SemSlice: Exploiting Relational Verification for Automatic Program Slicing

Bernhard Beckert, Thorsten Bormer, Stephan Gocht, Mihai Herda,
Daniel Lentzsch, and Mattias Ulbrich

Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
{beckert,bormer,herda,ulbrich}@kit.edu,
stephan.gocht@student.kit.edu
d.lentzsch@web.de

Abstract. We present SemSlice, a tool which automatically produces very precise slices for C routines. Slicing is the process of removing statements from a program such that defined aspects of its behavior are retained. For producing precise slices, i.e., slices that are close to the minimal number of statements, the program's semantics must be considered. SemSlice is based on automatic relational regression verification, which SemSlice uses to select valid slices from a set of candidate slices. We present several approaches for producing candidates for precise slices. Evaluation shows that regression verification (based on coupling invariant inference) is a powerful tool for semantics-aware slicing: precise slices for typical slicing challenges can be found automatically and fast.

1 Introduction

Program slicing. Program slicing [18] removes statements from a program in order to reduce its size and complexity while retaining some specified aspects of its behavior. Slicing techniques (or similar data dependency analyses) are used to optimize the result of compilers. Slicing is also a powerful tool for challenges in software engineering such as code comprehension, debugging, and fault localization, where the user is involved [3]. A recent study [6] shows that slicing can improve programming skills in novice learners.

The idea behind SemSlice. Traditional slicing techniques use an overapproximation of dependencies in a program and thus produce imprecise, non-minimal slices. SemSlice goes beyond purely syntactical dependency analysis and takes the semantics of statements and expressions into account. It can thus produce much more precise slices than related approaches. SemSlice is fully automatic and does not require auxiliary annotations (like loop invariants etc.). SemSlice finds slices by applying *regression verification* [7], an approach for proving relational properties of programs, to check whether generated slice candidates are valid slices, i.e., are equivalent to the original program with respect to the specified slicing criterion. Thus, the process has two steps: (1) The tool generates a slice candidate, i.e., a sub-program of the original program that is not

necessarily a valid slice for the given criterion. (2) The tool uses the existing but customized automatic relational verification engine to check whether the candidate is equivalent to the original program with respect to the criterion. This process is repeated and combined with syntactical slicing to iteratively refine obtained slices.

Handling loops. Precise slices are particularly difficult to obtain for programs with loops. SemSlice provides a higher degree of automation compared to existing semantics-aware slicing frameworks [1, 11] which are based on inferred or user-provided *functional* loop invariants. In the context of slicing, functional loop invariants have disadvantages: In order to prove that at the location of the slicing criterion the relevant program variables have the same value, strong loop invariants are needed (they have to fix unique values for the variables). Moreover, a loop invariant for the original program needs not be a valid invariant for the sliced program; a different second invariant may be needed. SemSlice does not employ functional invariants but operates on *relational coupling loop invariants* which formalize the difference between two program variants and thus escape this dilemma.

Contribution. SemSlice demonstrates the feasibility of using relational verification to compute precise slices. The advantage of this approach is that the candidate generation engine does not need to care about the correctness of the candidates – that is handled by the relational verifier. In addition to the three heuristics for generating candidates described and evaluated in this paper, SemSlice can easily be extended with others. Thus, SemSlice provides a platform for relational verification based slicing for the software slicing community.

Structure of this paper. The remainder of this paper is structured as follows. Section 2 introduces the concepts of program slicing and relational verification. The implementation of SemSlice is described in Section 3. The different approaches to generate slice candidates are introduced in Section 4. The paper is completed by a short evaluation in Section 5, a report on related tools in Section 6 and a conclusion in Section 7.

2 Background

2.1 Static Backward Slicing

SemSlice performs a variant of *static backward slicing* (as introduced by Weiser [18]), in which the *slicing criterion*—the specification of the behavioral aspects that must be retained—comprises a set of program variables and a location within the program. Statements which have no effect (a) on the value of the specified program variables at the specified point and (b) on how often the point is reached may be removed.

Formally, a *slice candidate* is a variant of the original program where zero or more statements have been replaced with the side-effect-free `skip` statement. A

```
1  int f(int h, int N){
2    int i = 0;
3    int x = 0;
4    while(i < N) {
5      if(i < N - 1)
6        x = h;
7      else
8        x = 42;
9      i++;
10   }
11   return x;
12 }
```

```
1  int f(int h, int N){
2    int i = 0;
3    int x = 0;
4    while(i < N) {
5      if(i < N - 1)
6        skip;
7      else
8        x = 42;
9      i++;
10   }
11   return x;
12 }
```

```
1  int f(int h, int N){
2    int i = 0;
3    int x = 0;
4    while(i < N) {
5      if(i < N - 1)
6        skip;
7      else
8        skip;
9      i++;
10   }
11   return x;
12 }
```

Figure 1: (a) Original program, (b) Slice with respect to variable x at line 8, (c) Incorrect slice candidate

slice candidate is considered a *valid slice* if, given the same input to the slice candidate and original program, the following two conditions hold:

1. During execution of the slice candidate and the original program, respectively, the location specified in the slicing criterion is reached for the same number of times.
2. When the location is reached for the ith time in the original program and for the ith time in the slice ($i \geq 1$), each variable specified in the slicing criterion has the same value in the original program's state and in the slice's state.

Figure 1 shows an example of static backward slicing. The goal is to slice the C routine in Figure 1a with respect to a slicing criterion, which requires the value of x at the `return` statement in line 11 to be preserved. A valid slice for this criterion is shown in Figure 1b: The assignment in line 6 has been taken from the program. Instead of removing it, we have replaced it by a synthetic `skip` statement without effects to keep the program structure similar to the input program. This line has no effect on the value of x after the loop as x is always set to 42 in the last loop iteration. To show that this program is a slice of the original, an overapproximating syntactical analysis (ignorant of the meaning of statements) is insufficient. A semantic analysis is required to determine that the last loop iteration always executes the else-branch. The slicing procedure needs to reason about loops and path conditions.

2.2 Relational Program Verification

Relational verification is an approach to prove specified relations between two given programs (or variants of the same program) to be valid. The tool that SemSlice relies on for relational verification is the automatic regression verification tool LLRêve [12], which takes two programs as input. If it terminates, the tool has either proved that the programs behave equivalently or it comes up with a counterexample input showing that the programs' semantics are different.

LLRêve operates on a generalized version of product programs [2] in which two programs are combined into one in order to be able to reason simultaneously

about corresponding loops in the programs. Thus, the program behavior needs not be fully encoded into functional loop invariants, but only the relation between the two behaviors. Relying on relational coupling invariants allows the verification engine to automatically infer the needed abstractions (loop invariants and function summaries) in more cases than if functional abstractions are used. As an example, for proving that the candidate shown in Fig. 1b is a correct slice with respect to the given slicing criterion, our approach has inferred the following coupling invariant:

$$\begin{aligned}&((N_1 - N_2 + i_2 - i_1 = 0) \wedge (N_1 - i_1 \geq 1))\\ \vee&((N_1 - N_2 + i_2 - i_1 = 0) \wedge (x_1 = x_2))\end{aligned}$$

Since our analysis considers two programs, the variable names occuring in the coupling invariant are annotated with 1 or 2 depending on which program they belong to. Note that the coupling invariant states that the value of x is the same in both programs (at the location specified by the slicing criterion).

3 Implementation

SemSlice[1] combines the slicing candidate generation with the existing tools clang, LLRêve, and Eldarica. Its workflow is shown in Figure 2. We implemented the candidate generation component and adapted LLRêve for checking slices candidates. The other components are used "of the shelf". A user-friendly web interface for SemSlice can be accessed at `formal.iti.kit.edu/slicing`.

Clang and LLVM. Clang[2] is a front end of the LLVM compiler infrastructure [14] and is used to compile C code into the LLVM intermediate representation (IR). The slicing process operates on the IR, and the resulting slice is also returned in IR. While a reverse transformation into C is possible in principle, it is currently not supported by the LLVM framework and adding this feature would require a significant effort. Building on top of LLVM reduces language complexity and allows us to use the API of LLVM which provides methods to modify the IR (e.g., to remove statements) and standard code analyses (like loop detection).

Candidate generation. A duplicate of the original program is modified by removing statements to generate a slice candidate. The choice which statements are removed depends on the chosen candidate selection method (see Section 4). The process can be iterated; then the results of previously checked slice candidates are taken into consideration for generating new ones.

LLRêve and Eldarica. SemSlice encodes the slicing criterion as a relational specification between the slice candidate and the original program in first order logic. From this input, LLRêve generates an SMT formula, more precisely a set of

[1] The SemSlice source code is available at `github.com/mattulbrich/llreve/tree/slicing`.

[2] `http://clang.llvm.org`

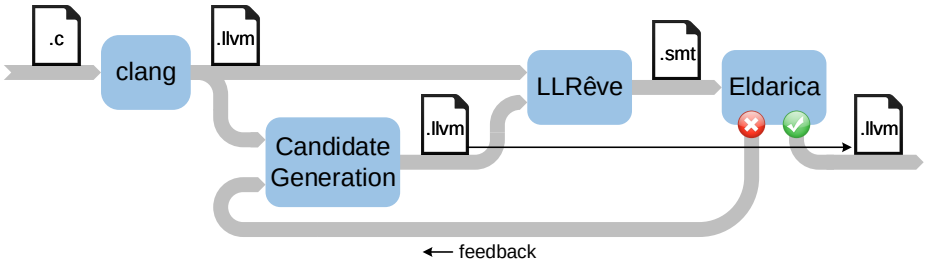

Figure 2: SemSlice Architecture for finding a valid slice

constrained horn clauses, in which the coupling invariants become uninterpreted predicate symbols to be inferred by the solver. Eldarica [10], a state-of-the-art SMT solver, checks the formula for satisfiability, i.e., for existence of sufficiently strong relational invariants. If the formula is satisfiable, a valid slice has been found. It may be refined further or taken as the final result. Otherwise, Eldarica may provide a counterexample, i.e., an input under which the criterion is evaluated differently in the slice candidate and the original program.

Implementation aspects. LLRêve can only check relational specifications in the final states of the two programs. A slicing criterion, however, may refer to any point within the program. We have adapted the clause construction in LLRêve to allow for relational conditions to be checked within functions.

Since a valid slice requires by definition that the criterion's statement is reached equally often in program and slice, SemSlice enforces that all loops are iterated equally often by encoding this requirement into the proof obligation. This requirement is a little too strict: for a loop not containing the criterion, there might be a correct slice which iterates the loop less often, but cannot be validated with our technique. This requirement enforces the mutual termination property (i.e. the slice terminates iff the original program also terminates) of the original program and the slice candidate. If it were to be relaxed than our approach would be able to validate more candidates, however, the mutual termination property would have to be shown through other means.

4 Slice Candidate Generation

SemSlice provides three methods for candidate generation. They differ in time requirements, number of generated candidates, and precision.

The naive *brute forcing* (BF) approach generates all possible slice candidates. It is complete in the sense that it finds the smallest slice that can be validated with relational verification. Section 5 shows that it runs surprisingly fast on small programs, but due to the exponential number of slice candidates it does not scale.

Single statement elimination (SSE) successively removes statements from the program. If removing a statement yields an invalid slice, SSE reverts and tries removing another statement. This results in a quadratic number of LLRêve

Table 1: Evaluation

Original			BF			SSE			CGS		
Example	Source	#stmts	time (s)	#stmts	#calls	time (s)	#stmts	#calls	time (s)	#stmts	#calls
count_occurrence_error	self	50				13	42	11			
count_occurrence_result	self	50				16	44	13			
dead_code_after_ssa	[17]	4	<1	2	4	<1	2	4	<1	2	1
dead_code_unused_variable	self	3	<1	2	2	<1	2	3	<1	2	1
identity_not_modifying	[8]	8	<1	3	3	<1	7	5	<1	6	1
identity_plus_minus_50	[1]	5	<1	2	4	<1	5	4	<1	5	1
iflow_cyclic	[17]	18	62	14	2197	<1	16	6	<1	17	1
iflow_dynfamic_override	self	15	23	8	1298	<1	11	8	<1	12	1
iflow_endofloop (Figure 1)	self	19	118	15	4065	<1	16	7	<1	18	2
intermediate	self	13	4	11	129	<1	12	5	<1	12	2
requires_path_sensitivity	[11]	20	647	16	26894	<1	17	10	<1	18	3
single_pass_removal	self	13	<1	3	7	<1	6	11	<1	8	1
unchanged_over_itteration	self	20	29	9	932	1	15	14	<1	20	2
unreachable_code_nested	self	10	<1	2	1	<1	9	1	<1	4	1
whole_loop_removable	self	20	15	8	469	<1	17	5	<1	17	2

invocations. Thus, it scales better than brute forcing, but cannot remove groups of statements that cannot be removed individually like `x:=x+50; x:=x-50`.

Counterexample guided slicing (CGS) works in the opposite direction: It successively adds statements to a candidate until it can be proved valid. In case of an invalid slice candidate, CGS uses the counterexample provided by Eldarica to choose which statements are to be added in the next iteration. In each iteration the slice candidates grow by at least one statement such that termination is guaranteed. On the considered examples, CGS terminates very fast after only a few iterations, but with potentially reduced precision compared to the other methods.

5 Evaluation

Table 1 shows an evaluation of SemSlice using a collection[3] of small but intricate examples (e.g., the example of Figure 1 or a routine in which the same value is first added and then subtracted) that each focus on a particular challenge for semantics-aware slicing. Some are taken from slicing literature [1, 4, 8, 11, 17] while others were crafted by ourselves. The second column indicates the source of each example, the third the number of LLVM-IR statements in the program. For each candidate generation method from Section 2.1, the table lists the number of statements in the smallest slice found by SemSlice, the (wall) time needed by the tool, and the number of calls to the SMT solver. The experiments were conducted on a machine with an Intel Core I5-6600K CPU and 16GB RAM. The exponential brute forcing approach works satisfactorily fast on functions with up to 20 statements, and while it requires more time than the other approaches, it computes more precise slices.

[3] The benchmarks are available at `github.com/mattulbrich/llreve/tree/slicing/slicing/testdata/benchmarks`

Modern coding conventions [15, pg. 34] suggest that functions should comprise at most some 20 lines of code, and our approach is capable to deal with challenges of that size. What hinders us slicing real-world programs is that SemSlice cannot yet deal with bit-operations, complicated heap structures, and deep calling hierarchies.

6 Related Work

Static slicing is an active research area, other semantic approaches have been published using abstract interpretation [9, 16] or term rewriting [8, 13]. In this section we focus on approaches with accessible tools.

GamaSlicer [5] features a graphical user interface and is designed for slicing Java programs annotated with JML. It provides multiple slicing algorithms, the most sophisticated one is *assertion based slicing* [1], which uses the specification as slicing criterion. A valid slice is obtained by removing statements from the original program such that the original specification still holds. Unlike SemSlice, this tool requires functional loop invariants from the user.

Tracer is a tool that runs in command line and computes slices for C programs based on *path sensitive backward slicing* [11]. It uses symbolic execution to find and remove unfeasible paths in a program and thereby increase precision compared to syntactic slicing. To cope with path explosion of the symbolic execution tree, parts of the tree are reused. This approach scales very well, but unfeasible data dependencies over multiple iterations of a loop like those in Figure 1a cannot be detected.

7 Conclusion

We presented SemSlice, a fully automatic tool to compute slices using semantic information. The approach uses relational verification to show that a slice candidate is equivalent to the original program with respect to a slicing criterion. Three different approaches to compute slice candidates were introduced.

The presented approach works well on small, but intricate programs. To be able to treat larger programs with SemSlice, we will in future work combine the regression verification engine with other better scaling, but less precise regression verification tools.

As our approach builds on top of a relational verification engine that infers relational coupling invariants, functional loop invariants need not be specified. Our evaluation shows powerful and highly precise program slicing can be implemented by relying on relational verification. That indicates that relational verification is indeed a very useful basis for building formal program analysis tools.

References

[1] Barros, J.B., Da Cruz, D., Henriques, P.R., Pinto, J.S.: Assertion-based slicing and slice graphs. Formal Aspects of Computing 24(2), 217–248 (2012)

[2] Barthe, G., Crespo, J.M., Kunz, C.: Relational verification using product programs. In: Butler, M., Schulte, W. (eds.) Proceedings, International Symposium on Formal Methods. Lecture Notes in Computer Science, vol. 6664, pp. 200–214. Springer (2011)
[3] Binkley, D., Harman, M.: A survey of empirical results on program slicing. Advances in Computers 62, 105–178 (2004)
[4] Canfora, G., Cimitile, A., De Lucia, A.: Conditioned program slicing. Information and Software Technology 40(11), 595–607 (1998)
[5] da Cruz, D., Henriques, P.R., Pinto, J.S.: Gamaslicer: an online laboratory for program verification and analysis. In: Proceedings of the Tenth Workshop on Language Descriptions, Tools and Applications. p. 3. ACM (2010)
[6] Eranki, K.L., Moudgalya, K.M.: Program slicing technique: A novel approach to improve programming skills in novice learners. In: Proceedings of the Conference on Information Technology Education. pp. 160–165. ACM (2016)
[7] Felsing, D., Grebing, S., Klebanov, V., Rümmer, P., Ulbrich, M.: Automating regression verification. In: Proceedings of the International Conference on Automated Software Engineering. pp. 349–360. ASE '14, ACM (2014)
[8] Field, J., Ramalingam, G., Tip, F.: Parametric program slicing. In: Proceedings of the Symposium on Principles of Programming Languages. pp. 379–392. ACM (1995)
[9] Halder, R., Cortesi, A.: Abstract program slicing on dependence condition graphs. Science of Computer Programming 78(9), 1240–1263 (2013)
[10] Hojjat, H., Konečný, F., Garnier, F., Iosif, R., Kuncak, V., Rümmer, P.: A Verification Toolkit for Numerical Transition Systems, pp. 247–251. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
[11] Jaffar, J., Murali, V., Navas, J.A., Santosa, A.E.: Path-sensitive backward slicing. In: Proceedings of the 19th International Conference on Static Analysis. pp. 231–247. Springer (2012)
[12] Kiefer, M., Klebanov, V., Ulbrich, M.: Relational program reasoning using compiler IR. In: Blazy, S., Chechik, M. (eds.) Verified Software. Theories, Tools, and Experiments, Revised Selected Papers. Lecture Notes in Computer Science, vol. 9971, pp. 149–165. Springer (Nov 2016)
[13] Komondoor, R.: Precise slicing in imperative programs via term-rewriting and abstract interpretation. In: Static Analysis, pp. 259–282. Springer (2013)
[14] Lattner, C., Adve, V.: Llvm: A compilation framework for lifelong program analysis & transformation. In: Code Generation and Optimization, 2004. CGO 2004. International Symposium on. pp. 75–86. IEEE (2004)
[15] Martin, R.C.: Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1 edn. (2008)
[16] Mastroeni, I., Nikolić, Đ.: Abstract program slicing: From theory towards an implementation. In: International Conference on Formal Engineering Methods, ICFEM 2010. Proceedings, pp. 452–467. Springer (2010)
[17] Ward, M.: Properties of slicing definitions. In: Ninth IEEE International Working Conference on Source Code Analysis and Manipulation. SCAM'09. pp. 23–32. IEEE (2009)
[18] Weiser, M.: Program slicing. In: Proceedings of the 5th International Conference on Software Engineering. pp. 439–449. IEEE Press (1981)