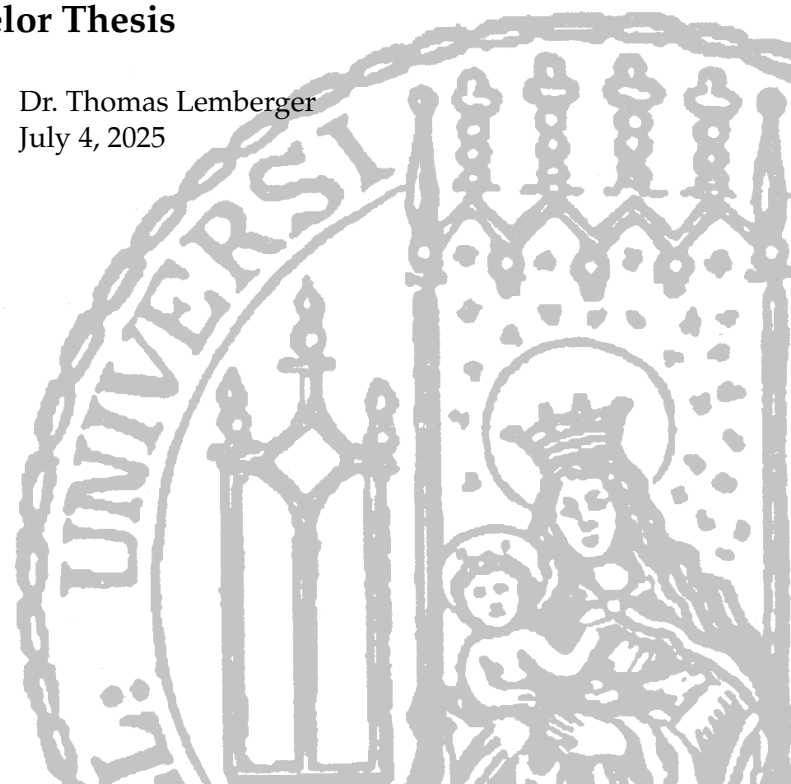
INSTITUT FÜR INFORMATIK
Ludwig-Maximilians-Universität München

LLM-Based Summary Simplification for Distributed Summary Synthesis

Caspar Spang

Bachelor Thesis

Supervisor Dr. Thomas Lemberger
Submission Date July 4, 2025



Statement of Originality

English:

Declaration of Authorship

I hereby confirm that I have written the accompanying thesis by myself, without contributions from any sources other than those cited in the text and acknowledgments.

Deutsch:

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, July 4, 2025 Caspar Spang

Abstract

The increasing scalability of verification systems ensures the reliability and safety of software, which is especially crucial for modern, large-scale applications. This thesis addresses the problem of unnecessarily large and complex SMT formulas that create inefficiencies within the distributed software-verification tool Multi-Processing Distributed Summary Synthesis (DSS). We approached this problem by implementing the TRIMMER; a microservice that leverages LLMs to continuously simplify SMT formulas from the DSS database of block summaries. Our evaluation demonstrates that the TRIMMER achieves average simplifications around 40 % of the original formula length, although the impact on verification times could not be fully evaluated, due to limited integration of the simplified formulas into DSS'S analysis. These findings indicate that simplifying SMT formulas with LLMs is definitely possible, but the approach cannot yet replace algorithmic simplification within the current configuration of Multi-Processing DSS.

Contents

Contents v

List of Figures vi

1 Introduction 1

2 Related Work 5

3 Background 9
3.1 Formal Methods and SMT . . . 9
3.2 Verification Tools and Model-Checking . . . 11
3.3 Implementation Tools and Google Gemini . . . 13

4 Contribution 15
4.1 Workflow and Approach . . . 15
4.2 Fetching messages . . . 15
4.3 Extracting the SMT formula . . . 17
4.4 Simplification . . . 18
4.5 Validation . . . 18
4.6 Replacing the SMT formula . . . 19
4.7 Updating the database . . . 20
4.8 Design choices . . . 20
4.9 Prompting the LLM . . . 23
4.10 Statistics Collection and Start Script . . . 24

5 Evaluation 27
5.1 Research Questions . . . 27
5.2 Evaluation Setup . . . 27
5.3 Evaluation Results . . . 28
5.4 Summary . . . 34
5.5 Threats to Validity . . . 34

6 Future Work 41

7 Conclusion 43

Bibliography 45

List of Figures

1.1 Integrating the TRIMMER into Multi-processing DSS 2

3.1 Original DSS concept 12
3.2 Multi-Processing DSS concept 12

4.1 Workflow of our approach 16
4.2 LLM client implementation 18

5.1 Boxplot of the average simplified lengths per task 29
5.2 Per task comparison of block amount, amount of original messages, average length of original messages 39
5.3 Walltimes of every task for every run 40

1 Introduction

The relevance of software verification has been growing immensely since the integration of artificial-intelligence (AI) systems into the development process of software systems, and into production code itself. Ensuring the reliability and safety of technologies created with or using AI hinges on robust, efficient, and scalable verification processes. The research project Multi-Processing Distributed Summary Synthesis (DSS) [43], a progressive iteration of the original Distributed Summary Synthesis project [27], aims at increasing said efficiency and scalability. Within this thesis project, we contribute to Multi-Processing DSS a microservice that employs AI for SMT formula simplification. Though this might seem contradictory at first, this approach is a well justified experiment to challenge whether this kind of simplification improves Multi-Processing DSS, and whether current AI models are up to par for such complex tasks.

Goals. Our goal is to improve the runtime and scalability of the Multi-Processing DSS project, an advanced verification tool for C programs that uses distributed analysis and parallelization. We add a new microservice (TRIMMER) to the ecosystem of DSS that connects to the database of block summaries and continuously simplifies them. For the simplification of block summaries we use LLMs (most notably Gemini's API) that receive an original formula and then generate a shorter, simplified formula. After simplification, the microservice checks for syntax and overapproximation of the original formula. Only then does it write the simplified version to the database. We evaluate by comparing the performance of DSS with and without the simplified messages on selected tasks from sv-benchmarks.

Motivation. Multi-Processing DSS pioneers parallelized, distributed analysis by breaking down a verification task into many independent analysis blocks that communicate with each other through messages. These messages contain block summaries that represent the postconditions and violation conditions of the analysis blocks. They can blow up in size very quickly for complicated programs (exceeding 50k characters), especially for loops, since the program states of each iteration are simply concatenated via conjunction. But not all the information contained in such long messages is needed for successful verification, which creates inefficiencies. Therefore, we aim to create elegant, simplified formulas that can be used instead of the originals while keeping the verification sound. By using these simplified messages during analysis, we try to improve the total verification runtime. Other approaches to simplify the formulas (e.g. quantifier elimination, using SMT solvers to simplify) have been unsuccessful, due to weak performance by the SMT solver Z3.

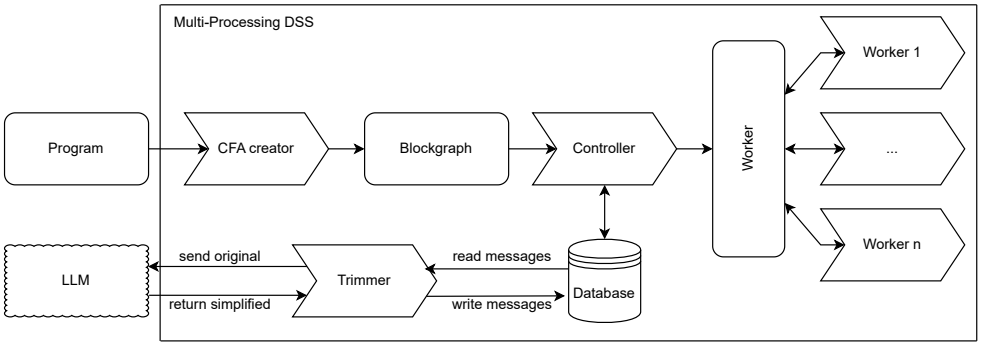


Figure 1.1: Integrating the TRIMMER into Multi-processing DSS

In exploratory work we have already shown that LLMs are capable of simplifying such SMT formulas while maintaining correct syntax and achieving overapproximation. We have also shown that simplification in real time is not feasible, as the delay from data transfer and time spent 'thinking' by the LLMs is too large. This leaves the continuous simplification of formulas as the best alternative to improve DSS.

Overview. We embed the TRIMMER as a Quarkus service into the existing microservice-based DSS project Figure 1.1. Multi-processing DSS takes in a C program and creates a control flow automaton from it, which is transformed into a blockgraph. Then, the Controller distributes the work onto workers - the analysis blocks. The Controller schedules message distribution between the workers by storing their messages in a database. Whenever a message is needed by a worker, it is provided by the Controller. The graphic shows that the TRIMMER is an entirely separate project from Multi-processing DSS. The only point of connection is the database. That is the instance the TRIMMER reads original messages from and continuously writes simplified counterparts to. We implement the TRIMMER in Kotlin, and continuously access the block summaries in form of SMT-LIB formulas through the database. The formulas are parsed before we send them to an LLM (in our implementation Google Gemini's API) with a prompt requesting a simplified formula that is at least implied by the original. Upon return of the formula, we check for correct SMT-LIB syntax and overapproximation. After that the new formula is written to the same database, where it is ready to be used by Multi-Processing DSS.

Example. A good example for a long SMT formula produced by Multi-Processing DSS is Listing 1. Such a formula is read from the Multi-Processing DSS database and sent to an LLM as String with a prompt like this:

> I have the following SMT formula in SMTlib2 format: " + formula + " It describes a C program in SSA form. Provide a smaller formula, you have to guarantee that the new formula is implied by the original formula! Also it is absolutely vital that the new formula is in valid SMTlib2 syntax! Make sure you do not overlook any necessary declarations! If you cannot find a result, return 'No smaller formula found'. Do not add any line breaks to the output!

```
(set-info :source |printed by MathSAT|)\\n(declare-fun |main::y@1| ()
    (_ BitVec 32))\\n(declare-fun |main::y@2| () (_ BitVec 32))\\n(
    declare-fun |main::x@1| () (_ BitVec 32))\\n(declare-fun |main::x@2
    | () (_ BitVec 32))\\n(declare-fun |main::x@3| () (_ BitVec 32))\\n
    (declare-fun |main::y@3| () (_ BitVec 32))\\n(assert (let ((.
    def_104 (bvslt |main::y@3| (_ bv10 32))))(let ((.def_101 (bvadd (_
    bv1 32) |main::y@3|)))(let ((.def_102 (= |main::y@2| .def_101)))(
    let ((.def_96 (bvadd (_ bv1 32) |main::x@3|)))(let ((.def_97 (= |
    main::x@2| .def_96)))(let ((.def_44 (bvadd (_ bv1 32) |main::x@2|))
    )(let ((.def_45 (= |main::x@1| .def_44)))(let ((.def_36 (= |main::
    x@1| (_ bv14 32))))(let ((.def_46 (and .def_36 .def_45)))(let ((.
    def_42 (bvadd (_ bv1 32) |main::y@2|)))(let ((.def_43 (= |main::y@1
    | .def_42)))(let ((.def_47 (and .def_43 .def_46)))(let ((.def_40 (
    bvslt |main::y@2| (_ bv10 32))))(let ((.def_41 (not .def_40)))(let
    ((.def_48 (and .def_41 .def_47)))(let ((.def_98 (and .def_48 .
    def_97)))(let ((.def_103 (and .def_98 .def_102)))(let ((.def_106 (
    and .def_103 .def_104))).def_106))))))))))))))))))
```

Listing 1: Long original formula produced by DSS

```
(declare-fun |main::y@2| () (_ BitVec 32)) (declare-fun |main::x@2| ()
     (_ BitVec 32)) (declare-fun |main::x@3| () (_ BitVec 32)) (declare
    -fun |main::y@3| () (_ BitVec 32)) (assert (let ((.def_96 (bvadd (_
     bv1 32) |main::x@3|))) (let ((.def_97 (= |main::x@2| .def_96))) (
    let ((.def_101 (bvadd (_ bv1 32) |main::y@3|))) (let ((.def_102 (=
    |main::y@2| .def_101))) (let ((.def_104 (bvslt |main::y@3| (_ bv10
    32)))) (let ((.def_103 (and .def_97 .def_102))) (let ((.def_106 (
    and .def_103 .def_104))) .def_106))))))))
```

Listing 2: Simplified counterpart of Listing 1

The LLM should return a shorter SMT formula. A validator then checks the simplified formula for correct syntax and whether it is implied by the original formula. For example the long input formula from Listing 1 is successfully reduced to the much shorter formula in Listing 2. In this case, the simplified formula omits a guard that forces a loop to have reached a certain threshold (y2 = 10). With the goal of upholding verification soundness, our evaluation must show that omitting such information does not lead to worse performance. Only after these checks the simplified formula is written to the DSS database.

Scope. Within this project we implement the microservice with appropriate unit and integration tests, statistics and metrics collection, and documentation. Also we conduct a broad evaluation on selected verification tasks from sv-benchmarks - a well maintained, large set of benchmark verification tasks for various C programs. We do not implement or evaluate any non-SMT techniques, nor any alternative approaches to formula simplification. This has been attempted previously, but proven too difficult so far. We also do not integrate any solvers other than MathSAT5 through JavaSMT, or any LLM backend other than Google Gemini. While we integrate the microservice into the Kubernetes cluster, we do not provide a full production deployment version, complete with CI/CD pipelines or optimisation. This is an ongoing research project. The TRIMMER uses Google Gemini, but we do

not study hallucination mitigation or certify the LLM's output beyond our SMT validation module. Within our evaluation we demonstrate practical results but do not provide formal worst-case complexity bounds or optimality guarantees concerning trimming.

Evaluation. These are the research questions we aim to answer in our evaluation:

RQ 1. How much shorter are the formulas produced?
RQ 2. How often does the TRIMMER'S simplification succeed?
RQ 3. How many API requests are wasted due to incorrect LLM responses?
RQ 4. Does the TRIMMER improve the verification time of DSS?
RQ 5. Are there cases where the TRIMMER leads to worse results?

We do not investigate the influence on memory consumption as Multi-processing DSS already breaks down a verification task into many very small blocks, which is very memory-efficient. To answer the questions we compare the performance of DSS with and without the simplified formulas on selected tasks from sv-benchmarks. DSS already provides us with the total verification time, from the TRIMMER we collect lengths of original and simplified formulas, total amount of original messages, amount of syntactically/logically correct simplifications, total LLM requests, reasons for unsuccessful LLM requests, and latencies of the LLM and SMT solver calls.
From the TRIMMER we collect metrics like: lengths of original and simplified messages and amount of logically correct/incorrect simplifications.

2 Related Work

Foundational Methods. The simplification of logical formulas dates back several decades, particularly in the context of program verification and theorem proving. The combination frameworks developed by Nelson and Oppen, and Shostak established the theoretical foundation for decision procedures used today. Nelson and Oppen published a first version of a 'simplifier' for Boolean logic formulas [48], which 'normalises' formulas by transforming them into conditional 'if-then-else' expressions. These conditional expressions are then algorithmically simplified, which marks the birth of computational simplification on multi-theory formulas. Shostak worked towards more general decision procedures for formulas involving diverse semantic constructs [54], laying important groundwork for the Satisfiability Modulo Theories (SMT). Subsequently, Clarke et al. introduced Counterexample-Guided Abstraction Refinement (CEGAR) [33], an iterative abstraction strategy that simplifies verification tasks by overapproximating complex conditions (the original program), refining them only when spurious counterexamples are detected. Similarly, the predicate abstraction approach used in the project SLAM [20] demonstrated significant scalability gains by abstracting complex arithmetic and pointer conditions into simpler Boolean formulas. Similarly to SLAM, we also use the principle of ensuring correctness through controlled simplification - where simplified formulas always remain sound overapproximations.

Origins of Simplification in Syntactic Simplification. The Simplify theorem prover [37] was among the first systems to apply rewriting-simplification before solving, showing that preprocessing significantly improved verification performance. Building on these findings, modern SMT solvers, including Z3 [35], Boolector [29], and CVC4 [23], rely heavily on preprocessing techniques to simplify input formulas before solving. Most work on preprocessing is published on Z3, where many new preprocessing features have been added since the original paper introduced the base 'simplifier' unit [28, 36, 44]. For example, Z3's tactics framework [36] employs a set of rewrite rules and constant-folding heuristics. The success of these techniques underscores the crucial role preprocessing plays in improving solver efficiency. In this work our focus is not on solver efficiency, but on simplification in a dynamic environment to reduce data-transfer times.

Logic-Based Approaches to Formula Reduction. Beyond syntactic preprocessing simplifications, logic-based methods have been explored extensively. Such methods are Quantifier Elimination (QE) and Craig Interpolation, for which engines are commonly exposed in tools like Z3, cvc5 [21], MathSAT5 [31, 32] and SMTInterpol [30].

Surveys highlight QE as a standard preprocessing step for program synthesis, quantified SMT, and CHC solving [39]. An example of a QE algorithm is Fourier-Motzkin elimination [41]. This algorithm eliminates a variable x from a system of linear inequalities by pairing every lower-bound of x with every upper-bound of x, which forms a new inequality that no longer includes x. This can be continued for as long as a variable has a lower- and upper-bound. Generally, QE can produce simplified formulas for DSS, but a previous attempt to implement it failed due to poor solver performance by Z3. Concerning Craig Interpolation, McMillan laid groundwork for generating simplified formulas by applying the Craig-Lyndon interpolation theorem to Model Checking [34, 45, 46, 47]. Interpolants serve as concise overapproximations in proof-based refinement loops, where they power models such as CPACHECKER [25]. For our simplification task they are complicated and costly to use; they add overhead by requiring an unsat proof for each interpolant and they require manual splitting of the formulas into at least two parts **A** and **B**, for which $\mathbf{A} \wedge \mathbf{B}$ has to be unsatisfiable. Since the interpolant **I** is constructed to contradict **B**, if a later block relies on properties found only in **B**, replacing **B** with **I** can cause the entire proof to fail or become an under-approximation. This makes the implementation of Craig Interpolation to simplify SMT formulas in DSS possible, but difficult. We leave this approach for future work.

Another algorithm by Dillig et al. [38] focuses only on the simplification of quantifier-free SMT formulas and is implemented in the software verification tool Ultimate Automizer [40]. The algorithm reduces the size of formulas by removing irrelevant predicates and redundant subformulas. This is done as long as an equivalent formula can be obtained by replacing any atomic formula with true or false. The evaluations of both Dillig et al. and the Ultimate Automizer show that this approach is very effective and lowers analysis times drastically, even though the algorithm itself is costly due to multiple solver calls. This approach is interesting for Multi-processing DSS, since formulas are always quantifier-free and can contain redundant subformulas. An implementation of this algorithm is left for future work as well.

LLMs and Neural Approaches. Recent advances with neural-symbolic methods have shown that LLMs can solve some complex mathematical tasks better than humans and algorithm-based tools due to their pattern recognition abilities. Given the example of integration problems, a neural network trained on graph representations of equations was able to outperform the linear-equation solver Mathematica significantly [42]. Furthermore, OpenAI's GPT-f model found new short proofs for the Metamath formalization language that were accepted by a formal mathematics community [49], showcasing the possibilities of incorporating LLMs into formal mathematical work.

In the realm of verification and solvers, NeuroSAT has shown that neural networks are able to predict properties of propositional formulas, replicating the behaviour of a SAT-solver and implicitly simplifying SAT formulas along the way [52]. Integrating a simplified NeuroSAT architecture into solvers like MiniSAT, Glucose, and Z3 enabled solving of up to 11% more problems than before, proving that neural networks have a justified future in real-world verification problems [51].

These successes highlight the potential of neural approaches to exploit patterns within logical formulas, however, they also exhibit limitations: They are often evaluated in isolated or offline contexts, rather than on real-world tasks, which questions the validity of the results. Further, they mandate coupling with formal-verification steps to check for potential errors.

Positioning our Contribution. As DSS produces block summaries dynamically for each verification run, the degree of simplification by static rewriting or constant-folding is limited. Furthermore, this dynamic environment leads to logic-based simplification methods requiring a lot of implementation effort: QE necessitates rewriting the formulas before simplification (as the formulas are quantifier-free bit-vector logic). Craig Interpolation needs an UNSAT partition, which needs to be created for each summary, and the method can drop variables not shared by A and B. This makes it difficult to reuse the interpolant.
These disadvantages leave LLMs as a justified alternative, for a couple of reasons:

- The pattern recognition capabilities might produce helpful simplifications, speeding up verification.
- An LLM might be able to find simplifications that syntactic or logic-based methods cannot find.
- It is a computationally cheap option, causing no overhead for the actual verification task.
- It is simple to implement, as it does not interfere with any of DSS' internal workings and does not require complicated transformations or algorithms.
- A modular approach enables simple swaps of LLM models and providers, future-proofing the approach.

Pairing this with syntax and overapproxmation checks, we aim produce a simplifier that reduces total verification time and keeps verification sound.

3 Background

Within this section, we explain the underlying data structures, techniques, and systems that are used in our microservice. Among those topics are the SMT-LIB syntax, JavaSMT, and MathSAT (Section 3.1), CPACHECKER and Multi-Processing DSS (Section 3.2), Quarkus, Kubernetes, and Google Gemini (Section 3.3).

3.1 Formal Methods and SMT

Satisfiability Modulo Theories. Multi-Processing DSS uses SMT (Satisfiability Modulo Theories) formulas instead of SAT (Boolean satisfiability) formulas. SAT solvers determine the satisfiability of propositional logic formulas (Boolean formulas). In more complex domains, like software verification, SAT solvers are often exchanged for SMT solvers because their more expressive language makes them easier to use. In most cases, satisfiability under the constraints of some background theory must proven. Background theories can, for example, be the theory of equality, or some theory of arithmetic. The research field devoted to proving this kind of satisfiability is called Satisfiability Modulo Theories (SMT) [24]. The formulas' syntax is defined by the SMT-LIB standard, which was first proposed in 2003 [50] and subsequently refined with the most recent update being version 2.7 [22].

SMT-LIB syntax. In SMT-LIB standard all commands, sorts, and terms are symbolic expressions (s-expressions). S-expressions exhibit a recursive, tree-like structure, where each node either consists of an atom or an expression combining two more S-expressions. Since SMT-LIB is purely optimised for machine parsing, not human readability, an expression always starts with the operator or function name, followed by the operands or function arguments. Normally, an SMT-LIB formula starts with some declarations of variables, constants, or functions, followed by the actual formula contained in an 'assert' block. Listing 3 shows an example SMT-LIB formula. It is a violation condition of the DSS analysis block starting in line 17 of the program in Listing 4. The first line is purely informational. The second line is the declaration of the variable 'x1' as 32 bit-vector. The third line is the actual formula: $x1 = 3$.
The default background theory of formulas in DSS is the quantifier-free theory of arrays, uninterpreted functions, and bit-vectors (QF_AUFBV). Semantically, the formulas describe a path that CPACHECKER followed through a program's Control Flow Automaton (CFA). Since DSS breaks programs down into blocks, a block from line 17-20 of Listing 4 creates the simple formula in Listing 3. We explain the DSS concept in greater detail in the following sections. Other blocks with more

```
1 (set-info :source |printed by MathSAT|)
2 (declare-fun |main::x@1| () (_ BitVec 32))
3 (assert (let ((.def_10 (= |main::x@1| (_ bv3 32)))).def_10))
```

Listing 3: Example SMT Formula

```
1  int main() {
2
3      int x = 0;
4      int y = 4;
5
6      if (y < 5) {
7          if (y == 4) {
8              x++;
9              x++;
10         }
11         x++;
12         y++;
13         y++;
14         y++;
15     }
16
17     if (x == 3) {
18         ERROR:
19         return 1;
20     }
21     return 0;
22 }
```

Listing 4: Corresponding code block

complicated paths create larger and more complex formulas; an example can be found in Listing 5. It describes a path through the nested if-statements of the program in Listing 4 (Lines 6-15). For complex verification tasks, path formulas can grow a lot longer, exceeding lengths of 50000 characters.

JavaSMT and MathSAT5. Due to the vastly different API's of various solvers, we use the unifying JavaSMT API to prevent lock-in effects and enable simpler switching between solvers [19]. Currently JavaSMT supports many popular solvers, like CVC5, MathSAT5, and Z3 [5]. As Multi-Processing DSS is implemented in the verifier CPACHECKER [27], which was originally implemented using MathSAT as backend SMT solver [25], we also implement the TRIMMER's formula validation unit using MathSAT as backend solver. The main reason for this is MathSAT's broad feature set [19], which covers all requirements for the TRIMMER. Also, its longstanding reliability and soundness guarantees, as well as its high performance, make MathSAT one of the best options for the task.

```
1 (set-info :source |printed by MathSAT|)(declare-fun |main::x@3| () (_
    BitVec 32))(declare-fun |main::y@3| () (_ BitVec 32))(declare-fun |
    main::x@4| () (_ BitVec 32))(declare-fun |main::y@4| () (_ BitVec
    32))(declare-fun |main::x@1| () (_ BitVec 32))(declare-fun |main::
    y@1| () (_ BitVec 32))(declare-fun |main::y@2| () (_ BitVec 32))(
    declare-fun |main::x@2| () (_ BitVec 32))(assert (let ((.def_124 (=
     |main::x@4| (_ bv0 32))))(let ((.def_32 (bvadd (_ bv1 32) |main::
    x@4|)))(let ((.def_119 (= |main::x@3| .def_32)))(let ((.def_27 (
    bvadd (_ bv1 32) |main::x@3|)))(let ((.def_117 (= .def_27 |main::
    x@2|)))(let ((.def_104 (bvadd (_ bv1 32) |main::x@2|)))(let ((.
    def_105 (= |main::x@1| .def_104)))(let ((.def_43 (bvadd (_ bv1 32)
    |main::y@4|)))(let ((.def_100 (= |main::y@3| .def_43)))(let ((.
    def_38 (bvadd (_ bv1 32) |main::y@3|)))(let ((.def_98 (= .def_38 |
    main::y@2|)))(let ((.def_95 (bvadd (_ bv1 32) |main::y@2|)))(let
    ((.def_96 (= |main::y@1| .def_95)))(let ((.def_79 (= |main::x@1| (_
     bv3 32))))(let ((.def_97 (and .def_79 .def_96)))(let ((.def_99 (
    and .def_97 .def_98)))(let ((.def_101 (and .def_99 .def_100)))(let
    ((.def_106 (and .def_101 .def_105)))(let ((.def_118 (and .def_106 .
    def_117)))(let ((.def_120 (and .def_118 .def_119)))(let ((.def_107
    (= |main::y@4| (_ bv4 32))))(let ((.def_121 (and .def_107 .def_120)
    ))(let ((.def_111 (bvslt |main::y@4| (_ bv5 32))))(let ((.def_122 (
    and .def_111 .def_121)))(let ((.def_123 (and .def_107 .def_122)))(
    let ((.def_125 (and .def_123 .def_124))).def_125))))))))))))))))))
    ))))))))
```

Listing 5: Complex SMT-LIB formula

3.2 Verification Tools and Model-Checking

CPACHECKER. CPACHECKER is the verifier for C programs in which Multi-Processing DSS is implemented [27]. First published in 2011, the project aimed at accelerating the process of converting a verification idea into actual experimental results, by implementing configurable program analysis (CPA) approaches for several abstract domains [25]. The idea was to simplify and unify, within one tool, many of the complicated steps required for experimental evaluation of a verification algorithm. Another goal of the project was to increase the validity of experimental results by providing comparable environments for each experiment through the CPA concept [25]. CPACHECKER uses abstract-predicate model checking, counter-example validation via SMT, and precision refinement through Craig Interpolation [33] [25]. It also exposes the option to run a CEGAR loop using these techniques. The path formulas shipped to Multi-Processing DSS originate from the validation step of this workflow [26]. We look at this step in more detail when we discuss the workings of DSS, since this step is where we aim to save time through the TRIMMER.

Distributed Summary Synthesis. The original DSS (also 'internal DSS') decomposes a monolithic verification task into a network of smaller, interconnected block analyses, each corresponding to a program control-flow block. The core DSS algorithm wraps CPACHECKER's predicate-abstraction CPA. The workflow is visualised in Figure 3.1, a C program is taken in as input. First, DSS creates a control-flow-automaton from this program, which is transformed into a block graph. From this blockgraph, the

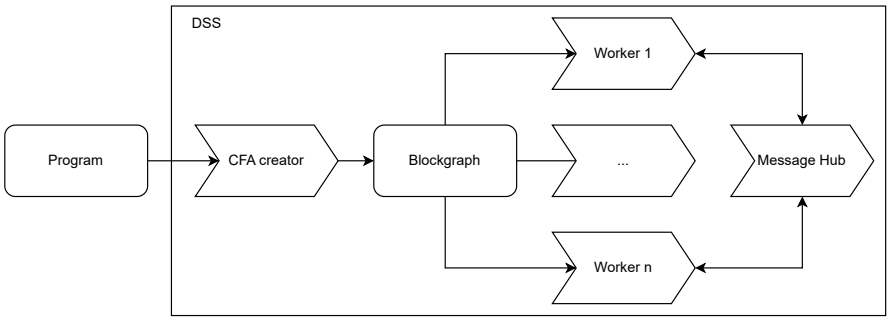


Figure 3.1: Original DSS concept

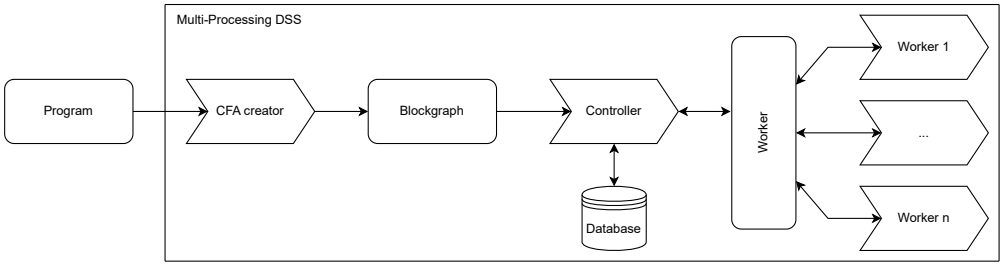


Figure 3.2: Multi-Processing DSS concept

analysis is split into blocks that are analysed by workers. The workers communicate through the message hub.

Each DSS analysis block iteratively synthesises entry summaries (postconditions from predecessor blocks) and exit summaries (violation conditions sent to predecessors, or if no violation conditions exist postconditions sent to successors) through asynchronous refinement. Every time a block receives updated summaries from its neighbours, DSS re-analyses only that block, instead of the entire program. New summaries are computed from that analysis and propagated further, which enables every block to be verified in parallel as independent analysis. Experimental results show that this approach utilises more CPU cores than the traditional single-threaded CPACHECKER analysis, and in return reduces overall response times for tasks with high exploitation potential for parallelisation [27]. A limiting factor to this initial version of DSS is that only one computer, i.e. only one CPU, is used. This puts harsh restrictions on the potential gain through parallelisation, even for modern multi-core processors with a lot of available memory. Therefore, Lemberger et al. proposed a multi-processing version of DSS, which transform block analyses into stateless microservices [43]. The workflow can be found in Figure 3.2. The workflow stays the same up to the newly added Controller service. The Controller is added to orchestrate the verification workflow more efficiently; it maintains the global block graph and routes pre- and postcondition messages to a scalable pool of worker services. These represent the workers from the original DSS concept, though in this

iteration they invoke CPACHECKER in single-block mode to compute summaries. The theory is that decoupling coordination from computation and leveraging containerised deployment, as well as advanced, well-maintained tools like Quarkus [18], Kubernetes [7], and Linkerd [8], enables horizontal scaling and flexible resource allocation.

How Distributed Summary Synthesis extends CPACHECKER. The classic single-threaded workflow is replaced by a message-driven, block-wise refinement workflow: After the abstraction and model-checking phases identify an abstract counter-example path, DSS divides the program's CFA into basic blocks. Then, for each block, the SSA path formula is built and its precondition and post-/ violation condition is published to a shared message database. Whenever new summaries arrive, a worker invokes the CPACHECKER engine, using the new messages as starting predicates [27]. This is also where the TRIMMER submodule can make a difference to the performance of Multi-Processing DSS. Currently, waiting on messages to be loaded to and from the database can take up a large part of the total verification time. With simplified messages, this waiting period can potentially be reduced, making the entire system more efficient.

3.3 Implementation Tools and Google Gemini

Within the current implementation of Multi-Processing DSS and our TRIMMER microservice, we use high-performing, well-maintained tools and frameworks. In the following we take a closer look at the tools used specifically for the TRIMMER.

Quarkus. Quarkus is a Java framework that shifts workload to build time, performs annotation processing, dependency-injection, and REST/gRPC endpoint generation. It provides a rich extension ecosystem (e.g. quarkus-grpc, quarkus-scheduler, quarkus-micrometer) that can be included at compile time. This enables us to produce a JVM image with low startup times, smaller memory footprint, and out-of-the-box support for reactive programming models (Mutiny). Within the TRIMMER, Quarkus packages the microservice as container image, provides the gRPC interface to start and stop, handles asynchronous LLM calls via Mutiny and a REST client, and schedules the trimming cycles [12, 14, 16, 17]. Furthermore, statistics are collected through the Micrometer and OpenTelemetry extensions [13, 15].

Kubernetes and Minikube. Kubernetes is the container orchestration platform that automates deployment. It includes many features, such as service discovery and load balancing, storage orchestration, and self-healing, which we make use of in the implementation of the TRIMMER [6]. The platform originates from the Google application Borg and was open-sourced in 2014, combining over 15 years of experience in running production workloads. Minikube is a tool that quickly sets up a local Kubernetes cluster [9]. We use this tool to set up our cluster whenever we are testing or running the microservice DSS version.

Google Gemini. Google Gemini is a family of multimodal large language models developed by Google DeepMind that are designed to process and reason over text, images, audio, video, and code. The Gemini models have consistenly ranked well

among other AI models on various benchmark tasks, with the latest 2.5-flash version performing better than many of its competitors in the category of fast performance on complex tasks [3].
Developers can access Gemini programmatically via the Gemini API, which exposes REST and gRPC endpoints. Requests can be made through free-form text prompts, which we use for the TRIMMER, structured prompts, and chat-style interactions. Through the API, clients send their prompts and configuration as JSON payloads and receive responses with generated content and metadata as JSON objects. Authentication is handled through API keys - to use the TRIMMER, such an API key must be provided. In contrast to normal chat usage the API lacks additional functionality such as web-search, conversational history management, tool integration, and safety-filtering pipelines. For the use case of the TRIMMER this is an advantage, as the API has lower latency and none of the additional features are required. An idea for future work is to experiment with structured prompts and data.

4 Contribution

The TRIMMER microservice is our key contribution. As a subproject within Multi-Processing DSS, it is divided into four main functional components. First, at the heart of the service sits the MessageProcessor, which can be seen as orchestration layer. Second, there is the MessageRepository, which interacts with the DSS database. Third, there is the LLM client. Fourth, we contribute the Validator. Within this chapter we explain how these units work together to form the TRIMMER by examining one processing cycle. After that we discuss design choices, LLM prompting, and statistics collection. All code can be found in the Gitlab repository of Multi-Processing DSS [11].

4.1 Workflow and Approach

Figure 4.1 shows the detailed workflow of the TRIMMER. A workflow, consisting of six stages, is started through the MessageProcessor. Once enabled, it schedules one workflow every 15s, which is configurable. The MessageRepository fetches messages from the database, from which the MessageParser extracts the SMT formulas. The extracted formula is sent to the LLM with a request for simplification by the LLM client. Upon return, the Validator validates the soundness of the simplification before the MessageRepository writes any new entries to the database. In the following sections we describe this workflow in more detail, following a message through one processing cycle.

4.2 Fetching messages

```
1 SELECT id, runId, blockId, messageType, messageContent, timestamp,
2        handled, wasShortenedTo, isShortenedFrom
3 FROM ${table.sqlName}
4 WHERE   wasShortenedTo IS NULL
5 AND     isShortenedFrom IS NULL
6 ORDER BY timestamp DESC
7 LIMIT ?
```

Listing 6: Fetch messages SQL statement

The first step of a workflow is to read a configurable amount of messages - the default hereby is 10 - via the MessageRepository from two of the database's ta-

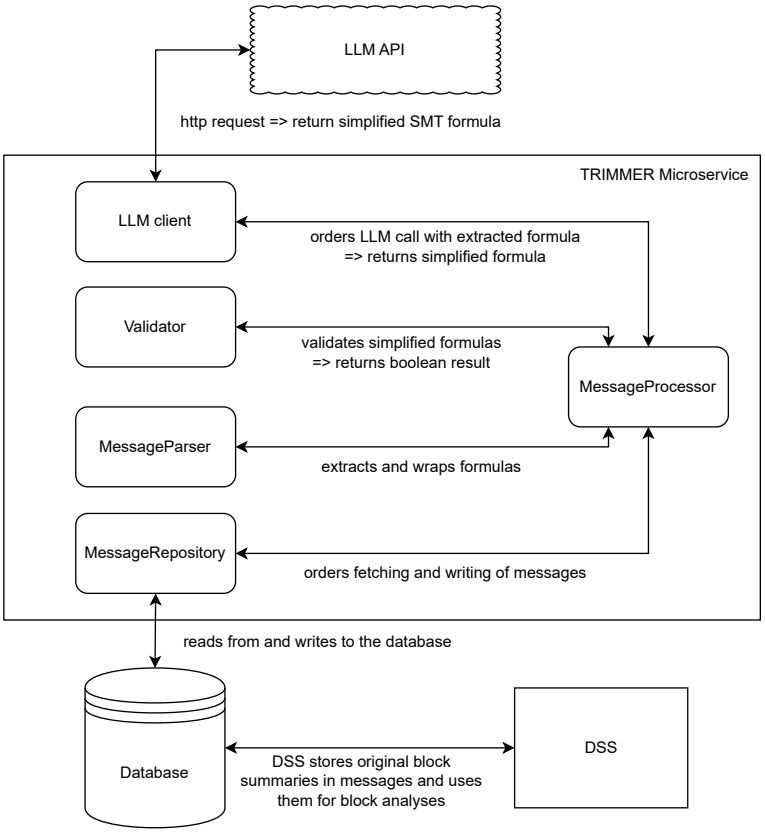


Figure 4.1: Workflow of our approach

```
val workDispatcher = Dispatchers.IO.limitedParallelism(parallelism)
supervisorScope {
    messages
        .map { msg ->
            async(workDispatcher) {
                if (processSingleMessage(table, msg)) {
                    messageCount.addAndFetch(1)
                }
            }
        }.awaitAll()
}
```

Listing 7: Limited parallelisation for message processing

bles - 'postconditionMessages' and 'violationCondtionMessages'. These tables are processed in parallel with Kotlin's coroutines to minimise waiting periods.
The MessageProcessor can be seen as a controller unit or orchestration layer. Upon starting, it delegates the MessageRepository to read from the database. This, in turn, executes the SQL statement from Listing 6. The table name is passed as a function

parameter '$table.sqlName' in Line 3. The statement grabs all the available fields of a message (lines 1,2), and makes sure to only grab original, unsimplified messages (lines 4,5), which we explain later. Ordering by 'timestamp' (Line 6) ensures that the most recent messages are simplified during every cycle. The amount of messages read at once is limited through a config property (Line 7). After executing, each message is instantly converted to a Kotlin data class object to minimise database communication. The MessageRepository returns a list of the fetched messages to the MessageProcessor.
Now that the MessageProcessor has messages as pure Kotlin data object, we start processing in parallel, as can be seen in Listing 7. The 'workDispatcher' (Line 1) coordinates Kotlin's coroutines; we invoke it with a limited amount of available threads through '.limitedParallelism(parallelism)'. The amount is configurable through a config property. The 'supervisorScope' (Line 2) acts as a safety net: Even if one thread is interrupted, or returns with an error, the other threads in the batch can finish their work correctly without crashing. For every message, the process is wrapped in an 'async' call to actually enable parallelisation (Line 5). We add a thread-safe counter that we increment with 'addAndFetch(1)' (Line 7) for every successfully processed message. Before moving on with a new batch of messages, we make sure to wait for all occupied threads to finish (Line 10).

4.3 Extracting the SMT formula

```
1  {"uniqueBlockId":"MV2",
2  "targetNodeNumber":66,
3  "type":"ERROR_CONDITION",
4  "payload":"{\"org.sosy_lab.cpachecker.cpa.callstack.CallstackCPA\":
5      \"69.__VERIFIER_assert\",
6  \"org.sosy_lab.cpachecker.cpa.functionpointer.FunctionPointerState\":
7      \"\",
8  \"org.sosy_lab.cpachecker.cpa.predicate.PredicateCPA\":\"
9      (set-info :source |printed by MathSAT|)
10     (declare-fun |__VERIFIER_assert::cond@1| () (_ BitVec 32))
11     (assert (let (
12         (.def_10 (= |__VERIFIER_assert::cond@1| (_ bv0 32)))
13     ).def_10))\",
14 \"org.sosy_lab.cpachecker.cpa.predicate.PredicatePrecision\":
15     {\"locationInstances\":{},
16     \"localPredicates\":{},\"functionPredicates\":{},\"global\":[]},
17     \"property\":\"CHECKED\",
18     \"sound\":\"SOUND\",
19     \"precise\":\"PRECISE\",
20     \"first\":\"true\",
21     \"origin\":\"N66->N68,N68->N69\"}",
22     "timestamp":"2024-10-16T07:09:51.883364215Z"}
```

Listing 8: messageContent String

The next step is to parse each message. For this, the MessageProcessor delegates extracting the SMT formula to the MessageParser. The formulas sit inside the 'messageContent' field, of which an example can be found in Listing 8. It always contains

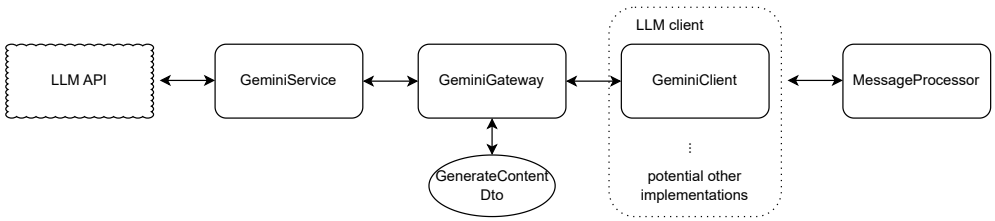


Figure 4.2: LLM client implementation

the top-level keys 'uniqueBlockId' (Line 1), which is the ID of the block that produced this message, 'targetNodeNumber' (Line 2), which contains routing information for the analysis, a 'type' (Line 3), which specifies the type of condition a message contains, and the 'payload' (Line 4), which holds the SMT formula under the key 'org.sosy_lab.cpachecker.cpa.predicate.PredicateCPA' (lines 8-13). The SMT formula is the only information of interest to the TRIMMER. The MessageParser disregards all other information and returns only this formula as String.

4.4 Simplification

After obtaining the stripped SMT formula as String, the MessageProcessor continues with the third step - the actual simplification. It requests a shortened formula through the LLM client. The request is managed by an abstracted REST client interface under the hood that uses three different abstraction layers: GeminiClient, GeminiGateway, and GeminiService (Figure 4.2). GeminiClient is our concrete implementation of the LLM client interface. It is injected at build time by Quarkus, so the MessageProcessor has no knowledge of the concrete implementation - this is symbolised by the arrow connecting only the LLM client and the MessageProcessor. The request is then passed to the GeminiGateway, which ensures proper formatting: The request String is wrapped into the JSON format Gemini's API actually expects, and Gemini's reply is unwrapped so that only the String response is returned to the MessageProcessor. For this purpose we added the data transfer object 'GenerateContentDto', which represents Gemini's request and response format accurately through multiple simple Kotlin data classes. The GeminiService represents the actual REST client that communicates with the API. It exposes the 'generateContent' method which takes in a configurable API key and the formatted request. The API key has to be set as environment variable inside 'dss-trimmer.deploy.yaml' for deployment. With this architecture the MessageProcessor only calls one interface method, making the underlying LLM technology totally interchangeable.

4.5 Validation

To make sure the response is a correct and meaningful simplification of our original SMT formula, we employ a validation step as fourth step in our workflow. Again in its role as orchestrating layer, the MessageProcessor calls both methods that

```
interface Validator {
    fun isSyntaxValid(formula: String): Boolean

    fun isOverapproximation(
        originalFormula: String,
        simplifiedFormula: String,
    ): Boolean
}
```

Listing 9: Validator interface

perform the checks. The Validator exposes two methods that override the Validator interface (Listing 9), one to check the syntax of the simplified formula and another to check whether the simplified formula is an overapproximation of its original counterpart. We discuss later why overapproximation is a reasonably meaningful simplification in this case. The Validator implements these checks using the SMT solver MathSAT5 through the JavaSMT API layer. Checking syntax is trivial: We parse the formula into solver readable format with the inbuilt parse function. If that succeeds without throwing any exceptions we know that the LLM reply must have correct SMT-LIB 2 syntax. Continuing with the overapproximation check, we build the negation of $original \Rightarrow simplified$:

$$\begin{aligned} & \neg(original \Rightarrow simplified) \\ \Leftrightarrow\ & \neg(\neg original \vee simplified) \\ \Leftrightarrow\ & original \wedge \neg simplified \end{aligned}$$

This means we tell MathSAT5 to prove $original \wedge \neg simplified$ is unsatisfiable. If this returns true, the original formula implies the simplified counterpart and our check returns true. In any other case, also exceptions or errors, we return false.

If the validation step fails, i.e. one of our validation methods returns false, the MessageProcessor exits the processing cycle of this message early, skipping any writes to the database. This ensures that only fully validated formulas are ever written to the database.

4.6 Replacing the SMT formula

Assuming the MessageProcessor has received a valid and meaningful simplified formula, we must now replace the SMT formula inside the payload before writing a new entry to the database. This makes up the fifth step in our workflow. For the verification with DSS it is essential to make sure that all other information the original message contained remains unchanged. Wrapping the formula back into the message is completed by the MessageParser. Looking back at Listing 8, it parses the entire field and only overwrites the formula under 'org.sosy_lab.cpachecker.cpa.predicate.PredicateCPA' inside of the payload. The function then returns the updated messageContent as String.

```
INSERT INTO ${table.sqlName}
    (runId,
    blockId,
    messageType,
    messageContent,
    timestamp,
    handled,
    isShortenedFrom)
    VALUES (?, ?, ?, ?, ?, ?, ?)
```

Listing 10: Insert SQL statement

```
UPDATE  ${table.sqlName}
    SET     wasShortenedTo = ?
    WHERE   id = ?
```

Listing 11: Update SQL statement

4.7 Updating the database

At this point the MessageProcessor has a valid simplified SMT formula wrapped inside the messageContent field. The last task of the TRIMMER is to update the database accordingly. For this purpose, we have introduced two new fields to the database, 'wasShortenedTo' and 'isShortenedFrom'. Both fields have a default value of null. They have the type Integer and reference the ID of another entry inside the same table of the DSS database. The field 'wasShortenedTo' should only ever contain values for messages that have a simplified counterpart inside the table, while 'isShortenedFrom' should only ever contain values for messages that have an original counterpart inside the table. This way both original and simplified always carry a reference to each other, which not only simplifies our experimental setup but should also be useful in the future. The MessageRepository takes over the database writing. It exposes a method to write the messages to the database, which prepares two SQL statements and executes them in the same commit to ensure database atomicity. Listing 10 shows the insert statement for the new entry containing the simplified SMT formula. The field 'isShortenedFrom' is set to the original message's ID here. Listing 11 shows the update statement for the existing entry of the original message. We only modify the field 'wasShortenedTo', inserting the newly created ID key from the insert statement.
This step concludes one iteration of the workflow, which is repeated as soon as the scheduler interval has passed or is stopped by invoking the MessageProcessor's stop() method.

4.8 Design choices

Layered and separated structure. Our implementation choices strictly follow the principles of separation of concerns and a layered architecture. At the highest layer we implemented the TrimmerService, which is the communication layer between

protobuf service (the client request) and the MessageProcessor. This restricts access, since, so far, only two methods are exposed at runtime, start() and stop().

A layer below sits the MessageProcessor, which can be thought of as the coordination (service) layer of the TRIMMER. It has three main jobs: First, it schedules the workflow on a fixed cadence and guards against incorrect usage (e.g. repeatedly calling start/stop). Second, it coordinates the workflow, as we discussed earlier. Third, it enables multiple workflows to run in parallel. Furthermore, this layer makes it simple to work on the trimmer. As most of the functional tasks are configured via interfaces, this layer only really has to change when you want to change the way a workflow behaves. Switching SMT solvers or LLMs leaves this layer and the rest of the service completely unaffected.

Below the MessageProcessor sit the separate logical modules that perform functional tasks. These are mainly the MessageRepository, the MessageParser, the Validator, and the LLM client. The MessageRepository works with any SQL-based datasource that is configured via Quarkus, since we only use standard SQL queries. A shortcoming is that the MessageParser, in combination with the MessageRepository, only works with this exact type of message formatting. If this changes in the future, we have to refactor those two modules. As a change in message formatting should not be frequent, since it also results in database changes, we believe this is not a big issue. Also, adapting these two classes should not be a big challenge, due to their limited and separated functionality.

In contrast, the two logical modules that are more prone to frequent switches are fully modular and have clearly defined contracts for their behaviour in form of interfaces. These are the LLM client and the Validator. Essentially following the Adapter pattern, the GeminiGateway and GeminiService adapt Google's REST API into the LLM client's interface, and the Validator deals with the JavaSMT/MathSAT5 details under the hood of the Validator interface. These modules allow for easy unit testing, mocking, and swapping implementations without changing core logic anywhere else.

Supporting the microservice are data classes, i.e. Message, MsgTable, and GenerateContentDto. These classes represent the data layer, which reflects change in information and is passed as communication object between different modules. The Message class hereby represents a message itself, exactly as it can be found within the database. This makes it easy to store all the fetched data in cache and manipulate it without repeated database accesses. MsgTable is a simple allowlist for the tables we can process messages from. It defines access clearly and makes the table names easily interchangeable. Finally, GenerateContentDto represents a data transfer object, which is why it consists of a cluster of simple data classes. It enables extremely simple wrapping of requests to and unwrapping of responses from the Gemini API JSON format.

Coroutine Pipeline. We use structured concurrency for our service, for two main reasons. It enables us to reproduce our experiment settings consistently, and it enables a configuration tailored to the specific problem and system setup. Structured concurrency means that we set a configurable limit to the coroutines that the dispatcher may start. This ensures that our experiments always run with the same

amount of parallel threads, independent of demand, LLM throughput, and message list size. Omitting this can create a threat to the internal validity of the experimental results. Therefore, we want to be able to control this at least in the experimental setting. For larger scale deployment, the limit can simply be set to a higher value. Nevertheless, as API requests cost money, it might be useful to keep a limit on the processing rate regardless.

Database Additions. The MessageRepository follows the repository pattern as single endpoint for database communication, executing all SQL statements. Writing simplified messages to the same database at the end of our workflow should not interfere with any of DSS' core logic, as this should not be adapted. Furthermore, we should prevent processing the same original messages more than once, and prevent processing an already simplified message again. This is achieved by adding the two new columns to the existing DSS database schema, 'wasShortenedTo' and 'isShortenedFrom'. These references to the corresponding counterpart make correct processing simple, without messing with any of the other information. For processing we only ever fetch messages where both of these fields are null, as can be seen in Listing 6; meaning such a message neither has a simplified counterpart, nor is simplified itself. For the analysis with DSS we create a new configuration property inside of the scheduler microservice, which handles message coordination at analysis level. The property, a boolean flag, switches between forcing DSS to use simplified messages whenever possible, and running DSS as before with only the original messages. Per default, the flag is set to the latter setting.

Configuration-Driven Behaviour. A large part of the service can be configured inside the TRIMMER's 'application.properties' file, such as batch size, limit to parallelism, timeout limit, LLM provider, and scheduler interval. This not only enables us to be consistent with experiments, but also makes the service highly configurable for the task and different kinds of deployment. For example, with these properties, we can decide if the service runs sequentially on a batch size of one, i.e. it really processes one message at a time, which is great for bugfinding and troubleshooting; or if it should run with a batch size of 100 and 100 parallel threads, if it has access to a multicore CPU. Configuring the LLM provider enables adding other LLMs, while keeping all original variations. The annotation as build property does not require any code changes apart from adding a new implementation of an LLM client. With that in place, a rebuild enables the new LLM.

Error Handling. We take several precautions to make the TRIMMER as robust as possible, but also let it crash if something really goes wrong. A frequent source of errors and inconsistencies is parallel processing. Using Quarkus, many of the main problems with parallelism can be circumnavigated easily. The usage of 'supervisorScope' around our parallelisation calls ensures that a failure to process one message or one table does not result in collapsing the entire processing batch. Concretely, it allows processing to continue even after one thread throws any exception. Also, to guard the entrypoint start()/stop() methods, we use Atomic flags to prevent race conditions. The gateway itself bridges the reactive HTTP calls into coroutines, so that no other threads attempting to call the LLM are blocked. As final step of the workflow, the writes to the database happen in one single commit, which ensures

```
I have the following SMT formula in SMTlib2 format: - SMT formula -
It describes a C program in SSA form. Provide a smaller formula, you
have to guarantee that the new formula is implied by the original
formula! Also it is absolutely vital that the new formula is in valid
SMTlib2 syntax! Make sure you do not overlook any necessary
declarations! If you cannot find a result, return 'No smaller formula
found'. Do not add any line breaks to the output!
```

Listing 12: More vague prompt, yielding better results

```
I have the following SMT formula in SMTlib2 format.
This SMT formula is used for software verification and resembles a
post- or violation condition of an analysis block. Your job is to
provide a simplified, shorter formula that can be used in place of the
original formula to complete the verification run. You have to
guarantee that the simplified formula is at least implied by the
original! Also it is absolutely vital that the new formula is in valid
SMTlib2 syntax! Return only the formula, without line breaks, in valid
SMTlib2 syntax! Validate your own result according to this checklist,
you should be absolutely certain that your simplification fulfills
these criteria!:
1. The simplified formula is 100% valid SMTlib2
syntax!
2. The simplified formula is definitely implied by the original
formula!
3. A verification run would achieve the same result when using the
simplified formula instead of the original formula!
----- inserted SMT formula -----
```

Listing 13: More detailed prompt, yielding worse results

database atomicity. All this creates fully parallelised processing capabilities, without common issues like race conditions and failed atomicity.
Furthermore, we make sure to catch many non-vital exceptions, such as parse errors, serialisation exceptions, and SQL and validation failures. We log these exceptions comprehensively for simple troubleshooting and leave the message that caused the issue unhandled, so further attempts at simplification may be made.

4.9 Prompting the LLM

Preparing this thesis project, we conduct many experiments on SMT formula simplification with LLMs. There are two main aspects that have to be validated before implementing within Multi-Processing DSS. First, we find the LLM model best suited for the task, combining precise formula simplifications with low latency and response times. We aim to achieve good scalability and high throughput, while generating high quality results for the verification tasks. Second, we find a prompt that instructs the LLM to deliver us exactly the responses we wanted.
We test OpenAI's o1-preview, o1-mini, o1, o3-mini, o3, as well as Google Gemini's 1.5 flash, 1.5 pro, 2.0 flash, 2.5 flash, 2.5 pro, and Deepseek's R1 models. Overall, the models that take time to 'think' (e.g. all of OpenAI's 'o…' models, Gemini's '-pro'

models) produce the highest quality results, but at the cost of taking about ∼1 minute for each reply. This contradicts our high throughput ambitions, which is why we look for a compromise between speed and accuracy. In the end, we decide on Gemini 2.5 Flash, the newest and most advanced model of Google's Flash series. This model performs well in our experimental tests by hand, while keeping response times closer to ∼1/2 seconds. For the prompt we go through several iterations to find the best solution. Counterintuitively, a more detailed and accurate prompt often delivers much worse results than a more vague, but expressive version. For example, the prompt in Listing 13 delivers worse results than the prompt in Listing 12 despite the first providing much more precise orders. It is possible that this is linked to a recently discovered phenomenon: Large Reasoning Models (which all of OpenAI's 'o...' models, Gemini's 2.* models, and Deepseek's R1 models are) face a complete accuracy collapse beyond certain complexities. This leads researchers to believe that these models are not deeply reasoning, but rather doing very advanced pattern matching [53]. Looking at our more complex, precise prompt, we can hypothesise that the task described is very complex, maybe too complex for pattern matching. Our simpler prompt, on the other hand, does not make the LLM aware of this complex context in the same detail, which potentially allows for better pattern matching performance. Further research has to be conducted.
Obviously, the performance in these intial experiments is only measured by the syntax and overapproximation checks; whether the verification runs can actually return the same results is evaluated later. Therefore, it remains questionable if the more vague prompt actually yields good results for verification.

4.10 Statistics Collection and Start Script

Mainly for our experiments, but also for ease of further development and LLM comparison, we collect statistics via Micrometer, which we include into the comprehensive DSS Grafana dashboard. We collect the LLM request count, differentiated by successful and unsuccessful LLM requests, the number of successful and unsuccesful overapproximation and syntax checks, as well as the ratio of the length of a simplified formula compared to the length of its original counterpart. From this information we plot two diagrams over a time axis. The first diagram includes three graphs, the total LLM requests per second, the wasted LLM requests per second, and the successful LLM requests per second. The second diagram goes into more detail on the wasted LLM requests. We plot the failed syntax checks per second, the failed overapproximation checks per second, the LLM request timeouts per second and the amount of times, where the LLM returns 'No smaller formula found', per second. This diagram allows for comparisons between different AI models in the future, making it easy to see what part of the task a specific model fails or excels at. Furthermore, we create two gauges that show the average length ratio of simplified formulas compared to their original counterparts, as well as the overapproximation success ratio, which compares the amount of successful overapproximations against the amount of unsuccessful overapproximations. The length ratio value gives us a good overview of the AI model's performance. Moreover, it shows us whether

simplified formulas are longer than their original conterparts, which would directly contradict the purpose of this microservice. Lastly, we add two diagrams plotting the latencies of SMT solver calls and LLM requests, which enables us to identify potential bottlenecks and compare solver and LLM performance in the future.

Starting a Verification Run with the Trimmer. We provide a new script to start a verification run and the TRIMMER. This is handy for our experiments and can be found in the Gitlab repository [11] as 'start-experiment-trimmer.sh'. It extends the usual 'start-run.sh' shell script, and works completely automatically, if the pods 'connector', 'trimmer', and 'postgres' are port-forwarded. Apart from that, usage is the same as with the standard script, and more information can be found in the repository. To make sure Multi-processing DSS actually has the ability to use some of the simplified messages, the config property 'dss.analysis.filterSimplifiedMessages' has to be set to true before (re-)building the project.

5 Evaluation

For our evaluation, we are interested in the of impact of simplified messages on the correctness and verification time of Multi-Processing DSS. Moreover, we benchmark the TRIMMER'S performance itself using multiple different configurations.

5.1 Research Questions

In the following, we conduct multiple experiments to answer these research questions:

RQ 1. How much shorter are the formulas produced?
RQ 2. How often does the TRIMMER'S simplification succeed?
RQ 3. How many API requests are wasted due to incorrect LLM responses?
RQ 4. Does the TRIMMER improve the verification time of DSS?
RQ 5. Are there cases where the TRIMMER leads to worse results?

5.2 Evaluation Setup

Technical Setup. The benchmarks run on a machine using Ubuntu 24.04.2 LTS (GNU/Linux 6.8.0-58-generic x86_64) with an AMD EPYC 7713 64-Core Processor (256 CPUs) and 2 TB of total memory. Multi-processing DSS runs on a minikube cluster, which is configured with 128 CPUs and 128 GB of memory. We use BenchExec in version 3.30 [1, 2] to run entire task sets automatically and to measure verification walltime. We configure BenchExec to use either the standard shell script or our new one.

Experiments. To make our results comparable, we produce baseline results with standard Multi-processing DSS, as well as experimental results where the TRIMMER continuously simplifies messages from the database. For experimental runs, we enable the newly added config property that allows Multi-Processing DSS to use simplified messages.

Choice of Tasks. Evaluation happens on a selected subset of tasks from sv-benchmarks in the reachability category. We choose 41 tasks that internal DSS can solve and take internal DSS more than 60 s to solve. This gives Multi-processing DSS the best possible chance of using simplified messages. The task set and all unprocessed results can be found in the Multi-processing DSS data repository [10]. For proof of

Table 5.1: Trimmer experimental configurations

Configuration	Amount of Parallel Threads	Batch Size
Minimal	8	10
Moderate	20	20
Proof of Concept	80	80

concept runs with a high volume of LLM requests we pick a subset of only two tasks, due to high costs.

TRIMMER settings. We experiment with three different configurations for the TRIMMER, which can be found in Table 5.1. This enables us to evaluate the effect of volume and speed of simplifications on Multi-processing DSS. Between experiments, we vary the amount of parallel threads the TRIMMER is allowed to use, and the batch size of messages that the TRIMMER processes in one cycle. The settings for cycle frequency (15 s) and LLM request timeout (20 s) stay the same for all experiments. A 15 s cycle frequency ensures we do not exceed per-minute API-request limits, while a 20 s timeout ensures that the TRIMMER continuously processes the most up-to-date messages. Also, we keep the same AI model (Google Gemini 2.5 Flash [4]) and the same SMT solver for all experiments (MathSAT5 in version 5.6.11).

5.3 Evaluation Results

RQ 1 (Amount of length reduction).
Evaluation Plan: We collect multiple metrics directly from the TRIMMER: the average length of all original formulas, the average length of original formulas that were successfully simplified, the average length of simplified formulas, and the average ratio of simplified length compared to original counterpart length. Comparing these values, we can answer this question directly.

Results: For the minimal configuration Table 5.2 shows the average length of original formulas that were simplified, the average length of simplifications, and the resulting ratio. Averaged over all requests the length of an original formula that is successfully simplified by the TRIMMER is about 437 characters, the average simplification length is about 178 characters, and the average ratio is 41 %. Figure 5.2 visualises the amount of analysis blocks, amount of original messages, and the average length of original messages for every task (for a baseline run). All scales on the x-axis are logarithmic for better visualisation. The amount of analysis blocks and the amount of original messages will become important later, but for this research question the average length of original messages is interesting. We find that for most tasks, the average original formula that is simplified by the TRIMMER is orders of magnitude shorter than the overall average formula. Figure 5.2 shows this in comparison to Table 5.2.
The box-plot diagram in Figure 5.1 visualises the distribution of the average ratios over all tasks. The highlighted tasks resemble the task with the maximum average

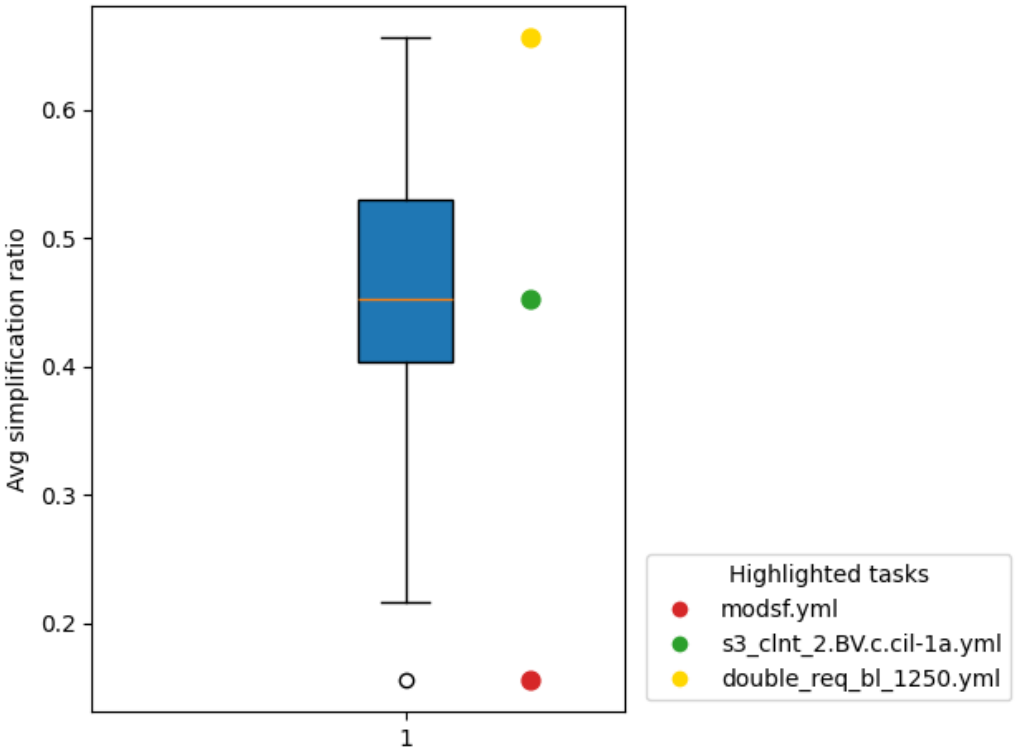


Figure 5.1: Boxplot of the average simplified lengths per task

ratio (yellow), a task with an average ratio around the median (green), and the task with the minimal average ratio (red). The blue box shows that the 25 % quantile starts around 0.4 and the 75 % quantile reaches around 0.53, with the median orange line sitting at around 0.45. Interpreting the diagram, we find that many of our runs achieve consistent simplification lengths within the blue box and only the red highlighted task is categorised as an outlier at under 1.5 IQR. We also see that the whiskers cover a large spectrum, reaching 0.22 on the lower end and 0.66 on the upper end. This tells us that the spread of ratios is rather large, showing that simplification is dependent on the task.

In summary, we find that the average length of a simplified formula compared to its original counterpart is 41 %. This means that the formulas on average are 59 % shorter, which shows that the TRIMMER works well. This value is consistent over multiple runs. The comparison between average length of all originals and originals that were simplified shows that the TRIMMER prefers to simplify shorter formulas. We believe this happens as longer messages take the LLM more time to process, which results in more frequent timeouts. We evaluate the effect of timeouts in greater detail when answering *RQ 2*.

RQ 2 (Success of simplifications).
Evaluation Plan: Again we collect metrics directly from the TRIMMER: the amount of overapproximation errors, the amount of syntax errors, the amount of request timeouts, and total amounts of successful and unsuccessful simplifications. With this data, we can evaluate how many simplifications are successful, and differentiate which errors occur most frequently.

Results: Table 5.3 shows the total, successful, and unsuccessful LLM requests per task, as well as the resulting usage of simplified messages during analysis. To answer this research question, we focus on the total, successful, and unsuccessful columns of Table 5.3. An LLM request is categorised as successful, if a simplified message was written to the database as result of the request. If that is not the case, a request is categorised as unsuccessful. To decide whether a simplified message is used during the analysis, we query the 'Requests' table of the database, which contains all message IDs that were distributed to any worker for a given verification task. For the minimal configuration, the data shows that the ratio of successful to total LLM requests is about 40 %. This is consistent with the other runs we have conducted.
Table 5.5 shows the request comparison table for moderate settings. Before the request limit is reached, the ratio of successful LLM requests compared to total LLM requests experiences a slight drop to 33 % (Minimal Settings: 40 %). With the proof of concept settings (Table 5.7), the same ratio experiences an increase to 45 %. Across all configurations there are large variations of total LLM requests between each task.

Table 5.4 shows a more detailed summary of the unsuccessful requests for minimal settings. Overapproximation errors are incremented every time a simplified formula has correct syntax, but fails the overapproximation check. Syntax errors are incremented each time a simplified formula has incorrect syntax. Both of these values are consistently low throughout all tasks. With minimal settings, 14 overapproximation errors and 89 syntax errors amount to 4.8 % of the 2 337 unsuccessful requests. A much larger portion can be attributed to timeouts. With 1 868 timeouts they make up 80 % of 2 337 unsuccessful LLM requests. The remaining 15 % of unsuccessful LLM requests are caused by other undocumented errors.
Table 5.6 shows the detailed summary of unsuccessful requests for moderate settings. We find that overapproximation errors (26) and syntax errors (85) only amount to 1.6 % of unsuccessful requests (6 882). Timeouts (4 092) make up 59.5 % of unsuccessful requests. A much larger portion of 38.9 % of all unsuccessful requests returns with some undocumented error. Table 5.8 shows the detailed summary of unsuccessful requests for proof of concept settings. Here, overapproximation errors (1) and syntax errors (20) only amount to 2.6 % of unsuccessful requests (769), while timeouts (648) make up 84.2 %. Undocumented errors only account for 13.2 %.

Overall, we see that the total amount of request varies strongly between tasks. Figure 5.3 visualises all the walltimes we gathered for every run differentiated by baseline runs (red) and experimental runs (blue). The tasks are sorted on the y-axis by increasing median walltime and the x-axis uses a logarithmic scale for better visualisation. In summary, it shows that walltimes vary a lot between tasks and

between runs. This is both the case for runs with TRIMMER and without. Comparing Table 5.3 to Figure 5.3 we find that tasks with longer verification time enable the TRIMMER to process more messages. Furthermore, we find variations in the ratio of successful to total LLM requests between 33 % and 45 %. The data shows no immediate reasons for this, but we attribute this fact mostly to variations in LLM responses between runs. In summary, this shows that the TRIMMER is not an efficient tool for formula simplification. We believe this inefficiency can be strongly reduced, which we refer to in Chapter 6.

Concerning the distribution of errors that cause a request to fail, we find that the TRIMMER shows great performance on overapproximation and syntax errors. With values between 1.6 % and 4.8 % these syntactic and semantic errors stay at a very reasonable, low threshold. This is consistent with other runs we conducted. Timeouts are an issue with this approach, and affect the efficiency of the TRIMMER. We also believe that they are the main reason why the TRIMMER simplifies shorter than average messages for each task (see answer to *RQ 1*). Timeouts are caused by the 20 s timeout configuration we keep fixed for our experiments. If we increased the timeout limit, more requests would return successfully - hypothetically with simplifications of longer messages. But the fact that the timeout limit is reached so often in the current configuration also shows: A timeout limit is necessary, otherwise the TRIMMER would spend too much time waiting on simplifications that arrive too late to be used by Multi-Processing DSS. Efficiency gains must be sought elsewhere.

RQ 3 (Wasted API Requests).
Evaluation Plan: Compare total requests to unsuccessful requests.

Results: The answer to *RQ 2* mostly answers this question as well. Table 5.3 shows that 60.4 % of total LLM requests are unsuccessful with minimal settings. Table 5.5 shows that 67.5 % of total requests are unsuccessful with moderate settings, only counting the tasks where request limits were not yet exceeded. Table 5.7 shows that 54.9 % of requests were unsuccessful with proof of concept settings. The reasons for this and detailed summaries were explained in our answer to *RQ 3*.

These statistics underline an underestimated factor of LLM simplification: Expenses. For a run with moderate settings on all 41 tasks, Google bills about 160 €. This means that continuous formula simplification at the moment is not the feasible, simple alternative that we wanted to implement for Multi-Processing DSS.

RQ 4 (Impact on Verification Time).
Evaluation Plan: BenchExec records walltimes for all of our tasks. We can compare these walltimes between baseline runs and experimental runs to evaluate the impact of the TRIMMER on the verification time. Moreover, we collect the amount of simplified messages used during each analysis. This allows us to validate if improvements are caused by the TRIMMER.

Results: Table 5.3 shows the LLM requests per task differentiated by their success and the usage of simplified messages for the minimal configuration. We can see that the usages stay very low (62 messages of 1 534 available messages over 41 tasks). From the 'successful' column of Table 5.3, we find that the amount of available simplified messages is low compared to the total amount of messages from Figure 5.2. We conduct experiments with moderate and proof of concept configurations in order to increase available simplified messages, which we expect to increase the amount of usages.

Table 5.5 shows the LLM requests per task differentiated by their success and the usage of simplified messages for the moderate configuration experiment. After the task 'gcd_2+newton_2_2.yml' completes, the TRIMMER reaches the daily request limit of the Gemini API (10 000), which is why for all remaining tasks of the set no more messages are simplified. This problem occurs consistently over multiple runs. From the available results we can see that the amount of available simplified messages ('successful') has increased dramatically, yet the usages per task stay very low or zero for all tasks except 's3_clnt_2.BV.c.cil-1a.yml' (s3 in the following). For s3, the total amount of original messages is 10 896. A usage of 66 simplified messages is still very low in comparison.

We conduct experiments with proof of concept settings (Table 5.1) to validate that the amount of available simplified messages is the limiting factor for usage during analysis. To avoid exceeding request limits we only conduct this experiment with two tasks: 's3_clnt_2.BV.c.cil-1a.yml' (s3) and '43_1a_cilled_ok_nondet_....yml'[1] ('43_1a' in the following). Figure 5.2 highlights why these specific tasks were chosen: First, they have a high amount of analysis blocks (s3: 149, 43_1a: 91). This means that many blocks communicate with each other by exchanging a lot of messages, which gives the analysis more opportunities to use simplified messages. Second, the amount of original messages available is very high (s3: 8 500 and 43_1a: 9 552), which gives the TRIMMER plenty of opportunities for simplification. Third, the average length of original messages is very high (s3: 14 858 characters, 43_1a: 26 758 characters), which means that using simplified messages can have a greater impact. The last reason lies in the observed walltimes for these tasks. Figure 5.3 shows that s3 and 43_1a have consistently high walltimes, which enables the TRIMMER to produce many simplified messages during the proof of concept experiment.

Table 5.7 shows the LLM requests to simplifications used comparison for the proof of concept configuration. We find that with this configuration the amount of available simplified messages does not increase a lot compared to Table 5.5. Furthermore, the amount of simplifications used during analysis stays consistently low over multiple runs; in this example run at 1 (43_1a) and 4 (s3). This disproves the hypothesis that more available simplified messages simply induce Multi-processing DSS to use more of these messages. We have one final hypothesis we want to validate: The TRIMMER'S

[1] 43_1a_cilled_ok_nondet_linux-43_1a-drivers–leds–leds-regulator.ko-ldv_main0_sequence_infinite_withcheck_stateful.cil.out.yml

simplifications arrive too late for the fast-paced analysis of Multi-Processing DSS.

We conduct one last experiment to validate our hypothesis that the TRIMMER's messages simply arrive too late to be relevant for the analysis. To eliminate the time factor, we complete a baseline run with the task s3 and then let the TRIMMER simplify a large part of the original messages. After that we remove all the messages with a timestamp before the median timestamp (∼1 300 messages). This eliminates many original messages and leaves mostly simplified messages in the database. A query validates that there are still 1 253 simplified messages left in the database, which we update to match the runID of the next run. We then start the verification of task s3 again, with the usage of simplified messages enabled for DSS. In theory, if only the time delay is an issue, DSS should now use simplified messages. But that is not the case, the counter for simplified messages used stays zero. This disproves our hypothesis that the time delay of the TRIMMER's simplification is the only hindering factor for usage during analysis. Instead there can be multiple other issues that the complex machinery of Multi-Processing DSS causes. We will refer to them in Chapter 6.

Overall, we find that the amount of simplifications used during analysis is too low to have any meaningful impact on the verification time of Multi-Processing DSS. We also find that the TRIMMER is capable of producing many correct simplifications in short time with higher throughput settings. With the current configuration of Multi-Processing DSS, these do not seem to be used systematically.

RQ 5 (Potential Negative Impact of the TRIMMER).
Evaluation Plan: Compare walltimes of tasks from baseline runs to experimental runs with regard to simplified messages used.

Results: As our answer to *RQ 4* finds: The usage of simplified messages during analysis is too low across all runs and tasks to conclude a meaningful statement on the impact of the TRIMMER on the verification times of Multi-Processing DSS. The task that used the highest amount of simplified messages (66) is s3 (same task as before). During that run, this task completed with an incorrect 'false' result in 354 s. The baseline run completed the same task with a correct 'true' result in 279 s. Checking the walltime visualisation in Figure 5.3 and the bar chart in Figure 5.2, we see that this task normally completes in around 200s to 400s and produces around 10 000 original messages. Therefore, we do not interpret this single isolated result towards any implications the TRIMMER has on Multi-Processing DSS.

In general, we have to leave this research question largely unanswered. The complex scheduling algorithm of Multi-Processing DSS seems to be more intricate than expected, and prevents a large number of messages simplified by the TRIMMER to be used by analysis.

5.4 Summary

Concluding the evaluation we find that the TRIMMER itself performs great with regard to syntactical and semantical correctness, with such errors only making up between 1.6 % and 4.8 % of total unsuccessful LLM requests. Timeouts, making up between 59.5 % and 84.2 % of unsuccessful requests, cause large inefficiencies and affect performance numbers. In total, between 33 % and 45 % of total LLM requests are successful. It is important to keep in mind that the timeout was only employed for better integration with Multi-Processing DSS. We believe omitting an enforced timeout will improve the efficiency of the TRIMMER drastically.
Regarding the impact of the TRIMMER on Multi-Processing DSS, we find that the current scheduling algorithm of Multi-Processing DSS does not allow enough simplifications to be used during analysis. We conducted multiple variations of experiments to find a working solution, but there are too many unknown variables to explore all possibilities in this thesis. Our experiments tell us:

- The limiting factor is not the amount of simplified messages available.
- The limiting factor is not solely the time delay, which the TRIMMMER needs to simplify a message.

5.5 Threats to Validity

Internal Validity. Our experiments were conducted on a shared server that multiple people had access to during our experiments. Variations in CPU performance and memory consumption could have had an influence on both Multi-processing DSS' and the TRIMMER's performance. Also, we measured walltime via BenchExec without using cgroups, which is not as accurate.
There is also a large portion of undocumented errors (19 % of unsuccessful LLM requests on minimal settings). Manually checking the logs shows that these are all cases where the LLM returns an empty payload, but there could be some other error raised by the LLM.
Our selection of tasks is based on results from benchmarks with internal DSS, not Multi-processing DSS. For this reason, the verification times may vary for our experiments, and selected tasks may not be most representative of effects.

External Validity. Tasks were selected specifically to suit Multi-processing DSS, not other verification tools. Also, only 41 tasks were evaluated. We assume that TRIMMER performance translates well to other tasks with similar SMT formulas, as our prompt does not include specific details on the type of formula.
In general, the TRIMMER's approach can only be used with verification tools that employ a formula-exchanging strategy. For analogue implementation, a database that stores such messages is necessary. Since most verifiers work with SMT formulas, it is possible to integrate some variation of the TRIMMER into many verification tools.

Table 5.2: Minimal Setting: Average formula lengths and reduction ratio (per task)

Task	Original Length	Simplified Length	Reduction Ratio
Total	437.24	178.16	0.41
43_1a_cilled_ok_nondet_....yml	797.72	241.30	0.30
aiob_4.c.v+lhb-reducer.yml	476.84	133.00	0.28
apache-get-tag.i.p+lhb-reducer.yml	0.00	0.00	0.42
arctan_Pade.yml	308.49	178.53	0.58
double_req_bl_1230.yml	629.32	331.75	0.53
double_req_bl_1232b.yml	359.74	218.67	0.61
double_req_bl_1250.yml	230.61	151.33	0.66
double_req_bl_1252b.yml	238.20	129.70	0.54
float21.yml	727.51	319.14	0.44
float_req_bl_1250.yml	0.00	0.00	0.22
float_req_bl_1252b.yml	264.91	154.78	0.58
gcd_2+newton_1_8.yml	454.85	171.22	0.38
gcd_2+newton_2_2.yml	425.99	178.72	0.42
id_build.i.p+nlh-reducer.yml	384.25	187.86	0.49
id_build.i.p+sep-reducer.yml	446.46	181.78	0.41
minepump_spec2_product19.cil.yml	402.81	149.00	0.37
minepump_spec2_product27.cil.yml	437.18	200.80	0.46
minepump_spec5_productSimulator.cil.yml	147.00	88.80	0.60
modsf.yml	834.76	130.37	0.16
modulus-2.yml	465.36	106.94	0.23
newton_1_6.yml	288.31	142.14	0.49
newton_1_7.yml	281.43	156.40	0.56
newton_1_8.yml	298.96	158.17	0.53
pc_sfifo_1.cil-2+token_ring.02.cil-1.yml	475.18	190.69	0.40
pc_sfifo_2.cil-1+token_ring.01.cil-1.yml	424.32	183.63	0.43
pc_sfifo_2.cil-1+token_ring.02.cil-1.yml	321.83	152.48	0.47
pc_sfifo_3.cil+token_ring.01.cil-1.yml	370.20	165.23	0.45
pc_sfifo_3.cil+token_ring.02.cil-1.yml	376.20	162.57	0.43
s3_clnt_2.BV.c.cil-1a.yml	432.66	196.12	0.45
s3_clnt_3.BV.c.cil-1a.yml	510.33	216.78	0.42
s3_srvr_2a.BV.c.cil.yml	556.40	257.12	0.46
s3_srvr_2a_alt.BV.c.cil.yml	547.35	217.32	0.40
sqrt_poly.yml	305.18	189.38	0.62
test_locks_10.yml	269.98	133.69	0.50
test_locks_9.yml	195.36	103.93	0.53

Table 5.3: Minimal Setting: LLM requests, outcomes, and simplified messages used (per task)

Task	Total	Successful	Unsuccessful	Simplified Used
Total	3871	1534	2337	62
43_1a_cilled_ok_nondet_....yml	260	103	157	7
aiob_4.c.v+lhb-reducer.yml	40	14	26	0
apache-get-tag.i.p+lhb-reducer.yml	20	13	7	0
arctan_Pade.yml	57	17	40	0
benchmark10_conjunctive.yml	0	0	0	0
double_req_bl_1230.yml	20	4	16	0
double_req_bl_1232b.yml	20	6	14	0
double_req_bl_1250.yml	20	9	11	0
double_req_bl_1252b.yml	20	12	8	11
float21.yml	40	7	33	0
float_req_bl_1230.yml	20	0	20	0
float_req_bl_1232b.yml	20	0	20	0
float_req_bl_1250.yml	20	5	15	0
float_req_bl_1252b.yml	20	10	10	0
gcd_2+newton_1_8.yml	541	200	341	0
gcd_2+newton_2_2.yml	503	167	336	0
id_build.i.p+nlh-reducer.yml	20	7	13	0
id_build.i.p+sep-reducer.yml	20	11	9	0
minepump_spec2_product19....yml	40	23	17	5
minepump_spec2_product27....yml	40	19	21	7
minepump_spec3_product19....yml	0	0	0	0
minepump_spec3_product59....yml	0	0	0	0
minepump_spec5_....yml	20	5	15	0
modsf.yml	236	54	182	4
modulus-2.yml	302	44	258	0
newton_1_6.yml	57	19	38	0
newton_1_7.yml	37	7	30	0
newton_1_8.yml	19	7	12	0
pc_sfifo_1.cil-2+token_ring.02....yml	380	208	172	2
pc_sfifo_2.cil-1+token_ring.01....yml	40	19	21	0
pc_sfifo_2.cil-1+token_ring.02....yml	60	31	29	1
pc_sfifo_3.cil+token_ring.01....yml	180	88	92	2
pc_sfifo_3.cil+token_ring.02....yml	140	68	72	1
s3_clnt_2.BV.c.cil-1a.yml	100	57	43	1
s3_clnt_3.BV.c.cil-1a.yml	80	46	34	2
s3_srvr_2a.BV.c.cil.yml	120	59	61	0
s3_srvr_2a_alt.BV.c.cil.yml	140	90	50	16
sqrt_poly.yml	19	8	11	0
sumt7.yml	0	0	0	0
test_locks_10.yml	140	67	73	3
test_locks_9.yml	60	30	30	0

Table 5.4: Minimal Setting: Reasons for unsuccessful requests (per task)

task	Overapprox. Errors	Syntax Errors	Timeouts
Total	14	89	1868
43_1a_cilled_ok_nondet_....yml	1	5	133
aiob_4.c.v+lhb-reducer.yml	0	0	6
apache-get-tag.i.p+lhb-reducer.yml	0	0	7
arctan_Pade.yml	7	0	18
double_req_bl_1250.yml	0	1	0
double_req_bl_1252b.yml	0	1	7
float21.yml	0	0	14
float_req_bl_1252b.yml	0	2	8
gcd_2+newton_1_8.yml	2	30	309
gcd_2+newton_2_2.yml	0	17	303
id_build.i.p+sep-reducer.yml	0	0	9
minepump_spec2_product19.cil.yml	0	1	5
minepump_spec2_product27.cil.yml	0	0	21
modsf.yml	0	0	173
modulus-2.yml	0	0	255
newton_1_6.yml	0	5	19
newton_1_7.yml	0	2	12
newton_1_8.yml	0	2	10
pc_sfifo_1.cil-2+token_ring.02.cil-1.yml	0	3	148
pc_sfifo_2.cil-1+token_ring.01.cil-1.yml	0	1	9
pc_sfifo_2.cil-1+token_ring.02.cil-1.yml	0	0	29
pc_sfifo_3.cil+token_ring.01.cil-1.yml	0	2	71
pc_sfifo_3.cil+token_ring.02.cil-1.yml	0	1	55
s3_clnt_2.BV.c.cil-1a.yml	0	0	33
s3_clnt_3.BV.c.cil-1a.yml	0	2	32
s3_srvr_2a.BV.c.cil.yml	0	2	44
s3_srvr_2a_alt.BV.c.cil.yml	0	4	46
sqrt_poly.yml	4	0	7
test_locks_10.yml	0	7	66
test_locks_9.yml	0	1	19

Table 5.5: Moderate Setting: LLM requests, outcomes, and simplified messages used (per task)

Task	Total	Successful	Unsuccessful	Simplified Used
Total	10200	3318	6882	82
modsf.yml	858	271	587	5
modulus-2.yml	1216	371	845	0
s3_clnt_2.BV.c.cil-1a.yml	360	168	192	66
s3_clnt_3.BV.c.cil-1a.yml	1580	497	1083	0
s3_srvr_2a.BV.c.cil.yml	2760	926	1834	5
s3_srvr_2a_alt.BV.c.cil.yml	520	304	216	6
gcd_2+newton_1_8.yml	1140	605	535	0
gcd_2+newton_2_2.yml	1766	176	1590	0

Table 5.6: Moderate Setting: Reasons for unsuccessful requests (per task)

Task	Overapprox. Errors	Syntax Errors	Timeouts
Total	26	85	4092
gcd_2+newton_1_8.yml	6	18	510
gcd_2+newton_2_2.yml	3	18	176
modsf.yml	2	8	535
modulus-2.yml	4	15	784
s3_clnt_2.BV.c.cil-1a.yml	0	10	145
s3_srvr_2a.BV.c.cil.yml	6	22	1775
s3_srvr_2a_alt.BV.c.cil.yml	5	5	167

Table 5.7: Proof of Concept: LLM requests, outcomes, and simplified messages used (per task)

Task	Total	Successful	Unsuccessful	Simplified Used
Total	1401	632	769	5
43_1a_cilled_ok_nondet_....yml	637	232	405	1
s3_clnt_2.BV.c.cil-1a.yml	764	400	364	4

Table 5.8: Proof of Concept: Reasons for unsuccessful requests (per task)

Task	Overapprox. Errors	Syntax Errors	Timeouts
Total	1	20	648
43_1a_cilled_ok_nondet_....yml	0	13	392
s3_clnt_2.BV.c.cil-1a.yml	1	7	256

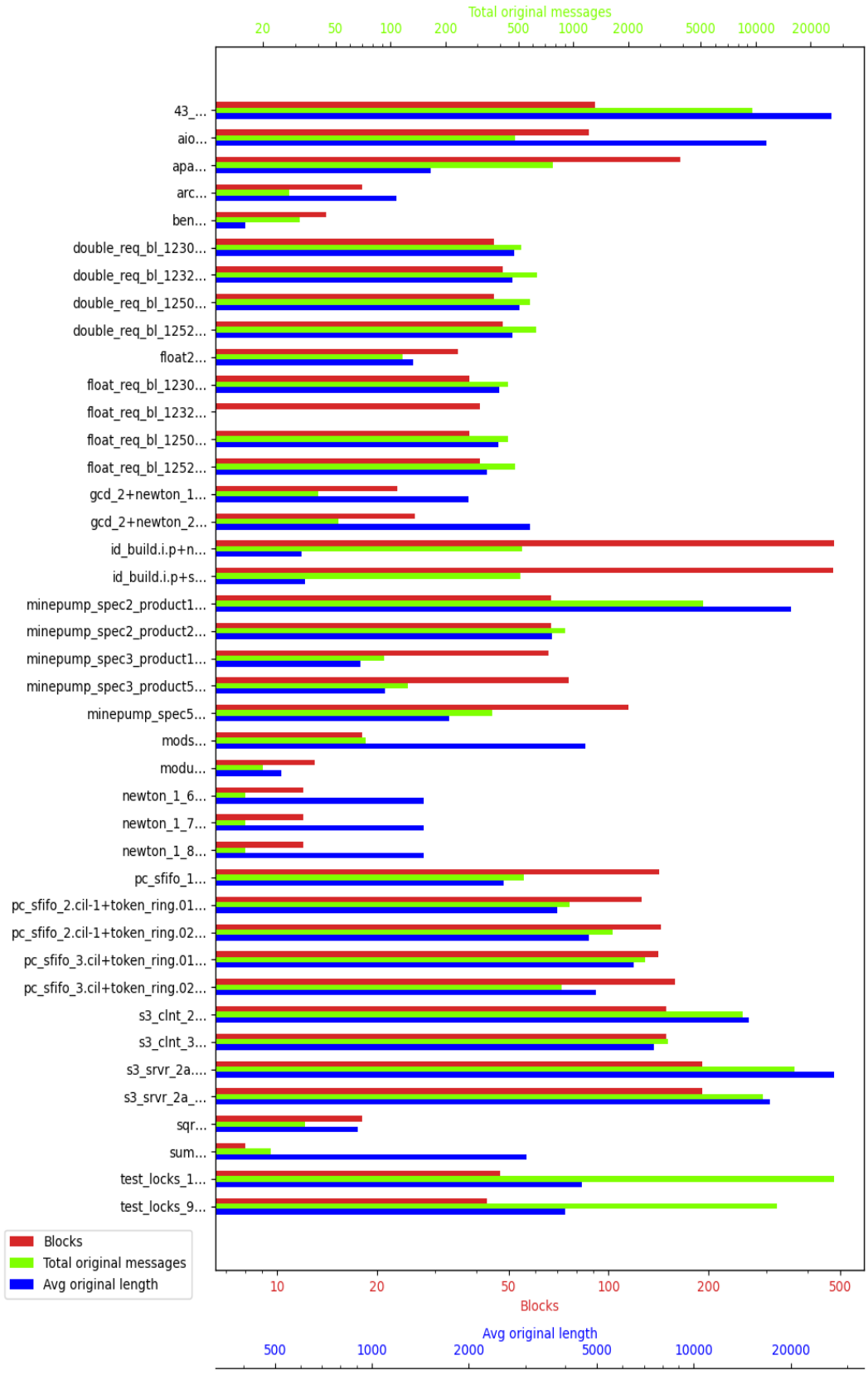


Figure 5.2: Per task comparison of block amount, amount of original messages, average length of original messages

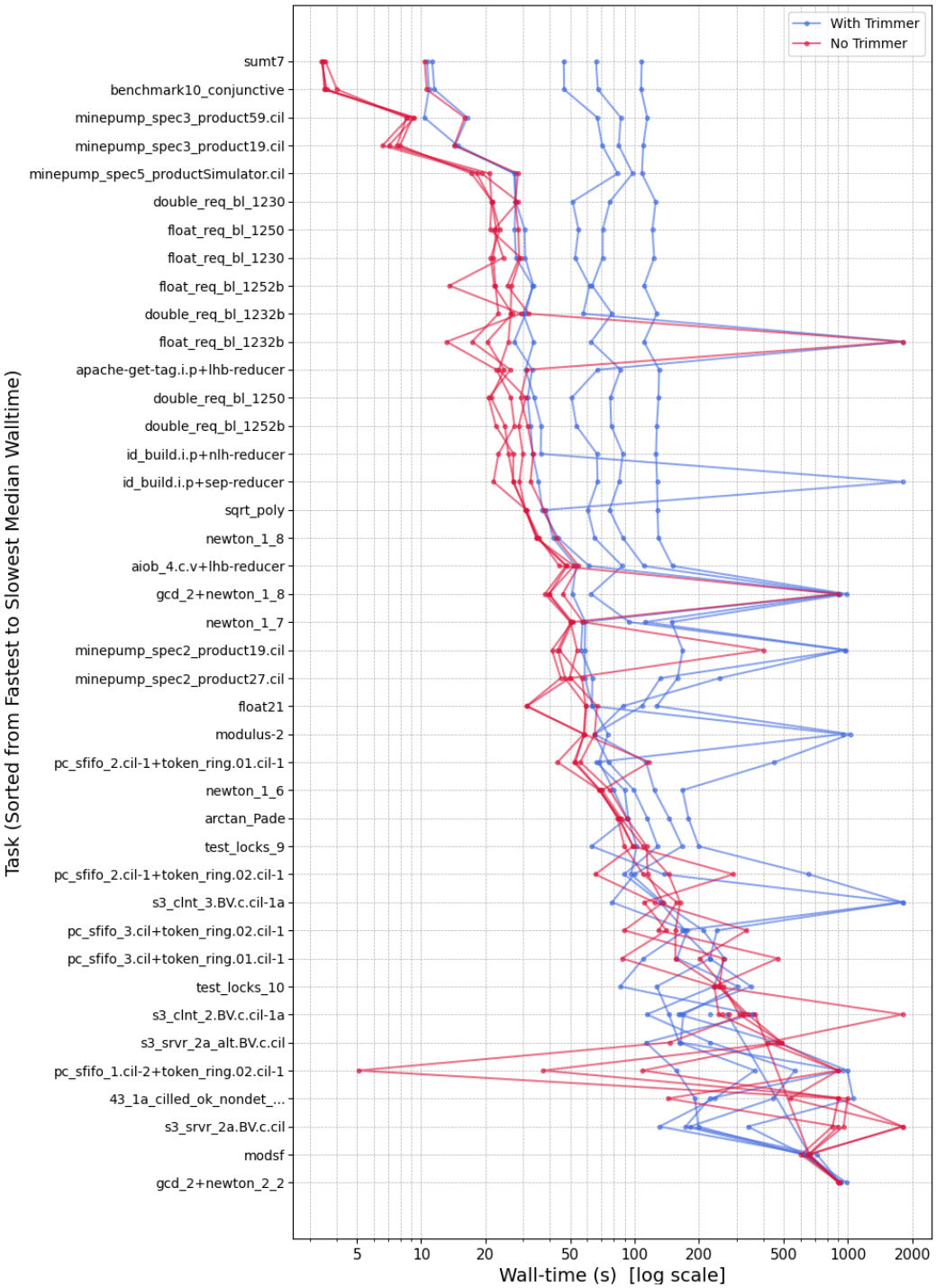


Figure 5.3: Walltimes of every task for every run

6 Future Work

Impact on Multi-Processing DSS. Further research should evaluate the TRIMMER's effect on Multi-processing DSS in closer detail. To do so, it would be necessary to find a configuration for Multi-processing DSS that allows it to use more simplified messages. Starting points for an analysis of the configuration can be whether the timestamp is an issue, or if each message's flag 'handled' needs to be set to false manually. Experimenting with larger block size of analyses could cause Multi-processing DSS to slow down the verification. Potentially, that would enable more of the simplified messages to be used. With more simplified messages in usage during analysis, a re-evaluation of the impact of the TRIMMER on Multi-Processing DSS would become necessary.

TRIMMER improvements. The TRIMMER currently lacks a heuristic or smart strategy for the selection of messages. This can cause the TRIMMER to request simplification of the same message multiple times, if a previous simplification has failed. Future work could implement such strategies, for example with backoffs if a simplification is attempted multiple times.

Integrating other LLMs. New LLM client implementations could be added and evaluated against each other. Especially in the ever-changing environment of LLMs, it is essential to integrate and evaluate new and improved AI models. The current evaluation already shows that syntax and overapproximation are not limiting factors for Google Gemini 2.5 Flash. Instead, it is mainly timing out. Newer and faster models could mitigate this.

Adding other Solvers. Similar to the LLM client, the Validator module could also be interchanged easily. It would be interesting to see if any new solver features could be integrated as pre-processing steps for LLM simplification. Should an algorithmic simplification process (e.g. quantifier elimination) be implemented for Multi-processing DSS, it could be evaluated against the LLM simplification.

Algorithmic Simplification. Our work finds that algorithmic, logical simplification cannot yet be replaced. Therefore, future work for Multi-processing DSS could attempt to implement algorithmic approaches like Quantifier Elimination, Craig Interpolation, or the simplification algorithm by Dillig et al. [38], which has shown promising results in the verification tool 'Ultimate Automizer' [40].

7 Conclusion

Our work proposes an approach to simplify SMT formulas with the help of modern LLMs. In particular, we have successfully implemented a microservice that continuously simplifies the block summaries of Multi-processing DSS, and evaluated the performance of the TRIMMER and Multi-processing DSS based on tasks from sv-benchmarks. Results showed that the performance of the TRIMMER with regard to syntactical and semantical performance is very good. Our approach is justified and delivers simplifications in the realm of 40 % of their original length, which is an indication for what LLMs might be able to achieve in the future. Also, we showed that enforced timeouts had a large impact on efficieny.

The impact of using simplified messages on verification with Multi-Processing DSS could not be fully evaluated. This was due to the current Multi-Processing DSS configuration not allowing enough simplified messages to be used during analysis. On the other hand, we were able to rule out the amount of available simplifications, and the time delay of simplification as sole causes for this effect. This indicates a direction for future work and analysis of the root causes.
Concluding our work, we find that simplifying SMT formulas with the help of LLMs is definitely possible, and a path worth exploring further. In contrast, with the current configuration, it is not the simple, drop-in replacement for algorithmic simplification we had aimed to integrate into Multi-Processing DSS. Should it be possible to find a configuration that allows Multi-Processing DSS to use the TRIMMER'S simplified messages, this will have to be re-evaluated.

Disclaimer. For development of the TRIMMER Github Copilot was used as autocompletion tool.

Bibliography

[1] Benchexec releases. https://github.com/sosy-lab/benchexec/releases. Accessed: 2025-06-29.

[2] Benchexec repository. https://github.com/sosy-lab/benchexec. Accessed: 2025-06-29.

[3] Google gemini flash benchmark results. https://storage.googleapis.com/model-cards/documents/gemini-2.5-flash-preview.pdf. Accessed: 2025-06-11.

[4] Google gemini model. https://deepmind.google/models/gemini/flash/. Accessed: 2025-07-02.

[5] Javasmt github repository. https://github.com/sosy-lab/java-smt. Accessed: 2025-06-05.

[6] Kubernetes overview. https://kubernetes.io/docs/concepts/overview/. Accessed: 2025-06-11.

[7] Kubernetes website. https://kubernetes.io. Accessed: 2025-06-11.

[8] Linkerd website. https://linkerd.io. Accessed: 2025-06-11.

[9] Minikube handbook. https://minikube.sigs.k8s.io/docs/handbook/. Accessed: 2025-06-11.

[10] Multi-processing dss data repository. https://gitlab.com/sosy-lab/research/data/distributed-summary-synthesis. Accessed: 2025-06-30.

[11] Multi-processing dss software repository. https://gitlab.com/sosy-lab/software/distributed-summary-synthesis. Accessed: 2025-06-11.

[12] Quarkus grpc guide. https://quarkus.io/guides/grpc. Accessed: 2025-06-11.

[13] Quarkus micrometer. https://quarkus.io/guides/telemetry-micrometer. Accessed: 2025-06-11.

[14] Quarkus mutiny. https://quarkus.io/guides/mutiny-primer. Accessed: 2025-06-11.

[15] Quarkus open telemetry. https://quarkus.io/guides/opentelemetry. Accessed: 2025-06-11.

[16] Quarkus rest. https://quarkus.io/guides/rest. Accessed: 2025-06-11.

[17] Quarkus scheduler. https://quarkus.io/guides/scheduler. Accessed: 2025-06-11.

[18] Quarkus website. https://quarkus.io. Accessed: 2025-06-11.

[19] D. Baier, D. Beyer, and K. Friedberger. Javasmt 3: Interacting with SMT solvers in java. In A. Silva and K. R. M. Leino, editors, *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II*, volume 12760 of *Lecture Notes in Computer Science*, pages 195–208. Springer, 2021.

[20] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In J. Launchbury and J. C. Mitchell, editors, *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, pages 1–3. ACM, 2002.

[21] H. Barbosa, C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar. cvc5: A versatile and industrial-strength SMT solver. In D. Fisman and G. Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022.

[22] C. Barrett, A. Stump, C. Tinelli, et al. The smt-lib standard: Version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)*, volume 13, page 14, 2010.

[23] C. W. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011.

[24] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability modulo theories. In A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 1267–1329. IOS Press, 2021.

[25] D. Beyer and M. E. Keremoglu. Cpachecker: A tool for configurable software verification. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 184–190. Springer, 2011.

[26] D. Beyer, M. E. Keremoglu, and P. Wendler. Predicate abstraction with adjustable-block encoding. In R. Bloem and N. Sharygina, editors, *Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2010, Lugano, Switzerland, October 20-23*, pages 189–197. IEEE, 2010.

[27] D. Beyer, M. Kettl, and T. Lemberger. Decomposing software verification using distributed summary synthesis. *Proc. ACM Softw. Eng.*, 1(FSE):1307–1329, 2024.

[28] N. S. Bjørner and K. Fazekas. On incremental pre-processing for SMT. In B. Pientka and C. Tinelli, editors, *Automated Deduction - CADE 29 - 29th International Conference on Automated Deduction, Rome, Italy, July 1-4, 2023, Proceedings*, volume 14132 of *Lecture Notes in Computer Science*, pages 41–60. Springer, 2023.

[29] R. Brummayer and A. Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In S. Kowalewski and A. Philippou, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5505 of *Lecture Notes in Computer Science*, pages 174–177. Springer, 2009.

[30] J. Christ, J. Hoenicke, and A. Nutz. Smtinterpol: An interpolating SMT solver. In A. F. Donaldson and D. Parker, editors, *Model Checking Software - 19th International Workshop, SPIN 2012, Oxford, UK, July 23-24, 2012. Proceedings*, volume 7385 of *Lecture Notes in Computer Science*, pages 248–254. Springer, 2012.

[31] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani. The mathsat5 SMT solver. In N. Piterman and S. A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7795 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2013.

[32] A. Cimatti, A. Griggio, and R. Sebastiani. Efficient interpolant generation in satisfiability modulo theories. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 397–412, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[33] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In E. A. Emerson and A. P. Sistla, editors, *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.

[34] W. Craig. Linear reasoning. a new form of the herbrand-gentzen theorem. *Journal of Symbolic Logic*, 22(3):250–268, 1957.

[35] L. M. de Moura and N. S. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[36] L. M. de Moura and G. O. Passmore. The strategy challenge in SMT solving. In M. P. Bonacina and M. E. Stickel, editors, *Automated Reasoning and Mathematics - Essays in Memory of William W. McCune*, volume 7788 of *Lecture Notes in Computer Science*, pages 15–44. Springer, 2013.

[37] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[38] I. Dillig, T. Dillig, and A. Aiken. Small formulas for large programs: Online constraint simplification in scalable static analysis. In R. Cousot and M. Martel, editors, *Static Analysis - 17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Proceedings*, volume 6337 of *Lecture Notes in Computer Science*, pages 236–252. Springer, 2010.

[39] I. Garcia-Contreras, H. G. V. K., S. Shoham, and A. Gurfinkel. Fast approximations of quantifier elimination. In C. Enea and A. Lal, editors, *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II*, volume 13965 of *Lecture Notes in Computer Science*, pages 64–86. Springer, 2023.

[40] M. Heizmann, M. Barth, D. Dietsch, L. Fichtner, J. Hoenicke, D. Klumpp, M. Naouar, T. Schindler, F. Schüssele, and A. Podelski. Ultimate automizer and the commuhash normal form - (competition contribution). In S. Sankaranarayanan and N. Sharygina, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part II*, volume 13994 of *Lecture Notes in Computer Science*, pages 577–581. Springer, 2023.

[41] L. Khachiyan. Fourier-motzkin elimination method. In C. A. Floudas and P. M. Pardalos, editors, *Encyclopedia of Optimization, Second Edition*, pages 1074–1077. Springer, 2009.

[42] G. Lample and F. Charton. Deep learning for symbolic mathematics. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020.

[43] T. Lemberger. Multi-processing for distributed summary synthesis, September 2024.

[44] Z. Lu, S. Siemer, P. Jha, J. D. Day, F. Manea, and V. Ganesh. Layered and staged monte carlo tree search for SMT strategy synthesis. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI 2024, Jeju, South Korea, August 3-9, 2024*, pages 1907–1915. ijcai.org, 2024.

[45] R. C. Lyndon. An interpolation theorem in the predicate calculus. 1959.

[46] K. L. McMillan. Interpolation and sat-based model checking. In W. A. H. Jr. and F. Somenzi, editors, *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2003.

[47] K. L. McMillan. Applications of craig interpolation to model checking. In G. Ciardo and P. Darondeau, editors, *Applications and Theory of Petri Nets 2005, 26th International Conference, ICATPN 2005, Miami, USA, June 20-25, 2005, Proceedings*, volume 3536 of *Lecture Notes in Computer Science*, pages 15–16. Springer, 2005.

[48] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, 1979.

[49] S. Polu and I. Sutskever. Generative language modeling for automated theorem proving. *CoRR*, abs/2009.03393, 2020.

[50] S. Ranise and C. Tinelli. The smt-lib format: An initial proposal. In *Proceedings of the 1st Workshop on Pragmatics of Decision Procedures in Automated Reasoning*. Citeseer, 2003.

[51] D. Selsam and N. S. Bjørner. Guiding high-performance SAT solvers with unsat-core predictions. In M. Janota and I. Lynce, editors, *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings*, volume 11628 of *Lecture Notes in Computer Science*, pages 336–353. Springer, 2019.

[52] D. Selsam, M. Lamm, B. Bünz, P. Liang, L. de Moura, and D. L. Dill. Learning a SAT solver from single-bit supervision. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[53] P. Shojaee*†, I. Mirzadeh*, K. Alizadeh, M. Horton, S. Bengio, and M. Farajtabar. The illusion of thinking: Understanding the strengths and limitations of reasoning models via the lens of problem complexity, 2025.

[54] R. E. Shostak. Deciding combinations of theories. *J. ACM*, 31(1):1–12, 1984.