

# Unsat Cores in Alloy

Formal Methods for Software Engineering WS22/23

Martin Tippmann and Carlos Poveda

Bauhaus-Universität Weimar

**Abstract.** Alloy Analyzer, employs unsatisfiable cores to reveal inconsistencies within system specifications. This work focuses on how the development detects these cores when no instance satisfies a given model, particularly using the run command. The unsatisfiable core, a set of conflicting formulas, serves as a critical asset for diagnosing potential specification errors. By exploring case studies and practical scenarios, we demonstrate the Analyzer's efficacy in refining model accuracy, offering insights into its indispensable role in system design and validation in complex software engineering applications.

## Introduction

### Background

Alloy is a prominent open-source language and tool designed for software modeling. It's celebrated for its utility in various domains, particularly in enhancing security mechanisms and advancing complex network designs like telephone systems. Alloy's platform, built upon the Kodkod model-finding engine, offers an extensive toolkit for software development and analysis. This includes detailed language documentation, user-friendly software downloads, and a wide array of practical case studies. The platform's evolution signifies its growing impact in the software engineering field, particularly in complex system design and architecture, showcasing its ability to streamline processes and foster innovative solutions.

Delta debugging is a methodical algorithm designed to autonomously pinpoint and minimize the specific conditions required to trigger a bug. This technique is versatile and can be utilized for identifying a range of failure-inducing factors, such as:

- The inputs to a program
- User interactions, including keystrokes, button activations, and mouse actions
- Modifications in program code, encompassing the addition, alteration, or removal of variables, functions, classes, and more.

Before proceeding further, it's essential to familiarize with some key terms:

- **Configuration:** This refers to a specific group of source code changes that lie between the last confirmed working version and the version exhibiting the regression.
- **Test:** A test case or function that evaluates whether a configuration triggers the failure (resulting in Failure), doesn't trigger the failure (resulting in Success), or yields inconclusive results (possibly due to interference or inconsistency).
- **Subset:** For two sets A and B, A is considered a subset of B if every element in A is also found in B.
- **Superset:** For two sets A and B, B is a superset of A if it includes every element present in A.
- **Union:** For two sets A and B, the union consists of all elements that are either in A, B, or both.

- **Intersection:** For two sets A and B, the intersection includes only those elements that are common to both A and B.

In considering the intricacies of the delta debugging algorithm, it's important to be aware of these potential complexities:

- **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 changes in the source code, which could prevent successful compilation and testing if not all dependent changes are included. A mechanism to identify and manage these dependencies, like a dependency graph, is essential.  
*A side thought:* Ensuring consistency in applying configurations could draw on aspects of the compilation process, like lexical analysis, which breaks down the code into tokens, followed by dependency analysis. This could theoretically help in understanding dependencies between code changes.

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.
- 
- **Granularity:** A single logical change might involve numerous lines of code, 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.
  - **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."

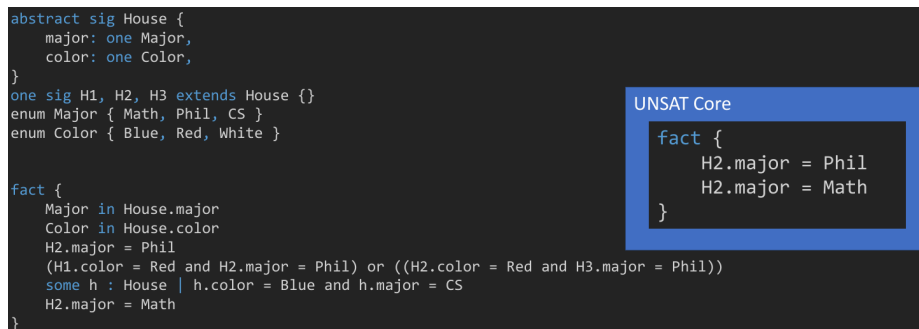
## 1.1 Motivation

The integration of the DDmin algorithm for identifying UNSAT cores in Alloy models presents a targeted approach to a crucial aspect of software modeling. UNSAT cores, the subsets of constraints in a model that cannot be satisfied simultaneously, show significant challenges in system design. DDmin methodical reduction and testing strategy offers a precise way to isolate these cores efficiently. By applying DDmin, we aim to enhance the diagnostic process, enabling a more detailed and rapid identification of conflicts within Alloy models. This specific application of DDmin aligns with the core needs of Alloy, promising a significant advancement in the field of model-based system analysis and design.

## 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, such as a bug or a failure, in a software program. 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.

## 3 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
}
  
```

The image shows an Alloy code snippet. A blue box highlights a section of the code labeled "UNSAT Core". This section contains a `fact` block with two lines: `H2.major = Phil` and `H2.major = Math`. These two lines are contradictory, as they assert that `H2.major` is both `Phil` and `Math` simultaneously, which is logically impossible.

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:

- **Unsatisfiable Core:** Any subset of a formula that is unsatisfiable.
- **Minimum Unsatisfiable Core:** The smallest subset (in terms of the number of clauses) of a formula that is unsatisfiable.

Given that removing these two lines makes the model satisfiable, they do form an unsatisfiable core. To assess whether it's a minimum unsatisfiable core, you would need to consider if there is

any smaller subset of clauses that also renders the model unsatisfiable. If no smaller unsatisfiable subset exists, then this core is indeed the minimum.

## 4 Implementation

The project leverages Java's robust object-oriented programming capabilities to implement a delta debugging algorithm for Alloy models. The implementation is divided into several key components, each serving a distinct role in the debugging process.

### Key Components and Their Functionalities

#### AlloyManager Class

- 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.

#### AbstractDDPlus Class

- Central to the implementation of the delta debugging algorithms.
- Provides methods for both the basic delta debugging approach and an extended version that handles 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's contribution to the failure.

#### DDPlusTest Class

- Implements the IDDPlusTest interface, providing custom testing logic tailored for Alloy models.
- Executes tests on different parts of the Alloy model based on the selected debugging mode.
- Interfaces with the Alloy Analyzer's API, performing model translations and evaluations to facilitate the debugging process.

## Technologies and Libraries

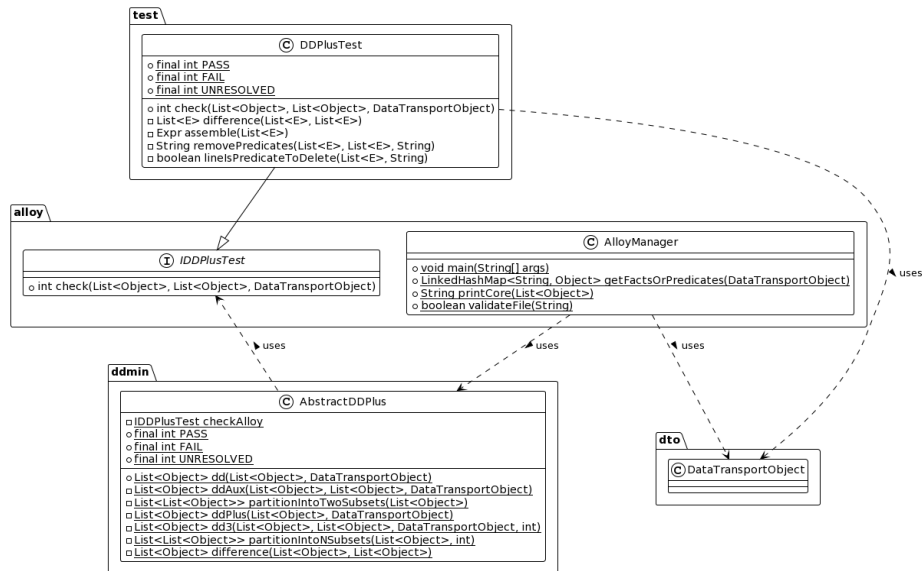
- **Java:** The choice of Java as the programming language is pivotal due to its strong object-oriented features, extensive standard library, and cross-platform capabilities.
- **Alloy Analyzer Integration:** The application integrates with the Alloy Analyzer, a tool for modeling and analyzing Alloy models, to manipulate and test these models programmatically.
- **Argparse4j:** This library is used for parsing command-line options, enhancing the flexibility and user-friendliness of the application by allowing users to specify various parameters and modes for debugging.

## Design Approach

The application is designed with clarity and modularity in mind. Each class has a well-defined responsibility, making the codebase maintainable and scalable.

- The use of interfaces and separation of concerns allows for easier testing and potential future extensions of the application.
- Exception handling and input validation are key aspects, ensuring robustness and reliability in different scenarios.
- This Java implementation forms the backbone of the project, harnessing the power of delta debugging to provide a valuable tool for analyzing and debugging Alloy models. The use of Java, combined with specific libraries and a clear architectural approach, makes the application robust, flexible, and capable of handling various debugging scenarios.

## UML Diagram



#### 4.1 Code Example

Example of Interference in the Delta Debugging Java Program

- **Understanding Interference in the Context:**

**Interference** in delta debugging refers to a situation where individual changes or modifications in a program do not independently cause a regression or bug, but a specific combination of these changes does. In our Java application, which focuses on debugging Alloy models, this could translate to certain combinations of factors like program inputs, user interactions, or code modifications leading to unexpected or erroneous behavior.

**Example in our Java Application:**

Consider the **AlloyManager** class processing an Alloy model with numerous modifications (e.g., changes in predicates, facts). Individually, these modifications may not cause any issues. However, when combined in a certain way, they might lead to a failure that is not evident when these elements are considered separately. This is a classic case of interference.

In the **AbstractDDPlus** class, while implementing the delta debugging algorithm, the method of recursively testing subsets of changes is crucial. The algorithm might need to consider not just individual changes but also their interactions. For example, a certain predicate modification in combination with a specific fact alteration might cause an Alloy model to behave unexpectedly, which would not occur if these changes were applied independently.

- **Handling Interference:**

Your implementation would need to account for this by not only testing individual changes but also various combinations of these changes. This approach can be complex, as the number of potential combinations increases with the number of changes.

The delta debugging process, as orchestrated by your classes, would involve systematically and iteratively testing different combinations of changes to identify the minimal set that leads to the failure. This is where the recursive approach in **AbstractDDPlus** plays a key role.

#### 4.2 Tool usage

The **AlloyManager** class serves as the main entry point and parses these arguments to determine the operational mode and process the Alloy model accordingly.

**Command-Line Arguments**

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

Purpose: Specifies the path to the Alloy model file.

Usage Example: --input path/to/model.als

Internal Processing: The provided file path is validated and used to load the Alloy model. This is the primary input for the tool and is essential for the debugging process to commence.

- **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

Internal Processing: When this mode is selected, AlloyManager directs the debugging process to scrutinize the facts within the Alloy model. It isolates sections related to facts and applies delta debugging techniques to identify minimal failure-inducing facts.

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

Purpose: Signals that the debugging process should concentrate on predicates within the Alloy model.

Usage Example: --predicates

Internal Processing: In this mode, the tool's focus shifts to predicates. Similar to the facts analysis mode, it isolates predicate-related elements in the model and employs delta debugging to pinpoint problematic predicates.

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

Purpose: Enables detailed trace logging to provide an in-depth view of the debugging process.

Usage Example: --trace

Internal Processing: With trace logging activated, AlloyManager produces verbose output throughout the debugging process. This includes detailed logs of the steps taken, decisions made, and intermediate results. It's particularly useful for understanding the debugging flow and for troubleshooting.

## Using the Tool

To use the tool, construct a command with the necessary arguments based on what you need to analyze or debug in your Alloy model. Here's how you can combine the arguments:

### Analyzing Facts with Trace Logging:

```
java -jar your-jar-file.jar --input "path/to/model.als" --facts --trace
```

### Analyzing Predicates:

```
java -jar your-jar-file.jar --input "path/to/model.als" --predicates
```

The combination of these arguments provides a flexible way to target specific aspects of the Alloy model and control the verbosity of the output, making the tool adaptable to various debugging needs.

### 4.3 Limitations of the Implementation

- Performance optimization and scalability improvements are potential areas of focus, aiming to handle larger and more complex Alloy models more efficiently.
- Refactoring for improved modularity and testability could be considered to enhance the maintainability of the codebase.
- The tool is currently not equipped to effectively process Alloy models that utilize **assert** and **check** instructions. These instructions are commonly used in Alloy for assertions and model checking, which are instructions for validating model properties and behavior.

## 5 Evaluation

### 5.1 Validation

To ensure the correctness of our implementation we have tested our prototype on 2 alloy files, the first one provided by Jan is called houses.als, the result activating the trace flag:

```
=====
ID : Element
=====
1 : this/Major in this/House . (this/House <: major)
2 : this/Color in this/House . (this/House <: color)
3 : this/H2 . (this/House <: major) = this/Phil
4 : OR[AND[this/H1 . (this/House <: color) = this/Red, this/H2 . (this/House <: major) = this/Phil],
AND[this/H2 . (this/House <: color) = this/Red, this/H3 . (this/House <: major) = this/Phil]]
5 : (some h | AND[h . (this/House <: color) = this/Blue, h . (this/House <: major) = this/CS])
6 : this/H2 . (this/House <: major) = this/Math
=====
Iteration 1 : [1, 2, 3] Result: PASS
Iteration 2 : [4, 5, 6] Result: PASS
Iteration 3 : [4, 5, 6, 1] Result: PASS
Iteration 4 : [4, 5, 6, 2, 3] Result: FAIL
Iteration 5 : [4, 5, 6, 2] Result: PASS
Iteration 6 : [4, 5, 6, 3] Result: FAIL
Iteration 7 : [1, 2, 3, 4] Result: PASS
Iteration 8 : [1, 2, 3, 5, 6] Result: FAIL
Iteration 9 : [1, 2, 3, 5] Result: PASS
Iteration 10 : [1, 2, 3, 6] Result: FAIL
=====
```

**Fact: line 23, column 14, filename=houses.als: this/H2 . (this/House <: major) = this/Phil**

**Fact: line 26, column 14, filename=houses.als: this/H2 . (this/House <: major) = this/Math**

```
=====
```



As we can see the problematic cores are the number 3 and number 6 (Elements ID list), our software needed 10 iterations to find the failing core.

## 5.2 Performance Cost

Given the specific hardware configuration of your Dell Latitude 7480, with an Intel i7 processor and 16 GB of RAM, the limited testing conducted on just two Alloy models does not fully leverage the laptop's capabilities. This setup is more than capable of handling complex and resource-intensive Java applications, suggesting that the application could be tested more extensively without straining the system. Expanding the testing to include a broader range of Alloy models would not only provide a comprehensive evaluation of the application's performance but also utilize the laptop's high processing power and substantial memory capacity to their full potential. This would ensure a thorough assessment of the application's functionality and efficiency across various scenarios.

This limitation highlights the need for accessing a more diverse set of Alloy models to thoroughly test and validate the application. Comprehensive testing with a variety of models is crucial to identify potential areas of improvement, ensure reliability across different use cases, and enhance the tool's overall capabilities. As we seek to expand our testing scope, collaboration with the Alloy community or sourcing models from different domains could be beneficial in overcoming this challenge.

## 6 Discussion & Conclusion

In conclusion, this project represents a significant step in the realm of delta debugging for Alloy models, yet there is ample room for enhancement. Future expansions could include adapting the tool to effectively handle assert and check instructions in Alloy, which are currently a limitation. Additionally, improving the methods for extracting predicates and facts could lead to more precise debugging. Looking further ahead, integrating the capability to analyze functions, signatures, and other structural elements that could cause errors in Alloy models would greatly enhance the tool's comprehensiveness and utility. This evolution would make the tool not only more versatile but also more aligned with the complex needs of Alloy model debugging.

## 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](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>