# 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:** *Febrero/Junio/Septiembre 2022*
**Calificación:** *Nota*

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

**DIA de MES de AÑO**

# Dedicatoria

*A Chun, por el apoyo*

# Acknowledgments

A Adrián, por su paciencia.

# Resumen

## DDC: un depurador declarativo para C++

Un resumen en castellano de media página, incluyendo el título en castellano. A continuación, se escribirá una lista de no más de 10 palabras clave.

## Palabras clave

Máximo 10 palabras clave separadas por comas

# Abstract

## DDC: a declarative debugger for C++

A declarative debugger for C++ is presented. A declarative debugger receives as input an incorrect computation and, after asking questions to an oracle (typically the user), outputs the node which is the root cause of the failure. We summarize all the relevant design decisions made, like building the execution tree and asking the user about the correctness of a certain node. We present the debugger's main features, such as two different strategies to traverse the debugging tree and selection of trusted vs. suspicious statements by means of labels.

## Keywords

declarative debugging, execution tree, C/C++

# Contents

# List of figures

# List of tables

# Listings

Chapter 1

# Introduction

*"Program testing can be used to show the presence of bugs, but never to show their absence!"*
— Edsger Dijkstra

In this section, 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 (**?** ), and one of its goals is to maintain backward compatibility with C (**?** ). As such, C++ has inherited the defects of C, such as lack of memory safety (**?** ). Despite these shortcomings, C++ is still one of the most used programming languages (**?** ), including being used in such important areas such as:

- Compilers: GCC (**?** ), LLVM (12).
- Databases: MySQL (**?** ), MongoDB (**?** ).
- Theorem provers: Z3 (**?** ), Lean (10).
- Debuggers: GDB (**?** ), rr (11).
- Machine learning frameworks: TensorFlow (**?** ), PyTorch (**?** ).
- Digital currency and smart contracts technology: Bitcoin (**?** ), Solidity (**?** ).

Furthermore, C++ is actively developed, with the last release being C++20. This, together with the upward trending usage statistics (**?** ), 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 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 (1).

Ya suponemos q
la gente sabe qu
es el debugging,
pero cuenta alg
más: qué es, est
los, depuradores
de C++ en IDE
Puedes simplem
esbozar algunas
cosas si las desa
las en prelimina

There are many debugging approaches, ranging from the fully manual to the almost fully automatic, like delta debugging or program slicing (**?** ). In this thesis we focus on declarative debugging, also called algorithmic debugging, which is a semi automatic approach in which the debugger builds an execution 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 expands the definition of declarative debugging in Chapter 2.

To the best of our knowledge, there is no declarative debugger for C++.

## 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 program: 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 execution tree.
4. Test cases as oracles: it has to use test cases to reduce the tree size.

Hablas de un de
rador declarativ
pero el lector no
tiene la idea int
itiva de qué es.
laza la idea en e
párrafo anterior

## 1.3. Main contributions

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

- Support for several programming languages, by means of using GDB.
- Non terminating programs can be debugged.
- Test cases can be used as oracles to reduce the tree size.
- 3 navigation strategies developed.
- 1 tree transformation developed.
- Easily extensible (more strategies, more 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 and the different tools needed to implement DDC.
- In Chapter 3 we review the state of the art in declarative debugging.
- Chapter 4 contains the definitions and proofs that form the theoretical basis of DDC.
- Chapter 5 presents the most important usage scenarios of the tool, describes how the debugger was implemented, lists its commands and compares its features to other declarative debuggers.
- Chapter 6 discuses the achievements and the future lines of work that can be taken.

Chapter 2

# Preliminaries

## 2.1. Declarative debugging

Declarative debugging, also called algorithmic debugging, is a debugging technique that consists in asking questions about the correctness of computations to the user until either a certain computation is narrowed down as buggy or no buggy computation is found.

A declarative debugger (DD) takes as argument a program execution which the user deems incorrect. The DD builds an execution tree (ET) of the execution of this program. Once the ET is built, the DD traverses it in a semi-automatic fashion, asking questions to the user about the correctness of sub-computations, until a buggy computation is found or no computations remain.

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

- Building the execution tree
- Simplifying the execution tree
- Navigating the execution tree

Falta contar qu
recorre de mane
semi-automátic
hasta que se de
un error

Cuenta que en l
siguientes subse
ciones contarás
y cual

No se si tanto d
talle esta bien p
erlo aqui o en E
tado del la ques

### 2.1.1. Building the execution tree

Building the execution tree (ET) is the central, critical stage of the debugging process. To build the ET, the DD must have access to the following information:

- When a new function or method begins executing, and state of variables at that point.
- When a new function or method returns, and state of variables at that point.
- Which function or method is the caller and which is the callee.

With this information, the debugger can build a node, and from nodes it can build the ET.

For formal definitions, please see chapter 4.

### 2.1.2. Simplifying the execution tree

Since the number of questions depends on the size of the tree (number of nodes, width and depth of the tree), most DDs provide certain functionality to reduce the ET size.

#### 2.1.2.1. Trusting computations

Most DDs built the ET assuming all functions or methods are suspect of being wrong. However, this is hardly ever the case.

By trusting some functions or methods, the debugger

In the case of debugging a C++ program, one option 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 (**?** ). This process requires manual input from the user in selecting those computations that can be trusted.

#### 2.1.2.2. Suspecting computations

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 in developing DDC. This process requires manual input from the user in selecting those computations that should be suspected.

#### 2.1.2.3. Using test cases as oracles

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

This functionality requires manual input from the user in selecting the test case to be executed.

This functionality is provided by DDC.

#### 2.1.2.4. Tree transformations

Tree transformations happen once the ET 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. This process is fully automatic, not needing any input from the user.

### 2.1.3. Navigating the execution tree

## 2.2. C++ programming language

The C++ programming language is a general-purpose, statically typed, compiled language (**?** ).

C++ supports several programming paradigms, including:

- Object-oriented programming (OOP).
- Functional programming.
- Generic 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 run time. Therefore, supporting OOP implies support for generic programming.

The requirements for a DD that supports OOP are greater than for functional programming, since 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 and 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. 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. Technologies used

### 2.3.1. GDB: The GNU Project Debugger

GDB is the most common debugger for C++ programs in GNU/Linux.

We make extensive use of the Python API, especially of the following classes:

#### 2.3.1.1. Frames

We use frames to determine the relationship between nodes. A frame

#### 2.3.1.2. Breakpoints

A breakpoint is a bookmark set in a certain location of a program, such a function or specific line, that 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 are 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, such as function or method name,

#### 2.3.1.3. Final breakpoints

#### 2.3.1.4. Values

#### 2.3.1.5. Symbols

### 2.3.2. rr: Record and Replay Framework

rr enhances GDB, providing support for record and replay functionality, as well as 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:

- Object state when returning.
- Arguments when returning (if passed as reference or pointers).
- Global variables when returning.

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

### 2.3.3. Nix: the purely functional package manager

Chapter 3

# State of the art

## 3.1.

# Chapter 4

# Definitions

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

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

- *$f$ is the name of the function/method executed.*
- *$I$ is the 3-tuple of inputs to $f$, $I = \{I_o, I_a, I_g\}$, where:*
  - *$I_o$ is the object state when it was called (if it is a method call).*
  - *$I_a$ is the n-tuple of input arguments when it was called (if any).*
  - *$I_g$ is the set of global variables when it was called (if any).*
- *$O$ is the 4-tuple of outputs of $f$, $O = \{O_o, O_a, O_g, O_r\}$, where:*
  - *$O_o$ is the object state when it returned (if it is a method call).*
  - *$O_a$ is the m-tuple of output arguments when it returned (if there where 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 (if any).*
  - *$O_r$ is the return value (if any).*

**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 constists 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. 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.*

Buscar palabra
mejor

Hablas mucho d
conjuntos pero
veces son tuplas
listas, no? Aquí
en adelante

He puesto tupla
de las dimension
apropiadas en v
de sets

Las variables gl
ales las pongo c
set

Aqui comparo lo
tamaños de las
plas de $I_a$ y $O_a$

Sale un poco ra
numeración de l
ejemplos

Con definir el te
rema exmpl al i
de la pagina qu
igual que el rest

when executing
Explica además
poco el listing s
se ha hecho ant

Explica un poco
qué llamada hac
y en qué contex
y luego explica
poco cada rama

- $I_g$ *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 have not.*

- $O_g$ *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

```
int number_of_cars = 0;

class Car {
public:
  Car() {
    x = 0;
    y = 0;
    number_of_cars++;
  };
  bool move(const int& xDelta, const int& yDelta) {
    x += xDelta;
    y += yDelta;
    return true;
  };
private:
  int x;
  int y;
};

int main() {
  Car my_car;
  my_car.move(10, 5);
}
```

**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 parent node.*

oblema es que
odo es una lla-
a a función, por
e falta algo, la
n de cuerpo
contiene la lla-
a a la función

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

- *A node can have 0 or more children nodes.*

Figure 4.1: Node representing the execution of a 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 children nodes is a leaf node.*
- *A node can have 0 or 1 parent nodes.*
- *A node that has no edge leading to its 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 (2)) to Weighted Marked Execution Tree (WMET).

**Definition 3** (Weighted marked execution tree)**.** *A weighted marked execution 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 WMET of partition (see 5.2 for the corresponding code) called with $my\_vector = \{10, 7, 8, 9, 1, 5\}$, $low = 0$ and $high = 5$. In it we can see three nodes:*

De qué tipo, eje
tando qué funci
sobre qué códig

- *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*

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

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

**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 $II_{swap}(I) \to O'$, $I$ being two pointers pointing to 10 and 1 respectively. On the other hand, Figure 4.4 is the node $n_{swap}(I) \to O$ 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.*

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

**Theorem 1.** Let $T = (N, E, W, M)$ be a weighted marked execution tree and S the top down strategy. If S receives T as input, it always terminates, producing either $\perp$ or $n' \in N$ as output.

*Proof.* See (4). □

**Theorem 2.** Let $T = (N, E, W, M)$ be a weighted marked execution tree, 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 $ii$ such that $II : f \to ii$, and S the top down strategy.

be este con-
de tal manera
ea comparable
nodos.

de arriba es
onjunto que se
ruye con ele-
os con pinta
odos lo que
nos es que el
unto no incluye
do. Lo mismo
ués.

o si hay que
parar $II_f(I) \to$
$f(I) \to O$ o
$O$

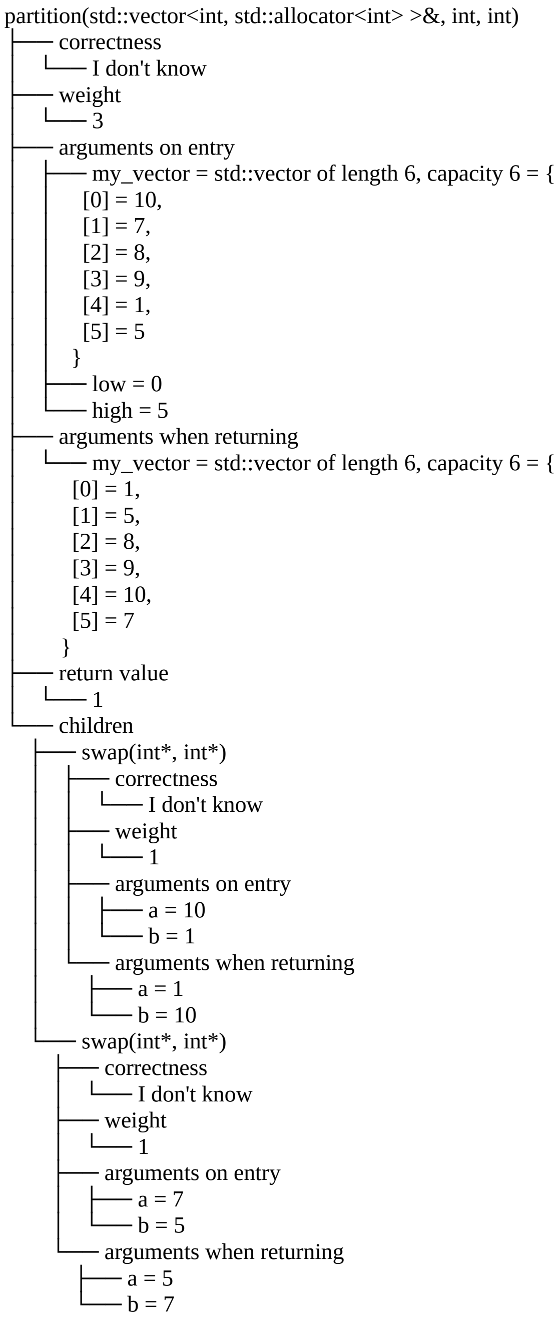
a decir que to-
us hijos son
ctos. Que no
a es un caso
cular de lo an-
r.

ntiendo esta
ción, creo que
ha introducido

re?

sitas definir
de tal manera
eas que se cor-
nde con un

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



```
partition(std::vector<int, std::allocator<int> >&, int, int)
├── correctness
│   └── I don't know
├── weight
│   └── 3
├── 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] = 1,
│       [1] = 5,
│       [2] = 8,
│       [3] = 9,
│       [4] = 10,
│       [5] = 7
│      }
├── return value
│   └── 1
└── children
    ├── swap(int*, int*)
    │   ├── correctness
    │   │   └── I don't know
    │   ├── weight
    │   │   └── 1
    │   ├── arguments on entry
    │   │   ├── a = 10
    │   │   └── b = 1
    │   └── arguments when returning
    │       ├── a = 1
    │       └── b = 10
    └── swap(int*, int*)
        ├── correctness
        │   └── I don't know
        ├── weight
        │   └── 1
        ├── arguments on entry
        │   ├── a = 7
        │   └── b = 5
        └── arguments when returning
            ├── a = 5
            └── b = 7
```

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
```

- If S receives T as input and returns a node $n'$, then $n' \in N$, $F(n') = f'$, $II(f') = ii'$ and $ii' \neq n'$.
- If S receives T as input and returns $\perp$, then $\forall n' \in N$, $F(n') = f'$, $II(f') = ii'$ and $ii' = n'$.

*Proof.* See (4). □

Chapter 5

# Tool description

DDC consists of approximately 900 lines of mostly statically typed Python.

It gets loaded into rr through the source command, which itself can be put inside a `.gdbinit` file.

In the following section, we will go through two usage scenarios:

- Bug finding
- Using test cases as oracles to reduce tree size

Then, we will explain the implementation of the tool and the most important design decisions made.

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

## 5.1. Usage scenarios

### 5.1.1. Bug finding

The execution of the code in 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 5.2.

Listing 5.1: Code that results in unexpected output

```
#include <quicksort.h>

int main()
{
  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 root cause of the error, we first have to record and replay the buggy program with `rr`. This is shown in figure 5.3.

Listing 5.3: Compiling, recording and replaying quickSort

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

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`5.3.1 command. 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. Now we can choose the nav-

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*)
```

igation strategy. Then, as shown in figure 5.3, the debugger asks us about the correctness of the root node of the execution tree. This question does not have the option to choose navigation strategy, as opposed to all other correctness questions, where the options are 5.4.

Figure 5.2: Choosing the navigation strategy

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

The debugging session ends when the buggy node is found 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.

### 5.1.2. 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 figure 5.4.

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;
}
```

In this listing, we see that we test ten vectors composed of random numbers from 0 to 9. When executing these test cases, the return code is 0, so we know that our `quickSort` implementation at least works for some vectors.

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.

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`5.3.9 command. 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 5.6 in `rr`. Again, the results of the command `info breakpoints` have been simplified.

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 `til-the-end` (see section 5.3.10) 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 figure 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 5.3.11).

Now we are ready to send the correct nodes from $rr_{server}$ to $rr_{client}$. This is done with the command `send-correct-nodes` (see section 5.3.12). 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 (insert reference)), we see that the WMET has 8 nodes (see 5.13) instead of 9 (see 5.12). This is because a swap node has been removed, more specifically, the node in figure 5.14.

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. Implementation

### 5.2.1. Tree building

Tree building is the fundamental process in which all other steps rely on.

#### 5.2.1.1. Adding a node to the tree

#### 5.2.1.2. Finishing a node

#### 5.2.1.3. Finishing the tree building process

The tree is considered built one of the following happens:

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

### 5.2.2. Tree transformation

#### 5.2.2.1. Simplified tree compression

### 5.2.3. General debugging algorithm

### 5.2.4. Strategies

#### 5.2.4.1. Top-down

#### 5.2.4.2. Divide and Query (Shapiro)

#### 5.2.4.3. Heaviest first

### 5.2.5. User answers to correctness questions

#### 5.2.5.1. I don't know

#### 5.2.5.2. Yes

#### 5.2.5.3. No

#### 5.2.5.4. Trusted

### 5.2.6. Test cases as oracles

## 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.

### 5.3.1. 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 it. It should be used at least once before starting the debugging session.

Arguments: location.

Location completion is enabled.

### 5.3.2. 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.

### 5.3.3. save-returning-node

It saves the arguments 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.

### 5.3.4. 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.

### 5.3.5. 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.

### 5.3.6. 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.

### 5.3.7. 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.

### 5.3.8. 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.

### 5.3.9. save-returning-correct-node

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

Arguments: none.

For internal use only.

### 5.3.10. til-the-end

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

Optional command used in gathering correct nodes.

Arguments: none.

### 5.3.11. 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.

### 5.3.12. 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.

### 5.3.13. 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.

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
```

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
>>>
```

Figure 5.8: Length lists

```
>>> 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
```

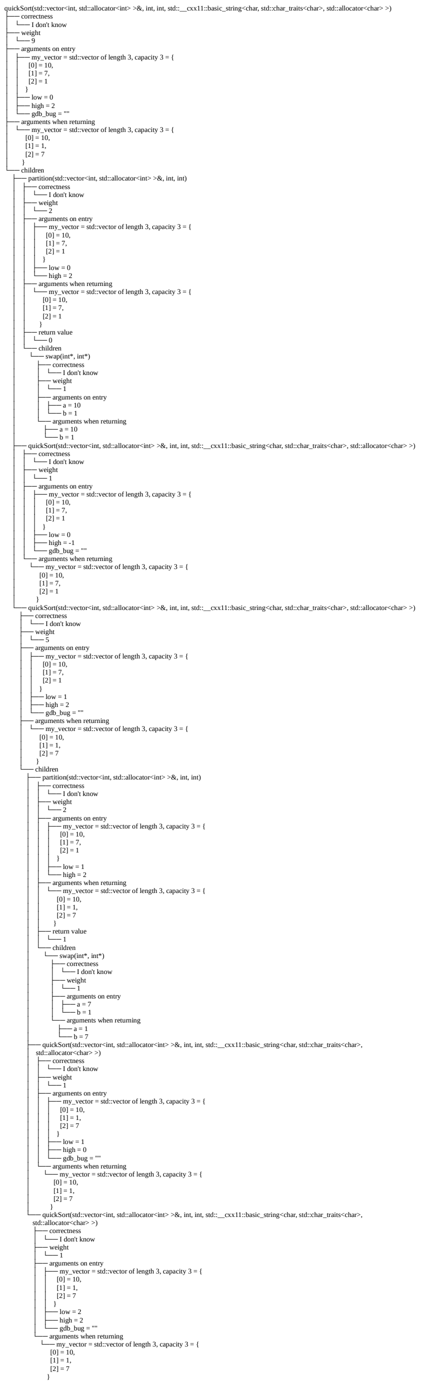
Figure 5.10: Sending nodes to client

```
    Sending node n# 87
```

Figure 5.11: Receiving nodes from server

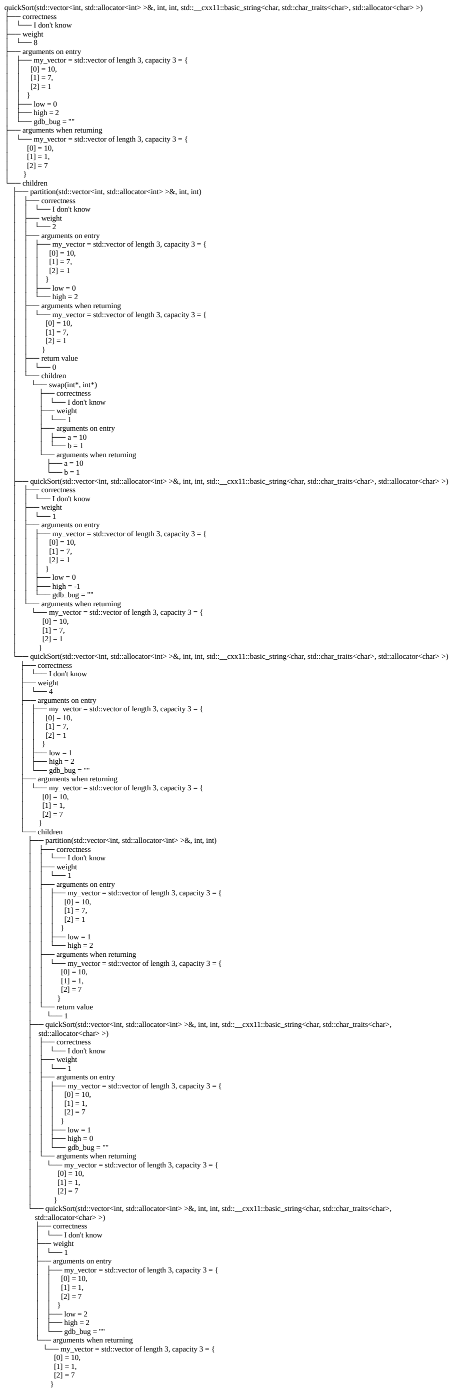
```
    Connected by ('127.0.0.1', 51126)
```

Figure 5.12: Tree of buggy quickSort before correct node elimination



```
quickSort(std::vector<int, std::allocator<int> >&, int, int, std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >)
├── correctness
│   └── I don't know
├── weight
│   └── 9
├── arguments on entry
│   ├── my_vector = std::vector of length 3, capacity 3 = {
│   │  [0] = 10,
│   │  [1] = 7,
│   │  [2] = 1
│   │ }
│   ├── low = 0
│   ├── high = 2
│   └── gdb_bug = ""
├── arguments when returning
│   └── my_vector = std::vector of length 3, capacity 3 = {
│      [0] = 10,
│      [1] = 1,
│      [2] = 7
│     }
└── children
    ├── partition(std::vector<int, std::allocator<int> >&, int, int)
    │   ├── correctness
    │   │   └── I don't know
    │   ├── weight
    │   │   └── 2
    │   ├── arguments on entry
    │   │   ├── my_vector = std::vector of length 3, capacity 3 = {
    │   │   │  [0] = 10,
    │   │   │  [1] = 7,
    │   │   │  [2] = 1
    │   │   │ }
    │   │   ├── low = 0
    │   │   └── high = 2
    │   ├── arguments when returning
    │   │   └── my_vector = std::vector of length 3, capacity 3 = {
    │   │      [0] = 10,
    │   │      [1] = 7,
    │   │      [2] = 1
    │   │     }
    │   ├── return value
    │   │   └── 0
    │   └── children
    │       └── swap(int*, int*)
    │           ├── correctness
    │           │   └── I don't know
    │           ├── weight
    │           │   └── 1
    │           ├── arguments on entry
    │           │   ├── a = 10
    │           │   └── b = 1
    │           └── arguments when returning
    │               ├── a = 10
    │               └── b = 1
    ├── quickSort(std::vector<int, std::allocator<int> >&, int, int, std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >)
    │   ├── correctness
    │   │   └── I don't know
    │   ├── weight
    │   │   └── 1
    │   ├── arguments on entry
    │   │   ├── my_vector = std::vector of length 3, capacity 3 = {
    │   │   │  [0] = 10,
    │   │   │  [1] = 7,
    │   │   │  [2] = 1
    │   │   │ }
    │   │   ├── low = 0
    │   │   ├── high = -1
    │   │   └── gdb_bug = ""
    │   └── arguments when returning
    │       └── my_vector = std::vector of length 3, capacity 3 = {
    │          [0] = 10,
    │          [1] = 7,
    │          [2] = 1
    │         }
    └── quickSort(std::vector<int, std::allocator<int> >&, int, int, std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >)
        ├── correctness
        │   └── I don't know
        ├── weight
        │   └── 5
        ├── arguments on entry
        │   ├── my_vector = std::vector of length 3, capacity 3 = {
        │   │  [0] = 10,
        │   │  [1] = 7,
        │   │  [2] = 1
        │   │ }
        │   ├── low = 1
        │   ├── high = 2
        │   └── gdb_bug = ""
        ├── arguments when returning
        │   └── my_vector = std::vector of length 3, capacity 3 = {
        │      [0] = 10,
        │      [1] = 1,
        │      [2] = 7
        │     }
        └── children
            ├── partition(std::vector<int, std::allocator<int> >&, int, int)
            │   ├── correctness
            │   │   └── I don't know
            │   ├── weight
            │   │   └── 2
            │   ├── arguments on entry
            │   │   ├── my_vector = std::vector of length 3, capacity 3 = {
            │   │   │  [0] = 10,
            │   │   │  [1] = 7,
            │   │   │  [2] = 1
            │   │   │ }
            │   │   ├── low = 1
            │   │   └── high = 2
            │   ├── arguments when returning
            │   │   └── my_vector = std::vector of length 3, capacity 3 = {
            │   │      [0] = 10,
            │   │      [1] = 1,
            │   │      [2] = 7
            │   │     }
            │   ├── return value
            │   │   └── 1
            │   └── children
            │       └── swap(int*, int*)
            │           ├── correctness
            │           │   └── I don't know
            │           ├── weight
            │           │   └── 1
            │           ├── arguments on entry
            │           │   ├── a = 7
            │           │   └── b = 1
            │           └── arguments when returning
            │               ├── a = 1
            │               └── b = 7
            ├── quickSort(std::vector<int, std::allocator<int> >&, int, int, std::__cxx11::basic_string<char, std::char_traits<char>,
            │ std::allocator<char> >)
            │   ├── correctness
            │   │   └── I don't know
            │   ├── weight
            │   │   └── 1
            │   ├── arguments on entry
            │   │   ├── my_vector = std::vector of length 3, capacity 3 = {
            │   │   │  [0] = 10,
            │   │   │  [1] = 1,
            │   │   │  [2] = 7
            │   │   │ }
            │   │   ├── low = 1
            │   │   ├── high = 0
            │   │   └── gdb_bug = ""
            │   └── arguments when returning
            │       └── my_vector = std::vector of length 3, capacity 3 = {
            │          [0] = 10,
            │          [1] = 1,
            │          [2] = 7
            │         }
            └── quickSort(std::vector<int, std::allocator<int> >&, int, int, std::__cxx11::basic_string<char, std::char_traits<char>,
              std::allocator<char> >)
                ├── correctness
                │   └── I don't know
                ├── weight
                │   └── 1
                ├── arguments on entry
                │   ├── my_vector = std::vector of length 3, capacity 3 = {
                │   │  [0] = 10,
                │   │  [1] = 1,
                │   │  [2] = 7
                │   │ }
                │   ├── low = 2
                │   ├── high = 2
                │   └── gdb_bug = ""
                └── arguments when returning
                    └── my_vector = std::vector of length 3, capacity 3 = {
                       [0] = 10,
                       [1] = 1,
                       [2] = 7
                      }
```

Figure 5.13: Tree of buggy quickSort after correct node elimination



```
quickSort(std::vector<int, std::allocator<int> >&, int, int, std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >)
├── correctness
│   └── I don't know
├── weight
│   └── 8
├── arguments on entry
│   ├── my_vector = std::vector of length 3, capacity 3 = {
│   │    [0] = 10,
│   │    [1] = 7,
│   │    [2] = 1
│   │   }
│   ├── low = 0
│   ├── high = 2
│   └── gdb_bug = ""
├── arguments when returning
│   └── my_vector = std::vector of length 3, capacity 3 = {
│        [0] = 10,
│        [1] = 1,
│        [2] = 7
│       }
└── children
    ├── partition(std::vector<int, std::allocator<int> >&, int, int)
    │   ├── correctness
    │   │   └── I don't know
    │   ├── weight
    │   │   └── 2
    │   ├── arguments on entry
    │   │   ├── my_vector = std::vector of length 3, capacity 3 = {
    │   │   │    [0] = 10,
    │   │   │    [1] = 7,
    │   │   │    [2] = 1
    │   │   │   }
    │   │   ├── low = 0
    │   │   └── high = 2
    │   ├── arguments when returning
    │   │   └── my_vector = std::vector of length 3, capacity 3 = {
    │   │        [0] = 10,
    │   │        [1] = 7,
    │   │        [2] = 1
    │   │       }
    │   ├── return value
    │   │   └── 0
    │   └── children
    │       └── swap(int*, int*)
    │           ├── correctness
    │           │   └── I don't know
    │           ├── weight
    │           │   └── 1
    │           ├── arguments on entry
    │           │   ├── a = 10
    │           │   └── b = 1
    │           └── arguments when returning
    │               ├── a = 10
    │               └── b = 1
    ├── quickSort(std::vector<int, std::allocator<int> >&, int, int, std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >)
    │   ├── correctness
    │   │   └── I don't know
    │   ├── weight
    │   │   └── 1
    │   ├── arguments on entry
    │   │   ├── my_vector = std::vector of length 3, capacity 3 = {
    │   │   │    [0] = 10,
    │   │   │    [1] = 7,
    │   │   │    [2] = 1
    │   │   │   }
    │   │   ├── low = 0
    │   │   ├── high = -1
    │   │   └── gdb_bug = ""
    │   └── arguments when returning
    │       └── my_vector = std::vector of length 3, capacity 3 = {
    │            [0] = 10,
    │            [1] = 7,
    │            [2] = 1
    │           }
    └── quickSort(std::vector<int, std::allocator<int> >&, int, int, std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >)
        ├── correctness
        │   └── I don't know
        ├── weight
        │   └── 4
        ├── arguments on entry
        │   ├── my_vector = std::vector of length 3, capacity 3 = {
        │   │    [0] = 10,
        │   │    [1] = 7,
        │   │    [2] = 1
        │   │   }
        │   ├── low = 1
        │   ├── high = 2
        │   └── gdb_bug = ""
        ├── arguments when returning
        │   └── my_vector = std::vector of length 3, capacity 3 = {
        │        [0] = 10,
        │        [1] = 1,
        │        [2] = 7
        │       }
        └── children
            ├── 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 3, capacity 3 = {
            │   │   │    [0] = 10,
            │   │   │    [1] = 7,
            │   │   │    [2] = 1
            │   │   │   }
            │   │   ├── low = 1
            │   │   └── high = 2
            │   ├── arguments when returning
            │   │   └── my_vector = std::vector of length 3, capacity 3 = {
            │   │        [0] = 10,
            │   │        [1] = 1,
            │   │        [2] = 7
            │   │       }
            │   └── return value
            │       └── 1
            ├── quickSort(std::vector<int, std::allocator<int> >&, int, int, std::__cxx11::basic_string<char, std::char_traits<char>,
            │  std::allocator<char> >)
            │   ├── correctness
            │   │   └── I don't know
            │   ├── weight
            │   │   └── 1
            │   ├── arguments on entry
            │   │   ├── my_vector = std::vector of length 3, capacity 3 = {
            │   │   │    [0] = 10,
            │   │   │    [1] = 1,
            │   │   │    [2] = 7
            │   │   │   }
            │   │   ├── low = 1
            │   │   ├── high = 0
            │   │   └── gdb_bug = ""
            │   └── arguments when returning
            │       └── my_vector = std::vector of length 3, capacity 3 = {
            │            [0] = 10,
            │            [1] = 1,
            │            [2] = 7
            │           }
            └── quickSort(std::vector<int, std::allocator<int> >&, int, int, std::__cxx11::basic_string<char, std::char_traits<char>,
               std::allocator<char> >)
                ├── correctness
                │   └── I don't know
                ├── weight
                │   └── 1
                ├── arguments on entry
                │   ├── my_vector = std::vector of length 3, capacity 3 = {
                │   │    [0] = 10,
                │   │    [1] = 1,
                │   │    [2] = 7
                │   │   }
                │   ├── low = 2
                │   ├── high = 2
                │   └── gdb_bug = ""
                └── arguments when returning
                    └── my_vector = std::vector of length 3, capacity 3 = {
                         [0] = 10,
                         [1] = 1,
                         [2] = 7
                        }
```

Figure 5.14: Correct node removed from execution tree

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

Chapter 6

# Conclusions and Future Work

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

## 6.1. 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 6.1, 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 a intuitive command line interface, avoiding a context switch, which can be as disruptive as an interruption (5). 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 (6) by

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

.

We also consider Goal 2 (no changes to program) as achieved. No code instrumentation, such as `# 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. These tree transformation reduces the tree size of recursive programs, therefore reducing the amount of questions the user has to answer. Also, we have build the necessary infrastructure to integrate other tree transformation painlessly in the future.

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

Figure 6.1: 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 (7), needed to gather arguments when returning, 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 (8), 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.

## 6.2. Future work

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

### 6.2.1. 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 execution tree. Ideally, DDC would be able to build the ET without any suspect breakpoints, by creating a node every time it enters a new function. Since we already have functionality to prune the ET of excess nodes (by trusting functions and using test cases as oracles, for example), the user could them reduce the size of the ET.

### 6.2.2. Benchmarking overhead of building execution tree

As mentioned in (6), the speed and memory footprint of building the debugging tree is one of the most important metrics in the development of algorithmic debuggers. We should collect a sample of real, large C++ programs and measure the time and memory needed to build their respective execution trees.

### 6.2.3. Formally verify all algorithms

We made an effort to formally verify all algorithms used in DDC. In Nagini (9), the tool we selected to perform the verification, we identified a bug[2], which was confirmed by Nagini main author, that blocked further work. Another option worth considering would be to develop the algorithms in a programming language such as Lean (10), which is also a theorem prover in which you can prove programs correct. Lean can generate code in C, which could be call from Python.

### 6.2.4. Support for C-style arrays

C++ arrays are inherited from C to maintain backward compatibility. As illustrated by figure 6.2, 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 examples programs provided with DDC use std::vector instead of C-style arrays.

Figure 6.2: 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 *
```

### 6.2.5. Test programming languages other than C++

The Rust code in listing 6.1 produces the node represented in figure 6.3.

Listing 6.1: QuickSort in Rust

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

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 6.3: quicksort tree

```
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 following function returns a non empty list:

Listing 6.2: Python function

```python
def get_pointer_or_ref(arguments, frame):
    if arguments is None:
        return []
    return [argument for argument in arguments
            if argument.value(frame).type.code in [gdb.TYPE_CODE_PTR,
                                                    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.

### 6.2.6. 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 (11). We could remove the rr dependency to support concurrency, but reverse execution in GDB (7) is more limited than in rr.

### 6.2.7. Implement more strategies

As mentioned in the survey (6), several navigation strategies have been developed. Of special interest is Optimal Divide and Query (2), which has been proven correct and optimal. One difficulty we may face when developing this strategy is that another weight marking has to be created for the debugging tree.

### 6.2.8. 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 create, 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 (12) may be needed.
- Testing private methods require creating a friend class to access them. Another option would be to skip them altogether.

# Bibliography

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

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

[3] Rafael Caballero, Enrique Martin-martin, and Salvador Tamarit. A declarative debugger for sequential erlang programs, 2013.

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

[5] Zahra Shakeri Hossein Abad, Mohammad Noaeen, Didar Zowghi, Behrouz H. Far, and Ken Barker. Two sides of the same coin: Software developers' perceptions of task switching and task interruption, 2018.

[6] Rafael Caballero, Adrián Riesco, and Josep Silva. A survey of algorithmic debugging. *ACM Comput. Surv.*, 50(4), aug 2017.

[7] Inc. Free Software Foundation. Running programs backward. `https://sourceware.org/gdb/onlinedocs/gdb/Reverse-Execution.html`. Accessed: 2022-01-06.

[8] Inc. Free Software Foundation. Supported languages. `https://sourceware.org/gdb/current/onlinedocs/gdb/Supported-Languages.html`. Accessed: 2022-01-06.

[9] Marco Eilers and Peter Müller. Nagini: A static verifier for python. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, pages 596–603, Cham, 2018. Springer International Publishing.

[10] Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction – CADE 28*, pages 625–635, Cham, 2021. Springer International Publishing.

[11] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. Engineering record and replay for deployability: Extended technical report, 2017.

[12] Chris Arthur Lattner. Llvm: An infrastructure for multi-stage optimization. Technical report, 2002.

Este texto se puede encontrar en el fichero Cascaras/fin.tex. Si deseas eliminarlo, basta con comentar la línea correspondiente al final del fichero TFMTeXiS.tex.

*–¿Qué te parece desto, Sancho? – Dijo Don Quijote –*
*Bien podrán los encantadores quitarme la ventura,*
*pero el esfuerzo y el ánimo, será imposible.*

*Segunda parte del Ingenioso Caballero*
*Don Quijote de la Mancha*
*Miguel de Cervantes*

*–Buena está – dijo Sancho –; fírmela vuestra merced.*
*–No es menester firmarla – dijo Don Quijote–,*
*sino solamente poner mi rúbrica.*

*Primera parte del Ingenioso Caballero*
*Don Quijote de la Mancha*
*Miguel de Cervantes*