DDC: a declarative debugger for C++
DDC: un depurador declarativo para C++

Trabajo de Fin de Máster
Curso 2021–2022

Autor
Roland Coeurjoly Lechuga

Director
Adrián Riesco Rodríguez

Máster en Métodos Formales en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

DDC: a declarative debugger for C++
DDC: un depurador declarativo para C++

Trabajo de Fin de Máster en Métodos Formales en Ingeniería Informática

Autor
Roland Coeurjoly Lechuga

Director
Adrián Riesco Rodríguez

Convocatoria: *Junio-Julio 2022*
Calificación: 9

Máster en Métodos Formales en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

9 de julio de 2022

Dedicatoria

A Chun, por el apoyo.

Acknowledgments

I would like to thank the Coq community, both in the Zulip chat and in Stack Overflow, for its invaluable help. The more I interact with this community, the more I want to be involved in it. Special thanks to Ana de Almeida Borges, Pierre Castéran, Emilio Jesús Gallego Arias, Paolo Giarrusso, Gaëtan Gilbert, Kenji Maillard, Karl Palmskog, Bas Spitters and Li-yao Xia. Thanks also to Adrián, my Master's thesis director, for his patience and wisdom all along.

Resumen

DDC: un depurador declarativo para C++

Presentamos un depurador declarativo para C++, llamado DDC. Un depurador declarativo recibe como argumento de entrada una computación incorrecta, construye un árbol de depuración basado en la ejecución del programa y, después de preguntar a un oráculo (tipicamente el usuario), indica el fragmento de código causante del fallo. Presentamos las principales características del depurador, tales como tres estrategias de navegación, el uso de casos de prueba como oráculo, capacidad de depurar programas que no terminan y una transformación de árbol.

Palabras clave

depuración declarativa, C/C++, GDB, verificación formal, Coq

Abstract

DDC: a declarative debugger for C++

A declarative debugger for C++ is presented, called DDC. A declarative debugger receives as input an incorrect computation, builds a debugging tree based on the execution of the program and, after asking questions to an oracle (typically the user), points out a fragment of code that is the cause of the failure. We present the debugger's main features, such as three different navigation strategies, using test cases as oracles, support for non-terminating programs and a tree transformation.

Keywords

declarative debugging, C/C++, GDB, formal verification, Coq

Contents

List of figures

List of tables

Listings

Chapter 1

Introduction

"If debugging is the process of removing software bugs, then programming must be the process of putting them in."
— Edsger Dijkstra

In this chapter, we present the motivation of the project, its goals, contributions, and the structure of the rest of the document.

1.1. Motivation

C++ is a programming language that dates back to 1979 (13), and one of its goals is to maintain backward compatibility with C (14). As such, C++ has inherited the defects of C, such as lack of memory safety (3). Despite these shortcomings, C++ is still one of the most used programming languages (9), including being used in such important areas such as:

- Compilers: GCC (36), LLVM (22).
- Databases: MySQL (37), MongoDB (31).
- Theorem provers: Z3 (11), Lean (26).
- Debuggers: GDB (25), rr (28).
- Machine learning frameworks: TensorFlow (2), PyTorch (30).
- Digital currency and smart contracts technology: Bitcoin (27), Solidity (6).

Furthermore, C++ is actively developed, with the last release being C++20. This, together with the upward trending usage statistics (9), tends to indicate that the need to develop, test, and debug C++ programs is going to continue into the future.

This need to debug C++ programs is what lead to this Master's thesis. Debugging is one of the most expensive part of developing software. It is estimated that between 35 and 50 percent of development effort is spent on validation and debugging tasks (29).

There are many debugging approaches, ranging from the fully manual to the almost fully automatic, like delta debugging or program slicing (38). In this Master's thesis we focus on declarative debugging (33), also called algorithmic debugging, which is a semi automatic approach in which the debugger builds a debugging tree of the program and

guides the user through it by asking questions about the correctness of sub-computations until a buggy function is found. We expand the definition of declarative debugging in Chapter 2.

Several declarative debuggers have been developed, including for object-oriented languages such as Java (18), but to the best of our knowledge, there is no declarative debugger for C++. We review the State of the Art of declarative debugging and its implementations in Chapter 3.

1.2. Goals

The goal of this Master's thesis is to develop a declarative debugger for the C++ language. This debugger should have the following features:

1. Integrated in workflow: it has to be integrated in the existing debugging workflow of the developer.
2. No changes to the program being debugged: the user has to be able to debug a program with no or few changes the program and its compilation (at most setting some compilation flags).
3. Tree transformations: it has to perform tree transformations to reduce the size of the debugging tree.
4. Test cases as oracles: it has to use test cases to reduce the number of questions posed to the user.

1.3. Main contributions

The main contribution of this Master's thesis is the development of a declarative debugger for C++, called DDC. The most notable characteristics of DDC are the following:

1. Support for several programming languages, by means of using GDB.
2. Non terminating programs can be debugged.
3. Test cases can be used as oracles to reduce the tree size.
4. Three navigation strategies have been developed.
5. One tree transformation has been developed.
6. It can be easily extended with more strategies and tree transformations.

The source code of the project is available at `https://github.com/RCoeurjoly/DDC` and its license is AGPL-3.0 License.

1.4. Structure of the document

The rest of the thesis is organized as follows:

- In Chapter 2 we introduce declarative debugging, C++, Coq and the different tools needed to implement DDC.

- In Chapter 3 we review the state of the art in declarative debugging and compare DDC's features to other declarative debuggers.
- Chapter 4 contains the definitions and proofs that form the theoretical basis of DDC.
- Chapter 5 describes the architecture of the debugger, presents the most important usage scenarios of the tool and lists its commands.
- Chapter 6 reviews the verification and benchmarks done for DDC.
- Chapter 7 discusses the achievements and the future lines of work that can be followed.

Chapter 2

Preliminaries

In this chapter we will introduce declarative debugging, the C++ programming language, Coq, and the technologies used to develop DDC.

2.1. Declarative debugging

Declarative debugging (33), also called algorithmic debugging, is a debugging technique that consists in (i) taking a program execution which the user deems incorrect, (ii) building an debugging tree (DT) abstracting the execution of this program, and (iii) asking questions about the correctness of computations to an oracle (usually the user, but other sources such as test cases can be used) until a certain computation is narrowed down as buggy. A debugging tree is an abstraction of the execution tree of a program. The abstraction may consist in omitting certain function or method executions or transforming it for improving how it can be traversed, which is called *navigation*.

In the following subsections we will discuss the most important stages in a declarative debugging session, namely:

- Building the debugging tree.
- Transforming the debugging tree.
- Navigating the debugging tree.

2.1.1. Building the debugging tree

Building the debugging tree is the central, critical stage of the debugging process.

To build the DT, the declarative debugger (DD) must have access to the following information:

- When a new function or method begins its execution, taking into account the state of the variables at that point.
- When a new function or method returns a value, and the state of variables at that point.
- Which function or method is the caller and which one is the callee.

With this information, the debugger can build a node, and using these nodes it can build the DT. For formal definitions, please see Chapter 4.

2.1.2. Transforming the debugging tree

The number of questions asked to the user depends on a combination of:

- the size of the tree (number of nodes, width and depth of the tree) and
- the navigation strategy.

Although reducing the size of the debugging tree does not always lead to a reduction in the number of questions (21), most DDs provide certain functionality to reduce the DT size.

2.1.2.1. Trusting or suspecting computations

Most DDs build the DT assuming all functions or methods are suspicious of being wrong. However, this is hardly ever the case. In the case of debugging a C++ program, one option to reduce the size of the DT would be to trust all functions and methods provided by the language implementation, which are located under the `std` namespace, or trusting everything provided by a reputable third party, like Boost (32). This process happens before starting to build the DT, and therefore it reduces the building time and memory footprint. Also, it requires manual input from the user for selecting those computations that can be trusted.

Another perspective is that when debugging a program, it is usually the case that the user believes the problem lays in the parts she developed, not in the external functions she calls. By suspecting only the functions and methods developed by the user (and implicitly trusting all other functions), the debugger can ignore all other functions, like those provided by the standard or by external libraries. This is the approach used by DDC. This process also happens before starting to build the DT: it requires manual input from the user in selecting those computations that should be suspected.

2.1.2.2. Using test cases as oracles

Another method to reduce the size of the DT is to have an external oracle mark computations as correct. This oracle might be the set of tests that use some of the suspect functions or methods. By collecting these test executions, which we assume are correct, we can discard the same executions if they are present in the DT of the program being debugged.

This functionality, provided by DDC, requires manual input from the user for selecting the test cases to be executed.

2.1.2.3. Tree transformations

Tree transformations happen once the DT has been built, so the improvement happens in terms of navigation time, not in building time or memory footprint.

There are several transformations proposed in the literature, like loop expansion and tree compression (21). These processes are fully automatic, so they do not need any input from the user. A tree transformation can reduce or increase the number of nodes in the tree. The latter is done to make the tree more balanced and therefore increasing the efficiency of the navigation strategies.

2.1.3. Navigating the debugging tree

Navigating the debugging tree is the phase when the debugger asks an external oracle questions about the correctness of certain nodes, with the purpose of finding a buggy one. The goal of this phase is to minimize the number of questions, to make the process as fast as possible.

2.1.4. Correctness answers

When asked about the correctness of a certain node, the user usually has the following possible answers at her disposal, with the corresponding effect on the debugging tree:

- Yes: the node has produced a correct set of outputs with the given inputs.

  The subtree rooted in this node can be eliminated from the debugging tree. Also, all nodes that have the same inputs and outputs can be removed from the debugging tree.

- No: the node has not produced a correct set of outputs with the given inputs.

  The subtree rooted in this node becomes the debugging tree.

- Trusted: this function or method is deemed correct for all inputs.

  All nodes representing an execution of this function or method can be removed from the debugging tree, no matter the inputs received.

- I don't know: the user cannot evaluate the correctness of this sub computation.

  The node can be removed from the debugging tree or the navigation strategy can choose another node, leaving this node in the tree. If the node is left in the tree, the debugging algorithm would be incomplete. To alleviate this issue, a debugger could ask again about these nodes at the end of the debugging session.

2.2. C++ programming language

The C++ programming language is a general-purpose, statically typed, compiled language (13). C++ supports several programming paradigms, including:

- Object-oriented programming (OOP).
- Generic programming.
- Functional programming.

In generic programming in C++, types are a parameter for classes or functions. Once a template is instantiated with a specific type, the compiler creates a class or function for that type. The instantiated class or function is what is executed at runtime. Therefore, supporting OOP implies support for generic programming.

The requirements for a DD that supports OOP are greater than for functional programming because in functional programming a function only:

- Takes arguments.
- Returns a value.

On the other hand, in OOP, apart from the arguments and return value, we have to monitor:

- Object state on entry.
- Object state when returning.
- Global variables on entry.
- Global variables when returning.

Therefore, support for OOP implies support for functional programming.

Being statically typed gives the advantage of providing type information for functions arguments, return values, variables, and classes. This is important for the debugger to be able to display information adequately to the user, making the debugging experience better.

The process of compilation usually removes a lot of information about the source code, with the purpose of creating a faster and smaller binary file. However, source code information is necessary to make the debugging session user friendly. This information includes class, function and variable names, among others. To make this information available for the debugger, the C++ program must be compiled with the debug information flag set to true.

2.3. The Coq Proof Assistant

Coq (5) is an interactive theorem prover. Coq implements a dependently-typed functional programming language. It also provides a series of tactics to transform incomplete proof goals into a complete proof (17). User defined lemmas can be used to solve more complex lemmas or theorems, which we use extensively.

Of special interest for this Master's thesis is Coq's Program tactic (35), which can be used to develop certified programs, that is, programs whose implementation and specification are intertwined.

This is the approach taken to verify our implementation of the generic debugging algorithm, whose details can be found in Chapter 6.

Once a verified program has been implemented in Coq, it can be extracted to an executable programming language such as OCaml, Haskell or Scheme (23). Since `GDB` only provides a Python API, we would need to call one of the extraction languages from Python, using a library such as `pythonlib` (10).

2.4. Technologies used

There are two main dependencies in this project:

- GDB, used as the general framework on DDC is built.
- rr, used for its reverse execution features.

2.4.1. GDB: The GNU Project Debugger

GDB (25) is a general purpose debugger for C and C++, with partial support for other languages. GDB allows the user to execute a program, stop the execution at any point,

inspect the state of the program, and change any part of the state of the execution. We make extensive use of its Python API, especially of the following classes:

Values. Values from the program being debugged represent the content of the program variables.

Types. The type of a certain value is needed in two occasions: when displaying the value and to check whether arguments are passed as reference or as pointers.

Frames. A frame contains the data relevant to a function call. Frames are organized hierarchically in a call stack, the frame $n + 1$ being the caller of frame n, the callee.

Symbols. A symbol is a variable or a function. We use symbols to determine the variable names. Symbols also provide the scope they belong to. This is needed to check if a variable is a global one or an argument to a function.

Breakpoints. A breakpoint is a bookmark set in a certain location of a program, such as a function or specific line, which tells the debugger to stop the execution. We make use of breakpoints for:

- Identifying functions/methods we want to add to the tree.
- Identifying functions/methods we want to build correct nodes from.
- Setting the final point, where the building of the debugging tree must stop.

A breakpoint has an attribute named *commands*, which is executed when it is reached, and if the method stop returns false. We make use of these commands to store the information of the current frame.

Final breakpoints. A final breakpoint is a breakpoint that is triggered when a function or method returns. Final breakpoints are used to gather the following information:

- The return value.
- The state of the object.
- The global variables.
- The arguments passed as reference or pointer.

2.4.2. rr: Record and Replay Framework

rr (28) enhances GDB, providing support for record and replay functionality, as well as for reverse execution. DDC uses rr for its reverse execution capabilities. This is needed because once a final breakpoint is reached, we need to step back to get:

- The object state when returning.
- The arguments when returning (if passed as reference or pointers).
- The global variables when returning.

Although GDB itself provides some support for reverse execution, rr is more complete in this respect.

Chapter 3

State of the Art

In this chapter we will summarize the state of the art in declarative debugging. The identification of issues and solutions comes from the 2017 survey (8). Latter research is included when appropriate. We also mention where DDC contributes to the state of the art: a new tree transformation called simplified tree compression and support for debugging non-terminating programs.

3.1. Issues of algorithmic debugging

The following issues were identified in the 2017 survey (8). They are divided into three categories:

- The Scalability Problem.
- User Experience and Effectiveness.
- Completeness.

3.1.1. The Scalability Problem

Scalability has been the main issue blocking adoption of declarative debugging in industry.

- Issue-1: Time Needed to Generate the Debugging Tree

  The theoretical minimum amount of time needed to generate the DT is the time needed to execute the program. In practice, some instrumentation is needed to monitor the execution.

- Issue-2: Memory Needed to Store the Debugging Tree

  A large debugging tree size may be gigabytes, which does not fit in main memory. Also displaying such a large tree may pose problems.

3.1.2. User Experience and Effectiveness

- Issue-3: Amount and Difficulty of the Questions

  The optimal navigation strategy, Optimal Divide and Query (19), has a query complexity of $O(N \cdot logN)$, which is unpractical for large executions.

- Issue-4: Rigidity and Loss of Control During the Navigation Phase
- Issue-5: Integration with Other Debugging Tools If a declarative debugger is not integrated into the other debugging tools used by the user, it creates the burden of context switching and learning a new environment.
- Issue-6: Bug Granularity

  Usually, the bug granularity of declarative debuggers is a function or method. Ideally it should be smaller, such as loop or even line.

3.1.3. Completeness

The following are some features that pose problems to declarative debuggers.

- Issue-7: Termination

  Non terminating programs usually cannot be debugged by DDs, apart from DDC.
- Issue-8: Concurrency

  The standard debugging tree is not prepared to store concurrent computations.
- Issue-9: I/O

  Programs that make use of the file system or a database cannot be handled by declarative debuggers.

3.2. Solutions of algorithmic debuggers

1. Scalability [deals with Issue-1 and Issue-2]

   The following solutions to alleviate Issue-1 and Issue-2 have been proposed:

   - Generate the debugging tree on demand.
   - Store the debugging tree in secondary memory, like in the file system or on a database.
   - Start the debugging session with an incomplete debugging tree.
2. Trusting Modules, Routines and Arguments [deals with Issue-1, Issue-2 and Issue-3]

   By trusting third party code, the debugging tree is reduced and the debugger has the ability to focus its questions on the user code.
3. Multiple Navigation Strategies [deals with Issue-3]

   Although an optimal navigation strategy with respect to the number of questions has been developed (19), it is advisable for declarative debuggers to implement more strategies. Also, the navigation strategy can change to adapt to the debugging tree.
4. Accepted Answers [deals with Issue-3]

   The oracle should have the following options available when answering a correctness question posed by the debugger:

   - Yes: the node should be removed from the tree.
   - No: the subtree rooted in this node should be the new debugging tree.

- Inadmissible: this computation should have not happened, therefore the problem lays in the caller of this node.
- I don't know: this question is too difficult, therefore continue asking question about other nodes.
- Trusted: this function or method is correct, therefore all other nodes that have this function signature should be pruned from the debugging tree.

5. Tracing Subexpressions [deals with Issue-3 and Issue-4]

   If the user knows which part of the node is incorrect, the debugger should provide the option to indicate it.

6. Debugging Tree Transformations [deals with Issue-3 and Issue-6]

   - Loop expansion: consists in a source code transformation from iterative loops to recursive functions to achieve a deep debugging tree, which is easier to debug than a wide debugging tree.
   - Tree compression: removes redundant nodes from the debugging tree preserving completeness.
   - Simplified tree compression: this transformation is a novel contribution of this Master's thesis. It has the same objectives as tree compression but can only be applied to nodes with one child.
   - Tree balancing: tries to transform the debugging tree into a binomial tree to make the navigation logarithmic in time and groups nodes so a single answer deals with several function executions at the same time.

7. Memoization [deals with Issue-3 and Issue-5]

   By memoizing answers, the debugger can avoid asking the oracle again when a node that has already been answered appears in the navigation phase.

8. Debugging Tree Exploration [deals with Issue-4]

   Instead of being limited to view the current node being questioned, the user should have the freedom to explore the debugging tree, selecting which subtree seems most suspicious of being buggy.

9. Undo Capabilities [deals with Issue-4]

   If the user makes a mistake when answering a question, it should be possible to undo the answer so that the debugger asks again.

10. Communication with an IDE [deals with Issue-5 and Issue-6]

    By communicating with an IDE, the declarative debugger can leverage the usability features and tools present in the IDE. Also, the learning curve gets reduced.

11. Different Levels of Errors [deals with Issue-6, worsens Issue-3]

    Some debuggers give the user the option of debugging loop or conditional branches once the buggy function has been identified.

12. Program Slicing [deals with Issue-6]

    By performing program slicing, the debugger extracts only the portion of code that executed in the session, discarding the rest.

13. Record and replay [deals with Issue-7, worsens Issue-1]

    With this technique, instead of dealing with a program execution, the debugger deals with a recording of a program execution. This allows the user to send the termination signal to the non-terminating program being recorded, therefore transforming a non-terminating program into a terminating one. The downside of this technique is that it requires the execution of the buggy program twice, once for recording purposes and another for building the debugging tree, therefore doubling the time needed to build the debugging tree (Issue-1).

    This feature is a novel contribution of this Master's thesis. A demonstration can be found in Chapter 5.

14. Unified debugging and testing [deals with Issue-3 and Issue-3]

    This technique was proposed in (7). Its purpose is to "integrate debugging and testing within a single unified framework where each phase generates useful information for the other and the outcomes of each phase are reused". This reduces the number of questions asked to the user (Issue-3), since it answers some with an external oracle, namely a test suite.

15. Generalized algorithmic debugging

    In the generalization proposed in (20), a node does not correspond to a function call, but can represent the execution of any amount of code. Also, it contains only code that has been executed in the execution passed to the declarative debugger. This improves the granularity of detected bugs (Issue-6). However, this generalization has not been incorporated into a declarative debugger, so the benefits remain unrealized.

3.2.1. Conclusions

Table 3.1 is an updated version of Table 1: Relation Between Issues and Solutions in (8). We have added in green those new features proposed after the publication of the survey. Like in (8), the symbol ✓ indicates that the feature solves or at least alleviates the issue, while ✗ indicates that the feature worsens it.

DDC supports debugging non terminating programs by means of record and replay debugging. This feature is not found in other declarative debuggers. However, by using record and replay, the debugging time is increased. This time penalty is explored more in depth in Chapter 6.

3.3. Current state of the art algorithmic debuggers

Table 3.2 is an updated version of Table 2. Comparison of Algorithmic Debuggers in (8). Apart from adding DDC to the table, we have included the debuggers that have been updated since (8), marking in green those features that have been updated. We see that DDC has, in general, the same features that other declarative debuggers. A notable gap is the inability to undo answers.

Table 3.1: Relation Between Issues and Solutions

Feature \ Issue	1	2	3	4	5	6	7	8	9
Navigation strategy			✓						
Answers			✓						
Trac. subexp.			✓	✓					
DT transf.			✓			✓			
Memoization			✓		✓				
DT exploration				✓					
Undo				✓					
Trusting	✓	✓	✓						
Scalability	✓	✓							
Levels of errors			✗			✓			
Program slicing						✓			
IDE					✓	✓			
Record and Replay	✗						✓		
Unified debugging and testing			✓		✓				
Generalized algorithmic debugging						✓			

Table 3.2: Comparison of Algorithmic Debuggers

Feature \ Debugger	DDC	Hat-Delta	EDD	Mercury Debugger	DES
Target language	C++	Haskell 98	Erlang	Mercury	Datalog SQL
Imp. language	Python	Haskell	Erlang / Java	Mercury / C	Prolog
Strategies	TD DQ HF	TD HD	TD DQ	TD DQ BW SD	TD DQ
DataBase / Memoization	NO/NO	NO/YES	YES/YES	YES/YES	NO/YES
Debugging tree	Main memory	File	Main memory	Main memory on demand	External database
Accepted answers	YES NO DK TR	YES NO DK	YES NO DK IN TR	YES NO DK IN TR	YES NO TR DK
Tracing subexpressions?	NO	NO	YES	YES	NO
Granularity	NO	YES	YES	NO	YES
DT exploration	YES	YES	YES	YES	NO
Transformations	STC	TC	-	-	-
Early start	NO	NO	NO	YES	NO
Undo	NO	YES	YES	YES	YES
Trusting	Fun	Mod	Fun	Mod Fun	Mod
External oracle	YES	NO	YES	YES	YES
GUI	NO	YES	YES	NO	YES
Version	0.0.1 (May 2022)	Hat 2.9.4 (November 2017)	July 2020	Mercury 22.01.1 (April 2022)	DES 6.7 (September 2021)

Chapter 4

Definitions

In this chapter we will provide definitions for the key ideas that establish the theoretical basis of the tool.

Definition 1 (Node)**.** *A node n is a function/method execution, denoted* $f(I) \rightarrow O$*, where:*

- f *is the function/method executed.*
- I *is a 3-tuple of inputs to* f*,* $I = \{I_o, I_a, I_g\}$*, where:*
  - I_o *is the object state when* f *was called if* f *is a method,* $\perp$ *otherwise.*
  - I_a *is an n-tuple of input arguments when* f *was called.*
  - I_g *is a set of global variables when* f *was called.*
- O *is a 4-tuple of outputs of* f*,* $O = \{O_o, O_a, O_g, O_r\}$*, where:*
  - O_o *is the object state when* f *returns if* f *is a method,* $\perp$ *otherwise.*
  - O_a *is the m-tuple of output arguments when* f *returns if there were passed as reference or pointer. Note that* $m <= n$*, where* $m = \mathbf{card}(O_a)$ *and* $n = \mathbf{card}(I_a)$
  - O_g *is the set of global variables when it returned.*
  - O_r *is the return value.*

Example 1. *Figure 4.1 is the node extracted from the execution of the method* `Car::move` *with arguments 10 and 5, called from* `main` *in Listing 4.1. This listing consists of a* `Car` *class, which only has one method,* `Car::move` *and the* `main` *function, from which we instantiate a* `Car` *object and move it. Note that it has no children nodes. It is composed of:*

- f *is* `Car::move(int const&, int const&)`*.*
- I_o *is the branch called* `object state on entry`*. It has two variables,* `x` *and* `y`*, both with a value of 0, since those are the values we assign in the constructor.*
- I_a *is the branch called* `arguments on entry`*. It displays the names of the arguments,* `xDelta` *and* `yDelta`*, and their value, 10 and 5, respectively. Since they are passed as reference, we also display their addresses in memory,* `@0x7ffe80e3cd08` *and* `@0x7ffe80e3cd0c`*.*

- I_g *is the branch called* `global variables on entry`. *In this branch we list all global variables upon entry on the method. In this case there is only one,* `number_of_cars`, *which got set to 1 when the constructor was called.*
- O_o *is the branch called* `object state when returning`. *In this branch we display the object state when the* `return` *statement is executed. If we did not have an explicit* `return` *statement, the content of this branch would be the object state at the curly bracket (}) that terminates the function or method.*
- O_a *is the branch called* `arguments when returning`. *This branch is present in the node because the arguments are passed by reference. Passing by pointer also would have created this branch. Passing by value would not have created this branch.*
- O_g *is the branch called* `global variables when returning`. *Same as* I_g, *but when returning. We display this branch even if the global variables are the same as in entry, as is the case in this example.*
- O_r *is the branch called* `return value`. *In C++ return values are not named, so we only have to display the value, in this case* `true`.

Listing 4.1: Car class

```
1  int number_of_cars = 0;
2
3  class Car {
4  public:
5    Car() {
6      x = 0;
7      y = 0;
8      number_of_cars++;
9    };
10   bool move(const int& xDelta, const int& yDelta) {
11     x += xDelta;
12     y += yDelta;
13     return true;
14   };
15 private:
16   int x;
17   int y;
18 };
19
20 int main() {
21   Car my_car;
22   my_car.move(10, 5);
23 }
```

Definition 2 (Edge)**.** *An edge is a hierarchical relationship between two nodes, a parent node and a child node, in which the child node represents a function or method call made from the body of the function corresponding to the parent node.*

Remark 1. *We can make the following observations:*

- *A node can have 0 or more children nodes.*
- *A node that has no children nodes is a leaf node.*
- *A node can have 0 or 1 parent nodes.*

Figure 4.1: Node representing the execution of the Car::move method

```
Car::move(int const&, int const&)
├── object state on entry = {
│     x = 0,
│     y = 0
│   }
├── arguments on entry
│   ├── xDelta = @0x7ffe80e3cd08: 10
│   └── yDelta = @0x7ffe80e3cd0c: 5
├── global variables on entry
│   └── number_of_cars = 1
├── object state when returning = {
│     x = 10,
│     y = 5
│   }
├── arguments when returning
│   ├── xDelta = @0x7ffe80e3cd08: 10
│   └── yDelta = @0x7ffe80e3cd0c: 5
├── global variables when returning
│   └── number_of_cars = 1
└── return value
    └── true
```

- *A node that has no parent node is the root node of the tree.*
- *Nodes that share the same parent node are siblings.*

We here expand the definition of Marked Execution Tree (MET in (19)) to Weighted Marked Debugging Tree (WMDT).

Definition 3 (Weighted marked debugging tree)**.** *A weighted marked debugging tree is a tree, denoted $T = (N, E, W, M)$, where N is a set of nodes, $E \subseteq N \times N$ is the set of edges, $M : N \to V$ is a total function that assigns to all the nodes in N a value in the domain $D = \{Wrong, Undefined\}$, and W is a total function that assigns to all the nodes in N a value which is the weight of the sub-tree rooted at node n in N, w_n, which is defined recursively as its number of descendants including itself (i.e.,$1 + \sum w_{n'} \mid n \to n') \in E$).*

Example 2. *Figure 4.2 is an example tree representing the WMDT of partition (see Listing 5.2 for the corresponding code) called with $my\_vector = \{10, 7, 8, 9, 1, 5\}$, $low = 0$ and $high = 5$. In this case we have the following three nodes:*

- *Function partition called with arguments $my\_vector = \{10, 7, 8, 9, 1, 5\}$, $low = 0$ and $high = 5$ (n_0).*
- *Function swap called with arguments pointer a to 10 and pointer b to 1 (n_1).*
- *Function swap called with arguments pointer a to 7 and pointer b to 5 (n_2).*

We can make the following remarks:

- n_0 *is the root node of the tree, since there is no edge leading to a parent node.*
- n_1 *and* n_2 *are siblings, since they share the same parent,* n_0*.*
- n_1 *and* n_2 *are leaf nodes, since they do not have any edges leading to children. Note the lack of* `children` *branch in both nodes.*
- *Leaf nodes (*n_1 *and* n_2*) have weight 1 each, denoted* w_{n_1} *and* w_{n_0} *respectively.*
- *The root node (*n_0*) has a weight* $w_{n_0} = w_{n_1} + w_{n_2} + 1 = 1 + 1 + 1 = 3$*.*
- *All three nodes are undefined, that is,* $M(n_0) = M(n_1) = M(n_2) = Undefined$*. This is denoted by the* `correctness` *branch having the value* `I don't know`*.*

Definition 4 (Intended interpretation)**.** *The intended interpretation of a function or method* f *given an input set* I*, denoted* $II_f(I) \rightarrow O'$*, is a mapping from* I *to* O'*, where* O' *is the expected result of executing* $f(I)$*.*

Definition 5 (Wrong node)**.** *A wrong node* $n_{wrong} = f(I) \rightarrow O$ *is a node that is not equal to its intended interpretation,* $II_f(I) \rightarrow O'$*.*

Definition 6 (Correct node)**.** *A correct node is a node that is equal to its intended interpretation.*

Definition 7 (Buggy node)**.** *A buggy node is a wrong node in which all its children are correct nodes.*

Example 3. *Figure 4.3 is the intended interpretation of swap* $II_{swap}(I) \rightarrow O'$*,* I *being two pointers pointing to 10 and 1, respectively. On the other hand, Figure 4.4 is the node* $n_{swap}(I) \rightarrow O$ *of the execution of swap given the same arguments. Since* $O' \neq O$ *(note that in Figure 4.4 swap returns* $a = 10$ *and* $b = 1$*, whereas in Figure 4.3* `swap` *returns* $a = 1$ *and* $b = 10$*), we can say that* n_{swap} *is a buggy node, and therefore there is a bug in the function swap, as we will see later.*

Definition 8 (Detected errors)**.** *When our debugger returns a buggy node* $n_{buggy} = f(I) \rightarrow O$*, an error in the definition of* f *has been detected.*

Definition 9 (Navigation strategy)**.** *A navigation strategy is a traversal of the WMDT in which the debugger asks an oracle to answer questions about the correctness of certain nodes, the possible answers being correct or wrong.*

Definition 10 (Top-down strategy)**.** *The top-down strategy is a navigation strategy in which when a node is deemed wrong, one of its children is asked; if it is deemed correct, one of its siblings is asked (34).*

Theorem 1 (Completeness)**.** Let $T = (N, E, W, M)$ be a weighted marked debugging tree, S the top-down strategy, and the root node of T a wrong node. If S receives T as input, it always terminates, producing a node $n' \in N$ as output.

Proof. See (24). □

Theorem 2 (Soundness)**.** Let:

Figure 4.2: WMDT of partition (see 5.2 for the corresponding code) called with $my\_vector = \{10, 7, 8, 9, 1, 5\}$, $low = 0$ and $high = 5$

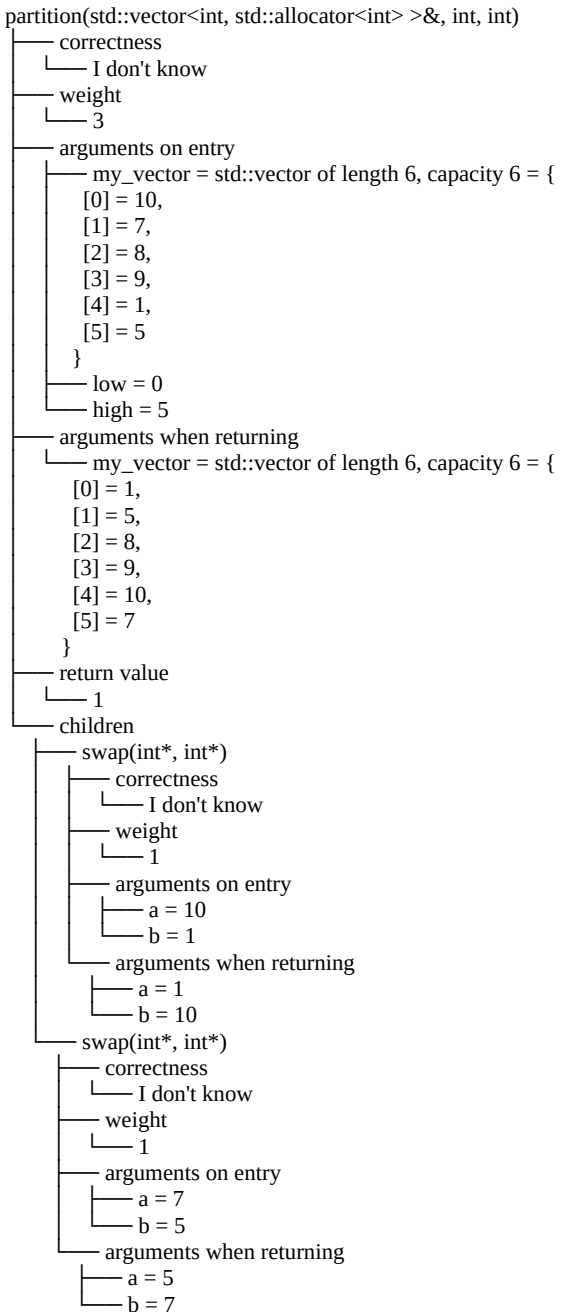

Figure 4.3: Intended interpretation of swap function given pointer a to 10 and pointer b to 1

```
swap(int*, int*)
├── arguments on entry
│   ├── a = 10
│   └── b = 1
└── arguments when returning
    ├── a = 1
    └── b = 10
```

Figure 4.4: Buggy node representing the execution of swap

```
swap(int*, int*)
├── correctness
│   └── no
├── weight
│   └── 1
├── arguments on entry
│   ├── a = 10
│   └── b = 1
└── arguments when returning
    ├── a = 10
    └── b = 1
```

- $T = (N, E, W, M)$ be a weighted marked debugging tree,
- the root of T a wrong node,
- F a total function such that for every node $n \in N$ there is a function or method f such that $F : n \to f$,
- II a total function such that for every function or method $f \in F$ there is an intended interpretation II_f such that $II : f \to II_f$,
- G a total function such that for every node $n \in N$ there is 3-tuple of inputs I such that $G : n \to I$, and
- S the top down strategy.

If S receives T as input, then:

- S returns a node n',
- $n' \in N$,
- $F(n') = f'$,
- $II(f') = II_{f'}$,
- $G(n') = I$, and

- $II_{f'}(I) \neq f'(I)$.

Proof. See (24). □

Chapter 5

Tool description

DDC consists of approximately 900 lines of mostly statically typed Python, which can be found in `https://github.com/RCoeurjoly/DDC/blob/main/declarative_debugger.py`. It is loaded into rr through the `source` command, which itself can be put inside a `.gdbinit` file.

In this chapter we will explain the architecture of the tool. Then, we will go through three usage scenarios:

- Finding bugs in terminating programs.
- Using test cases as oracles to reduce tree size.
- Finding bugs in non terminating programs.

Lastly, we will list all commands provided by the tool.

5.1. Architecture

In this section we detail and justify the most important design decisions made while developing DDC.

5.1.1. Tree building

To represent the debugging tree, we use the `Node` class. The `Node` class is a recursive data structure, since it has an attribute called `children` which is a list of `Node`s. This list of children nodes are sorted by execution time, that is, a child node in position n has executed before than a child node in position $n + 1$.

The `DebuggingSession` class has attribute of type `Node` to store the debugging tree. It also tracks if the debugging session has begun and if the tree has been built.

5.1.1.1. Adding a node to the tree

We add a node to the debugging tree once we hit a breakpoint. The node is added to the last position of the tree. At that moment we add the 3-tuple of inputs, denoted I in Chapter 4, to the node.

5.1.1.2. Finishing a node

When the finish breakpoint of a certain function it hit, we add the 4-tuple of outputs, denoted O in Chapter 4, to the node.

5.1.1.3. Finishing the tree building process

The tree is considered built if one of the following happens:

- A final point has been hit.
- The program has finished.
- The root node has been completed.

5.1.2. Tree transformations

After having the WMDT built, we apply all tree transformations available. A tree transformation is an algorithm that receives a WMDT an returns another WMDT. DDC implements the simplified tree compression algorithm, which is described next.

5.1.2.1. Simplified tree compression

The simplified tree compression algorithm consists of two steps:

- Compressing the root of the WMDT if it can be compressed.
- Compressing each of the children nodes of the root of the WMDT if they can be compressed.

Given a WMDT, we can compress nodes if:

- The root node has only one child node.
- The root and child nodes function names are the same.

To check if a node can be compressed we use a recursive function, which takes a node as an argument and returns a 2-tuple, whose elements are:

- bool: True if the node can be compressed. False otherwise.
- int: If the node can be compressed, it stands for the number of nodes to compress. Its value is zero otherwise.

This algorithm searches for the largest possible compression.

Given a WMDT and a number of nodes to compress n, the compress node is composed of:

- The 3-tuple of inputs to the root node.
- The 4-tuple of outputs of the node in depth n of the WMDT.
- The name of the root node with an indication that corresponds to n compressed nodes.

5.1.3. General debugging algorithm

The general debugging algorithm takes two arguments:

- The WMDT.
- A navigation strategy.

It returns either a buggy node or $\perp$. It consists of a loop, from which we exit if the WMDT is empty or a buggy node has been found. To determine if a node has been found we have to check that the WMDT consists of an incorrect node without children nodes.

Once inside the loop, first we select the node to be evaluated by means of the strategies, which are explained in Section Strategies. We then ask the user about the node. One of the options available to the user is to change strategy. If this is chosen, we ask the user her preferred strategy and continue the execution of the loop.

If no change in strategy is required, we execute the adequate action corresponding to the correctness answer (see Section User answers to correctness questions) and continue the execution of the loop.

5.1.4. Navigation strategies

A navigation strategy is an algorithm that takes a WMDT and returns a node.

To ease the task of adding more strategies to DDC, we implemented a wrapper to the strategies.

5.1.4.1. Top-down

Top-down is a recursive algorithm, which searches for the first node whose correctness is unknown. It chooses the first child of the root node whose correctness is unknown.

5.1.4.2. Divide and Query (Hirunkitti)

In Divide and Query (Hirunkitti), we select the child node whose weight is the closest to half of the root node weight.

5.1.4.3. Heaviest first

In heaviest first, we select the child node whose weight is the heaviest among all the children nodes.

5.1.5. User answers to correctness questions

When asked about the correctness of a node, the user has the option of answering (i) I don't know, (ii) Yes, (iii) No, and (iv) Trusted. We choose the next node and/or prune the debugging tree depending on the answer

5.1.5.1. I don't know

If the user does not know if the node is correct, we remove it from the WMDT and continue the debugging session.

5.1.5.2. Yes

If the node is deemed correct by the user, we remove it from the WMDT and all other nodes equal to the correct node.

5.1.5.3. No

If the node is deemed incorrect by the user, we continue the navigation phase with this node as the WMDT root.

5.1.5.4. Trusted

If the node is trusted by the user, we remove it from the WMDT and all other nodes which correspond to executions of the same function of the correct node.

5.1.6. Test cases as oracles

DDC implements the functionality of test cases as oracles with two global variables:

- In `CORRECT_NODES` we store a list of all nodes that have finished, that is, whose functions have returned.
- In `PENDING_CORRECT_NODES` we store a list of all nodes that have started but not finished.

To compare the gathered correct nodes with those within the debugging tree, we implement the `ComparableTree` class that derives from `rich.tree`, which is the class used to present the information to the user.

The `Node` class has a member function `get_tree`, which returns a `ComparableTree`. For comparison with elements in the `CORRECT_NODES` list, the node gets stripped of its correctness value, children and weight, comparing only inputs and outputs, denoted I and O respectively in Chapter 4.

Once a node finishes, we search for it in the `CORRECT_NODES` list. If found, we remove it from the debugging tree.

5.2. Usage scenarios

This section describes three usage scenarios with DDC. We will explain which commands should be executed and the expected output.

5.2.1. Finding bugs in terminating programs

The execution of the code in Listing 5.1 results in an unexpected output, $\{10, 5, 1, 7, 8, 9\}$, which is not sorted. It uses the implementations of functions `quickSort`, `partition`, and `swap` defined in Listing 5.2.

Listing 5.1: Code that results in unexpected output

```
1 #include <quicksort.h>
2
3 int main()
4 {
5   std::vector<int> my_vector{ 10, 7, 8, 9, 1, 5 };
```

```
  quickSort(my_vector, 0, my_vector.size()-1);
  std::cout << "Sorted vector: " << std::endl;
  print_vector(my_vector);
  return 0;
}
```

Listing 5.2: quickSort, partition and swap implementations

```
#include <iostream>
#include <vector>

void swap(int* a, int* b)
{
  if (*a != 10)
    std::swap(*a, *b);
}

int partition(std::vector<int> &my_vector, int low, int high)
{
  int pivot = my_vector[high];
  int i = (low - 1);
  for (int j = low; j <= high - 1; j++)
    {
      if (my_vector[j] <= pivot)
        {
          i++;
          if (i != j)
            swap(&my_vector[i], &my_vector[j]);
        }
    }
  if ((i + 1) != high)
    swap(&my_vector[i + 1], &my_vector[high]);
  return (i + 1);
}

void quickSort(std::vector<int> &my_vector, int low, int high)
{
  if (low < high)
    {
      int pi = partition(my_vector, low, high);
      quickSort(my_vector, low, pi - 1);
      quickSort(my_vector, pi + 1, high);
    }
}

void print_vector(const std::vector<int> &my_vector)
{
  for (size_t i = 0; i < my_vector.size(); i++) {
    std::cout << my_vector[i] << ' ';
  }
}
```

To begin a debugging session with the purpose of finding the cause of the error, we first have to record and replay the buggy program with `rr`. This is shown in Listing 5.3.

Listing 5.3: Compiling, recording and replaying quickSort

```
nix build
rr record ./result/bin/quicksort
rr replay
```

Figure 5.1: Setting the suspect functions

```
(rr) suspect-function quickSort(std::vector<int>&, int, int)
Breakpoint 1 at 0x40135e: file quicksort.h, line 44.
(rr) suspect-function partition(std::vector<int>&, int, int)
Breakpoint 2 at 0x401256: file quicksort.h, line 20.
(rr) suspect-function swap(int*, int*)
Breakpoint 3 at 0x401222: file quicksort.h, line 9.
(rr) info breakpoints
Num Type       Disp Enb Address         What
1   breakpoint keep y   0x40135e in quickSort(std::vector<int>&, int, int)
    add-node-to-session quickSort(std::vector<int>&, int, int)
2   breakpoint keep y   0x401256 in partition(std::vector<int>&, int, int)
    add-node-to-session partition(std::vector<int>&, int, int)
3   breakpoint keep y   0x401222 in swap(int*, int*)
    add-node-to-session swap(int*, int*)
```

Figure 5.2: Choosing the navigation strategy

```
Please choose navigation strategy
> Top-down
  Divide and Query (Hirunkitti)
  Heaviest first
```

Once inside the `rr` command prompt, we have to tell DDC in which functions we suspect the bug to reside. To do this, we use the `suspect-function` command described in command suspect-function. In this case, we suspect the functions we defined, that is, `quickSort`, `partition`, and `swap`. We execute the commands shown in Figure 5.1 in `rr`. Note that the results of the command `info breakpoints` have been simplified for display purposes. To build the debugging tree, we have to issue the `start-declarative-debugging-session`. Once the debugging tree has been built, we can choose the navigation strategy. Then, as shown in Figure 5.3, the debugger asks us about the correctness of the root node of the debugging tree. This question does not have the option to change the navigation strategy, as opposed to all other correctness questions, where the options are shown in Figure 5.4.

The debugging session ends when the buggy node is found, as shown in Figure 5.5. Although we only have the granularity to say which function is buggy, we also display with what inputs the buggy function misbehaves, so that the user can create a test case that reproduces the failure.

5.2.2. Using test cases as oracles to reduce tree size

Now, using the same buggy program as in the previous section, we are going to gather correct nodes by executing test cases. To do this, we first have to develop a C++ executable which uses the functions with suspect to be buggy, which we display in Listing 5.4.

In this listing, we test ten vectors composed of random numbers from 0 to 9. When

Figure 5.3: Correctness question of root node

```
You have selected Top-down!
Is the following node correct?
quickSort(std::vector<int, std::allocator<int> >&, int, int)
├── correctness
│   └── I don't know
├── weight
│   └── 14
├── arguments on entry
│   ├── my_vector = std::vector of length 6, capacity 6 = {
│   │     [0] = 10,
│   │     [1] = 7,
│   │     [2] = 8,
│   │     [3] = 9,
│   │     [4] = 1,
│   │     [5] = 5
│   │   }
│   ├── low = 0
│   └── high = 5
└── arguments when returning
    └── my_vector = std::vector of length 6, capacity 6 = {
          [0] = 10,
          [1] = 5,
          [2] = 1,
          [3] = 7,
          [4] = 8,
          [5] = 9
        }
> Yes
  No
  I don't know
  Trusted
```

Figure 5.4: Options to correctness question of non root node

```
> Yes
  No
  I don't know
  Trusted
  Change strategy
```

Figure 5.5: Buggy node found

```
Buggy node found
swap(int*, int*)
├── arguments on entry
│   ├── a = 10
│   └── b = 1
└── arguments when returning
    ├── a = 10
    └── b = 1
```

executing these test cases, the return code is 0, so we know that our `quickSort` implementation at least works for some vectors.

Listing 5.4: Test cases for quickSort

```
#include <quicksort.h>
#include <algorithm>
#include <cassert>

int main()
{
  std::vector<int> my_vector{ 0, 0, 0, 0, 0, 0 };
  const int n = 6;
  for (int i = 0; i <= 9; i++) {
    my_vector[0] = rand() % 9 + 1;
    my_vector[1] = rand() % 9 + 1;
    my_vector[2] = rand() % 9 + 1;
    my_vector[3] = rand() % 9 + 1;
    my_vector[4] = rand() % 9 + 1;
    my_vector[5] = rand() % 9 + 1;
    quickSort(my_vector, 0, n-1, "");
    if (!std::is_sorted(my_vector.begin(), my_vector.end()))
      return 1;
  }
  return 0;
}
```

Like with a buggy program, we first have to record the execution with `rr`. This shell will be named rr_{server}, to differentiate it from the debugging shell. This is shown in Listing 5.5.

Listing 5.5: Compiling, recording and replaying quickSort test cases

```
nix build .#test_quicksort
rr record ./result/tests/test_quicksort
rr replay
```

Once inside the `rr` command prompt, we have to tell DDC from which functions should the correct nodes be gathered. This can be done with the `save-correct-function` command described in the command save-returning-correct-node. In our example, we want to save correct nodes gathered from the functions we defined, that is, `quickSort`, `partition` and `swap`. Therefore, we execute the commands shown in Figure 5.6 in `rr`. Again, the results of the command `info breakpoints` have been simplified.

Figure 5.6: Setting the functions from which to gather correct nodes

```
(rr) save-correct-function quickSort(std::vector<int>&, int, int)
Breakpoint 1 at 0x40135e: file quicksort.h, line 44.
(rr) save-correct-function partition(std::vector<int>&, int, int)
Breakpoint 2 at 0x401256: file quicksort.h, line 20.
(rr) save-correct-function swap(int*, int*)
Breakpoint 3 at 0x401222: file quicksort.h, line 9.
(rr) info breakpoints
Num Type       Disp Enb Address         What
1   breakpoint keep y   0x40135e in quickSort(std::vector<int>&, int, int)
    add-node-to-correct-list quickSort(std::vector<int>&, int, int)
2   breakpoint keep y   0x401256 in partition(std::vector<int>&, int, int)
    add-node-to-correct-list partition(std::vector<int>&, int, int)
3   breakpoint keep y   0x401222 in swap(int*, int*)
    add-node-to-correct-list swap(int*, int*)
```

Figure 5.7: Starting an interactive Python shell inside rr

```
(rr) python-interactive
>>>
```

At this point, we can use the `GDB` commands `start` and `continue` (this last one repeatedly) to reach the end of the program. Alternatively, we have provided the convenience command `until-the-end` (see command until-the-end) that has the exact same functionality. Once we have reached the end of the program, all nodes have been collected. This can be checked by dropping into an interactive Python shell inside `rr` like is shown in Figure 5.7. To exit the Python shell, use `Ctrl-D`. Once inside the shell, we can examine the set of correct nodes. For example, we can check how many correct nodes we have gathered (see Figure 5.8) or display any of those nodes (see Figure 5.9).

Once all correct nodes have been collected, we have to start the debugging session of our buggy program with a separate `rr` execution (see Listing 5.3), which we will call rr_{client}, and collect the correct nodes already gathered from rr_{server}. This has to be done with the command `listen-for-correct-nodes}` (see command listen-for-correct-nodes).

Now we are ready to send the correct nodes from rr_{server} to rr_{client}. This is done with

Figure 5.8: Checking how many correct nodes have been gathered

```
>>> len(CORRECT_NODES)
156
>>> len(PENDING_CORRECT_NODES)
0
```

Figure 5.9: Printing a correct node

```
>>> print_tree(CORRECT_NODES[0])
partition(std::vector<int, std::allocator<int> >&, int, int)
├── arguments on entry
│   ├── my_vector = std::vector of length 6, capacity 6 = {
│   │     [0] = 2,
│   │     [1] = 8,
│   │     [2] = 1,
│   │     [3] = 8,
│   │     [4] = 6,
│   │     [5] = 8
│   │   }
│   ├── low = 0
│   └── high = 5
├── arguments when returning
│   └── my_vector = std::vector of length 6, capacity 6 = {
│         [0] = 2,
│         [1] = 8,
│         [2] = 1,
│         [3] = 8,
│         [4] = 6,
│         [5] = 8
│       }
└── return value
    └── 5
```

Figure 5.10: Sending nodes to client

```
Sending node n# 87
```

Figure 5.11: Receiving nodes from server

```
Connected by ('127.0.0.1', 51126)
```

the command `send-correct-nodes` (see command send-correct-nodes). In rr_{server} we can see that nodes are being sent, like shown in Figure 5.10. Likewise, in rr_{client} we can see that nodes are being received, like shown in Figure 5.11.

If we now start the debugging session (after setting the suspect functions like demonstrated in Figure 5.1), we see that the WMDT has 8 nodes (see Figure 5.14) instead of 9 (see Figure 5.13). This is because a swap node has been removed, more specifically, the node in Figure 5.12.

This node has been removed from the debugging tree because it appeared in a test case, therefore the debugger has deduced that this computation is correct.

5.2.3. Finding bugs in non terminating programs

As the main contribution of DDC, we present the process of debugging a non terminating program.

In Listing 5.6 we can see a non terminating implementation of the `partition` function. This is because the loop does not terminate, since we forgot to increment `j` after each iteration.

Listing 5.6: Non terminating partition implementation

```
1 int partition(std::vector<int> &my_vector, int low, int high)
2 {
3   int pivot = my_vector[high];
4   int i = (low - 1);
5   for (int j = low; j <= high - 1; j)
6     {
7       if (my_vector[j] <= pivot)
8         {
```

Figure 5.12: Correct node removed from debugging tree

```
swap(int*, int*)
├── arguments on entry
│   ├── a = 7
│   └── b = 1
└── arguments when returning
    ├── a = 1
    └── b = 7
```

Figure 5.13: Zoomable tree of buggy quickSort before correct node elimination

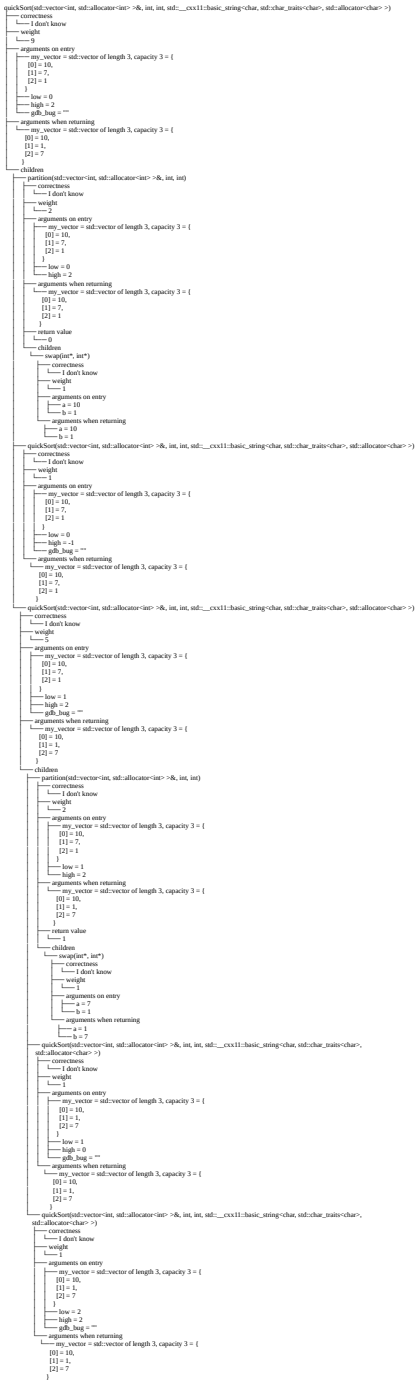

Figure 5.14: Zoomable tree of buggy quickSort after correct node elimination

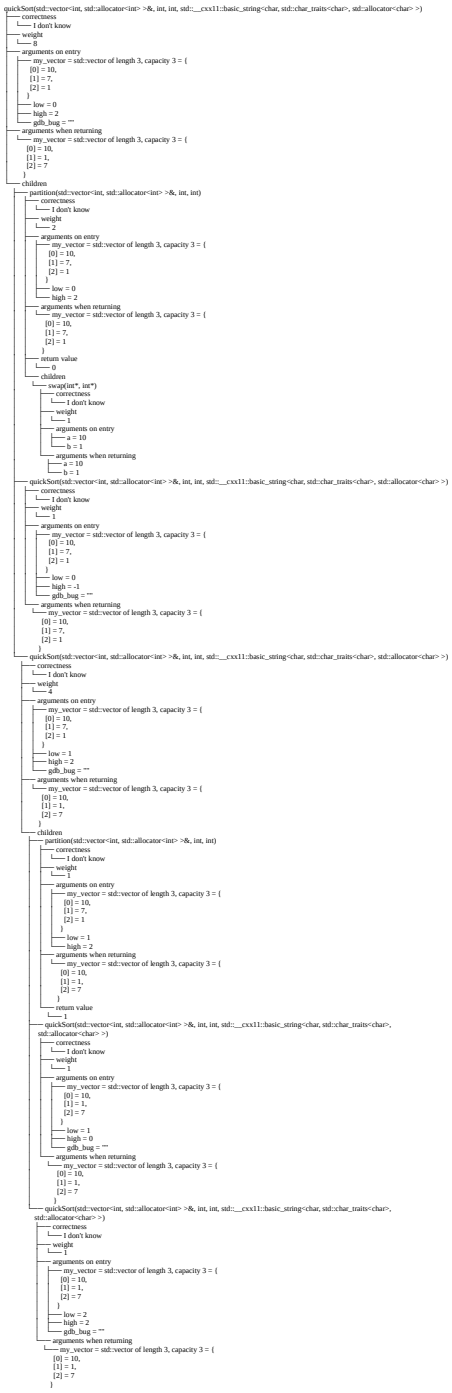

Figure 5.15: Buggy node of non terminating partition

```
partition(std::vector<int, std::allocator<int> >&, int, int)
├── correctness
│   └── I don't know
├── weight
│   └── 1
└── arguments on entry
    ├── my_vector = std::vector of length 6, capacity 6 = {
    │     [0] = 10,
    │     [1] = 7,
    │     [2] = 8,
    │     [3] = 9,
    │     [4] = 1,
    │     [5] = 5
    │   }
    ├── low = 0
    └── high = 5
```

```
9             i++;
10            if (i != j)
11              swap(&my_vector[i], &my_vector[j]);
12          }
13      }
14    if ((i + 1) != high)
15      swap(&my_vector[i + 1], &my_vector[high]);
16    return (i + 1);
17  }
```

The rest of the functions are unchanged from Listing 5.2 and the main function is the same as in Listing 5.1. First, we have to record this program execution, as shown in Listing 5.7.

Listing 5.7: Starting and stoping the recording of non terminating quickSort

```
1 rr record ./result/bin/quicksort
2 ^C
```

The `^C` command represents sending the `SIGINT` signal to rr, with stops the recording.

Now we can replay the recording, by executing `rr replay`. After, some questions, we find the node of partition in Figure 5.15. Note that the `return value` and the `arguments when returning` branches are missing, indicating that this function did not return.

5.3. Commands

We now list and describe all the available commands. These can be issued inside rr. Completion for these commands is enabled, that is, if you type `sus` and then `TAB`, `suspect-function` should appear. The following information can also be found with `help <command>`, where `<command>` is one of the commands that follow.

1. suspect-function

Mandatory command used in a debugging session.

Sets a breakpoint on the location provided as argument. It also adds the command add-node-to-session to the breakpoint created. It should be used at least once before starting the debugging session.

Arguments: location.

Location completion is enabled.

2. add-node-to-session

   Optional command used in a debugging session. Adds the current frame to the debugging tree. It expects to be called on entry to a function or method. It also creates a finish breakpoint for the current function or method, with the attach command save-returning-node.

   It can be used to create an ad hoc debugging tree.

   It is used by suspect-function.

   Arguments: none.

3. save-returning-node

   It saves the return value of the function or method that just returned. This is the command attached to finish breakpoints created with add-node-to-session.

   For internal use only.

   Arguments: none.

4. final-point

   Optional command used in a debugging session. Sets a breakpoint on the location provided as argument and adds the command finish-debugging-session to it. It can be used zero or more times.

   Arguments: location.

   Location completion is enabled.

5. finish-debugging-session

   Sets the Boolean variable describing that the tree has been built to true. This command is executed when a final-point is reached.

   Internal use only.

   Arguments: none.

6. start-declarative-debugging-session

   Mandatory command used in a debugging session. If the debugging tree has not been built, it builds it. Otherwise, it starts to ask correctness questions to the user.

   Arguments: none.

7. save-correct-function

   Sets a breakpoint on the location provided as argument, setting as command save-returning-correct-node. Mandatory command used in gathering correct nodes.

   Arguments: location.

   Location completion is enabled.

8. add-node-to-correct-list

   Optional command used in gathering correct nodes. Adds the current frame to the debugging tree. It expects to be called on entry to a function or method. It also sets a final breakpoint on current function or method, with the attached command save-returning-correct-node.

   Arguments: none.

   For internal use only.

9. save-returning-correct-node

   It saves the return value of the function or method that just returned into the appropriate node.

   Arguments: none.

   For internal use only.

10. until-the-end

    Continues the execution of the inferior until the program is not being run.

    Optional command used in gathering correct nodes.

    Arguments: none.

11. listen-for-correct-nodes

    Open a connection on localhost, port 4096, and wait for correct nodes to be sent.

    Executed by the client. Mandatory command used in gathering correct nodes. Must be executed before calling send-correct-nodes in the server.

    Arguments: none.

12. send-correct-nodes

    Send all the correct nodes gathered from tests to the client. Executed by the server. The client has to have issued the listen-for-correct-nodes before, otherwise a `ConnectionRefusedError` exception is thrown.

    Mandatory command used in gathering correct nodes.

    Arguments: none.

13. print-tree

    Prints the debugging tree. An exception is thrown if the debugging tree has not been built yet. Optional command that can be used in a debugging session.

    Arguments: none.

Chapter 6

Verification and benchmarks

In this chapter, we present the formal verification done in Coq and time benchmarks of the execution of DDC.

6.1. Verification

To help with the verification effort, we have chosen the Coq proof assistant (5). The proofs can be found in the following file: `https://github.com/RCoeurjoly/DDC/blob/database/proofs.v`.

We now present the types, functions, and lemmas defined in Coq.

6.1.1. Types

We have defined two inductive types:

- Correctness (see Figure 6.1): a simple inductive type, with the possible values presented in Chapter 2.
- Node (see Figure 6.2): an inductive recursive type, whose constructor, `mkNode`, takes as arguments:
  - a string to represent the content (f, I and O in Chapter 4) of the node,
  - a `Correctness` value and
  - a list of `Node`s.

Figure 6.1: Correctness type in Coq

```
Inductive Correctness : Type :=
| yes : Correctness
| no : Correctness
| trusted : Correctness
| idk : Correctness.
```

Figure 6.2: Node type in Coq

```
Inductive Node : Type :=
  mkNode
    {
    content : string
    ;correctness : Correctness
    ; children : list Node
    }.
```

6.1.2. Functions

Apart from the generic debugging algorithm, we have designed some auxiliary functions needed to reason about debugging trees:

- `or_list`: recursive function to calculate the disjunction of a list of propositions (Figure 6.3).
- `and_list`: recursive function to calculate the conjunction of a list of propositions (Figure 6.4).
- `is_node_in_tree`: recursive function to find if the content of a node is in a tree (Figure 6.5).
- `weight`: recursive function to calculate the weight of a tree, inspired by the definition in (19) (Figure 6.6).
- `are_all_idk`: recursive function to check that all nodes in tree have a correctness value of `I don't know` (Figure 6.7).
- `is_debugging_tree`: check if a tree is a debugging tree, defined as the root being incorrect and all children satisfy `are_all_idk` (Figure 6.8).
- `get_debugging_tree_from_tree`: returns the same tree as the input, only changing the correctness of the root node to incorrect (Figure 6.9).
- `generic_debugging_algorithm`: a dependently-typed program, which takes as input a debugging tree and finds the buggy node. We provide a measure, the weight of the input tree, which we must prove it decreases with each recursive call to prove termination of the algorithm (Figure 6.10). To simplify the proofs, the navigation strategy used is Top-down and the user always answers that the node is incorrect, therefore choosing the first child as the next focus of the algorithm.

6.1.3. Lemmas

The main goal of our verification effort is to confirm that the general debugging algorithm defined in Figure 6.10 type checks. To do this, we have to discharge four proof obligations:

1. The algorithm returns a debugging tree without children nodes when it receives as input a debugging tree without children nodes.

   This is proved by simplification, that is, execution, of the pattern matching.

Figure 6.3: or_list function in Coq

```
Fixpoint or_list (l : list Prop) : Prop :=
  match l with
    nil => False
  | hd::tl => or hd (or_list tl)
  end.
```

Figure 6.4: and_list function in Coq

```
Fixpoint and_list (l : list Prop) : Prop :=
  match l with
    nil => True
  | hd::tl => and hd (and_list tl)
  end.
```

Figure 6.5: is_node_in_tree function in Coq

```
Fixpoint is_node_in_tree (c : string) (m : Node) : Prop :=
  or (c = content m)
     (or_list
        (map
            (fun child => is_node_in_tree c child)
                (children m))).
```

Figure 6.6: weight function in Coq

```
Fixpoint weight (node : Node) : nat :=
  match children node with
    nil => 1
  | children => S (list_sum (map (fun child => weight child) (children)))
  end.
```

Figure 6.7: are_all_idk function in Coq

```
Fixpoint are_all_idk (node : Node) : Prop :=
  and
    (node.(correctness) = idk)
    (and_list
        (map
            (fun child => are_all_idk child)
                (children node))).
```

Figure 6.8: is_debugging_tree function in Coq

```
Definition is_debugging_tree (node : Node) : Prop :=
  and
    (node.(correctness) = no)
        (and_list
            (map
                (fun child => are_all_idk child)
                    (children node))).
```

Figure 6.9: get_debugging_tree_from_tree function in Coq

```
Definition get_debugging_tree_from_tree (n : Node) : Node :=
  mkNode (content n) no (children n).
```

Figure 6.10: generic_debugging_algorithm function in Coq

```
Program Fixpoint generic_debugging_algorithm
    (n : {n: Node | is_debugging_tree n})
    {measure (weight n)}:
    {m: Node | is_debugging_tree m /\ children m = nil} :=
  match children n with
    nil => n
  | head::tail => generic_debugging_algorithm
    (get_debugging_tree_from_tree head)
  end.
```

2. The algorithm returns a debugging tree when it receives a debugging tree with children nodes as input. This is proved by noting that:
   - `are_all_idk head` is a true proposition, since all children nodes of a debugging tree satisfy `are_all_idk` and
   - `get_debugging_tree_from_tree` returns a debugging tree when it receives a `are_all_idk` tree, which is the case for `head`.
3. The weight of `get_debugging_tree_from_tree head` is smaller than the weight of the input tree. This is proved by noting that:
   - the weight of `get_debugging_tree_from_tree head` is equal to the weight of `head`, since `get_debugging_tree_from_tree` always returns a tree with the same weight as its input tree and
   - the weight of `head` is smaller than the weight of its parent.
4. The weight of the input tree is a well-founded measure.

   This is proven by noting that the `weight` function (see Figure 6.6) returns a natural number, which is well-founded.

To discharge the proof obligation of the generic debugging algorithm, twenty four auxiliary lemmas were defined.

6.2. Benchmarks

In this section we present and discuss different time benchmarks of the execution of DDC. We also list the performance bottlenecks found.

6.2.1. Quicksort benchmarks

To gather benchmarks easily, we provide a quicksort implementation that gets its input vector from the command line. Then, we generate random lists of different lengths and pipe these lists to quicksort. All data was gathered with an Intel Core i5-4570 CPU @ 3.20GHz, with 4 cores and 16Gb of RAM computer.

In Figure 6.11 we plot the different execution times.

- `Execution` represents the native execution of the quicksort program.
- `Recording` represents the recording of the quicksort program with `rr`.
- `Tree building (main memory)` represents the building of the debugging tree with DDC, storing the tree in main memory.
- `Tree building (database)` represents the building of the debugging tree with DDC, storing the tree in a `MySql` database.
- `Trace debugging (GDB)` represents the time `GDB` needs to, after setting the three breakpoints, navigate until the end.

  The three breakpoints are set in:

  - quicksort,

- partition, and
- swap.

- `Trace debugging (rr)` represents the time `rr` needs to, after setting the three breakpoints, navigate until the end.

To reduce the time needed to build the debugging tree, we have created a branch in DDC (called `database`) in which we stored the debugging tree in a database, as proposed in (18). However, this did not improve the time significantly, as can be seen in Figure 6.11. Storing the debugging tree in a database is 43 percent faster than storing it in main memory (50 seconds versus 87 seconds for a input vector length of 256).

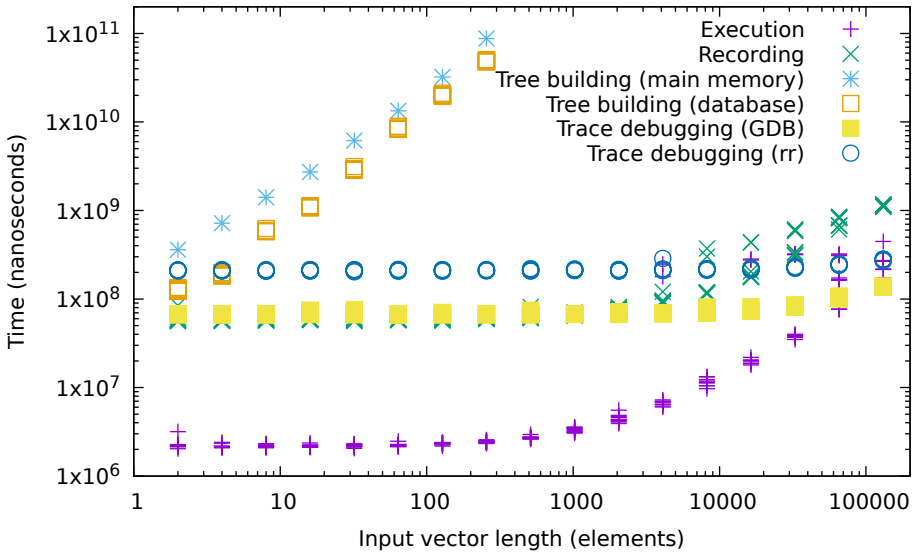

Figure 6.11: Quicksort execution vs record vs tree building

In Figure 6.12 we plot the debugging tree building time depending on the number of suspect functions set, for the same input list, which in this case is a list of length 256. As we can see, the time required for building the tree increases with the number of suspect functions set. The three data points correspond to the following:

- 3 suspect functions: quicksort, partition, and swap.
- 2 suspect functions: quicksort and partition.
- 1 suspect function: quicksort.

In Table 6.1 we can see the profile results of building a debugging tree with DDC. In this case, the program input is a quicksort execution with input vector length 256, and three suspect functions set (quicksort, partition, and swap).

We have identified two bottlenecks:

- Reverse stepping into a returning function (24.926 seconds).

  This is needed to gather the object state, output arguments and global variables, denoted O_o, O_a, and O_g respectively in Chapter 4.

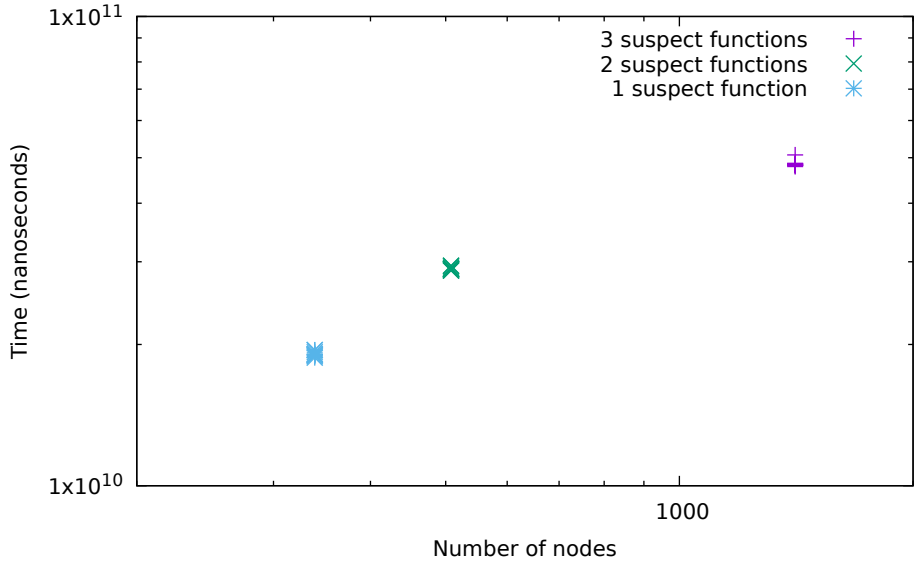

Figure 6.12: Nodes vs time

Table 6.1: DDC (database) profiling for quicksort with input vector length 256

cumtime	filename:lineno(function)
54.184	{built-in method builtins.exec}
54.184	<string>:1(<module>)
54.184	{built-in method _gdb.execute}
54.184	{declarative_debugger}.py:505(invoke)
38.693	{declarative_debugger}.py:282(invoke)
24.926	{declarative_debugger}.py:272({reverse_stepi})
16.078	{declarative_debugger}.py:847({print_value})
16.063	{method 'format_string' of 'gdb.Value' objects}
9.641	{declarative_debugger}.py:352(invoke)
8.281	{declarative_debugger}.py:374(<listcomp>)
7.804	{declarative_debugger}.py:299(<listcomp>)

- Getting the string representation of a value (16.063 seconds).

  This is needed to store the value (argument, object, global variable, or return value) into the database.

This two bottlenecks constitute around 75 percent of total execution time.

6.2.2. Z3 benchmarks

To test DDC with a industrial grade program, we chose the SMT solver Z3 (11). The execution used for these benchmarks is the simplest execution possible. We pass the SMT file shown in Figure 6.13 to Z3:

In Figure 6.14 we see the relationship between number of nodes versus execution time. We get the same relationship as in with quicksort data.

In Figure 6.15 we plot the number of breakpoints set versus execution time. We compare stopping at breakpoints without doing anything (data labeled `Plain breakpoints`) with creating a node each time a breakpoint is reached (data labeled `Tree building`).

Figure 6.13: SMT file passed to Z3 for benchmarking

```
(assert true)
(check-sat)
```

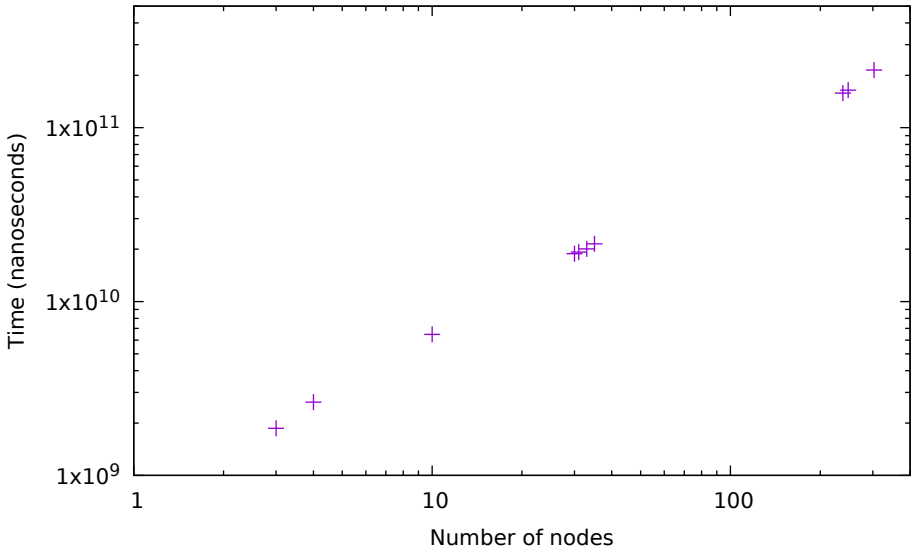

Figure 6.14: Nodes vs time (Z3)

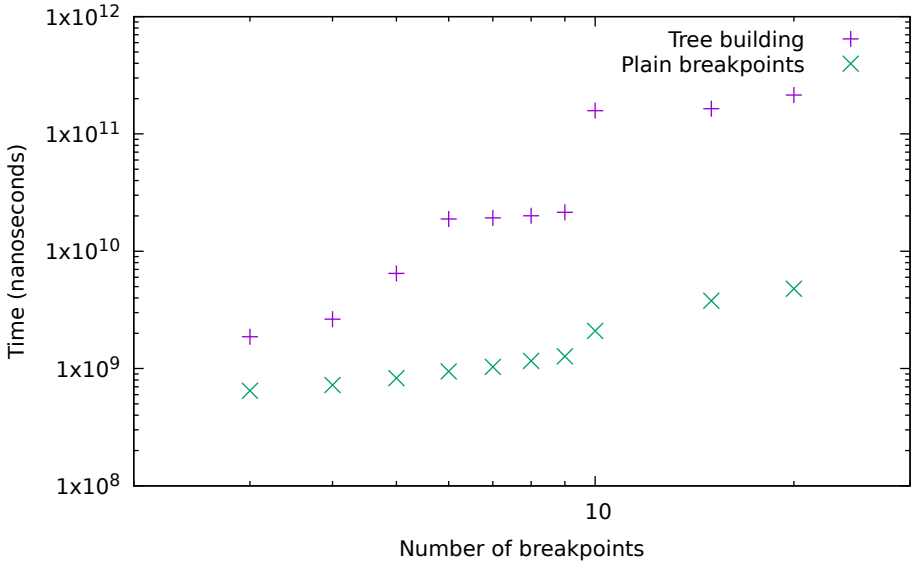

Figure 6.15: Breakpoints vs time (Z3)

Chapter 7

Conclusions and Future Work

We now proceed to discuss the contributions made, and how they compare to our initial goals.

7.1. Conclusions

We have successfully developed a declarative debugger for C++, fulfilling all its initial goals. In particular, we have achieved Goal 1 (integrated in workflow) by using GDB and rr as the foundation to DDC. GDB is arguably the most used C++ debugger in GNU/Linux and rr enhances GDB[1]. Once familiar with GDB, the commands provided by DDC can be easily inspected and interacted with. For example, when issuing the `suspect-function` command, auto-completion is enabled. Also, once the command is issued, the resulting breakpoint can be handled like any other breakpoint. This can be seen in figure 7.1, where after setting the suspect function (with command `suspect-function swap(int*, int*)`), we check that it appears in the list of breakpoints (`info breakpoints`), then disable it (`disable 1`) and finally delete it (`delete 1`).

During the navigation phase, DDC provides the user with an intuitive command line interface, avoiding a context switch, which can be as disruptive as an interruption (1). More importantly, an algorithmic debugging session can be intertwined with a generic debugging session in any way, that is, the user can start the algorithmic debugging session after inspecting the program and vice versa. With this functionality, we address Issue-6 raised in the survey (8) by

> allowing the user to switch from a trace debugger to the algorithmic debugger easily

.

We also consider Goal 2 (no changes to program) achieved. No code instrumentation, such as adding `# include` or `# define` directives, is needed to debug a program with DDC. Only compiling with debug symbols is required.

Goal 3 (tree transformations) has been fulfilled by implementing the simplified tree compression algorithm. This tree transformation reduces the tree size of recursive programs, therefore reducing the amount of questions the user has to answer. Also, we have built the necessary infrastructure to integrate other tree transformation painlessly in the future.

[1] https://rr-project.org/

Figure 7.1: Setting, checking, disabling and deleting a suspect function

```
(rr) suspect-function swap(int*, int*)
Breakpoint 1 at 0x401222: file quicksort.h, line 9.
(rr) info breakpoints
Num Type           Disp Enb Address  What
1   breakpoint     keep y   0x401222 in swap(int*, int*) at quicksort.h:9
    add-node-to-session swap(int*, int*)
(rr) disable 1
(rr) info breakpoints
Num Type           Disp Enb Address  What
1   breakpoint     keep n   0x401222 in swap(int*, int*) at quicksort.h:9
    add-node-to-session swap(int*, int*)
(rr) delete 1
(rr) info breakpoints
No breakpoints or watchpoints.
```

The final goal, Goal 4 (test cases as oracles), has also been accomplished. Although another debugging session has to be started to send the correct nodes to the client, we consider that this is convenient enough not to deter its usage. The collection of the correct nodes directly, without starting another debugging session, would be possible if rr supported changing the executable in a running session (analog to the `run` command in GDB), or if we removed rr as a dependency. However, we find it unlikely to remove the rr dependency since GDB support for reverse-stepping (15), needed to gather O_o, O_a and O_g (defined in Chapter 4), is much less robust than in `rr`.

Lastly, a notable feature of DDC not included in the initial set of goals is the ability to support several languages (16), not only C++. This has been accomplished by the use of the GDB Python API, which uses abstractions like frames, symbols and variables that are common to all languages supported by GDB. Although more work is needed to fully support languages other than C++ (see Future work below), we estimate that the effort to do so is comparatively small.

7.2. Future work

During the research, development and use of DDC, we have found the following research directions to pursue in the future.

7.2.1. Support building tree without suspecting a function or method

So far, the user has to suspect at least one function or method before DDC can build the debugging tree. Ideally, DDC would be able to build the DT without any suspect breakpoints, by creating a node every time it enters a new function. Since we already have functionality to prune the DT of excess nodes (by trusting functions and using test cases as oracles, for example), the user could then reduce the size of the DT.

7.2.2. Complete the formal verification of all algorithms

The following verifications remain to be done:

- The generic debugging algorithm returns a node that was in the initial debugging tree.
- The generic algorithm is correct and complete for all possible navigation strategies and user answers.

  For this, we would need to model the user as a probability distribution of all the possible answers. We could use a Coq library such as ALEA (4).

Also, it would advisable to use the program extraction facility of Coq (23) and use the verified algorithms from Python, limiting the use of Python code to the user interface (UI). In Nagini (12), the tool we initially selected to perform the verification, we identified a bug[2], which was confirmed by Nagini main author, that blocked further work with Nagini.

7.2.3. Support for C-style arrays

C++ arrays are inherited from C to maintain backward compatibility. As illustrated in Figure 7.2, when entering a function, an array changes its type to become a pointer to the type of the elements of the array. In this example, `arr` is of type `int [6]` before entering `quickSort` and becomes `int *` on entry. This makes it impossible to display an array to the user safely, since we do not know its size. To avoid dealing with this issue, the example programs provided with DDC use `std::vector` instead of C-style arrays.

Figure 7.2: Integer array variable changes to integer pointer upon entry to function

```
(rr) ptype arr
type = int [6]
(rr) c
Continuing.
(rr) ptype arr
type = int *
```

7.2.4. Test programming languages other than C++

The Rust code in listing 7.1 produces the node represented in figure 7.3.

Listing 7.1: QuickSort in Rust

```
1 pub fn quick_sort<T: Ord>(arr: &mut [T]) {
2     let len = arr.len();
3     _quick_sort(arr, 0, (len - 1) as isize);
4 }
```

This tree has two problems:

- Although it has already returned and the argument `arr` is mutable, the tree has no `args when returning` branch.

[2] https://github.com/marcoeilers/nagini/issues/150

Figure 7.3: Tree of quicksort using C-style arrays

```
quicksort::quick_sort<i32>
├── correctness
│   └── I don't know
├── weight
│   └── 3
└── args on entry
    └── arr = &mut  {
          data_ptr: 0x7ffd36788d80,
          length: 10
        }
```

- The representation of arr is not very helpful to the user, especially `data_ptr`.

The `args when returning` branch is created when the following function returns a non empty list:

Listing 7.2: Python function to select arguments passed as reference or pointer

```
1 def get_pointer_or_ref(arguments, frame):
2     if arguments is None:
3         return []
4     return [argument for argument in arguments
5             if argument.value(frame).type.code in [gdb.TYPE_CODE_PTR,
6                                                    gdb.TYPE_CODE_REF]]
```

Therefore, we must deduce that the argument `arr` is neither a pointer nor a reference. We should find out which type it is to support this Rust program.

7.2.5. Support concurrent programs

Currently, DDC does not support the debugging of concurrent programs. This is mainly due to the fact that `rr` does not support multi threaded programs (28). We could remove the rr dependency to support concurrency, but reverse execution in `GDB` (15) is more limited than in `rr`.

7.2.6. Implement more strategies

As mentioned in the survey (8), several navigation strategies have been developed. Of special interest is Optimal Divide and Query (19), which has been proven correct and optimal. One difficulty we may face when developing this strategy is that the debugging tree has to be marked with different weights.

7.2.7. Generate test cases from correct nodes

When the user confirms the correctness of a certain node, a test case should be generated. This presents us with several challenges:

- Once the test case is created, should we present them via terminal to the user or insert them in a file? If inserting them is desired, should we create a new test file or insert them in an existing one?

- Creating the objects and variables requires a deep understanding of C++. Using the Clang Python API from LLVM (22) may be needed.
- Testing private methods requires creating a friend class to access them. Another option would be to skip them altogether and only provide test cases for public methods.

Bibliography

Abad, Z. S. H., Noaeen, M., Zowghi, D., Far, B. H., and Barker, K. (2018). Two sides of the same coin: Software developers' perceptions of task switching and task interruption.

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. (2016). Tensorflow: A system for large-scale machine learning.

Akritidis, P. and Akritidis, P. (2011). Practical memory safety for c.

Audebaud, P. and Paulin-Mohring, C. (2009). Proofs of randomized algorithms in coq. *Science of Computer Programming*, 74(8):568–589. Special Issue on Mathematics of Program Construction (MPC 2006).

Bertot, Y. and Castéran, P. (2013). *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media.

Buterin, V. (2013). Ethereum white paper: A next generation smart contract & decentralized application platform.

Caballero, R., Martin-Martin, E., Riesco, A., and Tamarit, S. (2021). A unified framework for declarative debugging and testing. *Information and Software Technology*, 129:106427.

Caballero, R., Riesco, A., and Silva, J. (2017). A survey of algorithmic debugging. *ACM Comput. Surv.*, 50(4).

Carbonnelle, P. (2022). Pypl popularity of programming language. `https://pypl.github.io/PYPL.html`. Accessed: 2022-01-06.

Clerc, X. and Alekseyev, A. (2022). pythonlib. `https://github.com/janestreet/pythonlib`. Accessed: 2022-06-20.

de Moura, L. M. and Bjørner, N. (2008). Z3: An efficient smt solver. In *TACAS*.

Eilers, M. and Müller, P. (2018). Nagini: A static verifier for python. In Chockler, H. and Weissenbacher, G., editors, *Computer Aided Verification*, pages 596–603, Cham. Springer International Publishing.

Foundation, S. C. (2022a). When was c++ invented? `https://isocpp.org/wiki/faq/big-picture#when-invented`. Accessed: 2022-01-06.

Foundation, S. C. (2022b). Why is c++ (almost) compatible with c? `https://isocpp.org/wiki/faq/big-picture#why-compat-with-c`. Accessed: 2022-01-06.

Free Software Foundation, I. (2022a). Running programs backward. `https://sourceware.org/gdb/onlinedocs/gdb/Reverse-Execution.html`. Accessed: 2022-01-06.

Free Software Foundation, I. (2022b). Supported languages. `https://sourceware.org/gdb/current/onlinedocs/gdb/Supported-Languages.html`. Accessed: 2022-01-06.

Inria, C. and contributors (2022). Tactics. `https://coq.inria.fr/refman/proof-engine/tactics.html`. Accessed: 2022-06-20.

Insa, D. and Silva, J. (2010). An algorithmic debugger for java. In *2010 IEEE International Conference on Software Maintenance*, pages 1–6.

Insa, D. and Silva, J. (2011). An optimal strategy for algorithmic debugging. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 203–212.

Insa, D. and Silva, J. (2018). Algorithmic debugging generalized. *Journal of Logical and Algebraic Methods in Programming*, 97:85–104.

Insa, D., Silva, J., and Tomás, C. (2013). Enhancing declarative debugging with loop expansion and tree compression. In Albert, E., editor, *Logic-Based Program Synthesis and Transformation*, pages 71–88, Berlin, Heidelberg. Springer Berlin Heidelberg.

Lattner, C. (2002). LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL. *See* `http://llvm.cs.uiuc.edu`.

Letouzey, P. (2003). A new extraction for coq. In Geuvers, H. and Wiedijk, F., editors, *Types for Proofs and Programs*, pages 200–219, Berlin, Heidelberg. Springer Berlin Heidelberg.

Lloyd, J. W. (1987). Declarative error diagnosis. *New Gen. Comput.*, 5(2):133–154.

Matloff, N. and Salzman, P. J. (2008). *The Art of Debugging with GDB, DDD, and Eclipse*. No Starch Press, USA.

Moura, L. d. and Ullrich, S. (2021). The lean 4 theorem prover and programming language. In Platzer, A. and Sutcliffe, G., editors, *Automated Deduction – CADE 28*, pages 625–635, Cham. Springer International Publishing.

Nakamoto, S. (2009). Bitcoin: A peer-to-peer electronic cash system.

O'Callahan, R., Jones, C., Froyd, N., Huey, K., Noll, A., and Partush, N. (2017). Engineering record and replay for deployability: Extended technical report.

O'Dell, D. H. (2017). The debugging mindset: Understanding the psychology of learning strategies leads to effective problem-solving skills. *Queue*, 15(1):71–90.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc.

Plugge, E., Hawkins, T., and Membrey, P. (2010). *The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing*. Apress, USA, 1st edition.

Schling, B. (2011). *The Boost C++ Libraries*. XML Press.

Shapiro, E. Y. (1982). *Algorithmic Program Debugging*. PhD thesis, MIT.

Silva, J. (2011). A survey on algorithmic debugging strategies. *Advances in Engineering Software*, 42(11):976–991.

Sozeau, M. (2022). Progam. `https://coq.inria.fr/refman/addendum/program.html#program`. Accessed: 2022-06-20.

Stallman, R. M. and DeveloperCommunity, G. (2009). *Using The Gnu Compiler Collection: A Gnu Manual For Gcc Version 4.3.3*. CreateSpace, Scotts Valley, CA.

Widenius, M., Axmark, D., and DuBois, P. (2002). *Mysql Reference Manual*. O'Reilly & Associates, Inc., USA, 1st edition.

Zeller, A. (2006). Chapter 1 - how failures come to be. In Zeller, A., editor, *Why Programs Fail*, pages 1–26. Morgan Kaufmann, San Francisco.