University of Szeged
Doctoral School in Mathematics and Computer Science
Ph. D. Program in Informatics

Source Code Analysis and Slicing for Program Comprehension

Summary of the PhD Thesis

by

Árpád Beszédes
Department of Software Engineering

Advisor:

Dr. Tibor Gyimóthy

Szeged
2004

"Computers are good at following instructions, but not at reading your mind."
— Donald E. Knuth

Introduction

The booklet summarizes the results of the author of the thesis entitled "Source Code Analysis and Slicing for Program Comprehension." The dissertation concentrates on the topic of source code analysis in software reverse engineering. The author developed methods, technologies and tools to aid program comprehension.

It is a well-known phrase that computers are powerful, but not very clever. In order to follow our orders we need to develop software for them. Software development basically means "explaining" the abstract views of the developer and imagined functionality of the software to the computer, that is to provide it in a form that can be executed by it: this form is nothing else but the machine code. Between these two extremes and throughout the software life cycle we derive different models at different levels of abstraction, which all describe the same software system. Among these models the source code is the one that represents the final link between man and the machine, since it is written by the programmers based on higher level models and is translated by the compiler—in an automatic way—to machine code. So the higher level programming language needs to be understood by the machine (the compiler, to be more precise). For the latter the size and complexity of the source code does not cause any problems, only the available resources act as limits. It is also true that the compiler does not *comprehend* the program, it only performs the required transformations mechanically. On the other hand, there are numerous reasons for human comprehension of the source: to understand the way it works, what it does, or perhaps draw conclusions about aspects of it. Program comprehension is not an easy task, especially when we consider the possible complexity and size of the problem. Furthermore, the code is often unknown to the person involved: it may be the work of other programmers or the rationale behind a programmer's own code may be forgotten over time.

The need for program comprehension can arise in many situations throughout the software life cycle, perhaps beginning with debugging, which is needed as soon as the source is first created, but also later in the verification phase. Furthermore, in the phase of evolution when, for software maintenance, the only reliable source of information is the source code itself. This is due to the fact that nowadays software systems are rapidly changing, the market and the evolving new technologies always demanding another version. The new versions are often created from the existing systems by evolution, which in most cases results in a rapid development and, in consequence, the higher level models are incomplete, out of accord with the code, or simply unexisting. These problems implied the discipline of *reengineering* within software engineering. The first step in this process is always the *reverse engineering* of existing systems, when—based on the available models—we try to identify a system's components and their interrelationships, and create representations of the system in another form or at a higher level of abstraction [4]. This process most of the times is performed by analyzing the source code of the system.

Based on the previous it is obvious that there is a real need for software tools that can aid program comprehension, because many tasks of the comprehension process are mechanical and with the help of tools they can be made simpler and faster. Similar to the compiler, which can produce different representations of the program needed for achieving its goal, different *code analyzer tools* can also be developed that are capable of creating other kinds of representations of the code. The real advantage

is that by analyzing the source code, creating various models of it and by answering other questions about the code the software can be presented in a more suitable form for human processing.

Tool-supported program comprehension has several problems. One such is the set of *facts* extracted from the system, the way of their representation and storage, and the exchange of the data with other tools. The set of extracted facts is commonly referred to as the *model* of the software system, while the structure of the data within is called the *schema*. The basis for successful data exchange among tools is a common schema and physical format. Another basic element of a comprehension tool is the *analyzer*, which extracts the facts by parsing and analyzing the source code and by deriving different relationships from it. After this, the tools themselves can be of help in many ways for program comprehension, for which one example is program slicing. When slicing a program we provide only a subset of it to be investigated, by which we reduce the size of the problem that must be dealt with.

We addressed the above-mentioned set of problems when we started to develop and do research within a framework consisting of basic technologies and a set of tools, whose several basic elements are already operational, internationally accepted and are used in real environments. The current results of this long-term research and development are a general reverse engineering framework called Columbus [6, 7, 8, 9], and the supporting technologies and tools. Currently we are dealing with the C/C++ languages, since these are among the most widespread ones and they are around long enough to induce the need for reengineering and comprehension of existing systems written in these languages. Some concrete results of our development are a general analyzer and schema for C/C++ and other tools based on source code analysis such as our dynamic slicing tool [2, 3, 12]. In our toolset for performing the experiments we have some external tools as well, whose fields we do not want to deal with currently. These are, for example, the different software visualization tools and a static slicer.

The author elaborated two important aspects of the program comprehension framework (see Figure 1). The first one is the basis of all analysis tasks, a general C/C++ analyzer front end. It produces models of the software according to a well-defined schema, and it provides the kind of features that an analyzer of a compiler does not need to address as special requirements for the task of comprehension. The other result is a special code analysis tool, a dynamic program slicer, which has a variety of applications. The tool implements two efficient dynamic slicing algorithms that represent novel techniques and, compared to previous solutions, are significantly more efficient. Furthermore, implementation peculiarities are provided for a real programming language (C) in much more detail than anywhere previously available.

The results of the dissertation are grouped into four points, of which the first one deals with the first topic and three with the second. This summary is then also composed of two parts, having at the end of each part a summary of the corresponding results.

I C/C++ Source Code Analysis and Model Creation

I.1 Difficulties with the Analysis of C/C++

The C++ programming language is considered to be one of the most complex, mainly because it combines the low level programming possibilities (like direct memory access and pointer arithmetic) with high level paradigms (object-orientedness and generic programming).[1] This is of course an advantage from the programming-technology point of view, however it also causes the C/C++ to be very hard to analyze. The difficulty arises from the complex syntax full of ambiguities and a vast number of further semantic rules and other regulations that comprise the language.

Processing source code has basically two goals. The first is to compile it into an executable program using a compiler, and the other is our topic, code analysis. The latter is performed in achieving a number of goals in the fields of software maintenance, reverse engineering, documentation, program understanding and validation. A common property of the two processings is that a component is needed which analyzes the source code according to the syntactic and certain semantic rules of the language and creates its representation at a higher level of abstraction for further processing. This module is commonly referred to as the *front end*. The representation produced by the front end of a compiler is the abstract syntax tree and the data derived from it, which will be used to produce the machine code. Similarly, in the second case we also need to produce a higher level model of some kind, but in most cases it needs to be different from the one used by the compiler. Consequently, in a general purpose source code analyzer aimed at program comprehension and maintenance a special front end needs to be employed that bears special characteristics. Front ends of analyzers are commonly referred to as *fact extractors*.

Apart from recognizing the language correctly, a general purpose analyzer front end needs to fulfill a number of requirements that the compiler front end does not need to address. For this reason the technologies used by compilers are not suitable for our needs. In [9] we published a list of requirements, from which we overview the most important ones below:

- *Completeness*. Because of generality the parser needs to perform a complete analysis of the source, and the so-called "fuzzy" parsing technique is not enough, although it is used by many fact extractors but they are then capable of only a partial representation of the language.

- *Level of detail*. The parsers and models used by the compilers typically leave out details such as punctuations, parentheses, comments and redundant declarations. However, in many applications with fact extraction original, source-complete representation is required, so the front end needs to collect practically all available information from the source code.

- *Connectivity*. The internal representation produced from the source code will be utilized in arbitrary applications, so it is advisable to provide the possibility for easy connection with the remaining parts of the analyzer.

- *Fault tolerance*. A general front end needs to be capable of extracting as much information as possible from even incomplete or erroneous (uncompilable) code. Hence it needs to possess better fault tolerance capabilities compared to a compiler.

- *Analysis speed*. The analysis speed is less critical in the case of a general analyzer than for a compiler, since the latter is executed more frequently during the development. However, an

[1]The C++ practically includes the C language (its procedural predecessor), so in the following these two will be treated jointly.

analyzer used for the purpose of maintenance needs to process a large amount of source code at a stretch, so the speed issue cannot be neglected completely.

- *Preprocessing*. Obviously, the preprocessing needs to be performed in the case of a C/C++ front end (such as expanding the macros and processing include files). Besides this, preprocessor related facts need to be collected (to produce, for example, an *include-hierarchy* output).
- *Language dialects*. The C and C++ languages have many dialects, mainly because of language extensions defined by the compilers. Since we cannot afford to bind a general front end to a specific compiler, it needs to handle the major dialects. Furthermore, for the handling of unknown dialects an extendibility mechanism is desirable regarding the recognized language.
- *User interface*. The most general usability of a front end can be achieved if it can operate also in command-line mode, since in that case it can be invoked from any tool. Furthermore, most real software systems that need to be analyzed are equipped with some kind of a build system, into which the easiest integration is through a command-line operated front end.

Before we developed the front end we examined the available solutions for the same purpose. Some of them are based on compilers, so these are not suitable for general use. Next, those analyzers that are part of integrated development environments or other general program comprehension tools often extract only partial information from the source code or they do not provide suitable interfaces to promote data exchange. Hence we developed a general front end for C/C++.

I.2 The C/C++ Analyzer Front End

Due to the complexity of the languages the parsing cannot be solved using efficient parsing technologies ($LL(1)$ or $LALR(1)$). Furthermore, a complete front end needs to handle the representation of the analyzed source and to provide it in a suitable form for the remaining parts of the analyzer tools. Finally, it also needs to meet the above-mentioned functional and non-functional requirements.

We solved the completeness of the analysis and the sufficient level of details by utilizing two tools in our front end. Similar to the C/C++ compilers, a separate preprocessor is responsible for preprocessing the source, one that also collects the required preprocessor-related facts. The preprocessed code is further processed by the main language analyzer that also prepares the corresponding model. The preprocessor creates the preprocessed source (the `.i` file or an internal data stream) according to the language rules, the output being the same as that created by the compiler. In addition, it creates the model according to the preprocessor schema, which can be used for other analyses. We published this schema in [20] containing all the detailed information about the low level details of the source code, such as information about the tokens and other preprocessing information like about macros and their substitutions. The preprocessor operates using a similar technology as we present shortly for the language analyzer, so we will not provide these details here. In the next phase the language analyzer processes the preprocessed source and creates the appropriate language model. The two models together comprise all the information extracted from the original source code. In the present work we deal with the C/C++ language analyzer that comprises of the syntactic parser and the suitable mechanism for creating the model.

The analyzer is a command-line program that promotes its most general application. The input of the program is a complete preprocessed translation unit that will be processed by the syntactic parser by checking it against the syntax. The internal representation is built up in parallel with the parsing in a syntax-driven fashion. This means that as the parser processes a language element it also executes the so-called *syntactic actions*. The actions will practically comprise an interface to the parser, which

defines the operations according to the syntax. By the realization of the interface the operations called by the parser can be used to perform the "useful work," to build up the internal representation in our case, that is the source code model. The model is an Abstract Syntax Graph or ASG, that complies with the schema defined by our framework. The ASG corresponds to the abstract syntax tree extended with various relations among the nodes, like the resolved name references, but we do not perform deeper analyses on the graph such as flowgraph computation. The ASG also supports the parser, since for the correct parsing of C++ we need to store various structured information about the symbols. For example, the meaning of the individual identifiers needs to be recorded, which follows from the context (type, variable, etc.), as the syntactically ambiguous situations of the parsing can be resolved only with this kind of information. This information is acquired from the partial ASG that is just being built. The model is also written into a file, which will provide the basic connectivity for the remaining parts of the analyzers, furthermore an application programming interface (API) is also provided for processing the model.

The structure of the parser is traditional: the input text is first tokenized, that is we perform the lexical analysis on it, which will produce the token-stream for the syntactic parser as its input. The lexical analysis can be relatively easily managed using deterministic finite-state automata, since we can provide the corresponding language definition using regular expressions. The lexical analyzer performs some other low level processings as well, such as recognizing the comments, white-spaces and so on. Since the C++ language is not easy to parse, several techniques are proposed in the literature for that purpose, but almost none of them is pure parsing strategy. The result is generally some hand-written parser extended with a number of solutions to aid the basic parsing strategy. Some researchers even suggest using very expensive technologies for handling the complexity of the language, like some methods developed for natural language processing.

The technology employed in our parser is top-down (which is also suggested by the creators of the language and is used by many compilers as well) based on strong $LL(k)$ with $k = 2$. The algorithm is based on recursive descent procedures, where each procedure corresponds to a rule of the grammar. A procedure's body corresponds to the alternatives of the rule, in which the selection of the right alternative is assured by the strong $LL(k)$ condition. A nonterminal on the right-hand side means calling the corresponding procedure, while a terminal means advancing in the input, according to the order of the symbols in the rule (leftmost derivation). The parser is automatically generated from a grammar description using the PCCTS system [17] that generates the C++ source code of both the lexical and the syntactic parser. PCCTS supports the so-called *predicated* $LL(k)$ parsing that is a very powerful technique for resolving parsing ambiguities (see [16]). Basically, the method is to use the so-called *syntactic* and *semantic predicates*, which will provide the direction of parsing at those alternatives where the strong $LL(k)$ condition does not hold and because of this the grammar is ambiguous. Using the syntactic predicates we get practically infinite lookahead, while with the semantic predicates the semantic information about the next several symbols in the input can be used to resolve the ambiguities.

For creating the source code model (ASG) syntactic action interface mentioned above is used. The top-down parsing employed with recursive descent procedures provides a logical way of to guiding the building process. For this we use a mechanism, where during the execution of the recursive procedures we execute an action at each relevant point, and the implementation of the action will actually perform the building task. Actions are typically put at the beginning and end of the alternatives, and surrounding the procedure calls that correspond to the nonterminals on the right-hand sides. By taking advantage of object-orientedness, the actions provide a possibility for an even more general use of the parser, apart from the creation of the ASG. The interface provides a component-level reuse of it in a specialized tool with even more possibilities.

The built-up model corresponds to the Columbus schema, which evolved into its current, stabile form in parallel to the parser [5, 11]. During the building process a number of helping structures and algorithms are used, since the calling order of the actions by the parser is fixed, which is not always directly appropriate for the building. Furthermore, due to the recursively nested structure of the language at many places we need to use different stacks to store the built-up entities. Another relatively complex solution was needed for the resolution of names, that is to find the right entity referred by a name. For example, in the case of function calls this includes the determination of types of arguments incorporating the standard conversions, constant propagation and overload resolution. Apart from these, there are a number of auxiliary processings that the builder needs to perform, which we will not elaborate in detail here.

To meet the requirements overviewed above we employed a number of special solutions in the C/C++ front end. For high fault tolerance we built into the parser a mechanism mainly based on heuristics that is activated in the case of a syntax error. When that occurs after the error-message the parser tries to recover itself and continue the parsing at a suitable point afterwards. The essence of the solution is that in case of an error we skip several tokens on the input up to a token corresponding to some significant language element, such as a semicolon, which terminates declarations and statements. In parallel to this the parser will continue the analysis just at the according rule. To improve the analysis speed we provide a so-called precompiled headers technique, in which the commonly used header files that contain a large amount of common code are pre-analyzed and the so built up models are later used by the analyses of subsequent translation units. The C++ parser accepts the language defined by the standard and the model is also built up accordingly. Besides this, the parser supports the dialects of the Microsoft, Borland and GNU g++ compilers, furthermore, there is a possibility of extending the recognized language in a certain amount by giving a list of some pre-defined symbols to the parser.

The front end provides some other features to promote program comprehension as well. Such is the instantiation of C++ templates at source code level, about which more information can be found in [10].

The usefulness of the front end has been proved in several cases with real size systems. The analysis speed and the memory consumed are both suitable for very large systems as well. For example, in [9] we showed that the memory consumption is nearly linear with the size of the program (by considering e. g. the number of classes). Furthermore, the analysis time is comparable to the compiler's. In the mentioned experiment we investigated real size systems consisting of up to 5,000 classes.

The analyzer front end is the basis of all analysis tasks, including the simpler fact extractors which, for example, create a formatted documentation of the code, but other more complex algorithms as well, like the computation of dataflow information. However, for the successful analysis of a complete software system a number of other components are needed as well. In our case these are provided by the Columbus framework [6, 9], which gives some important features like project handling, model merging and filtering. The Columbus system is not bound to any programming language, currently the C++ front end is invoked as a command-line program. The current Columbus system provides several useful processings for C++ as well. There are formats corresponding to the schema, like XML-based models, a UML static structure (class) model and exchange formats defined by other tools. Apart from this, Columbus can produce further derived outputs as well that serve some special goal, like the computation of metrics, recognition of design patterns and source code auditing. The Columbus system and its outputs are used in several academic cooperations of ours. Until now we have registered more than 600 academic users worldwide.

Independent of the Columbus system, the front end can be used in numerous other applications as well; in practically any field where there is a need for analyzing C++ code. Using the interface of

syntactic actions there are even more possibilities of the parser's reuse. Using this kind of extension, so far we have experimented with different *source code instrumentation* applications (here the original code is extended with such instructions that perform some additional tasks at runtime, preserving the original behavior as well). For example, we employ instrumentation in the dynamic slicing application to produce the execution trace. Dynamic slicing is the topic of the second part of the present work.

The place of the front end within our program comprehension framework is shown in Figure 1.

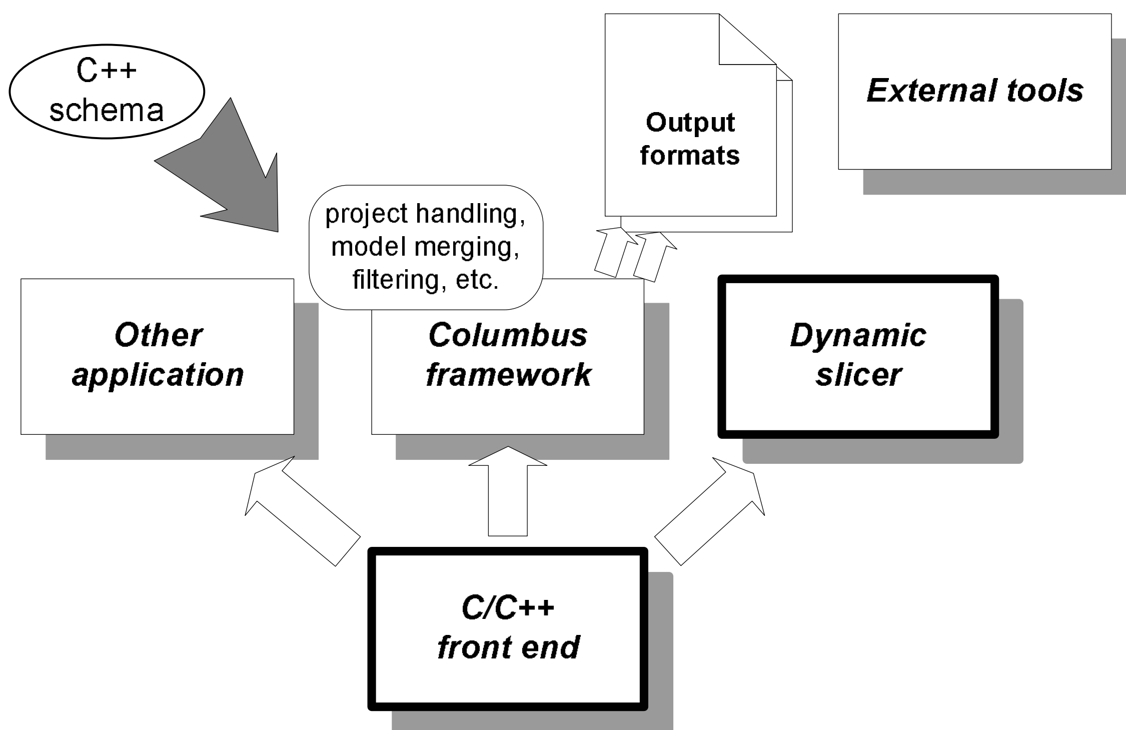

Figure 1: The C/C++ front end within the program comprehension framework.

I.3 Summary of Results of Part I.

The author has achieved the following results regarding C/C++ source code analysis and model creation:

I/1. *C/C++ Source Code Analysis and Model Creation.* The author developed the necessary technologies and implemented an analyzer front end that meets the special requirements for a general purpose analyzer. Furthermore, the tool creates a model of the source code according to a well-defined schema and provides interfaces for easy extendibility. Among others, we used novel solutions for tackling fault tolerance, speed of parsing and language dialects.

Results found in related publications We introduced the C++ schema in [5] according to which the model is created by the front end (the schema is not, but the building technology is the result of the author). In [9] we published the requirements for a general C++ front end. Furthermore we described the analyzer with its main features and provided some measurement results. These are the author's achievements. The preprocessor and the preprocessor schema was published in [20] along with some example models (the preprocessing schema is not the result of the author). The model creation strategy employed in the preprocessor is based on the technology used by the C++ analyzer, which is the author's own work.

II Dynamic Program Slicing

II.1 Program Slicing

The notion of *program slicing* and the slices have been defined in many different ways in the literature, but the basic idea is always the same: to attempt to reduce the size of the problem by achieving to investigate only those parts of the program to be analyzed that are relevant from a specific point of view. Program slicing [18, 21] is a program analysis technique proposed for many software- and reverse engineering fields including verification, maintenance, reengineering, program comprehension and debugging. Generally speaking, a slice of a program is its subset comprising of those instructions that directly or indirectly influenced or may influence the values of given variable occurrences at a given program point (this is known as the slicing criterion). The process of determining the subprogram is called program slicing.[2] The so reduced program will evaluate the involved variables equivalently at the given occurrence. In fact, some slicing algorithms actually compute executable programs, while others can produce arbitrary subprograms. However, executable slices are required only by some applications, but for these it should be said that these slices are less precise.

If we determine the subprogram such that it involves the relations for any possible execution then it is referred to as *static slicing*, whereas if only one specific execution is addressed then it is *dynamic slicing*. A dynamic slicing criterion also includes the parameters of a concrete execution of the program (a test-case with a set of program inputs) and a specific occurrence of the instruction involved during the execution (since an instruction may appear several times in the execution the different occurrences will be referred to as *actions*, which include the instruction and its serial number in the execution as well). The sequence of actions corresponding to the execution for a given test-case is called the *execution trace*, which includes some other information about the execution as well. In Figure 2 an example is given to illustrate the difference between the static and the dynamic slice (the former includes the whole program).

Over time a number of slicing methods have been developed. The majority of the practical methods compute the slices based on various dependences (control and data) among the program elements (variables, instructions, addresses, predicates, etc.). The literature is elaborate about the details of static slicing methods. For example, the work by Horwitz *et. al.* [13] served as the basis for a number of subsequent implementations and enhancements, whose basis is the program dependence graph – the PDG. However, relatively few publications appeared that deal with the practical sides of dynamic slicing and provide detailed algorithms. The fact that it is hard to find practically used algorithms can mean that the published methods are not suitable for handling real size programs.

The existing dynamic slicing algorithms employ different approaches. The first dynamic slicing algorithm was given by Korel and Laski that produced executable slices [15]. As it is known, executable slices are generally significantly larger than those that deal with the dependences only (according to Venkatesh's measurements the ratio is about 2–3 times [19]), however in our work we do not require this property, so we are working with a lot more precise slices. One of the most significant dynamic slicing algorithms was proposed by Agrawal and Horgan [1], based on which several enhanced methods were also published later [14, 22]. The basic method is based on the dependences among actions that are jointly represented using a *dynamic dependence graph* or DDG. The nodes of the graph correspond to the actions, while the edges represent the data and control dependences among them. To compute the dynamic slice based on the DDG is then simple: starting with the action from the criterion we traverse the graph and the instructions at all the reachable nodes will comprise the slice. The biggest

[2]This is the common understanding of slicing, which is also known as *backward slicing*. In contrast, a *forward slice* consists of those instructions that are dependent on a given program point. In the present work we deal only with backward slicing.

```
     #include <stdio.h>
     int n, a, i, s;
     void main()
     {
 1.    scanf("%d", &n);
 2.    scanf("%d", &a);
 3.    i = 1;
 4.    s = 1;
 5.    if (a > 0)
 6.      s = 0;
 7.    while (i <= n) {
 8.      if (a > 0)
 9.        s += 2;
         else
10.        s *= 2;
11.      i++;
       }
12.    printf("%d", s);
     }
```

```
     #include <stdio.h>
     int n, a, i, s;
     void main()
     {
 1.    scanf("%d", &n);
 2.    scanf("%d", &a);
 3.    i = 1;
 4.    s = 1;
 5.    if (a > 0)
 6.      s = 0;
 7.    while (i <= n) {
 8.      if (a > 0)
 9.        s += 2;
         else
10.        s *= 2;
11.      i++;
       }
12.    printf("%d", s);
     }
```

Figure 2: Static and dynamic slice for the criterion $(\langle \mathtt{a}=0, \mathtt{n}=2\rangle, 12^{15}, \{\mathtt{s}\})$

drawback of this method is that the graph can be very large since it has as many nodes as there are actions in the execution, while the number of edges is determined by the dynamic dependences that arise. The result is a graph of a practically unmanageable size in the general case. Besides this problem, the methods published are not given with sufficient detail regarding the analysis of real programming languages; they do not elaborate on the handling of numerous significant problems.

II.2 Efficient Dynamic Slicing Algorithms

Our approach to computing dynamic slices differs from the previous methods in several respects. The algorithms can be efficiently implemented, hence they can be used for the analysis of real size programs. This can also be attributed to the fact that the amount of data that needs to be maintained by the algorithms was kept to the minimum using simple data structures. We give two algorithms that are based on dynamic dependences. The first is *global*, meaning that it computes all dynamic slices for a given execution, that is it does not use a slicing criterion (the global algorithm was first published in [12], which was followed by its extension to C programs in [3]). The second algorithm is *demand driven*, which means that it computes a single dynamic slice based on a single request (the criterion). Both algorithms compute the same slices as the traditional dynamic dependence graph-based method, but it is very important to note that we use a very different approach. The algorithms have their advantages and drawbacks and therefore their application fields also differ. First we overview the basic principles of the algorithms and then we take a look at the necessary extensions for slicing C programs.

Computing the dependences among program elements is the basis of the algorithms. In both methods we follow the dynamically occurring data and control dependences among the statement occurrences (actions). In the case of the static dependence-based method all possible dependences need to be taken into account (see for example Horwitz *et. al.* [13]), and for that purpose the program dependence graph (PDG) needs to be constructed. Whereas in the dynamic case when a specific

execution is concerned, only the actually realized (dynamic) dependences need to be investigated. A control dependence between actions means the dependence on the action that was executed last among the (statically) potential dependences. Next, the realized data dependences are determined based on the last definition point of the variables. Based on this we do not need a complete static or dynamic dependence graph. Instead we compute a much simpler static structure that will be used by both algorithms in computing the slices.

The algorithms share a common static analysis phase, during which the so-called D/U program representation is built (definition-use). A statement of a program has the following D/U representation: $i.\ d:\ U$, where d is the variable that gets defined in instruction i, while the U set contains the variables that are used to compute the value of d at i. The representation captures the data dependences in such a way that it considers only the names of the defined and used variables, i. e. the relations between the actual occurrences need not be stored. Furthermore, the control dependences can be treated in the same way as the data dependences using this representation. The traditional methods did not employ such a generalization. The control dependences are represented by the so-called predicate variables, which are virtual variables corresponding to the direct control dependences among the instructions. After this the two algorithms compute the dynamic slice(s) using the D/U representation and the execution trace. Because of its simplicity, this data structure and its use do not introduce significant overheads.

The global algorithm starts processing the trace with the first executed instruction, and at each step it computes the dependence set corresponding to the defined variable at the actual instruction. For this it uses the most recently computed dependence sets of the used variables at the instruction and their last definition instruction (see [3, 12]). This way all dynamic slices corresponding to the defined variables at all instruction occurrences are attained and provided at the output (only the actually effective sets are stored in the memory). This algorithm is suitable for applications where all slices need to be computed with one pass through the execution trace. The number of iterations of the algorithm equals the length of the execution. In addition to this, the average computational complexity is determined by the average size of the dependence sets, which are generally in correspondence with the program size (this is practically the average slice size). The space requirements are determined as the product of the number and size of the sets: the first value is the number of all defined variables, while the second is the average slice size. It can be seen that the space requirement is significantly better than that of the graph-based method. The algorithm is shown in Figure 3. Here, $DynDep$ denotes the corresponding dependence sets, while LS is the last definition instruction. Furthermore, i^j is an action with the statement number and the execution step, while d is used to denote the defined variable in the corresponding instruction and U for the use set at that point.

The demand driven algorithm processes the trace each time for all individual slicing requests starting with the action of the criterion and traces back the dependences towards the very first action. The algorithm scans the dynamic dependences and keeps those actions not yet processed in a worklist. When it removes an action from the worklist it investigates all variables from the use set of the removed action and extends the worklist with the last definition action of those variables before the actual execution step. When all dependences have been processed and the worklist becomes empty the algorithm terminates by providing the slice with the instructions visited during the run. For the efficient functioning of the algorithm the trace needs to be stored in a special form that groups the actions according to the variables defined in them. This is represented by the so-called EHT (execution history) table, whose rows are constituted of the actions with the corresponding defined variables. This is needed because in every iteration of the algorithm an arbitrary definition action corresponding to the variables could be needed, which means that there is a need for searching among the execution steps backwards. The number of iterations of the algorithm varies, it is minimum

```
program  GlobalAlgorithm(P, x)
input:   P : a program
         x : a program input
output:  dynamic slices for all (x, i^j, U(i)) criterions

begin
  Store execution trace
  for j = 1 to number of steps
    i := statement executed at the jth step
    DynDep(d(i)) := ⋃_{u_k ∈ U(i)} (DynDep(u_k) ∪ {LS(u_k)})
    LS(d(i)) := i
    Output DynDep(d(i)) as the dynamic slice for criterion (x, i^j, U(i))
  endfor
end
```

Figure 3: Global algorithm

the number of instructions in the slice computed, but in the worst case it can be as much as the length of the execution. However, in the average case it will be correlated with the size of the slice. Beyond this the computational complexity is determined by the lookup operations in EHT, which is logarithmic with the number of steps executed. As for the space requirements, not counting the storage of the table (it can be kept on the disk), it is not significant since only the worklist needs to be maintained in the memory.

Comparing the two algorithms we may say that the global algorithm should be used when many slices need to be computed at once, for example in the applications of the union or decomposition slices. For a demand driven computation of a slice both methods can be used. If the storage of the execution history in the table form is not feasible then the global algorithm should be chosen. For individual tasks generally the demand driven algorithm is the better choice because of its better computational and space requirements. However, for very long executions this algorithm is suitable only if the storage of EHT and the lookup in it can be managed in an efficient way. Finally, if many slices are to be computed by executing the demand driven algorithm individually, above a certain number of the required slices the total cost will be better using the global algorithm.

In order to apply the algorithms to C programs several special problems needed to be solved and extend the algorithms with these. However, these extensions do not influence significantly the complexities. In our approach the execution trace is created by executing an instrumented version of the original program that will create all the necessary information about the execution. The instrumentation and the static analysis is performed using our C/C++ front end with the above-mentioned interfaces. This means that the extended D/U representation is derived from the ASG, while the instrumented code is created by reusing the action interface (in such a way that we access the recognized tokens and write them back supplemented with the instrumentation instructions).

We summarize the most important extensions for slicing C programs in the following points:

- Probably the most important extension is how we treat the different scalar variables, pointer and composite objects in a uniform way. This is done by performing all computations on *memory locations*, which has the advantage that the handling of the mentioned program elements will be significantly simplified since all dynamic information is available about the actual states of the objects of the current execution (as opposed to the static case where all possibilities need to be investigated). First we transform everything to memory addresses: we use the actual address

taken by the scalars, and the concrete values of pointers and array- and structure-elements, furthermore we also transform the variables in the criterion to memory addresses. After this the algorithms use these addresses as the "variables."

- The D/U program representation needs to be extended in several aspects, but these extensions do not influence the complexity. One such is that we introduce several other virtual variables for handling pointer indirections and accesses of structure fields, and for example for handling function calls as well. To handle pointers and other memory indirections (array-elements and structure fields) we use the so-called dereference variables that are used symbolically in the static D/U. These will be resolved to real addresses during the processing of the trace.
- A significant extension is the handling of arbitrary control dependences. More precisely, because of the unstructured control transfers (like the *goto* statement) in C a complete control dependence graph needs to be built, and based on it we create the appropriate predicate variables of the D/U representation. In the static case a statement can depend on more than one predicate, but in a concrete execution only one dependence will be realized. This is handled by the slicing algorithms by considering the last executed predicate among the potential dependences.
- Some further details are the handling of library functions by pre-created D/U structures, common handling of more compilation units and the mapping of physical program line numbers to logical statement numbers.

Apart from the extensions above, the demand driven algorithm deserves some further attention, since in this case another auxiliary structure needs to be maintained. This is the so-called AHT, the address history table, which stores in a compact form the actually taken addresses by the scalar and dereference variables throughout the execution. It is used during processing the execution to look up the address taken by an arbitrary variable at an arbitrary execution step.

The most important factor that influences the conceptual complexity of the algorithms is the fact that we are working with memory locations instead of simple (statically known) variables. This has the consequence that the number of all such used variables will be known at runtime only, which will be determined by the number of different memory locations used by the program. The result is that we need to use some more expensive data structures and algorithms at some points of the implementation. However, according to our measurements the number of different addresses is generally not significant compared to the size of the program and the static number of variables, and it is untypical for these values to be dependent on the length of the execution, hence this does not introduce unmanageable extra costs.

We performed a series of extensive measurements with the algorithms, and with the results we support our theoretical reasonings about the complexities of the algorithms. Namely, the measured execution time and memory consumption of the algorithms are much better than the complexities determined for the worst cases, they are better by several orders of magnitude. Furthermore, with both algorithms all significant factors are correlated with the program size or the number of different memory locations used by the program, rather than the number of instructions executed. We did not directly measure the performance and memory consumption, since the current prototype, unoptimized implementation would not represent real results. Instead we measured different parameters of the computations performed by the algorithms during execution (like internal data structures and numbers of steps), with the help of special routines built into the implementation. We used five small and medium size programs for the measurements that included the investigation of the sizes of the program representation, the slice sizes and the computational and space requirements of the two algorithms. We confirmed that the demand driven algorithm is generally faster in computing one slice not counting the building of the tables.

II.3 Applications of Dynamic Slicing

The most important application of dynamic slices is in debugging. For this we provided an extension for computing the so-called *relevant slices*, which provide more reliable results for certain code constructs than the traditional dynamic slices [12]. The problem is that in some cases the dynamic slice does not include certain instructions that actually did not affect the variables of the criterion, but they could have had effects on those variables had they been evaluated differently (we say that the variables are *potentially* dependent on the given instructions). We should mention that in this case we need to count with some static dependence information as well, since the investigation of all possible evaluations of the instructions is static processing.

However, the application range of dynamic slicing is much wider, for instance using our global algorithm many slices can be computed efficiently, so many maintenance and testing tasks can be aided. As a concrete application we elaborated on the method for computing the *union slices*, where the union of dynamic slices is determined for different executions of the program [2]. Using the union slice we can approximate the so-called *realizable slice* that exists only in theory and it represents the union of the dynamic slices for all possible executions of the program. The realizable slice is smaller than the static slice, but in practice it can only be approximated using the union slices computed with different test-cases. The union slice is bigger than any individual dynamic slice but is significantly smaller than the static slice, hence it is more precise. We support this by our measurements, where we found that the difference between the static and the realizable slice can be significant. The results of our experiments showed that the dynamic slices are generally small (about 5% of the program), while the static slices are much larger (more than 70% of the program on average). At the same time, the growth rate of union slices significantly declines after only a small number of representative executions added, and this tendency seems to be unchanging, from which we draw the conclusion that the relevant slice has been well approximated (see Figure 4). In addition, the size of the union slice will be significantly smaller than the one of the static slice, according to our measurements it is about 15% of the program. The effect is that union slices can be used in many applications instead of the static slice. As an example, for solving a certain problem in software maintenance we can start our investigations with a union slice computed for a well-chosen set of test-cases, and because the subset of the program will be significantly smaller we will probably perform the task more efficiently.

II.4 Summary of Results of Part II.

The author has achieved the following results regarding dynamic program slicing:

II/1. *Global Dynamic Slicing Algorithm for C.* We developed a global dynamic slicing algorithm, whose extension for real procedural languages (C, here) was given by the author. The algorithm determines all dynamic slices for a given execution of the program. The method includes the detailed solution for the interprocedural computation, arbitrary control transfers, dependences of all kinds of data, the collection of runtime, dynamic information about the execution of the program, and so on. To derive the static information the C/C++ front end given here is used. The efficiency of the algorithm was verified by complexity studies and detailed measurements.

Results found in related publications The basic global dynamic slicing algorithm was first published in [12], which is not the result of the author. The elaboration of the method, the design of the prototype and the measurements are the work of the author. In [3] we describe the details for slicing C programs, whose elaboration is the result of the author. The paper was considered to be the best paper of the conference, the European Conference on Software

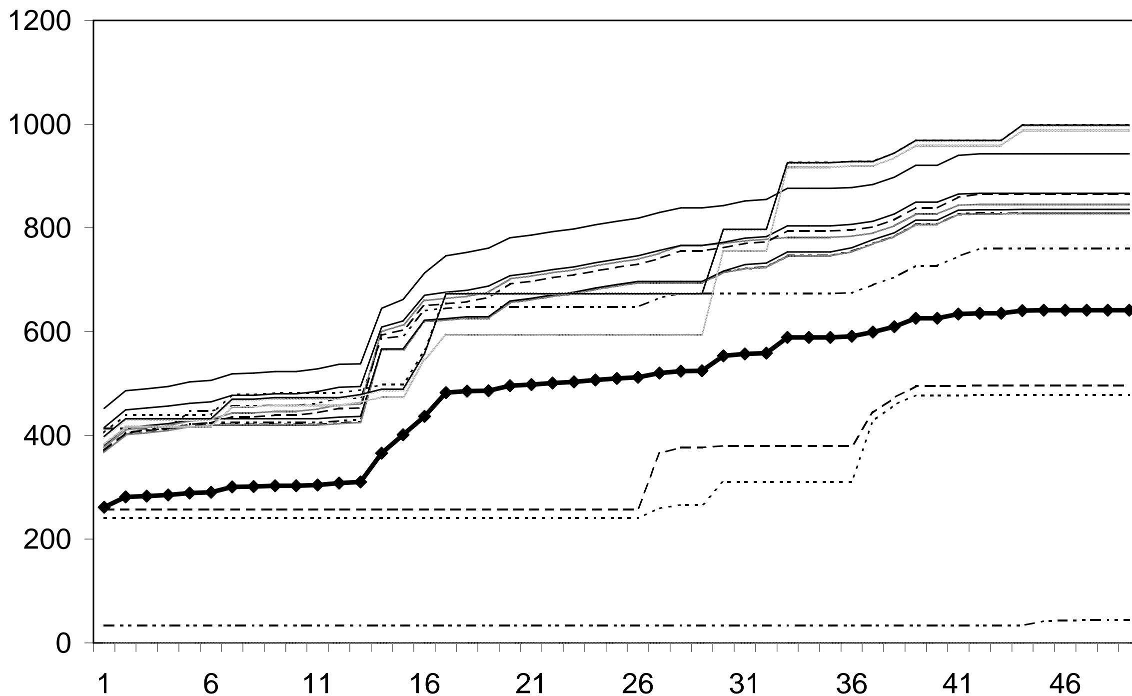

Figure 4: The growth of the union slices for an example program. The horizontal axis represents the addition of the test-cases, while the vertical one is the size of the union slice in number of instructions. The curves correspond to different criterions, the thick one being their average.

Maintenance, which is the biggest European, and one of the most significant international software maintenance conferences.

II/2. *Demand Driven Dynamic Slicing Algorithm for C.* The author elaborated a demand driven dynamic slicing algorithm and provided the details for slicing real procedural languages (C, here). The method uses the same static information but the processing of the execution is done differently and it computes one slice based on a single request. The efficiency of the algorithm was also verified by complexity studies and detailed measurements.

A paper describing the method is being prepared at the time of writing.

II/3. *Applications of Dynamic Slicing.* Among the numerous applications of dynamic slicing, the relevant slices are used for debugging, whose details were elaborated by the author. Another application is the use of union slices for software maintenance, this technique being the work of the author. Extensive measurements were also used to support the grounds of union slices.

Results found in related publications We published the computation of relevant slices in [12], in which the basic algorithm is not the result of the author. The elaboration of the method, the design of the prototype and the measurements are the work of the author. In [2] we published the motivation and the technology for the computation of union slices. These, along with the measurements and their evaluation are all the author's work.

References

[1] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, number 6 in SIGPLAN Notices, pages 246–256, White Plains, New York, June 1990.

[2] Árpád Beszédes, Csaba Faragó, Zsolt Mihály Szabó, János Csirik, and Tibor Gyimóthy. Union slices for program maintenance. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2002)*, pages 12–21. IEEE Computer Society, October 2002.

[3] Árpád Beszédes, Tamás Gergely, Zsolt Mihály Szabó, János Csirik, and Tibor Gyimóthy. Dynamic slicing method for maintenance of large C programs. In *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering (CSMR 2001)*, pages 105–113. IEEE Computer Society, March 2001.

[4] Elliot J. Chikofsky and James H. Cross II. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17, January 1990.

[5] Rudolf Ferenc and Árpád Beszédes. Data exchange with the Columbus schema for C++. In *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering (CSMR 2002)*, pages 59–66. IEEE Computer Society, March 2002.

[6] Rudolf Ferenc, Árpád Beszédes, and Tibor Gyimóthy. Extracting facts with Columbus from C++ code. In *Tool Demonstrations of the Eighth European Conference on Software Maintenance and Reengineering (CSMR 2004)*, pages 4–8, March 2004.

[7] Rudolf Ferenc, Árpád Beszédes, and Tibor Gyimóthy. Fact extraction and code auditing with Columbus and SourceAudit. In *Proceedings of the 20th International Conference on Software Maintenance (ICSM 2004)*, pages 513–513. IEEE Computer Society, September 2004.

[8] Rudolf Ferenc, Árpád Beszédes, and Tibor Gyimóthy. *Tools for Software Maintenance and Reengineering*, chapter Extracting Facts with Columbus from C++ Code, pages 16–31. Franco Angeli Milano, 2004.

[9] Rudolf Ferenc, Árpád Beszédes, Mikko Tarkiainen, and Tibor Gyimóthy. Columbus – reverse engineering tool and schema for C++. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2002)*, pages 172–181. IEEE Computer Society, October 2002.

[10] Rudolf Ferenc, Ferenc Magyar, Árpád Beszédes, Ákos Kiss, and Mikko Tarkiainen. Columbus – tool for reverse engineering large object oriented software systems. In *Proceedings of the Seventh Symposium on Programming Languages and Software Tools (SPLST 2001)*, pages 16–27. University of Szeged, June 2001.

[11] Rudolf Ferenc, Susan Elliott Sim, Richard C. Holt, Rainer Koschke, and Tibor Gyimóthy. Towards a standard schema for C/C++. In *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE 2001)*, pages 49–58. IEEE Computer Society, October 2001.

[12] Tibor Gyimóthy, Árpád Beszédes, and István Forgács. An efficient relevant slicing method for debugging. In *Proceedings of the Joint 7th European Software Engineering Conference and 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSE'99)*, number 1687 in Lecture Notes in Computer Science, pages 303–321. Springer-Verlag, September 1999.

[13] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–61, 1990.

[14] Mariam Kamkar, Nahid Shahmehri, and Peter Fritzson. Interprocedural dynamic slicing. In *Proceedings of the 4th International Conference on Programming Language Implementation and Logic Programming (PLILP'92)*, volume 631 of *Lecture Notes in Computer Science*, pages 370–384. Springer-Verlag, 1992.

[15] Bogdan Korel and Janusz W. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, October 1988.

[16] Terence J. Parr and Russell W. Quong. ANTLR: A predicated-$LL(k)$ parser generator. *Software – Practice and Experience*, 25(7):789–810, July 1995.

[17] The PCCTS web site. `http://www.antlr.org/pccts133.html`.

[18] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, September 1995.

[19] G. A. Venkatesh. Experimental results from dynamic slicing of C programs. *ACM Transactions on Programming Languages and Systems*, 17(2):197–216, March 1995.

[20] László Vidács, Árpád Beszédes, and Rudolf Ferenc. Columbus schema for C/C++ preprocessing. In *Proceedings of the Eighth European Conference on Software Maintenance and Reengineering (CSMR 2004)*, pages 75–84. IEEE Computer Society, March 2004.

[21] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, 1984.

[22] Xiangyu Zhang and Rajiv Gupta. Cost effective dynamic program slicing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 94–106, Washington, D. C., June 2004.