statements to find the programs that are invoked through a job unit. For a fully fledged parser, one must consider JCL (sub-)procedures and INCLUDE statements amongst others.

6.5 Translation to Kernel

The translation process involves two stages, where during the first stage, new mappings from fresh variables to the fields are created. All the mappings are stored using the EMF persistent storage to allow for later usage. In the second stage, based on this mapping, translation takes place. We use the imperative QVT Operational Mapping (OM) as part of the OMG standard QVT [12] transformation language to encode the mappings from Cobol to *Kernel*. The transformer proceeds top-down, traversing the Cobol model and recursively converting submodels to corresponding sequences of statements in the target model. The code snippet in Listing 1.1 depicts an excerpt of transformation rules used, here to map a MOVE statement in a Cobol program to its representation in the *Kernel* language.

```
mapping in COBOL::statements::Move::transformMove()
2  : KERNEL::statements::ExpressionStatement {
    init{
4     result := object KERNEL::statements::ExpressionStatement{label :=
    getFreshLabel();}
    }
6   result.expression := object KERNEL::expressions::Expression{label=getFreshLabel
    ()};
    result.expression.children := self.sender.transformOperandForUsage();
8   result.expression.children += self.receivers.transformOperandForDefinition();
}
```

Listing 1.1. The mapping rule for transforming a MOVE statement to Kernel

6.6 Limitation

Besides the CALL statement, Cobol provides another technique for calling intra-module procedures with the possibility of code sharing: the PERFORM statement. As is well-known, not all well-formed PERFORM statements have a valid semantic interpretation. The IBM compiler used in [16] uses a single continuation to store the dynamic fall-through successor of the exit of the subprocedures in a program, while storing its static successor in a special storage area. As a consequence of this implementation, recursive calls are not possible. Furthermore, abnormal behavior may occur if the control does not reach the exit of the PERFORM range. Since Cobol provides unstructured jumps, i.e. GOTO statement, a situation may arise where continuations are still active (mines), even though the control has jumped out of the perform range. Field and Ramalingam [16] try to overcome this problem by performing an analysis to conservatively identify PERFORM ranges that follow the conventional LIFO, before transforming it into a well-structured procedural representation. We have decided not to account for possible misuses of PERFORM statements, as it was not necessary when dealing with our legacy system. The legacy system we are working on is very well-structured and does not contain any GOTO statements. Thus such behavior cannot occur.

Second, we have decided to exclude the embedded execute statements for the time being, thus EXEC statements are mapped to **skip**. In Cobol, EXEC statement is used to embed SQL or CICS commands.

Finally, dynamic calls are excluded, as it requires accounting for string manipulation operations in Cobol such as literals constructed through STRING and UNSTRING operations. To ensure soundness of our analysis, we could potentially transform dynamic calls to static calls to all the procedures that match the call signature, however in this work, we map dynamic calls to simple uses of variable holding the value of the called program. This transformation makes our analysis unsound, however, as we shall see in the next section, the coding practice used in the *InterestCalculation* system guarantees the soundness of our analysis for building the use-def graph.

7 Evaluation

7.1 Example Case Study

Here we give an example Cobol program that is representative of our *InterestCalculation* system, depicted in Listing 1.2. Its corresponding generated representation in *Kernel* is given in Listing 1.4. The mapping from the set of referenceable elements in Cobol including datanames, record-names 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 1.3 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.

```
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 1.2. A representative Cobol program

Listing 1.4 depicts the resulting *Kernel* program from translation of Cobol program and JCL unit. Data flow analysis is performed to build the use-def graph for a particular job unit.

7.2 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 1.3. 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 1.4. 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 1 gives some metrics for the *InterestCalculation* system.

We use the classification of dependencies for Cobol as defined in [17]. 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.

The diagram in Figure 5 depicts an excerpt of the dependencies between Cobol source files and the copybooks. The green nodes correspond to the modules and the gray nodes represent the referenced copybooks.

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 previous sections 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

	Number of Files	KLoC	Highest (Out-/In)degree	Lowest (Out-/In)degree
Cobol Source Files	321	413.17	151	2
Used Copybooks	599	88.04	1055	1

Table 1. Metrics for InterestCalculation System

Fig. 5. An excerpt of data dependency graph in InterestCalculation

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. Figure 6 gives an excerpt of the inter-procedural dependencies through the CALL statement. The direction of link is from the calling module to the called module.

Through performing the exhaustive control flow analysis, we have identified the unreachable code [4]. There is less than 1% unreachable code, which is found in 5 modules. This can be partly due to an evolving software with continuous removal of products from the application logic that are no longer needed to be supported. Eliminating dead code not only helps with reducing the code space, but also with further transformations on the control flow of the program.

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 scalability of the analysis in a reasonable amount of time.

[4] Unreachable code is the set of statements in the source code that are not reachable from the main entry point.

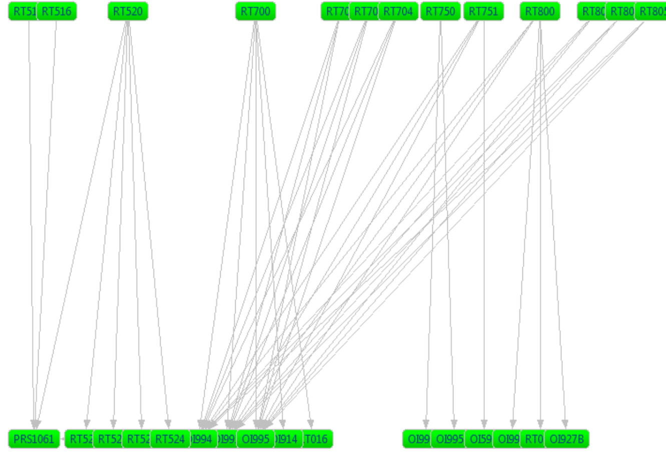

Fig. 6. An excerpt of functional dependency graph in InterestCalculation

8 Related Work

There have been numerous works over the years to give support to language-independent program analyses. One aspect of analyzing heterogeneous systems is concerned with fact extraction from the source code. In general, these approaches can be classified into two categories, one based on lexical analysis, and the other based on syntactic analysis. Lexical-based techniques rely on lexical methods to extract information from system artifacts. These techniques provide a flexible and robust solution that can easily be extended to deal with multi-lingual code bases at the cost of losing soundness and precision. The most well-known tools are grep, lex scannner gerenator [18], and AWK [19] using regular expressions to match patterns. On ther other hand, grammar-based techniques, employ parsers to build parse tree for performing sound and detailed data flow analyses. Although writing a parser for a language is tedious and expensive, alternative approaches based on island grammars [14], [20] are proposed for building robust parsers for large and multi-lingual systems.

Performing data flow analysis requires not only to deal with different syntactic units but also different semantic variation points in each implementation of the language. Moonen [1] proposes a generic data flow language called *Dhal* for performing different data flow analyses, however it fails to address issues related with semantic variation points at both control and data flow levels. Furthermore, Strout et al. [21] introduced the OpenAnalysis toolkit that uses an abstraction layer between the representations and the analysis engines to give support to language-independent program analysis. Reps et al. [22] present CodeSurfer/x86 and companion tools that can be used to recover the intermediate representations from low-level software artifacts. More recently, Strein et al. [23] proposed a language independent metamodel and an architecture to give support for new analyses, refactorings, and new front-ends of programming languages. Basten and Klint present DeFacto [24], a fact extraction technique which annotates the context-free grammar of a language with facts which can later be extracted from parse tree fragments to perform a software analysis. In a similar work, Mark Hills [25] presents a domain-specific language called DCFlow in Rascal [26] for declarative specification of the control flow graph for a variety of language constructs.

Analyzing multi-language systems requires dealing with references to data and control definitions that cross the language boundaries. Kullback et al. [15] try to overcome this problem by presenting a graph-based conceptual modeling approach using EER/GRAL to represent relevant aspects of each language into a common conceptual model. Mayer and Schroeder [27] propose a generic approach to understanding, analyzing and refactoring cross-language code by explicitly specifying the semantic cross-language links.

There is a growing trend in using MDE principles and techniques to perform reverse engineering tasks. MoDisco [28] proposes a generic and extensible framework for reverse engineering legacy systems in a model-driven fashion. It extracts models from Java source code. Heidenreich et al. [29] uses Jamopp to modernize Java applications. We have built on these approaches and present a toolset for extracting models from Cobol-based systems.

9 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 would like to further implement features in JCL by accounting for procedures and import of libraries, as well as including CICS/SQL embedded statements in the Cobol program. 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. L. O. Andersen, "Program analysis and specialization for the c programming language," Ph.D. dissertation, University of Cophenhagen, 1994.
7. *Enterprise Cobol for z/OS, V4.2, Language Reference.* IBM Corp., 2009.
8. J. Cheng, "Nondeterministic parallel control-flow/definition-use nets and their applications," *Parallel Computing: Trends and Applications," Elsevier Science Publishers BV (North-Holland)*, pp. 589–592, 1994.
9. T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* ACM, 1995, pp. 49–61.
10. A. Saeidi, J. Hage, R. Khadka, and S. Jansen, "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.
11. M. Brand, A. Deursen, J. Heering, H. Jong, M. Jonge, T. Kuipers, P. Klint, L. Moonen, P. Olivier, J. Scheerder, J. Vinju, E. Visser, and J. Visser, "The ASF+SDF meta-environment: A component-based language development environment," in *Compiler Construction*, ser. Lecture Notes in Computer Science, R. Wilhelm, Ed. Springer Berlin Heidelberg, 2001, vol. 2027, pp. 365–370.
12. OMG, "QVT. Meta Object Facility (MOF) 2.0 Query/View/Transformation specification," *Final Adopted Specification (November 2005)*, 2008. [Online]. Available: http://www.omg.org/spec/QVT/1.0/PDF

13. M. G. J. Van Den Brand, M. P. A. Sellink, and C. Verhoef, “Obtaining a COBOL grammar from legacy code for reengineering purposes,” in *2nd TPAS*, ser. Algebraic’97. Springer, 1997, pp. 6–16.
14. L. Moonen, “Generating robust parsers using island grammars,” in *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, 2001, pp. 13–22.
15. B. Kullbach, A. Winter, P. Dahm, and J. Ebert, “Program comprehension in multi-language systems,” in *Reverse Engineering, 1998. Proceedings. Fifth Working Conference on*, Oct 1998, pp. 135–143.
16. J. Field and G. Ramalingam, “Identifying procedural structure in Cobol programs,” in *ACM SIGSOFT Software Engineering Notes*, vol. 24, no. 5. ACM, 1999, pp. 1–10.
17. J. Van Geet and S. Demeyer, “Lightweight visualisations of Cobol code for supporting migration to SOA,” *Electronic Communications of the EASST*, vol. 8, 2008.
18. M. E. Lesk and E. Schmidt, “Lex: A lexical analyzer generator,” 1975.
19. A. V. Aho, B. W. Kernighan, and P. J. Weinberger, *The AWK Programming Language*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1987.
20. N. Synytskyy, J. R. Cordy, and T. R. Dean, “Robust multilingual parsing using island grammars,” in *Proceedings of the 2003 Conference of the Centre for Advanced Studies on Collaborative Research*, ser. CASCON ’03. IBM Press, 2003, pp. 266–278.
21. M. M. Strout, J. Mellor-Crummey, and P. Hovland, “Representation-independent program analysis,” in *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, ser. PASTE ’05. New York, NY, USA: ACM, 2005, pp. 67–74.
22. T. Reps, G. Balakrishnan, and J. Lim, “Intermediate-representation recovery from low-level code,” in *Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, ser. PEPM ’06. New York, NY, USA: ACM, 2006, pp. 100–111.
23. D. Strein, R. Lincke, J. Lundberg, and W. Lowe, “An extensible meta-model for program analysis,” *Software Engineering, IEEE Transactions on*, vol. 33, no. 9, pp. 592–607, 2007.
24. H. J. Basten and P. Klint, “Software language engineering,” D. Gašević, R. Lämmel, and E. Wyk, Eds. Berlin, Heidelberg: Springer-Verlag, 2009, ch. DeFacto: Language-Parametric Fact Extraction from Source Code, pp. 265–284.
25. M. Hills, “Streamlining control flow graph construction with dcflow,” in *Software Language Engineering*, ser. Lecture Notes in Computer Science, B. Combemale, D. Pearce, O. Barais, and J. Vinju, Eds. Springer International Publishing, 2014, vol. 8706, pp. 322–341.
26. P. Klint, T. van der Storm, and J. Vinju, “Rascal: A domain specific language for source code analysis and manipulation,” in *Source Code Analysis and Manipulation, 2009. SCAM ’09. Ninth IEEE International Working Conference on*, Sept 2009, pp. 168–177.
27. P. Mayer and A. Schroeder, “Cross-language code analysis and refactoring,” in *Source Code Analysis and Manipulation (SCAM), 2012 IEEE 12th International Working Conference on*, Sept 2012, pp. 94–103.
28. H. Bruneliere, J. Cabot, F. Jouault, and F. Madiot, “Modisco: a generic and extensible framework for model driven reverse engineering,” in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2010, pp. 173–174.
29. F. Heidenreich, J. Johannes, J. Reimann, M. Seifert, C. Wende, C. Werner, C. Wilke, and U. Assmann, “Model-driven modernisation of java programs with jamopp,” in *Joint Proceedings of the First International Workshop on Model-Driven Software Migration, MDSM*, 2011, pp. 8–11.