Technical Report

Braga, June 29, 2012

GamaPolarSlicer

Sérgio André dos Santos Areias
and Pedro Rangel Henriques
and Daniela da Cruz

University of Minho
Informatics Department

Abstract

The work here described was focus on the study and analysis of software and its components to improve reuse techniques. In software development, it is often desirable to reuse components instead of build it from scratch. In this context, we consider that reusing annotated components it is a rigorous way of assuring the quality of a system under development. The work reported also focused on the study of slicing techniques in order to certify that the integration of an annotated component with a contract, into a system, will preserve the behavior of the former.

The objective of the research done was to prove that the Caller-based Slicing applied to components with contracts allows a safe development based on reuse. At the end, we developed a tool as proof of concept to test our ideas. The tool allows to identify in a given system the annotated components, verifying for each of them if the calls respect its preconditions.

The paper describes the design and implementation of GamaPolarSlicer.

1 Introduction

The software industry has suffered big changes in the last two decades and the number of companies working on this business domain is growing from year to year. Every year a significant number of software systems is developed or improved. The reuse of software components was found to be a way to ease the development process, and at the same time reduce the costs associated to it. It is widely believed that this technique will enable software developers to create bigger and more complex systems with guarantee on the its quality and reliability. Despite the failures to introduce reuse as a systematic process on the software development, many efforts have been made to achieve this goal.

The decision to reuse raises a spectrum of issues, from requirements negotiation to product selection and integration. In order to reuse it is important to know how to choose the component that best fits respecting system requirements. According to the literature, selection of reusable components has proven to be a difficult task [MS93]. Usually, this is due to the lack of maturity on supporting tools that should easily find a component on a repository or library [SV03].

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 [SV03, SS07]. 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 [MS93].

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 [MS93].

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 [Mey92] facilitates modular verification and certified code reuse.

The contract for a component 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 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 annotated components.

Motivated by these promises of reuse and *design-by-contract* we developed a tool that allows us to to verify whether a concrete calling context preserves the precondition of the reused component, and display, whenever possible, a diagnostic or guidelines to correct any violation found during the verification. To achieve the objectives

listed above, 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 presentation of our results proceeds as follows: In the second section we start by reviewing software reuse. The third section introduces the basic architecture of the tool. The fourth section presents the tool technical designs, and how it works. The fifth section describes the user interface. The sixth section describes the results of the case studies applied to our tool. Section seven describes the related work. In the eight, and last section, the conclusions about the research work done are drown.

2 Software Reuse

Software reuse is defined has the process of software development using the knowledge or components from software previously developed. It is widely believe that this can allow considerable decreases on the development cost and also on the assembling and integration efforts [FK05, SS07].

The most common benefits in the literature are the increase of productivity, quality and reliability [FK05, RDKN03, SS07]. The importance of these benefits on the future of the software development lead reuse research to quickly spread, increasing substantially the number of contributions in this area.

Usually, organizations operate in a specific business domain. This fact, makes most of their systems to be variants from others previously developed. This leads to a growth of confidence on reuse benefits, during the development of new systems.

Even being aware of reuse advantages, it is far from its real potential. A few studies have been done in order to realize (i) the best way to apply reuse on the development of new software increasing the efficiency, and (ii) why a seemingly simple concept is so complex in practice.

During the history of reuse were pointed several obstacles to apply it. A few critical factors to achieve a reuse with success were being discussed in literature but while there is agreement in some factors there are some that still need research and a few more discussion. Based on past experiences of reuse, it is believed that top management and software developers are the first barrier to reuse adoption on industry [SV03]. Reuse barriers were divided in four main groups as is believed that these will include the more critical barriers to reuse adoption. These are: Top Management, Technical and Organizational Barriers, Human Issues, and Measurement of Reuse.

It is important to guarantee the reliability of a system with reused components. They can be integrated in thousands of different systems increasing the risk of possible failures [Mey92]. There is a strong belief that Design by Contract(DbC) can increase the reliability and bring several advantages to the process of software development [Fel03, Mey92, LC03, TBmJ06, JM97].

DbC is a designing approach for the construction of software introduced by Meyer

in 1986 [MNM87]. The main idea behind this technique is the development of components together with their specifications.

The contract works in a similar way as a human contract between a client and the supplier. It includes the benefits and obligations for both sides in order to guarantee a correct call (code context). Usually an obligation for one party implies a benefit for the other [Mey92].

These contracts are, usually, formed by assertions (preconditions, postconditions or invariants) that describe the requirements and the return conditions of a software component [Mey92, LC03, dCPH09].

If this assertions are not respected, it means that one of the sides have not fulfilled its contract obligations. We can identify the violator using the type of the violated assertion. If it is a precondition violation then it was the client not respecting the supplier requirements. If it is a postcondition violation then it was the supplier not delivering the agreed return.

The contract theory works in the opposite way of the idea of defensive programming. While the second encourages the development to protect all software modules checking blindly as many times as possible, adding redundancy and increasing software complexity, the first declines a contract that assigns the responsibility for every consistency condition both parties (client and supplier) [TBmJ06, Mey92].

The use of assertions would make unnecessary the use of the redundant checks included in the software packages developed using defensive programming. This is just one of the advantages. We will see more in the next section.

In 1986, Bertrand Meyer designed a language with native support for DbC, Eiffel [MNM87]. This was the first practical application of it.

3 Architecture

Our idea was to help guarantee the correct behavior when integrating an annotated component into a new system reusing it, creating a tool to automate this process. This integration should be smooth, in the sense of that it should not turn a correct system into an incorrect one.

To achieve this verification goal, it is necessary:

- to verify the component correctness with respect to its contract (using a traditional Verification Condition Generator, already incorporated in GamaSlicer[23], 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 focus on the second item, working with preconditions and backward slicing.

Figure 1 shows GamaPolarSlicer architecture, aiming at the easiness of the described process. The architecture is based on the classical structure of a language processor.

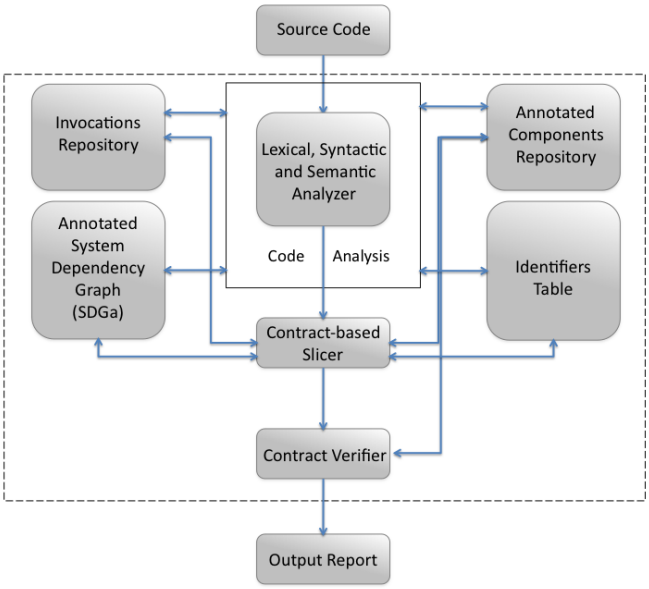

Figure 1: GamaPolarSlicer Architecture

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

Lexical Analyzer, Syntactic Analyzer, Semantic Analyzer the Lexical layer converts the input into symbols that will be later used in the Identifier 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 returned by 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) 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.

Identifier 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 Dependency 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.

Caller-based Slicer 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 Annotated System Dependency Graph this a subgraph of the original Annotated System Dependency Graph, with all the statements relevant to the particular call.

Contract Verifier 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 allow the possibility to provide some suggestions to solve these issues, and a graphic display of the violations over the Annotated System Dependency Graph.

4 Evaluating the Source Code

As depicted in the architecture (see Figure 1), 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 Annotated System Dependency

Graph(SDG_a) has previously said. 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 2), 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.

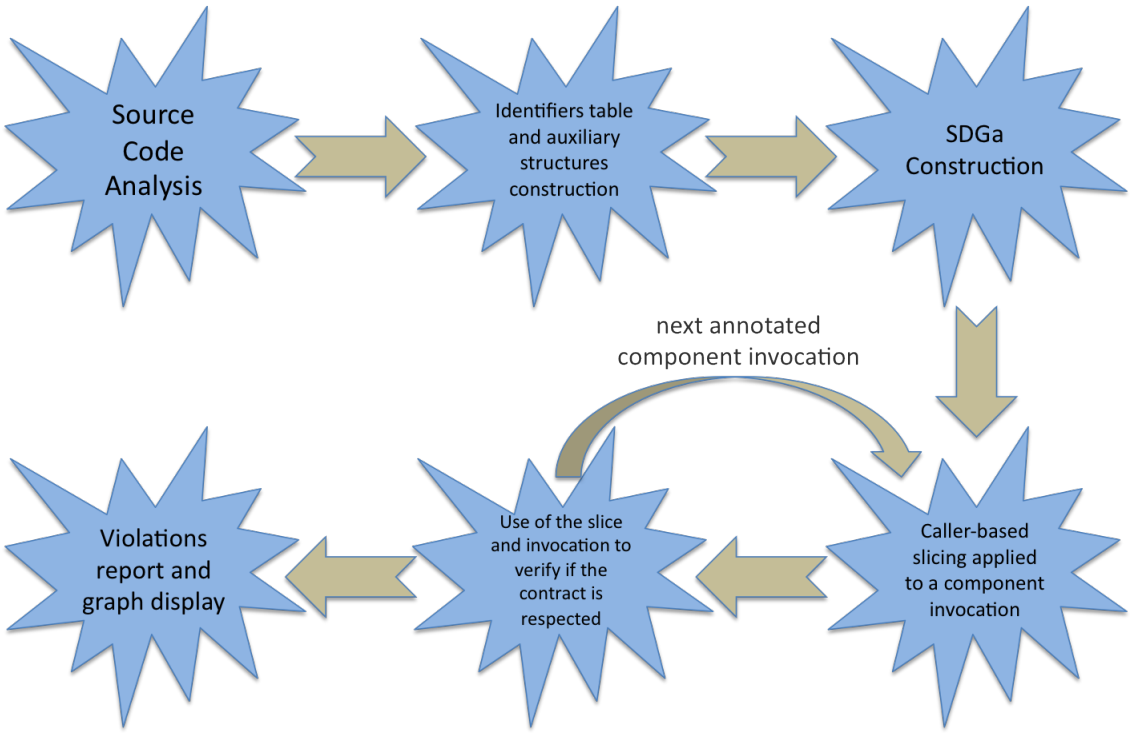

Figure 2: Tool Workflow

4.1 Parsing Data

In order to improve the readability and the efficiency of the code, we decided to use an EBNF(Extended Backus-Naur Form) grammar coded with AnTLR to generate a recognizer to the Java language. During this section we will show how we constructed the various data structures and how we processed the annotations found on the code.

AnTLR is a tool released in 1992 by Terence Parr [1]. AnTLR is a parser generator that uses LL(*) parsing.

The job of AnTLR in this work is to generate a recognizer for the Java language to be later used by the tool to parse the input and construct all the data structures. This recognizer is generated in the C# language to ease the integration of it later on.

The grammar used as input for the AnTLR was firstly developed by Terence Parr and later updated by AnTLR community.

This grammar specifies the Java language with the addition of JML annotations. This improvement was made by Daniela da Cruz [2].

Our contribute to this grammar focused on the semantic actions. Actions to construct the Identifiers Table, the repositories and the SDGa were implemented.

4.2 Constructing the Identifier Table

As usual in this type of work, the Identifier Table is always present and it is used by the tool as knowledge support. All the relevant information concerning the processed symbols is stored here.

The Identifier Table is divided in four different C# classes: `Entry`, `Scope`, `Table` and `MyType`.

The `MyType`, as the name implies, represents the type of a symbol. The table works in a similar way as multiple linked lists distinguished by their scope. Each recognized symbol is treated as an entry, and each entry has a scope associated to it. Each block of statements has a different scope and it is represented with a new linked list hanging on the entry to that block. This would not be necessary if we did not want to store all the symbols in a scope until the program end. Figure 3 illustrates how the table is filled.

Below we will show how we add a class or a method to the table in order to better understand how it works.

```
1 'class' Identifier (typeParameters)?     {
2         classEntry = $symTab.insert(Entry.Typ, $Identifier.text, new
              MyType(MyType.Class, MyType.voidType, $Identifier.text),
              $isPublic, $isStatic, $isFinal, owner);
```

Listing 1: Add an entry to the table

To a class, first we extract the name of it using the terminal symbol `IDENTIFIER`. With this information we create an entry in the table for this class. After that and before processing the symbols in the class body, we open a new scope in the table for the class (as seen before, each class has a different Identifier Table).

[1] http://www.antlr.org/wiki/display/ admin/Home
[2] http://alfa.di.uminho.pt/ danieladacruz/

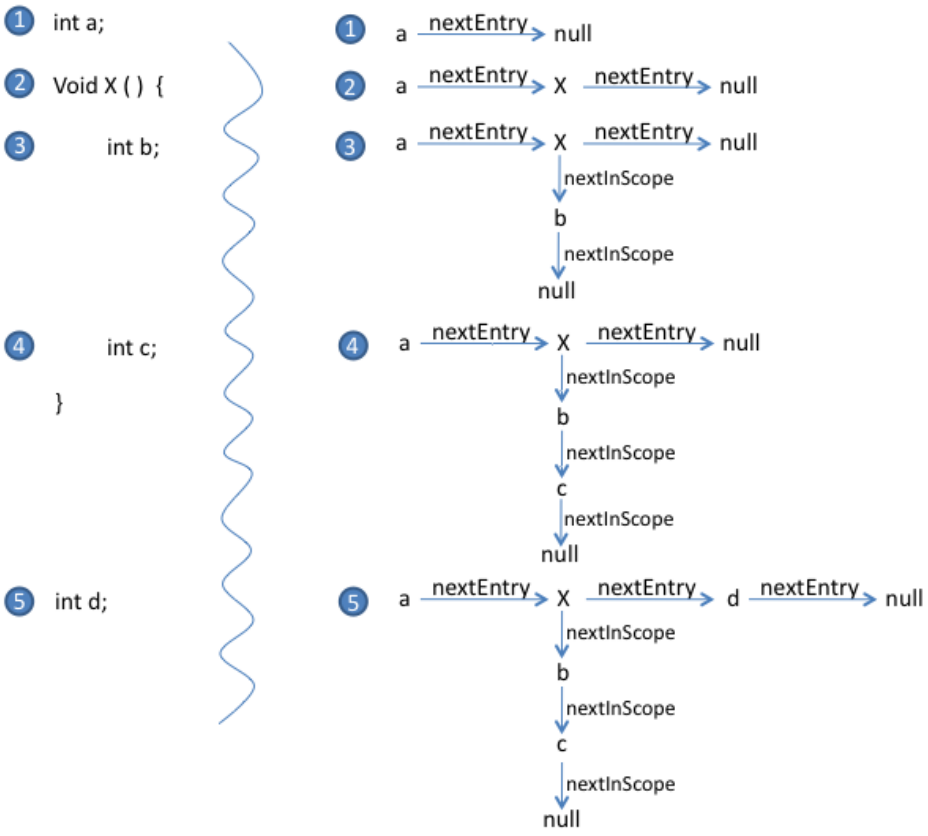

Figure 3: Identifier Table built in memory

```
1 ('extends' type)?
2 ('implements' typeList)?
3         { $symTab.openScope(0); }
```

Listing 2: Open a new scope

Now we just need to make sure that the grammar production that processes the class body inherits the class owner, i.e. , the owner of the following statements of the code (`classEntry`).

```
1 outer=classBody[symTab, classEntry, inGraph]
```

Listing 3: Inheritance of the owner class by the production for the class body

In the case of a method, we begin to identify its modifier. As we can have different ways to declare a method, we also have different productions on the grammar to recognize them. We will present here only one, as there are only a slightly changes between them.

After the modifiers, we have the return type of the method. This is another information that the production to recognize the method declaration, must include as inherit attributes.

```
1 :   type!        { outType = $type.outType; $inMethod.ret.type = $type.
      outType.name; }
2         out1=methodDeclaration[isPublic, isStatic, isFinal, outType,
              symTab, owner, inMethod]
```

Listing 4: Method return type

Now we just need to recognize the method name in order to have all the necessary info to create an entry on the table for this method.

```
1 :   Identifier { name = $Identifier.text;
2         method = $symTab.insert(Entry.Mth,$Identifier.text,type,
              $isPublic,$isStatic,$isFinal,owner);
```

Listing 5: Create a table entry for a method

With the entry created, the rest of the process is similar to what we already seen when adding a class; We open a new scope and put the entry as an inherit attribute in the production that recognizes the method body.

At the end, the table will have all the necessary info from all symbols recognized during the input analysis, and this information will remain stored until the end of processing.

4.3 Repositories

As the analysis and the contract verification are not simultaneous phases, the role of both, Annotated Components Repository and Invocations Repository, is crucial

in the development process.

The intersection between these repositories give us the set of calls to which the contract verification will be applied. Also, during the contract verification the Annotated Components Repository is used to provide the precondition and postcondition in cases of invocations found in the slicing result. We will explain it in more detail later in the document (see section 4.6).

To add a new call to the Invocation Repository, involves merely a verification to the number of arguments of the call. If an invocation has no arguments then is discarded since does not follow the purpose of this work.

```
1 |   a=Identifier ('.' e=Identifier)* (c=identifierSuffix[idTree])?
2 {
3     if(null != $c.outArgsList) {
4         Call node = new Call($a.text + auxcode,"",$a.line,"","");
5             node.owner = call_owner;
6            node.call_line = call_line;
7            invocationsList.Add(node);
8     }
```

Listing 6: Add a new call to the Invocation Repository

To add a new annotated component to the Annotated Components Repository is very similar. There is just the need to verify if the recognized method also includes a JML specification, otherwise it is discarded. It is only required the existence of a pre condition in the JML specification. The existence of a postcondition is not mandatory in the context of GamaPolarSlicer.

```
1 |  (a=jmlMethodSpecification?) modifiers
2 outerM=memberDecl[$modifiers.isPublic, $modifiers.isStatic, $modifiers.
      isFinal, symTab, owner, inMethod]
3 {
4     if ($outerM.isMethod)
5     {
6         $numMethods = 1;
7         if(null != $outerM.outMethod)
8         {
9             if(null != a) {
10                $outerM.outMethod.precondition = $a.pre;
11                $outerM.outMethod.poscondition = $a.post;
12                $outerM.outMethod.isAnnotated = true;
13                annotatedComponents.Add($outerM.outMethod);
14        }
```

Listing 7: Add a new annotated component to the Repository

4.4 Constructing the Annotated System Dependency Graph

The construction of this structure is much more complex than others seen before. The decision about the structure to use fell on the adjacency list due to the flexibility

it allows at the time of its implementation. Our implementation differs a bit from the usual as the nodes are not all as entries on the list, instead we only have entries for the nodes representing methods.

As the code can have methods with the same name (polymorphism), we decided to use the line to create the key to the node in the list. For example, if we have two methods with the name `Sum`, one in the line six, and the other on the line twelve, the key to the first node would be `Sum:6`, and to the second `Sum:12`.

As in a SDG_a we can have annotations associated to nodes, then the node for a method must store the precondition and the postcondition, in the case of have them.

Each method node has a list of all nodes connected with a control dependency. This type of nodes are a bit different of the one for the methods. These nodes take advantage of Object-Oriented inheritance. All type of statements are represented as `Node` (abstract class) and when needed they are converted to their type of nodes: `Loop`, `Conditional`, `Call`, `Assignment`, etc..

Nodes that represent a block of code, like a `Loop` node or a `Conditional` node, include a list that contains all the statements on the block which in turn are also of type `Node`.

Figure 4 illustrates how the graph is created in memory from the source code.

Until this point we have presented how the graph was implemented. Now we will show how we built it with a grammatical specification.

Most of the updates on the graph occur at the same time that occurs for the Identifier Table, and the implementation is quite similar with a few exceptions that we will explain with more detail.

4.5 Assignment or Variable Declaration

Assuming the grammar:

```
Z----->Y   Z
Y----->X   Y
X----->A   X
A----->a
```

To reach the symbol X, the parser will have always to check the productions Z and Y. That is what happens in our grammar with the assignment and calls. The way the grammar was developed, forces the recognizer to check the productions for an assignment or for a variable declaration every time a call is being processed.

As the call and the assignment are different types of nodes and have a particular data type, then this bring us a little inconvenient.

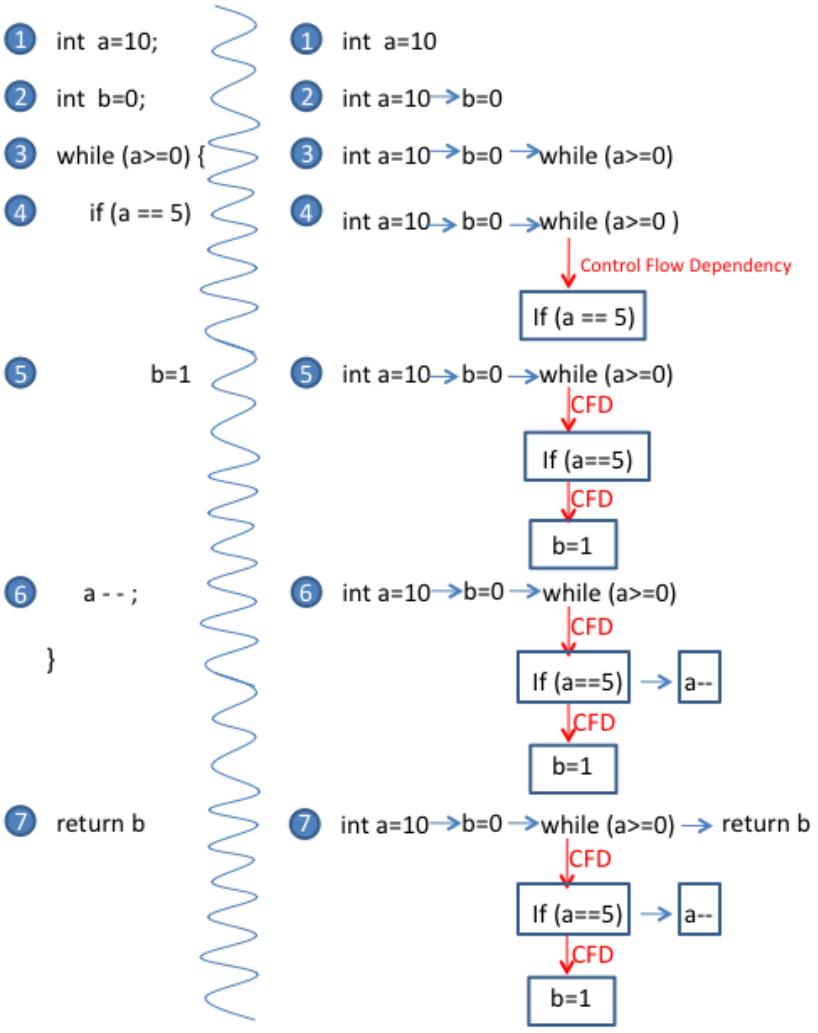

Figure 4: Graph built in memory

When we have a statement that is a call or is an assignment with no calls on its right side, then there is no problem. The problem is when we have an assignment with a call on the right side. When that happens, we want to create a node of the type `Call` instead of a node of `Assignment type`. The left side of the assignment is stored in the `Call` node as the destination symbol of the call result.

To do this, we begin to increment the number of possible assignments every time the production for assignments (`AssignExpr`) is checked. We do this to avoid the creation of multiple assignments node for the same statement. As we are just interested to have the leftmost assignment of the statement, and using the tree of the `AssignExpr` production we can reach it again, then this counter will give us the guarantee that we always know who is the leftmost assignment.

```
1 assignExpr returns [string value, string outCode, int line, Node
    outNode]
2     :     {isAssign++;} c=conditionalExpression (a=assignmentOperator
            b=assignExpr)?
```

Listing 8: assignExpr production

When we find the leftmost assignment there is the need to verify if the right side is a call or not. If it is a call we ignore the assignment node as we already have a node attribute to synthesize. If it is not, we have to create an `Assignment` node to assign to the attribute node to synthesize.

```
1 if(1 == isAssign) {
2     if($b.outNode != null) {
3         ((Call)$b.outNode).ret.var = $c.outCode;
4         $b.outNode.codeline = $c.outCode + $a.outCode + $b.outCode;
5         $outNode = $b.outNode;
6     }
7     else {
8         $outNode = new Assignment($c.outCode,$c.outCode + $a.outCode +
              $b.outCode,$c.line,$c.outCode,$b.outCode,$a.outSymbol);
9     }
10 }
```

Listing 9: Assignment nodes

But we can have more expressions beside calls and assignments. All those expressions are considered of the type `Expression`. These nodes are created every time the `AssignExpr` has the synthesized attribute node empty.

```
1 expression returns [string value, Node outNode, String outCode, int
    line]
2     :   a=assignExpr {
3             if(null == $a.outNode) {
4                 $outNode = new Expression($a.outCode,$a.outCode,$a.line
                      ,$a.outCode);
5             }
6             else {
```

```
7                   $outNode = $a.outNode;
8               }}
9       ;
```

Listing 10: Expression nodes

4.5.1 IF/ELSE IF

In practical terms the `if/else if` structure is very similar to the case structure. Multiple comparisons and the default case that can be taken as the final `else`.

Again, the way the grammar was designed brought us a few drawbacks. In the grammar there is no distinction between an `if/else` and an `if/else if` statement. We had to find a way to distinguish both statements in order to create the right node. In the graph, the node for an `if/else if` statement is represented in the same way as the node for `switch` statement.

Every time the production for and `if` statement is called, it is made a verification to check if this `if` is after an `else` making it an `else if` (`else if flag`). Also, we count how many times we do this verification in order to distinguish every time the production is used for an `if` or for an `else if`.

```
1 |     'if' {
2           if(true == isParentIf) {
3               isElseIf = true;
4           }
5           isTheIf++;
6       }
```

Listing 11: If or Else If verification

After that we must check the `if` body and the `else` statement, in case of it. Here there are a few things that need to be processed and verified as we have to decide between a `Switch` node and a `Conditional` node. After the recognizing of the `if` body, we verify if there is an `else` after it. If the `else` is defined and is the final `else` (`else flag`) we prepare all the `else` information and add it to the list of all `else/else if` statements found consecutively.

```
1 parExpression e=statement[symTab,owner] (options {k=1;}:t='else' {
    isParentIf = true;} f=statement[auxTab,auxOwner])?
2 if(true == $f.isElseOnly) {
3     Condition else_cond = new Condition("_DEFAULT",$t.line);
4     if(null == $f.outNodeList && null != $f.outNode) {
5         else_cond.addCaseLine($f.outNode);
6     }
7     else if(null != $f.outNodeList) {
8         else_cond.casecode = $f.outNodeList;
9     }
10    elseIfCode.Add(else_cond);
```

```
11 }
```

Listing 12: Final else verification

If the `else if flag` is activated and we are in the presence of the `if` statement, then we must create a `Switch` node.

```
1 if(true == isElseIf) {
2     if(1 == isTheIf) {
3         elseIfCode.Reverse();
4         $outNode = new Switch($parExpression.text,"",$parExpression.
              line,"_IF",elseIfCode);
5         elseIfCode = new ArrayList();
6     }
7 }
```

Listing 13: Else if node construction

If the `else if flag` is not activated we just need to check if we have to create a `Conditional` node with or without `else`. The `else flag` is now activated to alert for the end of the `if` statement.

```
1 if(null != f) {
2     $outNode = new Conditional($parExpression.text,"",$parExpression.
          line,$parExpression.text,auxcode,auxcode2);
3 }
4 else {
5     $outNode = new Conditional($parExpression.text,"",$parExpression.
          line,$parExpression.text,auxcode,new ArrayList());
6 }
7 $isElseOnly = true;
```

Listing 14: Conditional node

At the end of all these verifications we must decrease the counter, otherwise all `else if` statements will be ignored during the parser.

```
1 isTheIf--;
```

Listing 15: Decrease number of if/else if statements found

4.6 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 chapter:

- **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.

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:      }
```

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.

4.6.1 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;
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.

4.6.2 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. Line 10 of the example ?? (chapter ??) is an example of a call to an `I/O` function, and possible contract violation.

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.

4.6.3 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 4.6.1, 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.

5 Graphical Interface

The interface was developed aiming at easing the readability and to give a better understanding and visualization of the contract violations in a project. The tool was developed resorting to Windows Forms. Figure 5 shows the GamaPolarSlicer graphical interface.

The interface is divided into three windows: two small windows located on the left and one main window on the right, with four tabs in it.

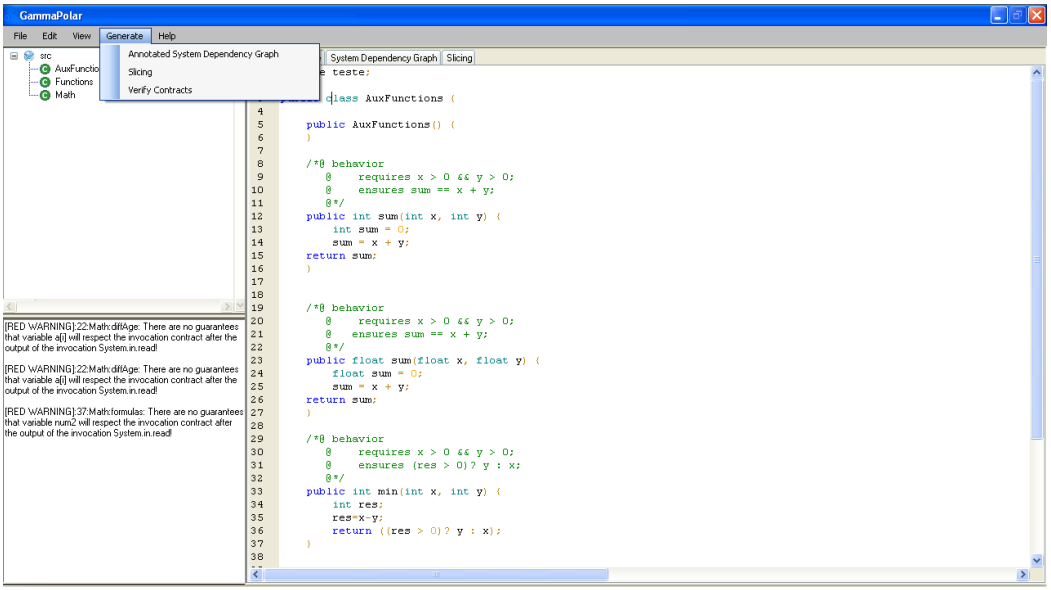

Figure 5: GamaPolarSlicer Graphical Interface

The tool provides a tree view to show all the components of the loaded project, and a text box to present the violations found during the verification. It also provides four type of views: the Code View, the Identifier Table View, the SDG_a View and the Slicing View.

All these tool components will be presented in more detail in the remainder of this section.

The tool provides an easy way to navigate within the project. A tree view, displayed on the top left window, is available with all the classes, packages or folders arranged hierarchically as we can see in Figure 6.

If a double click is issued on a tree item, the info (Code, Identifier Table, etc.) of the selected class will be loaded and presented on the four tabs of the main window on the right side. Not all the info will be loaded as the slicing information or the SDG_a needs the user request.

By default (after loading a software system and before selecting a tree item) the main window displays the information related to the first item of the tree.

The user has access to the Code view as the default view (the first tab on the right

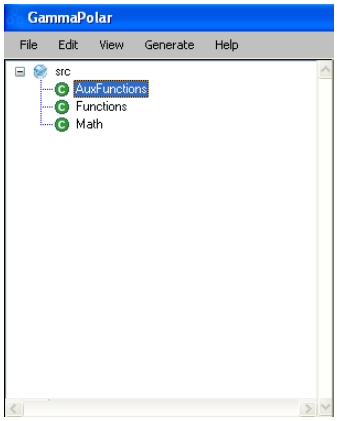

Figure 6: GamaPolarSlicer Tree view of the project content

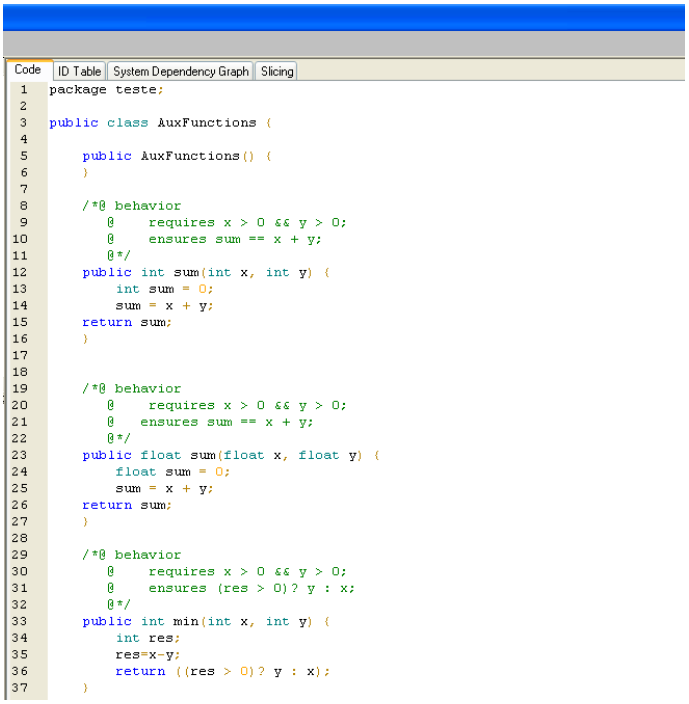

Figure 7: GamaPolarSlicer Code view

main window). The Java code is highlighted to increase its readability. To do this we used scintillaNet [3]. 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.

Figure 7 shows how a code fragment looks like when imported to our tool.

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 8 shows the identifier table of an entire class on the top, and the information of a single method after filtering on the bottom.

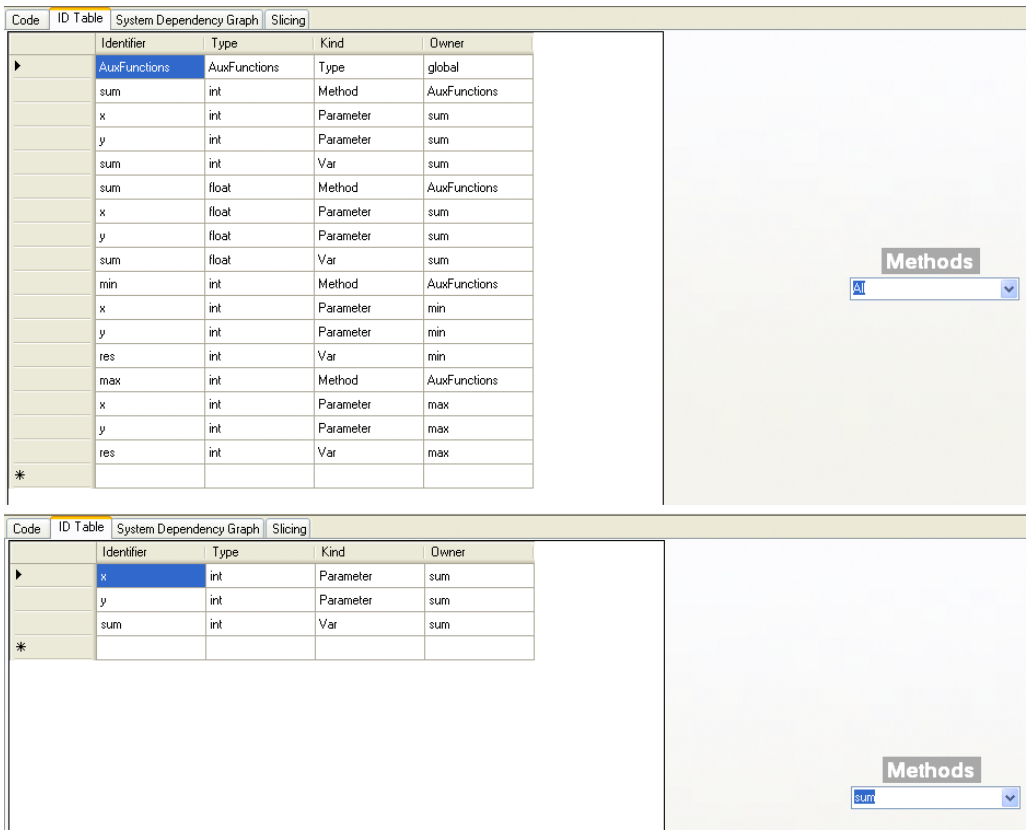

Figure 8: GamaPolarSlicer Identifier Table view of a class and a method

Finally when the user requests a verification of the components contracts, if one or more violations are detected the error message will be displayed in the left bottom window; the tool will present a textual description for each violation, marking their position on the source code (Figure 9). In the future, we intend to highlight all violations on the source code (first tab), and highlight them also on the SDG_a diagram (third tab).

It is almost certain that the interface will suffer a few more changes in the future,

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

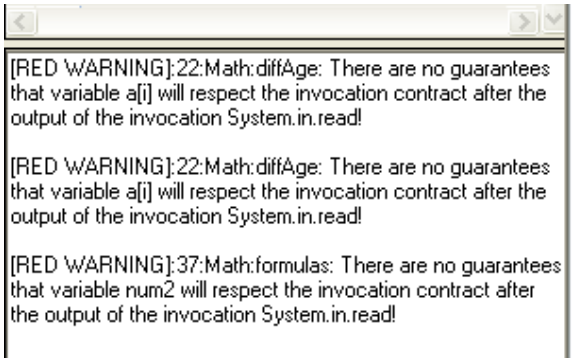

Figure 9: Contract violations alert

besides those already mentioned, to be as similar as possible with the others already developed within our group.

6 Testing the Tool

To test the performance of the tool we have used four case studies. Three of them are packages extracted from online repositories -Linked Queue, Dinning Philosophers, Bounded Buffer-. Most of the components in these packages have JML specifications.

The fourth package -Math Functions- was developed by us with a few JML specifications to test the answer to most of the contract verifications that the tool does.

The reasons that lead us to choose these packages, instead of others, were the complexity and the variety of JML specifications. In this section we will show in detail these packages and the respective results given by the tool.

6.1 Linked Queue

The first package contains the implementation of a linked queue with concurrency. The components in it are not very complex and the JML specifications are formed by a low average number of annotations (see Example 2).

The package is formed by four classes (see Figure 10): `LinkedQueue`, `Process`, `LinkedQueueDriver` and `LinkedNode`. In Figure 10 we can see all the symbols in one of the classes.

One of the important objectives is to see how the tool answers to the contract verification requests.

Errors As can be seen on the Figure 11, the report contains a warning because the context of one of the calls does not guarantee that its contract will be respected.

Example 2 Example of annotated methods from the Linked Queue package

```
/*@ behavior
   @    ensures true;
   @*/
public Object peek() {
    synchronized (head) {
        LinkedNode first = head.next;
        if (first != null)
            return first.value;
        else
            return null;
    }
}
```

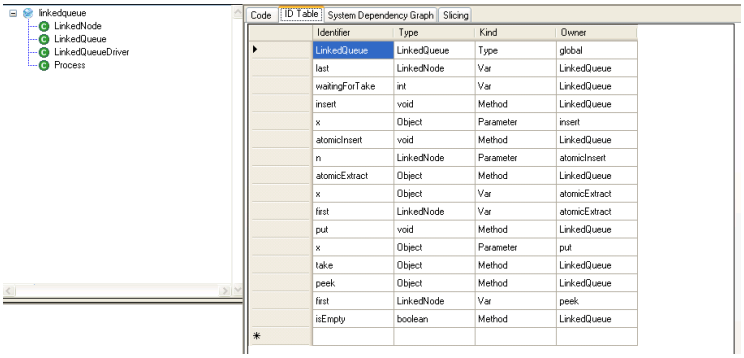

Figure 10: Identifier Table for the class LinkedQueue

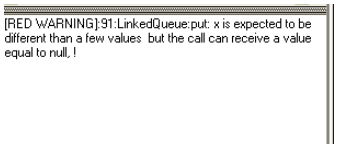

Figure 11: Violations found during the verification of the Linked Queue project

Performance (time) The time used by the tool to load and parse (create all data structures) the package is close to one second. The contract verification is almost immediate.

6.2 Dinning Philosophers

The second package contains the implementation of the dinning philosophers concurrency problem, stated by Edsger Dijkstra. This problem is often used to illustrate the problem of deadlock in a system.

The dinning philosophers consists in a circular table where philosophers can only be eating or thinking, and these actions can only be done one at a time. Each philosopher has necessarily a fork on his left and another on his right. To eat, they need to have both forks in their hands.

Compared to the first case study, the components in this one are very similar regarding the complexity, but the JML specifications have a significant increase on the average number of annotations and on their complexity (see Example 3).

Example 3 Example of annotated methods from the Dinning Philosophers package

```
/*@ behavior
   @   assignable this.numPhils, this.checkStarving;
   @   ensures this.numPhils == numPhils &&
   @           this.checkStarving == checkStarving &&
   @           \fresh(state) && state.length == numPhils;
   @*/
public DiningServer(int numPhils, boolean checkStarving)
{
    this.numPhils = numPhils;
    this.checkStarving = checkStarving;
    state = new int[numPhils];
    for (int i = 0; i < numPhils; i++)
    {
        state[i] = THINKING;
    }
}
```

This package is formed by three classes (see Figure 12): `DinningPhilosophers`, `DinningServer` and `Philosopher`. In Figure 12 we can see the Identifier Table of one of these classes.

Errors No violations found and immediate answer by the tool.

Performance (time) Concerning the time used by the tool to load and parse the whole package, compared with the first case study, it needs nearly one more

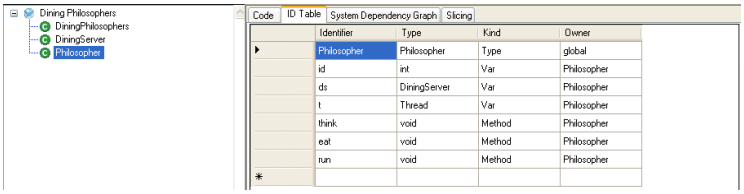

Figure 12: Identifier Table for the class Philosophers

second to do it. The contract verification is immediate. This due to the fact that no violations were found during it.

6.3 Bounded Buffer

The third package contains the implementation of the well-known bounded buffer data structure.

Concerning the components complexity, it increases a bit compared to the two cases studies seen before. Regarding JML specifications, the average number of annotations is lower compared with the second case study, but their complexity is bigger (see Example 4).

Example 4 Example of annotated methods from the Bounded Buffer package

```
/*@ behavior
    @   when count != 0;
    @   assignable buffer[*], takeOut, count;
    @   ensures takeOut >= 0 && takeOut < numSlots &&
    @           \result != null;
    @*/
 public synchronized Object fetch() {
     Object value;
     while (count == 0) {
         try {
             wait();
         }
         catch (InterruptedException e) {}
     }
     value = buffer[takeOut];
     takeOut = (takeOut + 1) % numSlots;
     count--; // wake up the producer
     if (count == (numSlots - 1)) {
         notify();
     }
     return value;
 }
```

This package is formed by four classes (see Figure 13): `BoundedBuffer`, `Consumer`, `Producer` and `ProducerConsumer`.

Figure 13 shows the Identifier Table constructed using the information in one of these classes.

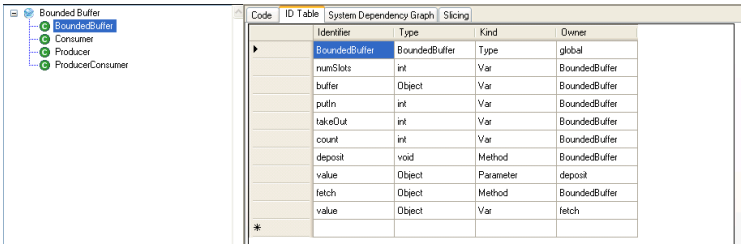

Figure 13: Identifier Table for the class BoundedBuffer

Errors No violations found and immediate answer by the tool.

Performance (time) The time used by the tool to load and parse the package is very similar to the time used in the second case study. It also consumes approximately one more second than the first. Relating to the contract verification, occurs the same as in the second case study.

6.4 Math Functions

Last but not least, the fourth package (the one developed by us) contains the implementation of some fairly common mathematical functions. Its components complexity is smaller compared to the other three cases studies. The number of statements per component is bigger, but quiet simple. Regarding JML specifications, they are formed by basic annotations with low complexity (see Example 5).

With this case study, we are more interested to test the performance of the tool when doing the contract verification instead of the performance when loading and parsing the package. Summarizing, compared to the other case studies the complexity is smaller but the number of violations is bigger.

This package consists in three classes (see Figure 14): `AuxFunction`, `Functions` and `Math`. Figure 14 shows the Identifier Table constructed using the information in one of these classes.

Errors In Figure 15 we can see the output report provided by the tool with all the violations found during the verification. The type of these violations differs which implies different checks, i.e., the tool has used different algorithms to each different type of violation found (see chapter 4 section4.6). Some violations found are due to I/O calls, and other due to conflicts in the preconditions comparison.

Example 5 Example of annotated methods from the Math Functions package

```
public static int diffAge() {
    int min = System.Int32.MaxValue, max = System.Int32.MinValue;
    int num, i, diff;
    int[] a;
    AuxFunctions aux_func;
    System.out.print("Number of elements: ");
    num = System.in.read();
    for(i=0; i<num; i++) {
        a[i] = System.in.read();
    }
    for(i=0; i<a.Length; i++) {
       max = aux_func.max(a[i],max);
       min = aux_func.min(a[i],min);
    }
    diff = max - min;
    System.out.println("The difference between the greatest "
                       + "and the smallest ages is " + diff);
    return diff;
}

/*@ behavior
   @    requires a >= 0;
   @    ensures square = sqrt a;
   @*/
public double sqrt(int a) {
    double square = Math.sqrt(a);
    return square;
}
```

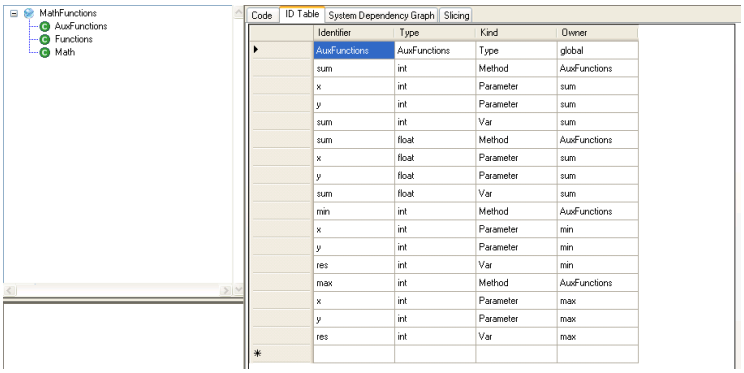

Figure 14: Identifier Table for the class AuxFunctions

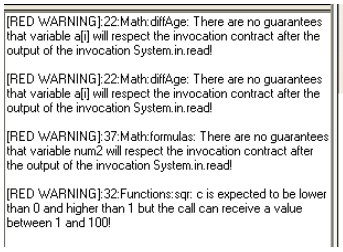

Figure 15: Violations found during the verification of the Math Functions project

Performance (time) Even with the increase of violations, the tool responded well and the process continued to be almost immediate.

7 Related Work

For many years techniques to support DbC were developed [WPF+10, FLRL00, NE02, DF06]. One of the most important type of tools developed are those that do inference of specification from the source code. A well-known tool for dynamic inference of invariants is the Daikon [Ern00, ECGN99].

As any other dynamic approach, Daikon relies on the quality of the tests to generate the candidates, what makes it not totally reliable [DF06]. Even so it presents positive results as shown by Nimmer and Ernst in [NE02]. They have shown that even with a few limitations, the specifications dynamically inferred by Daikon are reliable enough to be machine verifiable.

Another important tool is the ESC/Java [DRL+98], but to make static verification of annotations. This is one of the most capable tools to make modular checking of executable code.

ESC/Java presents many limitations and only detects a few types of errors [NE02]. One of the reasons that leads to avoid use of ESC/Java by the developers, is the fact that leaves to them the responsibility to write the annotations. This was one of the reasons that lead to the development of Houdini [FLRL00].

Houdini is an annotated assistant for ESC/Java. Analyzing the executable code, it generates a set of candidate assertions referred from the unannotated program, and uses ESC/Java to check them. This is done as many times needed to reach a fixpoint. Each cycle means that ESC/Java found invalid annotations that are immediately removed by Houdini from the program. This result is then given to ESC/Java and a new cycle begins.

The use of Houdini decreases significantly the time needed by ESC/Java to check an unannotated program, and also makes the result more reliable as reduce the number of false alarms by ESC/Java. All this as a pay off. In exchange, Houdini consumes lot of computing effort, and that is why it is used mainly for debugging instead of being used for code generation with safety certification.

In [DF06], Denney and Fischer concluded that with a new approach they can improve the results of this kind of tools. Their approach presented better results in respect to flexibility, extensibility and necessary effort. This even with a few deficiencies in their system.

In the last years, has been a considerable progress in this area. An example of it is the AutoFix project, particularly one of its components, the tool AutoFix-E [WPF+10] that is somehow related to our work.

Basically, this tool proposes solutions to a software fault and validates them relying on the contracts present on the software.

Its results have shown to be very promising. 10 data structures classes were used to the experiments. 42 faults were detected and from those 16 were fixed automatically Several of those fixes were identical to fixes proposed by experts.

8 Conclusion

This document describes a technique to integrate reusable components in a new system resorting to specifications associated with components. To support our beliefs we developed GamaPolarSlicer a tool that applies this technique. 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.

We are glad to conclude that our tool is capable not only to verify the components contracts, but also to do it with efficiency and without harming the comprehension of the program by the developer, based on the positive feedbacks that we received from different persons, from experienced users to reviewers and conference participants.

Our goal was to check if it was possible to do a contract verification with low computing effort and reasonable precision, and taking into account the obtained results, this was successfully accomplished. In the considered case studies, the tool presents small response times. Our work still needs more empirical studies so that we can strengthen our conclusions regarding its efficiency and reliability. Due to the fact that we still did not implement the SDG_a visualization, it was not possible for us to verify if it helps or not to understand the found flaws. This was another goal of this tool.

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. It is trivial when the annotations specify that values of variables can or cannot be equal to `null`. More than that, it becomes very hard to verify them and the tool becomes less accurate.

The algorithm presented in Section 4.6.3, “value vs precondition”, is not yet finished. The computation of the value of a variable usually needs other variables values. One

of the barriers is that we considered each component as a standalone component, that is, without any dependence with any other component. Any variable dependent on the component parameters will have its value limited by the annotations in the precondition. If the precondition has not annotations related to the parameter in question, then is very, very difficult to compute the variable value with precision. These dependencies between variables also force us, if we want to compute the value of a variable, to compute the values of all its dependencies using. These calculations could compromise the efficiency of application.

Another problem is that `while` construction can have statements that influence the control predicate in any position of its body. As the grammar treats the body as a black box, then in these cases we are not able to compute values that are dependent of a loop computation. This adds some inaccuracy to the system.

To sum up, we could say that our approach shows very mature results and we intend to improve them even more with some new features.

References

[dCPH09] Daniela da Cruz, Jorge S. Pinto, and Pedro R. Henriques. Reuse of annotated components. 2009.

[DF06] Ewen Denney and Bernd Fischer. A generic annotation inference algorithm for the safety certification of automatically generated code. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 121–130, New York, NY, USA, 2006. ACM.

[DRL+98] David L. Detlefs, K. Rustan, M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. In *SRC Research Report 159, Compaq Systems Research Center*, 1998.

[ECGN99] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 213–224, New York, NY, USA, 1999. ACM.

[Ern00] Michael D. Ernst. *Dynamically Discovering Likely Program Invariants*. Ph.D., University of Washington Department of Computer Science and Engineering, Seattle, Washington, Agosto 2000.

[Fel03] Yishai A. Feldman. Extreme design by contract. In *XP'03: Proceedings of the 4th international conference on Extreme programming and agile processes in software engineering*, pages 261–270, Berlin, Heidelberg, 2003. Springer-Verlag.

[FK05] W. B. Frakes and Kyo Kang. Software reuse research: Status and future. *Software Engineering, IEEE Transactions on*, 31(7):529–536, 2005.

[FLRL00] Cormac Flanagan, K. Rustan M. Leino, K. Rustan, and M. Leino. Houdini, an annotation assistant for esc/java, 2000.

[JM97] Jean-Marc Jézéquel and Bertrand Meyer. Design by contract: The lessons of ariane. *Computer*, 30:129–130, 1997.

[LC03] Gary T. Leavens and Yoonsik Cheon. Y.: Design by contract with jml, 2003.

[Mey92] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.

[MNM87] Bertrand Meyer, Jean-Marc Nerson, and Masanobu Matsuo. Eiffel: object-oriented design for software engineering. In *Proc. of the 1st European Software Engineering Conference on ESEC '87*, pages 221–229, London, UK, 1987. Springer-Verlag.

[MS93] N. A. M. Maiden and A. G. Sutcliffe. People-oriented software reuse: the very thought. In *Advances in Software Reuse - Second International Workshop on Software Reusability*, pages 176–185. IEEE Computer Society Press, 1993.

[NE02] Jeremy W. Nimmer and Michael D. Ernst. Automatic generation of program specifications. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 229–239, New York, NY, USA, 2002. ACM.

[RDKN03] Marcus A. Rothenberger, Kevin J. Dooley, Uday R. Kulkarni, and Nader Nada. Strategies for software reuse: A principal component analysis of reuse practices. *IEEE Trans. Softw. Eng.*, 29(9):825–837, 2003.

[SS07] Sajjan G. Shiva and Lubna Abou Shala. Software reuse: Research and practice. In *ITNG*, pages 603–609. IEEE Computer Society, 2007.

[SV03] Karma Sherif and Ajay Vinze. Barriers to adoption of software reuse a qualitative study. *Inf. Manage.*, 41(2):159–175, 2003.

[TBmJ06] Yves Le Traon, Benoit Baudry, and Jean marc Jézéquel. Design by contract to improve software vigilance. *IEEE Trans. Softw. Eng*, 32:2006, 2006.

[WPF+10] Yi Wei, Yu Pei, Carlo A. Furia, Lucas S. Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. In *ISSTA '10: Proceedings of the 19th international symposium on Software testing and analysis*, pages 61–72, New York, NY, USA, 2010. ACM.