Program Slicing to Understand Software Generators

Max Lillack a 
Johannes Müller b 
Ulrich W. Eisenecker a

a {lillack, eisenecker}@wifa.uni-leipzig.de
University of Leipzig
Information Systems Institute
Leipzig, Germany
b jmue933@aucklanduni.ac.nz
University of Auckland
Department of Computer Science
Auckland, New Zealand

ABSTRACT

Many large mainframe based business applications were developed with proprietary macro-based generator systems. To maintain these applications the legacy generators employed to generate them have to be maintained as well. This is a problem because of the generator code that is often hard to understand. A solution would be to replace the old generator technology with a new one. However, this requires a deep understanding of the old generators and the features of the programs they generate. This understanding is currently not available. We propose program slicing as a technique to support program understanding of generators and show its application in four use cases in that domain. We show the feasibility of the technique using real-world generators.

Categories and Subject Descriptors

D.2.7 [Distribution, Maintenance, and Enhancement]: Restructuring, reverse engineering, and reengineering

Keywords

Program Slicing, Macros, Software Generators, Program Comprehension

1. INTRODUCTION

In the 80s and 90s many large mainframe based business applications were developed with programming languages like Cobol and PL/I. These languages are rather complex and hard to handle, which is why many development projects employed program generators based on IBM Pacbase [2], FSP’s SWT01 [1] or DSTG’s ADS [3] to ease common software development tasks [19]. Many of these generators are macro-based, i.e. they introduce new language concepts around an existing target programming language like Cobol to facilitate program modularization, parameterization and composition as well as to eliminate repetitive tasks. Macros are written in a special programming language – the macro language – and embed statements of a target programming language. The macro languages used by the proprietary tools mentioned above differ with respect to their complexity, features and tools from the well-known macro language of the C preprocessor Cpp (see Section 2 for details).

The tools used to develop macro-based generators are now proprietary legacy technology with a small user base. This leads to two problems. First, experienced developers are rare and, second, tool vendors are ending their support. A straightforward approach to this problem would be to skip the generator and maintain the generated systems directly. However, this will lead to a maintenance nightmare because of the hardly understandable generated code and the missing features of the generator. Hence, the maintenance of a generated system is hard to ensure since any changes in the generated system require updates to an unsupported generator.

A possible solution is the modernization of the generator i.e. the replacement of legacy generator technology by easier maintainable, newer technologies. However, an important prerequisite for modernization is the understanding of the legacy generators. The understanding of legacy generators is complicated because of too much complexity caused by (1) the long history with many changes in the system and (2) the intermingling of macro and target language.

To cope with complexity several approaches have been developed. Program slicing is an established approach to reduce complexity. It allows a focused analysis of a defined aspect, i.e. the slicing criterion, of software source code.

Even though successful applications of program slicing indicate that it is a promising approach for supporting program understanding [6], it is currently not used for program understanding of legacy macro-based generators.

To fill this gap, we contribute a tool to slice generators (Section 4) and its application in four use cases related to understanding of generators. We present the use cases in Section 5 and the application to a set of real-world legacy generators in Section 6. But first we start in Section 2 with some background on macro-based generators and program slicing followed by a discussion on related work in Section 3. To conclude our paper, we summarize our findings and present an outlook to future research (Section 7).

2. BACKGROUND

We now introduce details on legacy macro-based generators and the properties of the used macro languages. Then we present details on program slicing and the algorithm we use to slice generators.
2.1 Macro-based Generators

The macro languages in a generator can have different properties which influence how they are connected to the embedded target language. Macros work on a lexical or syntactical level though combinations of both levels are possible [9]. Lexical macros do not depend on knowledge of the target language. They substitute tokens with character sequences. Lexical macros are independent of the target language they generate. On the other hand, syntactical macros are based on information about the target language. They work on its abstract syntax tree (AST).

The main advantage of lexical macros is their flexibility as they can be used to generate arbitrary software artifacts in any language but to the price of possibly syntactically invalid programs. The generated programs can be checked at compile-time at the earliest.

Because syntactical macros are executed in the context of the target language, they are safer to use but are limited to a set of available target language constructs.

As we will show in our implementation (Section 4), the flexibility of lexical macros is a problem for program analysis because dependencies between parts of the generated system are created in an uncontrolled way. This problem does not occur with syntactical macros.

Similar to the differentiation of lexical and syntactical macros, we consider the functionality of the macro language. We focus on legacy macro-based generators used to generate Cobol and similar programming languages. These generators differ from well-known macro-based generators such as the C preprocessor Cpp, which is still used in existing and new software systems. Cpp is mostly used to combine different input source files (#include) or to exclude certain parts of the source code (i.e. conditional compilation with #define/#ifdef).

Similar to Cpp, macro languages of legacy generators can be used to compose a target system from small fragments of target code. But on top of that, legacy generators provide complex macro language constructs, which are either evaluated during generation (lexical macros) or translated to constructs in the target language (syntactical macros). For example, a macro language provides a generic language to describe file access which is translated to different code depending on the target language and target platform. Users may prefer the macro language because of its syntax and the ability to generate different target languages from the same source. In the examples we use in our evaluation (Section 6), we see about half of statements are in the macro language the other half being statements of the embedded language Cobol.

2.2 Program Slicing

A great variety of approaches for program slicing, each with different properties, were developed in the last three decades [21]. Use cases of slicing include debugging, software maintenance and program comprehension [21].

The slicing process consists of two steps: selecting the slicing criterion and performing a slicing operation.

The slicing criterion is a crucial fact of a program analysis. For instance, in debugging, the slicing criterion may be the statement where an error is shown. In program comprehension, one might select an interesting function as the slicing criterion. The slicing criterion may consist of multiple statements, input and output parameters or whole procedures.

The slicing operation is performed with respect to the slicing criterion. From this slicing criterion the program is reduced, so the resulting program contains only a subset of statements satisfying a certain condition depending on the type of program slice.

There are different types of program slices, mainly forward and backward slices. A forward slice will result in a program with the slicing criterion and all statements which, in some way, are dependent on this criterion. For example, a forward slice on the initialization of a variable will include statements reading this variable. Similarly, a backward slice will result in a program only with statements having an influence on the statement selected in the slicing criterion. If we slice with respect to a variable read, the result will include the initialization of the variable.

Several algorithms for program slicing exist. In our work we use the slicing algorithm of Aung [5] because it allows both forward and backward slices. Additionally, it allows inter-procedural, executable program slicing and creates procedure specializations. We now explain those three important properties.

The algorithm works inter-procedural, i.e. it respects dependencies across calls of functions like passed local and global parameters. Recursive calls are possible as well. This property is important to analyze macros connected by macro calls.

Backward slices are executable. The result of a backward slice is a valid, executable program. In most cases the resulting program will be much smaller than the input program. We can use this to run the generator on the sliced macro and analyze the results on the generated system. Results of forward slicing are not necessarily executable but can be easily sanitized in an extra processing step restoring, for example, needed procedure calls.

The slicing algorithm creates specializations of procedures. If the input program contains a procedure f(int,int) the result may contain two specializations f1(int) and f2(int) instead. This way, unneeded parameters of sliced procedures are removed the same way as unneeded statements are removed.

As a special case of forward slicing it is possible to perform a feature removal [5]. In this case, a feature will be represented in the generator by a set of statements which implement a certain functionality. These statements of the feature implementation are used as the slicing criterion. A sliced program is created which excludes all elements of the feature and its consequences.

We now introduce an example to illustrate the process of backward slicing. Consider the program in Figure 1. It contains two procedures, with Main being the entry point. We set the statement print(g1) in the first procedure as the slicing criterion. The example contains different types of...
of dependencies. Each statement has a control dependency on the corresponding procedure. A call dependency transfers control from $Main$ to $Sub$. A parameter in dependency connects the actual in $(1$ and $2)$ to the formal in $(i$ and $j)$. The two global variables $g1$ and $g2$ are passed to the function in the same way. The formal out of $Sub$ $(g1$ and $g2)$ is connected to corresponding actual out of $Main$. From these actual out there is a data flow to the print statements.

The result of a backward slice with respect to the slicing criterion is shown in Figure 2. The slicing criterion consists of the print$(g1)$ statement, which requires the $Main$ procedure from which it is control dependent and the value of the global variable g1. The variable g1 is changed in the procedure $Sub$. Therefore, this procedure and its call from $Main$ is included in the slice. The procedure $Sub$ is specialized so it only contains statements which are relevant to the slicing criterion. In the example, the assignment of $g2$ is irrelevant and therefore removed.

The background on macro-based generators and on program slicing will later be used to introduce a special program slicing algorithm for macro-based generator programs.

3. RELATED WORK

Our work is related to work discussing the analysis of macro-based generators and of work discussing program slicing. We will discuss both streams in turn.

There are numerous approaches to analyze Cpp, as the most common macro language [7, 8, 16, 17, 18]. These approaches concentrate on the software variability implemented using the conditional compilation mechanism of Cpp. The focus remains on the target languages like C/C++ but uses some information from the macros. For example, the liveness of a variable in the target language should be checked for all valid variants of the software [17]. While this is an analysis on the target language, it needs to consider the macro language which implements the software variability as well to determine which variants are possible and how they may influence the liveness of the variable.

In our work, we focus on the macro language of legacy generators which are more complex than Cpp. These macro languages contain features commonly found in general purpose languages and special features for program generation like embedded domain-specific languages (DSL). These additional features require other analysis approaches than currently used in the analysis of Cpp.

Program slicing has been done for a long time with different goals [6]. Some approaches already use slicing on macro-based generators. Slicing of Cpp macros is done by Vidác et al. [20]. They too use program slicing to support the understanding of the relation between macro statements and C/C++ statements. However, they solely concentrate on Cpp and do not consider more complex macro languages as we do in our work. They are limited to basic forward and backward slicing.

We extend this approach by respecting more complex macro languages allowing e.g. for calls of macros and discuss how slicing can support program comprehension of generators. We also use executable slicing and slicing with procedure specializations to allow for further analyzes of the slicing results.

Kamischke et al. [15] use slicing on models like state machines to reduce the complexity of models of software systems with variability. While we share their motivation of using slicing, we base our approach on slicing of source code as the only available software artifact of legacy macro-based generators.

4. SLICING OF MACRO LANGUAGES

We base our implementation of the slicing algorithm on the descriptions of Aung et al. [5] already introduced in Section 2.2. The algorithm is language independent as it is working on a System Dependence Graph, SDG [14], an abstract representation of a program. The SDG consists of nodes representing statements, parameters and procedure entrances. The edges of the graph basically represent control and data flow with special edges for call dependencies and parameter flow.

The quality of a slicing result depends on the quality of detected dependencies modeled in an SDG. Missing dependencies in the SDG will lead to an incomplete slicing result. Wrong dependencies can lead to errors in the slicing result.

As long as we consider macro languages as normal programming languages, we could directly apply the program slicing algorithm. However, the main characteristic of macro languages is the mix of statements of the macro language and statements of the embedded target language. These two languages constitute two levels of possible dependencies.

The first level, in our case the macro language level, is handled by traditional program slicing. Dependencies on the macro level comprise elements defined and used within the macro language. Because we are able to parse the whole generator, including all its macros, we can correctly detect all these dependencies.

To correctly slice generators, we need to consider the second, i.e. the target language level (e.g. Cobol) as well. We introduce target-level dependencies as an extension of slicing in the context of macro-based generators. To detect these edges, we have to understand how the target system is generated and how the target system itself is working.

For target-level dependencies we have to differentiate syntactical and lexical parts of a macro depending of their origin.

Syntactical elements are given as special DSLs within the macro language. Programs encoded in these DSLs are not evaluated during generation, rather, they are translated into code in the target language. We extend our parser to handle these DSLs within the macro language.

<table>
<thead>
<tr>
<th>Main</th>
<th>SubProc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 int g1</td>
<td>1 .FILE MYFILE</td>
</tr>
<tr>
<td>2 print(g1)</td>
<td>2 .READ MYFILE</td>
</tr>
<tr>
<td>3 Call Sub(1)</td>
<td>3 .ADD SUBPROC</td>
</tr>
</tbody>
</table>

Figure 3: Macro inducing target-level dependencies

Consider the example in Figure 3. It consists of two procedures, the first one, $Main$, calling the other one. The $File$ command, part of a $File$ $DSL$, is creating code for file ac-
The command `Read`, part of the `File DSL` too, in the other procedure generates code to read a file with a given name. The `data dependency` in the generated code is obvious, yet it is not part of the macro language. The parameter of `Read` is just a string. The existence of the referenced file is not checked at run-time of the generator. A missing reference can only be detected by compiling or executing the generated system. Still, we include a `data dependency` edge from `File` to `Read`. This way, a backward slice of `Read` will include `File` as expected.

The SDG of the code from Figure 3 can be seen in Figure 4. For both procedures, (Main and Sub), there is a corresponding entry vertex. The remaining vertices represent the two statements `File` and `Add` from the Main procedure and `Read` from Sub. Each edge is annotated with the type of dependency it represents.

Given a complete documentation of the macro language, it is possible to consider all possible target-level dependencies induced by the macro DSLs. Another source of target-level dependencies are `lexical elements` of the macro language. These elements are used to compose existing fragments of target language statements into a target system.

The fragments of the target language are not `checked` by the generator framework whether they constitute valid source code. We are currently unable to reliably detect such dependencies and will ignore dependencies induced in this way. Our slicing result will be `incomplete` with respect to this type of dependency.

For example, at one point the macro may generate the initialization of a variable \( \mathbf{A} \) and at another point generates the use of a variable \( \mathbf{A} \). On the macro level, there is no data flow between these two elements. The dependency only manifests itself in the generated system. The detection of such dependencies requires comprehensive information of all variants of the generated system. A backward slice from the second statement generating the use of the variable will `not include` the statement defining the variable. This shows that our result is incomplete.

Irrespective of the source of the `target-level dependency`, we can respect them during program slicing. We extend the SDG to consider dependencies within the macro as well as in the generated system. Target-level dependencies are modeled in the SDG as edges, the same way as control or data flow dependencies are handled. The algorithm is left unchanged as it treats all control and data flow dependencies the same way. This way, the slicing results correctly consider dependencies on both levels.

We first create the SDG based on the source code of our macro-based generator. For this, we parse the macros and identify all dependencies (control and data flow as well as calls between macros and target-level dependencies). This step is `language dependent`, thus has to be implemented for every addressed macro-language. The cost of this step depends on the complexity of the input language and the quality of the documentation. We discuss the details of the language used in our examples in Section 6.1.

The rest of the algorithm is based on the SDG and thus `language independent`. From the SDG a Pushdown System (PDS), a concept from the model checking community [13], is constructed representing valid paths in the program. A set of standard automaton operations is performed using the existing libraries WALi [12] and OpenFst [4]. The result of the operations is a set of nodes and edges from which a program’s source code can be reconstructed.

5. USE CASES

We are interested in understanding legacy macro-based generators to modernize their implementation in the end. We have identified four main use cases in which program slicing can support understanding of legacy macro-based generators. Our main focus is information about connections of feature implementations and parameters controlling the generation process. We analyze the dependencies of a feature in Section 5.1 and analyze how a feature is affecting other features in Section 5.2. The identification of the influence of a parameter on a generator will be done in Section 5.3.

In a final use case, we show a method to split macros with respect to certain aspects thus reducing their complexity (Section 5.4).

5.1 Feature Dependencies

To support understanding of feature dependencies, we first select the implementation of a feature as the slicing criterion. The identification of a feature has to be done by some independent method [11].

A `backward slice` is performed resulting in a smaller but still executable program. The reduction of complexity is our main approach to support program understanding. As the result is still a valid program all other methods of program analysis or program understanding may be applied on the slicing result. Executable program slicing therefore can be easily integrated with other analysis steps.

The result of the `backward slice` can be interpreted as follows. Implementations of other features within the slice are `required features` of the selected feature. Parameters within the slice form the part of the `configuration` influencing the feature. Such parameters can modify how a feature is implemented or if a feature is included at all.

Further slicing operations are necessary to identify the dependencies between required features. Two features `share` a requirement, if they both have the requirement in their backward slice but one feature is not dependent on the other one. In the same way, i.e. by comparing different slices, `mutual` or `directed` dependencies can be identified. A `mutual` dependency happens if two features require each other. A `directed` dependency happens if a feature is dependent on another feature. This includes types of dependencies like control and data dependency.

A backward slice is not limited to the `condition` of a feature, which is the focus of variability analysis of Cpp [17]. The slice will not just contain presence information of a parameter but also domain-specific operations on it like setting default parameters or checking feature combinations.

![Figure 4: SDG of example showing target-level dependencies](image-url)
5.2 Influence of a Feature

Assuming we know which statements are used to implement a feature, we can use slicing to determine the influence of this feature. A feature has both an influence on the level of the macro language and on the generated system.

The influence on the macro level is determined by performing a forward slice. This slice will include all statements, which includes both macro and target language statements, influenced by this feature. We can use this information to e.g. derive depending features.

The influence of a feature on the generated system can be determined using the feature removal function of the slicing algorithm. The slicing criterion is created based on the statements implementing the feature. Then, two variants of the generator can be created: with and without the feature. Both variants are executable so they can be used to generate two variants of the target system. The delta of the generated code shows influence of the feature on the generated code.

During feature removal all statements of the features and all parts of the macro which are influenced by the feature are removed. This is necessary to create an executable and valid program.

If the influence of the feature is very large, many parts of the generated system will be removed along with the feature. To separate between direct and indirect influences we propose a stepwise removal of features, starting with features without much influence on other features.

To determine the order of feature removal, we start with any feature. If the nodes to be removed by feature removal of this feature contains another feature, we select one of these features and repeat the step. If a selected feature has no more other features in its influence, its influence on the target system is determined using the delta method explained above. The process continues with the remaining generator and ends, if no more features are in the generator. The result of this process is a fine-grained mapping of macro features to generated source code.

5.3 Influence of a Parameter

We can apply forward slicing to determine the influence of a parameter. We have already introduced forward slicing to analyze the dependencies of features. The same approach can be used to learn about the influence of a single parameter on the generator.

Parameters of the generator are the main mechanism to control variability. The forward slice will include all statements which read the parameter or values which are derived from the parameter.

The forward slicing with respect to a single parameter can be interpreted as follows. A slicing result containing no more than the slicing criterion, which is always in the result, means the parameter has no influence and thus can be removed from the generator or at least be ignored in the analysis. Statements of the target language in the slicing result show that the parameter has an influence on the generated result. If the result only includes macro statements, the parameter has no influence on the generated system. Nevertheless, the parameter might be useful e.g. by being used for side effects like debug statements.

The result is a static data flow analysis which could also be acquired by other means. Still, the algorithm for forward slicing is based on the same prerequisites like the SDG as the other slicing operation. Implementing it requires only little additional effort.

Figure 5 shows a small example of a program with two parameters var1 and var2. We perform a forward slice with var2 (i.e. the second line) as the slicing criterion. In the same figure on the right side, we show the result of the forward slice. As always, the slicing criterion is included in the result. The if statement with a condition on the parameter is included because there is a data flow from the slicing criterion to the statement. The statement is directly dependent on the parameter. The result includes the output statement. It is directly dependent on the if statement already in the result. Therefore the output statement is indirectly dependent on the parameter (i.e. the slicing criterion).

We can easily see that the condition of the if statement is never true and the included statement therefore unreachable. Program slicing does not evaluate any conditions and is therefore unable to consider reachability. This requires further analysis which we will discuss at the end of this paper (Section 7).

5.4 Procedure Specializations

A backward slice of a procedure includes all statements which are relevant for one of the elements in the slicing criterion. There are cases in which a procedure performs multiple aspects. A normal backward slice would create a procedure with implementations of all relevant aspects. Procedure specialization now can be used to improve this behavior by creating different procedures each containing only the aspects needed by the current usage context. The usage context comprises the information at which point the procedure is called.

For each call of a procedure the set of needed elements from the procedure is determined. Each set is now mapped to a new procedure. The call in the program is replaced by a call of this new procedure. So, n calls of a single procedure may lead to up to n different procedures in the slicing result.

Figures 6 and 7 show a simple example of the effect of procedure specialization. The source program (Figure 6) consists of two procedures, Main and Sub. A backward slice is performed with both print statements as slicing criterion. As the global variable global3 is not used by the two statements, its initialization in Sub can be removed. Both print
5.1 Setup

We perform our evaluation on the legacy generator tool ADS by DSTG. Currently, ADS is still used for applications, most of them in the banking and insurance sector. ADS is a good case for our approach because its language is much more complex than C++. ADS is used to generate Cobol and PL/I. The macro language used by ADS contains all elements of a generic procedural programming language like control structures, loops and the ability to call other procedures (i.e., other macros). ADS combines properties of lexical and syntactical macros.

Parsing of the ADS language is the main challenge. The language contains many DSLs which lead to large and complex grammar files. Language features of ADS are white space sensitive. While this is a typical property of legacy programming languages it is uncommon in modern languages and requires more effort of configuration of modern parser generators. The language allows late binding, which is a problem for all static analyzes.

On the other side, there are advantages of performing analysis on a legacy language. There are no pointers, exceptions or other advanced language features which hinder the construction of a control flow.

Due to space constraints, we select the two use cases influence of a feature and procedure specializations for our evaluation.

6.1 Setup

The steps for program slicing in our setup are shown in Figure 8.

First, we create an XML representation of the AST using the parser generator TXL [10]. The required grammar definition for parsing is not specific for program slicing and can be reused for other analyzes.

Using a custom tool implemented in C# we traverse the generated XML files and create an SDG. To build the SDG, we need to identify control and data flow within the macros as well as call dependencies between the macros.

The macro language in ADS contains most elements from general purpose languages. Typical language elements like conditions, loops, variable assignments, function calls are used to create the control and data flow. Late binding of procedure calls, i.e., the macro to be called, is determined during runtime which is a problem for correct static construction of the SDG. This is a known problem of static slicing [21]. Our examples make little use of this feature, so this could be resolved manually. In future releases, we plan to perform a prior analysis of the possible macros to be called.

We implemented a set of rules to detect target-level dependencies. As we explained above, we are unable to detect all target-level dependencies in a complete and correct way. In our setup, we focus on dependencies induced directly by the macro language. Such dependencies are induced by parts of the macro language which are transformed into the target language (syntactical macro elements).

We detect these cases by traversing the AST. For each target-level dependency we add an edge in the SDG. Fortunately edges representing target-level dependencies can be handled the same way as control or data dependencies. The slicing algorithm can be applied without a change.

Table 1 shows the sizes (count of nodes and edges) of the SDG for the generators of our evaluation. In all cases, the size of the SDG will be finite [5]. Currently we store the SDG in-memory using two hash sets for nodes and edges. For persistent storage, we implemented simple serialization
to csv files as well as storing the SDG in the graph database Neo4j.

From this SDG the states and transition of an automaton are derived. The automaton operations are implemented in C++ and use the libraries WALi and OpenFst. From the resulting automaton, a new SDG is constructed. The resulting SDG could be serialized to source code, a feature currently not implemented and thus performed manually.

6.2 Influence of a Feature

We use a generator with the name DEP-ASUB as our example. This generator creates a small Cobol application used for training sessions. The application can import data from a text file, can store it into a database, can display the current data set and can print a report. It is controlled by a terminal-like user interface.

The user macros comprise about 400 lines of code. Additionally, many system macros are used which we will not consider. The generated system consists of about 1700 lines of Cobol code.

We manually identified seven features. The dependency between these features is determined based on forward slices with respect to the corresponding features as the slicing criterion. The resulting features and their dependencies are shown as a feature diagram in Figure 9.

We will use feature removal to isolate the effects of each feature. The effect of the features is simply measured in lines of macro code and lines of generated Cobol code.

We start with the full generator containing all macros. Then, we remove features one by one, with an order such that no removed feature is required by another remaining feature. After each removal, we get an executable generator from which we generate a Cobol application and count its lines of code. The resulting lines of macro code and lines of Cobol after each feature removal can be seen in Figure 10.

Figure 10 also shows the resulting order of removed features. Starting with the complete generator, we remove the feature Start, then Output. The last feature to be removed is Menu. Many features depend on Menu, as it is used to call other features. We could also slice within the feature Menu using a single option as the slicing criterion. In this case, the menu options would be reduced and the corresponding features removed.

The remaining lines in the macro and the generated system are the program structure and other boilerplate code. This way, we can isolate each feature in the macro and the generator and explain them by an average of 250 lines of Cobol code and 50 lines of macro code. The effect would be much larger, if we consider all the helper methods.

6.3 Procedure Specializations

We show the effect of procedure specialization during slicing using the handling of default parameters. ADS provides many library functions including one to create code for file handling. There are 56 default parameters for this library. A simple system macro is setting all those default parameters. As with all default parameters, a user may override them or use the given default value.

In our generator DEP-ASUB we find six file definition using the library. Each file description is overriding some of the default parameters. Depending on how the file is used, different parameters are overridden. We create a backward slice with the operations to create the files as the slicing criterion.

Each call of the procedure setting the defaults is replaced by one of two specializations. The first specialization contains 6 parameter definitions, the second 8. Although the number of specializations is not limited by the slicing algorithm we commonly see only a few specializations per procedure. This observation conforms to the known results of slicing C programs [5].

As a result, 48 of 56 parameters are unused and can be excluded in the analysis of the program. This smaller program will be easier to understand.

Half of the files are created based on one specialization, the other half using the other one. All definitions of the first procedure are included in the second. So in three cases there are two additional parameters overridden compared to the other case. This way, we introduce redundancy in the code as the common six parameters are now defined in two procedures. This could be resolved by applying refactoring in order to reuse the definitions of the common parameters.

6.4 Performance

While still in development, we are already very optimistic concerning performance and scalability of our implementation. Most steps in the process are not required to be repeated for every slicing operation. For example, the creation of the XML and construction of the SDG is a one time operation and the result can be reused for any slicing.

1www.neo4j.org
operation. Using the largest generator in our evaluation, DEP-ASUB, we are able to perform a slice with a small slicing criterion within 2 seconds and a large slicing criterion (more than 1000 nodes) in 12 seconds. These values are achieved on a modern notebook and with none of the caching opportunities or even compiler code optimization activated.

7. CONCLUSIONS

In this paper we applied an existing algorithm for program slicing to macro-based generators. Generators are a challenge for program slicing because of the mixture of macro language and target language. We showed how to extend a slicing algorithm of Aung to analyze macro languages respecting both languages. Though some limitations remain.

We explained four use cases to support understanding of macro-based generators that can be implemented by different types of program slicing. In our evaluation we successfully applied to use cases to real-world generators.

Future works is twofold. First, we plan to extend the implementation and second, we plan further evaluations. To increase the usefulness of the slicing results we want to implement a path-sensitive program slicing. This feature is useful to e.g. see the consequence of concrete parameter values. In a more comprehensive evaluation, we want to apply our technique to larger systems to show their values. In an implementation and second, we plan further evaluations. To increase the usefulness of the slicing results we want to implement a path-sensitive program slicing. This feature is useful to e.g. see the consequence of concrete parameter values.

Finally, as understanding a software system is a human task, we need to perform an experiment showing empirically the usefulness of program slicing in program understanding of generators.

8. ACKNOWLEDGMENTS

This work is supported by the German Federal Ministry of Education and Research under grant number 01IS12043B.

We thank our industry partner Delta Software Technology GmbH for providing the necessary examples for our evaluation.

9. REFERENCES