Tukra: An Abstract Program Slicing Tool

Raju Halder and Agostino Cortesi
DAIS, Università Ca' Foscari Venezia, Venezia, Italy

Keywords: Program Slicing Tool, Dependence Graph, Abstract Interpretation.

Abstract: We introduce **Tukra**, a tool that allows the practical evaluation of abstract program slicing algorithms. The tool exploits the notions of statement relevancy, semantic data dependences and conditional dependences. The combination of these three notions allows **Tukra** to refine traditional syntax-based program dependence graphs, generating more accurate slices. We provide the architecture of the tool, some snapshots describing how it works, and some preliminary experimental results giving evidence of the accuracy improvements it supports.

1 INTRODUCTION

Program slicing emerged a useful technique that extracts from programs the statements which are relevant to a given behavior. It is a fundamental operation for addressing many software-engineering problems, *e.g.* program understanding, debugging, maintenance, testing, parallelization, integration, etc. The original static slicing algorithm by Mark Weiser (Weiser, 1984) is expressed as a sequence of data-flow analysis problems and the influence of predicates on statement execution, while Korel and Lasky (Korel and Laski, 1988) extended it to the dynamic context and proposed an iterative dynamic slicing algorithm based on dynamic data flow and control influence. Over the last 3 decades, several works on program slicing have been proposed based on the dependence graph representation (Agrawal and Horgan, 1990; Horwitz et al., 1990; Sarkar, 1991; Sinha et al., 1999).

Program slicing can be defined in concrete as well as in an abstract domain, where in the former case we consider exact values of the program variables, while in the latter case we consider some properties instead of their exact values. The notion of Abstract Program Slicing was first introduced by Hong, Lee and Sokolsky (Seok Hong et al., 2005). Mastroeni and Nicolić (Mastroeni and Nikolic, 2010) recently extended the theoretical framework of slicing proposed by Binkley (Binkley et al., 2006) to an abstract domain in order to define abstract slicing, and to represent and compare different forms of slicing in the abstract domain. Other remarkable works on abstract program slicing include (Cortesi and Halder, 2010; Mastroeni and Zanardini, 2008; Zanardini, 2008; Bhattacharya, 2011).

In (Cortesi and Halder, 2010), the authors applied the notion of semantic relevancy of statements, and proposed a slicing refinement for imperative programs by combining with statement relevancy the notions of semantic data dependences (Mastroeni and Zanardini, 2008) and conditional dependences (Sukumaran et al., 2010). The combination of these three notions allows us to refine traditional syntax-based dependence graphs into more precise semantics-based dependence graphs, leading to an abstract program slicing algorithm that produces more accurate abstract slices.

In this paper, we introduce **Tukra**[1]., a tool based on the theoretical proposal in (Cortesi and Halder, 2010) that allows the practical evaluation of abstract program slicing algorithms. The tool performs static intraprocedural slicing of a program in an abstract domain of interest. We provide the architecture of the tool, some snapshots describing how it works and some preliminary experimental results giving evidence of the accuracy improvements it supports. As far as we know, there is no other similar public tool available.

The rest of the paper is organized as follows: Section 2 presents a short introduction of the existing abstract program slicing algorithm that supports **Tukra**. Section 3 describes **Tukra**'s architecture, some snapshots describing how it works and an experimental re-

[1]"**Tukra**" is a hindi word which means "Slice". The source code of **Tukra** can be downloaded from *www.dsi.unive.it/∼avp/Tukra/*

Halder R. and Cortesi A..
Tukra: An Abstract Program Slicing Tool.
DOI: 10.5220/0004069801780183
In *Proceedings of the 7th International Conference on Software Paradigm Trends* (ICSOFT-2012), pages 178-183
ISBN: 978-989-8565-19-8

sult on a test program. Section 4 concludes the work.

2 BACKGROUND

The algorithm behind our slicing tool is proposed in (Cortesi and Halder, 2010) that combines the notions of statement relevancy, semantic data dependences and conditional dependences. Let us discuss these three notions and the slicing algorithm in brief.

In traditional Program Dependence Graphs (PDGs), the notion of dependences between statements is based on the syntactic presence of a variable in the definition of another variable or in a conditional expression. Therefore, the definition of slices at semantic level creates a gap between slicing and dependences. Mastroeni and Zanardini (Mastroeni and Zanardini, 2008) first introduced the notion of semantic data dependences which fills up the existing gap between syntax and semantics. The semantic data dependences which are computed for all expressions in the program over the states possibly reaching the associated program points, help in obtaining more precise semantics-based PDGs by removing some false dependences from the traditional syntactic PDGs. For instance, although the expression "$e = x^2 + 4w \bmod 2 + z$" syntactically depends on w, but semantically there is no dependence as the evaluation of "$4w \bmod 2$" is always zero. This can also be lifted to an abstract setting where dependences are computed with respect to some specific properties of interest rather than the concrete values. For instance, if we consider the abstract domain SIGN=$\{\top, pos, neg, \bot\}$, the expression e does not semantically depend on x *w.r.t*. SIGN, as the abstract evaluation of x^2 always yields to pos for all atomic values of $x \in \{pos, neg\}$. This is the basis to design abstract semantics-based slicing algorithms aimed at identifying the part of the programs which is relevant with respect to a property (not necessarily the exact values) of the variables at a given program point.

Sukumaran et al. (Sukumaran et al., 2010) presented a refinement of the traditional PDGs into Dependence Condition Graphs (DCGs) based on the notion of conditional dependences. A DCG is built from the PDG by annotating each edge $e = e.src \rightarrow e.tgt$ in the PDG with information $e^b = \langle e^R, e^A \rangle$ that captures the conditions under which the dependence represented by that edge is manifest. The first component e^R refers to *Reach Sequences*, whereas the second component e^A refers to *Avoid Sequences*. The informal interpretation of e^R is that the conditions represented by it should be true for an execution to ensure that $e.tgt$ is reached from $e.src$. The *Avoid Sequences* e^A captures the possible conditions under which the assignment at $e.src$ can get over-written before it reaches $e.tgt$. The interpretation of e^A is that the conditions represented by it must not hold in an execution to ensure that the variable being assigned at $e.src$ is used at $e.tgt$. It is worthwhile to note that e^A is relevant only for DDG edges and it is $\emptyset$ for CDG edges.

Cortesi and Halder (Cortesi and Halder, 2010) applied the notion of semantic relevancy of statements. It determines whether an imperative statement is relevant *w.r.t*. a property of interest, and is computed over all concrete (or abstract) states possibly reaching the statement. For instance, consider the following code fragment: $\{(1)\, x = input;\ (2)\, x = x+2;\ (3)\ print\ x;\}$. If we consider an abstract domain of parity represented by PAR=$\{\top, odd, even, \bot\}$, we see that the variable x at program point 1 may have any parity from the set $\{odd, even\}$, and the execution of the statement at program point 2 does not change the parity of x at all. Therefore, the statement at 2 is semantically irrelevant *w.r.t*. PAR. By disregarding all the nodes that correspond to irrelevant statements *w.r.t*. concrete (or abstract) property from a syntax-based PDG, we obtain a more precise semantics-based (abstract) PDG.

The combined effort of semantic relevancy of statements with the expression-level semantic data dependences by Mastroeni and Zanardini (Mastroeni and Zanardini, 2008) guarantees a more precise semantics-based (abstract) PDG. A further refinement of it can be achieved by applying the notion of conditional dependences by Sukumaran et al. (Sukumaran et al., 2010) that allows us to transform PDGs into DCGs and to identify unrealizable dependneces in them under the trace semantics of the programs. The removal of such unrealizable dependences yields to more refined semantics-based (abstract) DCGs.

The slicing algorithm **GEN-SLICE** makes use of two auxiliary algorithms. Given a program P and an abstract domain ρ, the algorithm **REFINE-PDG** generates semantics-based abstract PDG *w.r.t*. ρ. The algorithm **REFINE-DCG** converts the PDG (which is refined by the algorithm **REFINE-PDG**) into a DCG by computing the annotation over all dependence edges, and then again refines it into more precise one by removing some unrealizable dependence paths. Finally, **GEN-SLICE** performs slicing based on the dependence graphs obtained this way.

3 Tukra

In this section, we identify the possible inputs expected from the users, and the basic functional modules required to perform the slicing computations over the abstract domains.

The aim of designing **Tukra** is to provide user-friendly interfaces accelerating users to slice imperative programs in various abstract domains of interest. When performing abstract slicing of programs, users must provide the following inputs to **Tukra**:

1. **Program to be Sliced: Tukra** is able to perform slicing of programs covering a subset of imperative language constructs. At this preliminary stage of implementation, we do not focus on any specific programming language. We consider the following assumptions on the syntax of the input programs as below:
   - All control blocks should be enclosed with "{}" irrespective of the number of statements in it.
   - Empty control block must have "skip;" statement in it.
   - Run-time input for any statement is denoted by "?", *e.g.* "*var* =?;".
   - The syntax of the statement displaying variables' values is "$print(x,y,z)$;" where x, y, z are the program variables.

   Observe that the assumptions above do not cause severe limitations for **Tukra**: an improvement of the tool to support other language constructs can easily be achieved.
2. **Abstract Domain of Interest:** We provide two abstract domains in **Tukra** at this preliminary stage of implementation. The first one is SIGN=$\langle \top, pos, neg, zero, \bot \rangle$ that represents the sign property of variables, and the second one is PAR=$\langle \top, odd, even, \bot \rangle$ that represents the parity property of the variables. However, additional abstract domains can be integrated by implementing the corresponding interfaces designed for the abstract domains.
3. **Types of Semantic Computations: Tukra** can perform three types of semantic computations: statement relevancy, semantic data dependences and conditional dependences in the abstract domain chosen before. Users are provided options to choose either single or combination of multiple types of semantic computations.
4. **Slicing Criterion:** A slicing criterion is composed of two components: a program variable v and a program point p. Observe that in **Tukra**, slicing is performed based on the dependence graphs. Therefore, v must be defined or used at p.

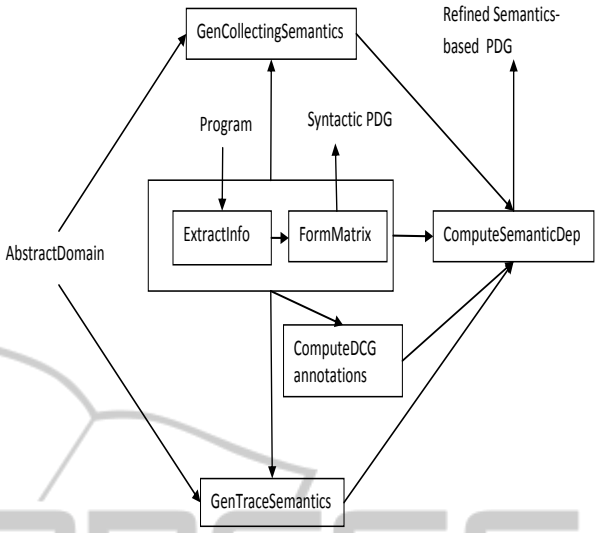

Figure 1: **Tukra**'s architecture.

3.1 Tukra's Architecture

The slicing tool consists of packages for building and refining dependence graphs, as well as a parser that translates the input program into an internal representations convenient for the future computations. We identify the following key modules for **Tukra**:

1. **ExtractInfo:** The module "ExtractInfo" extracts detail information about the input programs, *i.e.* the type of program statements, the controlling statements, the defined variables, the used variables, etc for all statements in the program and store them in a file as an intermediate representation.
2. **FormMatrix:** The module "FormMatrix" generates incidence matrix for Control Flow Graphs (CFGs) and Program Dependence Graphs (PDGs) of the input programs based on the information extracted by the module "ExtractInfo".
3. **GenCollectingSemantics** and **GenTraceSemantics:** Given an abstract domain, these modules compute the abstract collecting Semantics and abstract trace semantics of the input programs based on the information extracted by "ExtractInfo" and the CFG generated by "FormMatrix".
4. **ComputeDCGannotations:** This module computes the DCG annotations (Reach Sequences and Avoid Sequences) for all CDG and DDG edges of the PDG based on the information extracted by

"ExtractInfo" and the PDG generated by "FormMatrix".

5. **ComputeSemanticDep:** It computes statements relevancy, semantic data dependences and conditional dependences of the programs under all possible states reaching each program points of the program (collecting semantics) and by computing the satisfiability of DCG annotations under its abstract trace semantics.

The interaction between various modules mentioned above is depicted in Figure 1. Observe that "FormMatrix" generates a syntactic PDG based on which we can perform syntax-driven slicing *w.r.t*. a criterion, whereas "ComputeSemanticDep" refines the syntactic PDG into a semantics-based abstract PDG by computing statement relevancy and semantic data dependences under its abstract collecting semantics, and the conditional dependences based on the satisfiability of DCG annotations under its abstract trace semantics.

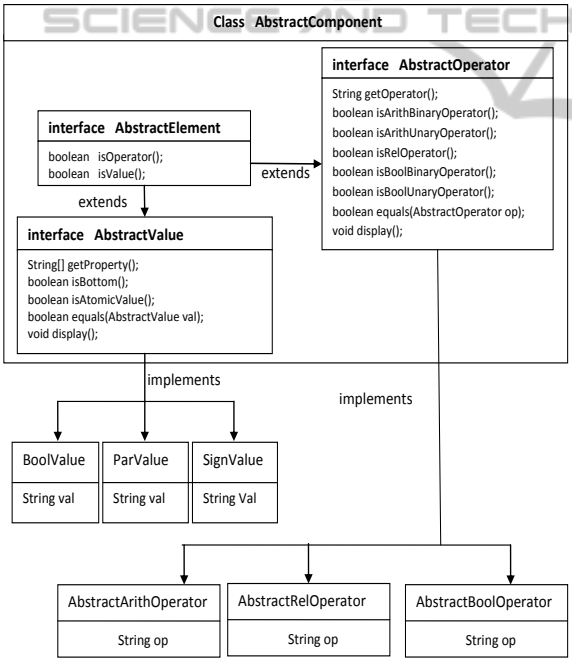

Figure 2: Designing abstract values and operators.

The design of abstract values, abstract operators, abstract domains and abstract environments are depicted in Figures 2, 3 and 4 respectively. Any abstract value such as sign value, parity value, boolean value, and any abstract operator such as arithmetic, relational, boolean operator implement the specifications represented by the interfaces "AbstractElement", "AbstractValue" and "AbstractOperator". The abstract domain implements the specifications represented by the interface "AbstractDomain". The abstract states at each program point in a program is defined by an abstract environment associated with the corresponding program point. The abstract environment is defined by the class "AbstractEnvironment". This design allows us to add any new abstract domain to **Tukra** by implementing the interfaces corresponding to the new abstract domain.

```
interface AbstractDomain
AbstractValue getTop();
AbstractValue getBottom();
AbstractValue getLUB(AbstractValue[] list);
AbstractValue getGLB(AbstractValue[] list);
AbstractValue[] getAtoms(AbstractValue val);
AbstractValue[] getSubValues(AbstractValue val);
AbstractValue evaluate(AbstractArithOperator ArithUnaryOp, AbstractValue val);
AbstractValue evaluate(AbstractArithOperator ArithBinaryOp, AbstractValue val1, AbstractValue val2);
String evaluate(AbstractRelOperator RelOp, AbstractValue val1, AbstractValue val2);
AbstractValue evaluate(AbstractBoolOperator BoolUnaryOp, AbstractValue val);
AbstractValue evaluate(AbstractBoolOperator BoolBinaryOp, AbstractValue val1, AbstractValue val2);
void getAbstractExpressionList(String PostfixExpr, AbstractEnvironment absEnv, LinkedList ExprList);
```

Figure 3: Interface abstract domain.

```
class AbstractEnvironment
String[] vars;
AbstractValue[] env;
void setEnvironment(String[] ProgVars, AbstractValue[] val);
void setEnvironment(AbstractValue[] val);
String[] getVariables();
AbstractValue[] getEnvironment();
AbstractEnvironment getModifiedEnvironment(String Variable, AbstractValue newVal);
AbstractValue getVariableValue(String Variable);
AbstractEnvironment getRestrictedEnvironment(String[] ProgVars);
boolean equals(AbstractEnvironment obj);
void display();
AbstractEnvironment getLUB(AbstractEnvironment obj, AbstractDomain ADobj);
boolean isAtomicEnvironment();
LinkedList getAtomicCovering(AbstractDomain ADobj);
LinkedList getXAtomicCovering(String[] X_Vars, AbstractDomain ADobj);
LinkedList getXSubCovering(String[] X_Vars, AbstractDomain ADobj);
```

Figure 4: Class abstract environment.

Tukra is implemented in Java. In Figures 5 and 6, we show some of the snapshots of the system.

We executed **Tukra** on a PC running with 2.27GHz Processor, Windows 7 Professional 64-bit Operating System and 4 GB RAM. Table 1(a) depicts a test program. Tables 1(b), 1(c) and 1(d) depict syntax-based, semantic data dependence-based (Mastroeni-Zanadini's approach) and Cortesi-Halder's algorithm-based slicing of the test program *w.r.t*. $\langle$L11, *y*, SIGN$\rangle$.

Table 1: A test program and its various slicing.

(a) A test program.

L1	$i = -2;$
L2	$x = ?;$
L3	$y = ?;$
L4	$w = ?;$
L5	$if(x \geq 0)\{$
L6	$x = x + w;$
L7	$y = 4 * w * 0;\}$
L8	$while(i \leq 0)\{$
L9	$y = y * 2;$
L10	$i = i + 1;\}$
L11	$print(x, y);$

(b) Syntactic slicing *w.r.t.* $\langle$L11, $y\rangle$.

L1	$i = -2;$
L2	$x = ?;$
L3	$y = ?;$
L4	$w = ?;$
L5	$if(x \geq 0)\{$
L7	$y = 4 * w * 0;\}$
L8	$while(i \leq 0)\{$
L9	$y = y * 2;$
L10	$i = i + 1;\}$
L11	$print(x, y);$

(c) Slicing *w.r.t.* $\langle$L11, y, SIGN$\rangle$ (Mastroeni-Zanadini).

L1	$i = -2;$
L2	$x = ?;$
L3	$y = ?;$
L5	$if(x \geq 0)\{$
L7	$y = 4 * w * 0;\}$
L8	$while(i \leq 0)\{$
L9	$y = y * 2;$
L10	$i = i + 1;\}$
L11	$print(x, y);$

(d) Slicing *w.r.t.* $\langle$L11, y, SIGN$\rangle$ (Cortesi-Halder).

L1	$i = -2;$
L2	$x = ?;$
L3	$y = ?;$
L5	$if(x \geq 0)\{$
L7	$y = 4 * w * 0;\}$
L8	$while(i \leq 0)\{$
L10	$i = i + 1;\}$
L11	$print(x, y);$

4 CONCLUSIONS

The sound, efficient and effective theoretical support behind **Tukra** may make it more attractive to the practical field, as it is able to generate more precise slice w.r.t. the literature. At present, **Tukra** can be regarded as a first generation system, in that it is mainly developed to support research. It is now in a preliminary stage and there are a lot of scopes to improve it in terms of algorithmic efficiency and generality.

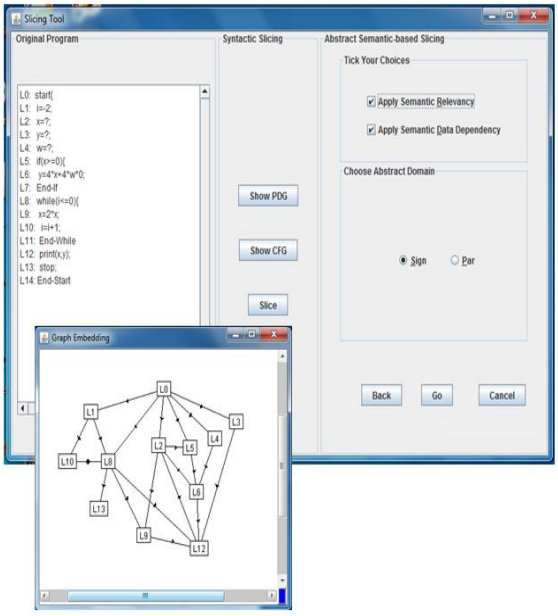

Figure 5: Syntactic slicing and PDG generation.

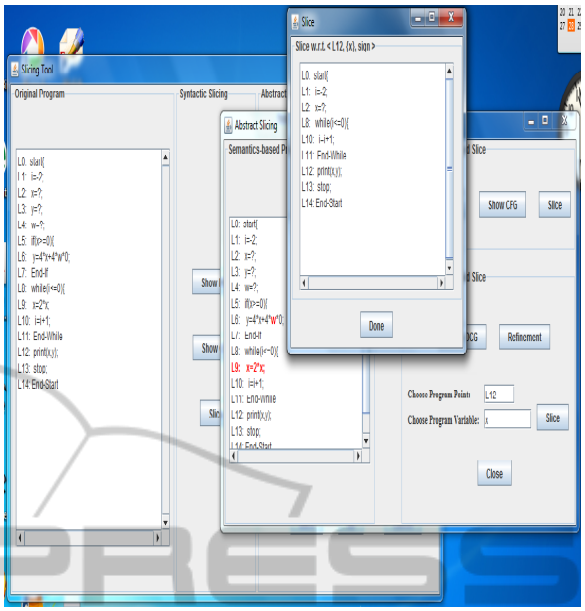

Figure 6: DCG-based slicing.

ACKNOWLEDGEMENTS

Work partially supported by RAS L.R. 7/2007 Project TESLA.

REFERENCES

Agrawal, H. and Horgan, J. R. (1990). Dynamic program slicing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '90)*, pages 246–256, White Plains, New York. ACM Press.

Bhattacharya, S. (2011). *Property Driven Program Slicing and Watermarking in the Abstract Interpretation Framework*. PhD thesis, Università Ca' Foscari Venezia.

Binkley, D., Danicic, S., Gyimóthy, T., Harman, M., Kiss, A., and Korel, B. (2006). A formalisation of the relationship between forms of program slicing. *Science of Computer Programming*, 62(3):228–252.

Cortesi, A. and Halder, R. (2010). Dependence condition graph for semantics-based abstract program slicing. In *Proceedings of the 10th International Workshop on*

Language Descriptions Tools and Applications (LDTA '10), pages 4:1–4:6, Paphos, Cyprus. ACM Press.

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

Korel, B. and Laski, J. (1988). Dynamic program slicing. *Information Processing Letters*, 29(3):155–163.

Mastroeni, I. and Nikolic, D. (2010). Abstract program slicing: From theory towards an implementation. In *Proceedings of the 12th International Conference on Formal Engineering Methods*, pages 452–467, China. Springer LNCS, Volume 6447.

Mastroeni, I. and Zanardini, D. (2008). Data dependencies and program slicing: from syntax to abstract semantics. In *Proceedings of the ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 125–134, San Francisco, California, USA. ACM Press.

Sarkar, V. (1991). Automatic partitioning of a program dependence graph into parallel tasks. *IBM Journal of Research and Development*, 35(5–6):779–804.

Seok Hong, H., Lee, I., and Sokolsky, O. (2005). Abstract slicing: A new approach to program slicing based on abstract interpretation and model checking. In *Proceedings of the 5th International Workshop on Source Code Analysis and Manipulation*, pages 25–34, Hungary. IEEE CS.

Sinha, S., Harrold, M. J., and Rothermel, G. (1999). System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In *Proceedings of the 21st International Conference on Software Engineering (ICSE '99)*, pages 432–441, Los Angeles, CA, USA. ACM Press.

Sukumaran, S., Sreenivas, A., and Metta, R. (2010). The dependence condition graph: Precise conditions for dependence between program points. *Computer Languages, Systems & Structures*, 36(1):96–121.

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

Zanardini, D. (2008). The semantics of abstract program slicing. In *Proceedings of the International Working Conference on Source Code Analysis and Manipulation*, pages 89–100, Beijing, China. IEEE Press.