Semantics-based Reverse Engineering

Peter Baumann, Jürg Fässler, Markus Kiser,
Zafer Öztürk, Lutz Richter *

Institut für Informatik der Universität Zürich
Winterthurerstr. 190
8057 Zürich
Switzerland

Email: {baumann,faessler,kiser,oeztuerk,richter}@ifi.unizh.ch

* *Jürg Fässler* is supported by KWF (Schweizerische Komission zur Förderung der wissenschaftlichen Forschung), grant nr. 2326.1, *Zafer Öztürk* is supported by Swiss Life Insurance and Pension Company.

Abstract

Program understanding is one of the key issues in software reverse engineering. It is common to distinguish between different levels of program understanding. Software maintenance tasks require at least a thorough semantic level understanding of programs.

In this paper we introduce and advocate a powerful approach to the semantic analysis of programs for reverse engineering purposes. The technique, we call *semantics-based reverse engineering*, has its roots in the formal static analysis methods developed in modern programming language theory. More precisely, we propose the use of denotational semantics and abstract interpretations in order to extract the relevant information for the understanding of the inner workings of existing programs. Such techniques are applied successfully in advanced optimizing compilers. Their use in reverse engineering is also very promising, as we will show below.

As a testbed for our research in semantics-based reverse engineering, we are developing a prototype of a program understanding system, called AEMES.

1 Introduction

We all know that software maintenance is a difficult and costly task. This is documented by the fact that an estimated 70 percent of the expenses during the life cycle of a program are spent on maintenance. Furthermore, the languages used in old programs are not very elegant from the point of view of today's standard of programming language design. They contain unstructured elements and provide only a minimum of abstraction mechanisms. Therefore, it is a must to use the most powerful techniques available today in order to support the complex task of understanding old programs.

The most powerful techniques in use today to statically analyze programs can be found in modern optimizing compilers for high-level programming languages. In particular, techniques derived from denotational semantics, known as abstract interpretations, become more and more widely used [13]. In addition, the methods applied in reverse engineering should have a sound foundation, as the extraction of wrong information during maintenance tasks can lead to new errors in new versions of the programs and has to be avoided.

In this paper we introduce the new idea of applying known techniques from advanced programming language theory to the problems of building powerful reverse engineering tools. Based on theoretical reflections about the potential possibilities and on our own practical experience in using these techniques, we propose and advocate the use of denotational semantics and the related static program analysis techniques (abstract interpretations) to develop tools for program understanding. To the best of our knowledge, this is the first paper which investigates the potential role of such formal program analyses in reverse engineering and shows how they can indeed be applied in practice.

Our paper is organized as follows. In the next section we emphasize the importance of program understanding in reverse engineering. The third section discusses in detail various formal semantics of programming languages and shows how one can apply abstract interpretations in reverse engineering. The fourth section presents the overall architecture of the program understanding system AEMES. We end the paper with conclusions and an outline of our future work in this area.

2 Program understanding

Existing program packages require continuous adaption and functional enhancements in order to keep them in line with the changing internal and external demands. All the activities needed to fulfill these requirements are summarized under the term software maintenance.

It is well-known that software maintenance is difficult because old programs are insufficiently structured and very often no reliable documentation exists for them. Also, application programs are usually not written well enough to be maintained at a later date. The consequence of all this is that maintenance engineers have to look for ways to rebuild the overall picture from a piece of software which should be as close as possible to that of the original designer and implementer.

Reverse engineering is nowadays a preferred way to overcome the aforementioned obstacles. By means of different kinds of information obtained by reverse engineering methods, the software maintenance engineer can be supported in the complex process of program understanding. Various structural information helps the maintainer to obtain different views of a program. Graphical as well as textual representations of the program facilitate the process of navigating through the architecture of a program system. Implied dependencies and relations are made visible and, therefore, guide the maintenance engineer in getting a stepwisely refined understanding of the concerned program.

Program understanding has the ultimate goal of comprehending the underlying functional and data concepts. This objective can be achieved on various levels of program understanding.

Program understanding starts with analyzing the syntactical units of a given source code which are delivered by code browsing and cross referencing singular items grouped into variables, reserved words, strings, constants etc. The next step in syntactical analysis consists of a parsing process which generates an internal representation of a program called abstract syntax tree. Information about syntactical structures is necessary in supporting program understanding but it helps only partially because syntactical units enlighten at most the local knowledge. But we can use the result of a syntactical analysis of a large program to partition that program into a couple of separate segments which accordingly are much more suited for further investigation than the monolithic original one.

A much deeper insight into the real behavior of a complex program system can be achieved by means of semantical analysis. At the semantic level we are able to obtain information about the control-flow and data-flow and in addition about the corresponding dependencies. It is possible to analyze the control flow links which mirror the sequencing of the program and the calling links created for example by `PERFORM` statements in COBOL. An extensive semantic analysis should also be capable of handling unstructured pieces of code, like ill-formed scopes and ranges. Control dependency analysis checks the reachability of code parts depending on the evaluation of conditional statements. The data layout and the used aliasing definitions are made visible by data structure analysis. The knowledge about these declarations is a prerequisite for the understanding of all used data and their interrelationship concerning data dependencies.

At the highest level of abstraction the reverse engineering process has to act as a vehicle for understanding given programs in the large as well as in the small. In other words, the maintenance engineer should be able to derive the consolidated data models and their complete static context from information obtained by reverse engineering methods. The dynamic data mappings are also highly desirable but require an exhaustive semantic analysis which may sometimes exceed the expenditure one can afford in terms of time and complexity. A similar situation arises with regard to the understanding of the functional concepts contained in programs. Such functional concepts are encoded by means of so-called plans. The recognition of such plans has often been performed by means of knowledge based techniques [24, 17].

Returning to the main task of adapting and enhancing functionality in existing sources, there are several levels to achieve stepwise enriched knowledge and understanding for existing program sources. Several questions have to be answered in order to accomplish a well-defined change in functional requirements. The navigation process running through the code has to determine at which parts (where) of the given program, and what kind of necessary changes have to be made in order to alter the operational behavior of the maintained program. Usually, one starts with simple syntactical analysis, and after a first insight concerning the *where* one proceeds with (sometimes several levels of) semantic analysis detecting all necessary global and local interrelationships to support the *what kind*. The recommended procedures are by no means unique because they are very much depending on individual techniques originated in the daily experience of maintenance people. However, the outlined levels and techniques of program understanding provide a substantial support to facilitate the crucial task of software maintenance. Central to the automation of the whole program understanding process is the ability to find tools that help the maintenance engineer to gain a semantic level understanding of the programs he or she has to maintain.

3 Formal Semantics and Reverse Engineering

3.1 Which formal semantics?

In order to obtain a thorough understanding of the behavior of a program it is indispensable to have a clear understanding of the programming language constructs. This is true not only for a maintenance engineer who wants to hand simulate the execution of a program, but also for developers of reverse engineering tools and environments.

A closer look at suitable tools for reverse engineering reveals that one can at least distinguish between syntax-based and semantics-based tools.

Syntax-based tools extract information from a program purely on the basis of syntactic information, i.e. by using parse trees, token lists, etc. The techniques employed in these tools are variants of scanning and parsing technologies. Examples of syntax-based tools include pretty printers, flow chart generators, program browsers, cross referencers and call graph generators. The limitation of this category of tools is that they can only present information about the structure of a program, but not about internal dependencies. The semantic context of syntactic constructs is lost. Therefore, these tools can only be used for the lowest level of program understanding, i.e. the syntactic level.

In contrast, the more powerful semantics-based tools provide insights into control-flow and data dependences. This kind of tools encompasses control-flow analyzers, data-flow analyzers, program slicers, etc. They directly support semantic level understanding and are a prerequisite for concept level understanding.

As we saw in the previous section, a maintenance engineer needs at least a correct semantic level understanding. This is why tools for software maintenance have to provide semantic information, too. Therefore, a presupposition for reverse engineering tools is a clear description of the semantics of the programming language constructs. The question that arises is what kind of description of programming language constructs should form the foundation on which reverse engineering tools and environments can be developed.

The first distinguishing characteristic of programming language descriptions is formality. There are informal presentations of semantics, as can be found in programming language manuals. Such descriptions give a rough idea of the meaning of the constructs, but do not always explain the meaning in all possible application contexts. Just remember the famous example of *ALGOL 60* [16], where it took several years to detect numerous ambiguities. To have a detailed semantic understanding of the language constructs becomes even more significant in reverse engineering, where one often has to understand programs written by others with different programming styles. This requires a deeper knowledge of a language than just programming skills.

Another reason for using formal semantics in reverse engineering is that analysis techniques must have a sound basis in order to guarantee that the extracted information is correct. After all it is this information on which a maintenance engineer builds his or her decisions. We do not want to introduce new errors in old programs because a reverse engineering tool gives incorrect information about the program behavior. This implies that the analysis has to be closely related to the semantics of the language.

Fortunately, there exist formal notations to precisely define the meaning of programming language constructs. It is beyond the scope of this paper to review all formal semantic theories and their varieties. Therefore, we concentrate on three standard formal semantics here:

- *Formal operational semantics:*

  Before the seminal lecture notes of G. Plotkin [25] were written, operational semantics of programming languages were defined with the help of abstract machines. Although the goal was to keep the particularities of these machines at a minimum in order to make the language as independent of the abstract machine as possible, there always remained some properties of the machine that influenced the definition.

  The first formal operational semantics, called *structural operational semantics,* was introduced by G. Plotkin in [25]. A second one, named *natural semantics,* is derived from structural operational semantics and presented in [14]. Both approaches use inductively defined transition systems to formally describe the evaluation relation of programs, i.e. they describe how a program is executed. The difference between the two is that structural operational semantics formalizes single step execution and natural semantics defines evaluation relations for complete execution. Formal operational semantics are closely related to natural deduction systems in mathematical logic in that the transition systems are given by axioms and inference rules as discussed in [14].

- *Denotational semantics:*

  Denotational semantics, based on work of D. Scott and Ch. Strachey, give the meaning of statements or programs at a more abstract level. A semantics consists of a set of functions, defined inductively over the abstract syntax tree, that map language constructs to mathematical values in what are known as domains [20, 6, 26, 30]. One distinguishes between direct-style and continuation-style semantics. Direct-style semantics are perhaps easier to understand at the beginning but become more complicated as soon as the control constructs of a programming language become more involved (e.g. goto's, exceptions). In contrast, it is much easier to model the control flow of real programming languages with continuations.

- *Axiomatic semantics:*

  Axiomatic semantics [9, 30] describe the meaning of a programming language construct indirectly through partial or total correctness assertions. These assertions provide a means to reason about properties of statements and expressions, but do not really say what the meaning of a construct is.

The questions that arises is whether such formal description techniques can successfully be used to develop reverse engineering tools and if so which one is appropriate. In order to answer this question one should first establish a set of criteria that have to be met by a formal semantics when used for program understanding purposes.

In our opinion, a formal description technique has to fulfill at least the three following requirements in order to be successfully applicable in reverse engineering:

1. The formal definition technique has to be applicable to the parts of the programming language in question that are relevant for reverse engineering methods.

2. The technique should support standard approaches to program analysis such as control-flow analysis, data-flow analysis, etc.

3. The technique should be easily and efficiently implementable.

Having examined the various formal semantics with respect to these requirements, we came to the following conclusions:

- Formal operational semantics are widely used nowadays to present formal semantics of real-world programming languages. The most prominent one is the definition of *Standard ML* [19]. There exist descriptions of program analysis techniques in operational terms. Unfortunately, to find efficient implementations of the logical rules of a formal operational semantics is not so easy and is the subject of ongoing research [10].

- Denotational semantics is used to provide formal semantics of many programming languages [26]. It is a very well understood method. Denotational semantics are also called standard interpretations. The reason for that is that they provide the intended meaning of programs. An advantage of denotational semantics is that there exist *abstract interpretations*. Abstract interpretations are formal static program analysis techniques and follow a pattern similar to standard interpretations. But instead of working with real values such as integers, strings, etc., abstract interpretations use abstract values representing classes of real values (e.g. positive versus negative integers). Today, abstract interpretations are used extensively in optimizing compilers for logic and functional programming languages during the semantic analysis and the optimization phase. Examples of abstract interpretations include control-flow and data-flow analyses. In addition, denotational semantics and abstract interpretations are easily implemented with the help of modern high-level functional programming languages such as Standard ML and Haskell, or logic programming languages such as Prolog [18, 27].

- The formalism named most often in the context of reasoning about program behavior is Hoare logic. But it is somewhat surprising that apart from some specifically designed languages such as *Euclid* it is only for *Pascal* that substantial parts have been described by Hoare triples. We are not alone with this observation. The following paragraph is a quotation from [11, page 13]:

> But time has shown, at least for languages which were designed without the explicit constraint that they should have an 'axiomatic semantics', the task of providing a complete axiomatization to be impractical.

We conclude that denotational semantics and the related static program analysis techniques indeed fulfill all three of the above requirements. The other two approaches have some serious drawbacks with respect to their potential use in reverse engineering. Therefore, we concentrate on the practical application of standard and abstract interpretations for reverse engineering purposes in the rest of this paper.

3.2 How much formal semantics is needed?

Since we do not just want to propose the use of formal semantic techniques in reverse engineering based on theoretical reflections, but also want to show its applicability, we are currently developing a reverse engineering environment, called AEMES. AEMES can be used for the reverse engineering of COBOL 74 programs.

The BNF grammar that we have for COBOL consists of more than 170 complex production rules taking up about 30 pages (the grammar for C has approximately 60 grammar rules that can be written down on one single page!), so we are certainly not working with a toy language here. Of course, the huge grammar of COBOL makes it quite unrealistic to develop a complete formal semantics for COBOL. But this is not really necessary. It is one of the key observations of our approach that we can apply formal static analysis techniques without having a full formal semantics for the target language.

First of all, when building a reverse engineering environment, it is not really necessary to build a compiler for the target language. This is true because we build analysis tools which in some sense provide *abstractions* of the program behavior. When building tools, the question becomes for which statements we need the semantic interpretation in order to be able to build a particular abstraction, and thus we only have to give the semantics for these statements. We have seen that for many tools (and many abstract interpretations) the relevant statements are simply the control constructs, which only make up about 10 percent of our COBOL grammar.

To illustrate our answer let us first have a look at a standard semantics and a functional dependency analysis of variables in an abstract interpretation style for a simple *While* programming language. We assume that the reader has some basic knowledge about denotational semantics, but it is not at all necessary to understand the details of the interpretations below. We want to emphasize that the structures of standard interpretations and abstract interpretations are amazingly similar with respect to control statements. The semantics of statements (Fig. 1) is given by inductively defining a function

$$Sem_{st} : Stm \rightarrow (State_{st} \rightarrow State_{st})$$

mapping each statement to a function from states to states. The functions

$$
\begin{array}{l}
Sem_{st} : Stm \rightarrow (State_{st} \rightarrow State_{st}) \\[1em]
Sem_{st}[\![\mathtt{x := a}]\!]s = s[x \mapsto A_{st}[\![\mathtt{a}]\!]s] \\
Sem_{st}[\![\mathtt{skip}]\!] = id \\
Sem_{st}[\![\mathtt{c1; c2}]\!] = Sem_{st}[\![\mathtt{c2}]\!] \circ Sem_{st}[\![\mathtt{c1}]\!] \\
Sem_{st}[\![\mathtt{if\ b\ then\ c1\ else\ c2}]\!] = cond_{st}(B_{st}[\![\mathtt{b}]\!], Sem_{st}[\![\mathtt{c1}]\!], Sem_{st}[\![\mathtt{c2}]\!]) \\
\qquad\qquad \text{where } cond_{st}(p, g1, g2)s = \begin{cases} g1\ s & \text{if } p\ s = true \\ g2\ s & \text{if } p\ s = false \end{cases} \\
Sem_{st}[\![\mathtt{while\ b\ do\ c}]\!] = FIX\ H \\
\qquad\qquad \text{where } H\ h = cond_{st}(B_{st}[\![\mathtt{b}]\!], h \circ Sem_{st}[\![\mathtt{c}]\!], id)
\end{array}
$$

Figure 1: The semantic function Sem_{st}

$$A_{st} : Aexp \rightarrow (State_{st} \rightarrow Value_{st})$$

and

$$B_{st} : Bexp \rightarrow (State_{st} \rightarrow Boolean_{st})$$

give the standard meaning of arithmetic and boolean expressions and are left unspecified as they are not of further interest here.

We can compare this standard semantics to the abstract interpretation (Fig. 2), which

$$
\begin{array}{l}
Sem_{abs} : Stm \rightarrow (State_{abs} \rightarrow State_{abs}) \\
\\
Sem_{abs}[\![\mathtt{x := a}]\!]s = s[x \mapsto A_{abs}[\![\mathtt{a}]\!]s] \\
Sem_{abs}[\![\mathtt{skip}]\!] = id \\
Sem_{abs}[\![\mathtt{c1; c2}]\!] = Sem_{abs}[\![\mathtt{c2}]\!] \circ Sem_{abs}[\![\mathtt{c1}]\!] \\
Sem_{abs}[\![\mathtt{if\ b\ then\ c1\ else\ c2}]\!] = cond_{abs}(B_{abs}[\![\mathtt{b}]\!], Sem_{abs}[\![\mathtt{c1}]\!], Sem_{abs}[\![\mathtt{c2}]\!]) \\
\qquad\qquad \text{where } cond_{abs}(p, g1, g2) \text{ depends on the abstract interpretation considered} \\
Sem_{abs}[\![\mathtt{while\ b\ do\ c}]\!] = FIX\ H \\
\qquad\qquad \text{where } H\ h = cond_{abs}(B_{abs}[\![\mathtt{b}]\!], h \circ Sem_{abs}[\![\mathtt{c}]\!], id)
\end{array}
$$

Figure 2: The semantic function Sem_{abs}

calculates the functional dependency of variables. The example is taken from [23]. As already mentioned, it is not necessary to understand the details of this analysis here. It is only important to observe that this abstract interpretation follows the same pattern as the standard interpretation. In general, different abstract interpretations for static program analysis follow similar patterns with respect to control constructs.

We have also observed that different abstract interpretations for COBOL have similar structures. We developed a denotational semantics of the control constructs of COBOL-74 [3]. Since COBOL-74 contains `goto` statements we used a continuation-style denotational semantics. The task was not too difficult, as it is well-known how to model the control-flow of `goto` statements by continuations. The only problem we had to solve was to model the flow of control after paragraphs and sections, since the flow of control in COBOL depends dynamically on the way a paragraph or section is entered. If called from a `PERFORM` statement the flow of control is determined by this `PERFORM` statement. But the flow of control is sometimes also determined by the syntactical ordering of the paragraphs and sections. This is the case when control reaches a paragraph or section not through a `PERFORM` statement, but through sequential execution. These observations are the reason that we had to include a dynamic continuation management in our semantics. The details are fully explained in [3], where we also provide an implementation of the semantics in the high-level functional programming language Standard ML.

Let us come back to the observation that the patterns of different abstract interpretations and of the standard interpretation are very similar. The next two figures show the standard interpretation and an abstract interpretation (again for functional dependency analysis) of the `if` statement in COBOL. In order to further emphasize the similarities between the two we do not show the formal definition, but we present our implementations in Standard ML. The standard interpretation `CS` (Fig. 3) is a function of type

```
CS (AstStatementIf(b, c1, c2)) env cont =
   let
      val econt =
         Econt(fn ev =>
            case ev of
               (Constant (SvBool bc))
            => if bc then CS c1 env cont
                else CS c2 env cont
            | _ => raise SEMEX "Boolean expected")
   in
     B b env econt
   end
```

Figure 3: The standard interpretation of the `if` statement

```
STMT_AST -> ENVIRONMENT -> CONT_ST -> CONT_ST
```

and the abstract interpretation `CA` (Fig. 4) is a function of type

```
STMT_AST -> ENVIRONMENT -> CONT_ABS -> CONT_ABS.
```

Again the details are of no interest here.

```
CA (AstStatementIf(b, c1, c2)) env cont =
   let
      val econt =
         Econt(fn ev =>
            case ev of
               (Constant (Prop bc))
            => condP (bc ,
                        (fn env => fn cont => CS c1 env cont),
                        (fn env => fn cont => CS c2 env cont)
                       ) env cont
            |  _ => raise SEMEX "Boolean expected")
   in
      B b env econt
   end
```

Figure 4: An abstract interpretation of the `if` statement

An even more impressive example is the `PERFORM` statement of COBOL-74. Here the implementations for the standard and abstract interpretation are the same (Fig. 5).

We see that in order to use standard and abstract interpretations for developing reverse engineering tools we need a semantic description of the control constructs of the target language. What else do we need? Do we need a standard interpretation of the atomic statements and expressions? Our experience is that such descriptions are not required. For most static analysis techniques it is sufficient to know which variables are used during computation and which ones are updated. What remains is to define the intended abstract interpretations of an atomic statement or expression in terms of this knowledge.

This central observation is the one that makes our approach practicable. That is true because abstract interpretations are generally much simpler than a full standard

```
C (StmtPerform(PerformId(AstIdent id))) (Env env) cont
    = Cont(fn (State(sto, fl, ContStore csto, fcl)) =>
        let val dv = env (Ident id)
        in case dv of
          (ParVal(ParCont(proc, contloc, _)))
            => let val newstate =
                 (State(sto, fl,
                        ContStore(subst csto contloc cont),fcl))
                   val (Cont cont') = proc cont
               in cont' newstate end
        | _ => raise SEMEX "ParVal expected" end)
```

Figure 5: The standard and abstract interpretation of the `PERFORM` statement

semantics since they neglect many details that would have to be considered in full detail in a standard semantics.

We conclude that even for languages such as COBOL-74, where it would be quite difficult and time-consuming to produce a standard semantics, it is possible to develop program analysis tools that have their foundations in the powerful technique of abstract interpretation. What is needed is a formal semantics of the control flow constructs and a description of the different roles of variables occurring in atomic statements and expressions.

3.3 Abstract Interpretations

To formulate a new static program analysis in terms of an abstract interpretation requires some familiarity with the concepts of denotational semantics. But our experience is that after having studied several examples of abstract and standard interpretations one gets an understanding of the general patterns they follow and one can start writing new interpretations quickly. Due to the availability of languages such as Standard ML it is also quite easy to implement such interpretations and to experiment with them.

Additionally, static program analysis in the form of abstract interpretation is not new. It started as early as in 1977 with the seminal paper of P. Cousot and R. Cousot [5]. Hence, there already are numerous publications proposing and studying different kinds of program analysis methods in abstract interpretation style. Many of them can be found in conference records and journals on theoretical and implementational aspects of programming languages or in books such as [13]. There is also a book [1] on abstract interpretation. An up-to-date overview and readable introduction to abstract interpretation is [12].

With respect to the relevant analysis approaches for reverse engineering one can find examples of abstract interpretations for data-flow analysis [22], functional dependency analysis [23], conditional constant propagation [28], Def-Use-analysis [28] or program slices [8, 29]. There are also publications that investigate the relationship between graph-based and semantics-based static program analysis techniques [4, 21]. Especially, [21] is recommended as a thorough investigation of termination properties of different program analyses. Another fact worth emphasizing is that by using continuation-style abstract interpretations one gets MOP-solutions to program analysis problems for free as is shown in [22].

To give an idea of what an abstract interpretation may look like and how it can be applied, we present an abstract interpretation for constructing program slices of the `MOVE` statement and statement composition in COBOL-74. This example is inspired by [8].

$$\delta : Stm \times VarSet \rightarrow VarSet$$

$$\delta([\![\texttt{MOVE e TO x}]\!], V) = \begin{cases} V \setminus \{\texttt{x}\} \cup Var(\texttt{e}) & \text{if } \texttt{x} \in V \\ V & \text{otherwise} \end{cases}$$

where $Var(\texttt{e})$ represents the set of variables occuring in $\texttt{e}$

$$\delta([\![\texttt{c1.c2}]\!], V) = \delta([\![\texttt{c1}]\!], \delta([\![\texttt{c2}]\!], V))$$

Figure 6: The auxiliary function δ for the `MOVE` statement and statement composition

First, we need an auxiliary function δ (Fig. 6) which given a statement `c` and a set of variables V returns the set of variables needed before `c` to determine the values of all variables in V after `c`.

Applying δ to the statement sequence `MOVE b TO a . MOVE b TO c . MOVE a TO d` and the variable set $\{d\}$, we expect the result to be $\{b\}$:

$$
\begin{aligned}
&\delta([\![\texttt{MOVE b TO a . MOVE b TO c . MOVE a TO d}]\!], \{d\}) = \\
&\qquad \delta([\![\texttt{MOVE b TO a}]\!], \delta([\![\texttt{MOVE b TO c . MOVE a TO d}]\!], \{d\})) = \\
&\qquad \delta([\![\texttt{MOVE b TO a}]\!], \delta([\![\texttt{MOVE b TO c}]\!], \delta([\![\texttt{MOVE a TO d}]\!], \{d\}))) = \\
&\qquad \delta([\![\texttt{MOVE b TO a}]\!], \delta([\![\texttt{MOVE b TO c}]\!], \{a\})) = \\
&\qquad \delta([\![\texttt{MOVE b TO a}]\!], \{a\}) = \\
&\qquad \{b\}
\end{aligned}
$$

We can now turn to the main interpretation function *slice* in our example. A program slice is an executable subprogram of the original program that behaves like the original one with respect to a given set of variables. Below, we give the definition of the function *slice* (Fig. 7) for the `MOVE` statement and statement composition, which given a statement `c` and a set of variables V returns the slice with respect to `c` and V. We would expect

$$slice : Stm \times VarSet \rightarrow Stm$$

$$slice([\![\texttt{MOVE e TO x}]\!], V) = \begin{cases} \texttt{MOVE e TO x} & \text{if } \mathtt{x} \in V \\ \epsilon & \text{otherwise} \end{cases}$$

where ϵ is the empty statement

$$slice([\![\texttt{c1.c2}]\!], V) = slice([\![\texttt{c1}]\!], \delta([\![\texttt{c2}]\!], V)) \smile slice([\![\texttt{c2}]\!], V)$$

where $\smile$ is concatenation

Figure 7: The function *slice* for the `MOVE` statement and statement composition

that only the two statements `MOVE b TO a` and `MOVE a TO d` are necessary to determine the value of d after the sequence `MOVE b TO a . MOVE b TO c . MOVE a TO d`:

$$
\begin{aligned}
& slice(\llbracket \texttt{MOVE b TO a . MOVE b TO c . MOVE a TO d} \rrbracket, \{d\}) = \\
&\qquad slice(\llbracket \texttt{MOVE b TO a} \rrbracket, \delta(\llbracket \texttt{MOVE b TO c . MOVE a TO d} \rrbracket, \{d\})) \smile \\
&\qquad\qquad slice(\llbracket \texttt{MOVE b TO c . MOVE a TO d} \rrbracket, \{d\}) = \\
&\qquad slice(\llbracket \texttt{MOVE b TO a} \rrbracket, \{a\}) \smile slice(\llbracket \texttt{MOVE b TO c . MOVE a TO d} \rrbracket, \{d\}) = \\
&\qquad \texttt{MOVE b TO a} \smile slice(\llbracket \texttt{MOVE b TO c . MOVE a TO d} \rrbracket, \{d\}) = \\
&\qquad \texttt{MOVE b TO a} \smile slice(\llbracket \texttt{MOVE b TO c} \rrbracket, \delta(\llbracket \texttt{MOVE a TO d} \rrbracket, \{d\})) \smile \\
&\qquad\qquad slice(\llbracket \texttt{MOVE a TO d} \rrbracket, \{d\}) = \\
&\qquad \texttt{MOVE b TO a} \smile slice(\llbracket \texttt{MOVE b TO c} \rrbracket, \{a\}) \smile slice(\llbracket \texttt{MOVE a TO d} \rrbracket, \{d\}) = \\
&\qquad \texttt{MOVE b TO a} \smile \epsilon \smile slice(\llbracket \texttt{MOVE a TO d} \rrbracket, \{d\}) = \\
&\qquad \texttt{MOVE b TO a} \smile \epsilon \smile \texttt{MOVE a TO d} = \\
&\qquad \texttt{MOVE b TO a . MOVE a TO d}
\end{aligned}
$$

Again, functions like δ or *slice* are easily implementable in Standard ML.

However, implementing a set of abstract interpretations is best done in a framework which takes advantage of all the common features of these interpretations. We are currently developing such a framework for analyzing COBOL-74 programs. This framework provides the central functionality for the reverse engineering tools in AEMES.

3.4 Elimination of goto statements

There is one more point we want to address. Most abstract interpretations found in the literature are given for structured programming languages. Unfortunately, COBOL-74 is not a structured language as it allows `goto` statements.

One solution would be to try to find abstract interpretations for unstructured statements. But this seems not to be easy. The reason is that `goto` statements can induce loops which are difficult to recognize.

The other solution is to first restructure programs and then to analyze the resulting programs. This is what we do. Fortunately, there exists a powerful restructuring technique presented in [2]. This technique starts from a continuation-style description of the flow of control of a program. Hence, it has the advantage that it fits very well within our general semantics-based approach.

It is worth mentioning that the method was developed by researchers familiar with semantics-based analysis techniques in advanced compiler technologies [7] and that this restructuring technique is more powerful than other methods [2].

4 The architecture of AEMES

After having presented the heart of our system and its theoretical basis, we give an overview of the ideas behind AEMES and its overall architecture.

The aim of the project AEMES (An Extensible Maintenance Engineering System) is the design and implementation of a prototype of an environment for software maintenance. AEMES is aimed at maintenance programmers whose daily business is the maintenance of COBOL 74 programs. Our decision to support COBOL 74 as the target language is a pragmatic one: On the one hand, our research has shown that there is a great need for support in the maintenance of programs written in COBOL 74, on the other hand, our decision allows us to deal with real-world problems, by coping with a language that is still widely used for commercial applications.

The architecture of AEMES consists of three main parts: the maintenance engine, the user interface, and a data store. Let us now have a look at all three parts separately:

4.1 Maintenance Engine

Maintenance Engine is a generic name for all the tools and methods that are part of AEMES and are available to the user for working with AEMES. In AEMES we make a distinction between two classes of tools: syntax-based tools (like simple program browsers, pretty-printers, and cross-referencers) and semantic-based tools (like data-flow analyzers, control-flow analyzers, and program slicers). The semantics-based tools of AEMES are based on the framework for static analysis of programs that we described earlier in this paper, whereas the syntax-based tools are directly implemented onto an internal representation of the programs that is maintained by the system.

The tools of the maintenance engine have no graphical user interface of their own.

Instead, they provide methods and functionality via a function call interface to the user interface part of AEMES. For that purpose, a communication protocol has been defined between the user interface of AEMES and the tools, which is based on a remote-procedure-call mechanism. This distinction between the functionality of a tool and the user interface of a tool is not visible to the programmer, and he or she will usually not be aware of it, but it is an important aspect of the architecture and has some advantages over letting each tool implement its own user interface.

4.2 User Interface

The user interface of AEMES is implemented as a stand-alone application running as a separate process; therefore we call it User Interface Application (UIA). Actions of the user are processed by the user interface, which, in turn, sends requests to the maintenance engine of AEMES and displays the results that are returned from the tools. All this is done through the aforementioned remote-procedure-call mechanism.

The strict separation of the Maintenance Engine and the user interface of AEMES offers many advantages:

- Consistency

  Since it is the UIA that is concerned with giving each tool its user interface in the environment, it is much easier to define a common look and to enforce a common behavior for all tools than it were if every tool implemented its user interface individually. A consistent look and feel makes it easier for the programmer to work with all the tools and the environment as such.

- User Control

  By letting the UIA implement and also control all the interfaces of the different tools it is much easier to let the user keep control of what is happening on the screen while he or she is working. This is true because the user is in control of all the user interfaces of the tools (the UIA), and the UIA, in turn, is in control of all the tools. This way, tools do not control each other *directly* - which can cause actions on the user interface that the user does not immediately understand and therefore lead to confusion - instead, the only instance that controls the behavior of tools is the UIA.

- Extensibility

  New tools can easily be added to AEMES. A tool simply has to obey the communication protocol between the Maintenance Engine and the UIA and it can be integrated into AEMES with very little effort. All that is needed is a new part in the UIA which provides a graphical user interface for that new tool. This can usually be implemented quite quickly, provided the UIA is designed carefully.

- Integration Mechanism

  The power of a tool environment lies not only in the capabilities of the individual tools, but also on how these tools work together. Therefore, a versatile mechanism for letting tools work together needs to be integrated. With our architecture, such a mechanism can be realized on the level of the UIA itself. This does not only make its implementation simpler than the one for a distributed mechanism would be, it also makes the integration mechanism more transparent to the user. In AEMES, we even go so far as to allow (and encourage!) a user to configure the exact degree of the integration of tools according to his or her own needs and desires.

  We call the mechanism that we have proposed for tool integration "the wireable user interface". However, it is beyond the scope of this paper to go into more detail, here. Descriptions of the wireable user interface integration mechanism can be found elsewhere [15].

4.3 Data storage

It is not very convenient to have the source code form of a COBOL program as a basis of what the tools are working on. Therefore, we partially parse the COBOL program when the user loads it into AEMES and store the result in an abstract syntax tree extended with "containers" and "annotations". Annotations are elements containing various pieces of information about the original program, whereas containers consists of as of yet unparsed elements. They will only be parsed if really needed by a tool.

Most parts of AEMES are implemented in Standard ML, except for the User Interface, which is written in C++, using the object-oriented application toolkit ET++. The communication between the user interface and the tools is done using standard UNIX communication mechanisms. The first prototype of AEMES is currently being implemented.

5 Conclusions and Future Work

We have proposed a new approach to building program analysis tools for reverse engineering. Based on the observation that semantics-based program understanding is indispensable in reverse engineering, we investigated the possibilities of semantics-based reverse engineering. Our conclusion is that denotational semantics and abstract interpretations are powerful mechanisms for static program analysis in the area of reverse engineering, thereby closing the gap between advanced programming language theory and reverse engineering. This is strongly confirmed by the practical experience we have gained in applying these methods to the problems of analyzing COBOL-74. Building implementations can be done in a straightforward manner in languages such as Standard ML. We also gave an overview of the reverse engineering environment we are developing. On the theoretical side we will continue to systematize our semantics-based reverse engineering approach, especially the concepts of the static analysis framework. On the practical side, we will carry on with the implementation of the environment AEMES.

Our approach is new in the sense that we use denotational semantics and abstract interpretations for developing reverse engineering tools and in that we show how these methods are made practical to analyze complex languages such as COBOL-74. The only other project we know of that tried to use some mathematical formalism was the REDO-Project [31]. But there, the goal was to employ the formal specification notation Z in order to model abstractions of program behavior.

We are convinced that the sound formalisms – be it formal programming language theory or formal methods – belong to the most powerful techniques available today. Like other research directions in software engineering, reverse engineering can not afford to leave such techniques unused while the problems to be solved are becoming more and more complex.

Acknowledgement

The authors would like to thank Ina Kraan for her helpful comments on early versions of this paper.

References

[1] Samson Abramsky and Chris Hankin, editors. *Abstract Interpretation of Declarative Languages*. Computers and Their Applications. Ellis Horwood, 1987.

[2] Z. Ammarguellat. A control-flow normalization algorithm and its complexity. *IEEE Transactions on Software Engineering*, 18(3):237–251, March 1992.

[3] P. Baumann, J. Fässler, M. Kiser, and Z. Öztürk. Beauty and the beast or A formal semantic description of the control constructs of COBOL and its implementation. Technical Report 93.39, Institut für Informatik der Universität Zürich, 1993.

[4] R. Cartwright and M. Felleisen. The semantics of program dependence. In *ACM SIGPLAN'89 Conference on Programming Language Design and Implementation*, pages 13–27, 1989.

[5] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction and approximation of fixpoints. In *ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[6] Carl A. Gunter. *Semantics of Programming Languages, Structure and Techniques.* MIT Press, 1992.

[7] Williams Ludwell Harrison. The interprocedural analysis and automatic parallelization of scheme programs. *LISP an Symbolic Computation: An International Journal*, 2(3/4):179–396, 1989.

[8] Ph. Hausler. Denotational program slicing. In *Proceedings of 22. Annual Hawaii International Conference on System Sciences*, pages 486–494, 1989.

[9] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[10] G. Huet and G. Plotkin, editors. *Logical Frameworks.* Cambridge University Press, 1991.

[11] Cliff B. Jones. The search for tractable ways of reasoning about programs. Technical Report UMCS-92-4-4, Department of Computer Science, University of Manchester, 1992.

[12] N.D. Jones and F. Nielson. *Abstract interpretation: a semantics-based tool for program analysis.* Oxford Science Publishing. to appear in Handbook of Logic in Computer Science.

[13] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.

[14] Gilles Kahn. Natural semantics. In *STACS*, number 247 in LNCS, pages 22–39. Springer-Verlag, 1987.

[15] M. Kiser. The wireable user interface of AEMES. Technical report, Institut für Informatik der Universität Zürich. to appear.

[16] D. Knuth. The remaining troublespots of ALGOL 60. *Communications of the ACM*, 10(10):611–617, 1967.

[17] W. Kozaczynski, J. Ning, and T. Sarver. Program concept recognition. In *7th Knowledge-Based Software Engineering Conference*, McLean, Virginia, September 1992.

[18] Peter Lee. *Realistic Compiler Generation.* MIT Press, 1989.

[19] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[20] Peter D. Mosses. *Denotational Semantics*, volume B of *Handbook of Theoretical Computer Science*, chapter 11, pages 575 – 632. The MIT Press / Elsevier, 1990.

[21] F. Nielson and H. Nielson. Finiteness conditions for fixed point iterations. In *ACM Conference on Lisp and Functional Programming*, pages 96–108, 1992.

[22] Flemming Nielson. A denotational framework for data flow analysis. *Acta Informatica*, pages 265–287, 1982.

[23] Flemming Nielson and Hanne R. Nielson. *Semantics with Applications.* John Wiley & Sons, 1992.

[24] J. Ning. *A Knowledge-Based Approach to Automatic Program Analysis.* PhD thesis, University of Illinois at Urbana-Champaign, October 1989.

[25] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Dep. of Computer Science, Aarhus University, September 1981.

[26] David A. Schmidt. *Denotational Semantics - A Methodology for Language Development.* WM. C. Brown Publishers, 1986.

[27] K. Slonneger. Executing continuation semantics: a comparison. *Software-Practice and Experience*, 23(12):1379–1398, 1993.

[28] G. Venkatesh. A framework for construction and evaluation of high-level specifications for program analysis tehcniques. In *ACM SIGPLAN'89 Conference on Programming Language Design and Implementation*, pages 1–12, 1989.

[29] G. Venkatesh. The semantic approach to program slicing. In *ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 107–119, 1991.

[30] Glynn Winskel. *The Formal Semantics of Programming Languages.* MIT Press, 1993.

[31] H. Zuylen. *The REDO Compendium of Reverse-Engineering.* John Wiley & Sons, 1993.