GammaPolarSlicer
A Contract-based Tool to help on Reuse

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

Departamento de Informática e CCTC
Universidade do Minho
Braga, Portugal

Abstract. In software development, it is often desirable to reuse existing software components. This has been recognized since 1968, when Douglas Mcllroy of Bell Laboratories proposed basing the software industry on reuse. Despite the failures in practice, many efforts have been made to make this idea successful.
In this context, 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 an annotated component with a contract into a legacy system will preserve the behavior of the former.
To complement the efforts done and the benefits of the 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.

1 Introduction

Reuse is a very simple and natural concept, however in practice it 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 in 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 difficult, even for experienced developers [1]. Another challenge to component reuse is to certify that the integration of such component in a legacy system is 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 new system, 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's 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 GamaPolarSlicer, a tool that we are currently developing to identify when an invocation is violating the component annotation, and display, whenever possible, a diagnostic or guidelines to correct it. For such a purpose, the tool implements the **caller-based slicing** algorithm, that takes into account the calls of an annotated component to certify that it is being correctly used.

The remainder of this paper is structured into 5 sections. Section 2 is devoted to basic concepts. In this section the theoretical foundation for GamaPolarSlicer is settle down; the notions of caller-based slicing and annotated system dependence graph are defined. Section 3 gives a general overview of GamaPolarSlicer, introducing its architecture; each block on the diagram will be explained. Sub-section 3.1 complements the architecture discussing the decisions taken to implement the tool and presenting the interface underdevelopment. Section 4, also a central one, illustrates the main idea through a concrete example. As to our knowledge we do not known any tool similar to GamaPolarSlicer, in Section 5 we discuss related work concerned with the use of slicing technique for annotated programs. Then the paper is closed in Section 6.

2 Basic Concepts

We consider that each procedure consists of a body of code, annotated with a precondition and a postcondition that form the procedure specification, or *contract*. The body may additionally be annotated with loop invariants. Occurrences of variables in the precondition and postcondition of a procedure refer to their values in the pre-state and post-state of execution of the procedure respectively.

2.1 Caller-based slicing

In this section, we briefly introduce our slicing algorithm.

Definition 1 (Annotated Slicing Criterion) *An* annotated slicing criterion *of a program $\mathcal{P}$ consists of a triple $C_a = (a, S_i, V_s)$, where $a \in \{\alpha, \delta\}$ is an annotation of $\mathcal{P}_a$ (the annotated callee), S_i correspond to the statement of $\mathcal{P}$ calling $\mathcal{P}_a$ and V_s is a subset of the variables in $\mathcal{P}$ (the caller), that are the actual parameters used in the call and constrained by α or δ.*

Definition 2 (Caller-based slicing) *A* caller-based slice *of a program $\mathcal{P}$ on an annotated slicing criterion $C_a = (\alpha, call_f, V_s)$ is any subprogram $\mathcal{P}'$ that is obtained from $\mathcal{P}$ 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 precondition satisfaction of the callee;*

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 [5], 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 be 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 [6]. 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.

The main goal of our caller-based slicing algorithm is to ease the use of annotated components by discovering statements that are critical for the satisfaction of the precondition or postcondition (i.e, that do not verify it, or whose value can lead to the non-satisfaction) before or after calling an annotated procedure (a tracing call analysis of annotated procedures). In the work reported here, we just deal with preconditions and statements before the call.

2.2 Annotated System Dependence Graph (SDGa)

In this section we present the definition of Annotated System Dependence Graph, SDG_a for short, that is the internal representation that supports our slicing-based code analysis approach. We start with some preliminary definitions.

Definition 3 (Procedure Dependence Graph) *Given a procedure $\mathcal{P}$, 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 $\mathcal{P}$, 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 affects 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 connected 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 receive the returned values.

Definition 4 (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.*

Definition 5 (Annotated System Dependence Graph) *An Annotated System Dependence Graph, SDG_a, is a SDG in which some nodes of its constituent PDGs are annotated nodes.*

Definition 6 (Annotated Node) *Given a PDG for an annotated procedure $\mathcal{P}_a$, an Annotated Node is a pair $< S_i, a >$ where S_i is a statement or predicate (control statement or entry node) in $\mathcal{P}_a$, and a is its annotation: a pre-condition α, a post-condition ω, or an invariant δ.*

The differences between a traditional SDG and a SDG_a are:

- Each procedure dependence 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) — the mapping between the variables present in the call statement (the actual parameters) and the formal parameters of the procedure — to associate the variables used in the calling statement with the formal parameters involved in the annotations.

3 GamaPolarSlicer — Architecture and Implementation

As referred previously, our goal is to ease the process of incorporating 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 assure this, there is the need to verify a set of conditions with respect to the annotated component and its usage:

- Verify its correctness with the respect to its contract (using a traditional *Verification Condition Generator*, already incorporated in GamaSlicer [7], available at `http://gamaepl.di.uminho.pt/gamaslicer`);
- Given a concrete call to the reusable component, verify if the concrete calling context preserves the precondition;
- Given a concrete call and the postcondition of the component, verify if it is properly used in the concrete 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 to achieve the second condition was based on the classical structure of a language processor. Figure 1 shows the defined GamaPolarSlicer architecture. Notice that the third and fourth conditions will be addressed by future projects.

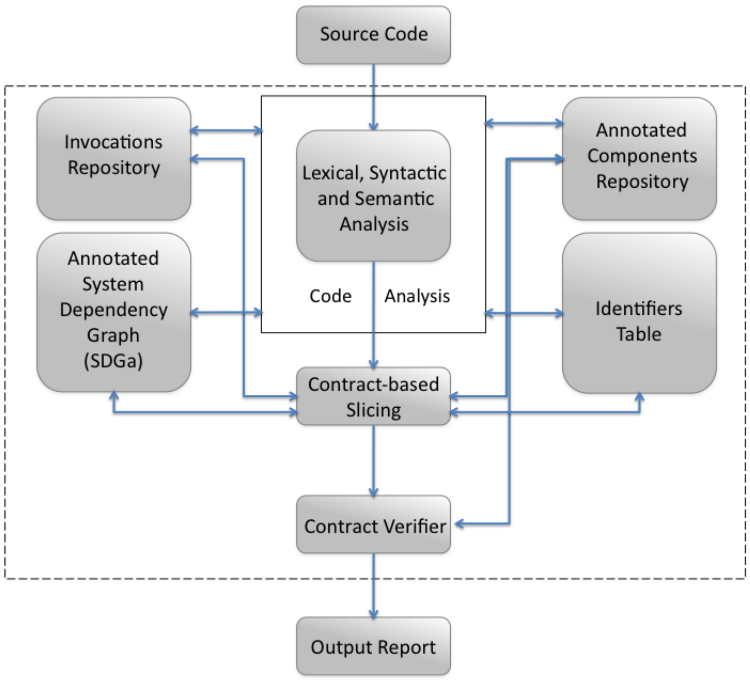


Fig. 1. GammaPolarSlicer Architecture

Source Code is 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 post condition 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.

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 Dependence 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 a later stage of this project, exists the possibility of also present suggestions to solve them.

3.1 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) [1]. JML is a formal behavior interface specification language, based on *design-by-contract* paradigm, that allows code annotations in Java programs [8]. JML is quite useful as it allows to describe how the code should behave when running it. Also it allows the specification of the syntactic interface [8]. 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 support one, our first thought was to implement it as an Eclipse [2] plugin. The major reasons that led to this decision were:

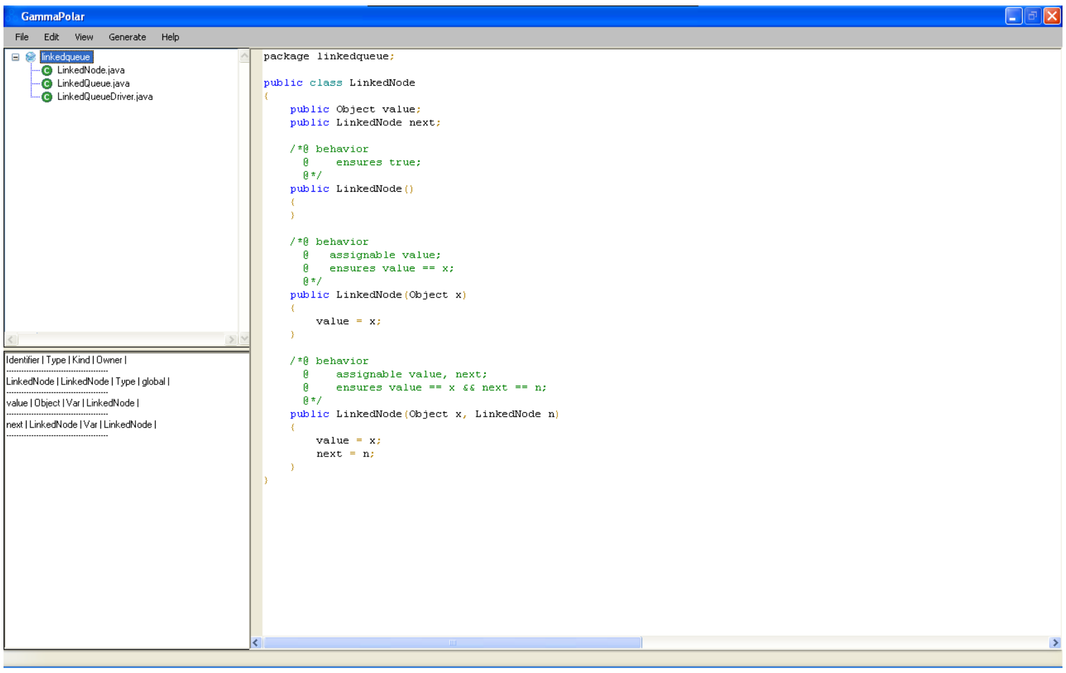
[1] http://www.cs.ucf.edu/ leavens/JML/

[2] http://www.eclipse.org/

– 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) [3] that allows a faster and intuitive way to develop Eclipse plugins;
– the built-in support for JML, freeing us from checking the validity of such annotations.

However, the parser generated for Java/JML grammar exceeded the limit of bytes allowed to a Java class (65535 bytes). Thus, this limitation led us to abandon the idea of the Eclipse plugin and implement GamaPolarSlicer using Windows Forms and C# (using the .NET framework).

Figure 2 shows the current interface of GamaPolarSlicer.



```
Identifier | Type | Kind | Owner |
----------------------------------------
LinkedNode | LinkedNode | Type | global |
----------------------------------------
value | Object | Var | LinkedNode |
----------------------------------------
next | LinkedNode | Var | LinkedNode |
----------------------------------------
```

```java
package linkedqueue;

public class LinkedNode
{
    public Object value;
    public LinkedNode next;

    /*@ behavior
      @    ensures true;
      @*/
    public LinkedNode()
    {
    }

    /*@ behavior
      @   assignable value;
      @   ensures value == x;
      @*/
    public LinkedNode(Object x)
    {
        value = x;
    }

    /*@ behavior
      @    assignable value, next;
      @    ensures value == x && next == n;
      @*/
    public LinkedNode(Object x, LinkedNode n)
    {
        value = x;
        next = n;
    }
}
```

Fig. 2. Interface of GamaPolarSlicer prototype

4 An illustrative example

To illustrate what we intended to achieve, please consider the Example 1 listed below that computes the maximum difference among student ages in a class. This component

[3] http://www.eclipse.org/pde/

reuses other two: the annotated component `Min`, defined in Example 2, that returns the lowest of two positive integers; and Max, defined in Example 3, that returns the greatest positive integer.

Example 1 DiffAge

```
public int DiffAge() {
      int min = System.Int32.MaxValue;
      int max = System.Int32.MinValue;
      int 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 gap between max and min age is " + diff);
      return diff;
}
```

Example 2 Min

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

```
public int Min(int x, int y) {
int res;
$res = x - y$;
return ((res < 0)? x : y);
}
```

Let us consider that we want to analyze the `Min` invocation present in the `DiffAge` component.

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

In the second step, a backward slicing process is performed taking into account the variables present in V_s. Then, using the obtained slices, the detection of contract

Example 3 Max

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

```
public int Max(int x, int y) {
int res;
res = x - y;
return ((res > 0)? x : y);
}
```

violations starts. For that, the precondition is back propagate (using symbolic execution) along the slice to verify if it is preserved after each statement. Observing the slice for the variable `a[i]`, listed in the example 4 below, it can not be guaranteed that all integer elements are greater than zero; so a potential precondition violation is detected.

Example 4 Backward Slicing 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);
}
```

The third step consists in the notification of all the contract violations detected. In the example above, the user will receive a *warning* alerting to the possible invocation of Min with negative numbers (what does not respect the precondition).

In order to help to visualize which contracts and statements are being violated, we display the SDG_a with such entities colored in red. Figure 3 shows a fragment of the SDG_a for the example above.

5 Related Work

In this section we review the published work on the area of slicing annotated programs, as those contributions actually motivate the present proposal.

In [9], *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.

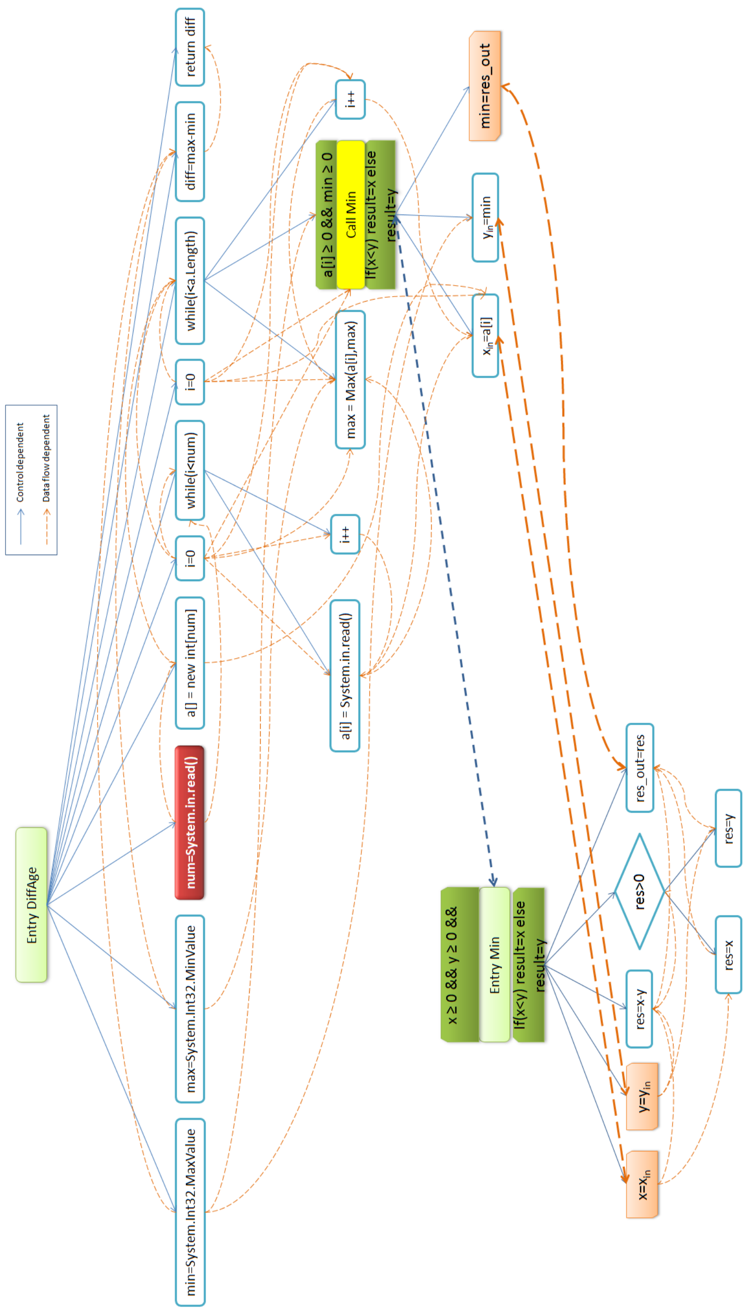


Fig. 3. Example of an Annotated System Dependence Graph

In [10], *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 [9], and *Chung et al* [10], provide algorithms for code analysis enabling to identify suspicious commands (commands that do not contribute to the post-condition validity).

In [11], *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.

6 Conclusion

As can be seen in section 4, 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 safety programs reusing annotated procedures.

The tool under construction, GamaPolarSlicer, was described in Section 3. 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 blocks of GamaPolarSlicer implement slice and graph-traversal algorithms that have a sound basis, as described in Section 2; 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/`.

Although reuse was not the topic of the paper (just some considerations were drawn in the Introduction), reuse is the main motivation for GamaPolarSlicer development. We are preparing an experiment to assess the validity of our proposal and the usefulness of the tool.

References

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. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7) (1976) 385–394
6. 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
7. 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)
8. Leavens, G.T., Cheon, Y.: Design by contract with jml (2004)
9. 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
10. 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
11. Harman, M., Hierons, R., Fox, C., Danicic, S., Howroyd, J.: Pre/post conditioned slicing. icsm **00** (2001) 138