DOI: 10.2298/CSIS110107006A

GammaPolarSlicer

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 McIlroy 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.

Keywords: safety reuse, caller-based slicing, annotated system dependency graph.

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 [9]. Sometimes this is due to the lack of maturity on supporting tools that should easily find a component in a repository or library [11]. 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 [11, 12]. 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 [9].

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 [9]. 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 [10] 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 4 complements the architecture discussing the decisions taken to implement the tool and presenting the interface underdevelopment. Section 5, 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 6 we discuss related work concerned with the use of slicing technique for annotated programs. Then the paper is closed in Section 7.

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 [7], 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 [1]. 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. Figure 1 shows an example of a SDG_a.

3. GamaPolarSlicer Architecture

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. It is necessary to:

- to verify the component correctness with the respect to its contract (using a traditional *Verification Condition Generator*, already incorporated in GamaSlicer [5], 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 whole process is a bit complex and was divided in a set of smaller problems (*divide and conquer*). The tool under discussion in this document will only focus on the second item, working with preconditions and backward slicing. Notice that the third and fourth conditions will be addressed by future projects.

The chosen architecture, designed to achieve the second condition, was based on the classical structure of a language processor. Figure 2 shows the defined GamaPolarSlicer architecture.

Source Code can be a Java project or only Java files to analyze by the tool.

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. These three layers, usually are always present in language processors.

Invocations Repository is the data structure where all function calls processed during the code analysis are stored. The contract verification will be applied to each one of these calls and the slicing criterion of each one will consider the parameters struct.

Annotated Components Repository is the data structure where all components with a formal specification (precondition and postcondition at least)

Fig. 1. SDG$_a$ for a program

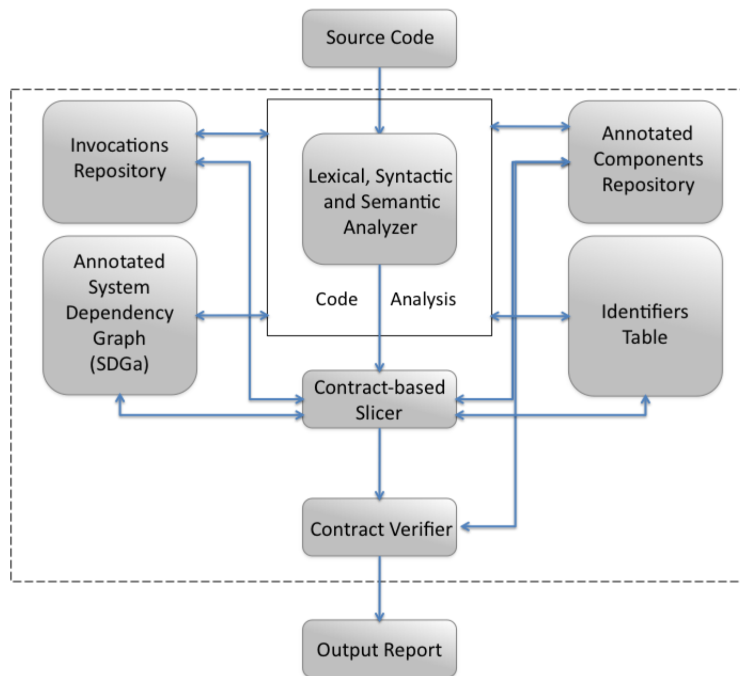


Fig. 2. GammaPolarSlicer Architecture

are stored. All these components will be later used in the slicing process in order to filter all the calls (from the invocation repository) defined without any type of annotation. This repository has an important role when verifying if the call respects the component contract.

Identifiers Table flags, always, an important role on the implementation of the processor. All symbols and associated semantic processed during the code analysis phase are stored here. It will be one of the backbones of all structures and of all stages of the tool process.

Annotated System Dependence Graph is the internal representation chosen to support our slicing-based code analysis approach. Constructed during the code analysis, this type of graph allows to associate formal annotations , like preconditions, postconditions or even invariants, to the its nodes (see Section 2.2).

Caller-based Slicing is the layer where the backward slicing is applied to each annotated component call. It 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, with all the statements relevant to the particular call.

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 describes all contract violations found during the whole process. All violations found are marked with the degree of relevance in order to aid the user in the revision process. In the future, the tool will provide some suggestions to solve these issues, and a graphic display of the violations over the SDG_a.

4. GamaPolarSlicer 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.

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

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:

– 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.

After the first days of the development process we realized that Java has a limitation regarding the number of bytes per class (only allows 65535 bytes per class). This limitation prevented us of continue the work with Java because the parser we were generating for Java/JML grammar exceeded this limit of bytes. This led us to abandon the idea of the Eclipse plugin and implement GamaPolarSlicer using Windows Forms and C# (under .NET framework). Figure 3 shows the current interface of GamaPolarSlicer.

4.1. Tool Workflow

As depicted in the architecture (see Figure 2), our tool is divided in a set of phases where each one solves a particular task. In this section we will explain how these phases interact with each other and how data flows between them.

The tool begins analyzing the source code (Java code/JML annotations) in order to extract all symbols and to construct all data structures. In order to ease the slicing process it is mandatory to have an appropriate data structure to support this type of techniques. For this job we have chosen the SDG_a(SDG$_a$) (see Section 2.2). Using all the gathered information during the code analysis we are able to construct this graph.

The graph and the Identifier Table construction are made once for each input file processed. At the end of these steps, the system will have a set of Identifier Tables and a set of SDG$_a$. The union between all the SDG$_a$ will result in the SDG$_a$ for the entire source code. The same happens to the set of Identifier Table.

After building all the data structures, the backward slicing is then applied to a component invocation and the resulting slices together with the component contract are used to verify if its call respects the contract. These steps are applied to the set of calls resulting of the intersection between the Invocation Repository and the Annotated Components Repository.

During this process (depicted in the Figure 4), if a violation is found, a textual report is issued. Also a graphic report can be selected. This graphic report uses the constructed SDG$_a$.

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

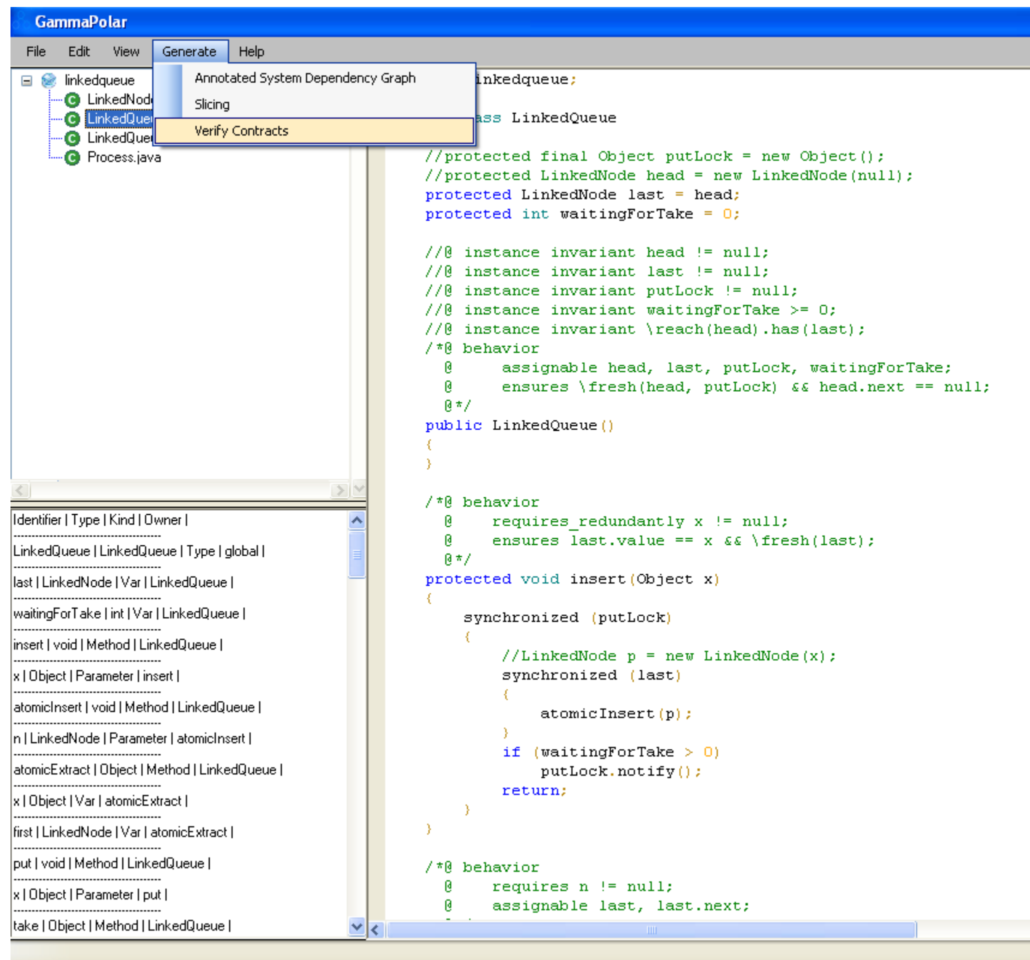


Fig. 3. Interface of GamaPolarSlicer prototype

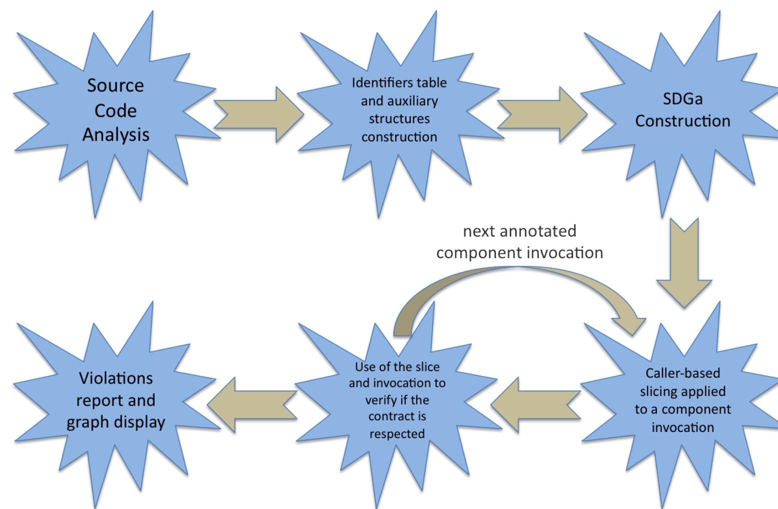


Fig. 4. Tool Workflow

4.2. Contract Verification Strategies

As already shown, the contract verification is applied upon the slices that result from the caller-based slicing process. This implies the verification of all statements on the slices to check possible violations. Depending on the statement type, there are a few critical verifications that need to be made. For readable purposes, we will use the following notation in the remainder of this paper:

– **Call** refers to the function invocation for which we want to apply the contract verification;
– **Caller** is the component where the call occurs;
– **Callee** is the component invoked.

Please consider the example 1 with two annotated components, where one of the components invokes the other.

On the notes in red, we can see that one of the parameters of the call we want to verify is also a parameter on the caller. As the verification is only made on caller (as standalone component), there is no way to verify the value of the parameter at the beginning. This lead us to the first critical verification, precondition versus precondition.

Precondition vs Precondition When the call and the caller share a parameter we decided to certify it value using the caller precondition. Doing this, we have three possible cases:

1. the caller has an annotation for the parameter and the callee does not;

Example 1 Precondition violation

```
1:  /*@ behavior
2:  @ requires a > 0;
3:  @ ensures pot = a^b;
4:  @*/
5:  public int sqr(int a, int b) {
6:        int pot = 1, i;
7:        for(i=0;i¡b;i++) {
8:             pot = mult(a, pot);
9:        }
10:       return pot;
11: }
12: /*@ behavior
13: @ requires c > 10 && d > 0;
14: @ ensures pot = c * d;
15: @*/
16: public int mult(int c, int d) {
17:       int res = c * d;
18:       return res;
19:       }
```

2. the caller does not have an annotation for the parameter and the callee does;
3. both, the caller and the callee, have an annotation for the parameter.

In the first case, it is obvious that does not change anything. If the callee does not have an annotation for the parameter then it means the parameter can assume any value.

The second case brings ambiguity to the problem. If the caller does not have an annotation for the parameter, then there is no way to guarantee that its value will respect the clause on the call contract. Even if after the verification of all statements, the value respects the clause, that value will always be dependent of the value received as parameter on the caller.

The third case, and the most complex one, gives us chance to predict a value for the parameter on the call moment. With the annotation we can calculate or predict the set of values the parameter can take during the execution of the method. To do this we have created an object with a set of flags that tell us what type of value we have and the range of values that can take.

Please consider that we have the following annotation:

```
requires x>0 && x<200
```

After processing this annotation, the object will have the flags for values higher than, lower than and between activated. The between flag is activated when the annotation contains a closed interval.

These flags also help us to make comparisons between annotations. We can compare preconditions with preconditions and even preconditions with postconditions. The last one is very important to the second critical verification.

Precondition vs Postcondition Most of all pieces of source code have function calls. When the call of these functions affects the value of a parameter on the call that we are trying to verify, then forces the verification of their postcondition (if defined). This is what we will discuss in this section.

When we found a statement with a function call in the slice result, we verify if the invoked component exists on the loaded source code. If it is an external component, like one included from an imported library, then we have no way to guarantee that the program will work correctly after this point.

During the review process of one of our papers we received a question that raised questions for another issue. The question was, ”(. . .) depends on the (human) reader’s knowledge that an input function might not have return a positive integer (or even any number); but how does the slicer knows this?” (the given example was using integers). When we identify a call to an external function, we add an entry on the output report with a warning, alerting to the fact that a few verifications must be make in order to guarantee that all calls, to an annotated component, that receive value as parameter, will have the contract respected. We recognize this type of functions using all the data structures constructed during the analysis process. If a call is found in a slicing result, but has no entry on the identifier table, then is considered a call to an external function.

Everything discussed until now in this section happen when found a call to an external function. But how about, when the function is on the identifier table and on the repositories? When this happen we have three possible cases:

1. the call we are verifying the contract has no annotation for the parameter with the resulting value of the function call;
2. the found call has no postcondition and the call we are verifying has an annotation for the parameter with the resulting value of the function call;
3. the found call has postcondition and the call we are verifying has an annotation for the parameter with the resulting value of the function call;

In the first case, the result of the found call makes no difference as the parameter has no restrictions of value.

The second case will generate a warning message as we are not able to predict the values of the parameter making impossible to guarantee that the contract will be respected.

The last case force the calculation of the possible values, to be used on the next iterations, using the postcondition. All the information is stored in the objects already seen. These objects are later used to compare the postcondition and precondition annotations regarding a particular parameter in order to find contract violations.

Values vs Precondition This last critical verification occurs every time during of the verification of the statements on the slicing result. Each time the parameter suffers a change, the values it can take must be recalculated. This may look easier than it really is.

If we have an assignment it is pretty easy to calculate the new value but if we have the same assignment inside an `if` block, for example, the complexity increases significantly. We must assure that both values (if the condition is true and if it is not) are used to compare with the call precondition.

Having all this in consideration, we decided to use a flexible list in order to store the list of values the parameter can accept. Every time we found a new path in the code to reach the call we are verifying, we create a new entry on the list with the calculated value. The way we have defined the object, seen in section 4.2, also allow us to compare values with annotations.

In case of violations, these comparisons always lead to error messages. At this point we are able to find contract violations without any doubts so there is no reason to generate warning messages.

4.3. Tool Views

The assessement made to other tools, developed under our group, have shown that a variety of views are in some way needed to work with the tool, and give a better understanding of its results. Following this, the tool provides four different types of views: the Code View, the Identifier Table View, the $\mathtt{SDG}_a$ View and the Slicing View.

The user has access to the Code view (Figure 5) as the default view where has access to the source code that will be used as input in the verification. The Java code is highlighted using scintillaNet [4] library to improve its readability. This is a library that can be imported by Visual Studio, that provides us a special text box where we can define all the color definitions we want to highlight our code.

The Identifier Table view shows the information collected for all symbols in the selected class. The information can be filtered in order to visualize only the details of the symbols in a particular method selected by the user. Figure 6 shows the identifier table of an entire class.

The $\mathtt{SDG}_a$ view shows all the control flow dependencies and data flow dependencies present on the source code. In order to help to visualize which contracts and statements are being violated, we display the SDG_a with such entities colored in red. The Figure 7 shows the representation of a graph by the tool.

The Slicing view (Figure 8) was included to show the result of an intermediate calculation. As the slicing isolates the statements relevant to a particular call (the statement that we are interested to our work), then will probably ease in the understanding and correctness of any error found during the verification.

[4] http://scintillanet.codeplex.com/

Code | ID Table | System Dependency Graph | Slicing

```
1   package teste;
2
3   public class AuxFunctions {
4
5       public AuxFunctions() {
6       }
7
8       /*@ behavior
9          @     requires x > 0 && y > 0;
10         @     ensures sum == x + y;
11         @*/
12      public int sum(int x, int y) {
13          int sum = 0;
14          sum = x + y;
15      return sum;
16      }
17
18
19      /*@ behavior
20         @     requires x > 0 && y > 0;
21         @    ensures sum == x + y;
22         @*/
23      public float sum(float x, float y) {
24          float sum = 0;
25          sum = x + y;
26      return sum;
27      }
28
29      /*@ behavior
30         @     requires x > 0 && y > 0;
31         @     ensures (res > 0)? y : x;
32         @*/
33      public int min(int x, int y) {
34          int res;
35          res=x-y;
36          return ((res > 0)? y : x);
37      }
38
39
40     /*@ behavior
41         @     requires x > 0 && y > 0;
42         @     ensures (res > 0)? x : y);
43         @*/
```

Fig. 5. GamaPolarSlicer Code View

Code | ID Table | System Dependency Graph | Slicing

Identifier	Type	Kind	Owner
AuxFunctions	AuxFunctions	Type	global
sum	int	Method	AuxFunctions
x	int	Parameter	sum
y	int	Parameter	sum
sum	int	Var	sum
sum	float	Method	AuxFunctions
x	float	Parameter	sum
y	float	Parameter	sum
sum	float	Var	sum
min	int	Method	AuxFunctions
x	int	Parameter	min
y	int	Parameter	min
res	int	Var	min
max	int	Method	AuxFunctions
x	int	Parameter	max
y	int	Parameter	max
res	int	Var	max

Methods

All

Fig. 6. GamaPolarSlicer Identifier Table View

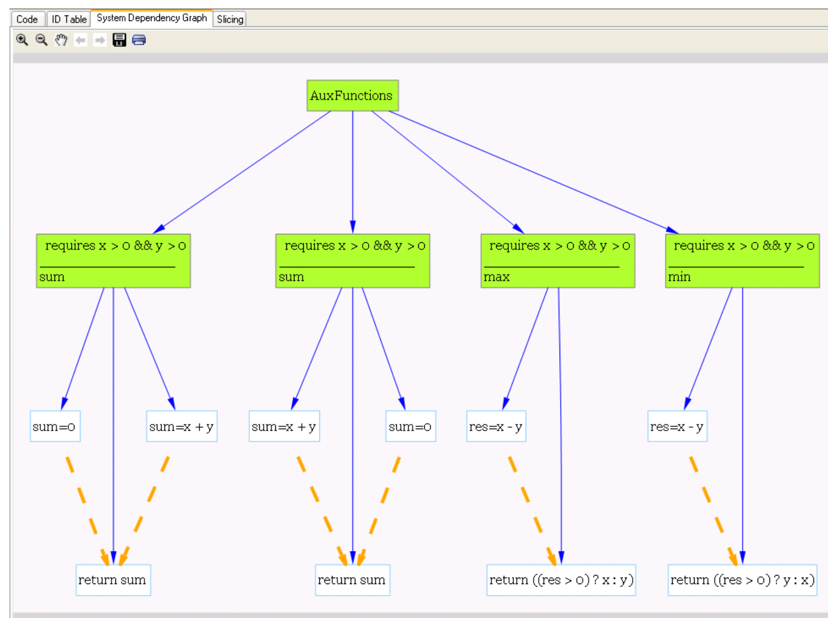


Fig. 7. GamaPolarSlicer SDG_a View

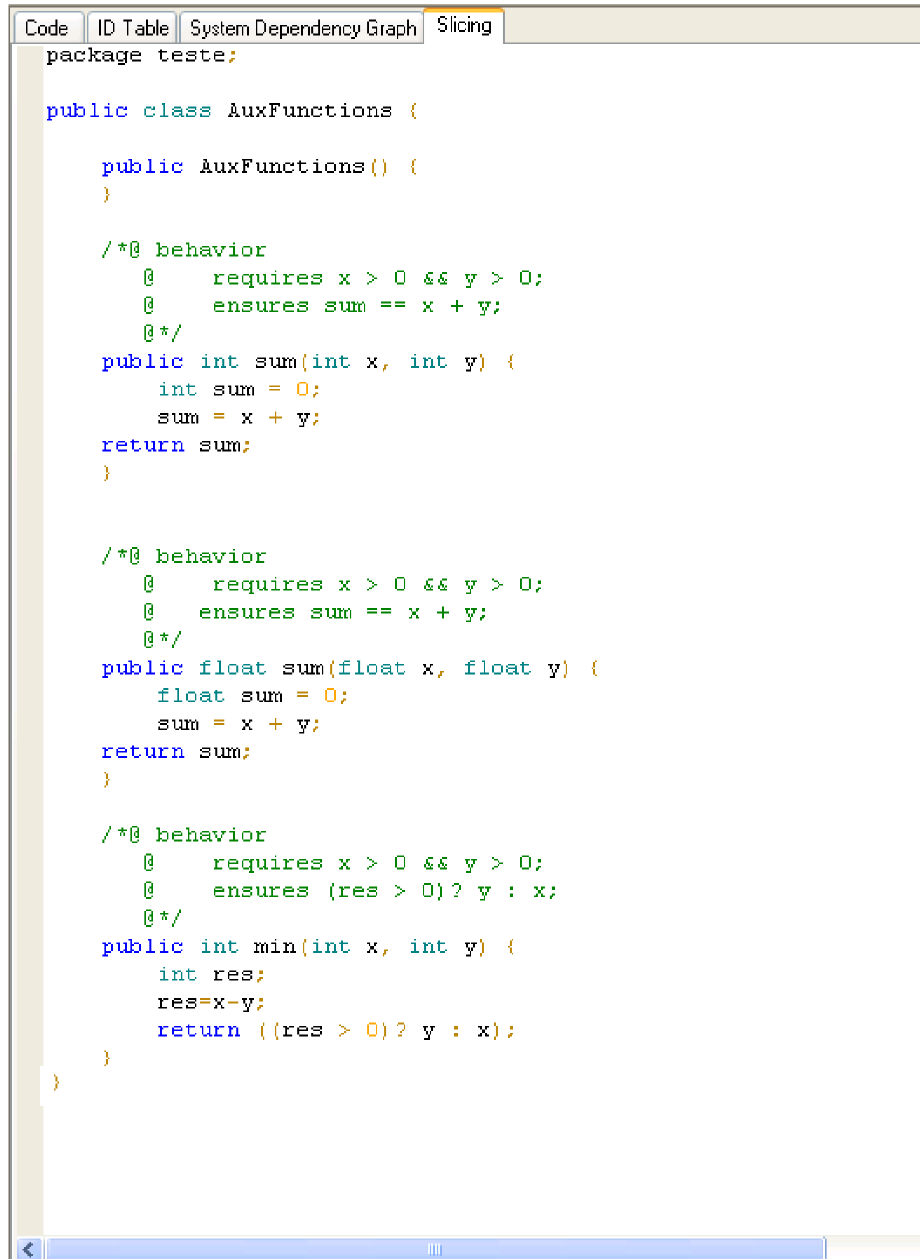


Fig. 8. GamaPolarSlicer Slicing View

5. An illustrative example

To illustrate what we intended to achieve, please consider the Example 2 listed below that computes the maximum difference among student ages in a class. This component reuses other two: the annotated component `Min`, defined in Example 3, that returns the lowest of two positive integers (Figure 5 shows the view of the code provided by GamaPolarSlicer); and Max, defined in Example 4, that returns the greatest positive integer.

Example 2 DiffAge

```
 1: public int DiffAge() {
 2:     int min = System.Int32.MaxValue;
 3:     int max = System.Int32.MinValue;
 4:     int diff;
 5:
 6:     System.out.print(”Number of elements: ”);
 7:     int num = System.in.read();
 8:     int[] a = new int[num];
 9:     for(int i=0; i¡num; i++) {
10:         a[i] = System.in.read();
11:     }
12:
13:     for(int i=0; i¡a.Length; i++) {
14:         max = Max(a[i],max);
15:         min = Min(a[i],min);
16:     }
17:
18:     diff = max - min;
19:     System.out.println(”The gap between max and min age is ” + diff);
20:     return diff;
21: }
```

Example 3 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)? x : y);
5: }
```

Example 4 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: }
```

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 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 5 below, it can not be guaranteed that all integer elements are greater than zero; so a potential precondition violation is detected.

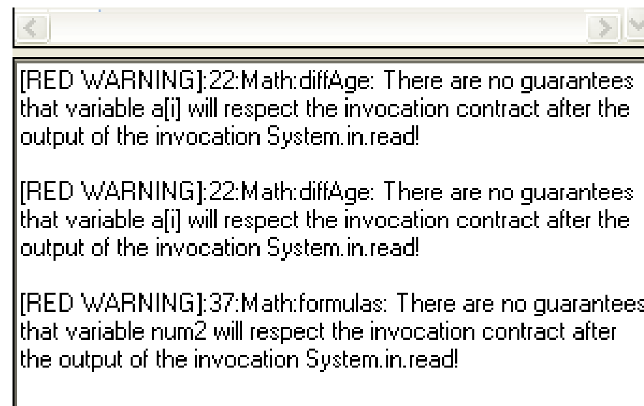
Example 5 Backward Slicing for a[i]

```
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* (Figure 9 shows the output report from GamaPolarSlicer) alerting to the possible invocation of Min with negative numbers (what does not respect the precondition).

6. 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.


[RED WARNING]:22:Math:diffAge: There are no guarantees that variable a[i] will respect the invocation contract after the output of the invocation System.in.read!

[RED WARNING]:22:Math:diffAge: There are no guarantees that variable a[i] will respect the invocation contract after the output of the invocation System.in.read!

[RED WARNING]:37:Math:formulas: There are no guarantees that variable num2 will respect the invocation contract after the output of the invocation System.in.read!

Fig. 9. GamaPolarSlicer Output Report

In [4], *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 [3], *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 precondition (the same present in *p-slice*) and strongest post-condition — they present algorithms for both slicing strategies, backward and forward.

Comuzzi et al [4], and *Chung et al* [3], provide algorithms for code analysis enabling to identify suspicious commands (commands that do not contribute to the postcondition validity).

In [6], *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.

7. Conclusion

As can be seen in section 5, 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.

At the moment, the tool is capable to apply the Caller-based Slicing to a program and compute precise slices. Also the computed slices are displayed by the tool to ease the comprehension of the program by the developer, allowing him to focus on the relevant aspects of the program. This tool is also very useful on the program comprehension on its general.

One of our goals was to check if it was possible to do a contract verification with low computing effort and reasonable precision. This was successfully accomplished but we still need more empirical studies so that we can strengthen our conclusions regarding its efficiency and reliability.

The tool still presents some limitations in the contract verification process. Expand it in order to process annotations regarding any Java data type does not appear to be an easy job and can make the tool to become less accurate.

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/`.

To test the behaviour of our algorithms and the tool output and performance, we selected a collection of diversified programs (medium size and medium complexity). As the objective of that phase was to verify the correctness of the algorithms and its coverage, we were exclusively concerned with the variety of the test cases concerning call paths and the kind of contract annotations. After that phase, that finished successfully, we will be concerned with scalability. For that purpose, a new collection of tests of large size will be used to measure the performance degeneration. We are aware that the biggest difficulty we will face is to find in the industrial or academic worlds huge programs with annotations, but we need to obtain them (even generating the test cases) to fully test GamaPolarSlicer.

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 [2].

References

1. 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. pp. 134–138. Springer-Verlag, Berlin, Heidelberg (2007)
2. Areias, S.: Contracts and Slicing for Safety Reuse. Master's thesis (Dec 2010)
3. 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. pp. 605–609. ACM, New York, NY, USA (2001)
4. 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. pp. 557–575. Springer-Verlag, London, UK (1996)
5. 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)
6. Harman, M., Hierons, R., Fox, C., Danicic, S., Howroyd, J.: Pre/post conditioned slicing. icsm 00, 138 (2001)
7. King, J.C.: Symbolic execution and program testing. Commun. ACM 19(7), 385–394 (1976)
8. Leavens, G.T., Cheon, Y.: Design by contract with jml (2004)
9. 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. pp. 176–185. IEEE Computer Society Press (1993)
10. Meyer, B.: Applying "design by contract". Computer 25(10), 40–51 (1992)
11. Sherif, K., Vinze, A.: Barriers to adoption of software reuse a qualitative study. Inf. Manage. 41(2), 159–175 (2003)
12. Shiva, S.G., Shala, L.A.: Software reuse: Research and practice. In: ITNG. pp. 603–609. IEEE Computer Society (2007), http://dblp.uni-trier.de/db/conf/itng/itng2007.html\#ShivaS07

Sérgio Areias got his degree in "Computer Engineering", at University of Minho (UM) in 2008. In 2010, he finished the M.Sc. with the dissertation "Contracts and Slicing for Safety Reuse", also at University of Minho. Since 2011, he is working at University of Minho under the research project named CROSS (An Infrastructure for Certification and Re-engineering of Open Source Software), and the research team of "gEPL, the Language Processing group".

Daniela da Cruz received a degree in "Mathematics and Computer Science", at University of Minho (UM), and now she is a Ph.D. student of "Computer Science" also at University of Minho, under the MAPi doctoral program. She joined the research and teaching team of "gEPL, the Language Processing group" in

2005. She is teaching assistant in different courses in the area of Compilers and Formal Development of Language Processors; and Programming Languages and Paradigms (Procedural, Logic, and OO). She was also involved in several research projects (CROSS, DSLpc, PCVIA).

Pedro Rangel Henriques got a degree in "Electrotechnical/Electronics Engineering", at FEUP (Porto University), and finished a Ph.D. thesis in "Formal Languages and Attribute Grammars" at University of Minho. In 1981 he joined the Computer Science Department of University of Minho, where he is a teacher/researcher. Since 1995 he is the coordinator of the "Language Processing group" at CCTC (Computer Science and Technologies Center). He teaches many different courses under the broader area of programming: Programming Languages and Paradigms; Compilers, Grammar Engineering and Software Analysis and Transformation; etc. Pedro Rangel Henriques has supervised Ph.D. (11), and M.Sc. (13) thesis, and more than 50 graduating trainingships/projects, in the areas of: language processing (textual and visual), and structured document processing; code analysis, program visulaization/animation and program comprehension; knowledge discovery from databases, data-mining, and data-cleaning. He is co-author of the "XML & XSL: da teoria a prática" book, publish by FCA in 2002; and has published 3 chapters in books, and 20 journal papers.

Jorge Sousa Pinto got his "Docteur de l'École Polytechnique" degree in 2001. Since then he has been a lecturer at the University of Minho, and since 2005 he has served as the deputy director of the Computer Science and Technology Centre at that university. He is the author of about 30 research papers, and a co-author of the textbook "Rigorous Software Development – An Introduction to Program Verification". His research interests comprise both deductive and model checking-based approaches to program verification. He is a senior member of the Association for Computing Machinery.

Received: January 7, 2011; Accepted: April 13, 2011.