Unsat Cores for Alloy

Martin Tippmann, 60672
Carlos Poveda, 123802
Bauhaus-Universität Weimar

Formal Methods for Software Engineering

Abstract

Alloy Analyzer, employs unsatisfiable cores to reveal inconsistencies within system specifications. This work focuses on detect these cores when no instance satisfies a given model. The unsatisfiable core, a set of conflicting formulas, serves as a tool for diagnosing potential specification errors. By exploring case studies and practical scenarios, we demonstrate the analyzer efficacy in refining model accuracy, offering insights in system design and validation.

1 Introduction

1.1 Background

1.1.1 Alloy Analyzer

Alloy is tool suite for modeling software, widely recognized for its effectiveness across various domains, including software engineering or networking designs. It's built on the Kodkod model-finding engine, which gives a wide range of tools for analyzing and developing software models. Alloy offers comprehensive language documentation, accessible software downloads, and a broad selection of case studies, reflecting its growing influence in Software Engineering, especially in the areas of design and architecture.

1.1.2 Delta Debugging

Delta debugging is an algorithm designed to identify and minimize conditions necessary to reproduce a bug. This method is adaptable, suitable for isolating failure-inducing factors such as program inputs, user interactions e.g., keystrokes, button presses, mouse movements, and changes in program code including variable, function, class additions, or deletions.

Delta debugging key terms:

- **Configuration:** A set of source code changes between the last known working version and the version that shows the regression.
- **Test:** A procedure to determine whether a configuration induces a failure, Success, or produces inconclusive results.
- **Subset and Superset:** For two sets A and B, A is a subset of B if all elements of A are in B. B is a superset of A if it contains all elements of A.
- **Union and Intersection:** The union of sets A and B includes all elements in either A, B, or both. The intersection of A and B contains elements common to both.

When learning about the delta debugging algorithm, it's important to remember that it might have some tricky parts to understand:

- **Interference:** Sometimes, individual changes in isolation might not lead to a regression, but when combined, they trigger it. It's crucial, therefore, to have a strategy to pinpoint the exact combination of changes responsible for the regression. In cases of interference, it's necessary to recursively test each half of the configuration, keeping the changes in one half consistently applied.
- **Inconsistency:** There might be dependencies among the elements in the subsets, which could prevent successful evaluation and testing if not all dependent changes are included.

Inconsistent configuration can be classified as:

  - **Integration Failure:** This occurs when a change cannot be applied to the source code. This might be due to the change requiring other changes that are not included in the configuration or being in conflict with another change, where a resolving change is missing.
  - **Construction Failure:** This type of failure arises when, despite all changes being applied, the resulting program contains syntactic or semantic errors, preventing it from being successfully constructed.
  - **Execution Failure:** This happens when the program, even after successful construction, does not execute correctly. The outcome of the test in such cases is unresolved, indicating a failure in the program's execution.

- **Preference:** If testing any subset A the result is unresolved, and testing its complement B passes, then A contains a failure-induced subset and is preferred. In the following test cases, B must remain applied to promote consistency.
- **Monotony:** If a change leads to failure, then any configuration including this change will also fail. Conversely, if a configuration doesn't cause failure, its subsets won't either, allowing their exclusion from the set of potential causes. However, considering interference, a non-failing subset combined with other changes might still contribute to a regression.
- **Unambiguity:** In some scenarios, a failure is caused by a single, specific configuration without interference. This implies that searching the alternative configurations might not be necessary for efficiency. However, for a comprehensive understanding and to identify all failure-inducing changes, examining both configurations is recommended."
- **Granularity:** A single subset might involve numerous elements, yet only a small part of it could be causing the regression. Therefore, there needs to be a way to break down these changes into smaller, more manageable segments.

1.2 Motivation

Incorporating the DDmin algorithm into the analysis of Alloy models serves as the main of our project's motivation. The central challenge we address is the detection of minimal unsatisfiable subsets (MUSes). MUSes, which highlight incompatible constraints within Alloy specifications, pose significant problem in the model validation process. The adoption of DDmin, known for its systematic reduction and precision, enables us to isolate these problematic subsets effectively and efficiently.
Our project aims to refine the debugging process by integrating DDmin, offering a targeted solution to dissect and understand constraint conflicts in Alloy models. This integration is motivated by the need for a more optimized and accurate approach to diagnosing unsatisfiability, which are often complex and time-consuming to resolve using conventional methods. By focusing on the precise identification of MUSes, we anticipate a marked improvement in the analysis workflow, facilitating quicker turnaround times and deeper insights into the structural intricacies of Alloy models.
This reason emphasizes our interest in enhancing the field of formal specification analysis through the use of algorithms like DDmin. Our goal is improve the reliability and effectiveness of Alloy model debugging. Through this project, we seek to provide tool for addressing unsatisfiability, elevating the overall quality and coherence of Alloy models.

2 Problem Statement

Unsatisfiable cores in Alloy are the smallest sets within modules that cannot be satisfied; removing even one element from these cores makes them satisfiable. Alloy calculates these cores using solvers like MiniSAT with core support. Delta Debugging (DDMin) is an algorithm designed to isolate a minimal set of changes or elements responsible for causing a specific problem. Our project involves implementing core calculation in Alloy using DDMin and comparing this method to Alloy native core calculation approach. This comparison aims to explore the efficiency and effectiveness of DDMin in identifying unsatisfiable cores in Alloy modules.

2.1 Problem Example

```
abstract sig House {
    major: one Major,
    color: one Color,
}
one sig H1, H2, H3 extends House {}
enum Major { Math, Phil, CS }
enum Color { Blue, Red, White }

fact {
    Major in House.major
    Color in House.color
    H2.major = Phil
    (H1.color = Red and H2.major = Phil) or ((H2.color = Red and H3.major = Phil))
    some h : House | h.color = Blue and h.major = CS
    H2.major = Math
}
```

Listing 1: Alloy Model

```
fact {
    H2.major = Phil
    H2.major = Math
}
```

Listing 2: UNSAT Core

The model becomes unsatisfiable due to the lines H2.major = Phil and H2.major = Math in a fact. This indeed creates an unsatisfiable condition because it asserts that H2.major is both Phil and Math simultaneously, which is logically impossible.
To determine if this unsatisfiable core is a "Minimum Unsatisfiable Core," let's recall the definitions:

- **Minimum Unsatisfiable Core:** The smallest subset (in terms of the number of clauses) of a formula that is unsatisfiable, and removing even one element from these core makes all model satisfiable (SAT).

- **Unsatisfiable Core:** Any subset of a formula that is unsatisfiable.

Removing even one of these two lines makes the model satisfiable, they do form an unsatisfiable core. To assess whether it's a minimum unsatisfiable core, We would need to consider if there is any smaller subset of clauses that also makes the model unsatisfiable. If no smaller unsatisfiable subset exists, then this core is indeed the minimum (MUS).

3 Implementation

The project uses the object oriented programming capabilities of Java in combination with the Alloy Tools library to implement a delta debugging algorithm for Alloy models. The application is divided into several packages each of which performs a different function in the debugging process.

3.1 Key components and their functionalities

3.1.1 AlloyManager class from package alloy:

- Serves as the main entry point of the application.
- Handles command-line arguments to determine the operation mode, such as analyzing facts or predicates in Alloy models.
- Validates and processes the input file, setting up the necessary environment for debugging.
- Orchestrates the debugging process by coordinating with other components.

3.1.2 AbstractDDPlus class from package ddmin:

- Implementation of the delta debugging algorithm.
- Provides methods for both the basic delta debugging approach and an extended version that solve unresolved cases.
- Includes recursive algorithms (dd and ddAux) for systematically reducing the input set to isolate the minimal failure-inducing subset.
- Implements logic to partition input data into subsets and evaluate each subset contribution to the failure.

3.1.3 DDPlusTest class from package test:

- Interacts with the Alloy API to execute tests on model subsets, determining their satisfiability.
- Serves as a concrete implementation of the IDDPlusTest interface, providing a specific strategy for testing the satisfiability of subsets within Alloy models.
- By implementing the testing logic required by the delta debugging algorithm, DDPlusTest plays a pivotal role in the debugging process. It allows for the systematic reduction and examination of model components.

3.1.4 DataTransportObject class from package dto:

- Centralizes data management, acts as a centralized hub for managing and transporting data throughout the system. It encapsulates all necessary information, such as Alloy models, analysis commands, and configuration settings, facilitating easy access and manipulation of this data across different components of the application.
- Enhances modularity and reusability, by serving as a common data carrier, the class enhances the modularity of the system. It allows different parts of the application to interact with a unified data structure, promoting reusability and reducing the coupling between components.
- Improves debugging efficiency playing a role in the debugging process by transporting subsets of the Alloy model and their corresponding results between the testing and debugging algorithms.

3.2 Technologies used in the project

3.2.1 Java

Java as programming language for this project was important by its ability to meet the specific needs of analyzing and debugging Alloy models. Java cross-platform compatibility ensures the tool can be used across various operating systems. The language extensive library ecosystem offers ready-to-use resources for efficiently implementing algorithms like DDmin. Additionally, Java object-oriented design principles support the development of robust, maintainable code. The fact that the Alloy Analyzer is also Java-based simplifies integration efforts, making Java a natural fit.

3.2.2 Alloy Tools

Choosing the Alloy library for this project was a decision primarily due to its integration with Java and Alloy models, which are central to our project goals. The Alloy library offers a specialized set of tools for debugging and analyzing Alloy models.

3.2.3 argparse4j

This library is employed to facilitate command-line argument parsing, enhancing the usability and configurability of the tool. It provides a robust framework for defining, and handling command-line options and arguments, and other configurations necessary for running the tool.

3.3 Design Approach

3.3.1 Modular Design

The project is structured into distinct packages and classes, each with a well-defined responsibility. For example, the alloy package for managing Alloy model analysis, the dto package for data transport objects, the ddmin package for implementing the delta debugging algorithm, and the test package for testing strategies. This modular design facilitates maintenance, scalability, and the potential integration of additional functionalities.

3.3.2 Extensible Design

By employing interfaces like IDDPlusTest, allowing for different implementations of testing strategies or debugging algorithms without altering the core logic. This approach supports future adaptations and enhancements to the debugging process as new requirements or improvements are identified.

3.3.3 User-Centric Command-Line Interface

Utilizing argparse4j to parse command-line arguments reflects a user-centric design philosophy, aiming to make the tool accessible and easy to use. The interface allows users to specify analysis parameters and configurations dynamically.

3.3.4 Integration with External Libraries

The decision to use external libraries such as Alloy Tools to evaluate the models demonstrates a design approach that emphasizes the use of existing, proven solutions rather than reinventing the wheel. This not only improved speed in development, but also ensured reliability and standardization.

3.4 UML Diagram

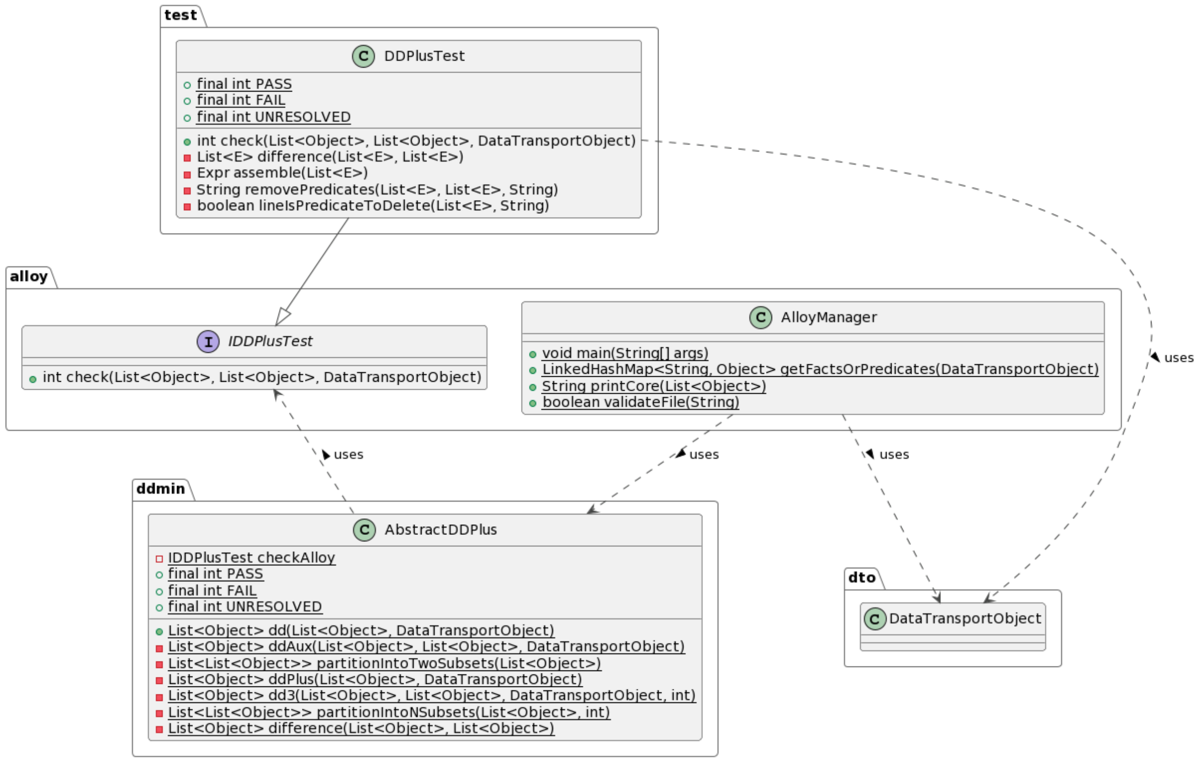
test
DDPlusTest
final int PASS
final int FAIL
final int UNRESOLVED
int check(List<Object>, List<Object>, DataTransportObject)
List<E> difference(List<E>, List<E>)
Expr assemble(List<E>)
String removePredicates(List<E>, List<E>, String)
boolean lineIsPredicateToDelete(List<E>, String)
alloy
IDDPlusTest
int check(List<Object>, List<Object>, DataTransportObject)
AlloyManager
void main(String[] args)
LinkedHashMap<String, Object> getFactsOrPredicates(DataTransportObject)
String printCore(List<Object>)
boolean validateFile(String)
uses
uses
uses
uses
ddmin
AbstractDDPlus
IDDPlusTest checkAlloy
final int PASS
final int FAIL
final int UNRESOLVED
List<Object> dd(List<Object>, DataTransportObject)
List<Object> ddAux(List<Object>, List<Object>, DataTransportObject)
List<List<Object>> partitionIntoTwoSubsets(List<Object>)
List<Object> ddPlus(List<Object>, DataTransportObject)
List<Object> dd3(List<Object>, List<Object>, DataTransportObject, int)
List<List<Object>> partitionIntoNSubsets(List<Object>, int)
List<Object> difference(List<Object>, List<Object>)
dto
DataTransportObject

Figure 1: UML Diagram of the System

3.5 Code Example

3.5.1 ddaux

```java
private static List<Object> ddAux(List<Object> input, List<Object> r, DataTransportObject dto) {
    if (input.size() == 1) {
        return input; // Found the problematic subset
    }

    List<List<Object>> subsets = partitionIntoTwoSubsets(input);
    List<Object> c1 = subsets.get(0);
    List<Object> c2 = subsets.get(1);

    List<Object> rUnionC1 = new ArrayList<>(r);
    rUnionC1.addAll(c1);

    List<Object> rUnionC2 = new ArrayList<>(r);
    rUnionC2.addAll(c2);

    int testResultC1 = checkAlloy.check(rUnionC1, input, dto);
    int testResultC2 = checkAlloy.check(rUnionC2, input, dto);

    if (testResultC1 == FAIL) {
        return ddAux(c1, r, dto);
    } else if (testResultC2 == FAIL) {
        return ddAux(c2, r, dto);
    } else if (testResultC1 == UNRESOLVED || testResultC2 == UNRESOLVED) {
        // Transition to Algorithm 2 if an unresolved case is encountered
        return ddPlus(input, dto);
    } else {
        List<Object> result = new ArrayList<>(ddAux(c1, rUnionC2, dto));
        result.addAll(ddAux(c2, rUnionC1, dto));
        return result;
    }
}
```

Listing 3: AbstractDDPlus.java - ddaux method

This method is designed to systematically isolate the minimal set of changes that lead to a failure in the program being debugged. It does this by recursively partitioning the set of all changes into smaller subsets and testing these subsets to identify whether they cause the failure.

3.5.2 lineIsPredicateToDelete

```java
public boolean lineIsPredicateToDelete(List<E> predicates, String line) {
    boolean isPredicateToDelete = false;
    for (int i = 0; i < predicates.size(); i++) {
        Func funcPredicate = (Func) predicates.get(i);
        String[] elementsNamePredicate = funcPredicate.label.toString().split("/");
        String namePredicate = elementsNamePredicate[elementsNamePredicate.length - 1];
        String regex = "\\bpred\\s+" + namePredicate + "\\s*(\\([^)]*\\))?\\s*\\{.*?\\}";
        Matcher matcherPred = findByRegex(regex, line);
        if (matcherPred.find()) {
            isPredicateToDelete = true;
            break;
        }
    }
    return isPredicateToDelete;
}
```

Listing 4: DDPlusTest.java - lineIsPredicateToDelete method

We designed to determine whether a specific predicate in the model corresponds to a predicate that should be removed. We choose to explain this method because the use of a regular expression that helps us to create predicate names in a dynamic way and compare them with the ones sent by the ddmin methods.

3.5.3 dd3

```java
private static List<Object> dd3(List<Object> input, List<Object> r, DataTransportObject dto, int n) {
        if (input.size() == 1) {
            return input; // Found the problematic subset
        }

        List<List<Object>> subsets = partitionIntoNSubsets(input, n);
        List<Object> cPrime = new ArrayList<>();
        List<Object> rPrime = new ArrayList<>(r);
        // int nPrime = Math.min(cPrime.size(), 2 * n);
        int nPrime;

        for (List<Object> ci : subsets) {
            List<Object> ciUnionR = new ArrayList<>(ci);
            ciUnionR.addAll(r);

            int ti = checkAlloy.check(ciUnionR, input, dto);

            if (ti == FAIL) {
                return dd3(ci, r, dto, 2); // Found in ci
            } else if (ti == UNRESOLVED) {
                List<Object> zi = new ArrayList<>(input);
                zi.removeAll(ci);
                zi.addAll(r);
                int tiPrime = checkAlloy.check(zi, input, dto);
                if (tiPrime == PASS) {
                    cPrime.addAll(ci);
                }
            }
        }

        if (!cPrime.isEmpty()) {
            nPrime = Math.min(cPrime.size(), 2 * n);
            rPrime.addAll(cPrime);
            List<Object> complementcPrime = difference(input, cPrime);
            return dd3(complementcPrime, rPrime, dto, nPrime); // Preference
        } else if (n < input.size()) {
            return dd3(input, r, dto, input.size()); // Try again
        }

        return new ArrayList<>(); // Nothing left
    }
```

Listing 5: AbstractDDPlus.java - dd3 method

This method was adapted to further refine the debugging process, improving the ability to isolate the cases of changes that cause failures when the evaluation of the subsets remain in the undefined state.

3.6 Tool usage

Input File(`-i` or `--input`):

- **Purpose:** Specifies the path to the Alloy model file.
- **Usage Example:** `--input path/to/model.als`

Facts Analysis Mode(`-f` or `--facts`):

- **Purpose:** Indicates that the tool should focus on analyzing errors related to facts in the Alloy model.
- **Usage Example:** `--facts`

Predicates Analysis Mode(`-p` or `--predicates`):

- **Purpose:** Signals that the debugging process should concentrate on predicates within the Alloy model.
- **Usage Example:** `--predicates`

Trace Logging (`-t` or `--trace`):

- **Purpose:** Enables detailed trace logging to provide an in-depth view of the analysis performed by ddmin.
- **Usage Example:** `--trace`

3.6.1 Examples tool usage

- Analyzing Facts with Trace Logging

```
        java -jar DeltaDebugging-0.1.0.jar --input "path/to/model.als" --facts --trace

```

- Analyzing Predicates

```
        java -jar DeltaDebugging-0.1.0.jar --input "path/to/model.als" --predicates

```

3.7 Limitations

Currently, the tool is not equipped to effectively process alloy models using assert and check instructions. Another improvement for future versions is related to the ability to do error evaluation not separately (fact or predicate) but not both at the same time.

4 Evaluation

4.1 Javadoc

All code was documented following javadoc standards [1] command used to generate documentation:

```
javadoc -private -classpath ./lib/org.alloytools.alloy.dist.jar:./lib/argparse4j-0.9.0.jar:./lib/
  slf4j-nop-1.6.6.jar -d docs -sourcepath src -subpackages alloy:ddmin:dto:test
```

The documentation generated by the command can be found at:

`https://krpovmu.github.io/DeltaDebugging/`

4.2 Validation

To ensure the correctness in our implementation we have tested our prototype on 100 alloy model files [2] extracted from Alloy4Fun [3]. These checks were automated in benchmarks.sh script [4], we use Java Flight Recorder (JFR) that is a monitor tool which collects information about the events in a Java Virtual Machine (JVM) during the execution of a Java application. JFR is part of the JDK distribution, and it's integrated into the JVM. The results were generated in folder jfr [5] and analyzed using the script performance_costs_summary.sh [6],then the result of this script was saved on performance_costs_summary.csv [7], the final analysis was performed with the script in python using pandas and matplotlib performance_costs_graphs.py [8]

[1]Java doc https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html
[2]Project models https://github.com/krpovmu/DeltaDebugging/tree/master/resources/als_models
[3]Alloy4fun models https://github.com/haslab/Alloy4Fun/wiki/EM-19-10
[4]Benchmark script https://github.com/krpovmu/DeltaDebugging/blob/master/resources/benchmarks.sh
[5]jfr output files https://github.com/krpovmu/DeltaDebugging/tree/master/resources/jfr_output
[6]Performance cost script https://github.com/krpovmu/DeltaDebugging/blob/master/resources/performance_cos
[7]Performance cost results https://github.com/krpovmu/DeltaDebugging/blob/master/resources/performance_co
[8]Performance costs graphs https://github.com/krpovmu/DeltaDebugging/blob/master/resources/performance_c

4.3 Performance Cost

The analysis cost performance covers several aspects such as CPU time usage, heap memory usage, garbage collection impact, concurrency costs, and I/O operations.

All tests were executed on a laptop Dell Latitude 7480, with an Intel Core i7-6600U 2.60GHz and 16 GB of RAM.

The performance cost analysis can also be found at:

`https://krpovmu.github.io/DeltaDebugging/performance_costs_graphs/`

4.3.1 Graphs Overview

- **CPU Time Usage (%):** Shows the percentage of CPU time used by each model, helping to identify models that are more CPU-intensive.
- **Heap Usage (MB):** Displays the amount of heap memory used by each model in megabytes, indicating the memory footprint of the models.
- **GC Total Pause Time (ms):** Represents the total time in milliseconds that the application was paused for garbage collection, highlighting the impact of GC on application responsiveness.
- **Concurrency Cost (ms):** Illustrates the time spent in milliseconds waiting due to concurrency issues, such as lock contention, which can impact application performance.
- **I/O Written Duration (ms) and I/O Read Duration (ms):** Show the duration of I/O operations for writing and reading, respectively, in milliseconds, providing insight into the cost of I/O operations on performance.

4.3.2 Graphs

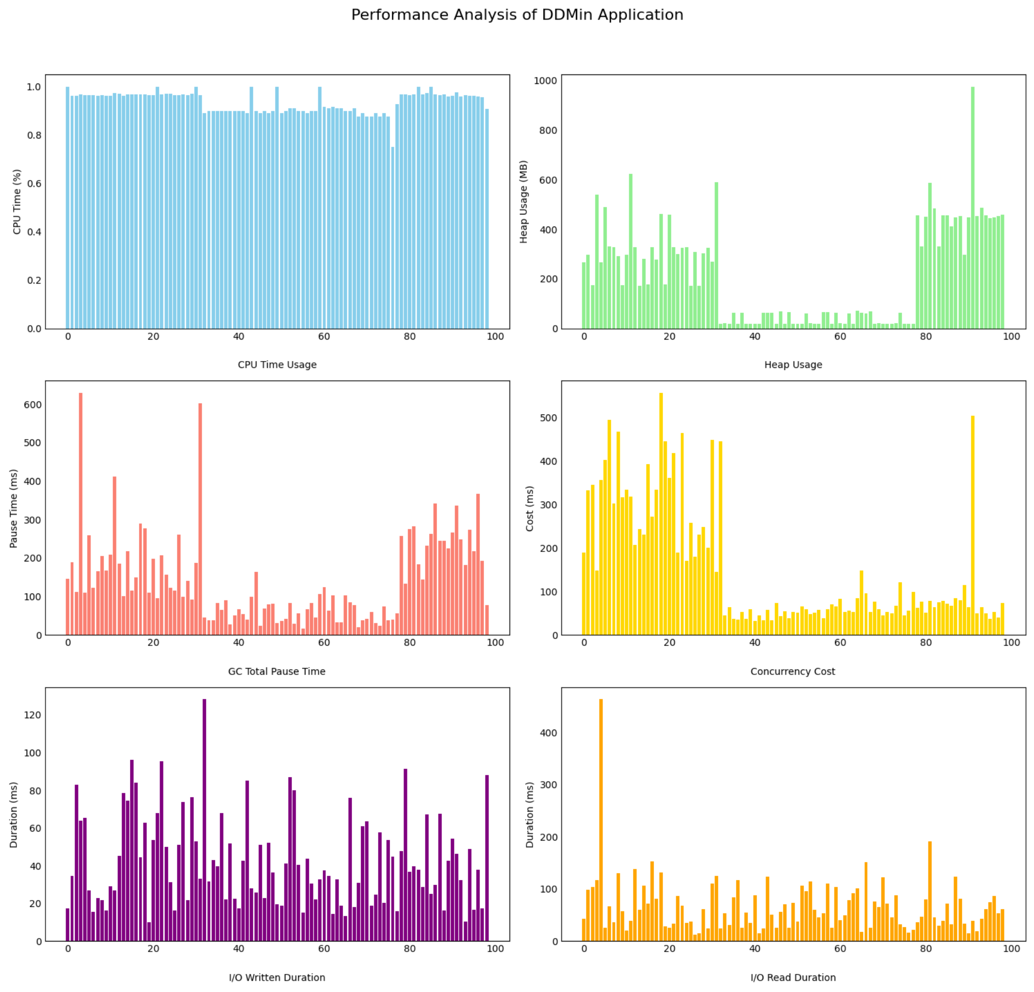

Figure 2: Performance costs graphs

4.4 Benchmarks DDmin Implementation vs Native (Alloy Minisat UNSAT Core)

We performed several benchmarks focused on 2 points, execution time and correctness, for these tests We used 20 als models which are included in als_models folder on resources, these tests were performed on a Dell Latitude 7480 laptop with Intel Core i7-6600U dual core 64 bits processor with 16 GiB of ram, all the specifications of the laptop where the tests were performed are in the inxi.txt file located in the resources directory of the project.

The benchmark analysis can also be found at:
`https://krpovmu.github.io/DeltaDebugging/comparision_implementation_native/`

4.4.1 Performance

The first graph compares the execution times between the custom implementation and the native tool for each model. You can observe how each pair of bars represent a model, allowing you to compare directly the performance of both implementations.

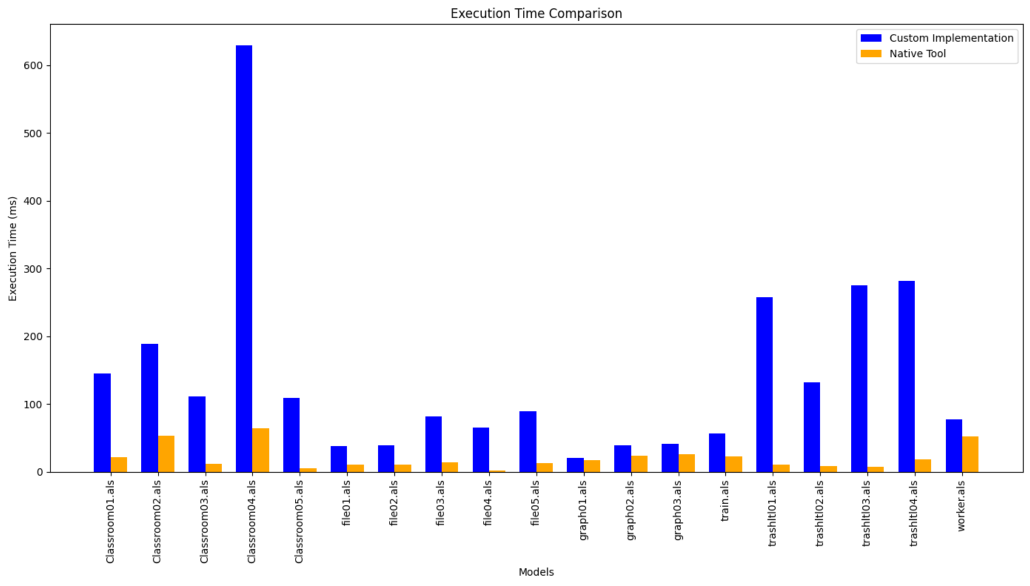

Figure 3: Execution times

4.4.2 Correctness

The second graph shows the correctness of the custom implementation compared to the native tool's results. A model's result is considered ”correct” if the outcomes from both implementations match. The bars represent the count of models where the custom implementation was correct versus incorrect.

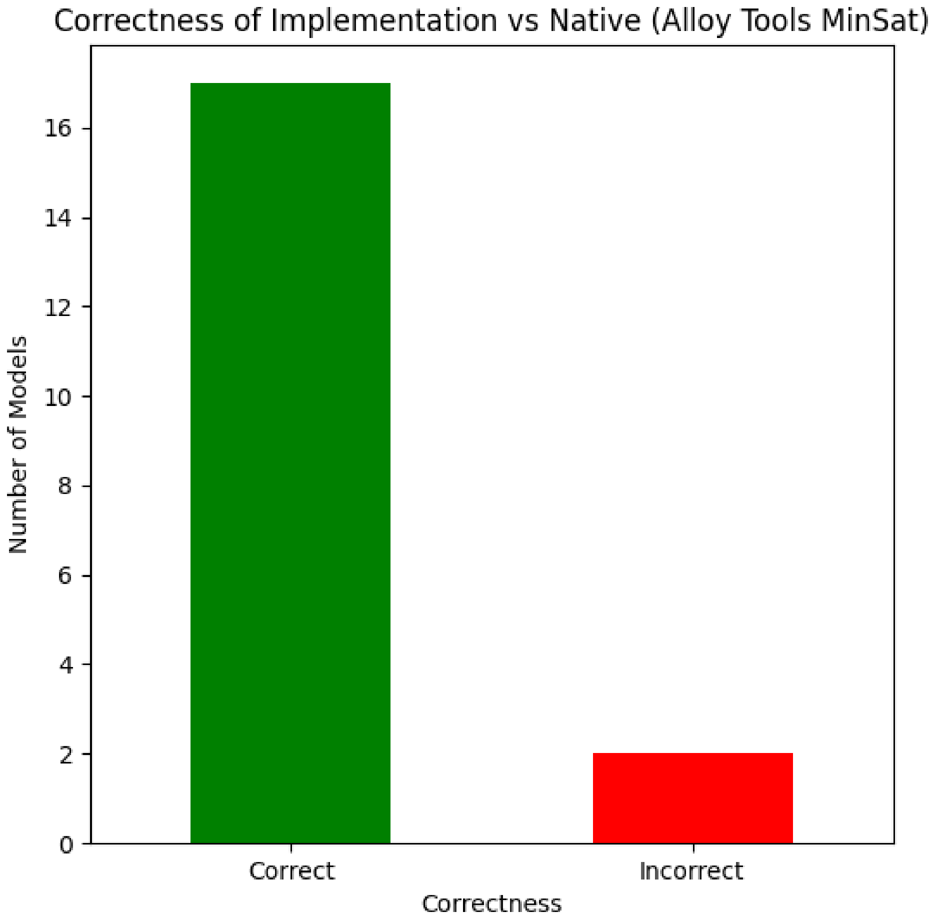

Figure 4: Correctness

References

[1] Zeller, A. (1999). Yesterday, my program worked. Today, it does not. Why? *ACM Sigsoft Software Engineering Notes*, 24(6), 253–267. `https://dl.acm.org/doi/10.1145/318774.318946`

[2] Lynce, I., & Marques-Silva, J. (2004). On Computing Minimum Unsatisfiable Cores. *Journal Article*. `https://eprints.soton.ac.uk/262252/1/jpms-sat04a.pdf`

[3] Liffiton, M. H., & Sakallah, K. A. (2007). Algorithms for computing minimal unsatisfiable subsets of constraints. *Journal of Automated Reasoning*, 40(1), 1–33. `https://link.springer.com/article/10.1007/s10817-007-9084-z`

[4] Torlak, E., Chang, F. S., & Jackson, D. (2008). Finding minimal unsatisfiable cores of declarative specifications. In *Springer eBooks* (pp. 326–341). `https://link.springer.com/chapter/10.1007/978-3-540-68237-0_23`

[5] Zeller, A. (2009). Simplifying problems. In *Elsevier eBooks* (pp. 105–127). `https://www.sciencedirect.com/science/article/abs/pii/B9780123745156000058`