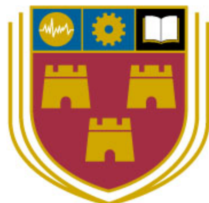
Ada Runtime Error Generator

Research Document

Date:13/11/2020



Student: Derry Brennan

Student number: C00231080

Supervisor: Chris Meudec

DECLARATION

I hereby declare that this research project titled "Ada runtime error generator" has been written by me under the supervision of Dr. Christophe Meudec.

The work has not been presented in any previous research for the award of bachelor degree to the best of my knowledge.

The work is entirely mine and I accept the sole responsibility for any errors that might be found in the work, while the references to published materials have been duly acknowledged.

I have provided a complete table of reference of all works and sources used in the preparation of this document.

I understand that failure to conform with the Institute's regulations governing plagiarism constitutes a serious offence.

Signature: Derry Brennan Date: 29/04/2021

Derry Brennan (Student)

C00231080 (Student Number)

The above declaration is confirmed by:

Signature: Chris Meudec Date: 29/04/2021

Dr. Christophe Meudec (Project Supervisor)

Abstract

"Ada is a state-of-the-art programming language that development teams worldwide are using for critical software: from microkernels and small-footprint, real-time embedded systems to large-scale enterprise applications, and everything in between [28] ." It is particularly used by the military, avionics and many other fields where safety is of critical importance.

With the reliance of safety in Ada it is pertinent to look into the ability to have run-time error free programs. An error that happens in the field could cause the loss of life or the destruction of property. My goal is to produce a prototype proof-of-concept tool which will be able to take Ada code and be able to tell the programmer if there is any possibility of runtime errors in their code and where. This document will be focused on my research into the different areas required for this project such as Ada itself, parsers and the different types of runtime errors.

This research project is based upon the work of my supervisor's software Mika, an automated test input generator for Ada code. The aim is to expand and build upon this software and if it is possible to expand it into the areas of runtime error detection.

Table of Contents

Abstract **2**
Table of Contents **3**
Table of Figures **7**
1 Introduction **8**
1.1 Ada runtime error generator 8
1.2 Runtime Errors/Exceptions 8
1.3 Ada 10
1.4 Parsers 13
1.4.1 Steps of language processing 14
2. Motivation **17**
3.Market Analysis **23**
4. Similar Tools **25**
5. Relevant Technologies and Algorithms **29**
5.1 Software Fault Tree Analysis (SFTA) 29
5.2 Constraint Satisfaction Problem (CSP) 29
5.3 Symbolic Execution 30
5.4 Prolog 32
5.5 SAT solver (Boolean satisfiability problem) 32
6.The Learning Curve **37**
6.1 Overview of the Process 37
6.2 Compilation 39

6.3 Parsing 42

6.4 Parsing additions 45

7. Limitations of implementation 56

8. Mika Extension for Text Editor 60

8.1 Visual Studio Code Extension For Mika 66

9. Testing the Limitations of Mika 68

10. Conclusion 69

11. Bibliography 71

Table of Figures

Figure 1 Sample of type ranges in Ada 9

Figure 2 Sample of Ada code 11

Figure 3 Simple code snippet 14

Figure 4 Example of lexical analysis tokenization 14

Figure 5 Example of a parse tree [29] 15

Figure 6 example Ada .ads file 18

Figure 7 main file used to execute the example program 18

Figure 8 Example Ada program that will produce a division by 0 error 19

Figure 9 Symbolic execution performed on the example program 20

Figure 10 output of the program in fig. 8 20

Figure 11 Test data generated by the Mika tool to provide full coverage through the fig. 8 program 21

Figure 12 Mika tools log output, including the division by 0 error 22

Figure 13 Example code GNATProve 27

Figure 14 output from GNATprove run on fig 13 27

Figure 15. Sample code, (Baldoni et al., 2018)[18] 31

Figure 16. Symbolic execution tree, (Baldoni et al., 2018)[18] 31

Figure 17 - DIMACS Format, Source: SpringerLink [Online] 34

Figure 18 Chart of the Mika parsing steps 38

Figure 19 Steps taken to generate test input 39

Figure 20 - Project with output files in VS2019 40

Figure 21 - Error received trying to execute the project 41

Figure 22 - Linking the project to required files from mika 41

Figure 23 - the correct way to obtain executable file 42

Figure 24 - The output from compilation, with debug specified 42

Figure 25 - IF statement from the ada.y grammar definitions 43

Figure 26 - The definition of safety to being the integer 5 44

Figure 27 - How the parser was used to build up a prolog term in example.pl 45

Figure 28 - constraint.adb example used to display constraint errors 46

Figure 29 - output from the compilation and execution of constraint above 46

Figure 30 - Ada code with custom type and custom range 47

Figure 31 - Output showing the constraint error of exceeding range of type 47

Figure 32 - Expanding the complexity while testing ranges 48

Figure 33 - Compiler still finds the error even when inside an else branch 48

Figure 34 - Ada code to showcase the ‘First & ‘Last of types 49

Figure 35 - Output from fig 34 49

Figure 36 - A further exploration of the range of a type 50

Figure 37 - Output from fig 36 50

Figure 38 - A sample Ada program with an array used within it 51

Figure 39 - How the array is represented within the generated constraint.pl file 51

Figure 40 - Additions in the ada.y file for when an indexed_component is encountered 52

Figure 41 - constraint.pl after the additions to the ada.y file had been implemented initially 53

Figure 42 - constraint.pl after using the tic() function instead of the character ‘ 53

Figure 43 - Sample Ada code used to test number overflows 54

Figure 44 - Output from fig 43 54

Figure 44 - change in the code to exceed the range of Custom_Int X(4) -T(42) = -38 outside the range of Custom_Int 55

Figure 45 - Compiler warning and error on the execution of the code from fig 44 55

Figure 46 - Compiler warning and error on the execution of the code from fig 44 57

Figure 47 - Ada.y additions for indexed_component, including the tic() in place of the character ‘ 58

Figure 48 - Source code reconstructed in the constraint.pl file 58

Figure 49 - an unexpected outcome within constraint.pl, rune was being called inside the package_specifications 59

Figure 50 - Example of Sublime Text GUI 60

Figure 51 - Example of Emacs GUI 61

Figure 52 - Example of Lisp code 62

Figure 53 - A sample program that would produce a division by zero once B reached 0 68

Figure 54 - Output from Mika while running with the exception flag 68

1 Introduction

1.1 Ada runtime error generator

This research document contains the research conducted in order to construct this Ada runtime error generator. The runtime error generator has been constructed to be able to take some Ada code as an argument and return if there is the possibility of a run-time error to occur within that code.

Initially as a proof-of-concept the project will focus on the main run-time error of division by zero and will expand upon that as the project continues.

For the construction of this tool the project will be basing this work on previous work done by Dr. Chris Meudec, the project's supervisor.

1.2 Runtime Errors/Exceptions

Runtime errors [3], sometimes called runtime exceptions in computer programming, is the term given when something unexpected happens during the execution of the program. The most basic example of this would be a division by zero error.

This may occur when asking the user for input to perform a division by and the user enters a zero. This would cause the program to crash if the programmer had not thought of this possibility and provided exception handling to prevent the crash from occurring.

These types of errors are impossible for the compiler to catch as the values are not set before the user enters their input and if there is no exception handling, such input could cause the program to cease functioning at a critical moment.

There are other types of runtime errors too such as providing inaccurate results to calculations. For example in an 8 bit system the range of integers is up to 255 so if one were to add 2 to this one would expect to get 257 as

the result. The result returned would be actually 1 as the value wraps back around to and starts from zero again once it has reached its maximum value. There is a modular type in Ada that handles this behaviour if it is expected.

Ada on the other hand checks its integers for overflow. There are 2 types within Ada. Machine level overflow, where the number is greater than the maximum value or less than the minimum value, which would yield a Storage_Error. The other is a Type-Level overflow where the operation on an integer exceeds its given range defined for a type which would yield a Constraint_Error. Both of these are run time errors.

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
   type My_Int is range 1 .. 20;
   A : My_Int := 12;
   B : My_Int := 15;
```

Figure 1 Sample of type ranges in Ada

Numbers in computing in modern times often are of an order of magnitude larger than this 8 bit example now with 64bit systems handling values up to 18,446,744,073,709,551,615 but this overflow can still occur if not accounted for.

Out of bounds exceptions are another type of error that all novice programmers experience at some stage of their education. If we make reference to an array of data but use an index or pointer that is not correct depending on the language being used, it could either be accessing memory not associated with our array and thus getting back incorrect values, or even overwriting memory with a new value that was not intended causing unknown errors at a future stage.

Some languages like Java have in-built out of range exceptions, but others like C or C++ do not and will let these instructions happen which can lead to undefined behavior.

1.3 Ada

Ada [1] is the product of a competition run by the US military in 1975 to find a secure and safe language to consolidate the multiple languages that they have been using previously on their embedded systems. The Department of Defense in America set out a list of requirements needed in their higher level languages and the first iteration of this was called project Strawman [2]. After a competition between four contractors, Ada, or as it was known then project Green, was selected as the winner. Jean Ichbiah was the leader of the group who produced Ada. The name Ada was chosen after Countess Ada King who is widely known as one of the first computer programmers.

After the Department of Defense began to implement Ada they continued to produce further proposals for what the language needed. They followed the naming scheme of Woodman, Tinman, Ironman, Sandman, Steelman proposals and these continued to be updated and revised upon.

As Ada was refined and started to become used in every aspect of the military, it began to be requested as a requirement for any future military projects.

Initially, from its inception in 1975, Ada was focused on both embedded and real-time systems but as it evolved in the 90s it was further expanded upon to include support for object oriented programming still with the focus on reliability and safety. One of Ada's key features is that it improves code safety and maintainability using the compiler to find syntax and logical errors in favour of runtime errors. This is where this project will come in, as it would be a good idea to be able to check both to further improve the safety and reliability of the code.

```
1   with Ada.Text_IO; -- adding a library
2
3   procedure SimpleProgram is -- main function, can be called anything
4      A,B,C : Integer; --  Variable declaration section
5   begin  --start of the code section
6      A:= Integer'Value (Ada.Text_IO.Get_Line);
7      B:= Integer'Value (Ada.Text_IO.Get_Line);
8      C:= A+B;
9
10     if C = 0 then
11        Ada.Text_IO.Put_Line("Result is 0");
12     elsif C > 0 then
13        Ada.Text_IO.Put_Line("Positive result :" & Integer'Image (C));
14     else
15        Ada.Text_IO.Put_Line("Negative result :" & Integer'Image (C));
16     end if;
17  end SimpleProgram;
```

Figure 2 Sample of Ada code

There are a number of dialects of Ada available; the two most prominent are SPARK [4] and Ravenscar [5]. SPARK is a subset of Ada with a focus on formal verification and static analysis. It does this through the use of contracts; these are behavioral properties that the developer must implement correctly and can then be checked against the verification toolset. SPARK Ada has a focus of eliminating runtime errors through a tool called GNATprove [27], which generates verification conditions (VC) that are used to assert that certain properties hold for the given program. This check is only proven theoretically though and the application of test data to cover all the branches of the code would still be very useful. It could then be tested again in a concrete manner if any runtime errors occurred with the application of the test data also. With these VCs it can determine if runtime errors, division by zero, array index out of bounds, numerical overflow or type range violation can occur within the program. As it keeps to preset standards of safety and security as outlined in the formally defined specification (contract), SPARK is used in the avionics field where safety and stability are key.

The GNATprove tools runtime checks come at a cost too, in that it increases the program size through additions of pre and post conditions to

every procedure and function that needs to be formally verified and also in terms of the execution time of the program. The goal of this is to provide statistical proof to demonstrate that none of these errors can occur during runtime.

Ravenscar [41] is another subset of Ada again focused on safety critical and real time computing. Ravenscar focuses on scheduling theory, such as the ability to assign characteristics to tasks within the program, defined in terms of deadlines. Deadlines are time constraints the given task has to be completed within.

- Hard: must complete on time, failure may result in failure at system level.
- Firm: must meet deadline under “average” or “normal” conditions. Completing a task after a deadline has no value and missing a deadline may cause system level degradation of service.
- Soft: must meet the deadline under “average” or “normal” conditions. But there is still value in completing the task even after the deadline has passed.
- Non-critical: A task with no strict deadline

The focus on scheduling in Ravenscar is to prevent Deadlock, Livelock, Missed deadlines or Blocking of tasks. It achieves this with a restricted scheduling model designed to limit the upper bound on blocking time, prevent deadlocks and to ensure that there is enough processing power to allow all critical tasks to complete within their deadlines.

Ravenscar also implements static analysis as a verification of the code, control flow to make sure there is no semantically unreachable code, data flow that ensures there are no variables with undefined values within the code, symbolic execution for code verification without the need for a formal specification and formal code verification which has a focus on runtime error checking. It does this through the use of preconditions, which need to

be met for an operation to precede and postconditions that hold following the successful call of an operation.

As software becomes more prevalent in every aspect of our lives, the safety of the code behind that software has begun to come under more scrutiny. From the disastrous Boeing 737 Max [42], to the Irish leaving Certificate software bug [43] in 2020 that affected the grades of thousands of students, it has become abundantly clear that errors in code can have real world impact upon the lives of people. This has led Nvidia, a leader in the world of machine learning and artificial intelligence to have implemented SPARK, the Ada subset in its hardware firmware, specifically related to self-driving cars [44]. Nvidia sees this as a way to increase the safety of their code by helping verify that the code is free from bugs and vulnerabilities. Through a study [45] conducted by AdaCore they determined that by working with SPARK a cost and time saving of close to 40 percent could be achieved.

1.4 Parsers

In the context of this project the parsing I will be discussing is the parsing of computer languages. There are many different parsing tools available such as the Lex [6] and Yacc [7] tools. Parsing is a term used to describe the analysis of code that conforms to a certain syntactic structure. The parser will break the code down into its most basic parts or tokens. It will then form those tokens into a tree structure with each branch being the syntactic relationship to each of the tokens.

To go into greater depth the lexical analysis stage takes code in as a string and removes any user comments and white spacing and reads the rest of the values assigning them as tokens. Common token names are identifier (variables names), keywords (e.g if, while, return, etc), separators (parenthesis, curly braces, semicolon), operators (+ , = , <, *), literals (True, “string”).

1.4.1 Steps of language processing

A tool such as Lex [6] will follow many steps to perform its task and this project will explore them here. Code will be supplied to the program as seen in figure 3. This is called the scanning stage or lexical analysis, here it has broken down the code into tokens.

```
A := B + 4;
```

Figure 3 Simple code snippet

The code from fig 4 would be the outcome of lexical analysis, having separated the code supplied into its tokens.

```
[(IDENTIFIER, A), (OPERATOR, :=), (IDENTIFIER, B),
(OPERATOR, +), (LITERAL, 4), (SEPARATOR, ;)]
```

Figure 4 Example of lexical analysis tokenization

After the lexical analysis has broken the code down into its tokens then that is passed to the parser (syntax analysis), such as Yacc (yet another compiler compiler) [7] where it forms a parse tree from the tokens.

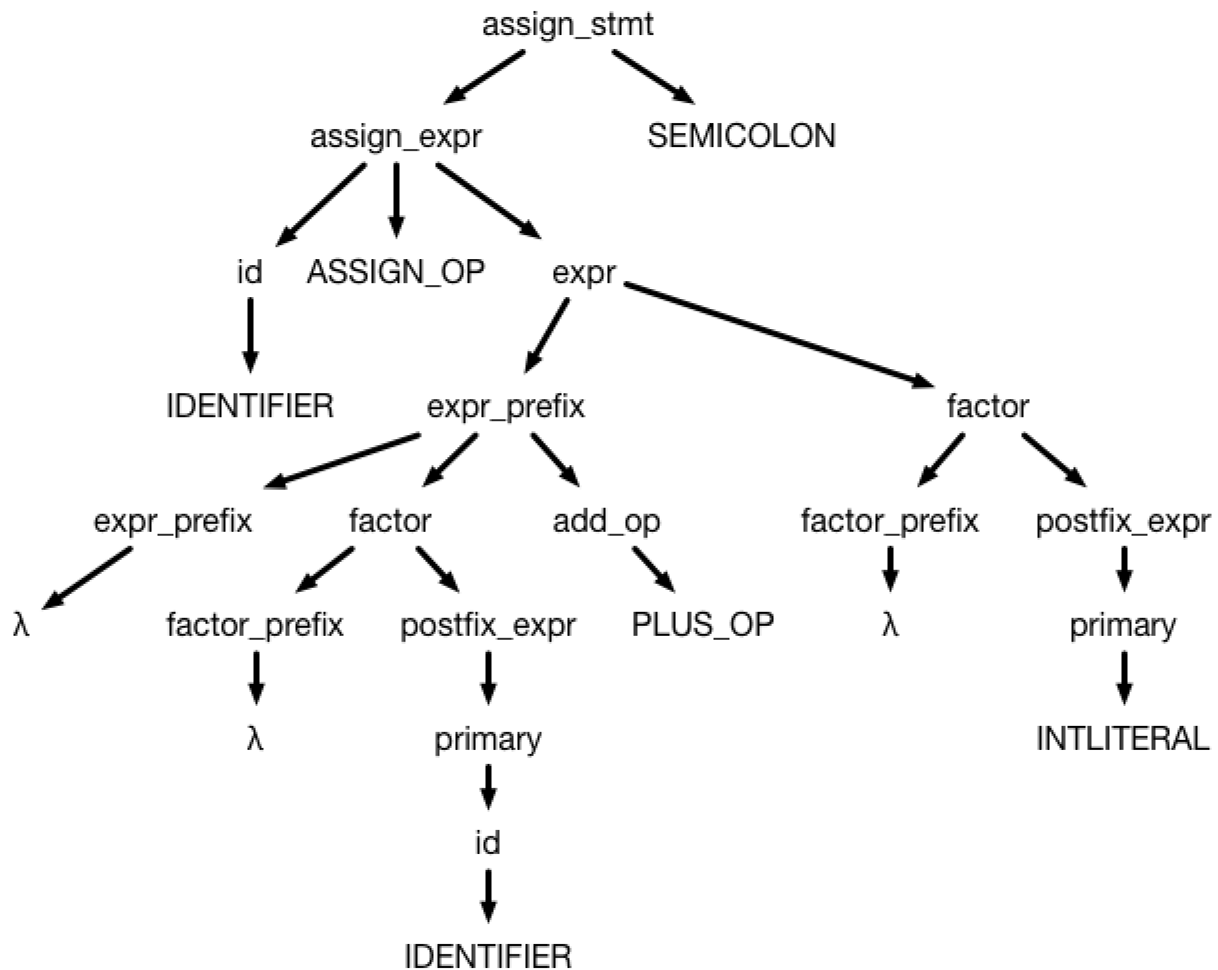


Figure 5 Example of a parse tree [29]

The tree above shows how the parser makes sense of the tokens passed to it by the lexical analyzer, here it being an assignment statement: an assignment expression ":=" followed by a semicolon ";". It can then be seen as the more complex form of assignment expression of an IDENTIFIER followed by an assignment expression ":=" followed by an expression.

That expression is then broken down into its expressions with extra steps to enforce the order of operations. What is left is a tree where its leaves are the tokens of the program feed to it by the lexical analyzer when read from left to right and ignoring the lambda expressions which are empty strings in effect it is left with:

IDENTIFIER ASSIGN_OP IDENTIFIER PLUS_OP LITERAL SEMICOLON

Through the rules applied to the parser it is possible to enforce variations of the standard rules applied and would allow for the insertion of annotations into the source code to allow for greater freedom to programmatically alter the source code to input the test data and also perform checks to see if the execution of the code would end up causing a division by zero for example

2. Motivation

The motivation for this project came from the feeling that there has been so much effort in general to improve the semantic and syntax errors in code, but nearly no tools help look for runtime errors.

Runtime errors are particularly problematic in critical systems such as avionics and rocketry, as well as any other system where human lives or other extremely valuable assets depend on the smooth operation of the system. These systems need to be tested for all outcomes and errors and if there was a tool that could do these things it would be of great benefit to developers as a whole.

One example of runtime errors causing great destruction is the Patriot missile failure of february 25 1991 [15]. Patriot missiles are used as missile counters aimed and exploding with oncoming missiles before they reach their destination. The patriot missiles tracked time by multiplying the system's internal clock by 1/10 to get the time in seconds. This calculation was done using a 24bit fixed register and 1/10 produces a non terminating binary value, so to fit into the register it had to be chopped short introducing a time error when multiplied with the larger time value. The patriot missile command was operational for 100 hours and this error over the course of that time altered the system's time by 0.34 seconds. Due to the travel speed of the oncoming missile being 1.676 meters per second, this caused the interception to miss by approximately 500 meters and cost the lives of 28 soldiers.

Another example is the explosion of the Ariane 5, June 4th 1996 [16]. The Ariane 5 was an unmanned rocket built by the European Space Agency. The total development cost of the project was $7 billion, but unfortunately it exploded shortly after launch when a 64bit floating point number used to track the horizontal velocity of the rocket was converted into a 16 bit signed integer. The number was larger than the largest storable 16 bit signed integer of 32,768 and caused the program to fail and the rocket to

explode. The cost of the rocket and its consignment was estimated to be $500 million.

These two examples of runtime errors causing both the loss of lives and the loss of property are the driving force behind this project and hopefully the project can make the software engineering process capable of finding these errors before they are deployed.

Below is an example program showing a runtime error.

example.ads

```
1   package Example is
2      procedure Bar;
3      procedure Foo(A , B : in out Integer);
4   end Example;
```

Figure 6 example Ada .ads file

gmain.adb

```
1   with Example;
2   procedure Gmain is
3   begin
4      Example.Bar;
5   end Gmain;
```

Figure 7 main file used to execute the example program

```ada
package body Example is
    procedure Foo (A , B : in out Integer) is
        X : Integer := 1;
        Y : Integer := 0;
    begin
        if A /= 0 then
            Y := 3 + X;
        end if;
        if B = 0  then
            X := 2 * (A + B);
        end if;
        A := 100/(X - Y);
    end Foo;

    procedure Bar is
        A : Integer := 2;
        B : Integer := 0;
    begin
        Foo(A, B);
    end Bar;
end Example;
```

Figure 8 Example Ada program that will produce a division by 0 error

The code from the program in figure 8 can be delved into deeper by performing symbolic execution on it. The variables are given a symbolic value and the conditional statements that are met along the flow of the program along with the symbolic values then form a boolean expression that can be evaluated to see if there exists a value for the variables that is valid. Below in figure 9 is an example of the symbolic execution.

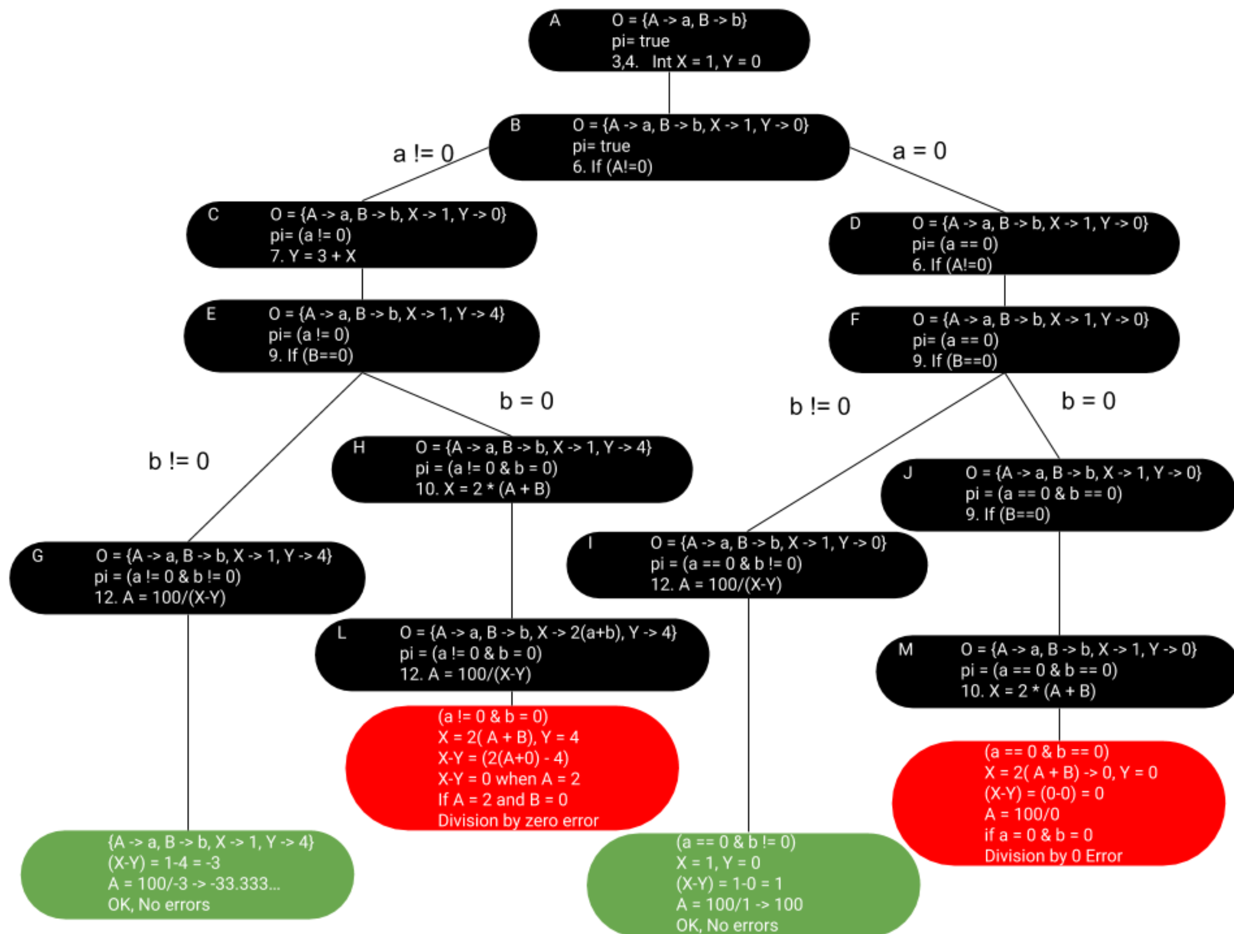


Figure 9 Symbolic execution performed on the example program

```
C:\Ada test code\Runtime errors>gnatmake example
gcc -c example.adb

C:\Ada test code\Runtime errors>gnatmake gmain
gcc -c gmain.adb
gnatbind -x gmain.ali
gnatlink gmain.ali

C:\Ada test code\Runtime errors>gmain

raised CONSTRAINT_ERROR : example.adb:12 divide by zero

C:\Ada test code\Runtime errors>
```

Figure 10 output of the program in fig. 8

```
----------------------------------------------------------
--              MIKA TEST INPUTS GENERATOR              --
-- Copyright Midoan Software Engineering Solutions Ltd. --
--                http://www.midoan.com/                --
----------------------------------------------------------
-- Original Directory:  C:/Ada test code/Runtime errors/
-- Target Directory:    C:/Users/derry/OneDrive/Documents/Ada test code/Runtime errors/example_mika/foo_3/
-- Package:       example
-- Subprogram:          foo
-- Strategy:            branch
-- Elaboration:         Not Ignored
-- Coverage Depth:      Subprogram Only
-- Time Stamp:          23:11:2020 19:03:43
----------------------------------------------------------
TEST NUMBER 1
PREDICTED COVERED BRANCHES :
Number 2, outcome true, in file example.adb, on line 9 and column 11
Number 1, outcome true, in file example.adb, on line 6 and column 11

BRANCHES COVERAGE INCREASED BY BRANCHES:
Number 2, outcome true, in file example.adb, on line 9 and column 11
Number 1, outcome true, in file example.adb, on line 6 and column 11

CONSTRUCTED TEST INPUT
a = 10720455
b = 0

CODE BEHAVIOUR
a = 0
b = 0
----------------------------------------------------------
TEST NUMBER 2
PREDICTED COVERED BRANCHES :
Number 2, outcome false, in file example.adb, on line 9 and column 11
Number 1, outcome true, in file example.adb, on line 6 and column 11

BRANCHES COVERAGE INCREASED BY BRANCHES:
Number 2, outcome false, in file example.adb, on line 9 and column 11

CONSTRUCTED TEST INPUT
a = 8206166
b = 10778680

CODE BEHAVIOUR
a = -33
b = 10778680
```

Figure 11 Test data generated by the Mika tool to provide full coverage through the fig. 8 program

```
Mika Generator ERROR
MIKA : START Generating Values
MIKA : END Generating Values
MIKA : START Print Solution Input Variables
MIKA : END Print Solution Input Variables
MIKA : START Print Solution Output Variables
MIKA : END Print Solution Output Variables


####################################
=>MIKA: a fatal error has occurred
        Error Code: 1024904
        Message: Division by a ground 0
        Arguments: x : 100
=>Report error to http://www.midoan.com/support.html to have it addressed.
```

Figure 12 Mika tools log output, including the division by 0 error

Mika is an automated test input generation software that was built originally to provide branch, Decision and Modified Condition/Decision Coverage (MC/DC) and was not originally designed to handle the finding of Runtime errors in code. When it does encounter these errors at the moment it just returns what happened. One of the goals of the project will be to implement exception handling within the mika_ada_generator (test input generator) to better handle these errors now that they are being explicitly targeted. In figures 6-12 above it is shown how a small program with the potential for a division by zero works at runtime and after passing it through the Mika tool to auto generate test data to achieve 100% branch coverage also. The division by zero error is not handled and the goal of this project is to allow the Mika tool to output information to the user to show them exactly where in the code the error occurred and with what test data brought the program to this state.

3.Market Analysis

The cost of testing in software development is already very substantial with the estimated costs of testing Minor or Negligible software being between 20%-40% of sales [17] of a piece of software. Software Criticality can be broken down into:

Severity	Dependability Consequence	Safety Consequence
Catastrophic	Failures propagate	Loss of life or Huge loss of assets
Critical	Loss of project	Causing injury or moderate loss of assets
Major	Major setback to the project(time and money)	
Minor or Negligible	Minor setback to project	

But when a software failure could lead to millions in lost assets or even loss of human life the cost associated with testing that software increases even more so as having functional but safe and robust code is the most important factor to the success of that program.

Taking the example of the Ariane 5 [16], the total cost of the project was $7 billion and the lost assets were $500 million. If a tool were available to test for the runtime error that caused this integer overflow it would have been of great value. The Ariane 5 actually used the Ada language and when trying to program the storage of a 64 bit floating point number being stored as a 16-bit integer, the Ada compiler did actually raise warnings, but these warnings were turned off because the programmers thought that it could never overflow [48], if the tool creating here could provide test input that would cause a number overflow the programmers would not turn off the warnings and proceed.

So the project proposes that by targeting Ada and already widely used language for critical software development as the target platform for the runtime error generator, there would be a substantial interest in such a tool if we are successful in the development of such. Since the potential cost of such a tool versus the immense cost of a failure such as this again makes sense.

Some of the potential customers for our service would be [49]

- Airbus
- Boeing
- Lockheed-Martin
- Saab
- TGV (French high speed rail)
- New York Subway
- Paris Metro
- Commercial rockets: Ariane, Atlas & Delta
- European Space Agency
- Reuters
- Multiple European Financial institutions (specific names unavailable)
- Hinkley Nuclear Power Station, England
- American, Australian, Canadian, English, Swedish Militaries
- NATO

4. Similar Tools

There already exist some tools that achieve a similar output to what this project intends to do. These include:

- Reactis for C

  Reactis [11] is an automated test generation tool. It has three main components: the Tester, Simulator and the Validator.

  The tester will generate test cases for the supplied program automatically and through applying these to the code has the ability to find runtime errors. The test data is generated with coverage in mind with the aim to get the most coverage of the code from the supplied test data.

  The simulator provides a GUI (graphical user interface) that supports debugging of the code in an interactive fashion with displays of the test coverage provided.

  The validator allows the programmer to formalize assertions and coverage targets. It will then simulate the program trying to break the assertions or targets. If a runtime error or assertion fails it will return the test data that produced the failure allowing the programmer to follow the exact sequence of steps taken to reproduce such.

  Reactis for C is a product of Reactive Systems, inc. and the exact technology used to produce this software is not readily available.

- Backstop for Java

  Backstop [14] is a project for Java code that aims to simplify the error messages produced when a program encounters a runtime error. The goal of this is to make novice programmers more comfortable with how they handle these errors when they appear without a confusing stack trace message. The stack trace is converted to a more human

readable format and this is displayed to the user. This could be a great educational tool in bridging the gap for new users to understand more complex error messages but in relation to this project it will have little relevance.

- Ada SPARK using GNATprove

  Spark [8] is a language based on Ada and it could be seen to be a tighter subset of Ada that focuses upon writing code for high integrity applications, such as avionics or financial systems. Spark encourages programmers to write code that is sound from the start just by the nature of the techniques used in the construction of the application.

  Spark uses annotations in the form of Ada comments. Since they are in comment form they are ignored by the Ada compiler so Spark code can be compiled into an executable using the Ada compiler. The annotations come in two types, Flow analysis and formal proofs. The proof annotations allow the assignment of preconditions and postconditions of subprograms, assertion of loop constants and the declaration of proof functions. These conditions allow Spark to generate Verification Conditions that can be verified by proof checking tools such as the SPADE Automatic Simplifier [9]. The flow analysis annotations allow for checking that modes of parameters and global variables match the detailed interdependencies provided in the annotations by the Examiner[10].

  GNATprove is an additional tool available for Spark that interprets the Spark annotations as they are interpreted at run time during tests. This allows the executables semantics to have the verification of run time checks, similar to the formal proofs in Spark itself and which can be verified statically by GNATprove.

  As seen in figure 4 below, GNATprove found the possibility of 5 runtime errors in the given code under certain conditions. With this

knowledge the programmer could then take steps to prevent these errors from occurring without the need for exception handling.

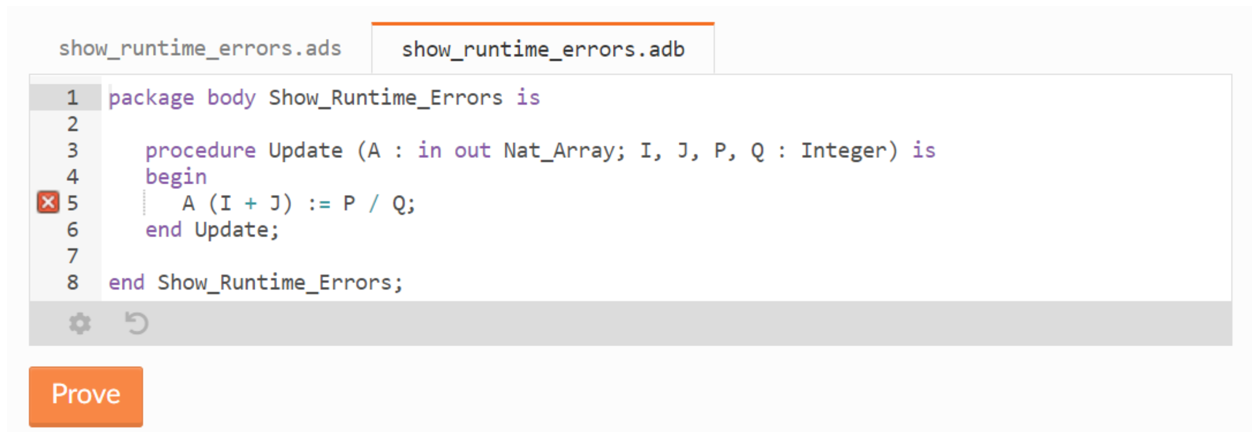


Figure 13 Example code GNATProve

https://learn.adacore.com/courses/intro-to-spark/chapters/03_Proof_Of_Program_Integrity.htm

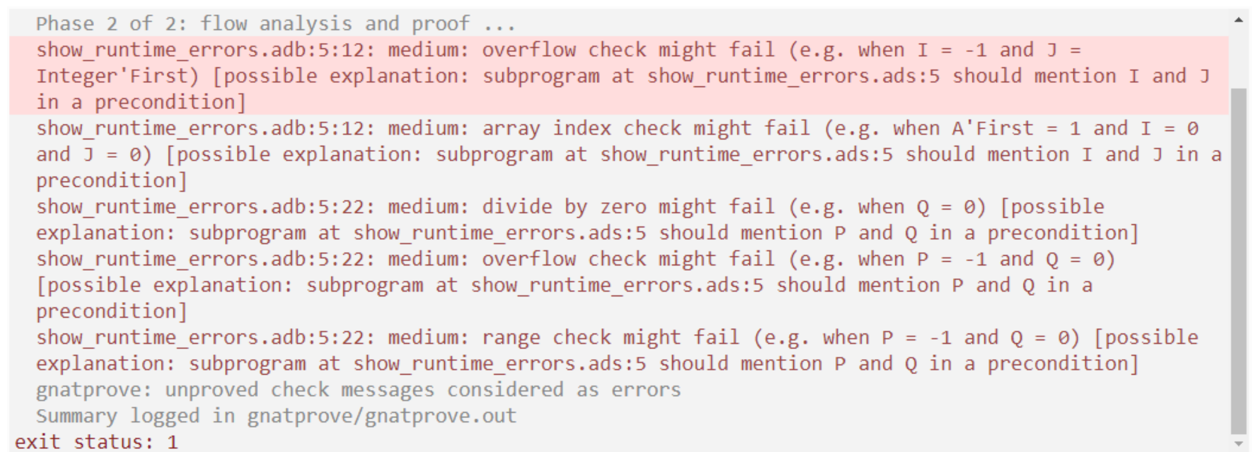


Figure 14 output from GNATprove run on fig 13

In the above example from fig 5 and 6 GNATprove can be seen finding first the potential of an integer overflow error if J was equal to the lowest possible integer and I was equal to -1. Instead of producing an even lower negative number it would wrap around and return the highest integer value.

The next error it finds potential for is an array index check. If the first index in the array was 1 and I and J were both zero, it would be trying to access memory not assigned to array A.

The third potential error found is a divide by zero error, stating that if Q is assigned 0 then the division would cause a runtime error.

The fourth and fifth potential errors are both related to the division by zero error, stating that the result may be an overflow or out of range error, but since division by zero is undefined it's hard to say for sure.

This output is very useful and informative, but the hope for this Ada runtime error generator is to provide, not potential errors, but given sample test data generated by the Mika tool it will provide a result to say if any of the test data would result in a concrete runtime error. This is much more presusave than just a warning that says that something is possible, providing test input that the developers can run the program with and produce this error will be very convincing that it is indeed possible and can happen once the software is released if the necessary steps are not taken to safeguard against it.

This Idea of concrete information being provided back to the developer is key to this project, not only will the presence of an error be found, but reproducible steps are given to them also. With this information in hand it will make the eventual prevention of the error from occurring.

5. Relevant Technologies and Algorithms

5.1 Software Fault Tree Analysis (SFTA)

A fault tree is a technique used in engineering to assess the risks and reliability of a project. It is a logical diagram that displays the relationships between events using AND and OR gates to establish under what circumstances a fault can occur. A software fault tree [13] analysis is a procedure that is applied to critical subsystems of some software to determine the paths and conditions that could be followed to reproduce a certain error or system failure. Applied to the system critical sections of software from an early stage in development the SFTA is a good way of detecting potential errors in the code before it becomes too complex during development. It is a useful step in determining how errors can come about in the flow of the program and a good first step in finding a way to eliminate those errors. Although very interesting, this technique would not be too beneficial to the project as it focuses more on the process rather than the code.

5.2 Constraint Satisfaction Problem (CSP)

The constraint satisfaction [12] problem has a set of variables {A} within the section of the code under review and a set of values for those variables {B} and a set of constraints {C} placed on some or all the variables that restrict the values that these variables can take. With these values set up it is the goal to determine if there is an assignment of a value to each variable {A} from {B} that meets the constraints in {C}. If this is the case then the problem is said to be satisfiable. But if the problem comes out to be unsatisfiable then the next step is to find if there is at least an optimal solution where as many variables can be assigned values within the constraints as possible.

Most CSP algorithms use search algorithms to look for all the possible assignments for the variables and given enough time these will always determine if the problem is satisfiable or not. There are a number of search algorithms to choose from for doing the searching and the examples used in [12] are a simple backtracking algorithm, forward checking and thirdly a MAC (maintaining arc consistency).

For example if one were to run the tool on a small piece of code that had variables {X, Y ,Z} where Z is going to be used as a divisor This would be our set {A}, {B} would be {.. , -2, -1 ,0 , 1, 2 , ..} and then {C} would just have one constraint {Z != 0}. This would be our constraint satisfaction problem for the division by zero within that code.

While this is very interesting, it will not be used in this current project as a custom solver is already in place, but expanding knowledge into other approaches is useful in fully understanding the problem at hand.

5.3 Symbolic Execution

Symbolic execution is a program analysis technique whereby the values of variables are changed to a symbolic value. As these values are stored as a program's flow happens and any conditional statements that it encounters, a boolean expression can be built up that can then be used to evaluate to find a value for a variable to meet all the requirements to allow the program to progress down the different paths.

An example being:

```
1.   void foobar(int a, int b) {
2.       int x = 1, y = 0;
3.       if (a != 0) {
4.           y = 3+x;
5.           if (b == 0)
6.               x = 2*(a+b);
7.       }
8.       assert(x-y != 0);
9.   }
```

Figure 15. Sample code, (Baldoni et al., 2018)[18]

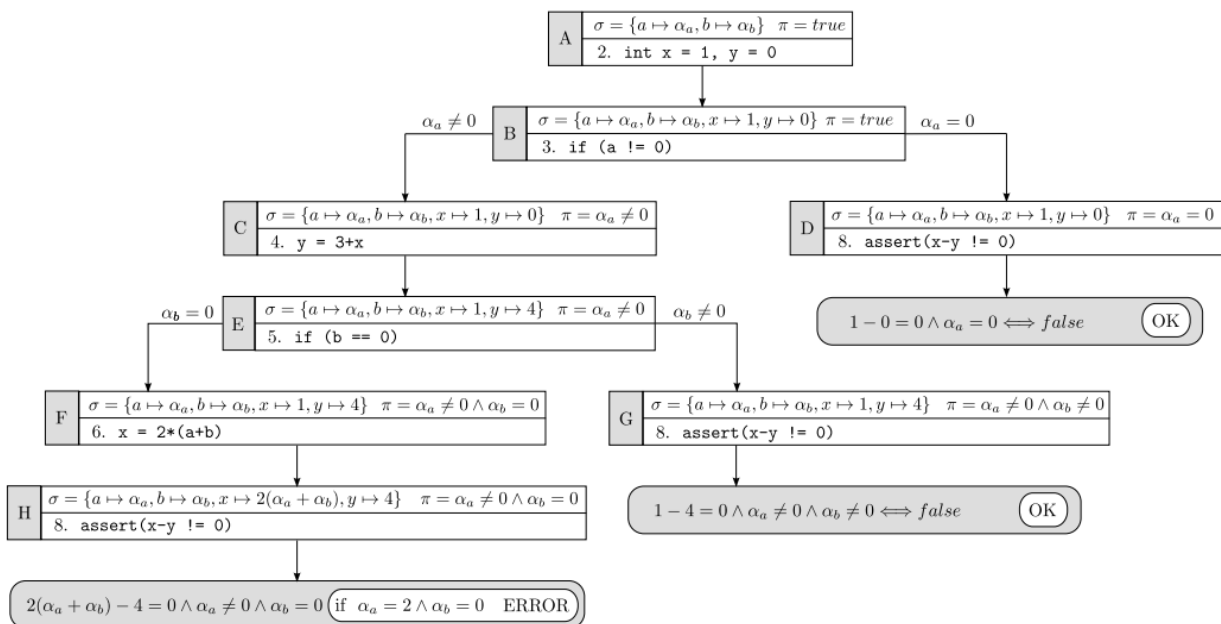


Figure 16. Symbolic execution tree, (Baldoni et al., 2018)[18]

There are many very interesting developments in symbolic execution and it has become rather popular in recent times with the evolution of Boolean satisfiability problem solvers. It is discussed at length in a number of

videos online, such as this Microsoft talk [19] and many educational resources are available on it too such as the Coursera online lecture [20] and MIT Open Lecture [21].

5.4 Prolog

Prolog [26] is a functional programming language that specializes in natural language, computational language and also artificial intelligence. Prolog is very specialised in handling predicate logic problems. It will take a number of facts or rules and compute the relations of those facts. The draw of prolog here is that it can take the output of the symbolic execution and make a prolog program from it and this will allow us to follow down the different paths of the code. If prolog ever comes to an unsatisfiable statement it has backtracking built in, where it will go back up to the previous statement and try a different path

5.5 SAT solver (Boolean satisfiability problem)

In logic and computer science, the SAT [25] or propositional satisfaction problem is the problem of deciding whether an interpretation exists that satisfies a given Boolean formula.

SAT asks if it is possible to reliably substitute the variables of a given Boolean formula with TRUE or FALSE values in such a way that the formula is evaluated as TRUE. If that is the case, then the formula is satisfiable. In contrast, if there is no such assignment, the function represented by the formula for all the possible variable assignments is false and the formula is said to be unsatisfiable.

These solvers are binaries that accept input in the form of a CNF formula text file and write to the console the corresponding output.

A SAT solver is a method that takes a CNF(conjunctive normal form) [24] which can be seen as a set of clauses and each clause as a set of literals, as input and outputs either a Boolean satisfactory assignment to the

variables used in the CNF formula if the formula is consistent or if it is not, UNSAT.

The success of Boolean Satisfaction (SAT) solvers has seen massive improvement in the past few years. [24]

Aside from the worst-case exponential run time of all known algorithms, in areas as diverse as software and hardware verification, automated test pattern generation, planning, scheduling, and even challenging algebra issues, satisfaction solvers are progressively leaving their mark as a general purpose tool.

There exist two ways to pass a formula to an SAT solver. The first one is by using a semi-standard file format known as DIMACS, and the second by using the SAT solver as a library.

Developers prefer to use SAT solver as a library, but the DIMACS format allows prototyping the applications faster, and quickly tests the performance of different solvers on the application's problem.

DIMACS has a line oriented format that consists of a comment line which starts with “c”, a summary line starting with “p” that contains information about the type and size of the problem in the file and a clause line which comprises space-separated numbers ending with 0. Each non-zero number for this line indicates a literal, with the negative numbers being negative literals of that variable, and 0 being the end of a line [22]. The figure below shows a CNF converted into DIMACS.

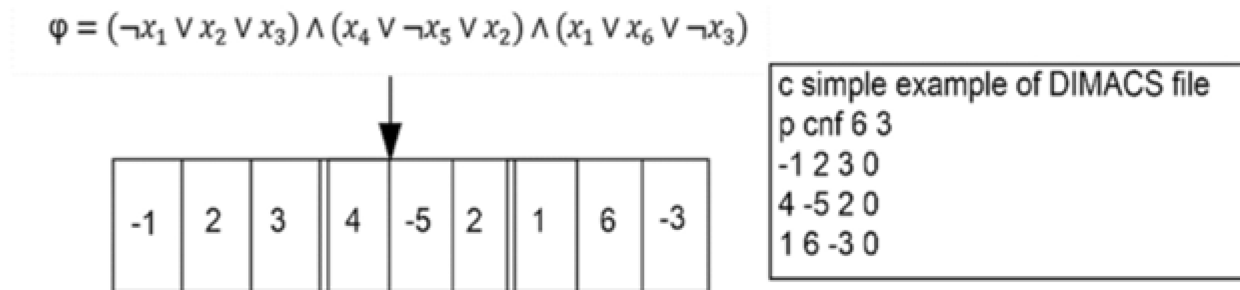


Figure 17 - DIMACS Format, Source: SpringerLink [Online]

https://link.springer.com/article/10.1007/s12652-020-02247-w

Modern SAT solvers can be categorized in two groups: Local Search based solvers and Conflict Driven Clause Learning (CDCL) based solvers.

Local Search solvers try to find a satisfactory assignment for the input Boolean CNF formula by changing randomly, the initial assignment using bit flips until a satisfying assignment is reached.

A standard SAT Local Search based algorithm consists of an initialization phase and a local search phase. In the first phase, all variables are assigned true values. The truth value of a single, heuristically chosen variable is modified at each stage of the local search phase. Exceptions are solvers based on evolutionary algorithms, which preserve solutions and use recombination techniques. The search process is terminated when a suitable assignment is found or when a bound on the runtime is met or exceeded. Almost all SAT Local Search algorithms are incomplete and they cannot determine the satisfiability of the formula. [23]

The CDCL solvers are the evolution of the Davis-Putnam-Logemann-Loveland (DPLL) algorithm, which is a relatively simple improvement of the naïve backtracking algorithm. CDCL is complete

because it replies to SAT that a solution exists and it is sound as it communicates to SAT an unsatisfiable formula.

To explain how CDLC works, the naïve backtracking and the DPLL algorithms will be explained as follows.

The naïve backtracking algorithm starts its work with picking a variable without an assigned truth value and if there are none, return SAT. Then a truth value will be assigned. Ultimately it is checked if all clauses in the formula are still potentially satisfiable. If they are, it will start from picking an unassigned variable, and if they aren't satisfiable another truth value is picked for the variable. If they are not satisfiable and both truth values have been used, backtrack. If there is nowhere to backtrack then return UNSAT.

DPPL algorithm introduces two new concepts, positive literal and negative literal. A literal is positive if it evaluates to true when its variables are assigned truth values and they are negative otherwise.

This algorithm aims to speed up the check for unsatisfiable clauses by updating the state of the clauses based on variable assignment. This means that after a truth value is assigned, all clauses that contain a literal of the variables selected at the beginning, are going to be updated accordingly. If they contain a positive literal, it means that they are satisfied, and they can be removed completely from further analysis. If they contain a negative literal, they cannot be satisfied using that variable and the literal can be removed from them.

The idea behind the CDCL algorithm is that a conflict, when an empty clause is created, is caused by a variable assignment that happened sooner than it was detected. If this problem can be identified when the conflict was caused it will be possible to backtrack several steps at once, without running into the same conflict multiple times.

The CDCL implementation analyzes the present conflict, finds the earliest variable assignment involved in the conflict and jumps back to the

assignment. The conflict clause is added to the problem, to avoid revisiting the search space that is involved in the conflict [23].

6.The Learning Curve

6.1 Overview of the Process

The process of developing test inputs for the supplied source code at a basic level requires the mika_ada_parser and the mika_ada_generator.

The mika_ada_parser is compiled from ada.l (Flex file) and ada.y (Bison file). The Flex file contains the information related to the lexemes and their patterns and what token they form. The Bison file contains the context free grammar and how these supplied tokens from the lexical analysis are formed and also generates the prolog term file (foo.pl) of the source code.

Mika_Ada_Parser.exe:

- Lexical Analysis:
  - Ada.l
  - Tokens
  - Lexemes
  - Patterns
  - Symbol Table or node information in abstract syntax tree (Lexeme, Token, Type, Address, etc...)
- Syntax Analysis (Parser)
  - Ada.y (custom context free grammar used for mika)
  - Parse Tree (leaf nodes of tree are the source code)
  - Syntax Errors
  - Semantic Analysis

Mika_ada_generator:

- Generated Prolog code (foo.pl)
  - Symbolic Execution
  - Inherent backtracking

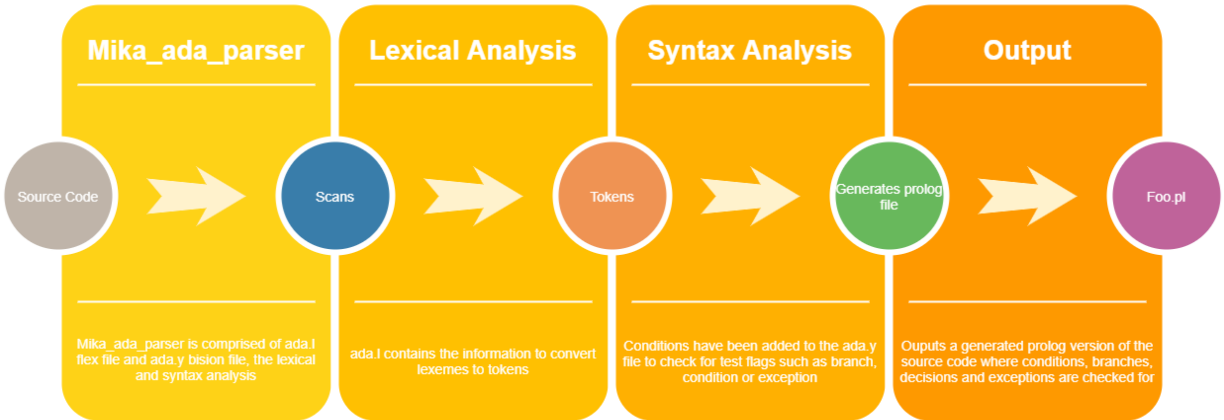


Figure 18 Chart of the Mika parsing steps

The Ada source code is passed into the Mika_ada_parser which is compiled from the ada.l and ada.y files, the lexical analysis (flex) and syntax analysis (bison). Additions for the exceptions switch have been placed in the ada.y file to add queries to the foo.pl file when exception conditions are met, e.g a division symbol for division by zero or an indexed element for the array index out of bounds check.

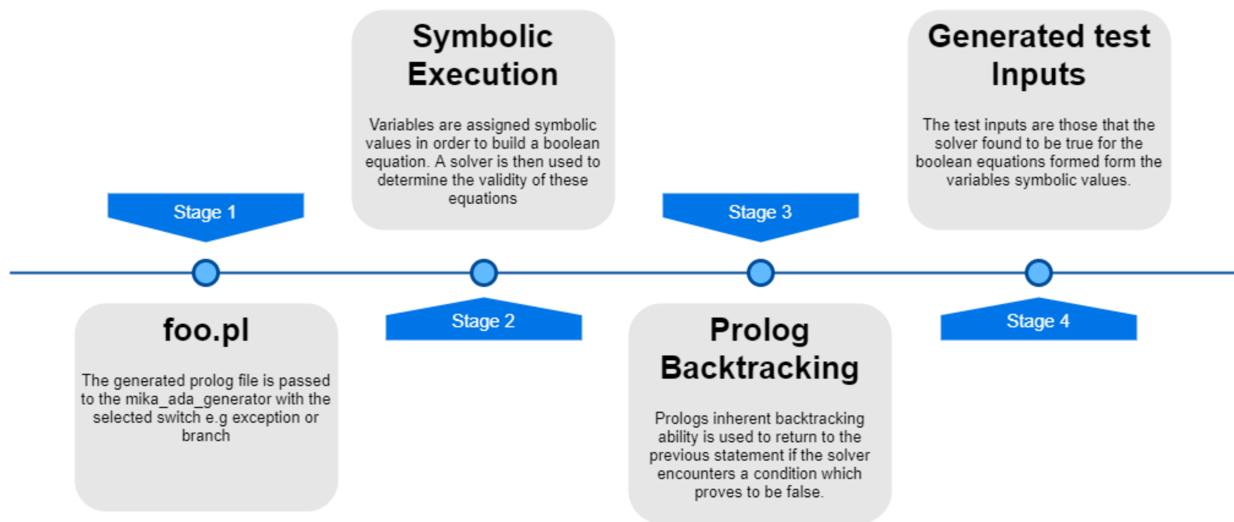


Figure 19 Steps taken to generate test input

The foo.pl file is the prolog terms version of the Ada source code unter test. This contains all typing information, package specifications and the source code and mika referencing for the common tokens and the specific tokens such as variables, procedures, functions and custom types used in the source code as prolog facts.

At the end of the file contains the queries that will be run to perform the symbolic execution on the variables in question and see if a condition exists where it evaluates to be true, if true test inputs are found that meet the supplied requirement.

6.2 Compilation

The first part of the process of understanding how to accomplish the task set out by the project was to compile the Ada parser used by the Mika tool.

There are two Flex files, ourxref.l and ada.l and two bison files ourxref.y and ada.y. The ourxref files need to be run through Flex and Bison respectively to produce output files used for compilation, these are lex.yy.c produced from ourxref.l and ourxref.tab.c and ourxref.tab.h produced from the ourxref.y file.

Once these output files have been produced they need to be added to a blank Visual Studio 2019 project. This is where some difficulties were encountered, see figure 15 below.

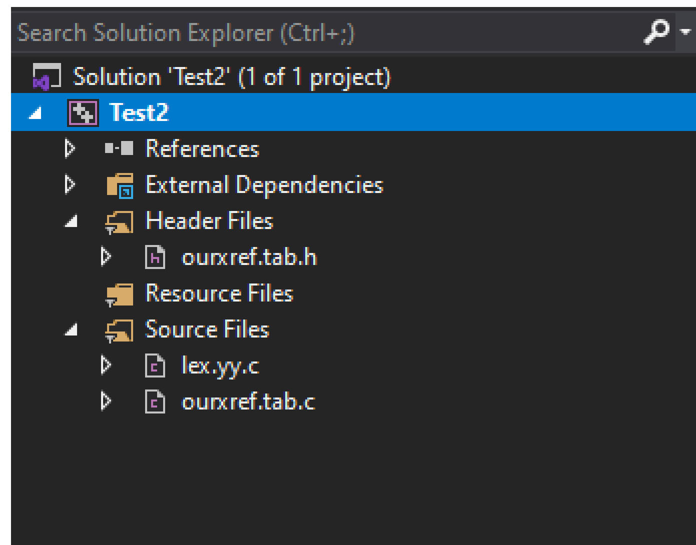


Figure 20 - Project with output files in VS2019

Upon executing this step of the project, errors were met as shown in figure 16. A double check was done to make sure that the required folders were linked to the project's C/C++ include directories and the correct folders were indeed present as shown in figure 17.

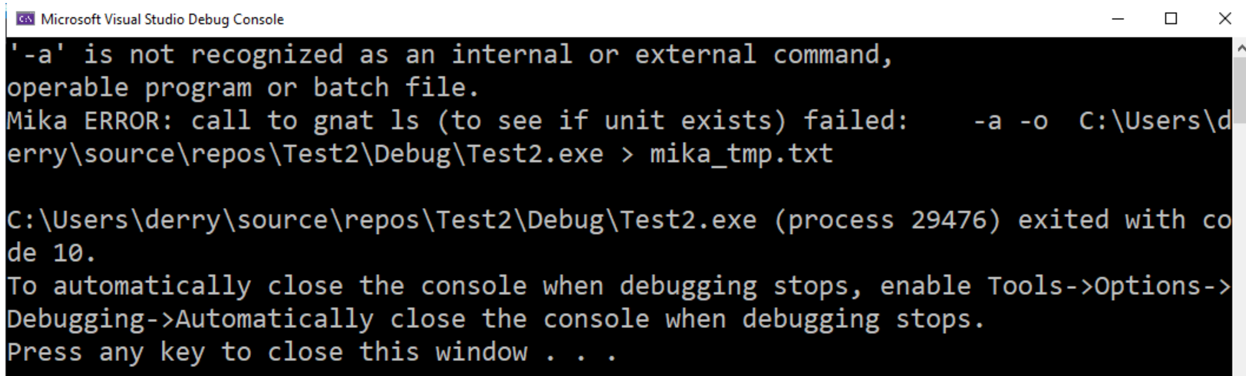


Figure 21 - Error received trying to execute the project

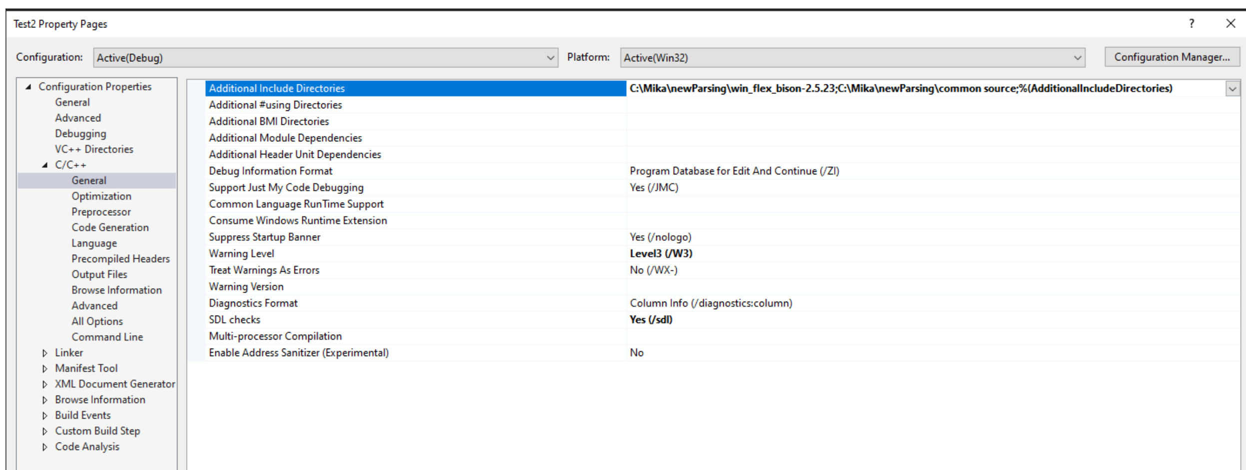


Figure 22 - Linking the project to required files from mika

After a prolonged period of trial and error, the correct way of obtaining the needed executable files from the compilation of the project was determined. As is depicted in figure 18, right clicking on the solution and selecting build compiles the output files and generates the necessary files (figure 19). With this knowledge in hand, the next step was to compile the ada.l and ada.y files in the same manner.

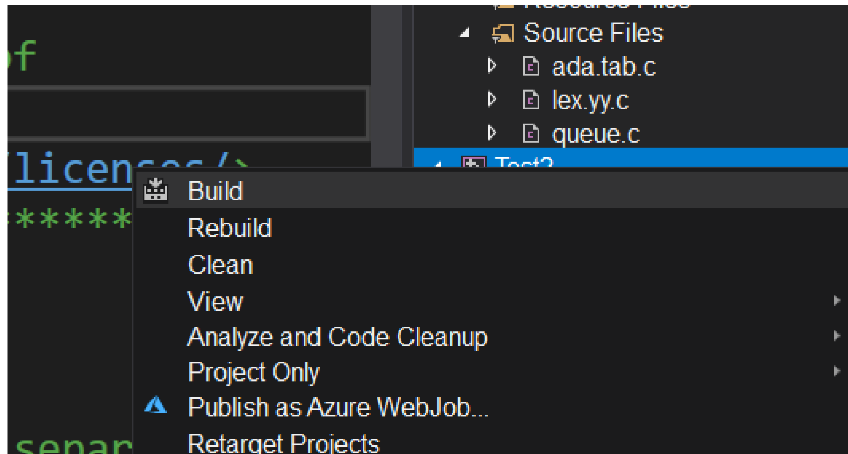


Figure 23 - the correct way to obtain executable file

« Test2 › Debug

Search Debug

Name	Date modified	Type	Size
ada_parser.exe	28/11/2020 15:25	Application	271 KB
ada_parser.ilk	28/11/2020 15:25	Incremental Linke...	753 KB
ada_parser.pdb	28/11/2020 15:25	Program Debug D...	628 KB
Test2.exe	27/11/2020 15:43	Application	76 KB
Test2.ilk	27/11/2020 15:43	Incremental Linke...	512 KB
Test2.pdb	27/11/2020 15:43	Program Debug D...	460 KB

Figure 24 - The output from compilation, with debug specified

The compilation of the Ada parser went somewhat smoother than the initial step, but there were some hitches encountered here too. Upon selecting the build option the linker stated that it was missing files related to the queue. These files were included in the C/C++ include directory as they were for the ourxref compilation. The solution that was achieved was just to place the files that it wanted directly into the project solution and then the compilation successfully completed.

6.3 Parsing

Now with the parser built it was time to run it on some test code. The parser was run on the example code from figure 8 and a supplied example code from Mika relating to dates.

To begin parsing from the command line the following command is entered: mika_ada_parser -M"C:\Mika\bin" -f"C:\GNAT\2010\bin" -gnat05 -d "file to parse". The parser executable was copied into the folder where the target code was present and the above command line entry was tried. Environmental variables had been set for both the Mika\bin and the GNAT\2010\bin but unfortunately there was no bin folder in the Mika project and no way to compile it yet to produce one. A GUI version of Mika had been downloaded to provide examples for how it worked earlier so the \bin folder in that version was referenced instead and the parsing completed.

Initially with a very limited understanding of how the parser actually worked, a study of a section of the bison file and how it was parsed in relation to the two sample pieces of code was undertaken. The section in question being the IF statement and how a parser sees that.

```
newParsing > ada_parser > ada.y
2677                                  ;
2678
2679 if_statement : IF cond_clause_list else_opt END IF ';'
2680                {$$ = malloc((SAFETY+strlen($2)+strlen($3)+14) );
2681                 strcpy($$, "if_stmt([");
2682                 strcat($$, $2);
2683                 strcat($$, "], ");
2684                 strcat($$, $3);
2685                 strcat($$, ")");
2686                 free($2);
2687                 free($3);
2688                }
2689              ;
2690
```

Figure 25 - IF statement from the ada.y grammar definitions

In figure 20 using bison the rules of how an if statement in Ada is formed is given, the definition on line 2679, if_statement : IF cond_clause_list else_opt END IF ';'. Here it is given a rule that an if statement (if_statement) consists of an if (IF), followed by a list of conditional clauses (cond_clause_list). This can be a list of just one conditional clause or many, followed by an optional else statement (else_opt). This does not need to be present to

make a valid if statement, but it is looked for all the same, followed by an end if statement (END IF) and lastly a semicolon (';').

Underneath this definition is how the parser will build up the if statement from the tokenization of the source code. The next line on 2680, {$$ = malloc((SAFETY+strlen($2)+strlen($3)+14)); has a few interesting qualities, firstly the '$$' is the 'if_statement' from the above line and this line of code is an assignment statement, firstly allocating memory for the variable using malloc [30] which is a C language function which allocates memory of a given size and returns a pointer to it. As seen in figure 21 'SAFETY' was defined as 5 within the program, so 5 is added to the length of the second token ($2) and added to the third token ($3), plus 14. Once the memory has been allocated to $$ (if_statement) the code precedes to copy in a string and concatenate further strings to it.

```
#define SAFETY 5                        //number of characters added to malloc
```

Figure 26 - The definition of safety to being the integer 5

Starting with the string "if_stmt([", followed by concatenating the string value of the second token ($2) which was the list of conditional clauses (cond_clause_list), followed by another string "], ", followed by the third token ($3) which was the optional else (else_opt) and finally another string ")". This leads to the building up of the string seen below in figure 22, consisting of lines 164 to 168.

```
C: > Ada test code > Runtime errors > example_mika > example.pl
163   stmts([
164               if_stmt([if_clause(bran(1, deci(1, cond(1, A_367 <> 0))),
165               stmts([
166               assign(Y_370, 3 + X_369)
167                   ]))],
168               else(stmts([]))),
169               if_stmt([if_clause(bran(2, deci(2, cond(2, B_368 = 0))),
170               stmts([
171               assign(X_369, 2 * (A_367 + B_368))
172                   ]))],
173               else(stmts([]))),
174               assign(A_367, 100 / (X_369 - Y_370))
175         ]),
176     no_exceptions)
177
178                   ))
179   ,
```

Figure 27 - How the parser was used to build up a prolog term in example.pl

There is a lot more happening to make up the list of conditional clauses and that is what shall next be looked into.

6.4 Parsing additions

Having tackled the division by zero check in the parser, the next step was to move on to other types of runtime errors. The first that came to mind was an index out of bounds error, for example trying to access the sixth element in an array of length five.

In Ada a CONSTRAINT_ERROR [31] is raised whenever an attempt is made to violate a range constraint. This is an exception type within Ada itself and is used for other errors as well, such as overflows, null dereferences and the previously tackled division by zero.

Let us have a look at some of these errors in practice:

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Constraint is
    type Custom_Int is range 0 .. 5;
    X: Custom_Int := 6;
begin
    Put("Did our assignment work?");
end Constraint;
```

Figure 28 - constraint.adb example used to display constraint errors

```
C:\Users\derry\OneDrive\Desktop\random code>gcc -c constraint.adb
constraint.adb:4:22: warning: value not in range of type "Custom_Int" defined at line 3
constraint.adb:4:22: warning: "Constraint_Error" will be raised at run time

C:\Users\derry\OneDrive\Desktop\random code>gnatbind constraint

C:\Users\derry\OneDrive\Desktop\random code>gnatlink constraint

C:\Users\derry\OneDrive\Desktop\random code>constraint

raised CONSTRAINT_ERROR : constraint.adb:4 range check failed
```

Figure 29 - output from the compilation and execution of constraint above

In figure 23 above we see the creation of a custom integer type that has a range from 0 to 5. Next the variable X, which is of the Custom_INT type, is attempted to be assigned a value of 6. We can see that the compiler helpfully told us that a constraint error would be raised at run time and we can see from the code that it indeed halted execution of the code as the print out of the string to the console was never reached. This is the most basic example and we will delve deeper into this.

```
1  with Ada.Text_IO; use Ada.Text_IO;
2  procedure Constraint is
3      type Custom_Int is range 0 .. 5;
4      X: Custom_Int := 4;
5  begin
6      X := (X + 2);
7      Put(Custom_Int'Image(X));
8  end Constraint;
```

Figure 30 - Ada code with custom type and custom range

```
C:\Users\derry\OneDrive\Desktop\random code>gcc -c constraint.adb
constraint.adb:6:13: warning: value not in range of type "Custom_Int" defined at line 3
constraint.adb:6:13: warning: "Constraint_Error" will be raised at run time

C:\Users\derry\OneDrive\Desktop\random code>constraint

raised CONSTRAINT_ERROR : constraint.adb:6 range check failed

C:\Users\derry\OneDrive\Desktop\random code>
```

Figure 31 - Output showing the constraint error of exceeding range of type

From the next example we can see that the compiler is still smart enough to see that a constraint error will be raised even though X is initialized within the range specified it is added to during execution and raises the error. What if the assignment addition happened in a different branch?

```
1   with Ada.Text_IO; use Ada.Text_IO;
2   procedure Constraint is
3       type Custom_Int is range 0 .. 5;
4       X: Custom_Int := 4;
5       Y: Custom_INT := 2;
6   begin
7       if (X / Y) > Y then
8           X := (X + 1);
9       elsif (X / Y) = Y then
10          X := (X + (X / Y));
11      end if;
12      Put(Custom_Int'Image(X));
13  end Constraint;
```

Figure 32 - Expanding the complexity while testing ranges

```
C:\Users\derry\OneDrive\Desktop\random code>gcc -c constraint.adb

C:\Users\derry\OneDrive\Desktop\random code>constraint

raised CONSTRAINT_ERROR : constraint.adb:10 range check failed

C:\Users\derry\OneDrive\Desktop\random code>
```

Figure 33 - Compiler still finds the error even when inside an else branch

As can be seen in the above code the compiler warning about constraint errors was not present during compilation this time, yet the error was still raised at runtime. This shows how a reliance solely on the compiler to help catch these errors is unwise as a simple obfuscation of the decision paths caused the compiler to not see ahead to this potential error.

```
1   with Ada.Text_IO; use Ada.Text_IO;
2   procedure Constraint is
3
4       type Custom_Int is new Integer range 0 .. 5;
5       X: Custom_Int := 4;
6       Y: Custom_Int := 1;
7   begin
8       if (X / Y) > Y then
9           X := (X + 1);
10      elsif (X / Y) = Y then
11          X := (X + 2);
12      end if;
13      Put_Line("Custom_Int'First = " & Custom_Int'Image(Custom_Int'First));
14      Put_Line("Custom_Int'Last = " & Custom_Int'Image(Custom_Int'Last));
15      Put_Line("Integer'First = " & Integer'Image(Integer'First));
16      Put_Line("Integer'Last = " & Integer'Image(Integer'Last));
17  end Constraint;
```

Figure 34 - Ada code to showcase the 'First & 'Last of types

```
C:\Users\derry\OneDrive\Desktop\random code>constraint
Custom_Int'First =  0
Custom_Int'Last =  5
Integer'First = -2147483648
Integer'Last =  2147483647
```

Figure 35 - Output from fig 34

In the above example we can see that any specified type range is accessible through the use of the TYPE'First and TYPE'Last [32]. This can be used for other structures too such as arrays and since array out of bounds is another type of runtime error that is of interest.

```
1  with Ada.Text_IO; use Ada.Text_IO;
2  procedure Constraint is
3      type Custom_Int is new Integer range 0 .. 5;
4      type Index is range 11 .. 15;
5      --                   ^ Low range can be any value, not just 0
6      type Custom_Int_Array is array (Index) of Custom_Int;
7      Array1: Custom_Int_Array := (1,2,3,4,5);
8      X: Custom_Int := 4;
9      Y: Custom_Int := 1;
10 begin
11     for I in Index loop
12         Put_Line(Custom_Int'Image (Array1(I)));
13     end loop;
14     Put_Line("Array1'First = " & Index'Image (Array1'First) & " Array1'Last = " & Index'Image (Array1'Last));
15     Put_Line("Next we will attempt to enter 5 into Array(10) which is outside of the range of the array");
16     Array1 (10) := 5;
17 end Constraint;
```

Figure 36 - A further exploration of the range of a type

```
C:\Users\derry\OneDrive\Desktop\random code>constraint
 1
 2
 3
 4
 5
Array1'First =  11 Array1'Last =  15
Next we will attempt to enter 5 into Array(10) which is outside of the range of the array

raised CONSTRAINT_ERROR : constraint.adb:16 range check failed
```

Figure 37 - Output from fig 36

In the above example it is shown how given an array the range of its bounds is accessible through the use of the ‘First and ‘Last also and an attempt to access an index outside of that range produces a constraint error at runtime. This will be useful in checking if a runtime error would be possible whenever an array index is being accessed.

```
procedure Constraint is
    type Custom_Int is new Integer range 0 .. 50;
    type Index is range 1 .. 10;
    --              ^ Low range can be any value, not just 0
    type Custom_Int_Array is array (Index) of Custom_Int;
    Array1: Custom_Int_Array := (1,2,3,4,5,6,7,8,9,10);
    X: Custom_Int := 4;
    Y: Custom_Int := 1;
    W: Index := 2;
    R: Index := 4;
    T: Custom_Int := 42;
begin
    if 10 /= 0 then
            T := 10;
        end if;
    Array1 (R) := T;
```

Figure 38 - A sample Ada program with an array used within it

Using an example procedure from above we assign into the created array1. After running this file through the Mika_ada_parser we can see how it breaks this down.

```
stmts([
                if_stmt([if_clause(bran(1, deci(1, cond(1, 10 <> 0))),
                stmts([
                assign(T_373, 10)
                    ]))],
                else(stmts([]))),
                assign(indexed(Array1_368, [R_372]), T_373)
    ]),
  no_exceptions)
```

Figure 39 - How the array is represented within the generated constraint.pl file

We can see above that it sees it as an

`indexed(Array1_368, [R_372])`

What needs to be inserted will be something akin to:

`indexed(Array1_368, [rune(1, R_372 > Array1_368'Last || R_372 < Array1'First, R_372) ])`

Next step is to find the parsing rule in ada.y where the changes need to be made.

```
/*can be a array access, a function/procedure call, a type conversion, or a subtype_indication with index_contraint */
/*we will have to differentiate between many things*/
indexed_component : name '(' value_list ')'
                  {$$ = malloc(SAFETY+strlen(tmp_s)+strlen($1)+strlen($1)+strlen($1)+strlen($3)+strlen($3)+strlen($3)+43);
                   itoa(runtime_nb++, tmp_s, 10);
                   print_coverage_details(RUNE, tmp_s, current_unit, yylineno, column+1);
                   strcpy($$, "indexed(");
                   strcat($$, $1);
                   strcat($$, ", [");
                   strcat($$, "rune(");
                   strcat($$, tmp_s);
                   strcat($$, ", ");
                   strcat($$, $3);
                   strcat($$, " > ");
                   strcat($$, $1);
                   strcat($$, "\'Last || ");
                   strcat($$, $3);
                   strcat($$, " < ");
                   strcat($$, $1);
                   strcat($$, "\'First,");
                   strcat($$, $3);
                   strcat($$, ")");
                   strcat($$, "])");
                   free($1);
                   free($3);
                  }
```

Figure 40 - Additions in the ada.y file for when an indexed_component is encountered

The first implementation was unsuccessful as everything between the apostrophes from ‘Last to ‘First was interpreted as a string by prolog so another approach had to be taken and there was already a solution in the parser that had not been encountered by the author. `TIC(object, (range || first || last))` can be used to get the values associated with array’first or array’last without the use of the apostrophe that got confused for a string below.

```
stmts([
            if_stmt([if_clause(bran(1, deci(1, cond(1, 10 <> 0))),
            stmts([
            assign(T_373, 10)
                ]))],
            else(stmts([]))),
            assign(indexed(Array1_368, [rune(34, R_372 > Array1_368'Last || R_372 < Array1_368'First,R_372)]), T_373)
      ]),
  no_exceptions)
```

Figure 41 - constraint.pl after the additions to the ada.y file had been implemented initially

After the needed changes were made in the parser the resulting output to the prolog file looked more syntactically correct.

```
stmts([
            if_stmt([if_clause(bran(1, deci(1, cond(1, 10 <> 0))),
            stmts([
            assign(T_373, 10)
                ]))],
            else(stmts([]))),
            assign(indexed(Array1_368, [rune(34, R_372 > tic(Array1_368, last) || R_372 < tic(Array1_368, first) ,R_372)]), T_373)
      ]),
  no_exceptions)
```

Figure 42 - constraint.pl after using the tic() function instead of the character ‘

There were more difficulties to overcome and these will be covered in section 7 in more detail.

The next step would be to implement the next runtime check on number overflows. Ada has a very good check for these already, but as the code branches off into different decisions the compiler tends to not pick up every possibility. Using another simple program shown below the possibilities of this error will be explored.

```
1   with Ada.Text_IO; use Ada.Text_IO;
2   procedure numberOverflow is
3       type Custom_Int is new Integer range 0 .. 50;
4       X: Custom_Int := 4;
5       Y: Custom_Int := 1;
6       T: Custom_Int := 42;
7   begin
8       Put_Line(Custom_Int'Image(T));
9   end numberOverflow;
```

Figure 43 - Sample Ada code used to test number overflows

Here there is a procedure called numberOverflow, inside there is a custom integer type called Custom_Int made with a range from 0 to 50 and three Custom_Int variables X, Y and T. Initially the value of T is output to the console.

```
C:\Users\derry\Documents\GitHub\MikaRuntimeError\RuntimeErrors>numberOverflow
 42
```

Figure 44 - Output from fig 43

Next let us try to go outside of the range of this type by subtracting T from X.

```
7   begin
8       Y := X - T;
9       Put_Line(Custom_Int'Image(Y));
10  end numberOverflow;
11
```

Figure 44 - change in the code to exceed the range of Custom_Int X(4) -T(42) = -38 outside the range of Custom_Int

As was expected the compiler caught this constraint error during compilation and raised a warning and upon execution the constraint_error was raised.

```
C:\Users\derry\Documents\GitHub\MikaRuntimeError\RuntimeErrors>gnatmake numberOverflow.adb
gcc -c numberoverflow.adb
numberoverflow.adb:8:12: warning: value not in range of type "Custom_Int" defined at line 3
numberoverflow.adb:8:12: warning: "Constraint_Error" will be raised at run time
gnatbind -x numberoverflow.ali
gnatlink numberoverflow.ali

C:\Users\derry\Documents\GitHub\MikaRuntimeError\RuntimeErrors>numberOverflow

raised CONSTRAINT_ERROR : numberoverflow.adb:8 range check failed
```

Figure 45 - Compiler warning and error on the execution of the code from fig 44

7. Limitations of implementation

Having started out with some success implementing the runtime error check for division by zero in the Ada parser, it seemed like it would be straightforward to then move on to other runtime errors and insert checks for them also inside the parser. After setting about to do just that there were a number of areas of concern that came to light, initially taking an array out of bounds as an example. The idea being that whenever an array index was referenced that there would be a check to make sure it was within the range of array'First and array'Last [32] (Ada implementation to return the lower and upper bounds of an object with a range).

The type of the array was available to be used within Mika and that was something of great use, as without that, it would have been impossible to get the range values returned. But pressing forward another problem was raised. When referencing an array index the syntax `Array (i)` is used, this is very similar structurally to how one would call a function or procedure, e.g

```
Function Square (A: Integer) return Integer is

Begin

    Return A * A;

End Add;
```

This function would be called " `Square (2)` " which is structurally identical to the array index reference. That is not the end of the similarities though, type conversions or sometimes known as casting also takes the same format, and an example of numeric type conversion would be `Integer(1.6)` [33] would return the integer 2.

The parser when it encounters any of these behaves in the same way in breaking the code up into tokens under the indexed_component [34]. Now if

the original approach was taken to take the value from inside the parentheses and test it against the range of what preceded it would work fine for an array, but not so much for a function, as it does not have a range and calling Square'First would throw its own exception. Using an example program called constraint as a testing environment, shown below.

```
2   procedure Constraint is
3       type Custom_Int is new Integer range 0 .. 50;
4       type Index is range 1 .. 10;
5       type Custom_Int_Array is array (Index) of Custom_Int;
6       Array1: Custom_Int_Array := (1,2,3,4,5,6,7,8,9,10);
7       X: Custom_Int := 4;
8       Y: Custom_Int := 1;
9       W: Index := 2;
10      R: Index := 4;
11      T: Custom_Int := 42;
12  begin
13      if 10 /= 0 then
14              T := 10;
15          end if;
16      Array1 (R) := T;
17  end Constraint;
```

Figure 46 - Compiler warning and error on the execution of the code from fig 44

There is only one reference to an indexed element on line 16 where `Array1 (R) := T;` yet when run through the mika_ada_parser having made the following changes to the ada parser:

```
1900 indexed_component : name '(' value_list ')'
1901     {$$ = malloc(SAFETY+strlen(tmp_s)+strlen($1)+strlen($1)+strlen($1)+strlen($3)+strlen($3)+strlen($3)+56);
1902      itoa(runtime_nb++, tmp_s, 10);
1903      print_coverage_details(RUNE, tmp_s, current_unit, yylineno, column+1);
1904      strcpy($$, "indexed(");
1905      strcat($$, $1);
1906      strcat($$, ", [");
1907      strcat($$, "rune(");
1908      strcat($$, tmp_s);
1909      strcat($$, ", ");
1910      strcat($$, $3);
1911      strcat($$, " > ");
1912      strcat($$, "tic(");
1913      strcat($$, $1);
1914      strcat($$, ", last) || ");
1915      strcat($$, $3);
1916      strcat($$, " < ");
1917      strcat($$, "tic(");
1918      strcat($$, $1);
1919      strcat($$, ", first) ,");
1920      strcat($$, $3);
1921      strcat($$, ")");
1922      strcat($$, "])");
1923      free($1);
1924      free($3);
1925     }
1926    ;
```

Figure 47 - Ada.y additions for indexed_component, including the tic() in place of the character '

The output in the prolog file constraint.pl is as such:

```
stmts([
            if_stmt([if_clause(bran(1, deci(1, cond(1, 10 <> 0))),
            stmts([
            assign(T_373, 10)
                ]))],
            else(stmts([]))),
            assign(indexed(Array1_368, [rune(34, R_372 > tic(Array1_368, last) || R_372 < tic(Array1_368, first) ,R_372)]), T_373)
    ]),
  no_exceptions)
```

Figure 48 - Source code reconstructed in the constraint.pl file

Everything seems to have translated well into the prolog file here until the first parameter in the rune() call is examined. This is a counter used to track how many times rune has been called for backtracking purposes and as there is only one indexed component here the assumption would be that it would have a value of 1 here too. Upon deeper investigations into the prolog file to see what happened it is found that within the package specification, in GNAT or MIKA is an assignment of a subtype indication of characters as Ascii characters. These are stored as enumerators and referenced at the start of the compilation and this has inadvertently called the rune() for each of these character subtype indications.

```
package_specification(
 Ascii_274,
 local_decl([  object(constant,
   [Nul_ascii_275],
   subtype_indication(may_be_null,Character_15, no_constraint),
   indexed(tic(Character_15, val),
   [rune(1, 0 > tic(tic(Character_15, val), last) || 0 < tic(tic(Character_15, val), first) ,0)])),
```

Figure 49 - an unexpected outcome within constraint.pl, rune was being called inside the package_specifications

This should not cause an error as it should still fall within valid ranges of the character array, but it is an unnecessary extra 33 checks for prolog to go through. And if there was some form of function call or procedure call within the program then it would not work at all. The parser just can't tell the difference between these similar looking entities and as such it would make it very difficult to use going forward in real code where the use of functions and procedures is everywhere. If the step were taken to insert these checks for runtime exceptions within the symbolic execution stage instead,it would have more flexibility in determining what type of object we are working with at any given stage. Thus we could focus the index out of range to specifically those objects that would be at risk of causing this. This was an unforeseen problem at the inception of this project and although there has been success in the implementation of the division by zero and array out of bounds exception it would be a very inefficient way of proceeding with the application to other exceptions as the ambiguity would only increase with every exception added. The way to continue on from here would be to have the checks conducted from with in the symbolic executor as all the type information is available there, this would result in a narrowing down of the element under inspection, for instance if an array was being looked for, with in the symbolic executor they are know as an array, whereas at the parsing stage they are the much more generic "indexed element" which covers far too many elements.

8. Mika Extension for Text Editor

With the project branching out into the implementation of an extension to allow developers to dynamically query their Ada code within a text editor. The text editor that the extension would be written in had to be picked. Below the pros and cons of some of the most popular text editors [50] will be discussed and an eventual choice will be selected.

- Sublime Text

  Sublime Text is a text editor for code and markup. There is a free tier and a premium tier available. One of the major drawbacks of this text editor besides the fact that there is a price involved, is the fact that it is not an open source software. It does have a nice user interface which is intuitive and easy to navigate.

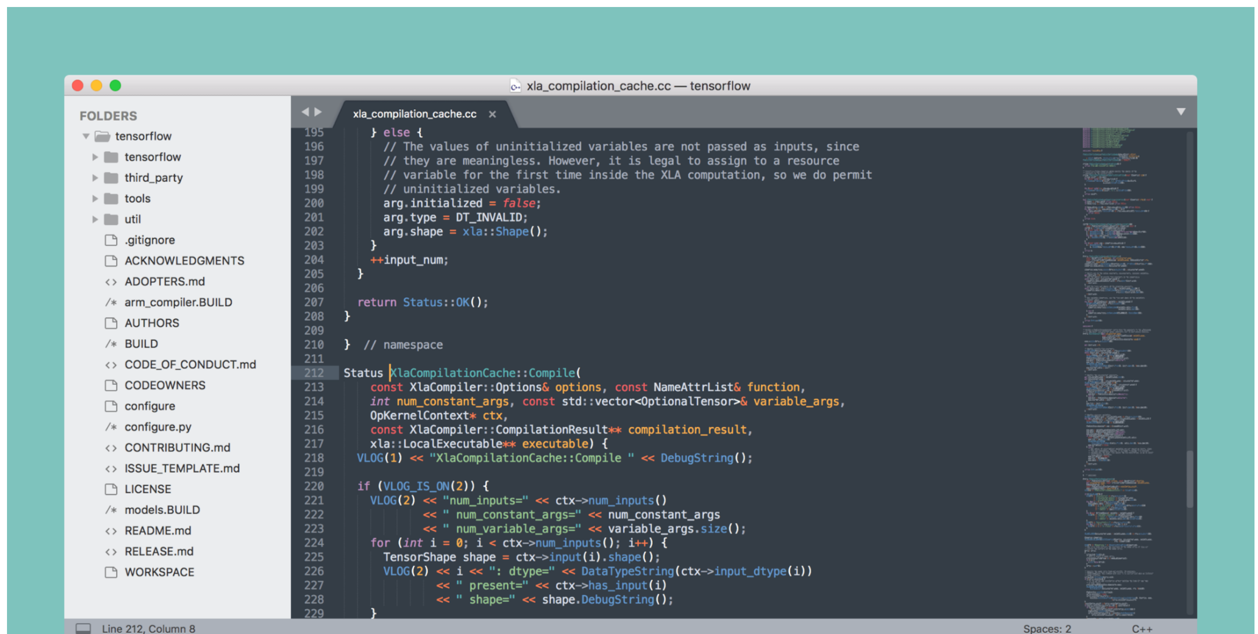


Figure 50 - Example of Sublime Text GUI

  For the purpose of creation of extensions (plugins), they are written in python. Being in python comes with its benefits of being familiar and popular. It also has access to all of Pythons packages which is extensive. There are a number of excellent starter guides also for the

creation of these plugins, with the best found being at Envatotuts+ [51].

- Emacs

Figure 51 - Example of Emacs GUI

Extensions for Emacs are written in Lisp. Lisp is an old language that was initially developed in 1958 by John McCarthy, it uses a fully parenthesized prefix notation as shown below.

```
42
43  ;Erase all dimensions
44  (command ".erase" (ssget "X" (list(cons 0 "DIMENSION"))) "")
45
46  ;explode all mtext
47  (command "._explode" (ssget "_X" '((0 . "MTEXT"))) "" )
48
49  text style set option 1
50  (while (setq st (tblnext "STYLE" (not st)))
51          (setq st (entget(tblobjname "STYLE" (cdr(assoc 2 st)))))
52           (entmod (subst '(3 . "isocp.shx")(assoc 3 st) st))
53  )
54
55    (if (setq tss (ssget "_X" '((0 . "*TEXT"))))
56      (repeat (setq n (sslength tss))
57        (setq tdata (entget (ssname tss (setq n (1- n)))))
58        (entmod (subst '(7 . "STANDARD") (assoc 7 tdata) tdata))
59      ); repeat
60    ); if
61
62  )
```

Figure 52 - Example of Lisp code

Emacs is a complex Text editor that is highly customizable with many different extensions. This is very useful, but this complexity comes at the cost of the basic Emacs not being to everyone's tastes and only achieving desired usability once all necessary extensions are installed. This can also lead to an issue of using Emac on multiple devices, the experience is not the same until all extensions are installed again.

It is also a command line interface (CLI) centric text editor, favouring keyboard shortcuts and command line commands to operate tasks. Emacs does have a basic graphical user interface (GUI) but as the stack overflow survey [50] suggests GUI heavy editors are increasing their market share and increasing in popularity with Visual Studio code accounting for over 50% of user preferences while Emac managing 4.5% and another CLI based editor VIM making up 25.4%.

Emacs is particularly geared towards Linux and with Mika being only available on the Windows platforms this combined with some of the other points mentioned above has ruled it out as a contender for the editor to be selected for this extension.

- Atom text

Figure 53 - Example of Atom Text GUI

Atom text is an electron based editor. Electron is a framework for development of GUI applications using web technologies such as Chromium and Node.js.

Both Atom text and Electron were developed by GitHub and are open source. Atom Text was released in 2014, Microsoft forked a version of Atom Text as their base for Visual Studio Code which they released in 2015, before Microsoft's eventual purchase of GitHub in 2018 for $7.5 Billion.

Extensions in Atom Text are written in JavaScript and have a healthy extension building community, although its popularity has fallen off

due to being so similar to Visual Studio Code (VS code) and Microsoft owning both seem to be favouring the development of VS code.

- Notepad++

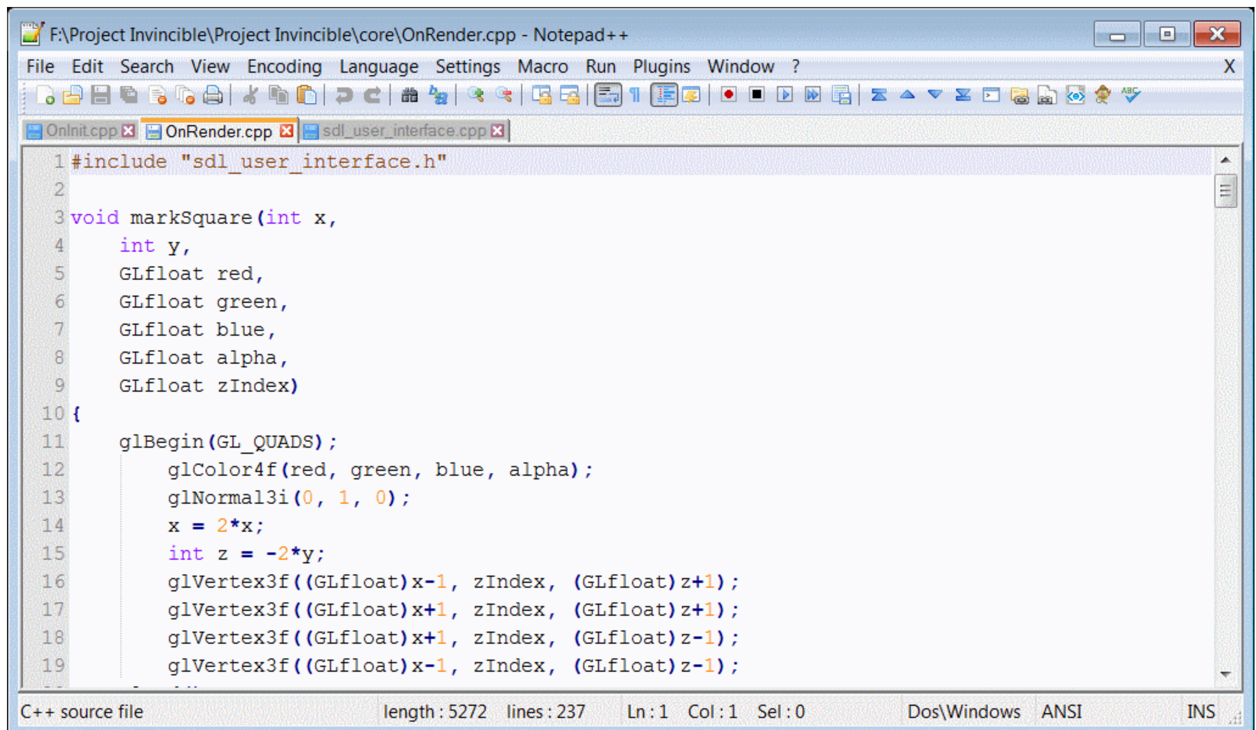


Figure 54 - Example of Notepad++ GUI

Notepad++ is a very popular text editor that has a reputation for being able to open any file needed, but it's extension experience is not very mature or well supported [52]. Extension are written in C++, with Ada, C#, 'D' and Dephi as other languages supported. The idea of writing an Ada extension in Ada was quite enticing, unfortunately the documentation on the Ada support was missing. Notepad++ is also an open source project and it only runs on windows.

- Visual Studio Code

Visual Studio Code (VS code), owned by Microsoft and released in 2015 it rose in popularity rapidly gaining over 50% of the stack

overflow most popular development environments and tools vote in a survey carried out on over 87,000 developers [50].

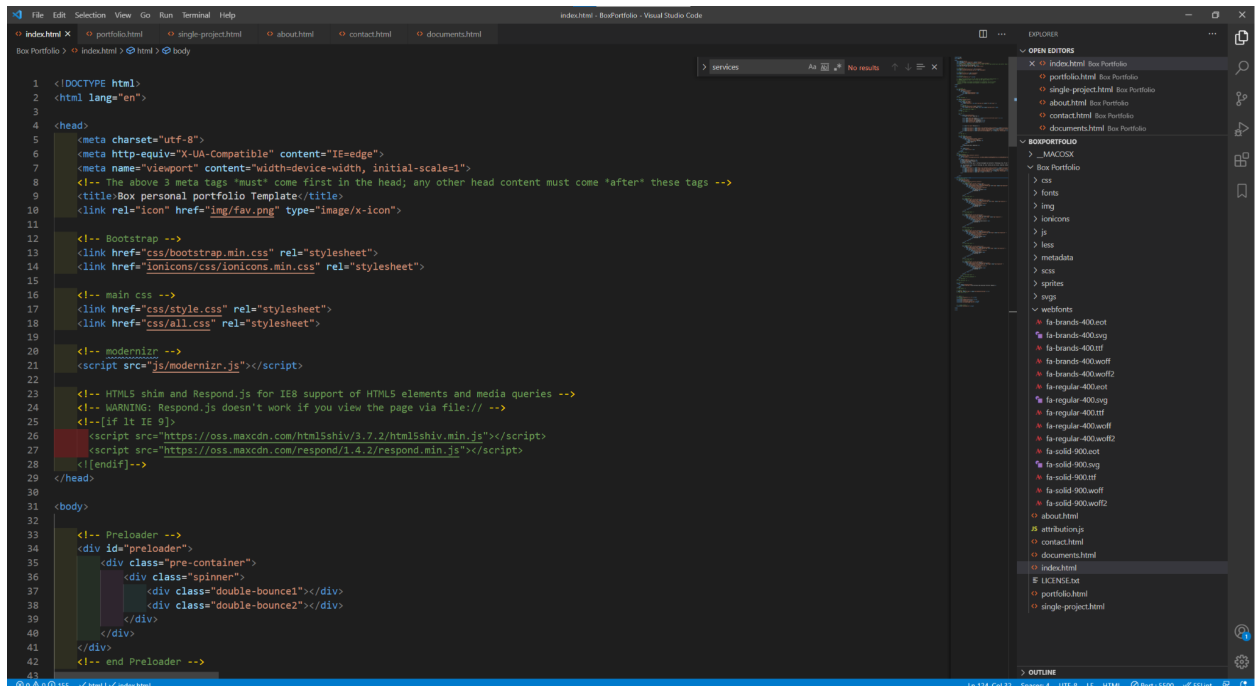


Figure 54 - Example of Visual Studio Code GUI

Extensions for VS code are developed in either Typescript or JavaScript, with Typescript being a strict syntactical superset of JavaScript and adds optional static typing to the language.

As with Atom Text which VS code was based upon it is both an open source project and based upon the Electron framework. This has led to a highly expanding Extension development with over 120 extensions already achieving over 1 million downloads each [53]. No statistics for how many total extensions are available but new extensions are being continuously released.

VS code is a cross platform application that runs on Windows, Apple, Linux and Arm systems such as Arduino. This far reach is also a reason for its popularity along with the fact that all extensions and settings are sync able across devices using signing into a Microsoft account, no other text editor offers this convenience.

After consideration of the possible text editors in which to write the Mika extension the final choice was Visual Studio Code, based primarily on its popularity. With such a reach among developers it is likely that this would also reach more Ada developers and Mika users than any other platform.

8.1 Visual Studio Code Extension For Mika

After the realization that the parser was not going to be the correct area to do the runtime checks, another idea of doing an extension for Mika within Visual Studio Code was presented. The idea being that the extension would allow a developer to insert a special annotation into their code at a preferred line, along the lines of "--#MIKA " followed by variables from that program. The parser would then take this annotation and apply the needed check into the prolog code to make the test generation attempt to match what was supplied. For example if the comment "--#MIKA Y == 4 and X == 2" were inserted, the symbolic executor would then try to provide test cases to make both these statements true by this line in the follow of execution if there was a combination available to make it true. It should be noted that any boolean expression works here such as records, enumerations, arrays and other function calls not just integers and floats.

In order to write a Visual Studio Code extension you need to have node.js installed and then either npm [35] (node package manager) or yarn [36] (Yet Another Resource Negotiator) as your package manager. Yarn is a newer package manager and seems to be more favoured amongst developers. It is feature rich and made with ease of use in mind. However, the decision was taken to go with npm for the project as it is a longer standing package manager with more documentation enabling faster learning of how to use it.

The language used to write the code for the extension is a choice between typescript and javascript [54]. Typescript was developed by Microsoft with the goal being to apply typing to scripting languages such as javascript, hence the name. It is a superset of javascript and aims to make

applications written in this language more scalable than applications that use javascript.

Javascript is a scripting language that has become one of the most widely used programming languages. The author has had a small amount of experience with javascript and the decision was made to apply this option to this area of the project as it is more familiar, although typescript seems like it might be the best option if the extension was to be of greater size. With a project of greater size conforming to strongly types enforced by typescript would be beneficial and it may be a choice for future development if this continues to be worked upon. The application of static types and classes, modules and interfaces helps to construct a much more cohesive application. For this extension though as it has a relatively small scope JavaScript comes out as being the correct choice.

Expanding upon this extension in future to make it more user friendly and supply answers in a popup rather than a new tab could be investigated. Visual Studio Code's extension guidelines [55] do state that over usages of notifications should not be overused. Although this is definitely an area where improvements can be made in the future.

9. Testing the Limitations of Mika

In order to put the new exception finding feature of Mika to the test a number of tests will be carried out to determine if an otherwise reachable exception is found by the test driver.

The first test was involving loops and a small program was written:

```
C: > Ada test code > Runtime errors > adaloops.adb > ...
1   package body adaloops is
2   procedure test(X: out Integer) is
3       B : Integer := 100;
4   begin
5       while B > -1 loop
6           X := 10000 / B;
7           B := B - 1;
8       end loop;
9   end test;
10  begin
11      null;
12  end adaloops;
```

Figure 53 - A sample program that would produce a division by zero once B reached 0

The Integer B starts at 100 and decrements down over the course of the loop, once it reaches 0 and an exception should be raised but when run with the exception flag in Mika it runs out of memory before the exception is reached.

```
MIKA : END Elaborating
! Resource error: insufficient memory
Mika Generator ERROR
```

Figure 54 - Output from Mika while running with the exception flag

10. Conclusion

In conclusion the Project of the Ada Runtime Error generator was very enlightening. The fact that two of the runtime errors (division by zero and array index out of bounds) have been implemented and produce correct results when run through Mika is very promising. The fact that the area of implementation within the Ada parser proved to be too ambiguous for the full range of run time errors should not detract from the fact that it is a possibility for future projects to tackle within the symbolic executor where all types are readily available.

The application of such a program would be immensely beneficial both from a software safety perspective but also financially for interested companies. The cost of automatic software verification software is quite high, AdaCore's SparkPro has a number of different versions and has pricing only on request, taking into account the size of the organisation. So an investment of time and effort into being able to provide code validation and verification as an open source project would have immense benefits.

The same technique could be applied to other languages also, not limiting the scope to just Ada. This is doubly interesting as Ada is a niche language already with a relatively small user base and has a similar tool already available with SparkPro. Whereas if applied to another language such as C or Python where there is a far greater user base and very limited tools already available the benefits could be great.

The addition of the Visual Studio Code extension also provides an interesting addition to the ways in which code can be queried, Being able to dynamically query variables on a certain line of code is both very useful for understanding the code under scrutiny but also will help with the generation of further test inputs or to see if a certain condition can be met or not without having to leave the Visual Studio Code environment. This tool could also be expanded upon in future to work along with the suggested expansion of the Runtime Error generator into other languages

making a more unified software package where the results you require are as easily accessible as possible to the user.

11. Bibliography

[1] Booch, G, Bryan, D and Petersen, C. Software Engineering with Ada, 1994, 3rd Edition Addison-Wesley, ISBN 0-8053-0608-0

[2] Whitaker, W.A., 1993. Ada—the project: the DoD high order language working group. ACM SIGPLAN Notices, 28(3), pp.225.

[3] Kádár, I. (2017). The optimization of a symbolic execution engine for detecting runtime errors. Acta Cybernetica, 23(2), 573-597. Available at: https://doi.org/10.14232/actacyb.23.2.2017.9 [Accessed 11 November 2020].

[4] Barnes, J.G.P., 2003. High integrity software: the spark approach to safety and security Pearson Education.

[5] Burns, A., 1999. The Ravenscar profile. ACM SIGAda Ada Letters, 19(4), pp.49-52.

[6] Lesk, M, and Schmidt, E., Lex - A Lexical Analyzer Generator, Available at: http://dinosaur.compilertools.net/lex/index.html [Online], Accessed on: 27/10/2020

[7] Johnson, S., Yacc: Yet Another Compiler-Compiler, Available at: http://dinosaur.compilertools.net/yacc/index.html [Online], Accessed on: 27/10/2020

[8] Barnes, J. High Integrity Ada: The Spark approach, 1997, Addison-Wesley, ISBN 0-201-17517-7

[9] Spark Team, 2011, User Manual [online] Available from: https://docs.adacore.com/sparkdocs-docs/Simplifier_UM.htm [accessed 28/10/2020]

[10] Spark Team, 2011, Examiner User Manual [online] Available from: https://docs.adacore.com/sparkdocs-docs/Examiner_UM.htm [accessed 28/10/2020]

[11] Anon, Reactive Systems Inc, 2011 Finding Bugs in C Programs with Reactis for C [online] Available from: https://www.reactive-systems.com/c-runtime-errors.html [accessed 28/10/2020]

[12] Brailsford, S., Potts, C. and Smith, B., 1999. Constraint satisfaction problems: Algorithms and applications. European Journal of Operational Research, [online] 119(3), pp.557-581. Available at: https://www.sciencedirect.com/science/article/pii/S0377221798003646 [Accessed 1 November 2020].

[13] Ovstedal, E., 1991. Using Fault Tree Analysis in Developing Reliable Software. IFAC Proceedings Volumes, [online] 24(13), pp.77-82. Available at: https://www.sciencedirect.com/science/article/pii/S1474667017513695 [Accessed 1 November 2020].

[14] Murphy, C. Kim, E., Kaiser, G. Cannon, A, 2008, Backstop: a tool for debugging runtime errors [online] Available at: https://www.cs.columbia.edu/wp-content/uploads/sites/7/2011/03/3477-Murphy-Backstop-SIGCSE2008.pdf [Accessed 3 November 2020].

[15] Mitchell, G., 2000. Placebo defense: Operation desert mirage? The rhetoric of patriot missile accuracy in the 1991 Persian Gulf War∗. Quarterly Journal of Speech, [online] 86(2), pp.121-145. Available at: https://nca.tandfonline.com/doi/abs/10.1080/00335630009384286 [Accessed 5 November 2020].

[16] G. Le Lann, "An analysis of the Ariane 5 flight 501 failure-a system engineering perspective," Proceedings International Conference and Workshop on Engineering of Computer-Based Systems, Monterey, CA, USA, 1997, pp. 339-346, doi: 10.1109/ECBS.1997.581900. Available at: https://ieeexplore.ieee.org/abstract/document/581900 [Accessed on 5 November 2020].

[17] Lazic, Ljubomir. (2006). Software Errors Analysis and Detection:A Survey. Available at: https://www.researchgate.net/publication/323548522_Software_Errors_Analysis_and_DetectionA_Survey [Accessed on 5 November 2020].

[18] Baldoni, R., Coppa, E., D'elia, D., Demetrescu, C. and Finocchi, I., 2018. A Survey of Symbolic Execution Techniques. ACM Computing Surveys, [online] 51(3), pp.1-39. Available at: http://goo.gl/Hf5Fvc [Accessed 8 November 2020].

[19] Microsoft, 2020. Role Of Symbolic Execution In Software Testing, Debugging And Repair. [video] Available at: https://www.youtube.com/watch?v=pkm3ItOLfRs&ab_channel=MicrosoftResearch [Accessed 8 November 2020].

[20] Coursera, 2020. Basic Symbolic Execution. [video] Available at: https://www.coursera.org/lecture/software-security/basic-symbolic-execution-U9R38 [Accessed 8 November 2020].

[21] MIT OpenCourseWare, 2020. Symbolic Execution. [video] Available at: https://www.youtube.com/watch?v=yRVZPvHYHzw&ab_channel=MITOpenCourseWare [Accessed 8 November 2020].

[22] Hořeňovský, M., 2018. codingnest.com. [Online] Available at: https://codingnest.com/modern-sat-solvers-fast-neat-underused-part-1-of-n/ [Accessed 08 November 2020].

[23] R.KhudaBukhsh, A., 2015. www.sciencedirect.com. [Online] Available at: https://www.sciencedirect.com/science/article/pii/S0004370215001678 [Accessed 08 November 2020].

[24] Sanchit Batra, A. R., 2018. cse.buffalo.edu. [Online] Available at: https://cse.buffalo.edu/~erdem/cse331/support/sat-solver/index.html#:~:text=A%20SAT%20solver%20is%20a,UNSAT%20if%20it%20is%20not. [Accessed 08 November 2020].

[25] Budinich, M., 2019. The Boolean SATisfiability Problem in Clifford algebra. Theoretical Computer Science, [online] 784, pp.1-10. Available at: https://www.sciencedirect.com/science/article/pii/S0304397519301938 [Accessed 11 November 2020].

[26] Meudec, C., 2001, AT Gen: Automatic Test Data Generation using Constraint Logic Programming and Symbolic Execution Software Testing Verification and Reliability. [Online] Available at: https://doi.org/10.1002/stvr.225 [Accessed 11 November 2020].

[27] Docs.adacore.com. 2020. Formal Verification With Gnatprove — SPARK User's Guide 22.0W. [online] Available at: https://docs.adacore.com/spark2014-docs/html/ug/en/gnatprove.html [Accessed 11 November 2020].

[28] Adacore.com. 2020. About Ada - Adacore. [online] Available at: https://www.adacore.com/about-ada [Accessed 14 November 2020].

[29] Kulkarni, M., 2020. ECE 468 - Fall 2017. [online] Engineering.purdue.edu. Available at: https://engineering.purdue.edu/~milind/ece468/2017fall/assignments/step2/ [Accessed 14 November 2020].

[30] Point, T., 2020. C Library Function - Malloc() - Tutorialspoint. [online] Tutorialspoint.com. Available at: https://www.tutorialspoint.com/c_standard_library/c_function_malloc.htm [Accessed 29 November 2020].

[31] Stanbrough, D., n.d. *Exceptions*. [online] Goanna.cs.rmit.edu.au. Available at: http://goanna.cs.rmit.edu.au/dale/ada/aln/11_exceptions.html [Accessed 7 February 2021].

[32] Barnes, J. and Fauconnier, H., 2001. *Programmer en ADA 95*. 2nd ed. Paris: Vuibert, p.117.

[33] Europe, A., 2006. *Type Conversions*. [online] Adaic.org. Available at: <https://www.adaic.org/resources/add_content/standards/05rm/html/RM-4-6.html> [Accessed 17 February 2021].

[34] Europe, A., 2006. *Indexed Components*. [online] Ada-auth.org. Available at: <http://www.ada-auth.org/standards/12rm/html/RM-4-1-1.html> [Accessed 17 February 2021].

[35] npm, D., 2021. *Packages and modules | npm Docs*. [online] Docs.npmjs.com. Available at: <https://docs.npmjs.com/packages-and-modules> [Accessed 24 February 2021].

[36] Yarn, D., 2021. *Plug'n'Play*. [online] Yarnpkg.com. Available at: <https://yarnpkg.com/features/pnp#the-node_modules-problem> [Accessed 24 February 2021].

[37] Bar-Gil, G., 2020. *NPM vs. Yarn: Which Package Manager Should You Choose?*. [online] WhiteSource. Available at: <https://www.whitesourcesoftware.com/free-developer-tools/blog/npm-vs-yarn-which-should-you-choose/> [Accessed 24 February 2021].

[38] TypeScript, D., 2021. [online] Typescriptlang.org. Available at: <https://www.typescriptlang.org/docs/handbook/intro.html> [Accessed 24 February 2021].

[39] contributors, M., 2021. *JavaScript | MDN*. [online] Developer.mozilla.org. Available at: <https://developer.mozilla.org/en-US/docs/Web/JavaScript> [Accessed 24 February 2021].

[40] Jansen, P., 2021. *index | TIOBE - The Software Quality Company*. [online] Tiobe.com. Available at: <https://www.tiobe.com/tiobe-index/> [Accessed 24 February 2021].

[41] Team, S., 2010. *The SPARK Ravenscar Profile*. [online] Docs.adacore.com. Available at: <https://docs.adacore.com/sparkdocs-docs/Examiner_Ravenscar.htm> [Accessed 6 March 2021].

[42] Campbell, D., 2019. *The many human errors that brought down the Boeing 737 Max*. [online] The Verge. Available at: <https://www.theverge.com/2019/5/2/18518176/boeing-737-max-crash-problems-human-error-mcas-faa> [Accessed 6 March 2021].

[43] Duffy, R., 2020. *Leaving Cert: About 6,500 students set to receive higher grades after coding error*. [online] TheJournal.ie. Available at: <https://jrnl.ie/5218969> [Accessed 6 March 2021].

[44] SUNDARAM, S., 2019. *Securing the Future of Autonomous Driving with Adacore | NVIDIA Blog*. [online] The Official NVIDIA Blog. Available at: <https://blogs.nvidia.com/blog/2019/02/05/adacore-secure-autonomous-driving/> [Accessed 6 March 2021].

[45] Rommel, C., 2018. *Controlling Costs with Software Language Choice — How Ada Can Help*. [online] AdaCore. Available at: <https://www.adacore.com/papers/controlling-costs-with-ada> [Accessed 6 March 2021].

[46] Meudec, C., 2021. *Midoan Mika: Test data generation for Ada*. [online] Mika User Manual. Available at: <http://www.midoan.com/download/current/user_manual.pdf> [Accessed 6 March 2021].

[47] Code, V., 2021. *Visual Studio Code User and Workspace Settings*. [online] Code.visualstudio.com. Available at: <https://code.visualstudio.com/docs/getstarted/settings> [Accessed 22 April 2021].

[48] Hatton, L., 1999. *The Ariane 5 bug and a few lessons*. [online] Leshatton.org. Available at: <https://www.leshatton.org/Documents/Ariane5_STQE499.pdf> [Accessed 24 April 2021].

[49] Feldman, M., 2014. Who's Using Ada?. [online] Www2.seas.gwu.edu. Available at: https://www2.seas.gwu.edu/~mfeldman/ada-project-summary.html [Accessed 10 January 2021].

[50] Overflow, S., 2019. Stack Overflow Developer Survey 2019. [online] Stack Overflow. Available at: <https://insights.stackoverflow.com/survey/2019#development-environments-and-tools> [Accessed 26 April 2021].

[51] Bond, W., 2011. How to Create a Sublime Text 2 Plugin. [online] Code Envato Tuts+. Available at: <https://code.tutsplus.com/tutorials/how-to-create-a-sublime-text-2-plugin--net-22685> [Accessed 26 April 2021].

[52] Npp-user-manual.org. n.d. Extend functionality with plugins | Notepad++ User Manual. [online] Available at: <https://npp-user-manual.org/docs/plugins/> [Accessed 26 April 2021].

[53] Marketplace.visualstudio.com. 2021. All categories Extensions for Visual Studio Code in Marketplace. [online] Available at:

<https://marketplace.visualstudio.com/search?target=VSCode&category=All%20categories&sortBy=Installs> [Accessed 26 April 2021].

[54] Code, V., 2021. *Your First Extension*. [online] Code.visualstudio.com. Available at: <https://code.visualstudio.com/api/get-started/your-first-extension> [Accessed 28 April 2021].

[55] Code, V., 2021. *Extension Guidelines*. [online] Code.visualstudio.com. Available at: <https://code.visualstudio.com/api/references/extension-guidelines> [Accessed 28 April 2021].