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

Trabajo de Fin de Máster
Curso 2020–2021

Autor
Roland Coeurjoly Lechuga

Director
Adrián Riesco Rodríguez

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

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

Trabajo de Fin de Máster en Métodos Formales en Ingeniería Informática

Autor
Roland Coeurjoly Lechuga

Director
Adrián Riesco Rodríguez

Convocatoria: *Febrero/Junio/Septiembre 2021*
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

Agradecimientos

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++

Índice

Índice de figuras

Índice de tablas

Capítulo 1

Introduction

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

1.1. Motivation

Debugging is the most expensive part of developing software. It is estimated that between 80 and 90 percent of development effort is spent on debugging tasks. (reference?)

Furthermore, no paradigm (imperative, functional, logical, etc) or language can claim that it removes the need to debug.

Therefore, it is important to develop tools and workflows to alleviate this issue.

Declarative debugging (also known as algorithmic debugging) takes a semiautomatic approach.

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

There are declarative debuggers for:

- Maude
- Java
- Erlang

Developing a declarative debugger for C++ involves figuring out a way to treat pointers in a way useful for the user.

1.2. Goals

The goal of this thesis is to develop a declarative debugger for the C++ language.

This debugger, called DDC, should have the following features:

- The debugger has to scale to real programs.
- Integrated in the existing debugging workflow of the developer.
- The user has to be able to debug a program with no or few changes to its compilation (at most setting some compilation flags).

scalability issue:

Provide a way for the user to choose what are the suspect functions or methods. This would reduce the number of nodes in the execution tree (ET), therefore reducing the time to build it and its memory footprint. Provide a way for the user to choose a point in the code that, if reached, triggers the end of the building of the ET and begins the asking questions to the user.

Use test cases to reduce the number of nodes in the tree, therefore reducing the number of questions needed to find the buggy node.

Usability:

Integrate the declarative debugger into the debugger most used by C++ developers.

This would provide several benefits:

The user would set the breakpoints (both the suspect function or methods and the final point) in a way that is identical to her usual debugging workflow.

Common features when setting breakpoints like auto-completion

The user would not have to switch tools between normal debugging and declarative debugging.

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 to reduce the tree size
- 3 asking strategies developed
- 1 tree transformation developed
- easily extensible (more strategies, more tree transformations)

Capítulo 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.

Then, the DD builds an execution tree of the execution of this program.

2.1.1. Building the tree

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

2.1.1.1. Strategies for reducing the size of the tree

Since the number of questions depends on the size of the tree (number of nodes, width and depth of the tree), most DD have the option of either trusting some computations before hand, to avoid adding to the tree, or focusing on some computations and only adding those to the tree.

trusting computations

Trusting all computations but some suspect computations

using test cases to eliminate nodes

2.2. C++ programming language

The C++ programming language is a general-purpose, statically typed, compiled language.

In C++, a method (or member function) is function that is defined inside a class. It has, therefore, access to all members of that class.

DDC can build an execution tree composed of function

2.3. Technologies used

2.3.1. GDB: The GNU Project Debugger

2.3.2. rr: Record and Replay Framework

2.3.3. Poetry: Dependency Management for Python

2.3.4. Nix: the purely functional package manager

Capítulo 3

State of the art

3.1.

En el estado de la cuestión es donde aparecen gran parte de las referencias bibliográficas del trabajo. Una de las formas más cómodas de gestionar la bibliografía en LaTeX es utilizando **bibtex**. Las entradas bibliográficas deben estar en un fichero con extensión *.bib* (con esta plantilla se proporciona el fichero biblio.bib, donde están las entradas referenciadas más abajo). Cada entrada bibliográfica tiene una clave que permite referenciarla desde cualquier parte del texto con los siguiente comandos:

- Referencia bibliografica con cite: Bautista et al. (1998)
- Referencia bibliográfica con citep: (Oetiker et al., 1996)
- Referencia bibliográfica con citet: Krishnan (2003)

Es posible citar más de una fuente, como por ejemplo (Mittelbach et al., 2004; Lamport, 1994; Knuth, 1986)

Después, latex se ocupa de rellenar la sección de bibliografía con las entradas **que hayan sido citadas** (es decir, no con todas las entradas que hay en el .bib, sino sólo con aquellas que se hayan citado en alguna parte del texto).

Bibtex es un programa separado de latex, pdflatex o cualquier otra cosa que se use para compilar los .tex, de manera que para que se rellene correctamente la sección de bibliografía es necesario compilar primero el trabajo (a veces es necesario compilarlo dos veces), compilar después con bibtex, y volver a compilar otra vez el trabajo (de nuevo, puede ser necesario compilarlo dos veces).

Col1	Col2	Col2	Col3
1	6	87837	787
2	7	78	5415
3	545	778	7507
4	545	18744	7560
5	88	788	6344

Capítulo 4

Definitions

In this chapter we will provide definitions for the key ideas needed to approach the building of the tool.

No empieces la ción con una su sección, pon alg texto explicand qué va a ir.

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

Usar entorno de finition o algo a Cuando una def nición salga de artículo pon la en el nombre de definición (esto para las siguien

- *f is the name of the function/method executed.*
- *I is the set of inputs to f, $I = \{I_a, I_o, I_g\}$, where:*
  - *I_a is the set of input arguments when it was called (if any).*
  - *I_o is the object state when it was called (if it is a method call).*
  - *I_g is the set global variables when it was called (if any).*
- *O is the set of outputs to f, $O = \{O_a, O_o, O_g\}$, where:*
  - *O_a is the set of output arguments when it returned (if there where passed as reference or pointer).*
  - *O_o is the object state when it returned (if it is a method call).*
  - *O_g is the set of global variables when it returned (if any).*

Definition 2 (Edge)**.** *An edge is hierarchical relationship between two nodes, a parent node and a child node. A node can have 0 or more children nodes. A node can have 0 or 1 parent nodes. A node that has no edges leading to children nodes is a leaf node. A node that has no edge leading to its parent node is the root node of a tree. Nodes that share the same parent node are siblings.*

Aristas?

We here expand the definition of MET (Insa y Silva, 2011) to WMET.

Las comillas en tex son “asi”

Definition 3 (Weighted Marked Execution Tree)**.** *A weighted marked execution tree (WMET) is a tree $T = (N, E, W, M)$where N are the nodes, $E \subseteq N \times N$ are the 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 is defined recursively as its number of descendants including itself (i.e.,$1 + \sum w_{n'} \mid n \to n') \in E$).*

falta referencia

Usar entorno m temático y los c mandos corresp dientes en todas fórmulas

Definition 4 (Intended interpretation)**.** *The intended interpretation (II) of a function/-method f given inputs I is (adapted from Caballero et al.) $II = f(I) \to O$*

Definition 5 (Buggy node)**.** *A buggy node is a node that is not equal to its intended interpretation.*

A buggy node is marked as incorrect by the user.

At first we considered inferring that the root node of the tree, upon completing of the building of the tree, is wrong, since the user started a debugging session.

Upon closer analysis we decided against doing so, to provide more flexibility to the user.

es confuso,
ue DDC detec-
solo tipo de
, porque so-
y un tipo de
buggy. Otra
es que pueda
uggy por todas
razones, pero
stinguen en la
mación dada al
rio?

Definition 6 (Detected errors)**.** *DDC can help the user detect the following errors:*

- *A function/method was called with the wrong arguments*
- *A function/method returns a wrong value*
- *A global variable is visible from the function/method scope when it should not*
- *A global variable is not visible from the function/method scope when it should*
- *A method modifies its object when it should not*
- *A method does not modify its object when it should do so.*
- *A function/method modifies an argument passed by reference or pointer when it should not*
- *A function/method does not modify an argument passed by reference or pointer when it should do so*
- *An argument passed by reference or pointer has the wrong value when returning.*
- *A function/method does not terminate when it should do so*
- *A function/method calls a sub-computation when it should not.*
- *A function/method does not call a sub-computation when it should do so.*

Definition 7 (Completion of the debugging session)**.** *The debugging session is finished when: The WMET is empty, therefore no buggy node has been found The WMET consists of one node marked wrong, therefore the buggy node has been found.*

Definition 8 (Correctness)**.**

Capítulo 5

Tool description

DDC consists of approximately 900 lines of statically typed Python.
It gets loaded into rr through the source command.

5.1. Implementation

5.1.1. Tree building

5.1.1.1. Adding a node to the tree

5.1.1.2. Finishing a node

5.1.1.3. 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

5.1.2. Tree transformation

5.1.2.1. Simplified tree compression

5.1.3. General debugging algorithm

5.1.4. Strategies

5.1.4.1. Top-down

5.1.4.2. Divide and Query (Shapiro)

5.1.4.3. Heaviest first

5.1.5. User answers to correctness questions

5.1.5.1. I don't know

5.1.5.2. Yes

5.1.5.3. No

5.1.5.4. Trusted

5.1.6. Test cases as oracles

5.2. Commands

5.2.1. suspect-function

gdb.COMPLETE LOCATION

5.2.2. add-node-to-session

5.2.3. save-returning-node

5.2.4. final-point

gdb.COMPLETE LOCATION

5.2.5. finish-debugging-session

5.2.6. start-declarative-debugging-session

5.2.7. save-correct-function

gdb.COMPLETE LOCATION

5.2.8. save-returning-correct-node

5.2.9. add-node-to-correct-list

5.2.10. til-the-end

5.2.11. listen-for-correct-nodes

5.2.12. send-correct-nodes

5.2.13. print-nodes

Capítulo 6

Conclusions and Future Work

Conclusions.

By using GDB as a framework to build the declarative debugger, we have the benefit of supporting not only C++ but several programming languages (https://sourceware.org/gdb/current/onlinedoc Languages.html).

This is done through a common Python API.

Also, by using a language with a broad amount of libraries such as Python, the user interface and execution tree representation have been easy to implement.

During the development and use of the debugger, some execution trees did not match expectations.

Upon close examination, these maybe caused by GDB.

The first is that with recursive calls

The second is that if two function calls are identical in terms of arguments passed to them, then their frames are identical.

The workaround implemented to avoid building an erroneous execution tree is two fold:

When filling information about the call on entry, the current node is appended to the list of node if: nodes is empty last node's frame is invalid (meaning is no longer active) last node's frame is not a parent of current node.

When filling information about the call on return, only fill it if it is empty.

Another future line of improvements could come from the presentation of node information to the user.

Two issues have been highlighted by doing this project.

The first is arrays in C++. C++ arrays are typed as pointers to the first element of the array when passed as argument to a function. (insert screen capture of GDB to prove this)

Then, it does not seem to be a way to differentiate between pointers and arrays when working in GDB.

To surmount this difficulty, the examples programs use std::vector instead of C-style arrays.

Another issue with variable display is pointers.

Some functions, like swap(int *, int *) in the example provided, do pointer arithmetic.

Maybe we should display both the memory address pointed by the pointer and the content.

Another line of future development could be

6.1. Conclusions

6.2. Future work

6.2.1. Benchmarking overhead of building execution tree

6.2.2. Support concurrent programs

6.2.3. Test programming languages other than C++

6.2.4. Implement more strategies

6.2.5. Formally verify all algorithms

Nagini fails Cross hair fails (recursion too deep) Maybe Lean to C, and the C to Python

6.2.6. Generate test cases from correct nodes

6.2.7. Support for C-style arrays

6.2.8. Report GDB bugs

6.2.8.1. Wrong backtrace with recursive functions

6.2.8.2. Frame ID is the same for function call with the same arguments

6.2.9. Increase granularity in error detection

6.2.10. Increase flexibility inside a debugging session

6.2.11. Interactive/collapsible execution tree

Bibliografía

Y así, del mucho leer y del poco dormir, se le secó el celebro de manera que vino a perder el juicio.
(modificar en Cascaras\bibliografia.tex)

Miguel de Cervantes Saavedra

Bautista, T., Oetiker, T., Partl, H., Hyna, I. y Schlegl, E. *Una Descripción de LaTeX 2ε*. Versión electrónica, 1998.

Caballero, R., Martin-martin, E. y Tamarit, S. A declarative debugger for sequential erlang programs. ????

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

Knuth, D. E. *The TeX book*. Addison-Wesley Professional., 1986.

Krishnan, E., editor. *LaTeX Tutorials. A primer*. Indian TeX Users Group, 2003.

Lamport, L. *LaTeX: A Document Preparation System, 2nd Edition*. Addison-Wesley Professional, 1994.

Mittelbach, F., Goossens, M., Braams, J., Carlisle, D. y Rowley, C. *The LaTeX Companion*. Addison-Wesley Professional, segunda edición, 2004.

Oetiker, T., Partl, H., Hyna, I. y Schlegl, E. *The Not So Short Introduction to LaTeX 2ε*. Versión electrónica, 1996.

Apéndice A

Título del Apéndice A

Contenido del apéndice

Apéndice B

Título del Apéndice B

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