Towards GUI Functional Verification using Abstract Interpretation

Abdulaziz Alkhalid and Yvan Labiche
Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada

Keywords: Abstract Interpretation, Static Analysis, GUI Verification.

Abstract: Abstract interpretation is a static analysis technique used mostly for non-functional verification of software. In this paper, we show the status of the technology that implements abstract interpretation which can help in GUI-based software verification. Specifically, we investigate the use of the Julia tool for the functional verification of a Graphical User Interface (GUI).

1 INTRODUCTION

Abstract interpretation was developed in the seventies and has since then been used in many aspects of computer science such as static analysis and verification (Cousot and Cousot, 2014). For a software S, it is impossible to write a software system S' that computes all possible executions based on all possible inputs and to represent them mathematically (Cousot, 2016). This is an undecidable problem. Abstract interpretation can be used in the verification of undecidable properties of software. This is done by the use of mathematical approximations of the data structures and possible ranges of values (Cousot and Cousot, 2014). Hence, abstract interpretation is used in the formal description and verification of undecidable properties of software (Cousot and Cousot, 2014). It is a technique for automatic static analysis that consists in replacing a specific part of code by a less detailed abstraction in order to calculate some properties of the program (Boulanger 2013). This technique enables detecting runtime errors such as division by 0 and overflow. It can also detect shared variables and dead code (Boulanger, 2013).

In this paper, we study the flow of data provided by the user (input data) to the GUI, though the GUI up to the application logic. The GUI processes the data and passes it (output data) to the logic. We refer to this as an input-output relation in the rest of the paper.

We try to use abstract interpretation to verify the Graphical User Interface of GUI-based software. This paper answers the research question: Is it possible to verify functional properties of the GUI of a GUI-based software using abstract interpretation applied on the GUI code?

The rest of this papers is organized as follows. Section 2 presents background and related work about abstract interpretation. Section 3 presents the input-output relation. Section 4 explains the proposed solution. Section 5 presents the conclusions.

2 BACKGROUND ON ABSTRACT INTERPRETATION AND RELATED WORK

Abstract interpretation depends on using mathematical approximation concepts (Cousot, 2016). It can be defined as follows: "a unified model for static analysis of programs by approximation of fixpoints" (Cousot and Cousot, 1977). An invariant is a property which holds for all trajectories of the software, i.e., all software execution paths (Cousot, 2016). Abstract interpretation analyzes software trajectories (Cousot and Cousot, 2010) and identifies safe zones and forbidden zones. When a trajectory does not lead to an error it is called a safe zone. A forbidden zone is a part of a trajectory that may lead to an error (Cousot and Cousot, 2010). An example of an error that leads to a forbidden zone is an overflow in a condition of a while loop. Abstract interpretation builds global variables for the state of the software in those trajectories and then partitions those trajectories into stages that represent similar behaviours of the trajectories (Cousot and Cousot, 2010). If the abstraction is safe, then each abstract invariant represents a superset of the concrete states

Alkhalid, A. and Labiche, Y.
Towards GUI Functional Verification using Abstract Interpretation.
DOI: 10.5220/0006917103810388
In *Proceedings of the 13th International Conference on Software Technologies (ICSOFT 2018)*, pages 381-388
ISBN: 978-989-758-320-9

after each instruction. In the presence of a loop, the repeated interpretation of loop instructions leads to the abstract invariant getting stabilized, a fixpoint having been reached (Boulanger, 2013).

We did not notice the use of abstract interpretation for the verification of the GUI of software in the literature. However, we found a framework called Bandera, which enables the automatic extraction of finite-state models from the source code of software (Corbett et al., 2000). Bandera uses abstract interpretation in order to extract the finite-state model. In addition, we noticed the use of abstract interpretation by Airbus France in hard real-time avionics software, such as flight control software which is always expected to react in time (Thesing et al., 2003).

We investigated an abstract interpretation tool called Julia to be used in our experiments (Spoto and Jensen, 2003). Julia performs abstract interpretation of Java software. Julia observes conditional statements in the code, collects information about methods, and determines when and how they are being called (Spoto, 2005). The abstract interpretation using Julia is performed through a fixpoint calculation, focusing on program points called watchpoints (Spoto, 2005). This is done through a software component that receives software byte code and dumps results into a report file that describes the output (Spoto, 2005). Spoto (author of Julia) and Jensen observed that the information provided by many static analyses is significant or useful only at a limited set of software points, which they call the watchpoints (Spoto and Jensen, 2003). For example, information about a variable which can contain zero at run-time is useful only before a division (Spoto, 2005). Class analysis includes identifying which call leads to a specific target method (Spoto, 2005). Hence a watchpoint must be put before the call of the software (Spoto, 2005). Julia depends on this idea to focus on specific software points.

Julia uses a hierarchy of semantics, which is a concept used to classify semantics into two types: Trace and watchpoint (Spoto and Jensen, 2003). A trace is a sequence of states for a piece of code, while a watchpoint is a program point that plays an essential role in program behavior. A piece of code can be represented by a set of traces to create a sequence of states for that code. There is a trace for every possible input state for the code. Abstract interpretation tries and tests several approximation mechanisms (Cousot and Cousot, 1977). However, practically, Julia allows the user to add watch point at any part of the code she/he wants.

As for our scope, we use the Julia static analyzer as follows. Julia provides a set of features that the user can use to analyze the software statically. We are interested in a feature that enables us to find numerical invariants. This feature has been implemented in Julia version 2.3.4. In this deployment of Julia, there is a new option named termination checker which is available in the Julia wizard. This option is usually used in operations related to termination of Java programs. This has been an issue for some researchers as some Java programs do not terminate completely until shutting down the Java virtual machine. There is a branch of research on this topic that investigates how to make sure that a program has terminated (Spoto et al., 2010). However, in our scope of verification of software, we are interested in an option called dumpNumericalAssertions. By turning it on through clicking the check box in the Julia wizard, Julia generates an output text file with numerical invariants at user-defined program points.

Julia observes conditional statements in the code. It also collects information about the software methods such as when and how they are being called. Julia also tries to find an activation frame, which is a description on when and how a software method is called. Then, Julia uses a lookup procedure to find the target software method (software function) of the call. Then, Julia creates an activation frame for the called method (Spoto, 2005). The activation frame for a method is a setting that simulates the circumstances when a call happens. Finally, Julia moves the output of the called method into the stack of the caller.

Abstract interpretation using Julia is performed through software components that receive software bytecode and dump results into a report file that describes the output. One component is a code preprocessor. A fixpoint engine is another component that uses an external module to abstract bytecode. It also has its own fixpoint strategies. A third component is a library that works as a low-level interface to .class files (Spoto, 2005).

3 INPUT-OUTPUT RELATION

We refer to as *input variable* any variable in the GUI code that receives a value from the user. We refer to as *argument variable* any variable in the header of a method in a Control class. We use the term *input-output relation* to refer to the relation between these two kinds of variables. The input data is received from a human actor and assigned to an input variable.

Through some control flow in the UI code this data reaches an argument variable. This is a way to model the flow followed by data when a Boundary class converts data received from a human actor into a form that can be dealt with in a Control class.

We distinguish between six different kinds of multiplicities. In the Many to One (N-1) multiplicity, several input variables to Boundary classes are used to form (i.e. compute) one argument variable to a function in a Control class. In a One to One (1-1) multiplicity, one input variable to a Boundary class becomes one argument variable to a Control method. In a Many to Many (N-M) multiplicity, many input variables to Boundary classes contribute to many argument variables. In a Many or One to zero (1..N-0) multiplicity, one or more variables do not contribute to any argument variable to the Control class. In a One to Many (1-M) multiplicity there is one input variable to the GUI that contributes to many arguments of methods in Control classes. In a Zero to one or Many (0-1..M) multiplicity, there is no input variable to the GUI but one or many arguments to Control methods.

4 PROPOSED SOLUTION

We use Julia static analyzer (Spoto, 2005). Our objective of using Julia is to obtain a mathematical formula (invariant) that represents the code. Specifically, the GUI code handles the user input to obtain an output that will be given to the functionality of the logic code. We want to extract a mathematical function from the code which implements operations from the input to the output. We can then compare the obtained mathematical function with the expected one. The expected function is the predicted one. The software developer or tester predicts that a code (related to input-output relation) implements such a function. If the actual and predicated mathematical functions are the same, there is no fault in the code. Hence, we can verify the GUI. Subsection 4.1 describes an illustrative example using Julia. Subsection 0 describes our tries for obtaining the invariant. Subsection 0 describes our tries to write the invariant as Java code.

4.1 Simple, Illustrative Example

Figure 1 shows the main window of our example software when the user enters an input, using the text field, and presses the `copy` button. We use a simple software which we use to explain how we utilize abstract interpretation. The software that we design uses the Entity-Control-Boundary (ECB) design principle (Bruegge and Dutoit, 2000). Boundary represents the GUI and Control represents the logic (implementing use cases) while Entity classes hold data/state.

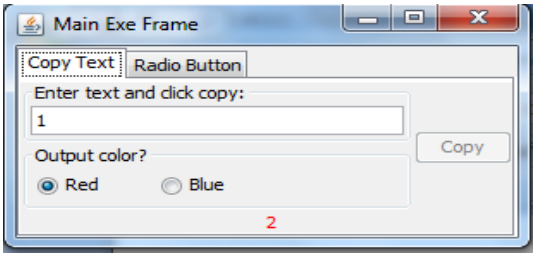

Figure 1: The main window of simple software after typing the input and clicking the copy button.

The Entity class is `MyTextEntity`, which has a functionality for creating some text files on the hard disk and printing text on the standard output stream. `TextControl` is a Control class that receives inputs from the Boundary class and makes a call to the Entity class. `RadioComponent`, `CopyTextComponent`, and `MainExeFrame` are Boundary classes which receive the input from the user, process it and pass it to the Control class.

The software takes the user input, adds the value 1 to it, and then shows it in the label under the radio buttons. The output is 2 in this case. In this specific example, the `CopyTextComponent` instance receives the input value "1" from the `MainExeFrame` instance. The class `CopyTextComponent` adds a value of "1" to it, and then it passes it to the Control class. This code intentionally introduces a fault. In the normal execution, this software should show the input on the output without any semantic change. The code which increases the input value breaks the ECB principle; the Control class should do this. Figur shows a sample of the code from class `CopyTextComponent`.

```
1    int userInput =
     Integer.parseInt(inputExpression.ge
     tText()); // get the user input
     from the text field
2    int callArgument = userInput + 1;
     // process the entered value by
     simply adding 1 to it.
3    TextControl t = new
     TextControl(Integer.toString(callAr
     gument));// passing the value to a
     constructor of class TextControl in
     the Control package
4    t.printCopyMessage(evt.toString(),0
     );
```

Figure 2: The part of the source code of the GUI that changes the input value.

The `CopyTextComponent` instance obtains a value from the GUI (line 1) and stores it in the `userInput` variable. Then (line 2), it declares a variable called `callArgument`. The `callArgument` variable takes the value of `userInput` and adds the value 1 to it. Then, it passes it to the Control class instance. When the user enters a value in the text field and presses the copy button, the click by the user triggers the method `actionPerformed`, which creates an instance of the `TextControl` class. Then, the call proceeds to the method `printCopyMessage()`. The method `printCopyMessage()` triggers the `log()` method in the Entity class called `MyTextEntity`.

We can notice that the value of the input variable provided to the GUI is passed to the constructor of the Control class when the actor's action results in a call to the `actionPreformed` method. Hence, we need to make sure that the input given to the `TextControl` instance is the right input–in this case, the string value received from the text field. The code converts the string to an integer and increases the output by a value of one. The code converts the value (after the increment) to a string. After that, the code passes the string to the Control class instance. One procedure would be to add a static variable for every single input provided by the user. We add a watchpoint (Line 3) to the code as shown in Figure. Figure shows a part of the class CopyTextComponent after adding a `watchpoint`.

```
1 int userInput =
  Integer.parseInt(inputExpression.getText
  ()); // get the user input from the text
  field
2 int callArgument = userInput + 1; //
  process the entered value by simply
  adding 1 to it.
3 Watchpoint.analyzeHere();
4 TextControl t = new
  TextControl(Integer.toString(callArgumen
  t));// passing the value to a
  constructor of class TextControl in the
  Control package
5 t.printCopyMessage(evt.toString(),0);
```

Figure 3: The adding of Julia watchpoint in the source code.

The added line should be in the part of the code that we want to analyze. Notice that, even though we modify the code, Julia performs a static analysis and the code, including the added part, is not executed. Usually, this is the part of GUI code that exists before the code that making a call to a Control class. It is actually at the intersection of Control and GUI. The tester can use a tool such as Atlas (Kothari 2017) to identify this part in the code. Then, Julia, when checking the modified code of Figure, generates an output log file with invariants (just one in this example since there is only one watchpoint).

```
********** PathLengthAnalysis
of public
CopyTextComponent.actionPerformed(java.aw
t.event.ActionEvent):void *
*********
normal execution: OL3 - OL4 = -1
exceptional execution: OS0 >= 1, OL3 -
OL4 = -1
                72: open call
com.juliasoft.julia.checkers.Watchpoint.a
       nalyzeHere():void []:public
CopyTextComponent.actionPerformed(java.aw
  t.event.ActionEvent):void:141 (offset:
                   72)
```

Figure 4: The content of the file NumericalInvariant.pl generated by Julia.

There is an invariant at the call to `analyzeHere()` at line 3 but applied at the bytecode level. There, the constraint inferred by Julia is `OL3 - OL4 = -1`. Specifically, **Error! Reference source not found.** shows the content of a file called `NumericalInvariant.pl` generated by Julia.

That is, the local variable 3 (OL3) is equal to the local variable 4 (OL4) minus 1. Here, Julia refers to local variables in the Java bytecode. Julia would help by finding the mathematical function that represents the input-output relation (the output as a function of the input). In this case, the mathematical function is the difference between variables (OL3 and OL4) in a text file (**Error! Reference source not found.**). It is: `OL3 - OL4 = -1`. We can understand from the file `NumericalInvariant.pl` that the local variable three (OL3) stands for `userInput` and the local variable four (OL4) stands for `callArgument`. Julia does not tell us that OL3 represents `userInput`. We should find out that by ourselves by understanding the output text file `NumericalInvariant.pl`.

We conclude that we verified the GUI for this particular scenario. The fault is discovered by reading and analyzing the text file generated by Julia. In other words, there is a fault for the variable `callArgument` (it does not have the same value entered by the user) thereby breaking the ECB principle.

4.2 Obtaining the Invariant

The illustrative example shown above is to obtain an invariant. Obtaining the invariant is done by using the termination checker.

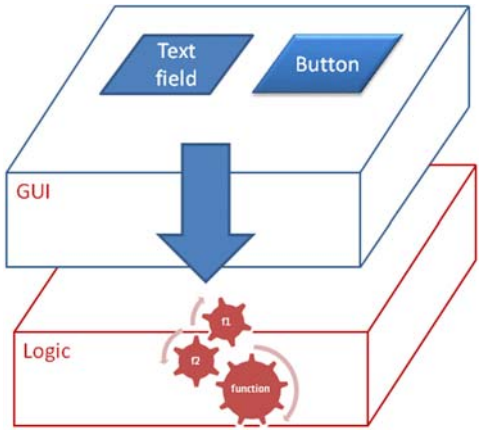

Figure 5: Passing input from GUI to logic.

We tried the termination checker on a dummy case study following the steps that the authors of Julia mentioned. However, we faced issues / limitations we describe next when we apply static analysis using Julia on a real case study. In the dummy case study, there is a text field in the GUI. The text field receives an input from the user (**Error! Reference source not found.**). Then, the GUI class changes the value of that input. The GUI class passes the modified input to the logic class of the software (non-GUI classes). The following code shows the call to Julia at the intersection of a GUI class and a logic class. The code is inside a GUI class. We make an instance of a Control class `TexControl` which is a logic class.

```
1    int
     userInput=Integer.parseInt(inputExpre
     ssion.getText()); // get the user
     input from the text field
2    int callArgument = userInput + 1; //
     process the entered value by simply
     adding 1 to it.
3    com.juliasoft.julia.checkers.Watchpoi
     nt.analyzeHere();
4    TextControl t = new
     TextControl(Integer.toString(callArgu
     ment));// pass to the logic
```

We tried to extend the dummy case study to handle two input variables instead of one. Hence, we had two text fields on the GUI instead of one. So instead of the old code, we had the following new block of code:

```
1    int userInput =
     Integer.parseInt(inputExpression.getT
     ext()); // get the user input from
     the first text field
2    int userInput1 =
     Integer.parseInt(inputExpression1.get
     Text()); // get the user input from
     the second text field
3    int callArgument = userInput +
     userInput1; // process the entered
     value by simply adding 1 to it.
4    com.juliasoft.julia.checkers.Watchpoi
     nt.analyzeHere();
5    TextControl t = new
     TextControl(Integer.toString(callArgu
     ment)); // pass to the logic
```

We followed Julia online-help by setting the variable `dumpNumericalInvariant` to true. In the first case (section 4.1), Julia generates a file named `dumpNumercialInvariants.pl` that has the following information: `OL3 - OL5 = -1`. This means that: `userInput - callArgument = -1`. We expected in this new case to obtain: `OL6 - OL5= - OL3` which means that: `userInput1 - callArgument = - userInput`. Alternatively, an acceptable output will be `OL3 - OL5 = - OL6` which means that: `userInput - callArgument = - userInput1`. We mention an acceptable output because when we compare the two pieces of code, we think that is a plausible output. We found that Julia cannot do the second analysis though it can do the first one. The file `NumericalInvariants.pl` generated by Julia does not include any invariant in the second case. We conclude that Julia has limitations when the code becomes complex.

We tried other checkers to solve this issue. For example, we tried the injection checker (Burato et al., 2017). This checker is supposed to report issues related to taint analysis (Burato et al., 2017). For example, if a parameter of a function is suspected to be tainted (e.g., changed by malicious code), the injection checker of Julia reports that part of the code which contains the parameter that produces a risk. Unfortunately, when we used the injection checker, we got an error during the analysis. Julia did not continue the analysis and reports an error. The error message simply says an error while connecting to the server. The deployed version of the tool is not ready for such an analysis. We tried other options. For example, we explored whether we can use an option in Julia called the polyhedra for small software to study the invariants. The authors of Julia suggested the use of the polyhedra option. The authors claim that polyhedra has a high computational ability. We prepared a very simple Java class in which a method a() calls a method b(). If we define two integer variables x = 1 and y = 1 inside a() and then another integer variable z = x + y also inside a(), then we pass z to b(), Julia can find the invariant using `dumpNumericalInvariant = true`. However, if we move the declaration of x and y to the `main()` function, Julia cannot find the

invariant using the `dumpNumericalInvariant`. So, we tried to set `onlyPolyhedra` to true and run the analysis. We got the same error message that we obtained when we try the injection checker. **In summary, Julia helps us with a specific case study, but we faced an issue on a complicated case study.**

Julia can also use both bounded differences (bugseng, 2017) and polyhedra together, falling back to bounded differences when polyhedra are not needed. This option is unavailable online due to internal issues related to the company that owns Julia. One possibility would be to use polyhedra and bounded differences for the application under analysis and bounded differences only for the libraries used by the software developer such as Java swing library. However, while Julia authors suggest this, they also they confirm that the Julia team has never tested the polyhedra.

The fact that polyhedra does not scale up in some cases is due to the amount of code that Julia must check. Since static analysis tries to analyze the code statically, it does so through approximations. If the code uses a lot of libraries, such as any decent Java program, then there is simply too much code (and especially code that is not necessarily accessible to the static analyzer, e.g., in Java libraries) for the analyzer to finish in a decent amount of time (if at all).

We searched for ways to proceed in such circumstances (limitations) to help the static analyzer. In other words, to reduce the computations that the static analyzer need to do. For instance, some steps can be done to help the analyzer, so that the analyzer has fewer computations/assumptions to make. If the code we analyze uses a function from a library that we (humans) know returns an integer between 1 and 10, then, we can inform the analyzer of such a situation. Hopefully, such step could help the analyzer. We thought of different ways to help Julia finds relationships between variables defined in the code. Specifically, we believe we could ourselves look at the code and identify information that could simplify Julia's analysis. We wondered whether other kinds of annotations could be of any help to us, including: (1) Annotations to specify bounded ranges of values for different types of variables (other than numerical variables); (2) Annotations to specify specific values for variables. These values could be enumerated or continuous; (3) Annotations to specify exceptional / illegal values. These values could be indicated easily as we already do that for handling invalid input. Hence, using annotation, Julia could excludes such values from the set of possible values of a variable. If we have an option, then, we can express bounded ranges, using specific annotations; (4) Annotations to tell Julia to ignore some calls which we know are irrelevant to the analysis we ask Julia to perform; (5) Annotations to tell Julia to ignore some variables; (6) Annotations to tell Julia that some calls / variables that we (humans) can tell would not affect results. Hence, again Julia can exclude them from computations. For instance, one possibility might be to tell Julia to ignore a specific call (e.g., a call to display the UI) in the analysis.

We did not find annotations in Julia to do the listed items above. That might be related to the analysis based on bounded difference: there are bounded differences that Julia can automatically infer from the code, and there are situations where Julia needs help to do so. We found annotations for a specific example; they are however not interesting to us. The example is related to a checker called `GuardedBy` checker. As for the termination checker which is totally different from the `GuardedBy`, we did not find if there is a way to use those annotations. We only found annotations that Julia understands for code synchronization issues.

4.3 Writing the Invariant as Java Code

Our objective is to know whether a mathematical function does represent the code related to the input-output relation. Instead of asking Julia to obtain the invariant, we can ask Julia to tell us whether our predicted invariant is correct. We believe that we reduce the computations that Julia needs to do when we write the invariant and ask Julia to verify it instead of asking Julia to find it. The validation of an invariant is easier than finding that invariant. Hence, the answer of Julia will be yes or no. Yes: means the invariant (mathematical function/condition) is a valid one. No: means the invariant is invalid. Since our invariant is a predicted one, it may be a subset of the real invariant. This could be a limitation of this solution.

In this section, we present a way through which we write the invariant as a Java code. Recall that the invariant is a condition that should always be satisfied regardless of the input of the software.

We write an if-statement in the Java code that has the invariant as a condition. If the condition is satisfied, then the if-statement will call an infinite loop that we write inside the body of that if-statement. We ask Julia to analyze the code for a possible non-termination of the software. When the program goes into an infinite loop, it does not terminate. Julia should warn about such a possible scenario. The non-termination usually happens due to

bad practices by the software developer. The code out of such practices leads the software to freeze (i.e., stop working, hanging). We highlight the fact that if the condition of the added if-statement is satisfied, then this leads the program to stop/hang (does not terminate), hence, Julia should warn about such a possible non-termination. It means that Julia should statically evaluate the code to decide whether the condition is satisfied.

Since our condition represents an input-output relation, we can use Julia to validate whether this mathematical function represents the code. In other words, we try to use the termination checker to detect if an invariant holds instead of generating the invariant. We show next how we add an invariant in the Java code. Hence, if the invariant does hold (satisfied all the time), the while(true) in the following code is called, and the termination checker gives us a warning. Unfortunately, we faced a limitation of Julia if the variables are uninitialized. We show this limitation below.

We use an example to show up-to-which level Julia can help us. The example shows that Julia works on the simple case. When we update the example to have more complex code, Julia does not work well. We show this limitation in an incremental way: we start with a working copy of the example and ends up with an unsuccessful one.

For the following code, there is no warning reported by Julia. The condition of the if-statement is not satisfied. The program does not proceed to the while loop. Julia works well with this example (it gives no warning).

```
public class WhileTrue {
        public WhileTrue() {
        }
        public static void main(String[] args) {
                int x = 0;
                int y = 0;
                if (x == 1 || y== 1) while (true);
                System.out.println("Passed while ture");
        }
}
```

For the following code, all is fine as well, there is a warning reported by Julia. The termination checker finds that the program will go into an infinite loop. This is expected because the condition in the if-statement is satisfied. It means the program will not terminate. The termination checker warns about a possible non-termination.

```
public class WhileTrue {
        public WhileTrue() {
        }
        public static void main(String[] args) {
                int x = 0;
                int y = 0;
                if (x == 0 || y== 0) while (true);
                System.out.println("Passed while ture");
        }
}
```

If we use a dummy case study which takes two inputs from the GUI, Julia throws a warning for the while(true) whether the condition (invariant) is stratified or not. It means that Julia cannot evaluate the code. Here is the example:

```
public TextControl(String t) {
        // verification code
        int guiVariable=Integer.parseInt(t);
        int expectedGUIvariable=CopyTextComponent.
            getInputExpression() +
            CopyTextComponent.getInputExpression2();
        if (guiVariable == expectedGUIvariable) while
(true);
        // end of verification code
        myText = new MyTextEntity(t);
        }
```

The issue that we face is that Julia reports a warning in the two cases of invariant:

```
guiVariable == expectedGUIvariable
```

or:

```
guiVariable != expectedGUIvariable
```

In our code, whenever we call the constructor of `TextControl`, we pass a value that is equivalent to the value inside the `expectedGUIvariable`. Hence, if Julia were to evaluate the code, it should warn in the case of equality. Julia does not evaluate the whole code so it warns in both cases. Probably, Julia works at the level of function and not the whole software. In other words, Julia reports a warning in both scenarios because it was unable to evaluate the condition. The reason for this limitation with Julia is that the variables are uninitialized, and they will get their values only at run-time. So, Julia cannot evaluate them. We conclude that while Julia works with a simple example, we however cannot use Julia when we have several variables.

5 CONCLUSION

We investigated the use of abstract interpretation to verify the GUI of a GUI-based software. We discuss the possibility of implementing this approach using Julia. The technology we suggest has limitations. For example, Julia does not generate a mathematical function that describes a software method all the time. However, this technology is evolving. We expect that the technology will overcome these limitations shortly.

ACKNOWLEDGEMENTS

This research has been funded by the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

Boulanger, J.-L. (2013). Static analysis of software: The abstract interpretation, *John Wiley & Sons*.

Bruegge and Dutoit (2000). "Object-Oriented Software Engineering: Using UML, Patterns and Java."

bugseng. (2017). "Numerical differences." Retrieved 2017, 2017, from http://bugseng.com/products/ppl/abstractions.

Burato, E., P. Ferrara, et al. (2017). Security Analysis of the OWASP Benchmark with Julia. *Pro. of the First Italian Conference on Cybersecurity (ITASEC17)*. Venice.

Corbett, J. C., M. B. Dwyer, et al. (2000). Bandera: Extracting finite-state models from Java source code. Software Engineering, 2000. *Pro. of the 2000 Int. Conf. on, IEEE*.

Cousot, P. (2016). "Abstract Interpretation in a Nutshell." from http://www.di.ens.fr/~cousot/AI/IntroAbsInt.html.

Cousot, P. and R. Cousot (1977). Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, ACM: 238-252.

Cousot, P. and R. Cousot (2010). "A gentle introduction to formal verification of computer systems by abstract interpretation." *Logics and Languages for Reliability and Security* 25: 1-29.

Cousot, P. and R. Cousot (2014). Abstract interpretation: past, present and future. *Proc. of the Joint Meeting of the Twenty-Third EACSL Annual CSL and the 29th Annual ACM/IEEE Symposium on LICS*, ACM: 2.

Gomaa, H. (2000). Designing Concurrent, Distributed, and Real-Time Applications with UML, Addison-Wesley Professional.

Kothari, S. (2017). "Atlas." Retrieved 2017, from http://www.ensoftcorp.com/atlas/.

Spoto, F. (2005). Julia: A generic static analyser for the java bytecode. *The 7th Workshop on FTfJP'2005, FTfJP'2005*, Glasgow, Scotland, July 2005. Available at www.sci.univr.it/~spoto/papers.html.

Spoto, F. and T. Jensen (2003). "Class analyses as abstract interpretations of trace semantics." *ACM TOPLAS* 25(5): 578-630.

Spoto, F., F. Mesnard, et al. (2010). "A termination analyzer for Java bytecode based on path-length." *ACM TOPLAS* 32(3): 8.

Thesing, S., J. Souyris, et al. (2003). An abstract interpretation-based timing validation of hard real-time avionics software. *P. 2003 Int. Conference on Dependable Systems and Networks, 2003, IEEE*.