BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

PROGRAM LOOP UNWINDING IN THE 2LS FRAMEWORK

ROZBALOVÁNÍ SMYČEK PROGRAMŮ V NÁSTROJI 2LS

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR FRANTIŠEK NEČAS
AUTOR PRÁCE

SUPERVISOR Ing. VIKTOR MALÍK
VEDOUCÍ PRÁCE

BRNO 2022

Brno University of Technology
Faculty of Information Technology

Department of Intelligent Systems (DITS) Academic year 2021/2022

Bachelor's Thesis Specification

24719

Student:	**Nečas František**
Programme:	Information Technology
Title:	**Program Loop Unwinding in the 2LS Framework**
Category:	Formal Verification

Assignment:

1. Get acquainted with the 2LS framework for formal verification of C programs. Study existing concepts that the framework builds on, mainly its internal program representation and the verification techniques that the tool uses.
2. Investigate the current state of program loop unwinding that is present in 2LS and identify its defects.
3. Propose an improved method for loop unwinding suitable for the 2LS framework that allows to correctly unwind all types of programs whose verification 2LS supports.
4. Implement the proposed method in 2LS.
5. Evaluate the created solution on benchmarks from at least 2 sub-categories of the International Competition on Software Verification (SV-COMP). Discuss the impact of the solution on verification abilities of 2LS.

Recommended literature:

- Brain, M.; Joshi, S.; Kroening, D.; et al.: Safety Verification and Refutation by k-Invariants and k-Induction. In Proceedings of the 22nd International Static Analysis Symposium, LNCS, vol. 9291. Springer. 2015. pp. 145-161.
- Schrammel, P.; Kroening, D.: 2LS for Program Analysis - (Competition Contribution). In Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems, LNCS, vol. 9636. Springer. 2016. pp. 905-907.
- Malík, V., Hruska, M., Schrammel, P., & Vojnar, T.: Template-based verification of heap-manipulating programs. In 2018 Formal Methods in Computer Aided Design (FMCAD). IEEE. 2018. pp. 1-9.

Requirements for the first semester:

- The first two items of the assignment.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Malík Viktor, Ing.**
Head of Department:	Hanáček Petr, doc. Dr. Ing.
Beginning of work:	November 1, 2021
Submission deadline:	May 11, 2022
Approval date:	November 3, 2021

Abstract

The goal of this work is to propose an improved unwinding mechanism for the 2LS formal verification tool. 2LS is a static analysis framework for C programs based on reasoning about programs using an SMT solver. It combines multiple common verification techniques into an algorithm called kIkI. One of the crucial parts of the algorithm is loop unwinding. Unfortunately, the existing solution does not correctly support unwinding of loops containing operations with dynamically allocated memory. Our proposed solution is based on unwinding loops in a GOTO program rather than the SSA form, making it possible to correctly handle dynamic objects and operations over them. The proposed solution has been implemented in the 2LS framework and our experiments on a set of benchmarks from the International Competition on Software Verification (SV-COMP) show that it improves soundness of analysis of programs working with dynamic objects.

Abstrakt

Cílem této práce je navrhnout vylepšený mechanismus rozbalování smyček pro analyzátor 2LS. 2LS je nástroj pro statickou analýzu C programů založený na usuzování o programech pomocí SMT solveru. Kombinuje několik běžných verifikačních technik do algoritmu zvaného kIkI. Jednou z klíčových součástí tohoto algoritmu je rozbalování smyček programu. Současné řešení bohužel neumožňuje správně rozbalovat smyčky obsahující operace s dynamicky alokovanou pamětí. Námi navrhované řešení je založeno na rozbalování smyček v GOTO programu namísto SSA formy, díky čemuž je možné správně pracovat s dynamickými objekty a operacemi s nimi. Navržené řešení bylo implementováno v nástroji 2LS a naše experimenty na sadě testů z mezinárodní soutěže ve verifikaci software (SV-COMP) ukazují, že zvyšuje korektnost analýzy programů pracujících s dynamickými objekty.

Keywords

program analysis, formal verification, static analysis, 2LS Framework, loop unwinding, k-induction, bounded model checking, SSA form, GOTO programs, dynamic memory

Klíčová slova

analýza programů, formální verifikace, statická analýza, nástroj 2LS, rozbalování smyček, k-indukce, bounded model checking, SSA forma, GOTO programy, dynamická paměť

Reference

NEČAS, František. *Program Loop Unwinding in the 2LS Framework.* Brno, 2022. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Viktor Malík

Rozšířený abstrakt

2LS je nástroj pro statickou analýzu C programů založený na odvozování invariantů pomocí SMT solveru. Díky kombinaci více různých verifikačních technik dokáže 2LS nalézt chyby v programech i dokázat platné vlastnosti programů. 2LS se v současnosti zaměřuje na analýzy jako např. správnost operací s ukazateli, terminace, neterminace a neporušení hranic polí.

Invarianty v 2LS jsou odvozovány pomocí algoritmu kIkI, který kombinuje principy abstraktní interpretace, k-indukce a bounded model checking (BMC) způsobem, který do značné míry eliminuje slabé stránky jednotlivých přístupů díky vzájemné komplementaci. Důležitou součástí metod k-indukce a BMC je rozbalování smyček programu, které umožňuje rozbalit přechodovou relaci programu a tím např. detekovat chyby pomocí BMC. Současná implementace využívá jednoduchého rozbalení smyček v interní reprezentaci single static assignment (SSA), kdy je tělo smyčky jednoduše nakopírováno před smyčku. Tento přístup ovšem nedokáže zajistit plnohodnotnou podporu pro analýzu programů pracujících s dynamickou pamětí kvůli paměťovému modelu využívanému v 2LS. Dokonce analýza takových programů často vede k nekorektnímu výsledku analýzy. Navíc je současné řešení nedostatečně zdokumentované, což komplikuje jeho údržbu. Cílem této práce je navrhnout alternativní přístup k rozbalování smyček, který umožní rozbalování smyček všech programů, které 2LS momentálně podporuje, a zároveň správně podporuje operace s dynamickou pamětí.

Navržené řešení rozbaluje smyčky v interní reprezentaci GOTO programů (které odpovídají grafům toku řízení). Díky tomu je po rozbalení možné provést úpravy alokací dynamických objektů (které jsou zaváděny právě v této reprezentaci) a následně spočítat novou formu SSA pro další analýzu. Díky tomuto sledu úprav je při výpočtu nové SSA formy možné po rozbalení smyček korektně zohlednit nový stav dynamických objektů v programu, a tedy správně reprezentovat operace s dynamickou pamětí. Nad nově spočtenou SSA formou je poté nutné udělat některé modifikace, které provádělo i původní řešení a jsou nutné pro některé typy analýz. V případě k-indukce nebo BMC jsou přidány tzv. předpoklady, aby mohla být následná analýza přesnější.

Navrhované řešení bylo implementováno v nástroji 2LS. Nástroj 2LS je postaven nad knihovnou CProver, která poskytuje mnohá užitečná rozhraní, např. pro práci s GOTO programy. V rámci této práce jsme aktualizovali verzi knihovny CProver využívanou v 2LS (původní byla více než 4 roky stará a od té doby proběhlo mnoho změn a oprav), abychom mohli využít nejnovější rozhraní pro rozbalování smyček v GOTO programech a také usnadnili budoucí vývoj 2LS. S naimplementovaným řešením jsme prováděli experimenty nad sadou úkolů z mezinárodní soutěže ve verifikaci software (SV-COMP). Experimenty ukázaly, že implementované řešení dosahuje lepších výsledků na úlohách obsahujících práci s dynamickou pamětí (např. v kategorii Memory Safety), ovšem je zde zhoršení v jiných kategoriích z důvodu nevyužívání inkrementálního SAT solveru.

Na základě toho jsme implementaci řešení upravili způsobem, že nový přesnější mechanismus pro rozbalování smyček je využit jen v případě, kdy se v programu vyskytují operace s dynamickou pamětí, jinak je použit starý a výkonnější algoritmus. Takto implementované řešení dosahuje lepších výsledků než řešení původní napříč všemi našimi experimenty a zásadně rozšiřuje verifikační schopnosti 2LS při práci s programy s dynamickou pamětí. Dále také práce pojednává o novém řešení, které by kompletně nahradilo řešení původní a zároveň by využívalo inkrementální SAT solving. Toto řešení ovšem nebylo implementováno.

Program Loop Unwinding in the 2LS Framework

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Ing. Viktor Malík. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
František Nečas
May 9, 2022

Acknowledgements

I would like to thank my supervisor Ing. Viktor Malík for his support and patient explanations during the work on this thesis and also previously the Project Practice courses. I would also like to thank my parents for their continuous support during my studies.

Contents

Chapter 1

Introduction

Research in the field of software verification is evolving very fast. There are a lot of tools being developed, often using different approaches, each with its advantages and disadvantages. Oftentimes, tools focus on analysing only a single type of property and while excelling at such analysis, they fail to analyse complex programs since those usually require analysis of a combination of properties.

One of the tools combining multiple approaches into a single scalable framework is 2LS. By using a combination of common verification techniques, 2LS can find errors in the system as well as prove its true properties. 2LS focuses on analysing the most common properties of sequential C programs, such as reachability of assertions, validity of pointer operations, array bounds, or termination.

The analysis in 2LS is based on translating a program into a single static assignment (SSA) form, computing inductive invariants of loops, and reasoning about the program properties using an SMT solver. This is all connected in an algorithm called kIkI that combines common verification techniques, namely abstract interpretation, k-induction, and bounded model checking. The main principle of k-induction and bounded model checking is checking correctness of a program by unrolling its transition relation. In order to analyse loops using these methods, they must be unwound during the analysis, making the algorithm for loop unwinding crucial for verification in 2LS.

Unfortunately, the current implementation of the unwinding algorithm does not support correct representation of operations over dynamically allocated memory according to the memory model utilised in 2LS. This prevents usage of k-induction and bounded model checking for some programs as their verification would be unsound. Moreover, the current unwinder implementation is not documented very well, which makes it difficult to be maintained, as was shown by a significant bug being discovered only recently even though it has been present for a long time. To address the mentioned issues, the goal of this work is to propose and implement an alternative approach to loop unwinding supporting all types of analysis that 2LS currently allows while also correctly handling operations with dynamic memory. Contrary to the previous unwinder of 2LS which directly unwinds the SSA form, our solution is based on unwinding the GOTO program (a CFG-based representation coming from the CProver framework which 2LS is built on) and updating the dynamic objects inside the GOTO program. This approach facilitates correct representation of operations over dynamic memory.

The proposed solution has been implemented in the 2LS framework. It can correctly analyse programs operating with dynamic memory inside loops, as well as soundly detect memory leaks. Based on experiments with benchmarks from the International Competition on Software Verification (SV-COMP), the solution improves verification capabilities of 2LS. We have observed a notable increase in correctly verified tasks, as well as of the overall score. As a part of this work, we have also updated the version of the underlying CProver framework used in 2LS.

The rest of this text is organised as follows. In Chapter 2, we introduce the most essential concepts of program verification in 2LS, mainly focusing on those that are related to loop unwinding and intermediate representations. Chapter 3 gives an overview of the state-of-the-art of loop unwinding and gives more insight into the current state of unwinding in 2LS. Our proposed solution and its components are described in Chapter 4. An overview of the CProver update and other implementation details are described in Chapter 5. Chapter 6 gives an overview of the experiments and benchmarks performed in order to evaluate the implemented solution.

This thesis extends work which was previously done as a part of Project Practice 1 and Project Practice 2 courses. In particular, some parts of Chapter 2 were taken from the reports written in these courses.

Chapter 2

Foundations of the 2LS Tool

The main goal of this thesis is to improve unwinding in the 2LS tool. The unwinder is one of the core components of the tool, its functionality is crucial in most of the performed analyses. This chapter briefly describes the main concepts used in 2LS, mainly focusing on those that are related to unwinding.

2LS is a static analysis framework aimed at analysing sequential C programs. It is built upon the CProver framework which provides basic functionality for program analyses and working with internal representations. 2LS allows verification of user-defined assertions, memory safety, array bounds, and termination and non-termination properties.

Since it would be difficult to directly analyse C source code, analysers often make use of various internal representations. Section 2.1 describes the representations used in 2LS, namely GOTO programs (provided by the CProver framework) and single static assignment (SSA) form. The analysis in 2LS is based on inference of inductive invariants using an algorithm called kIkI that combines common verification techniques – abstract interpretation, k-induction and bounded model checking. Section 2.2 gives an overview of these concepts and how they tie together into a single algorithm.

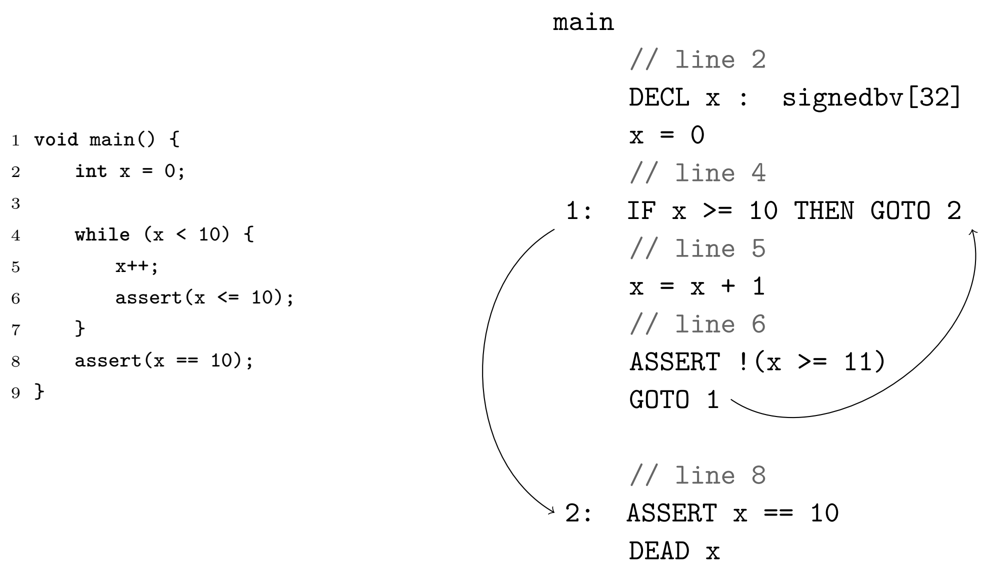
2.1 Intermediate Representations in 2LS

In order to conveniently and efficiently analyse source programs, 2LS utilises two types of intermediate representations. First, the source program is converted into a control flow graph, called a GOTO program, using the CProver framework. A GOTO program consists of GOTO functions that correspond to the functions of the original program. Each GOTO function is then transformed into a single static assignment (SSA) form which allows constructing a logical formula describing the semantics of the function. 2LS uses these formulae to verify correctness of the source program.

2.1.1 GOTO Programs

GOTO program is a language-independent representation similar to the ones used in compilers [8]. A GOTO program consists of GOTO functions, each of which is a list of GOTO instructions. Every instruction has a distinct type, a code expression, a guard expression, a source code location, and, optionally, targets for the next instruction which form the program into a control flow graph. At the time of writing, there are 19 types of instructions of various types, e.g., computational (`ASSIGN` or `GOTO`) or informational (`LOCATION` or `DEAD`). From the point of view of program verification, important instructions are those for assumptions and assertions about the code, called `ASSUME` and `ASSERT`, respectively. An assumption makes the thread of execution expect the guard expression to hold, whereas an assertion expresses that the guard expression should hold for all possible executions. Oftentimes, the goal of verification is to prove that all assertions of the given program hold (or to find a counterexample if they do not hold).

The conversion from C to GOTO takes place in two phases [8]. At first, each instruction in the source program is converted into its GOTO instruction equivalent, however at this point, control flow aspects of the program are not encoded yet. These are added later in the second phase; `if-else` and `while` constructions are replaced by equivalent `GOTO`s and instructions are grouped into basic blocks starting with labels. Moreover, variables are extended with information about their lifespan using `DECL` and `DEAD` instructions for variable declaration and variable deletion, respectively. A conversion of a simple C program can be seen in Figure 2.1. One of the advantages of this representation is the possibility to serialize and deserialize it, e.g., to a binary format.



```
1 void main() {
2     int x = 0;
3
4     while (x < 10) {
5         x++;
6         assert(x <= 10);
7     }
8     assert(x == 10);
9 }
```

```
main
        // line 2
        DECL x :  signedbv[32]
        x = 0
        // line 4
  1:    IF x >= 10 THEN GOTO 2
        // line 5
        x = x + 1
        // line 6
        ASSERT !(x >= 11)
        GOTO 1

        // line 8
  2:    ASSERT x == 10
        DEAD x
```

Figure 2.1: Conversion of a simple C program to GOTO program

As mentioned above, each instruction contains a guard expression and a code expression. The guard expression represents a condition under which the instruction is executed [8] and the code expression represents the effect of the instruction, e.g., an assignment. Expressions of the program inside GOTO programs can be thought of as subtrees of the Abstract Syntax Tree (AST) known from compilers [3]. A node of an expression can be of various types, e.g., based on the children count we can differentiate unary and binary expressions. This forms an inheritance hierarchy of expressions in the CProver framework. Figure 2.2 shows a simple expression calculating the sum of two integer constants.

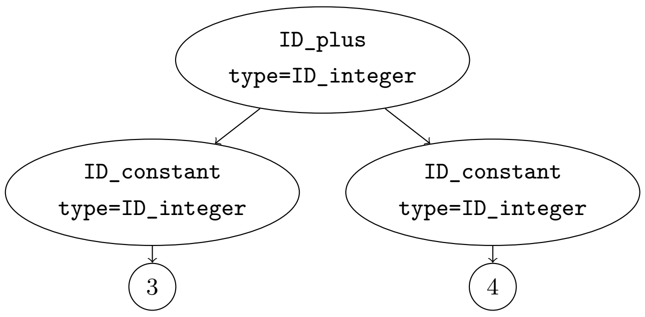


Figure 2.2: Expression tree of sum of two integer constants

2.1.2 Single Static Assignment Form

Single Static Assignment (SSA), as the name suggests, is an intermediate representation of programs within which each variable is assigned to at most once. One of the advantages of this representation is that it is simple to convert it to a logical formula representing the program semantics as the conjunction of formulae corresponding to individual program statements.

A translation from GOTO to SSA requires splitting each variable v into several variables v_i at each assignment to v where i is the location of the assignment. Every R-value usage of v is replaced by the corresponding variable v_i, where i is the last node where v was assigned to before the given use of v [11]. In order to ensure that there is always a single such node, additional assignments must be introduced at joint points (e.g., at the beginning of loops) of the program. These are called Φ (*phi*) nodes and have the form of an assignment $x = \Phi(y, z)$ meaning that x is assigned the value of y if the control reaches this node via the first entering edge (e.g., from before the loop), and x is assigned the value of z if the node is reached via the second entering edge (e.g., from the end of the loop).

In 2LS, the standard SSA is extended with control flow information. To achieve this, special variables called *guards* are introduced. For each program location i, a boolean variable g_i describes the condition under which the given location is reachable.

In order to use the generated formula with an SMT solver, 2LS makes the SSA acyclic by cutting loops at their end of the body. This is done by replacing each variable coming from the end of a loop body by a fresh unconstrained variable (called a loop-back variable). Moreover, the choice in the corresponding Φ node between the loop-back variable (coming from the end) and the variable coming from before the loop is made non-deterministic by using an unconstrained boolean variable (a so-called loop-select variable) [2]. This transformation is crucial for verification in 2LS since it over-approximates the control flow of the program by using the free variables. The precision of this representation is then typically improved by computing a loop invariant which reduces the possible values of the loop-back variable. A loop invariant describes a property that holds at the end of any iteration of the loop. An example of this transformation can be seen in Figure 2.3. The loop has been cut at the end of its body: a free loop-back variable x_2^{lb} is passed to the loop head instead of the actual value of x expressed by x_2. The choice of the value in the Φ node is based on a free boolean loop-select variable g_2^{ls}. Variables x_2^{lb} and g_2^{ls} are free variables, making this representation an over-approximation of the original program semantics.

Figure 2.4 shows translation of a program introduced in Figure 2.1 to its SSA over-approximation. Line 1 is the entry of the program; it is always reachable, therefore g_1 is

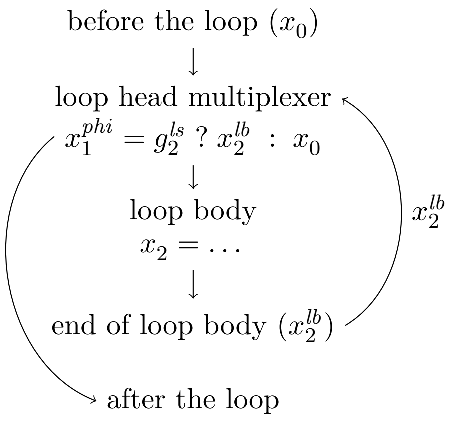


Figure 2.3: Encoding of a loop into SSA used in 2LS

true. The variable x is defined and initialised at line 2. The loop head at line 5 is reachable if the previous location is also reachable, therefore its guard g_4 is equal to g_2. The guard g_7 expresses that the loop body is reachable if the loop itself is reachable (g_4 is true) and if the loop condition is true ($x_5^{phi} < 10$). Guard g_8 represents the assertion inside the loop; the assertion must be true in order for the program to get past it. The program can reach the statement following the loop (represented by guard g_{12}) only if the loop was reachable and the loop condition was false. Line 13 represents the final assertion. Once it is reachable, the value of x must be equal to 10. Finally, the end of the function is reachable if the assertion is reachable and if it was satisfied.

```
1 void main() {
2     int x = 0;
3
4     while (x < 10) {
5         x++;
6         assert(x <= 10);
7     }
8     assert(x == 10);
9 }
```

```
1      DECL x : signedbv[32]
2      x = 0
3
4   1: IF x >= 10 THEN GOTO 2
5
6      x = x + 1
7
8      ASSERT !(x >= 11)
9
10     GOTO 1
11
12  2: ASSERT x == 10
13
14     DEAD x
15     END_FUNCTION
```

1 $g_1 = true$
2 $x_2 = 0$
3
4 $g_4 = g_1$
5 $x_5^{phi} = (g_{10}^{ls} \ ? \ x_{10}^{lb} \ : \ x_2)$
6 $x_6 = x_5^{phi} + 1$
7 $g_7 = \neg(x_5^{phi} \geq 10) \wedge g_4$
8 $g_8 = \neg(x_6 \geq 11)$
9
10 // loop back
11
12 $g_{12} = x_5^{phi} \geq 10 \wedge g_4$
13 $x_5^{phi} = 10 \vee \neg g_{12}$
14
15 $g_{15} = x_5^{phi} = 10 \wedge g_{12}$

Figure 2.4: Encoding of a simple GOTO program into SSA

2.1.3 Memory Model Used in SSA

Among C constructions whose encoding into SSA is challenging are those dealing with pointers and dynamically allocated memory. To this end, we describe them in detail in this section. In 2LS, dynamic memory allocations are encoded using so-called abstract dynamic objects. Each abstract dynamic object represents a set of concrete dynamic objects allocated at the same allocation site (allocated by the same `malloc`) [13]. Each call of `malloc` is replaced by a new abstract dynamic object[1], the replacement at location i follows the pattern:

$$\texttt{malloc(sizeof(x))} \longrightarrow \&\mathit{dynamic\_object}\$i. \tag{2.1}$$

where the type of $\mathit{dynamic\_object}\$i$ is x.

The conversion of pointer dereferencing operations to SSA is done using a static *points-to* analysis which for each pointer computes a set of objects in memory which it can be dereferenced into in each program location where the pointer is used [11]. Based on this analysis, operations via pointers (i.e., reads and writes to memory) are encoded using a case-split of objects which a pointer can be dereferenced into. Let us assume that a pointer p can be dereferenced into a set of objects O at a program location i. To facilitate encoding of memory operations at i, we introduce a new variable $drf(p_i)$ representing a dereference of p at i.

Then, an R-value (memory read) expression $p \to f$ is represented by an SSA formula [13]:

$$\bigwedge_{o \in O} p = \&o \Rightarrow drf(p).f = o.f \wedge ((\bigwedge_{o \in O} p \neq \&o) \Rightarrow drf(p).f = o_\perp) \tag{2.2}$$

where $o_\perp$ represents an unknown object. Accessing the value of an unknown object can be thought of as a program error. Formula 2.2 consists of two main parts, the first describing the fact that if the pointer p points to one of the objects from the set O (computed by the *points-to* analysis), then the value of the field f of the pointer p is equal to the value of the field f of the object that p points to. The second part describes that if p does not point to any object in the set O, then the value of the field f is unknown, i.e., the read is invalid.

Similarly, writing to a memory through an L-value expression $p \to f$, can be represented by the formula [13]:

$$\bigwedge_{o \in O} o.f = (p = \&o \wedge g_i^{os}) \; ? \; drf(p).f \; : \; o.f \tag{2.3}$$

expressing the fact that $o.f$ gets updated if p points to the object o, otherwise its value remains intact. A new unconstrained Boolean variable g_i^{os} is introduced to ensure that the value of field f is changed in only one of the concrete objects abstracted by o and the others remain unchanged.

This approach brings a significant advantage with regard to performance since it is sufficient to compute the *points-to* analysis only once (before the main analysis) which removes the need of keeping track of pointers during abstract interpretation of the program. However, it also has its drawbacks. It is not possible to modify sections of SSA which manipulate the dynamic memory model since the changes could influence the set of objects which a pointer can be dereferenced into and hence the *points-to* analysis would have to be recomputed. This is especially a problem for unwinding of loops that contain memory allocations, as described in detail in Section 3.2.2.

[1]In some cases, multiple abstract objects are used for an allocation site to ensure soundness. This is not relevant for this work, though, hence, for simplicity, we only assume a single abstract dynamic object for each `malloc`.

Verifying Memory Safety When analysing programs with dynamic memory, oftentimes errors stem from invalid pointer operations rather than user-defined assertions. 2LS supports analysis of such errors by instrumenting the processed GOTO program with new assertions which check that the memory operations present in the program are valid. This includes assertions for pointer dereferencing safety, `free` safety and absence of memory leaks.

To check for dereferences of `null`, for each expression containing a dereference $*p$ at location i, we need to verify that the assertion $p_j \neq null$ holds with p_j being the version of p valid at i [13].

When checking if a `free` call is valid, we need to make sure that the given pointer has not been freed already (this is an error referred to as double `free`). The same property should be verified when dereferencing a pointer. To facilitate this type of analysis, 2LS introduces a new special variable, its full name is $\_\_\_CPROVER\_deallocated$, for simplicity, we will refer to it as $deall$. This variable is initialised to `null` and it is then non-deterministically set to the address of the object to be freed in a `free` call. In particular, a call of the form $free(p)$ at a program location i is replaced by a formula $deall = g_i^{deall} \; ? \; p_j \; : deall_k$ with p_j and $deall_k$ being versions of p and $deall$ valid at i, respectively, and g_i^{deall} being a free Boolean variable.

Then, similarly to `null` dereference analysis, 2LS must verify that in every location containing a pointer dereference $*p$ or a $free(p)$ call, the assertion $p_j \neq deall_k$ holds. This approach is sound thanks to $deall$ being an over-approximation of all the freed addresses. However, it is also often imprecise; for an abstract object representing a set of concrete objects, freeing one concrete object does not mean that all the objects were freed [13]. Precision of this approach is improved by modifying the way `malloc` calls are transformed to the SSA form. At each allocation site, 2LS introduces one more concrete object ao_i^{co} which is guaranteed to be allocated only once (this is ensured by checking that no pointer points to this object). Then, for each `malloc` and `free` call, we only allow these special concrete objects to be assigned to $deall$, hence the `free` safety check is also done only on the concrete objects, which reduces imprecision.

Absence of memory leaks is verified similarly to the `free` safety. A new variable, $\_\_\_CPROVER\_memory\_leak$, is introduced for this purpose; it is initialised to `null`. Inside `malloc` calls, the variable is non-deterministically set to the result of the given `malloc` call. Then, inside `free` calls, if the value of $\_\_\_CPROVER\_memory\_leak$ is equal to the pointer freed in the given `free`, $\_\_\_CPROVER\_memory\_leak$ is reset to `null`. Finally, 2LS verifies that at the end of the program the assertion $\_\_\_CPROVER\_memory\_leak = null$ holds.

2.2 Techniques Used for Verification

There are a lot of approaches in the field of software verification, each with its own advantages and disadvantages. For example, inference of inductive invariants is a sound method (meaning that if the verification claims that the program is correct with regard to the specification, it is indeed correct), however, this approach does not scale well to larger systems. On the other hand, bounded model checking scales significantly better but it cannot prove correctness of the system and hence can be only used to find errors.

2LS tries to combine multiple approaches into a single algorithm called kIkI which aims to mitigate the downsides that each approach would have on its own. This section gives an overview of its main components – abstract interpretation, k-induction and bounded model checking – and describes how they are all connected.

2.2.1 Representing Programs as Logical Formulae

Since it is difficult to analyse C programs directly, most analysers make use of internal representations during analysis. 2LS uses logical formulae which can easily be created from the SSA form described in Section 2.1.2. The main advantage of using logical formulae for analysis is the possibility of passing the formulae directly to an automatic SMT or SAT solver while reasoning about the program's properties.

The state of a program is defined by an interpretation of logical variables corresponding to the program variables. For a vector of variables $\boldsymbol{x}$, a predicate $Init(\boldsymbol{x})$ describes the initial set of states. A transition relation of the program is described by a formula $Trans(\boldsymbol{x}, \boldsymbol{x}')$. Based on these two predicates, the set of reachable states can be computed as the least fixed-point of the transition relation starting from the set of states described by $Init(\boldsymbol{x})$. The transition relation in 2LS is encoded using the SSA form. Since the SSA form is computed from a valid C program without syntactic and semantic errors (this is checked by the CProver framework) and it is acyclic (as described in Section 2.1.2), the formula derived from the SSA form is satisfiable (there are no contradictions in a valid source program). Henceforth, we will consider the transition relation $Trans(\boldsymbol{x}, \boldsymbol{x}')$ to be satisfiable in the following sections.

Oftentimes, the goal of verification is to show that the set of reachable states is error-free. In the following sections, we will use a predicate $Err(\boldsymbol{x})$ in order to describe that the state $\boldsymbol{x}$ (described by the values of program variables) contains an error, e.g., an assertion does not hold.

2.2.2 Abstract Interpretation

Abstract interpretation is one of the most prominent methods in the area of static analysis. Instead of focusing on concrete states of a program (which is problematic as the set of all reachable states may not be computable), abstract interpretation focuses on their abstraction. This is often sufficient because analyses usually reason only about some properties of the program. For example, instead of focusing on all possible combinations of all program variables, we only focus on a variable x whose values we want to reason about.

For a concrete domain P of program states, abstract interpretation defines the abstract domain Q where each element corresponds to an element in the concrete domain. We also define the concretisation ($\alpha : P \rightarrow Q$) and abstraction ($\gamma : Q \rightarrow P$) functions mapping between these two domains.

An abstract interpretation I of a program is then a tuple [5]:

$$I = (Q, \sqcup, \sqsubseteq, \top, \bot, \mathcal{T}^{\#}) \tag{2.4}$$

where

- Q is the abstract domain with its concretisation and abstraction functions,
- $\top \in Q$ is the supremum of Q,
- $\bot \in Q$ is the infimum of Q,

- $\sqcup$: $Q \times Q \to Q$ is the join operator combining multiple abstract states into one, $(Q, \sqcup, \top)$ is a complete semilattice,
- $\sqsubseteq \subseteq Q \times Q$ is an ordering on $(Q, \sqcup, \top)$ defined as $x \sqsubseteq y \Leftrightarrow x \sqcup y = y$,
- $\mathcal{T}^{\#}$: $\mathit{Instruction} \times Q \to Q$ defines the interpretation of abstract transformers.

The framework of abstract interpretation approximates the set of reachable states by computing the least fixpoint of $\mathcal{T}^{\#}$ in the abstract domain. In order to ensure soundness of the analysis, there must be a Galois connection on $(P, \leq, Q, \sqsubseteq)$ meaning that $\forall p \in P, \forall q \in Q$:

$$p \leq \gamma(q) \Leftrightarrow \alpha(p) \sqsubseteq q. \tag{2.5}$$

The computed abstract value is an over-approximation of the set of all reachable concrete program states, hence this approach suffers from possible false positives (i.e., a situation when a property does not hold in the abstract interpretation framework but holds for the set of reachable program states).

2.2.3 Bounded Model Checking

While abstract interpretation over-approximates the semantics of a program, another technique, Bounded Model Checking (BMC), under-approximates the program in order to find violation of properties and the corresponding counterexamples [1]. BMC is based on checking program paths bounded by a chosen integer $k \in \mathbb{N}$ and unwinding the transition relation of the program. BMC introduces a predicate $T[k]$ describing k steps of the transition relation:

$$T[k] = \bigwedge_{i=0}^{k-1} \mathit{Trans}(\boldsymbol{x}_i, \boldsymbol{x}_{i+1}) \tag{2.6}$$

BMC also defines a predicate $P[k]$ describing k states being error-free:

$$P[k] = \bigwedge_{i=0}^{k-1} \neg \mathit{Err}(\boldsymbol{x}_i) \tag{2.7}$$

Using $T[k]$ and $P[k]$ as defined above and $\mathit{Init}(\boldsymbol{x})$ describing the set of initial states, finding property violations up until an unwinding limit k using BMC can be formalised as checking the satisfiability of the formula:

$$\exists \boldsymbol{x}_0 \dots \boldsymbol{x}_k . \mathit{Init}(\boldsymbol{x}_0) \wedge T[k] \wedge \neg P[k+1] \tag{2.8}$$

This approach, however, suffers from false negatives depending on the bound k. It is a situation where a property holds for the kth unwinding but it does not hold for some lth unwinding where $l > k$. In order to avoid the choice of a fixed bound k, incremental bounded model checking (IBMC) can be used [2]. It repeatedly uses BMC starting from $k = 0$ and increasing linearly. In each step, it assumes that the previous steps were error-free, simplifying the test for property violation:

$$\exists \boldsymbol{x}_0 \dots \boldsymbol{x}_k . \mathit{Init}(\boldsymbol{x}_0) \wedge T[k] \wedge P[k] \wedge \mathit{Err}(\boldsymbol{x}_k) \tag{2.9}$$

2.2.4 k-Induction

The k-induction technique can be seen as an extension of IBMC that can find property violations as well as prove true properties [2]. The concept of IBMC is extended with the concept of a k-inductive invariant which is a predicate $KInv$ having the following property:

$$\forall \boldsymbol{x}_0 \ldots \boldsymbol{x}_k . I[k] \wedge T[k] \implies KInv(\boldsymbol{x}_k) \tag{2.10}$$

where $I[k]$ represents the fact that $KInv$ must hold for all previous states:

$$I[k] = \bigwedge_{i=0}^{k-1} KInv(\boldsymbol{x}_i). \tag{2.11}$$

A system is safe if and only if there is a k-inductive invariant $KInv$ which satisfies [2]:

$$\begin{aligned}\forall \boldsymbol{x}_0 \ldots \boldsymbol{x}_k . (Init(\boldsymbol{x}_0) \wedge T[k] \Rightarrow I[k]) \wedge \\ (I[k] \wedge T[k] \Rightarrow KInv(\boldsymbol{x}_k)) \wedge \\ (KInv(\boldsymbol{x}_k) \Rightarrow \neg Err(\boldsymbol{x}_k))\end{aligned} \tag{2.12}$$

2.2.5 Template-based Verification

One of the main problem of k-induction is that the inference of k-inductive invariants is costly. Moreover, directly using a solver to find $KInv$ would require handling second-order logic. Reasonably efficient solvers of such kind are currently not available, hence 2LS reduces the problem to first-order logic by iteratively using a first-order solver. This is done by restricting the form of the inductive invariant $KInv$ to the form $\mathcal{T}(\boldsymbol{x}, \boldsymbol{\delta})$ where $\mathcal{T}$ is a fixed expression, a so-called template, over program variables $\boldsymbol{x}$ and template parameters $\boldsymbol{\delta}$. Choosing a fixed template corresponds to the choice of an abstract domain in abstract interpretation – it only captures some properties of the program. By using a template, invariant inference is reduced to a first-order search for template parameters $\boldsymbol{\delta}$ [2]:

$$\begin{aligned}\exists \boldsymbol{\delta} . \forall \boldsymbol{x}_0 \ldots \boldsymbol{x}_k . (Init(\boldsymbol{x}_0) \wedge T[k] \implies \mathcal{T}[k](\boldsymbol{\delta})) \wedge \\ (\mathcal{T}[k](\boldsymbol{\delta}) \wedge T[k] \implies \mathcal{T}(\boldsymbol{x}_k, \boldsymbol{\delta}))\end{aligned} \tag{2.13}$$

where $\mathcal{T}[k](\boldsymbol{\delta}) = \bigwedge_{i=0}^{k-1} \mathcal{T}(\boldsymbol{x}_i, \boldsymbol{\delta})$. Although the problem is now expressible in first-order logic, the formula contains quantifier alternation which poses a problem for the current SMT solvers. 2LS solves this by iteratively checking the negated formula (to turn $\forall$ into $\exists$) for various choices of constants d as candidates for the values of the parameters δ. The template formula $\mathcal{T}(\boldsymbol{x}, d)$ is an invariant if Formula 2.14 is unsatisfiable.

$$\begin{aligned}\exists \boldsymbol{x}_0 \ldots \boldsymbol{x}_k . \neg(Init(\boldsymbol{x}_0) \wedge T[k] \implies \mathcal{T}[k](d)) \vee \\ \neg(\mathcal{T}[k](d) \wedge T[k] \implies \mathcal{T}(\boldsymbol{x}_k, d))\end{aligned} \tag{2.14}$$

The algorithm for invariant inference takes an initial value of $d = \bot$ and iteratively solves the second part of the disjunction in Formula 2.14 using an SMT solver:

$$\mathcal{T}[k](d) \wedge T[k] \wedge \neg\mathcal{T}(\boldsymbol{x}_k, d). \tag{2.15}$$

If the formula is unsatisfiable, then the formula $\mathcal{T}(\boldsymbol{x}_k, d)$ is an invariant; otherwise the model of satisfiability returned by the solver is joint with the previous abstract value d using a domain-specific join operator [2].

Guarded Templates Since 2LS uses the SSA form rather than control flow graphs, templates cannot be used directly. Instead, so-called *guarded templates* are used. A guarded template is a conjunction of expressions, so-called template rows, where each row r follows the form $G_r \implies \mathcal{T}_r$. The formula G_r is a conjunction of the SSA guards associated with the definition of variables occurring in $\mathcal{T}_r$. This makes sure that the computed property is valid only if the variables used inside it are defined.

For instance, a guarded loop invariant of a loop modifying a vector of variables x_l, has the form:

$$(g_{lh} \wedge g_{lh}^{ls}) \implies \mathcal{T}(x_l, \delta) \tag{2.16}$$

where lh is the program location of the loop head of loop l and guard g_{lh} expresses the reachability of the loop l from the beginning of the program. Guard g_{lh}^{ls} is a free loop-select variable driving the choice between values of variables coming from before the loop and from the end of the loop body as described in Section 2.1.2. If $g_{lh} \wedge g_{lh}^{ls}$ is equal to *true*, the loop has been reached and the loop-back variables are defined hence the loop invariant defined by the template $\mathcal{T}(x_l, \delta)$ constraining the values of variables x_l can be used.

Following the example given in Figures 2.1 and 2.4, we demonstrate how loop invariants are calculated for the template polyhedra abstract domain, in particular its special type, the interval abstract domain. In the example, there is only one loop-back variable, hence $x_l = [x_{10}^{lb}]$. The template for interval abstract domain has the form:

$$\mathcal{T}([x_{10}^{lb}], (d_1, d_2)) \equiv x_{10}^{lb} \geq d_1 \wedge x_{10}^{lb} \leq d_2 \tag{2.17}$$

where d_1 and d_2 are template parameters which are to be inferred during the analysis. The template expresses that all reachable values of x_{10}^{lb} lie in the interval $[d_1, d_2]$. To simplify the example, we will only consider 1-inductive invariants which transforms Formula 2.15 to the form:

$$\mathcal{T}(x_0, \delta) \wedge Trans(x_0, x_1) \wedge \neg\mathcal{T}(x_1, \delta). \tag{2.18}$$

Since the transition relation $Trans(x_0, x_1)$ in 2LS is represented using the SSA form described in Section 2.1.2 which has been made acyclic, we can assume that it is satisfiable. Moreover, in every iteration, we only solve the current instances of the invariant. In Formula 2.18, there are two instances of the template. The first instance, $\mathcal{T}(x_0, \delta)$, describes the loop invariant for the program state before the execution of the loop. Its guarded form is:

$$(g_4 \wedge g_{10}^{ls}) \implies \mathcal{T}([x_{10}^{lb}], (d_1, d_2)). \tag{2.19}$$

The second instance of the template, $\mathcal{T}(x_1, \delta)$, describes the loop invariant after execution of the loop. For the variable x, the SSA instance corresponding to its value from the end of the loop is x_6. Its definition is guarded by guard g_7. Therefore, the guarded form of the second template instance is:

$$(g_4 \wedge g_7) \implies \mathcal{T}([x_6], (d_1, d_2)). \tag{2.20}$$

The inference begins with setting the initial value $\delta = \perp$ with $\mathcal{T}(x, \perp) \equiv false$. The initial formula to solve (described in Formula 2.18) is:

$$(g_4 \wedge g_{10}^{ls}) \implies false \wedge \neg((g_4 \wedge g_7) \implies false). \tag{2.21}$$

To satisfy the formula, $g_4 \wedge g_{10}^{ls}$ must evaluate to $false$. Since g_4 is trivially true, the only way to find a model of the formula is to set $g_{10}^{ls} = false$. This corresponds to the first iteration of the loop, since the Φ node gets the value of x from before the loop ($x_5^{phi} = x_2 = 0$). The value of x from the end of the loop is $x_6 = 1$ and it is used to improve the current invariant such that $d_1 = d_2 = 1$.

In the second iteration, the newly updated invariant is used and the formula to solve becomes:

$$\begin{aligned}(g_4 \wedge g_{10}^{ls}) &\implies (x_{10}^{lb} \geq 1 \wedge x_{10}^{lb} \leq 1) \wedge \\ \neg((g_4 \wedge g_7) &\implies (x_6 \geq 1 \wedge x_6 \leq 1)).\end{aligned} \tag{2.22}$$

To satisfy this formula, the solver must use $g_{10}^{ls} = true$, $x_{10}^{lb} = 1$, and hence $x_6 = 2$. Using this computed value causes the template to be updated to $d_2 = 2$; the candidate invariant after the second iteration is:

$$x_{10}^{lb} \geq 1 \wedge x_{10}^{lb} \leq 2. \tag{2.23}$$

Similarly, the template would be enhanced in the subsequent iterations until the formula is unsatisfiable. This occurs once the template has the form (including the guards):

$$(g_4 \wedge g_{10}^{ls}) \implies x_{10}^{lb} \geq 1 \wedge x_{10}^{lb} \leq 10 \tag{2.24}$$

meaning that the equation to solve by the solver is:

$$\begin{aligned}(g_4 \wedge g_{10}^{ls}) &\implies (x_{10}^{lb} \geq 1 \wedge x_{10}^{lb} \leq 10) \wedge \\ \neg((g_4 \wedge g_7) &\implies (x_6 \geq 1 \wedge x_6 \leq 10)).\end{aligned} \tag{2.25}$$

In this case, the guard $g7 = \neg(x_5^{phi} \geq 10) \wedge g_4$ is $false$ and $x_6 \geq 1 \wedge x_6 \leq 10$ is false too, hence the second part of the conjunction is false and the whole formula is unsatisfiable. The computed invariant can be then used to prove that the assertions in the program always hold.

2.2.6 *k*I*k*I Algorithm

In the previous sections, we gave an overview of common verification algorithms, each with its own advantages and main areas of use. Generally, these can be summarised as follows:

Abstract Interpretation is useful for proving true properties of a program by over-approximating its semantics in an abstract domain. However, it suffers from false positives.

Bounded Model Checking may be used for finding property violations along with counterexamples. It is not suitable for proving true properties due to possible false negatives caused by the choice of bound k.

***k*-induction** can prove true properties as well as find violations and counterexamples.
However, computing k-inductive invariants is rather expensive.

2LS uses a combination of these approaches called the kIkI algorithm. Its main concept can be seen in Figure 2.5. Initially, it sets $k = 1$ and checks whether the initial program states contain errors. Afterwards, a k-inductive invariant is computed in an abstract domain and assumptions that the checked property holds for all previous states are added. The computed invariant is checked whether it is sufficient to prove safety [2]. If a property was violated, BMC is used to check whether the violation is reachable. In case it is not reachable, the counterexample may be spurious and the process is repeated with a higher k. This algorithm may produce an inconclusive result if a maximal k is reached and a counterexample or a sufficient invariant have not been found up until this point.

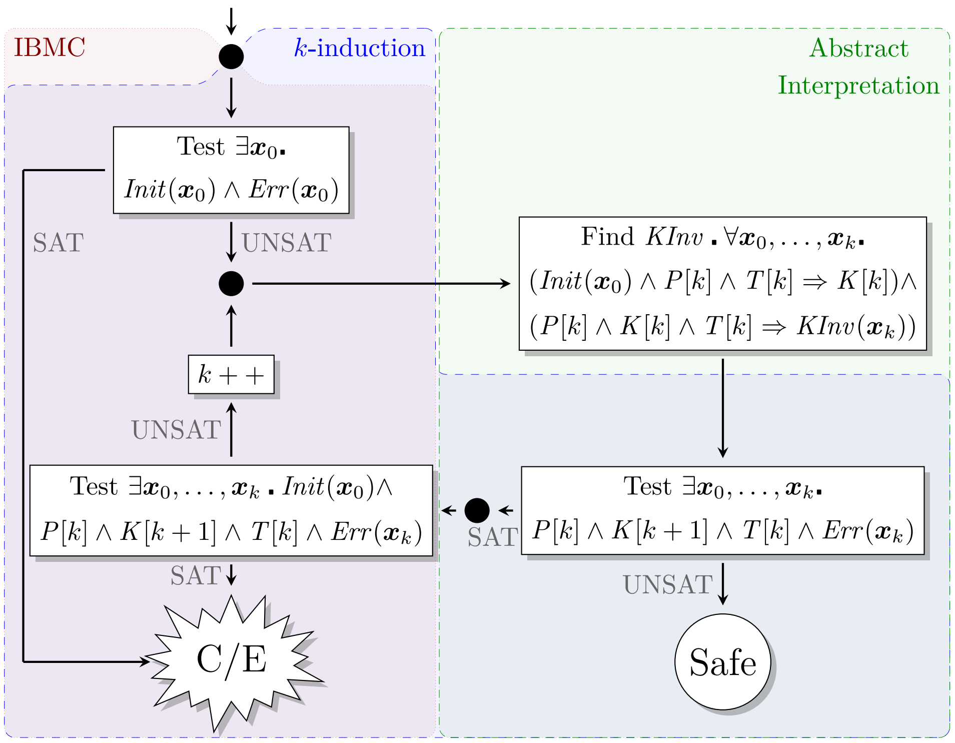

IBMC
k-induction
Abstract Interpretation
Test ∃x₀. Init(x₀) ∧ Err(x₀)
SAT
UNSAT
Find KInv . ∀x₀, …, x_k. (Init(x₀) ∧ P[k] ∧ T[k] ⇒ K[k])∧ (P[k] ∧ K[k] ∧ T[k] ⇒ KInv(x_k))
k++
UNSAT
Test ∃x₀, …, x_k . Init(x₀)∧ P[k] ∧ K[k+1] ∧ T[k] ∧ Err(x_k)
SAT
Test ∃x₀, …, x_k. P[k] ∧ K[k+1] ∧ T[k] ∧ Err(x_k)
SAT
UNSAT
C/E
Safe

Figure 2.5: The kIkI algorithm [2]

Chapter 3

Loop Unwinding in Software Verification

Loop unwinding (also referred to as loop unrolling) is a technique commonly utilised in compilers (as an optimisation technique) [16] and in software verification. The concept of this technique is simple; a loop is prepended with several copies of the loop body. In compilers, this may reduce the overhead of a loop (by reducing the number of jumps) and also enable more complex optimisations to take place [16].

In software verification, the main use-cases for loop unwinding are Bounded Model Checking and k-induction which unroll the transition relation of the program as described in Section 2.2.3 and Section 2.2.4. In order to unroll the transition relation and check for violation of properties in programs with loops, the loops must first be unwound so that the BMC framework can analyse the program. Another common task where unwinding may be used is non-termination analysis [12]. One of the approaches to non-termination analysis incrementally unwinds the given loop and tries to find two unwindings in which the program states (defined by interpretation of program variables) are identical [14].

Many state-of-the-art analysers utilise loop unwinding of some sort, e.g., CBMC [18][9], ESBMC [6][15], Dartagnan [10], or Ultimate Automizer [7]. The verification approach in 2LS is, however, vastly different from these tools due to using an SSA form (and a special dynamic object representation) with an SMT solver, therefore, loop unwinding requires a special approach. 2LS already features a specific method for loop unwinding, however, it suffers from many problems. The most important one is that it cannot be used for programs working with dynamically allocated memory, which greatly limits its applications. Therefore, in this work, we propose a way to improve the loop unwinding in 2LS. To this end, we use the unwinding method from the CProver framework, which underlies 2LS.

In this chapter, we describe the current state of this method in CBMC – the C Bounded Model Checker from the CProver framework – and also give an overview of the current unwinding method in the 2LS framework. We describe its special properties and concentrate on its defects.

3.1 Loop Unwinding in CBMC

Since CBMC is a part of the CProver framework, it makes use of the GOTO program intermediate representation described in Section 2.1.1. For verification purposes, unwinding is performed in the usual manner based on backwards `GOTO` instructions (which are in most cases a result of transformation of `while` and `for` loops) with one caveat. The loop body is copied k times with each copy being guarded by an `if` statement containing a condition equivalent to the condition of the loop that is being unwound. The `if` statements are necessary to cover cases where k is larger than the actual number of iterations required [4]. To ensure that the chosen bound k is sufficient, a new assertion called *unwinding assertion* is introduced after the copies of the loop body containing the negation of the former loop condition. If the bound k does not completely unwind the loop, the BMC framework will find a counterexample for the *unwinding assertion* and increase the unwinding bound, or report to the user that the specified bound is not sufficient. The concept of *unwinding assertions* is demonstrated in Figure 3.1 on an example of a simple loop which is unwound two times and an assertion is added after the two copies of the loop body.

```
while (x < 10)
    x++;
```

```
if (x < 10)
    x++;
if (x < 10)
    x++;
assert(!(x < 10));
```

Figure 3.1: Unwinding of a simple loop in C with an *unwinding assertion*

The unwinding strategy used in CBMC revolves around fully unwinding a single loop using an incremental approach before proceeding to the next loop [18]. For use-cases different from BMC (e.g., program pre-processing), the *unwinding assertion* may not be necessary, hence the CProver framework also provides 2 alternative modes. The first mode is referred to as *partial* unwinding; it is identical to the behaviour described above, however the *unwinding assertion* is not introduced. The second mode is referred to as the *continue* mode: the original loop is kept intact and new copies of the loop body are prepended.

3.2 Loop Unwinding in 2LS

In the current implementation of 2LS, loops are unwound in the SSA form. Unlike in CBMC, 2LS unwinds all loops up to the given bound k rather than unwinding a single loop until it is fully unwound. Unwinding is performed in the usual manner; the loop body is simply copied [2]. The unwound SSA nodes are suffixed with a number based on the current unwinding (suffix 0 denoting the original loop) to distinguish between nodes in various unwindings.

The topmost loop head multiplexer is kept and its loop-back variable is constrained with the bottommost unwinding. The variables inside a new unwinding must be connected to the existing loop or unwindings using new SSA equalities. After the loop, values of variables from potential loop exits must be merged since the loop may exit at various points. This is done using a choice based on the satisfiability of guard conditions of the individual unwindings. An unwinding of a simple loop in SSA can be seen in Figure 3.2. We can observe that the original loop along with the newly unwound loop body now form the new loop which is represented using unconstrained loop-back and loop-select variables as discussed in Section 2.1.2. The new unwinding is connected to the loop body using variable equalities and the exit value of the loop is merged from the potential exit points based on the guard and the condition of the unwound body.

$$\begin{tikzcd}
\text{before the loop } (x_0) \arrow[d] \\
\begin{array}{c}\text{loop head multiplexer}\\ x_1^{phi}\%1 = g_2^{ls}\%0\ ?\ x_2^{lb}\%0\ :\ x_0\end{array} \arrow[d] \arrow[dddddd, bend right=60] \\
\begin{array}{c}\text{loop body 1}\\ x_2\%1 = \ldots\end{array} \arrow[d] \\
\begin{array}{c}\text{connect unwinding}\\ x_1^{phi}\%0 = x_2\%1\end{array} \arrow[d] \arrow[ddd, bend right=50] \\
\begin{array}{c}\text{loop body 0}\\ x_2\%0 = \ldots\end{array} \arrow[d] \\
\text{end of loop body } (x_2^{lb}\%0) \arrow[uuuu, bend right=60, "x_2^{lb}\%0"'] \\
\begin{array}{c}\text{exit point merge}\\ x_1^{phi} = g_1\%1 \land c_1\%1\ ?\ x_1^{phi}\%1\ :\ x_1^{phi}\%0\end{array} \arrow[d] \\
\text{after the loop}
\end{tikzcd}$$

Figure 3.2: Unwinding of a simple loop in SSA in 2LS

The way unwinding is performed is incremental, in the sense that the construction of the formula for the solver is monotonic [2]. This allows 2LS to make use of incremental SAT solving which increases its efficiency when unwinding is being performed.

3.2.1 Assertion Handling in 2LS Unwinder

When operating in the BMC or the k-induction modes, 2LS also manipulates the assertions present in the program to improve precision of its analysis. These modifications are done starting from the second unwinding, meaning that the innermost unwinding is always kept untouched.

Firstly, assertions in the unwindings are converted to so-called *constraints*. Unlike assertions, *constraints* are pushed to the SAT solver (i.e., it is assumed that they hold) instead of being checked for satisfiability. This behaviour corresponds to the IBMC assumption that the previous states have been checked and are proven to be error-free as described in Section 2.2.3. Figure 3.3 demonstrates a program where this modification facilitates verification. In the second iteration of the loop, the values of variables x, y, and z return to their initial state. Henceforth, by introducing a *constraint* after the second unwinding, 2LS can now verify that the assertion in the original loop body holds based on the equality of variables in the loop and in the second unwinding.

Secondly, if the analysed program contains an assertion after a loop, so-called *hoisted constraints* are newly introduced to the unwinding when 2LS is operating in the k-induction mode. These logically connect the condition under which a loop is exited with an assertion after the loop. For each assertion a present after the given loop, a constraint in the form

```
void main() {
  int x = 1, y = -1, z = 1;

  while (1) {
    z = y;
    y = x;
    x = -x;
    assert(x == z);
  }
}
```

Figure 3.3: A simple non-terminating program where *constraints* facilitate verification

$cond \implies a$ is introduced where *cond* is a condition under which the given assertion is reachable – a disjunction of exit conditions from all unwindings in the preceding loop. For some programs, this enables proving true properties using k-induction, e.g., when there is an `assert(0)` call after a loop which does not terminate. An example of such a program is given in Figure 3.4. Starting from the second unwinding, 2LS adds a constraint in the form $g_4\%n \wedge c_4\%n \implies \neg g_9$ with n being the current unwinding number, c_4 the condition under which the loop exits, g_4 the guard of the loop, and $\neg g_9$ being the assertion condition. By assuming that such a constraint holds, if the SAT solver returns a model where the condition is false (and hence the loop never ends), 2LS can verify that the assertion after the loop holds thanks to the hoisted constraint.

```
 1 void main() {
 2   int x = 1, y = -1, z = 1;
 3
 4   while (x != y) {
 5     z = y;
 6     y = x;
 7     x = -x;
 8   }
 9   assert(0);
10 }
```

Figure 3.4: A simple non-terminating program where *hoisted constraints* facilitate verification

3.2.2 Deficiencies of the Approach

The implemented approach is very efficient, however it has some problems as was outlined in Section 2.1.3. One of the minor downsides is that the unwound SSA is not easily readable by humans since the output order does not match the program control flow. What is much more problematic is the fact that the simple copying and renaming approach does not support programs working with dynamic memory. Figure 3.5 shows a simple example C program where the current implementation fails while using k-induction (2LS claims that the program is correct making the analysis unsound). A linked list of an unknown length is constructed, all nodes contain the value 1 except for the third node which contains the value 2. The second loop then expects all the nodes to contain value 1, which does not hold in all executions.

```
int main() {
  List t;
  List p = 0;
  int i = 0;
  while (__VERIFIER_nondet_int()) {
    t = (List) malloc(sizeof(struct node));
    t->h = i == 2 ? 2 : 1;
    t->n = p;
    p = t;
    i++;
  }
  while (p != 0) {
    assert(p->h == 1);
    p = p->n;
  }
}
```

Figure 3.5: A simple program using dynamic memory with an error

The problem with the current implementation is that abstract dynamic objects of the program are introduced once, before the beginning of analysis. During the construction of SSA, the static *points-to* analysis is computed based on the dynamic object instances. Then, pointer operations are encoded based on this analysis as described in Section 2.1.3. However, when unwinding the loops, the call to function `malloc` is copied, which should result in new dynamic objects being introduced. This addition should also invalidate the previously calculated *points-to* analysis since the pointer inside the loop can now also point to the new dynamic objects created in the unwinding. Henceforth, it is also necessary to update the memory reads and writes in the SSA form. This is not achievable by simply copying the relevant parts of the SSA form since dynamic objects are computed only once before the analysis and inserted into the GOTO program representation.

Chapter 4

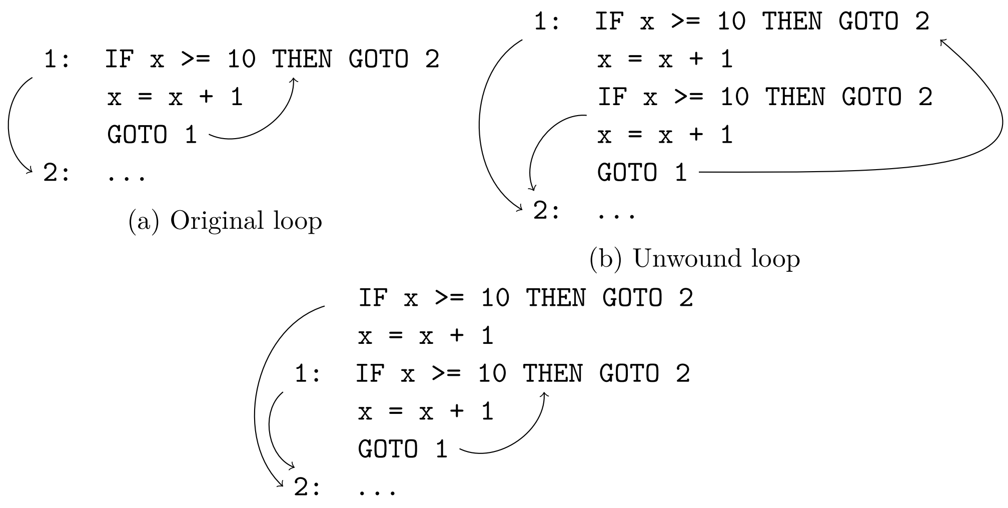
Design of the New Unwinding Solution

To overcome the issues described in the previous chapter, we propose an alternative solution to unwinding for 2LS. By unwinding in the GOTO program representation rather than in the SSA form, we can update the set of dynamic objects in the program as well as compute a new *points-to* analysis based on the newly introduced dynamic objects. Section 4.1 gives an overall overview of the proposed approach, the individual components of our solution are then expanded on in further sections.

4.1 The Main Unwinding Loop

Unwinding in 2LS is done on demand during analysis, either before abstract interpretation (if the user requests it) or incrementally as a part of the BMC and the k-induction modes. When unwinding is requested, we first need to unwind all loops in the GOTO program. Since one of the important features of 2LS is inference of loop invariants, we need to make sure that loops exist even after the unwinding is done. Hence, we use the *continue* mode of CProver's GOTO unwinder. Afterwards, further modifications to the GOTO program are required to make the program suitable for analysis by 2LS. These include correctly introducing new dynamic objects and updating checks related to correctness of memory operations based on the new dynamic objects. We also need to correctly connect the unwound loops; in order for some analyses to work correctly, the original loop and the newly unwound body must be part of a larger loop as was shown in Figure 3.2.

We must consider that unwinding may be done multiple times during analysis (e.g., as a part of k-induction). By modifying the loop connections to form the large loop covering all the unwindings, we remove information about the original loops. If we were to unwind such program multiple times, we would copy not just the original loop but also the previously unwound loop body along with it, which is not a desired behaviour. Therefore, we need to keep track of the original loop connections in the program and reset the loop connections to their original state before every unwinding so that only a single body is copied. Figure 4.1 gives an example of this process; first a loop is unwound and the target of its backwards `GOTO` is modified to form the full loop including the unwinding (Figure 4.1b). Then, before unwinding the GOTO program for the second time, the loop connection is reset (Figure 4.1c) so that the instructions to be unwound can be detected correctly.



```
1:  IF x >= 10 THEN GOTO 2
    x = x + 1
    GOTO 1
2:  ...
```

(a) Original loop

```
1:  IF x >= 10 THEN GOTO 2
    x = x + 1
    IF x >= 10 THEN GOTO 2
    x = x + 1
    GOTO 1
2:  ...
```

(b) Unwound loop

```
    IF x >= 10 THEN GOTO 2
    x = x + 1
1:  IF x >= 10 THEN GOTO 2
    x = x + 1
    GOTO 1
2:  ...
```

(c) Reset loop connection before the next unwinding

Figure 4.1: Connecting loops inside GOTO program

Once all these modifications to the GOTO program are done, we can compute the new SSA representation. During this process, a new *points-to* analysis is computed, henceforth the dynamic objects in SSA reflect the new state of dynamic objects inside the GOTO program. At last, several modifications to the computed SSA have to be performed, e.g., constraints and hoisted constraints are introduced as described in Section 3.2.1. Figure 4.2 gives an overview of all the steps required during unwinding.

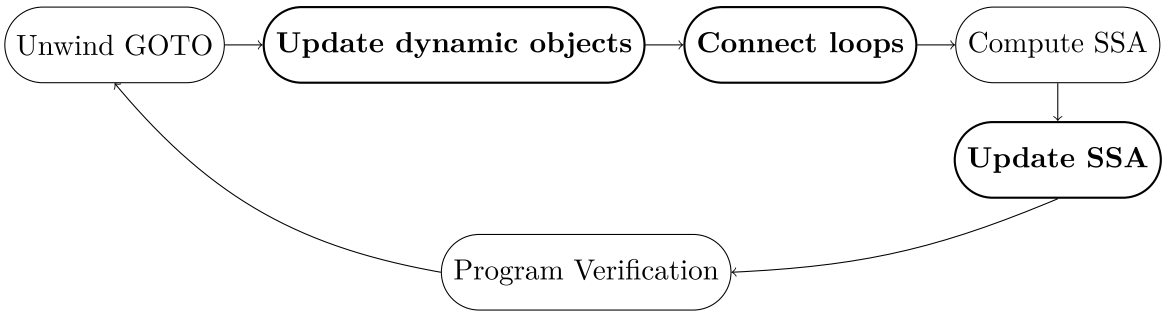


Figure 4.2: The main unwinding loop

4.2 Unwinding of Dynamic Objects

As previously described, dynamic objects are initialised before creating the SSA and their names are set based on the number of the GOTO location where they are created. However, the GOTO unwinder provided by CProver cannot correctly handle unwinding of these objects since they are specific to 2LS. After unwinding, we need to rename the objects based on the new location numbers. Creating of a dynamic object can be detected based on the `#malloc_result` flag which 2LS sets on the assignment expression containing a `malloc` call.

The expression tree of such an assignment must be traversed and the old dynamic objects are renamed in all subtrees of the AST according to the freshly renamed dynamic objects. While updating the assignment, it is also necessary to update the condition under which the concrete dynamic object (used for verifying `free` safety, see Section 2.1.3) is allocated since the set of dynamic objects has changed. The recursive AST traversal renaming dynamic objects can be seen in Algorithm 1. The location in dynamic objects is updated based on the new location numbers and concrete object selection condition is updated based on the new dynamic objects.

Algorithm 1: Renaming dynamic objects

```
Function renameDynamicObjects(loc, expr):
    Input: loc: GOTO location of the malloc call
           expr: AST subtree being processed
    if expr is dynamic object then // Update existing object
        update location number in expr to loc
    else if expr is concrete object selection then
        re-compute selection condition in expr
    end
    foreach $op \in operands(expr)$ do
        renameDynamicObjects(loc, op)
    end
```

4.3 Unwinding of Memory Leak Checks

Memory leak checks inside 2LS are currently not implemented strictly as was described in Section 2.1.3. Due to how abstract objects are handled, the variable keeping track of leaked pointers is not updated non-deterministically but rather is always set to the resulting pointer inside `malloc` calls. This introduces unsoundness of memory leak checks when multiple allocation sites are present (e.g., when a loop containing `malloc` has been unwound). Henceforth, additional adjustments of the unwound GOTO program are needed.

Rather than using a single variable for the entire program scope, we introduce a new variable for tracking memory leaks for each allocation site. In particular, for a location i containing a `malloc` call, we introduce the variable $\_\_CPROVER\_memory\_leak\$i$. Then, let ML be the set of all such variables. For each $v \in ML$, we insert the GOTO code depicted in Figure 4.3 right after each `free` call. This is done at the beginning of the analysis as well as after each unwinding.

```
   IF v != ptr THEN GOTO N
   v = null
N: ...
```

Figure 4.3: GOTO code to be inserted after each `free` call for a variable v and a freed pointer *ptr*

Finally, at the beginning of the program, we initialise all variables from ML to `null` and then at the end of the program, we check if the assertion

$$\bigwedge_{v \in ML} v = null \tag{4.1}$$

holds. This approach allows us to soundly track memory leaks and moreover, we can also provide information about the origin of the possible memory leak, i.e., an allocation site where the object was allocated, which is something that has not been possible in 2LS before.

4.4 Unwinding for k-induction and BMC

As discussed in Section 3.2, support for k-induction and BMC inside 2LS requires some extra modifications during unwinding. Firstly, all the unwound loop bodies must be connected into a single loop. Secondly, assertions in the SSA form need to be updated.

4.4.1 Connecting Unwound Loops

Connecting loop bodies into a single loop can be done by post-processing the unwound GOTO program after updating the dynamic objects. In order to be able to perform the loop modification, we need to detect the topmost unwinding and then update the target of the corresponding backwards `GOTO` instruction to the beginning of the first unwinding. Hence, we need to keep track of what has been unwound. This can be achieved using CProver's capabilities to set arbitrary flags onto expressions. After unwinding a single loop, CProver provides GOTO program pointers which allow distinguishing between the newly added unwindings. With this information, we can set an integer flag on the first instruction of each unwinding describing which unwinding starts on the given instruction. This integer flag will be further referred to as the *unwind number*.

Then, we can utilise this information to correctly reconnect the loop connections to facilitate k-induction and BMC while backing up the original loop connections. While iterating through a GOTO program, the topmost unwinding can be detected based on the *unwind number* flag being equal to the current unwinding. In order to support nested loops, we need to keep track of the topmost unwindings in a stack; upon finding a backwards GOTO, we change its target to the instruction kept on top of the stack and then pop it from the stack. As explained in the overall unwinding loop design (see Section 4.1), we also need to be able to reset the loop connections to their original states in order to be able to perform unwinding of only a single loop body. For this purpose, we keep track of the original `GOTO` targets in a map and then reset the `GOTO` targets before unwinding the next time. Algorithm 2 describes how the loops are connected after unwinding the GOTO program. We keep track of topmost unwindings and upon finding a backward `GOTO` instruction, we modify its target to the latest encountered topmost unwinding and store the original connection.

4.4.2 Updating Assertions

Once a new SSA form has been computed based on the unwound GOTO program, we need to update its assertions as described in Section 3.2.1 in order to improve the precision of the analysis. We can iterate the computed SSA and if the *unwind number* is higher than 2, we transform assertions into constraints and add hoisted constraints. However, these

Algorithm 2: Updating connection of loops

```
Input: $u$: The current number of unwindings
       $f$: The unwound GOTO function
Result: Backup of the original connections
1  $S = makeStack()$
2  foreach $inst \in f$ do
3      if unwind number of inst $= u$ then
4          $S.push(inst)$
5      end
6      if inst is backwards GOTO then
7          back up the target of $inst$
8          $t = S.pop()$
9          set target of $inst$ to $t$
10     end
11 end
```

transformations cannot be done when dynamic memory is used. This is caused by the fact that we introduce new dynamic objects to the program, and therefore we cannot assume (and add a constraint) that the assertion holds in the previous unwindings because, unlike with simple arithmetic without heap, we have not actually proven the previous unwinding to be error-free due to how dynamic objects are represented in 2LS.

4.5 Incremental Unwinding Design

While the approach described above is sound and allows correctly analysing programs with dynamic memory using k-induction, it does not scale as well as the original unwinding implementation. This is mainly because the SSA form must be freshly recomputed after each unwinding and hence incremental SAT solving cannot be used. As a part of this work, we propose a high-level design of an approach which uses our new unwinding method but makes use of incremental SAT solving as much as possible.

Suffixing SSA variables The overall approach to unwinding remains similar; we still need to unwind the GOTO program, rename dynamic objects, and then update the SSA. However, to make use of incremental SAT solving, we need to reuse as many parts of the SSA between unwindings as possible. That way, the SAT solver does not need to re-evaluate these parts and hence runs much faster. Hence, rather than recomputing the SSA from scratch after GOTO modifications, we only add new SSA nodes based on the unwound loop bodies. To distinguish between unwindings, a similar approach to the old SSA unwinding can be used – suffixing the SSA variables with the unwinding number. The semantics of this suffix must, however, slightly change. Previously, the loop body was copied, suffixed with the current unwinding number, and then connected as the outermost unwinding. On the other hand, when unwinding a loop in a GOTO program, the new unwound bodies are above the loop (the innermost unwinding). This means that previously going from the start of the program, unwindings were suffixed in descending order, whereas with the new approach, they will be in ascending order.

Modifications of the SSA The order of unwinding indices also influences further modifications of the SSA. Previously, 2LS modified assertions as described in Section 3.2.1 in the newly created loop bodies. However, with unwindings being created directly above the loop, we need to modify the previously created unwinding rather than the new one as constraints are not introduced in the innermost unwinding. For example, after the second unwinding during k-induction, we need to modify assertions in the previously created unwinding (the first unwinding). On the other hand, this simplifies forming of the whole loop enclosing all the unwindings, as it is sufficient to create the main Φ node once, when unwinding for the first time. Afterwards, the new unwindings need to be only connected to the previous unwinding and to the original loop body by using variable equalities and new merge exit conditions need to be introduced for this branch. To simplify these modifications and also operations with the unwound SSA (e.g., non-termination analysis), we propose introducing a map which maps an unwinding number to the beginning and the end of the SSA section containing the given unwinding.

Handling of dynamic objects In order to correctly support operations with dynamic objects, *points-to* analysis needs to be recomputed once the objects in GOTO programs are updated. However, by modifying the *points-to* analysis, SSA equalities representing operations with dynamic memory become invalid. Henceforth, we need to keep track of such nodes and calculate them again based on the newly computed *points-to* analysis. The previous equalities representing operations with dynamic memory need to be removed from the solver and the new equalities must be introduced. The incremental solver used in 2LS operates as a stack of contexts with each context containing logical formulae. To facilitate the changes to dynamic objects, we propose keeping all formulae of the SSA related to dynamic memory in a special context at the top of the context stack and popping it from the stack after each unwinding. This partially defeats the purpose of incremental SAT solving, however due to the memory model of 2LS, combination of incremental SAT solving and dynamic objects is not possible when unwinding is required.

Chapter 5

Implementation

The solution proposed in Chapter 4 has been implemented in 2LS without the usage of incremental SAT solving described in Section 4.5. At the time of writing, a pull request introducing the changes to unwinding is undergoing code review on the upstream repository of 2LS on GitHub[1]. In this chapter, we give a brief overview of some of the more significant implementation details. As a part of this work, we have also updated the version of the underlying CProver framework used in 2LS, this process is described in Section 5.1.

5.1 Update of CProver Framework in 2LS

As mentioned above, the CProver framework plays a crucial role in 2LS; it provides parsing of C programs to GOTO programs and many more useful interfaces (e.g., an abstract interpretation framework) which 2LS utilises for verification. However, previously 2LS used an old release of the CProver framework that most importantly contained multiple bugs. At the time of beginning the update process, 2LS employed CProver's 5.6 release, while the latest release was 5.37. These releases were more than 4 years apart and the range of releases encapsulated more than 15,000 commits.

Using an old version of a framework introduces various problems in the development process, e.g., outdated documentation, missing support from the upstream development team or missing features which have been added in a later version (and are listed in the current documentation of the framework but cannot be used in the code base utilising the library). The main motivation for the update in the context of this work was to get access to the latest GOTO unwinding implementation from the CProver framework available.

The update process to the latest CProver framework posed several problems. Firstly, as already discussed, the number of changes to the CProver framework was large. Moreover, changes to the interface (e.g., of classes) were not done in respect to semantic versioning [17] and oftentimes they were done in a minor release. Some of the changes made 2LS uncompilable, sometimes even whole classes which 2LS relied on were removed from the framework, making it necessary to find a substitute. Furthermore, some changes were purely semantic, meaning that 2LS would compile but would not behave as with the previous version. Fixing such errors turned out to be the most difficult aspect of the whole process.

Secondly, 2LS uses its own fork of the CProver framework. This fork contains extra commits which can be classified into 3 categories:

[1] https://github.com/diffblue/2ls/pull/161

- Back-ported bug fixes from the main repository of CProver which were implemented in later releases.
- 2LS-specific feature implementations, such as memory leak instrumentation options.
- Fixes related to International Competition on Software Verification (SV-COMP) and its witness format which has been changing throughout the years. Since 2LS uses CProver for creating witnesses, it was necessary to adjust the implementation.

Lastly, the whole process is very prone to error as there are a lot of changes that need to be done. Hence, it is necessary to carefully review the code changes and make it easy to review them. The commit history that will then be merged should also be as clean as possible. These problems make it impossible to update the version to the latest one in a single pull request as there would be too many changes (even semantic) to make 2LS work correctly and reviewing such a pull request properly would be very difficult.

With respect to all of the above, we came up with the following solution. Instead of doing one big update, we split the update process into multiple steps, each dealt with only updating CProver (and 2LS) to the next version, making the number of changes to review smaller. The goal was to have a correctly working 2LS version after each step of the update process, which would act as a backup.

Before doing all the upgrades, we first classified the extra CProver commits that 2LS uses and evaluated whether they are still necessary or if (and when) they were applied in CProver. The first step of each version update was rebasing our extra commits on top of a newer CProver version. During this step, the commits that we had classified as unnecessary or duplicate were removed. This step oftentimes caused a merge conflict which had to be resolved. Then, we created a pull request against the 2LS-specific CProver fork containing only the necessary commits.

The second step focused on adjusting 2LS to the new CProver changes. To make reviewing the changes to 2LS easier, we created a commit for each type of change describing why that change was necessary. The adjustments were often inspired by how CProver handled the change in its code base. In a lot of cases, the removed or modified methods were marked as deprecated and mentioned an alternative which could be used as a replacement. In other cases, the commit message or the pull request which introduced the change provided more context of how the adjustment should be done. To check for correct functionality and semantics, the regression test suite of 2LS was used. After thorough reviews of both created pull requests, the 2LS changes were squashed into a single commit containing a list of changes, in order to maintain a clean commit history (where every commit can be compiled without errors). Both pull requests were merged at the same time along with a patch version update.

At the time of writing, 2LS has been successfully updated to CProver 5.37 release via GitHub pull requests[2]. In total, these pull requests contained approximately 1900 code additions and 1500 code deletions which highlights the large scope of the required changes. Below are listed some of the major changes that affected 2LS code base the most:

- Constant propagation module in CProver has undergone changes and in certain releases replaced variables which were not constants. This required turning off constant propagation in 2LS and re-enabling it in a later release once it was fixed.

[2]Update to 5.7 (https://github.com/diffblue/2ls/pull/148), 5.8 (https://github.com/diffblue/2ls/pull/149), 5.9 (https://github.com/diffblue/2ls/pull/150), 5.10 and 5.11 (https://github.com/diffblue/2ls/pull/151), 5.12 (https://github.com/diffblue/2ls/pull/152), 5.37 (https://github.com/diffblue/2ls/pull/155)

- Simplified interface for GOTO program parsing allowed removal of some 2LS code segments which were duplicated from CProver.
- CProver modified memory assertions for structure dereferencing. Previously, expression $p \to n$ where p is a pointer and n is a structure field would result in an assertion checking the validity of a pointer p. After the changes, the assertion checks the validity of the pointer p with the correct offset to point to the field n.
- Multiple import paths have either been changed or completely removed, often being replaced by the C++ standard library modules.
- Class `property_checkert` which 2LS checkers inherited from was completely removed. However, its members, such as an enumeration of potential results, were added to a new module in CProver.
- Abstract interpretation interface has been modified to more closely resemble its theoretical foundations as described in Section 2.2.2.
- Expressions now require being explicitly typed in order to access their operands.
- Incremental solver in CProver now offers an interface for adding new contexts and popping the existing ones, hence this logic could be removed from 2LS.

5.2 Employing GOTO Unwinder in 2LS

The proposed solution has been implemented as an unwinding middle layer inside 2LS so that all of the necessary transformations described in Chapter 4 can be performed in an abstract manner; without modifying the existing analysis code.

To be able to keep track of where individual unwindings start (see Section 4.4), we utilise a simple implementation trick. In order to unwind the GOTO program, we make use of CProver's lowest level unwinding interface which unwinds the loop upon passing it the pointers to the loop head and backwards `GOTO`. This approach provides us with the most control over the process. Upon finding a backwards `GOTO` instruction, we follow its target to find the loop head. Then, by creating a new instruction pointer and moving it one instruction backwards, we arrive at the last instruction prior to the loop. Hence, when we unwind the loop using the `goto_unwindt` interface, we can now safely start iterating from the previously saved instruction pointer and find the beginnings of unwindings based on the list of iterators which CProver returns as `iteration_points`.

5.2.1 Unwinding Strategy Selection

Since the extension of the unwinder exploiting incremental SAT solving has not been fully implemented, we can observe a notable performance regression if only the new unwinder is utilised. The exact numbers can be found in Section 6.2. In order to overcome this issue, we have introduced a general unwinder abstract class which both the old SSA unwinder and the new GOTO unwinder inherit from and implement its methods.

Then, we conditionally switch between the two strategies to unwinding based on the analysed C program. If dynamic memory is present (this is detected based on the presence of `malloc` calls) and any form of unwinding is requested, our new solution is used. Otherwise, the old pure SSA unwinder is utilised for better performance. Such approach is the best combination; it is sound when operating with dynamic memory and sound and performant otherwise.

The unwinder consists of 2 abstract classes. The first one, `local_unwindert`, contains logic for unwinding a single function. The second, `unwindert`, then encapsulates all the local unwinders and allows access to them based on function names. This relation is depicted in Figure 5.1 (note that the class diagram is not complete; most implementation details of the individual classes have been left out).

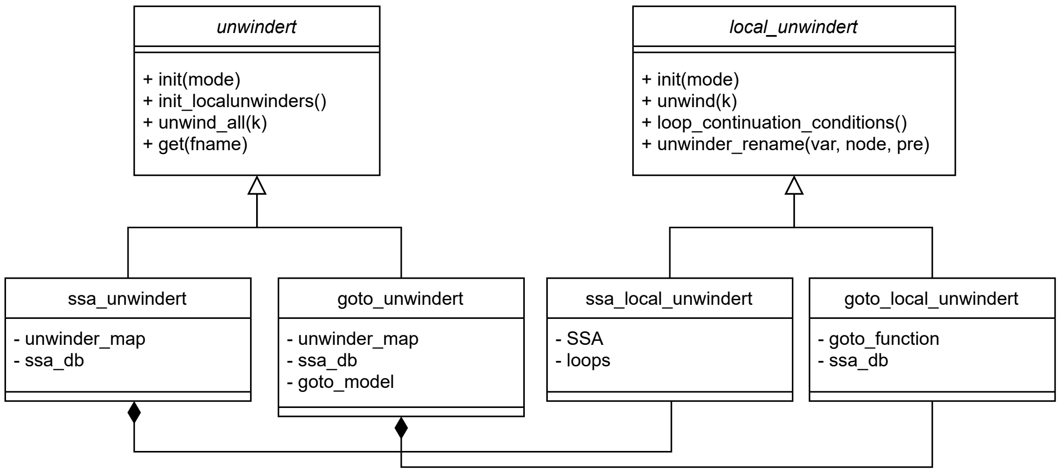


Figure 5.1: Class diagram of the classes related to unwinding

5.2.2 Non-termination Analysis

The newly implemented GOTO unwinder does not provide support for non-termination analysis. The current implementation of non-termination analysis in 2LS heavily relies on suffixes of SSA variables inside unwindings exactly in the form the old solution provided them. They are, however, not present in the new GOTO unwinder because the SSA is recomputed from scratch. The analysis would have to be partly reimplemented in order for this approach to work.

Implementing support for non-termination analysis into the new GOTO unwinder would be possible, however, it is not strictly necessary due to what types of analysis 2LS currently supports. Combination of termination or non-termination analysis with dynamic memory is not supported at all and since we are turning on the new solution only when dynamic memory is present (as described in Section 5.2.1), for the purposes of non-termination analysis, using the old solution is always sufficient. However, the possibility of combining non-termination analysis with dynamic memory operations may be an interesting area of further research once the solution proposed in Section 4.5 is implemented.

Chapter 6

Results and Experiments

To evaluate the impact and properties of our unwinding implementation, we have performed a series of experiments. Firstly, we checked the correctness of our solution using 2LS regression tests and we have also introduced new tests. This is described in Section 6.1. In order to evaluate the improvements brought by the implemented solution, we experimented with the benchmarks from the International Competition on Software Verification (SV-COMP). The results of these benchmarks are presented in Section 6.2.

6.1 2LS Regression Tests

2LS contains a suite of regression tests covering the currently supported analyses. These are often simple programs verifying the basic functionality, only some of the tests are more complex (e.g., those coming from SV-COMP). We used this test suite to verify that our unwinding solution (without strategy switching) does not break the supported analyses. This turned out to be the case, with the exception of non-termination which is not supported as described in Section 5.2.2. However, once strategy switching was turned on, all regression tests passed.

As the next step, since our solution expanded the capabilities of 2LS, we added new tests to the regression test suite. Firstly, some tests were marked as known bugs inside 2LS; these were often related to dynamic memory and unwinding. We updated and relabelled several such tests since they now work correctly with the new implementation. Secondly, we introduced new tests in the *heap data* category. This category features programs whose verification requires combined analysis of dynamic memory allocation and reasoning about the data in the allocated structures. Before this work, 2LS was only able to prove true properties in this category, but it lacked the unwinding method to check for counterexample reachability. This was made possible with our approach, hence we introduced false versions (versions containing errors) of the tests present in the category. In addition, the tests in this category were contributed to SV-COMP by the 2LS team and we plan to do the same with the new tests.

We also introduced some more arbitrary tests checking k-induction. Lastly, we added 2 tests for verifying the effect of our changes to memory leak safety; one being a benchmark from SV-COMP (where a circular doubly linked list of length 3 is non-deterministically freed and a leak occurs inside one of the 6 possible free orders) and the other being a simplified version of the program which failed in the previous implementation.

The results of running the updated regression test suite show that our changes did not have any negative effects on 2LS and that they improved verifying capabilities of 2LS.

6.2 SV-COMP Benchmarks

One of the most respected collections of benchmarks in the verification community is the collection from International Competition on Software Verification (SV-COMP). The goal of SV-COMP is to compare state-of-the-art analysers with regard to their proving capabilities and performance. The collection consists of multiple benchmark categories which are further divided into subcategories. Each subcategory then contains individual benchmarks which consist of a C program and a property to be verified. The most important subcategories for this work are *Heap Safety* and *Memory Safety* since they deal with dynamic objects whose handling was not correct in the previous implementation of the unwinder.

In SV-COMP, the overall result is expressed using a score that is calculated based on the number of correct and incorrect verification conclusions. The scoring for each benchmark is as follows:

- +1 for finding an error inside an incorrect program (*correct false*).
- +2 for proving a correct program to be error-free (*correct true*).
- -16 for reporting an error in a correct program (*incorrect false*).
- -32 for reporting correctness of an incorrect program (*incorrect true*).
- 0 for an unknown result, including analyser crashes.

We have executed 2LS multiple times on a subset of SV-COMP to analyse the effect of the CProver update, as well as of the new unwinder (both on its own and with strategy switching). Our runs included the entire *Memory Safety* category, a single subcategory for *Termination* (as unwinding is important there), *Heap Reachability*, and 3 more subcategories from the Reach Safety category which are more focused on pure performance – *Loops*, *Floats*, and *Control Flow*. The experiments were run on an Intel Xeon 5000 CPU at 3.5 GHz running Ubuntu 16.04. The runs were limited to 15 GB of memory and 5 minutes of CPU time to replicate the environment of SV-COMP (which uses a 15-minute timeout) as closely as possible without the benchmarks running excessively long.

6.2.1 Experimenting with the CProver Update

Table 6.1 gives a comparison of scores between the old 2LS (built on top of CProver 5.6) and the updated 2LS (built on top of CProver 5.37, without the unwinding modifications) to show the effect of the framework update. Note that this version has already been used in SV-COMP 2022 where it showed better results compared to the previous years. However, the collection of benchmarks may have slightly changed between the years, hence we try to make a more fair comparison.

We can observe that while some issues in 2LS have been fixed (e.g., inside *Reach Safety*), some new were introduced during the update of the CProver framework. The new incorrect results are mainly in the *Memory Safety* category which is caused by the fact that it is the most fragile analysis in 2LS which previously relied a lot on specific GOTO instruction sequences coming from CProver and in many cases these have changed. However, some of

Table 6.1: A comparison of 2LS built on top of CProver 5.6 and 2LS built on top of CProver 5.37

	Reach Safety 1574 tasks		**Memory Safety** 408 tasks		**Termination** 250 tasks	
	Before	After	Before	After	Before	After
Correct results	776	776	111	92	164	164
Correct true	592	603	55	55	116	116
Correct false	184	173	56	37	48	48
Incorrect results	5	2	0	3	0	0
Incorrect true	3	1	0	1	0	0
Incorrect false	2	1	0	2	0	0
Score	**1240**	**1331**	**166**	**83**	**280**	**280**
CPU time per task (s)	104.2	104.2	42.2	47.5	58.4	57.6

these issues can be fixed further down the line if more time is dedicated to their resolution. Overall, we can see that the CProver update had a neutral effect in terms of performance in benchmarks, however there is still the added benefit of more convenient development of 2LS.

6.2.2 Comparing the Unwinding Solutions

In our second experiment, we used the same collection of benchmarks in order to check efficiency of our implemented solution. Table 6.2 compares the performance of the old unwinding solution with the new unwinding implementation without switching unwinding strategies.

Table 6.2: A comparison of unwinding with the old SSA unwinder and with the new GOTO unwinder without strategy switching

	Reach Safety 1574 tasks		**Memory Safety** 408 tasks		**Termination** 250 tasks	
	SSA	GOTO	SSA	GOTO	SSA	GOTO
Correct results	776	625	92	143	164	116
Correct true	603	432	55	81	116	116
Correct false	173	193	37	62	48	0
Incorrect results	2	0	3	6	0	0
Incorrect true	1	0	1	2	0	0
Incorrect false	1	0	2	4	0	0
Score	**1331**	**1057**	**83**	**96**	**280**	**232**
CPU time per task (s)	104.2	108.0	47.5	47.5	57.6	20.44

We can observe multiple interesting facts from these results. Firstly, we can see that while non-termination analysis is not supported correctly in the new unwinder, termination still works. Secondly, there is a large increase in correctly verified memory safety tasks thanks to the new handling of dynamic objects and also of memory leak instrumentation. Lastly, we can see more *correct false* results in the *Reach Safety* category; based on our inspection, these are from the *Heap Reachability* subcategory. On the other hand, there is also a large drop in *correct true* tasks. As hinted by the CPU time per task, this stems from the fact that a lot more tasks resulted in a timeout due to the performance regression caused by not using incremental SAT solving. Many of the executed benchmarks, especially those in the loops subcategory, are very performance-focused and often require tens or hundreds of unwindings, where the lack of incremental SAT solving hurts the performance of 2LS.

6.2.3 Evaluating Strategy Switching

Based on the results of the first experiment with unwinding, we implemented the conditional unwinding strategy switching presented in Section 5.2.1. Comparison of the old SSA unwinder and the final solution with strategy selection can be seen in Table 6.3.

Table 6.3: A comparison of unwinding with the old SSA unwinder and combination of SSA and GOTO unwinder which uses strategy selection

	Reach Safety		**Memory Safety**		**Termination**	
	1574 tasks		408 tasks		250 tasks	
	SSA	Switching	SSA	Switching	SSA	Switching
Correct results	776	814	92	143	164	164
Correct true	603	622	55	81	116	116
Correct false	173	192	37	62	48	48
Incorrect results	2	1	3	6	0	0
Incorrect true	1	0	1	2	0	0
Incorrect false	1	1	2	4	0	0
Score	**1331**	**1420**	**83**	**96**	**280**	**280**
CPU time per task (s)	104.2	107.4	47.5	47.5	57.6	57.6

We can see improvements in score across all the categories; the performance regressions inside *Reach Safety* category have been worked around by the strategy switching leading to an increase in both *correct false* and *correct true* results thanks to supporting dynamic memory.

On the other hand, there are a few new incorrect results inside the *Memory Safety* category; 3 of the 6 incorrect results are identical to the old version and are related to some different bugs in 2LS. We have tried to investigate the failures and they seem to be related to the way how 2LS tracks object deallocation and dereferencing of freed objects. This may be a similar issue to handling memory leak checks (which we solved), however trying a similar solution introduced even more problems. Hence, we assume that the solution will not be trivial. These few benchmarks need to be further investigated, the score potential of the new solution is very high (as seen by the number of correct benchmarks in the *Memory Safety* category) if they are fixed.

Note that while the new incorrect results seem like a regression in soundness of 2LS, it is in fact an improvement. When running SV-COMP benchmarks, a so-called *competition mode* is used. Inside *competition mode*, no unwinding was ever done when dynamic memory was detected and hence errors related to unwinding of dynamic objects were hidden away, otherwise there would be significantly more incorrect results in the old version.

Chapter 7

Conclusion

In this work, we proposed a novel mechanism for loop unwinding for the 2LS framework. The solution unwinds loops in a GOTO program (a CFG-based representation coming from the CProver framework) rather than in the SSA form as the original solution in 2LS did. This enables updating the set of dynamic objects in the program and correctly representing operations with dynamic memory in the SSA form. The proposed approach allows 2LS to soundly unwind programs which operate with dynamic memory inside loops and hence find counterexamples if an error is introduced in one of the iterations.

The proposed solution has been implemented in the 2LS framework. As a part of our implementation, we have updated the underlying CProver framework used in 2LS to its latest version, which simplifies future development of 2LS. We performed several experiments with the implementation using benchmarks from International Competition on Software Verification (SV-COMP) and the results show that 2LS with our extension can now correctly analyse more programs dealing with dynamic memory. In other words, our extension has improved verification capabilities of 2LS.

The performed experiments also demonstrate that the newly implemented solution does not scale as well as the old mechanism due to not exploiting incremental SAT solving. Based on these findings, we implemented an unwinding strategy selection which utilises both of the approaches based on the program being analysed. Overall, this improves the score of 2LS in SV-COMP. We also present a proposal for an alternative approach which would have the advantages of the newly implemented approach as well as utilise the incremental SAT solving.

In future, we would like to implement the mentioned approach and completely remove the original SSA unwinder. Correct unwinding of dynamic objects opens up multiple directions for further development of 2LS, for example combining termination analysis with dynamic objects (which is not currently supported) or improving the internal representation of dynamic objects in a way that would allow reasoning about more properties. Lastly, having the GOTO program synchronised with the SSA form even after unwinding possibly facilitates implementation of support for recursive function calls.

Bibliography

[1] Biere, A., Cimatti, A., Clarke, E. and Zhu, Y. Symbolic Model Checking without BDDs. In: Cleaveland, W. R., ed. *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, p. 193–207. ISBN 978-3-540-49059-3.

[2] Brain, M., Joshi, S., Kroening, D. and Schrammel, P. Safety Verification and Refutation by k-Invariants and k-Induction. In: Blazy, S. and Jensen, T., ed. *Static Analysis: 22nd International Symposium, SAS 2015, Saint-Malo, France, September 9-11, 2015, Proceedings*. 1st ed. Springer, 2015, p. 145–161. DOI: https://doi.org/10.1007/978-3-662-48288-9. ISBN 978-3-662-48288-9.

[3] Brain, M., Schrammel, P. and Kloos, J. *Representations* [online]. 2021 [cit. 2021-12-29]. Available at: http://cprover.diffblue.com/background-concepts.html#representations_section.

[4] Clarke, E., Kroening, D. and Lerda, F. A Tool for Checking ANSI-C Programs. In: Jensen, K. and Podelski, A., ed. *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*. Springer, 2004, vol. 2988, p. 168–176. Lecture Notes in Computer Science. ISBN 3-540-21299-X.

[5] Cousot, P. and Cousot, R. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: ACM. *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. Association for Computing Machinery, January 1977, p. 238–252. DOI: 10.1145/512950.512973. ISBN 9781450373500.

[6] Gadelha, M. R., Monteiro, F., Cordeiro, L. and Nicole, D. ESBMC v6.0: Verifying C Programs Using k-Induction and Invariant Inference. In: Beyer, D., Huisman, M., Kordon, F. and Steffen, B., ed. *Tools and Algorithms for the Construction and Analysis of Systems*. Cham: Springer International Publishing, 2019, p. 209–213. ISBN 978-3-030-17502-3.

[7] Heizmann, M., Chen, Y.-F., Dietsch, D., Greitschus, M., Hoenicke, J. et al. Ultimate Automizer and the Search for Perfect Interpolants. In: Beyer, D. and Huisman, M., ed. *Tools and Algorithms for the Construction and Analysis of Systems*. Cham: Springer International Publishing, 2018, p. 447–451. ISBN 978-3-319-89963-3.

[8] Khazem, K. and Brain, M. *Goto-programs* [online]. 2021 [cit. 2021-12-28]. Available at: http://cprover.diffblue.com/group__goto-programs.html.

[9] KROENING, D. and TAUTSCHNIG, M. CBMC – C Bounded Model Checker. In: ÁBRAHÁM, E. and HAVELUND, K., ed. *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, p. 389–391. ISBN 978-3-642-54862-8.

[10] LEÓN, H. Ponce-de, FURBACH, F., HELJANKO, K. and MEYER, R. Dartagnan: Bounded Model Checking for Weak Memory Models (Competition Contribution). In: BIERE, A. and PARKER, D., ed. *Tools and Algorithms for the Construction and Analysis of Systems*. Cham: Springer International Publishing, 2020, p. 378–382. ISBN 978-3-030-45237-7.

[11] MALÍK, V. *Template-Based Synthesis of Heap Abstractions*. Brno, CZ, 2017. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Available at: https://www.fit.vut.cz/study/thesis/19901/.

[12] MALÍK, V., MARTIČEK, Š., SCHRAMMEL, P., SRIVAS, M., VOJNAR, T. et al. 2LS: Memory Safety and Non-termination. In: BEYER, D. and HUISMAN, M., ed. *Tools and Algorithms for the Construction and Analysis of Systems*. Cham: Springer International Publishing, 2018, p. 417–421. ISBN 978-3-319-89963-3.

[13] MALÍK, V., HRUSKA, M., SCHRAMMEL, P. and VOJNAR, T. Template-Based Verification of Heap-Manipulating Programs. In: IEEE. *2018 Formal Methods in Computer Aided Design (FMCAD)*. 2018, p. 1–9. DOI: 10.23919/FMCAD.2018.8603009. ISBN 978-0-9835678-8-2.

[14] MARTIČEK Štefan. *Synthesizing Non-Termination Proofs from Templates*. Brno, CZ, 2017. Master's thesis. Brno University of Technology, Faculty of Information Technology. Available at: https://www.fit.vut.cz/study/thesis/13436/.

[15] MORSE, J., CORDEIRO, L., NICOLE, D. and FISCHER, B. Handling Unbounded Loops with ESBMC 1.20. In: PITERMAN, N. and SMOLKA, S. A., ed. *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, p. 619–622. ISBN 978-3-642-36742-7.

[16] MUCHNICK, S. S. *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998. ISBN 1558603204.

[17] PRESTON WERNER, T. *Semantic Versioning 2.0.0* [online]. June 2013. Available at: https://semver.org/.

[18] SCHRAMMEL, P., KROENING, D., BRAIN, M., MARTINS, R., TEIGE, T. et al. Incremental bounded model checking for embedded software. *Formal Aspects of Computing*. Sep 2017, vol. 29, no. 5, p. 911–931. DOI: 10.1007/s00165-017-0419-1. ISSN 1433-299X. Available at: https://doi.org/10.1007/s00165-017-0419-1.

Appendix A

Contents of the Included Storage Media

The included storage media contains source files of 2LS with our extension, as well as the source files of the thesis. The structure of the root directory is the following:

```
/
├── 2ls/ ........................................ 2LS directory
│   ├── lib/
│   │   └── cbmc/ ................. Source code of CBMC (CProver framework)
│   ├── src/ ...................................... Source files of 2LS
│   └── regression/ ........................... Regression tests of 2LS
├── doc/ .............................. LaTeX source files of this text
└── README ........................................... README file
```

The `2ls/src` directory contains the implementation of 2LS including our extension. The directory is divided into multiple subdirectories, most of our work has been done inside the `2ls/src/ssa` directory in modules related to unwinding and SSA transformations.

Source code of the CProver framework can be found inside `2ls/lib/cbmc` directory. This source code is based on CProver's 5.37 release with 2LS-specific extensions as discussed in Section 5.1. The directory `doc` contains the source LaTeX files of this text as well as the PDF version.

Appendix B

Compilation and Running

The project can be compiled and run using the source files present on the included storage media. The most convenient way for compiling the project is by using the `build.sh` script under the `2ls` directory which compiles CBMC (including the Glucose SAT solver) and then compiles 2LS. These steps can also be performed manually, e.g., by running `make glucose-download && make` inside the `2ls/lib/cbmc/src` directory and then `make` inside the `2ls/src` directory. The compiled executable is present inside the `2ls/src/2ls` directory.

2LS can be run using the command `2ls SOURCE_FILE OPTIONS` where `SOURCE_FILE` is a valid compilable C program and `OPTIONS` are flags for 2LS. Some of the options most relevant for this work, are `--unwind N`, `--k-induction`, `--incremental-bmc`, and their combination with `--heap`, `--pointer-check`, and `--memory-leak-check`.

We recommend using 2LS regression tests as examples of potential ways to use our extension. Each regression test contains a `test.desc` file which describes the options which are to be used for the test. For example, analysis of our motivation example presented in Section 3.2.2 could be performed using the following command:

```
$ 2ls regression/heap/simple_false_kind2/main.c --heap --intervals
                    --no-propagation --k-induction.
```

Appendix C

2LS Regression Tests

2LS contains a suite of regression tests; they are included on the storage media inside `2ls/regression` directory. All the tests can be run using the present `Makefile`. The tests are divided into categories based on the type of analysis they check. Each category has its own `Makefile` and can be run on its own. There are currently 10 categories:

heap Contains tasks using heap manipulation. We relabelled `list_iter`, `list_unwind` and `array_unwind` tasks since 2LS can now analyse them correctly, as well as added our motivation example under `simple_false_kind2`.

heap-data Contains tasks that require reasoning about data inside data structures dynamically allocated on the heap. As a part of this work, we added `false` counterparts to some of the existing tasks.

instrumentation Contains tests for GOTO program instrumentation with computed invariants.

interprocedural Tasks requiring interprocedural analysis.

invariants Tests the computing of invariants in various domains.

kiki Tests focusing on aspects of the *k*I*k*I algorithm, mainly *k*-induction.

memsafety Contains task focused on all kinds of memory safety (free safety, null dereference safety, leak safety). We extended this category with `nondet_free_kind`, `nondet_free_leak_kind`, `null_deref_kind` and `simple_leak_kind` tasks.

nontermination Tasks focused on non-termination analysis.

preconditions Contains tasks aimed at computing backward and forward preconditions and postconditions of functions in the analysed program.

termination Tasks focused on termination analysis.