Computing Super Reduced Program Slices by Composing Slicing Techniques

David Insa & Sergio Pérez & Josep Silva
Departamento de Sistemas Informáticos y Computación (DSIC)
Camino de Vera s/n 46022
Valencia, España
{dinsa,serperu,jsilva}@dsic.upv.es

ABSTRACT

Program slicing is a technique to extract the part of a program (the slice) that influences or is influenced by a set of variables at a given point. Computing minimal slices is undecidable in the general case, and obtaining the minimal slice of a given program is computationally prohibitive even for very small programs. Hence, no matter what program slicer we use, in general, we cannot be sure that our slices are minimal. In this work, we present a method to automatically produce a new notion of slice that we call super reduced slice because it is constructed with the combination of different slicing techniques, including the composition of standard program slicers.

CCS Concepts

•**Theory of computation** → *Program constructs;* •**Software and its engineering** → *Software development techniques;*

Keywords

Program Analisys; Program Slicing; Erlang

1. INTRODUCTION

Program slicing is a technique for program analysis and transformation whose main objective is to extract from a program those statements (the *slice*) that influence or are influenced by the values of one or more variables at some point of interest, often called *slicing criterion* [13, 12, 1, 9]. This technique has been adapted to practically all programming languages, and it has many applications such as debugging [3], program specialization [8], software maintenance [5], code obfuscation [7], etc.

In the general case, determining the minimal slice is undecidable [13]. For this reason, almost all program slicing techniques guarantee that their computed slices are complete (i.e., they contain all statements that do influence the slicing criterion), but, in general, they are not precise.

Reducing the size of the computed slices would speed up many software processes. For instance, compilers use program slicing to remove dead code, and many analyses use program slicing as a preprocessing stage to detect variable dependencies. Hence, making slicing accurate would also improve the later analyses based on it. In this work, we describe a process to produce super reduced slices. Even though our implementation has been designed for Erlang, the proposed ideas and the described methodology are directly applicable to any other language.

2. SUPER REDUCED PROGRAM SLICES

In this section, we combine several ideas and techniques so that they successively refine a slice in a complementary way. The combination takes into account the cost of the different techniques, and it minimizes the amount of work done by those techniques that are more expensive. We also describe how we have instantiated this methodology for Erlang in our implementation. We call the slices produced by our technique *super reduced slices* (SR-slices) because none of the techniques used to produce them can further reduce them by themselves. The methodology follows the schema shown in Figure 1.

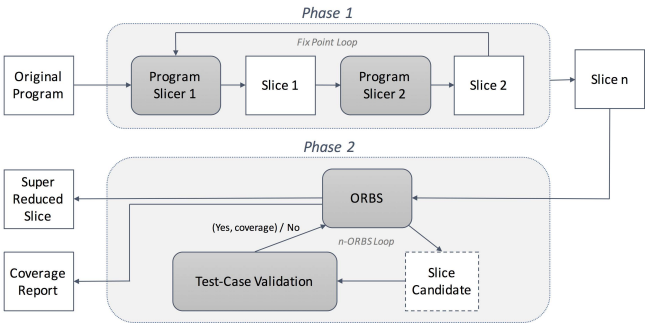

Figure 1: Combination of techniques to produce super reduced program slices.

Given a program (in the figure *Original Program*), we compute its SR-slice (in the figure *Super Reduced Slice*) following two independent phases. Each phase has been enclosed inside a box, and it implements different slicing processes (enclosed inside dark grey boxes) that produce different slice

refinements (enclosed inside white squares). In particular, the output of the first phase is a slice of the original program, which is the input of the second phase. All processes are fully automatic. Observe that this is a general scheme that can be adapted to any other language. We only need to instantiate the dark grey components: two program slicers and a test-case generator (the observation-based slicing (ORBS) [2] technique is already paradigm-independent and works for any language).

• **Phase 1:** In the first phase, we propose the use of a set of slicers to repeatedly slice the original program until a fixpoint is reached. Different program slicers usually implement different techniques and optimizations to further reduce the size of a slice. Hence, we can use an already existing program slicer to produce a first slice that we can use as the starting point to further reduce its size with another program slicer. Thus, given a program P, $slice_A(slice_B(P))$ could take advantage of the combined power of slicer A and slicer B.

In our Erlang implementation, we use two slicers. First, we slice the original program with one slicer, and then we slice the slice with another slicer. The produced slice is sliced again with the first slicer, and so on until the slice cannot be further reduced. The first slicer is *Slicerl* [11], and the second slicer is *e-Knife*, a new static slicer for Erlang implemented by us from scratch.

One important property of Slicerl and e-Knife is that the slices produced by both of them are complete. Hence, the output of phase 1 is always a complete slice, because the sequential composition of complete slicers produces a complete slicer.

In our setting, the fixpoint of phase 1 was reached in only one iteration (the slice produced by e-Knife (*Slice 2*) could not be further reduced by Slicerl). The whole phase was fully automatic.

• **Phase 2:** It is formed by two main modules: ORBS and test case generation and validation. The former is fully automatic, the latter can be also automatic depending on the selected test case generator.

Firstly, we implemented a variant of ORBS that iterated over the Erlang Dependence Graph (EDG) [10] of *Slice n*. The EDG is an adaptation of the System Dependence Graph (SDG) [6] for Erlang. Roughly, this variant iteratively removes from the EDG each single node together with its subtree. Each removal of subtrees produces a *Slice Candidate*. For each valid (i.e. compilable) slice candidate, test cases are generated to compare the behaviour of the original program and the candidate (the EDG without the removed subtree). If they show the same behaviour, then that subtree is removed. Otherwise, it is kept. The iterative process is incremental and continues until no more nodes can be removed.

This adaptation of ORBS to EDGs includes two design decisions that are worth to notice: (i) at each step, we try to remove the selected node and its subtree. (ii) It works top-down. This is more efficient because first it tries to remove entire functions, clauses, and data structures before trying with their components.

The second module used in this phase is in charge of the test case generation and validation. In particular, every valid slice candidate produced by ORBS is intensively tested by comparing its behaviour with the one of the original program. If they show the same behaviour, then the tentatively removed subtree is actually removed. Otherwise, it is kept. Clearly, the quality of this phase depends on the quality of the test case generator used. In our particular implementation, we combine white-box testing (we use the Erlang's concolic testing tool CutEr [4]) with black-box testing (we use a random test case generator).

3. EMPIRICAL EVALUATION

Table 1 summarizes the empirical evaluation of our particular implementation of the proposed methodology. Concretely, it compares the size of the successive refinements of all the slices, and the time needed by all the processes of both phases. Each line represents a different benchmark. We selected 20 benchmarks from public repositories. All of them are publicly available at:

http://www.dsic.upv.es/~jsilva/slicing/bencher/

For each benchmark, column `Nodes` represents its number of EDG nodes, which corresponds to the size of the programs/slices. In the case of the slices, we also include the percentage of nodes that remain in the slice with respect to the original program. Note that some of the benchmarks cannot be treated by Slicerl. Column `Time` shows the time expended in each phase measured in seconds (s). Finally, in phase 2, column `iter / time` uses the notation `X (Y)`, where `X` is the number of different iterations performed by the algorithm (i.e., the number of configurations that were checked, where each configuration is the result of removing 1 node (and its subtree) from the EDG), and `Y` is the total time used to check the `X` configurations. Note that the value of `Y` is significantly increased when phase 2 further reduces the slice. This happens because the testing phase, which is exhaustive and costly (e.g., it generates test cases to prove that the node cannot be removed), is only activated when a node can be removed from the slice without producing compilation errors.

4. CONCLUSIONS

This work presents a technique to produce super reduced slices. The technique includes the use of several tools, including program slicers, white box and black box test case generators, coverage tools, etc. that are combined in such a way that they increase their synergy and their performance. We have also implemented several tools including a new program slicer called e-Knife.

In the process of designing the new technique, we had to define new algorithms to further reduce the size of our slices. In particular, we have adapted the ORBS technique to Erlang, where we use the EDG instead of the PDG, and where we work with expressions instead of lines.

We have instantiated the proposed technique to Erlang. The empirical evaluation of the technique produced interesting results. In particular, we have shown that two Erlang slicers, Slicerl and e-Knife, are complementary and should be combined if the size of the slice is critical.

It is also interesting to remark that our technique is fully automatic, and thus, it can be used itself as a very precise

	Original	Phase 1				Phase 2	
		Slicerl		e-knife			
	Nodes	Time (s)	Nodes	Time (s)	Nodes	Iter. / Time	Nodes
B1	28	0,953	22 (78,57%)	2,121	22 (78,57%)	18 (1 s.)	22 (78,57%)
B2	40	0,828	32 (80,00%)	2,154	32 (80,00%)	28 (2 s.)	32 (80,00%)
B3	53	0,819	33 (62,26%)	2,123	33 (62,26%)	117 (19013 s.)	24 (45,28%)
B4	87	-	-	2,193	74 (85,06%)	193 (16816 s.)	63 (72,41%)
B5	54	0,991	51 (94,44%)	2,171	51 (94,44%)	42 (3 s.)	51 (94,44%)
B6	133	1,02	65 (48,87%)	2,184	60 (45,11%)	229 (34026 s.)	40 (30,08%)
B7	133	1,011	47 (35,34%)	2,158	47 (35,34%)	126 (24126 s.)	27 (20,30%)
B8	128	0,869	95 (74,22%)	2,221	95 (74,22%)	253 (42755 s.)	34 (26,56%)
B9	128	0,877	91 (71,09%)	2,319	91 (71,09%)	82 (19448 s.)	38 (29,69%)
B10	80	0,948	75 (93,75%)	2,225	73 (91,25%)	256 (149356 s.)	58 (72,50%)
B11	59	0,812	57 (96,61%)	2,183	54 (91,53%)	112 (153 s.)	52 (88,14%)
B12	225	-	-	2,394	195 (86,67%)	657 (16994 s.)	183 (81,33%)
B13	359	-	-	2,781	333 (92,76%)	1088 (536981 s.)	229 (63,79%)
B14	454	-	-	2,52	118 (25,99%)	138 (34498 s.)	117 (25,77%)
B15	454	-	-	2,182	94 (20,70%)	83 (7 s.)	94 (20,70%)
B16	66	0,935	46 (69,70%)	2,148	46 (69,70%)	41 (3 s.)	46 (69,70%)
B17	209	0,933	75 (35,89%)	2,181	75 (35,89%)	214 (34248 s.)	55 (26,32%)
B18	209	0,96	84 (40,19%)	2,199	64 (30,62%)	137 (11344 s.)	50 (23,92%)
B19	209	0,864	117 (55,98%)	2,234	97 (46,41%)	127 (12843 s.)	38 (18,18%)
B20	820	-	-	3,214	647 (78,90%)	10124 (2441 s.)	234 (28,54%)

Table 1: Empirical evaluation of the obtained slices

slicer, because it takes a program, and produces a SR-slice. In this respect, it is important to highlight that the slices produced after the first phase are complete, but the slices produced after the second phase are based on testing and, thus, completeness is not ensured for them. However, completeness is ensured for the set of test cases considered.

5. ACKNOWLEDGEMENT

This work has been partially supported by the EU (FEDER) and the Spanish *Ministerio de Economía y Competitividad* under grant TIN2013-44742-C4-1-R and TIN2016-76843-C4-1-R, and by the *Generalitat Valenciana* under grant PROMETEO-II/2015/013 (SmartLogic). Sergio Pérez was partially supported by the Spanish *Iniciativa de Empleo Juvenil* Programme (MINECO) and the European Social Fund in collaboration with the *Sistema Nacional de Garantía Juvenil* under the grant PEJ-2014-A-24709 (*Promoción de Empleo Joven e Implantación de la Garantía Juvenil 2014*, MINECO).

6. REFERENCES

[1] D. Binkley and K. B. Gallagher. Program Slicing. *Advances in Computers*, 43(2):1–50, apr 1996.

[2] D. Binkley, N. Gold, M. Harman, S. Islam, J. Krinke, and S. Yoo. ORBS: Language-independent Program Slicing. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 109–120, New York, NY, USA, 2014. ACM.

[3] R. A. DeMillo, H. Pan, and E. H. Spafford. Critical slicing for software fault localization. *SIGSOFT Softw. Eng. Notes*, 21(3):121–134, may 1996.

[4] A. Giantsios, N. Papaspyrou, and K. Sagonas. Concolic testing for functional languages. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming (PPDP '15)*, pages 137–148. ACM, 2015.

[5] A. Hajnal and I. Forgács. A demand-driven approach to slicing legacy COBOL systems. *Journal of Software Maintenance*, 24(1):67–82, 2012.

[6] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions Programming Languages and Systems*, 12(1):26–60, 1990.

[7] A. Majumdar, S. J. Drape, and C. D. Thomborson. Slicing obfuscations: Design, correctness, and evaluation. In *Proceedings of the 2007 ACM Workshop on Digital Rights Management*, DRM '07, pages 70–81, New York, NY, USA, 2007. ACM.

[8] C. Ochoa, J. Silva, and G. Vidal. Lightweight program specialization via Dynamic Slicing. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Curry and Functional Logic Programming*, WCFLP '05, pages 1–7, New York, NY, USA, 2005. ACM.

[9] J. Silva. A vocabulary of Program Slicing-based techniques. *ACM Computing Surveys*, 44(3), 2012.

[10] J. Silva, S. Tamarit, and C. Tomás. System Dependence Graphs in sequential Erlang. In *Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering (FASE 2012)*, volume 7212 of *Lecture Notes in Computer Science*, pages 486–500. Springer, 2012.

[11] J. Silva, S. Tamarit, C. Tomás, and D. Insa. Slicerl, September 2011.

[12] F. Tip. A survey of Program Slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

[13] M. Weiser. Program Slicing. In *Proceedings of the 5th international conference on Software engineering (ICSE '81)*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.