Electronic Communications of the EASST
Volume X (2010)

Proceedings of the Fourth International Workshop on Foundations and Techniques for Open Source Software Certification (OpenCert 2010)

Safe Integration of Annotated Components into Open Source Projects

Sérgio Areias, Daniela da Cruz, Pedro Rangel Henriques and Jorge Sousa Pinto

16 pages

Guest Editors: Luis S. Barbosa, Antonio Cerone, Siraj A. Shaikh
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Safe Integration of Annotated Components into Open Source Projects

Sérgio Areias[1]*, Daniela da Cruz[2], Pedro Rangel Henriques[3] and Jorge Sousa Pinto[4]

[1] pg13381@alunos.uminho.pt
[2] danieladacruz@di.uminho.pt
[3] prh@di.uminho.pt
[3] jsp@di.uminho.pt
Departamento de Informática
Universidade do Minho

Abstract:

The decision to reuse software components raises a spectrum of issues, from requirements negotiation to product selection and integration. The decision of using existing software versus building from scratch custom software is one of the most complex and important of the entire development/integration process. The correct tradeoff is reached after having analyzed advantages and issues correlated to the reuse.

Despite the reuse failures in real cases, many efforts have been made to make this idea successful.

In this context of reuse software in open source projects, we address the problem of reusing annotated components as a rigorous way of assuring the quality of the application under construction. We introduce the concept of *caller-based slicing* as a way to certify that the integration of a component annotated with a contract into a system will preserve the correct behavior of the former, avoiding malfunctioning after integration.

To complement the efforts done and the benefits of slicing techniques, there is also a need to find an efficient way to visualize the annotated components and their slices. To take full profit of visualization, it is crucial to combine the visualization of the control/data flow with the textual representation of source code. To attain this objective, we extend the notion of System Dependence Graph and slicing criterion.

Keywords: Caller-based slicing, Annotated System Dependency Graph

1 Introduction

Reuse is a very simple an natural concept, however in practice is not so easy. According to the literature, selection of reusable components has proven to be a difficult task [1]. Sometimes this is due to the lack of maturity on supporting tools that should easily find a component on

* This author is sponsored by the Foo Society under Grant Nr. 42-23.

a repository or library [2]. Also, non experienced developers tend to reveal difficulties when describing the desired component in technical terms. Most of the times, this happens because they are not sure of what they want to find [2, 3]. Another barrier is concerned with reasoning about component similarities in order to select the one that best fits in the problem solution; usually this is an hard mental process [1].

Integration of reusable components has also proven to be a difficult task, since the process of understanding and adapting components is hard, even for experienced developers [1]. Another challenge to component reuse is to certify that the integration of such component in a open-source software system (OSS) keeps it correct. This is, to verify that the way the component is invoked will not lead to an incorrect behavior.

A strong demand for formal methods that help programmers to develop correct programs has been present in software engineering for some time now. The Design by Contract (DbC) approach to software development [4] facilitates modular verification and certified code reuse. The contract for a component (a procedure) can be regarded as a form of enriched software documentation that fully specifies the behavior of that component. So, a well-defined annotation can give us most of the information needed to integrate a reusable component in a OSS, as it contains crucial information about some constraints safely obtaining the correct behavior from the component.

In this context, we say that the annotations can be used to verify the validity of every component invocation; in that way, we can guarantee that a correct system will still be correct after the integration of that component. This is the motivation for our research: to find a way to help on the safety reuse of components.

This article introduces the concept of **caller-based slicing**, an algorithm that takes into account the calls of an annotated component to certify that it is being correctly used. To support the idea, we also introduce GamaPolarSlicer, a tool that implements such algorithm: to identify when an invocation is violating the component annotation; and to display a diagnostic or guidelines to correct it.

The remainder of paper is structured into 5 sections. Section 2 is devoted to basic concepts crucial to the understanding of the remaining of the paper: the notions of slicing and system dependency graph are introduced. Section 3 formalizes the definition of caller-based slicing. Section 4 defines the concept of annotated System Dependency Graph (SDGa). Section 5 illustrates the main idea through a concrete example. Section 6 gives a general overview of GamaPolarSlicer, introducing its architecture, functionalities and implementation details. Section 7 discusses related work on slicing programs with annotated components. Section 8 discusses related work on visualization of (sliced) programs. Then the paper is closed in Section 9.

2 Basic Concepts

In this section we introduce both the original concepts of slicing and system dependency graph.

2.1 Slicing

Since Weiser first proposed the notion of slicing in 1979 in his PhD thesis [5], hundreds of papers have been proposed in this area. Tens of variants have been studied, as well was algorithms to compute them. Different notions of slicing have different properties and different applications. These notions vary from Weiser's syntax-preserving static slicing to amorphous slicing which is not syntax-preserving; algorithms can be based on dataflow equations, information flow relations or dependence graphs.

Slicing was first developed to facilitate program debugging [6, 7, 8], but it is then found helpful in many aspects of the software development life cycle, including software testing [9, 10], software metrics [11, 12], software maintenance [13, 14], program comprehension [15, 16], component re-use [17, 18], program integration [19, 20] and so on.

Program slicing, in its original version, is a decomposition technique that extracts from a program the statements relevant to a particular computation. A program slice consists of the parts of a program that potentially affect the values computed at some point of interest referred to as a *slicing criterion*.

Definition 1 (Slicing Criterion) *A* static slicing criterion *of a program P consists of a pair $C = (p, V_s)$, where p is a statement in P and V_s is a subset of the variables in P.*

A slicing criterion $C = (p, V_s)$ determines a projection function which selects from any state trajectory only the ordered pairs starting with p and restricts the variable-to-value mapping function σ to only the variables in V_s.

Definition 2 (State Trajectory) *Let $C = (p, V_s)$ be a static slicing criterion of a program P and $T =< (p_1, \sigma_1), (p_2, \sigma_2), ..., (p_k, \sigma_k) >$ a state trajectory of P on input I. $\forall i, 1 \leq i \leq k$:*

$$Proj'_C(p_i, \sigma_i) = \begin{cases} \lambda & if\, p_i \neq p \\ < (p_i, \sigma_i | V_s) > & if\, p_i = p \end{cases}$$

where $\sigma_i | V_s$ is σ_i restricted to the domain V_s, and λ is the empty string.

The extension of $Proj'$ to the entire trajectory is defined as the concatenation of the result of the application of the function to the single pairs of the trajectory:

$$Proj_C(T) = Proj'_C(p_1, \sigma_1)...Proj'_C(p_k, \sigma_k)$$

A program slice is therefore defined behaviorally as any subset of a program which preserves a specified projections in its behavior.

Definition 3 (Static Slicing) *A* static slice *of a program P on a static slicing criterion $C = (p, V_s)$ is any syntactically correct and executable program P' that is obtained from P by deleting zero or more statements, and whenever P halts, on input I, with state trajectory T, then P' also halts, with the same input I, with the trajectory T', and $Proj_C(T) = Proj_C(T')$.*

Related work of slicing programs taking into account the annotations of a program will be referred in Section 7.

2.2 System Dependency Graph

The use of dependency graphs to visualize the data and control flow of a program has been widely accepted in the last years (Section 8).

Before exploring the use Dependency Graphs for visualization and comprehension, we present below the definitions of Procedure Dependency Graph and System Dependency Graph.

Definition 4 (Procedure Dependence Graph) *Given a procedure $\mathscr{P}$, a Procedure Dependence Graph, PDG, is a graph whose vertices are the individual statements and predicates (used in the control statements) that constitute the body of $\mathscr{P}$, and the edges represent control and data dependencies among the vertices.*

In the construction of the PDG, a special node, considered as a predicate, is added to the vertex set: it is called the *entry* node and is decorated with the procedure name.

A control dependence edge goes from a predicate node to a statement node if that predicate condition the execution of the statement. A data dependence edge goes from an assignment statement node to another node if the variable assigned at the source node is used (is referred to) in the target node.

Additionally to the natural vertices defined above, some extra assignment nodes are included in the PDG linked by control edges to the entry node: we include an assignment node for each formal input parameter, another one for each formal output parameter, and another one for each returned value — these nodes are connect to all the other by data edges as stated above. Moreover, we proceed in a similar way for each call node; in that case we add assignment nodes, linked by control edges to the call node, for each actual input/output parameter (representing the value passing process associated with a procedure call) and also a node to receiving the returned values.

Definition 5 (System Dependence Graph) *A System Dependence Graph, SDG, is a collection of Procedure Dependence Graphs, PDGs, (one for the main program, and one for each component procedure) connected together by two kind of edges: control-flow edges that represent the dependence between the caller and the callee (an edge goes from the call statement into the entry node of the called procedure); and data-flow edges that represent parameter passing and return values, connecting $actual_{in,out}$ parameter assignment nodes with $formal_{in,out}$ parameter assignment nodes.*

3 Caller-based slicing

In this section, we introduce our slicing algorithm. We start by extending the notion of static slicing and slicing criterion to cope with the contract of a program.

Definition 6 (Annotated Slicing Criterion) *An* annotated slicing criterion *of a program $\mathscr{P}$ consists of a triple $C_a = (a, S_i, V_s)$, where a is an annotation of $\mathscr{P}_a$ (the annotated callee), S_i correspond to the statement of $\mathscr{P}$ calling $\mathscr{P}_a$ and V_s is a subset of the variables in $\mathscr{P}$ (the caller), that are the actual parameters used in the call and constrained by α or δ.*

Definition 7 (Caller-based slicing) *A* caller-based slice *of a program $\mathscr{P}$ on an annotated slicing criterion $C_a = (\alpha, call_f, V_s)$ is any subprogram $\mathscr{P}'$ that is obtained from $\mathscr{P}$ by deleting zero or more statements in a two-pass algorithm:*

1. *a first step to execute a backward slicing with the traditional slicing criterion $C = (call_f, V_s)$ retrieved from C_a — $call_f$ corresponds to the call statement under consideration, and V_s corresponds to the set of variables present in the invocation $call_f$ and intervening in the precondition formula (α) of f*

2. *a second step to check if the statements preceding the $call_f$ statement will lead to the satisfaction of the callee precondition.*

For the second step in the two-pass algorithm, in order to check which statements are respecting or violating the precondition we are using abstract interpretation, in particular symbolic execution.

According to the original idea of *James King* in [23], symbolic execution can be described as "instead of supplying the normal inputs to a program (e.g. numbers) one supplies symbols representing arbitrary values. The execution proceeds as in a normal execution except that values may he symbolic formulas over the input symbols."

Using symbolic execution we will be able to propagate the precondition of the function being called through the statements preceding the call statement. In particular, to integrate symbolic execution with our system, we are thinking in use JavaPathFinder [24]. JavaPathFinder is a tool than can perform program execution with symbolic values. Moreover, JavaPathFinder can mix concrete and symbolic execution, or switch between them. JavaPathFinder has been used for finding counterexamples to safety properties and for test input generation.

To sum up, the main goal of our caller-based slicing algorithm is to facilitate the use of annotated components by discovering statements that are critical for the satisfaction of the precondition, i.e., that do not verify it or whose values can lead to its non-satisfaction (a kind of *tracing call analysis of annotated procedures*).

4 Annotated System Dependency Graph (SDGa)

In this section we present the definition of Annotated System Dependency Graph, SDG_a for short, that is the internal representation that supports our slicing-based code analysis approach.

Definition 8 (Annotated System Dependence Graph) *An Annotated System Dependency Graph, SDG_a, is a SDG in which some nodes of its constituent PDGs are annotated nodes.*

Definition 9 (Annotated Node) *Given a PDG for an annotated procedure $\mathscr{P}_a$, an Annotated Node is a pair $< S_i, a >$ where S_i is a statement or predicate (control statement or entry node) in $\mathscr{P}_a$, and a is its annotation: a pre-condition α, a post-condition ω, or an invariant δ.*

The differences between a traditional SDG and an SDG_a are:

- Each procedure dependency graph (PDG) is decorated with a precondition as well as with a postcondition in the entry node;

- The *while* nodes are also decorated with the loop invariant (or true, in case of invariant absence);
- The *call* nodes include the pre- and postcondition of the procedure to be called (or true, in case of absence); these annotations are retrieved from the respective PDG and instantiated as explained below.

We can take advantage from the *call linkage dictionary* present in the SDG_a (inherited from the underlying SDG) to associate the variables used in the calling statement (the actual parameters) with the formal parameters involved in the annotations.

Given a program and an annotated slicing criterion, we identify the node of the respective SDG_a that corresponds to the criterion (yellow node in Figure 1). After building the respective caller-based slice, the critic statements will be highlighted in the graph, making easier to identify the statements violating the precondition (red nodes in Figure 1).

5 An illustrative example

To illustrate the previous definitions and our proposal, consider the program listed below (Example 1) that computes the maximum difference among student ages in a class.

Example 1 DiffAge

```
public int DiffAge() {
    int min = System.Int32.MaxValue, max = System.Int32.MinValue, diff;

    System.out.print("Number of elements: ");
    int num = System.in.read();
    int[] a = new int[num];
    for(int i=0; i<num; i++) { a[i] = System.in.read();  }

    for(int i=0; i<a.Length; i++) {
        max = Max(a[i],max);
        min = Min(a[i],min);
    }

    diff = max - min;
    System.out.println("The difference between the greatest " +
                       "and the smallest ages is " + diff);
    return diff;
}
```

This program reuses two annotated components: `Min`, defined in Example 2, that returns the smallest of two positive integers; and Max, defined in Example 3, that returns the greatest of two positive integers.

Suppose that we want to study (or analyze) the call to `Min` in the context of `DiffAge` program.

For that purpose, the slicing criterion will be: $C_a = (x \geq 0 \&\& y \geq 0, Min, \{a[i], min\})$

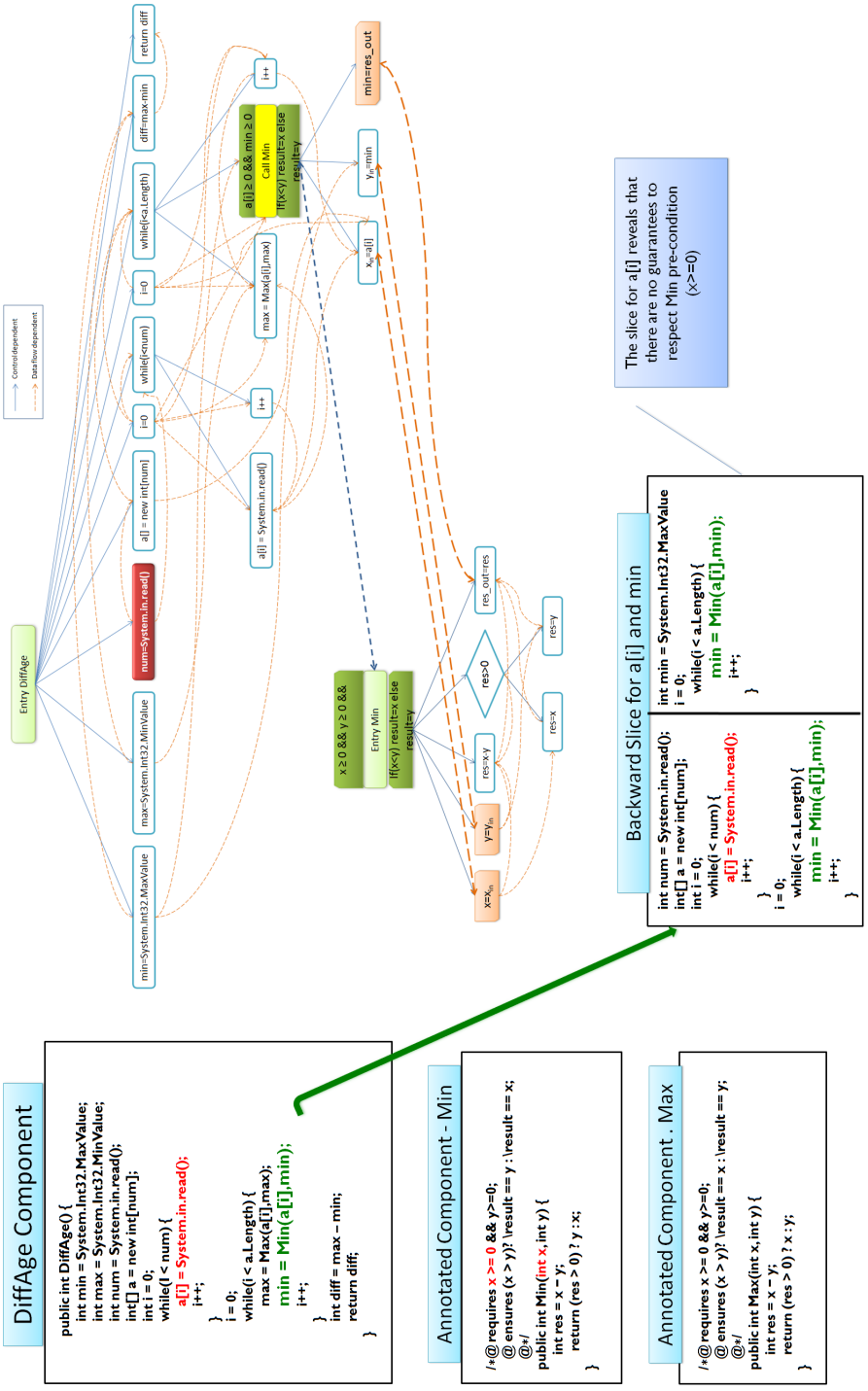

Figure 1: SDG$_a$ for a program and its role on Caller-based Slicing

With this criterion, a backward slicing process is performed taking into account the variables present in V_s. Then, using the obtained slices, the detection of contract violations is executed. For that, the precondition is back propagated (using symbolic execution) along the slice to verify if it is preserved after each statement. Observing the slice corresponding to the variable `a[i]` (see Example 4 below), is evident that it can not be guaranteed that all integer elements are greater than zero; so a potential precondition violation is detected.

Example 2 Min

$/*$ @ requires $x \geq 0$ && $y \geq 0$
@ ensures $(x > y)?$ \result $== x :$ \result $== y$
@ $*/$

```
1: public int Min(int x, int y) {
2: int res;
3: res = x − y;
4: return ((res > 0)? y : x);
5: }
```

Example 3 Max

$/*$ @ requires $x \geq 0$ && $y \geq 0$
@ ensures $(x > y)?$ \result $== y :$ \result $== x$
@ $*/$

```
1: public int Max(int x, int y) {
2: int res;
3: res = x − y;
4: return ((res > 0)? x : y);
5: }
```

Example 4 Backward Slice for a[i]

```
int[] a = new int[num];
for(int i=0; i<num; i++) { a[i] = System.in.read();  }
for(int i=0; i<a.Length; i++) {
    max = Max(a[i],max);
    min = Min(a[i],min);    }
```

All the contract violations detected will be reported during the next step. In the example above, the user will receive an *warning* alerting to the possibility of calling Min with negative numbers (what does not respect the precondition).

As referred, in order to visualize the contracts that are violated and the critical statements, we display the SDG_a with such entities colored in red (see Figure 1). The role of the SDG_a will be crucial not only to understand the data and control flow of a program as well as to understand the impact of the annotations and their violations.

6 GamaPolarSlicer

In this section, we introduce GamaPolarSlicer, a tool that we are building to implement our ideas; it will become available to open source communities, as soon as possible. This project is being developed in the context of the *CROSS project — An Infrastructure for Certification and Re-engineering of Open Source Software* at Universidade do Minho[1].

First we describe the architecture of the tool, and then we give some technical details about its implementation.

6.1 Architecture

As referred previously, our goal is to ease the process of incorporate an annotated component into an existent system. This integration should be smooth, in the sense of that it should not turn a correct system into an incorrect one.

To achieve this goal, it is necessary:

- to verify the component correctness with respect to its contract (using a traditional *Verification Condition Generator*, already incorporated in GamaSlicer [25], available at http://gamaepl.di.uminho.pt/gamaslicer);
- to verify if the actual calling context preserves the precondition;
- to verify if the component is properly used in the actual context after the call;
- Given a reusable component and a set of calling points, specify the component body according to the concrete calling needs.

The chosen architecture is based on the classical structure of a language processor. Figure 2 shows GamaPolarSlicer architecture.

- **Source Code** — the input to analyze.
- **Lexical Analysis, Syntactic Analysis, Syntactic Analysis** — the Lexical layer converts the input into symbols that will be later used in the identifiers table. The Syntactic layer uses the result of the Lexical layer above and analyzes it to identify the syntactic structure of it. The Semantic layer adds the semantic information to the result from the Syntactic layer. It is in this layer that the identifier table is built.
- **Invocations Repository** — is where all invocations found on the input are stored in order to be used later as support to the slicing process.
- **Annotated Components Repository** — is where all components with a formal specification (precondition and postcondition at least) are stored. It is used in the slicing process only to filter the invocations (from the invocation repository) without any annotation. Has an important role when verifying if the invocation respects component's contract.

[1] More details about this project can be found in http://wiki.di.uminho.pt/twiki/bin/view/Research/CROSS/WebHome

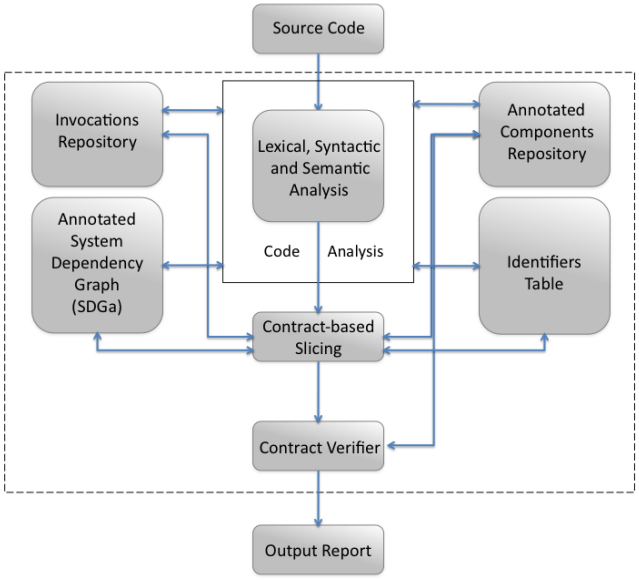

Figure 2: GammaPolarSlicer Architecture

- **Identifiers Table** — has an important role on this type of programs as always. All symbols and associated semantic found during the analysis phase are stored here. It will be one of the backbones of all structures supporting the auxiliary calculations.
- **Annotated System Dependency Graph** — is the intermediate structure chosen to apply the slicing.
- **Caller-based Slicing** — uses both invocations repository and annotated components repository to extract the parameters to execute the slicing for each invoked annotated component. The resulting slice is a SDG_a this a subgraph of the original SDG_a.
- **Contract Verification** — using the slice that resulted from the layer above, and using the component contract, this layer analyzes every node on the slice and verifies in all of them if there are guarantees that every annotation in the contract is respected.
- **Output Report** — presents a view of all violations found during the whole process to the user. In future, we intend to include the possibility of present suggestions to solve these issues.

6.2 Implementation

To address all the ideas, approaches and techniques presented in this paper, it was necessary to choose the most suitable technologies and environments to support the development.

To address the *design-by-contract* approach we decide to use the Java Modeling Language (JML) [2]. JML is a formal behavior interface specification language, based on design-by-contract paradigm, that allows code annotations in Java programs [26].

JML is quite useful as allows to describe how the code should behave when running it [26]. Preconditions, postconditions and invariants are examples of formal specifications that JML provides.

As the goal of the tool is not to create a development environment but to enhance an existing one, we decided to implement it as an Eclipse [3] plugin.

The major reasons that led to this decision were: the large community and support. Eclipse is one of the most popular frameworks to develop Java applications and thus a perfect tool to test our goal; the fact that it includes a great environment to develop new plugins. The Plugin Development Environment (PDE) [4] that allows a faster and intuitive way to develop Eclipse plugins; has a built-in support for JML, freeing us from checking the validity of such annotations.

A scratch of the first version of GamaPolarSlicer prototype is depicted in Figure 3.

7 Related Work — Slicing

In this section we review the published work on the area of slicing annotated programs, as those contribution actually motivate the present proposal. Although the works referred use the annota-

[2] http://www.cs.ucf.edu/ leavens/JML/
[3] http://www.eclipse.org/
[4] http://www.eclipse.org/pde/

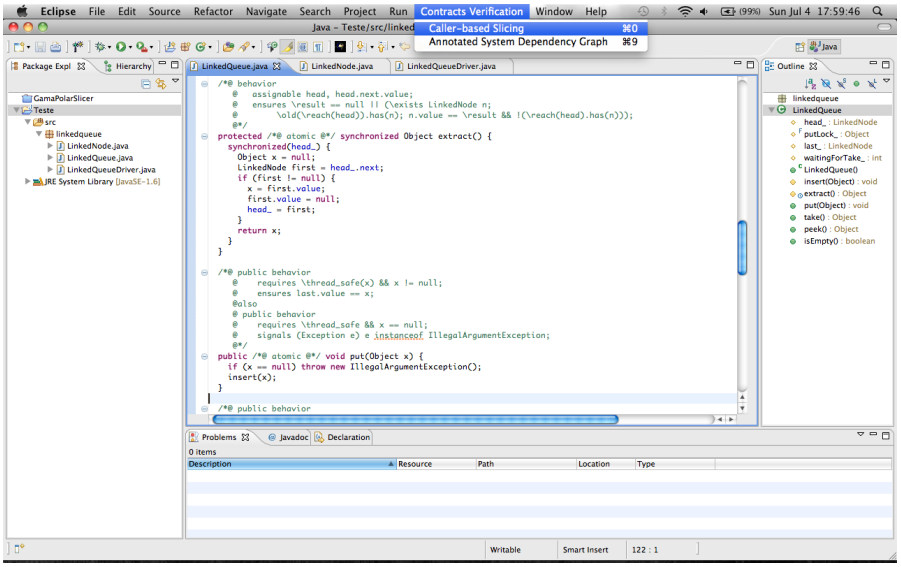

Figure 3: GamaPolarSlicer prototype

tions to slice a program, the concrete goal of such works differs from ours. The main difference is that we do not assume that all the procedures in a program are annotated and correct with respect to these contracts. We are assuming that only the procedure being integrated is annotated.

In [27], *Comuzzi et al* present a variant of program slicing called *p-slice* or *predicate slice*, using Dijkstra's weakest preconditions (wp) to determine which statements will affect a specific predicate. Slicing rules for assignment, conditional, and repetition statements were developed. They presented also an algorithm to compute the minimum slice.

In [28], *Chung et al* present a slicing technique that takes the specification into account. They argue that the information present in the specification helps to produce more precise slices by removing statements that are not relevant to the specification for the slice. Their technique is based on the weakest pre-condition (the same present in *p-slice*) and strongest post-condition — they present algorithms for both slicing strategies, backward and forward.

Comuzzi et al [27], and *Chung et al* [28], provide algorithms for code analysis enabling to identify suspicious commands (commands that do not contribute to the postcondition validity).

In [16], *Harman et al* propose a generalization of conditioned slicing called pre/post conditioned slicing. The basic idea is to use the pre-condition and the negation of the post-condition in the conditioned slicing, combining both forward and backward conditioning. This type of program slicing is based on the following rule: "Statements are removed if they cannot lead to the satisfaction of the negation of the post condition, when executed in an initial state which satisfies the pre-condition". In case of a program which correctly implements the pre- and post-condition, all statements from the program will be removed. Otherwise, those statements that do not respect the conditions are left, corresponding to statements that potentially break the conditions (are either incorrect or which are innocent but cannot be detected to be so by slicing). The result of this work can be applied as a correctness verification for the annotated procedure.

In [29], *Cruz et al* propose the contract-based slicing notion. Given any specification-based slicing algorithm (working at the level of commands), a contract-based slice can be calculated by slicing the code of each individual procedure independently with respect to its contract (called an *open slice*), or taking into consideration the calling contexts of each procedure inside a program (called a *closed slice*).

8 Related Work — Visualization of (sliced) programs

As in this paper we also focus on the visualization of programs with annotated components, and their slices that trace the calls with respect to the called preconditions, we devote this section to review the contributions on the area of slice visualization that more directly influence our proposal.

In [30], *Ball et al.* present SeeSlice, an interactive browsing tool to better visualize the data generated by slicing tools. The SeeSlice interface facilitates the visualization by making slices fully visible to user, even if they cross the boundaries of procedures and files.

In [31], *Gallagher et al.* propose an approach in order to reduce the visualization complexity by using decomposition slices. A decomposition slice is a kind of slice that depends only on a variable (or a set of variables) and does not consider the location of the statement (a traditional slice depends on both a variable and a statement in a program). The decomposition slice visualization implemented in Surgeon's Assistant [32] visualizes the inclusion hierarchy as a graph using the VCG (Visualization of Compiler Graphs) [33].

In [34], *Deng et al* present Program Slice Browser, an interactive and integrated tool which main goal is to extract useful information from a complex program slice. Some of the features of such tool are: adaptable layout for convenient display of a slice; multi-level slice abstractions; integration with other visualization components, and capabilities to support interaction and cross-referencing within different views of the software.

In [21], *Krinke* presents a declarative approach to layout Program Dependence Graphs (PDG) that generates comprehensible graphs of small to medium size procedures. The authors discussed how a layout for PDG can be generated to enable an appealing presentation. The PDG and the computed slices are shown in a graphical way. This graphical representation is combined with the textual form, as the authors argue that is much more effective than the graphical one. The authors also solved the problem of loss of locality in a slice, using a distance-limited approach; they try to answer research questions such as: 1) why a statement is included in the slice?, and 2) how strong is the influence of the statement on the criterion?

In [22], *Balmas* presents an approach to decompose System Dependence Graphs in order to have graphs of manageable size: groups of nodes are collapsed into one node. The system implemented provides three possible decompositions to be browsed and analyzed through a graphical interface: nodes belonging to the same procedure; nodes belonging to the same loop; nodes belonging to the two previous ones.

9 Conclusion

As can be seen along the paper, the motivation for our research is to apply slicing, a well known technique in the area of source code analysis, to create a tool that aids programmers to build correct open source programs reusing annotated procedures.

The tool under construction, GamaPolarSlicer, was described in Section 6. Its architecture relies upon the traditional compiler structure; on one hand, this enables the automatic generation of the tool core blocks, from the language attribute grammar; on the other hand, it follows an approach in which our research team has a large knowhow (apart from many DSL compilers, we developed a lot of Program Comprehension tools: Alma, Alma2, WebAppViewer, BORS, and SVS). The new and complementary blocs of GamaPolarSlicer implement slice and graph-traversal algorithms that have a sound basis, as described in Sections 2, 3, and 4; this allows us to be confident in there straight-forward implementation.

GamaPolarSlicer will be included in Gama project (for more details see http://gamaepl.di.uminho.pt/gama/index.html). This project aims at mixing specification-based slicing algorithms with program verification algorithms to analyze annotated programs developed under Contract-base Design approach. GamaSlicer is the first tool built under this project for intra-procedural analysis that is available at http://gamaepl.di.uminho.pt/gamaslicer/.

We believe that this set of tools will save time and make safer the process of build open-source software systems.

Bibliography

[1] Maiden, N.A.M., Sutcliffe, A.G.: People-oriented software reuse: the very thought. In: Advances in Software Reuse - Second International Workshop on Software Reusability, IEEE Computer Society Press (1993) 176–185

[2] Sherif, K., Vinze, A.: Barriers to adoption of software reuse a qualitative study. Inf. Manage. **41**(2) (2003) 159–175

[3] Shiva, S.G., Shala, L.A.: Software reuse: Research and practice. In: ITNG, IEEE Computer Society (2007) 603–609

[4] Meyer, B.: Applying "design by contract". Computer **25**(10) (1992) 40–51

[5] Weiser, M.D.: Program slices: formal, psychological, and practical investigations of an automatic program abstraction method. PhD thesis, Ann Arbor, MI, USA (1979)

[6] M., K.: Interprocedural dynamic slicing with applications to debugging and testing. PhD thesis, Linkoping University, Sweden (1993)

[7] Agrawal, H., DeMillo, R.A., Spafford, E.H.: Debugging with dynamic slicing and back-tracking. Software - Practice and Experience **23**(6) (1993) 589–616

[8] Weiser, M., Lyle, J.: Experiments on slicing-based debugging aids. In: Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers, Norwood, NJ, USA, Ablex Publishing Corp. (1986) 187–197

[9] Binkley, D.: The application of program slicing to regression testing. Information and Software Technology **40**(11-12) (1998) 583–594

[10] Harman, M., Danicic, S.: Using program slicing to simplify testing. Software Testing, Verification & Reliability **5**(3) (1995) 143–162

[11] Ottenstein, L., Thuss, J.: Slice based metrics for estimating cohesion (1993)

[12] Lakhotia, A.: Rule-based approach to computing module cohesion. In: ICSE '93: Proceedings of the 15th international conference on Software Engineering, Los Alamitos, CA, USA, IEEE Computer Society Press (1993) 35–44

[13] Cimitile, A., Lucia, A.D., Munro, M.: A specification driven slicing process for identifying reusable functions. Journal of Software Maintenance **8**(3) (1996) 145–178

[14] Gallagher, K.B., Lyle, J.R.: Using program slicing in software maintenance. IEEE Transactions on Software Engineering **17**(8) (1991) 751–761

[15] Lucia, A.D., Fasolino, A.R., Munro, M.: Understanding function behaviors through program slicing. In: Proceedings of the 4th Workshop on Program Comprehension. (1996) 9–18

[16] Harman, M., Hierons, R., Fox, C., Danicic, S., Howroyd, J.: Pre/post conditioned slicing. icsm **00** (2001) 138

[17] Beck, J., Eichmann, D.: Program and interface slicing for reverse engineering. In: ICSE '93: Proceedings of the 15th international conference on Software Engineering, Los Alamitos, CA, USA, IEEE Computer Society Press (1993) 509–518

[18] Cimitile, A., Lucia, A.D., Munro, M.: Identifying reusable functions using specification driven program slicing: a case study. In: ICSM '95: Proceedings of the International Conference on Software Maintenance, Washington, DC, USA, IEEE Computer Society (1995) 124

[19] Binkley, D., Horwitz, S., Reps, T.: Program integration for languages with procedure calls. ACM Trans. Softw. Eng. Methodol. **4**(1) (1995) 3–35

[20] Horwitz, S., Prins, J., Reps, T.: Integrating noninterfering versions of programs. ACM Trans. Program. Lang. Syst. **11**(3) (1989) 345–387

[21] Krinke, J.: Visualization of program dependence and slices. In: ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance, Washington, DC, USA, IEEE Computer Society (2004) 168–177

[22] Balmas, F.: Displaying dependence graphs: a hierarchical approach. J. Softw. Maint. Evol. **16**(3) (2004) 151–185

[23] King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7) (1976) 385–394

[24] Anand, S., Păsăreanu, C.S., Visser, W.: Jpf-se: a symbolic execution extension to java pathfinder. In: TACAS'07: Proceedings of the 13th international conference on Tools and algorithms for the construction and analysis of systems, Berlin, Heidelberg, Springer-Verlag (2007) 134–138

[25] da Cruz, D., Henriques, P.R., Pinto, J.S.: Gamaslicer: an online laboratory for program verification and analysis. In: Proceedings of the 10th Workshop on Language Descriptions Tools and Applications (LDTA'10). (2010)

[26] Leavens, G.T., Cheon, Y.: Design by contract with jml (2004)

[27] Comuzzi, J.J., Hart, J.M.: Program slicing using weakest preconditions. In: FME '96: Proceedings of the Third International Symposium of Formal Methods Europe on Industrial Benefit and Advances in Formal Methods, London, UK, Springer-Verlag (1996) 557–575

[28] Chung, I.S., Lee, W.K., Yoon, G.S., Kwon, Y.R.: Program slicing based on specification. In: SAC '01: Proceedings of the 2001 ACM symposium on Applied computing, New York, NY, USA, ACM (2001) 605–609

[29] da Cruz, D., Henriques, P.R., Pinto, J.S.: Contract-based slicing. In: Proceedings of the Fourth Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'10). (2010) to appear.

[30] Ball, T., Eick, S.G.: Visualizing program slices. In: VL. (1994) 288–295

[31] Gallagher, K., O'Brien, L.: Reducing visualization complexity using decomposition slices. In: Proc. Software Visualisation Work., Adelaide, Australia, Department of Computer Science, Flinders University (11–12 Dezembro 1997) 113–118

[32] Gallagher, K.: Visual impact analysis. In: ICSM '96: Proceedings of the 1996 International Conference on Software Maintenance, Washington, DC, USA, IEEE Computer Society (1996) 52–58

[33] Sander, G.: Graph layout through the vcg tool. In: GD '94: Proceedings of the DIMACS International Workshop on Graph Drawing, London, UK, Springer-Verlag (1995) 194–205

[34] : Program slice browser. In: IWPC '01: Proceedings of the 9th International Workshop on Program Comprehension, Washington, DC, USA, IEEE Computer Society (2001) 50