BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

GENERIC TEMPLATE-BASED SYNTHESIS OF PROGRAM ABSTRACTIONS

GENERICKÁ SYNTÉZA INVARIANTŮ V PROGRAMU ZALOŽENÁ NA ŠABLONÁCH

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **MATEJ MARUŠÁK**
AUTOR PRÁCE

SUPERVISOR **Ing. VIKTOR MALÍK**
VEDOUCÍ PRÁCE

BRNO 2019

Brno University of Technology
Faculty of Information Technology

Department of Intelligent Systems (DITS) Academic year 2018/2019

Master's Thesis Specification

Student: **Marušák Matej, Bc.**
Programme: Information Technology Field of study: Intelligent Systems
Title: **Generic Template-Based Synthesis of Program Abstractions**
Category: Formal Verification
Assignment:

1. Study principles of analysis of programs based on template-based synthesis of program abstractions. Study the source code of the 2LS framework that uses this method.
2. Study the existing templates representing various abstractions of program semantics (i.e. abstract domains) and their combinations that 2LS uses. Focus on finding similarities in synthesis of abstractions across the abstract domains.
3. Design and implement a generic solution to synthesis of abstractions from templates that will be usable for all existing abstract domains and that will simplify adding of new abstract domains in future.
4. With the help of the proposed solution, explore new possible domain combinations.
5. Evaluate your solutions on regression tests from the 2LS framework and on benchmarks from the Software Verification Competition SV'COMP.
6. Discuss advantages and limitations of the approach and possible extensions.
7. Write the final text of the Master's thesis in English.

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.: Template-based Synthesis of Heap Abstractions. Master's thesis, Brno University of Technology, Brno (2017)
- Chen, H.-Y., David, C., Kroening, S., Schrammel, P., Wachter, B.: Bit-precise procedure-modular termination analysis. In: Proc. of ACM TOPLAS'18, 2018.

Requirements for the semestral defence:

- Items 1 and 2 of the assignment.

Detailed formal requirements can be found at http://www.fit.vutbr.cz/info/szz/

Supervisor: **Malík Viktor, Ing.**
Head of Department: Hanáček Petr, doc. Dr. Ing.
Beginning of work: November 1, 2018
Submission deadline: May 22, 2019
Approval date: November 1, 2018

Abstract

The goal of this work is to design and to implement a generic strategy solver for the 2LS tool. 2LS is an analyser for static verification of programs written in C language. A verified program is analysed by an SMT solver using abstract interpretation. Conversion from an abstract state of the program into a logical formula, that an SMT solver can work with, is done by a component called strategy solver. In the current implementation, there is one strategy solver for each abstract domain. Our approach introduces a single generic strategy solver, which makes creating new domains easier. Also, this approach enables migration of the existing domains and hence the codebase can be reduced.

Abstrakt

Cieľom tejto práce je návrh a implementácia generického strategy solveru pre nástroj 2LS. 2LS je analyzátor na statickú verifikáciu programov napísaných v jazyku C. Verifikovaný program je za využita abstraktnej interpretácie analyzovaný SMT solverom. Prevod z abstraktného stavu programu do logickej formule, s ktorou vie pracovať SMT solver vykonáva komponenta nazývaná strategy solver. Aktuálne pre každú doménu existuje jeden takýto solver. Navrhované riešenie vytvára jeden obecný strategy solver, ktorý zjednodušuje tvorbu nových domén. Zároveň navrhovaný spôsob umožnuje prevedenie existujúcich domén a teda zmenšuje program analyzátora.

Keywords

formal verification, 2LS, static analysis, SSA form, abstract domain, strategy solver, SMT solving, abstract interpretation

Klíčová slova

formálna verifikácia, 2LS, statická analýza, SSA forma, abstraktná doména, strategy solver, SMT solving, abstraktná interpretácia

Reference

MARUŠÁK, Matej. *Generic Template-Based Synthesis of Program Abstractions.* Brno, 2019. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Viktor Malík

Rozšířený abstrakt

Nástroj 2LS, pôvodne vyvíjaný na univerzite v Oxforde a dnes zastrešovaný britskou firmou DiffBlue, je statický analyzátor programov napísaných v programovacom jazyku C. Zatiaľčo existuje mnoho programov na statickú analýzu, tento nástroj sľubuje pokryť dva hlavné nedostatky v tejto oblasti – spoľahlivo pracovať na skutočných programoch a zároveň na týchto programoch vedieť analyzovať rôzne a komplexné vlastnosti. 2LS dokáže analyzovať bezpečnostné a ukončujúce podmienky vzťahujúce sa ku rovnosti premenných, toku dát cez numerické premenné či k práci s dynamickými dátovými štruktúrami.

Základom tohto nástroja je novovytvorený algoritmus nazývaný kIkI. Tento algoritmus efektívne kombinuje obmedzené overovanie modelov (bounded model checking), k-indukciu (k-induction) a abstraktnú interpretáciu (abstract interpretation). Tento algoritmus využíva abstraktnú interpretáciu k odvodzovaniu induktívnych invariantov (inductive invariants), ktorých výpočet je jednoduchší ako výpočet všetkých dosiahnuteľných stavov. Nakoľko výpočet induktívnych invariantov je náročný aj pre dnešné SMT solvery, je tento problém redukovaný použitím tzv. šablón. Iteratívnym prístupom sú hľadané parametre šablón dovtedy, kým nie je nájdený invariant. Tento iteratívny prístup riešenia je v nástroji 2LS implementovaný v komponente nazývanej strategy solver.

Pre každý typ analýzy, ktorú 2LS ponúka, je nutné definovať šablónu spolu s abstraktnou hodnotou, ktorá zachytáva určitú analyzovanú vlastnosť programu. Tieto dve informácie sú uložené v tzv. abstraktných doménach. Zároveň pre každú doménu musí byť definovaný spôsob spájania aktuálnych hodnôt parametrov šablóny s novonájdenými hodnotami z SMT solveru. Tento algoritmus sa označuje ako join operátor. Nástroj 2LS definuje pre každú doménu konkrétny strategy solver, ktorý obsahuje iteratívny algoritmus na odvodzovanie invariantov spolu s join operátorom.

Vzhľadom na to, že spomínaný iteratívny algoritmus je veľmi podobný pre všetky domény, táto práca navrhuje generický strategy solver, ktorý by bol univerzálne použiteľný pre rôzne domény. Hlavnou výhodou generického riešenia je uľahčenie pridávania nových domén v budúcnosti, nakoľko pri tvorbe novej domény nie je potrebné implementovať iteratívny algoritmus na usudzovanie invariantov.

Okrem tzv. jednoduchých domén 2LS ponúka aj špeciálne domény, ktoré sú založené na spojení iných, už existujúcich, domén. Táto pridaná komplexita dokáže využívať vlastnosti všetkých kombinovaných domén a vďaka tomu usudzovať silnejšie invarianty, ktoré by nebolo možné nájsť pomocou jednoduchých domén. Momentálne existujú dva rôzne prístupy ku kombinácií domén.

Táto práca taktiež navrhuje generické riešenie pre oba typy týchto kombinácií. Zatiaľčo hlavnou výhodou generického strategy solveru pre jednoduché domény bolo uľahčenie pridávania nových domén, pri generickom riešení pre kombinačné domény ide o ich absolútne nahradenie – generické riešenie dynamicky pri štarte programu môže vybrať ľubovoľné domény a spustiť analýzu s ich kombináciou bez nutnosti definovania tejto kombinácie kdekoľvek v kóde 2LS.

Aby mohli domény využívať generický strategy solver, musia poskytovať určité rozhranie, za pomoci ktorého strategy solver s doménami komunikuje. Táto práca navrhuje rozhranie, v ktorom je pri implementácii domény možné danú doménu použiť ako s generickým strategy solverom pre jednoduché alebo pre kombinačné domény.

Všetky tri navrhnuté generické strategy solvery boli implementované spolu s navrhnutým rozhraním domén. Existujúce jednoduché a kombinačné domény boli zmigrované a využívajú generické riešenie strategy solverov. Táto migrácia viditeľne zredukovala počet súborov existujúcich v implementácií domén. Jediné domény, ktoré neboli presunuté sú tie, ktoré

pri odvodzovaní invariantov využívajú binárne vyhľadávanie. Takýto prístup vyžaduje ďalší generický strategy solver.

Nakoľko boli pri tejto práci zmenené kritické časti analýzy programov, bolo nutné overiť, či tieto zmeny negatívne neovplyvnili momentálne schopnosti tohto nástroja. Táto verifikácia bola uskutočnená za pomoci regresných testov, ktoré existujú v nástroji 2LS a testujú veľké množstvo rôznych vstupných programov. Zároveň bola na detailnejšie overenie použitá sada testov zo súťaže SV-COMP 2018 (International Competition on Software Verification 2018).

V neposlednom rade boli vykonané experimenty s novými kombináciami domén – na konkrétnom príklade bolo ukázané, že je veľmi jednoducho možné použiť novú, doteraz neimplementovanú kombináciu domén a analyzovať program, ktorý predtým nebolo možné správne overiť.

Generic Template-Based Synthesis of Program Abstractions

Declaration

Hereby I declare that this master's thesis was prepared as an original author's work under the supervision of Ing. Viktor Malík. The supplementary information was provided by prof. Ing. Tomáš Vojnar, Ph.D. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Matej Marušák
May 20, 2019

Acknowledgements

I would like to express my sincere gratitude to my advisor, Ing. Viktor Malík, for the continuous support during working on this thesis and valuable advice and patience. His guidance was highly appreciated and made this work possible.

Contents

Chapter 1

Introduction

As computers play bigger and bigger role in our lives, the complexity of code running on them grows as well. More complex programs mean wider possibilities of where bugs can occur. To make sure that a program runs as expected, it is necessary to be verified. This can be done in two ways – using static or dynamic analysis. While dynamic analysis counts on an execution of the program (i.e. using tests written by developers), static analysis checks the code itself and hence it covers all program traces.

2LS tools [15], originally created at the University of Oxford and today being developed by the company DiffBlue, is a static analyzer for programs written in the C language. Although there are many tools available for a static analysis, this tool promises to cover two most common weaknesses in this area – to be able reliably work with real world programs and on these programs to be able to analyse different and complex properties. 2LS is capable of verifying safety and termination properties that are related to variables equality, to data-flow among numerical variables, or to work with dynamic data structures.

The core of this tools is a novel algorithm called k-invariants k-induction (kIkI)[5]. This algorithm efficiently combines bounded model checking, k-induction, and abstract interpretation. In kIkI abstract interpretation is used to infer inductive invariants, which computation is easier that computation of the set of all reachable states of the analysed program. Since computation of inductive invariants is challenging even for today available SMT solvers, this problem is reduced using so called templates. Iteratively parameters of these templates are refined until an invariant is found. This iterative approach is implemented in 2LS tool in a component called strategy solver.

For each type on analysis which exists in 2LS, it is necessary to define template together with abstract value, which describes a specific analysed property. These two information are encapsulated in so called abstract domains. Moreover, for each abstract domain needs to be defined an algorithm for joining the current values of the template parameters with a newly found values by an SMT solver. This algorithm is called join operator. 2LS tool defines for each domain a specific strategy solver, which contains the iterative algorithm for invariant inference together with the join operator.

Since the iterative algorithm is very similar for all abstract domains, this work proposes a generic strategy solver, which could be used for any domains. The main advantage of this generic solution is a simplification of the process of adding a new domain, as when a new domain is going to be created, only the abstract domain with the abstract value is needed to be designed and implemented since the iterative algorithm for invariant inference is going to be provided by the generic strategy solver.

Moreover to so called simple domains, 2LS implements also specialized domains, which are based on others, already existing domains. This added complexity is capable of using properties of all combined domains and therefore infer stronger invariants, which could not be found by single domains. Currently there are two different approaches to domain combinations.

This work also proposes a generic solution for both types of domain combinations. While the main advantage of the generic strategy solver for simple domains was to ease the process of adding a new domains, for the generic solution for domain combinations it is a complete replacement – 2LS with this generic strategy solver can upon starting select any abstract domains and start analysis of their combination without this specific combination being defined anywhere in this tool.

For domains being able to use generic strategy solver they have to implement a specific interface, through which strategy solver communicates with domains. This work proposes such interface, which when implemented by a domain, this domain can be used both with strategy solver for simple and combination domains.

All three proposed generic strategy solvers has been implemented together with the proposed domain interface. Existing simple and combination domains were migrated and currently are using these generic strategy solvers. This migration visibly reduced number of files in the domains implementation. Only those domains were not migrated, which use binary search for invariant inference. This approach requires another generic strategy solver.

Since this work changed vital parts of the code responsible for program analysis, it was necessary to validate that these changes did not negatively effected current capabilities of this tool. This verification was done by regression tests which exist in 2LS tools and contain a large amount of different input programs. Moreover, for a better validation a set of tests from SV-COMP 2018 (International Competition on Software Verification 2018) has been used.

Lastly, experiments were conducted with new domain combinations – it was shown on a specific example that it is very simple to use a new, not yet implemented, domain combination and to analyse program, which could not be correctly analysed before.

The rest of the thesis is structured as follows. Chapter 2 provides an overview of the used verification approaches and of the kIkI algorithm. Chapter 3 describes how 2LS encodes the source program and how a source program from language C is translated into a logical formula. The core part of this work is based around *template-based verification* which is introduced in Chapter 4 together with the algorithm for invariant inference which is the part implemented by strategy solvers. Our proposal of generic strategy solvers is described in Chapter 5 followed by Chapter 6 focused on combination strategy solvers. In the Chapter 7 is described the implementation part of this work. Experiments done in this work together with results are presented in Chapter 8 and finally a conclusion and future work is summarised in Chapter 9.

Chapter 2

Verification in the 2LS Tool

2LS [15] is a program verification framework built upon the CPROVER [1] verification framework that combines three verification approaches in order to efficiently analyze programs written in C language.

In this chapter these approaches are introduced and described. In order to be able to describe these methods a program representation called *transition system* is introduced in Section 2.1. Using this representation the three verification approaches are introduced – namely *bounded model checking* in subsection 2.2.1, *k-induction* in 2.2.2 and *abstract interpretation* in part 2.2.3. The last named requires knowledge of *concrete semantics* which is also formally introduced in subsection 2.1.

All of these approaches and techniques are combined into a novel *k-invariant k-induction* (so-called *k*I*k*I) algorithm [5], which is the core verification algorithm of 2LS tool. This algorithm is described in Section 2.3.

2.1 Program as Logical Formulae

To simplify the following descriptions of verification approaches, the source program is viewed as a transition system. A *program state* x is represented by current values of all program variables including current value of the program counter and the state of all related memory (such as stack and heap).

Using transition system, all reachable states of a program execution create a *concrete semantics*. Let S be the set of all program states and let the *transition relation* $\mathcal{T} \subseteq S \times S$ define for each state a set of all its possible successors in the program execution [13].

Assuming $S_k = \mathcal{T}^k(S_0)$ as the set of all reachable states starting from S_0 after execution of k steps, S_r can be also defined as the least fixed point of $\mathcal{T}$ as

$$S_r = \bigcup_{i \in \mathbb{N}} \mathcal{T}^i(I) \tag{2.1}$$

where I is set of all possible initial states of a program. With this definition S_r defines the concrete semantics of the analysed program.

The set of all reachable states can be used for analysis of program properties, however, computing this set is often difficult and not possible for real world program representation as the set grows exponentially. Instead inductive invariants are used (described in Section 2.2.2).

Also, often to prove program correctness it is enough to show that the set of all reachable states does not intersect with the set of error states, denoted by the predicate *Err(x)*. Alternatively, to prove that the program is not safe, it is enough to find n-step counter-example, that can be described by formula:

$$\exists x_0, \ldots, x_k.Start(x_0) \wedge \bigwedge_{i \in [0,n-1]} Trans(x_i, x_{i+1}) \wedge Err(x_n) \tag{2.2}$$

where predicate $Start(x)$ encodes that $x \in I$ and predicate $Trans(x, x')$ that $x' \in \mathcal{T}(x)$.

2.2 Verification Approaches

This section describes some of more well known approaches for program verification as well as their strengths and weaknesses. All of these approaches are used in some kind of form in the 2LS verifying algorithm, since they complement and reinforce each other.

2.2.1 Bounded Model Checking

Bounded model checking builds upon and extends the idea of a basic *model checking*, where the program to be verified is presented as a finite state machine. All reachable states are represented by some structure which is checked whether it satisfies all desired properties. It can be easily used to provide counterexamples by finding sets of states not satisfying the verified properties. The problem of this approach lies in its complexity, which grows exponentially with the size of the program being verified. An improved method, called *symbolic model checking*, represents sets of states as boolean expressions. Manipulating such expressions can be efficiently done by using *Binary Decision Diagrams* (BDD), however it still can be memory expensive. This problem is possible to solve with *bounded model checking* (BMC), which checks only for executions of length less than i which means solving formula:

$$\exists x_0, \ldots, x_k.Start(x_0) \wedge \bigwedge_{i \in [0,k-1]} Trans(x_i, x_{i+1}) \wedge \bigvee_{i \in [0,k]} Err(x_i) \tag{2.3}$$

This approach can provide counter-examples for only limited length so it represents under-approximation of the set of reachable states and therefore can fail to find counter-examples that require a longer transition sequences.

The value k can be iteratively increased until an upper bound is reached (no upper bound is needed to be specified, but then the algorithm may never terminate). This way, program correctness cannot be proven, only counterexamples can be found [3]. It is also possible to increase the bound value itself – in this case the approach is called *Incremental BMC* (IBMC) [10]. The bound value usually starts at value 0, which means solving formula:

$$\exists x_0.Start(x_0) \wedge Err(x_0) \tag{2.4}$$

Then the bound value is increased (usually linearly) and BMC is executed. This process can be iteratively repeated until upper bound value is reached.

2.2.2 K-induction

Contrary to BMC, that is in practice usable only for finding counter-examples, *k-induction* is able to show program safety as well as to find counterexamples. It is done by utilising *k-inductive invariants* which are general versions of a *inductive invariant*.

Each inductive invariant describes a fixed-point of a transition relation but it is not guaranteed to be the least one nor is it guaranteed to include *Start(x)*. It is defines as:

Definition 2.2.1 *An invariant Inv is called inductive if it meets the property:*

$$\forall x, x'.(Inv(x) \wedge Trans(x, x') \Rightarrow Inv(x')) \tag{2.5}$$

Then a k-inductive invariant is defined as:

Definition 2.2.2 *Inductive invariant KInv is an k-inductive invariant if it meets the following property:*

$$\forall x_0, \ldots, x_k. \bigwedge_{i \in [0,k-1]} KInv(x_i) \wedge \bigwedge_{i \in [0,k-1]} Trans(x_i, x_{i+1}) \implies KInv(x_k) \tag{2.6}$$

K-inductive invariants have following properties [4] [13]:

Lemma 2.2.1 *Every inductive invariant is a 1-inductive invariant and vice versa.*

Lemma 2.2.2 *Every k-inductive invariant is a (k+1)-inductive invariant.*

Lemma 2.2.3 *Showing that k-inductive invariant exists implies that an inductive invariant exists.*

Lemma 2.2.4 *K-inductive invariant is not necessarily an inductive invariant, usually a corresponding inductive invariant is much more complex.*

A program can be considered to be safe if and only if exists a k-inductive invariant *KInv* that satisfies:

$$\begin{aligned} \forall x_0, \ldots, x_k. \Big(Start(x_0) \wedge \bigwedge_{i \in [0,k-1]} Trans(x_i, x_{i+1}) \implies \bigwedge_{i \in [0,k-1]} KInv(x_i)\Big) \wedge \\ \Big(\bigwedge_{i \in [0,k-1]} KInv(x_i) \wedge \bigwedge_{i \in [0,k-1]} Trans(x_i, x_{i+1}) \implies KInv(x_k)\Big) \wedge \\ \Big(KInv(x_k) \implies \neg Err(x_k)\Big) \end{aligned} \tag{2.7}$$

Finding a k-inductive invariant is enough to prove that an inductive invariant exists (according to Lemma 2.2.3(but also due to Lemma 2.2.4 it does not imply that the k-inductive invariant is an inductive invariant). However finding k-inductive invariants is hard to implement and very similarly to IBMC, increasing k can be used to simplify the process. It means solving following formula:

$$\begin{aligned} \exists x_0, \ldots, x_k. \Big(Start(x_0) \wedge \bigwedge_{i \in [0,k-1]} Trans(x_i, x_{i+1}) \wedge \bigwedge_{i \in [0,k-1]} \neg Err(x_i) \wedge Err(x_k)\Big) \vee \\ \Big(\bigwedge_{i \in [0,k-1]} Trans(x_i, x_{i+1}) \wedge \bigwedge_{i \in [0,k-1]} \neg Err(x_i) \wedge Err(x_k)\Big) \end{aligned} \tag{2.8}$$

and if this formula is not satisfiable the program is safe (and $\neg Err(x)$ is a k-inductive invariant) [9].

2.2.3 Abstract Interpretation

In contrast to previous approaches, which are usually used for under-approximations of the set of reachable states of the source program, *abstract interpretation* is based on an over-approximation.

As already mentioned the set of reachable states is generally not computable. Moreover, usually only a certain property of the analyzed program is needed to reason about and therefore it is sufficient enough to approximate program states as elements of a simpler domain – so called *abstract domain* – which approximates the concrete domain. An element of an abstract domain – called an *abstract value* – typically contains a set of concrete program states. Having the concrete domain P of all program states and the abstract domain Q, the following two functions are defined [7]:

- $\gamma : Q \to P$ is a *concretisation function* that defines a mapping from an abstract value to a concrete value.
- $\alpha : P \to Q$ is a *abstraction function* that defines a mapping from a concrete value to an abstract value. $\alpha(p)$ is the most precise abstract value from Q which contains p.

Definition 2.2.3 *Abstract interpretation I is defined as [8]:*

$$I = (Q, \sqcup, \sqsubseteq, \top, \bot, \mathcal{T}^{\#}) \tag{2.9}$$

where

- *Q is the abstract domain with defined concretisation and abstraction functions*
- *$\sqcup : Q \times Q \to Q$ is the* join operator*, $(Q, \sqcup, \top)$ is a complete semilattice*
- *$(\sqsubseteq) \subseteq Q \times Q \to Q$ is an ordering on $(Q, \sqcup, \top)$. $x \sqsubseteq y \overset{def}{\iff} x \sqcup y = y$*
- *$\top \in Q$ is the supremum of Q*
- *$\bot \in Q$ is the infimum of Q*
- *$\mathcal{T}^{\#} : Instr \times Q \to Q$ defines the interpretation of abstract transformers*

As abstract values are over-approximations of the set of reachable concrete values, it is possible that false positives can be generated. It is usually due to abstract value representing set of concrete values, from which some are not reachable in the original program. One of the ways how to minimize this effect is to a use more precise abstract domain.

2.3 *k*I*k*I Algorithm

In this algorithm introdued by 2LS, techniques from Section 2.2 strengthen and reinforce each other. Overview of this algorithm is depicted in Figure 2.1.

The letter k in the algorithm name represents a bound value that is increased in the program run if it is needed. Very similarly to other mentioned approaches (such as IBMC) also in this algorithm some maximum value for k must be set up, otherwise the analysis may not terminate. In case this maximum value is reached and the property being analyzed was not proven (or refuted) the analyses ends up inconclusive.

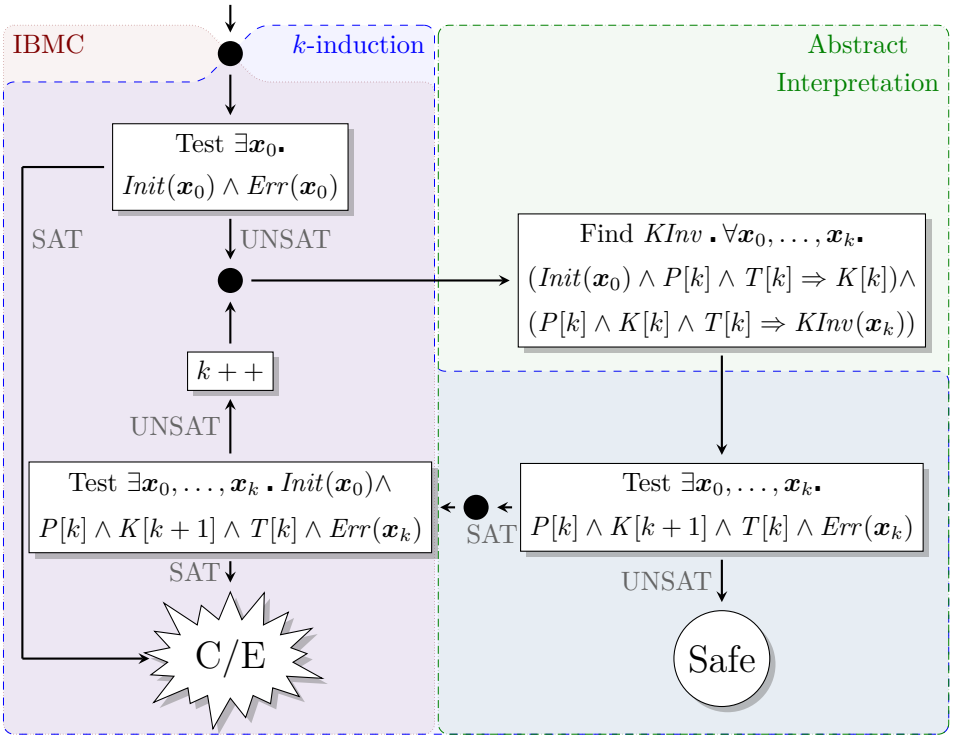


Figure 2.1: The kIkI algorithm [5]

The kIkI algorithm starts with setting up $k = 1$. Then k-induction (described in subsection 2.2.2) uses invariant that has been generated by abstract interpretation in some abstract domain (described in subsection 2.2.3) for proving true properties. If the invaraint is not sufficent to prove safety (a violation was found by a SMT solver) it is tested by bounded model checking (described in subsection 2.2.1) to see if this violation is real counter-examples reachable in k-steps. If counter-example is shown to be valid, analyses ends with providing this counter-example and showing that the input program contains a problem. If BMC could not prove that this counter-example is valid, value of k is increased and the loop is repeated.

This algorithm enables 2LS to efficiently analyze programs and to reason about various properties by finding invariants and proving its correctness or by finding counter-examples and therefore finding errors in them.

Chapter 3

Representation of Programs as Logical Formulae

All approaches mentioned in Section 2.2 see the source program as transition system (described in Section 2.1). Even though the transition system is directly usable for analyses, there are some properties of programs which can be used for improving this system. For example, most of the states of the program counter directly identify its next value (i.e. most of the instructions do not branch) and most of the transactions update a single variable. Therefore, states in a transition can be joined into a symbolic value resulting in reduction of size. Rather than creating a transition system and consequently reducing it, it is more efficient to convert the program into the *static single assignment* (SSA) form. Another advantage of using the SSA form is that it can be easily converted into a logical formula, that can be utilised with solver-based approaches. In 2LS, an extended form of the SSA form is used, which includes over-approximation of the loops so that a solver can be used to reason about abstractions of the program. [5]

In this chapter, the general SSA encoding is introduced in Section 3.1. Next, extensions that 2LS introduces into the SSA form are described – namely introduction of guards to encode information about control-flow (Section 3.2) and over-approximation of loops by cutting them (Section 3.3). Finally, an example of such conversion is shown in Section 3.4.

3.1 SSA Encoding

SSA representation is a form of program encoding in which each variable is assigned exactly once. Since, in a program, one variable may be assigned more than once, for each original program variable a set of SSA variables is introduced. Each assignment into a variable v is replaced by an assignment into v_i, where v_i is a fresh, not yet assigned variable [13].

In the SSA form at each join point of the original program an additional assignments are needed – so-called ϕ (phi) nodes. ϕ node has form $x = \phi(y, z)$, which means that to x is assigned value of y if the node was reached through the first entering edge and value of z is assigned to x if the node was reached through the second entering edge [2].

The logical formula corresponding to the original program is then a conjunction of SSA formulae for all program statements. The formulah can be used to reason about the program using an SMT solver. For acyclic code, the SSA form is a formula that exactly represents the strongest post condition of running the code.[5]

3.2 Encoding Control-Flow

The SSA form encodes information about data-flow among variables by design. However, no information about control flow of the program is contained in this representation. Such missing information may be whether a body of a specific condition was executed or whether a specific part of code was reached. For keeping track of these information, special variables called *guards* are introduced into the SSA representation in 2LS. A guard encodes a boolean information whether a specific program location was reachable and under what conditions. An example can be a guard at the begging of the analyzed program which is always set to `True` (assuming that the program can be always started) or a guard after a loop, which encodes that the loop has finished. The latter guard is encoded as a conjunction of a guard encoding that the loop head is reachable and of a guard encoding that the loop condition is `False` and therefore the loop terminated.

3.3 Over-approximation of Loops

The logical formula created from the SSA form is made acyclic in order to be usable with SMT solver. This is done by cutting loops at their end. For this, each ϕ node that selects between initial value of a variable that is edited in the loop and the value that comes from the loop end is edited. In such ϕ node the value coming from the loop end is represented by a new free variable i^{lb} – so-called *loop-back value*. Also the non-deterministic selection is done by a loop-select variable g^{ls} that is also a free variable. In this way the SSA form is acyclic.

Since these two introduced variables are free, the SSA form with cut loops is an over-approximation of the program being represented. To refine the over-approximation, a *loop invariant* is used. A loop invariant constrains the value of a variable by a property that holds for each iteration of the loop. As an example for a variable i that is being looped from one to five a loop invariant could have the form:

$$i^{lb} \geq 1 \wedge i^{lb} \leq 5 \tag{3.1}$$

3.4 Conversion into SSA encoding

In this section a process of converting a source program in C language into an SSA form is demonstrated. This algorithm consists of two steps. Firstly converting C program into a GOTO representation, which represents the source code in the form of a control-flow graph where locations contain program statements and are connected by edges which represent possible program flows. Second step is conversion of the GOTO representation into SSA form, which consist of 4 main actions:

- **Split variables** - The basic property of SSA form is that each variable is assigned exactly once. To comply with this property, each occurrence of a variable on the left side of assignment is replaced by a new name, usually created as a combination of the variable name and of the number of occurrence. If a variable is on the right side of an assignment or used in a different statements, then this variable is replaced with its last assigned version.

- **Introduce guards** - This 2LS specific property of SSA form, which was introduced in Section 3.2, requires to introduce guards into the SSA representation. This is done

by creating a new guard at the beginning of each non-branching block. This is mainly important for places that branch or loop.

- **Introduce ϕ nodes** - Another new assignments that are needed to be added into the SSA form are ϕ nodes. These nodes are introduced for each conditional and looping statement. More specifically for each variable that is modified inside these statements. For conditions the selection in the ϕ node is controlled by the branch condition. For loops a free boolean loop-back variable is used as described in Sectionoverapprox.
- **Over-approximate function calls** - Function calls are replaced by the over-approximating placeholders. More about this step can be found for example in [13].

This conversion is illustrated on a simple program below. Also a control flow graph (CFG) is shown in Figure 3.4 to demonstrate the SSA form better.

```
1 int i = 10;
2 while (i > 0)
3     i--;
4 return 0;
```

Figure 3.1: The example input source program [14]

```
1   signed int i;
2   i = 10;
3 1:
4   IF !(i >= 1)
5       THEN GOTO 2;
6   i = -1 + i;
7   GOTO 1;
8 2:
9   return_value = 0;
10  dead i;
```

Figure 3.2: Corresponding GOTO representation to source program from Figure 3.1 [14]

The SSA form in Figure 3.3 corresponds to statements from the GOTO representation from Figure 3.2. In the following text, each line of the SSA form will be shortly described and its form will be explained.

Line 1 of the SSA form contains $guard_0$ that encodes that the program is reachable. As mentioned before, this guard is set to `True` as it is assumed that every program can start.

Line 2 of SSA corresponds to first two lines of GOTO representation. Variable `i` was renamed to i_1 since this is the first occurrence of this variable on the left side of an assignment.

Line 3 contains a ϕ node for the variable i that is being modified inside of the upcoming loop. The node assigns to the variable i_2^{phi} either a loop-back variable i_4^{lb} or a previous value of i which is now named i_1. This selection is based on the value of a loop select variable $guard_4^{ls}$.

1 $guard_0$ == TRUE
2 i_1 == 10
3 i_2^{phi} == ($guard_4^{ls}$? i_4^{lb} : i_1)
4 $cond_2$ == !(i_2^{phi} >= 1)
5 $guard_2$ == $guard_0$
6 i_3 == -1 + i_2^{phi}
7 $guard_3$ == (!$cond_2$ && $guard_2$)
8 $cond_4$ == TRUE
9 $main_5^{return\_value}$ == 0
10 $guard_5$ == ($cond_2$ && $guard_2$)

Figure 3.3: Corresponding SSA representation to source program from Figure 3.1 [14]

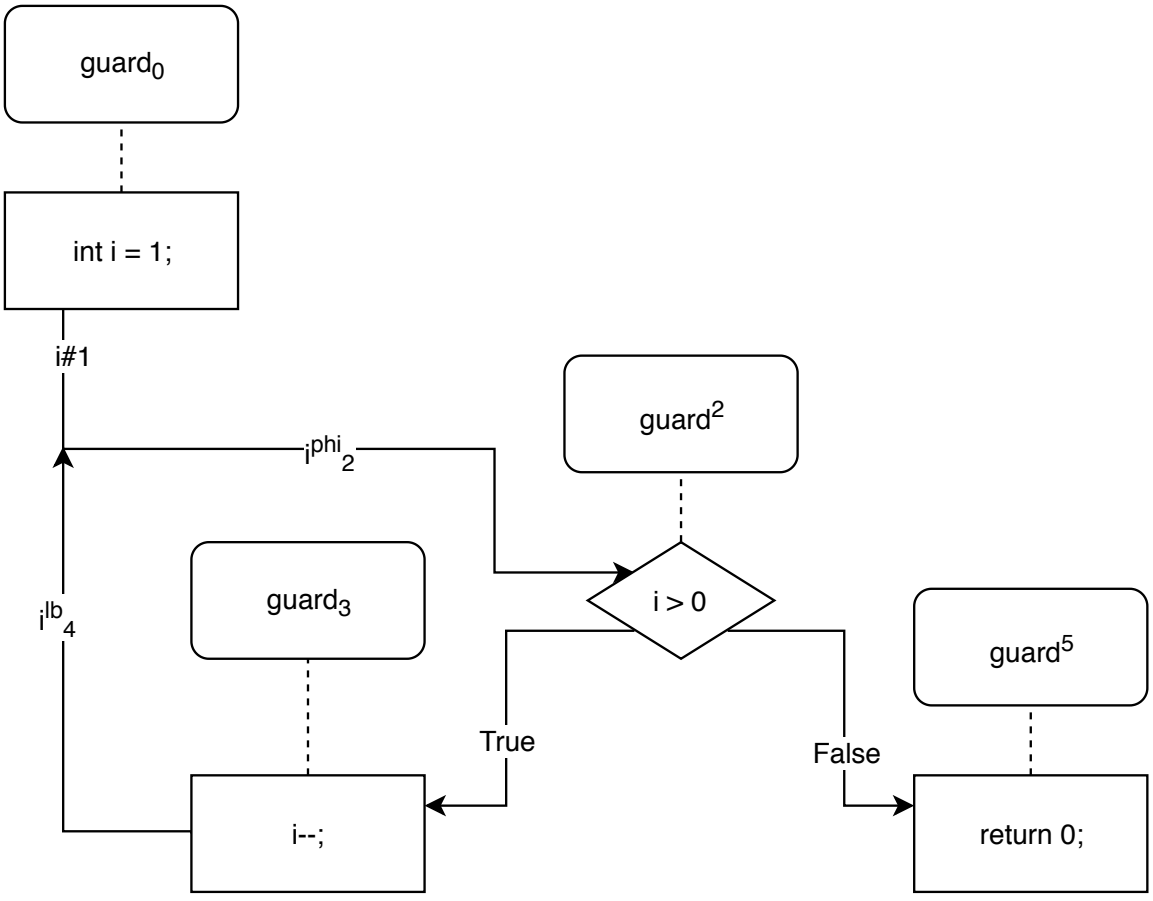


Figure 3.4: CFG representing SSA form from 3.3 [14]

Lines 4 and 8 both represent a loop condition. However, since in C language there are two types of loops – ones with condition at a loop head (for and while) and ones with condition at the loop end (do-while), the SSA form needs two conditions. One is always unused and set to `True`, which in this case is $cond_4$. The other condition contains the loop condition directly.

On Line 5 is a guard called $guard_2$ that encodes the reachability of the loop head. In this example it is reachable when the program itself is reachable and therefore equals to $guard_0$.

Line 6 directly corresponds to the 6th line of the GOTO representation. It is the loop body.

$guard_3$ on the line 7 encodes the reachability of the loop body. It encodes that the loop condition holds and that the loop head is reachable. When these two conditions are true, then the loop body is reachable as well.

Finally, line 9 represents a reachability of the code after the loop. Therefore it is a conjunction of $guard_2$ (expressing that the loop head was reachable) and $cond_2$ (expressing that the loop condition does not hold).

Chapter 4

Template-based Verification

It is easier to compute inductive invariants than the set of all reachable states (even when represented by an abstract domain). However, it is not guaranteed that such inductive invariant includes the starting state nor that it is the least one. Finding inductive invariants using abstract interpretation means solving the following formula:

$$\begin{aligned} \exists_2 Inv. \forall x, x'. (Start(x) \Rightarrow Inv(x)) \wedge \\ (Inv(x) \wedge Trans(x, x') \Rightarrow Inv(x')) \end{aligned} \tag{4.1}$$

To check for safety, it is needed to check satisfiability of the formula:

$$\forall x : Inv(x) \Rightarrow \neg Err(x) \tag{4.2}$$

The core advantage of the *kIkI* algorithm is in the *inference of inductive invariants*, which, as seen in Formula 4.1 needs to be solved by solver capable of handling (the existential fragment of) second-order logic. As today no reasonably efficient solver for this type of logic exists, the problem is reduced into an iterative use of quantifier-free first-order logic for which and SMT solver may be used. This reduction is done by so-called *templates*, which are described in Section 4.1. The iterative process of solving these formula is described in subsection 4.1.1.

Also it is more efficient to convert the source program into *static single assignment* form (described in Section 3.1) as in this form the whole source program is described by a logical formula.

4.1 Invariant Inference via Templates

A template is a fixed, parameterised first-order logic formula that represents an abstract domain which selects only those properties of the analyzed program which are relevant for the analysis.

Template has the form $\mathcal{T}(x, \delta)$, where x represents program variables and δ represents template parameters. The direct reduction of Formula 4.1 is then:

$$\begin{aligned} \exists \delta. \forall x, x'. (I(x) \Rightarrow \mathcal{T}(x, \delta)) \wedge \\ (\mathcal{T}(x, \delta) \wedge T(x, x') \Rightarrow \mathcal{T}(x', \delta)) \end{aligned} \tag{4.3}$$

Formula 4.3 contains quantifier alternation ($\exists\forall$), which is challenging for today's SMT solvers. Therefore, the formula is negated (to turn $\forall$ to $\exists$) and the parameter δ is deduced

by iteratively solving Formula 4.6 for different choices of d as value of δ (described in section 4.1.1). Here, d corresponds to an abstract value that represents the set of all x that satisfy the formula $\mathcal{T}(x, d)$. Also, an abstract value $\bot$ is defined to represent an empty set and $\top$ to represent the whole domain x. Subsequently, the following formulae may be defined:

$$\mathcal{T}(x, \bot) \equiv false \tag{4.4}$$

$$\mathcal{T}(x, \top) \equiv true \tag{4.5}$$

Formula 4.6 describes an invariant if and only if is unsatisfiable [13].

$$\begin{aligned} \exists x, x'. \neg(I(x) \Rightarrow \mathcal{T}(x, \delta)) \vee \\ \neg(\mathcal{T}(x, d) \wedge T(x, x') \Rightarrow \mathcal{T}(x', d)) \end{aligned} \tag{4.6}$$

4.1.1 Algorithm for Invariant Inference

The problem of having $\exists\forall$ in Formula 4.3 is solved by iterative solving formula 4.6. The basic algorithm for solving this formula using SMT solver is to repeatedly check satisfiability of the formula for abstract value d. The algorithm starts with setting up the initial value of d to $\bot$ and solving formula 4.7. If this formula is satisfiable, the model of satisfiability is joined with the current invariant. This *join* is specific for each domain. When no model of satisfiability can be found, it means an invariant was found. Convergence of this method is guaranteed (as the abstract domain is finite), but 2LS tool uses different optimization for different abstract domains to make it more effective.

$$\mathcal{T}(x, d) \wedge Trans(x, x') \wedge \neg(\mathcal{T}(x', d)) \tag{4.7}$$

An optimization in the SMT solver, called *incremental solving*, speeds up solving of this formula. The main idea lies in the fact that $Trans(x, x')$ does not change and therefore does not need to be re-solved (it is solved the first time and then is assumed to hold for all subsequent iteration). It is only being checked if it is still satisfiable with a different values of the invariant [11].

4.2 Guarded Templates

Since verification in 2LS is based on logical formulae generated from SSA form, invariants do not hold information in which context of program they are valid – in contrast to using for example a control-flow graph in which case invariants computed with abstract interpretation can be bound to a certain program state. To overcome this problem in 2LS, so-called *guarded templates* are used.

A guarded template has a form

$$G \Rightarrow \mathcal{T}(x, d) \tag{4.8}$$

where G is a conjunction of SSA guards. The computed invariant is limited to be only used when its guards hold true – which means that the part of code that the invariant describes was reached and executed.

4.3 Loop Invariants

Another limitation that is needed to be introduced is limitation of loop-back variables (described in Section 3.3). A loop invariant describes a condition that holds for each iteration of the loop body at its end (just before branching back to the loop head). It is expressed as a guarded template and it has form

$$(g_{lh} \wedge g_{lh}^{ls}) \Rightarrow \mathcal{T}(x_l, d) \tag{4.9}$$

where x_l is a set of loop-back variables for a loop l and g_{lh} is a guard encoding that the head of loop l is reachable. g_{lh}^{ls} is the loop-select that selects between value of variable coming to the loop head and the loop-back value coming from the end of the loop (as described in Section 3.3).

Example of Computing a Loop Invariant

In this example a computation of a loop invariant is shown on an example from Figure 3.3. The program contains only one loop and the set of all its loop-back variables contains just one variable, i_4^{lb}. In this example an *interval abstract domain* [7] is used. For each variable an interval is computed in which the value of the variable lies. hence the template has the form:

$$\mathcal{T}(\{i_4^{lb}\}, (d_1, d_2)) \equiv i_4^{lb} \geq d_1 \wedge i_4^{lb} \leq d_2 \tag{4.10}$$

where d_1 and d_2 are template parameters whose value is inferred during the analysis.

An accelerated solving (described in subsection 4.1.1) is now used and more specifically Formula 4.7 is being iteratively solved to infer the template parameters d_1 and d_2. The algorithm takes as long as the formula is satisfiable. For simplicity of this example the transition relation $Trans(x, x')$ is assumed to be always satisfiable (since we assume a syntactically correct program represented by an acyclic SSA form with over-approximated effect of loops) and therefore in each iteration of the loop only the current instance of the invariant is solved.

Formula 4.7 contains two instances of the template – $\mathcal{T}(x, d)$ and $\mathcal{T}(x', d)$. The first instance describes the loop invariant before the loop body is executed (here represented by $Trans(x, x')$) and the second instance represents the same invariant but after the body has been executed. Therefore the first template is defined as:

$$(guard_2 \wedge guard_4^{ls}) \Rightarrow \mathcal{T}(\{i_4^{lb}\}, (d_1, d_2)) \tag{4.11}$$

and the second one as:

$$(guard_2 \wedge guard_3) \Rightarrow \mathcal{T}(\{i_3\}, (d_1, d_2)) \tag{4.12}$$

where $guard_2$ encodes reachability of the loop head, $guard_4^{ls}$ is the loop-select variable for the loop-back variable i_4^{lb}. $guard_3$ guards the reachability of the end of the loop body and therefore also the reachability of the SSA variable i_3 which represents the value of i at the end of the loop body and which corresponds to the loop-back variable i_4^{lb}.

The iterations of the accelerated solving work now like this:

- From subsection 4.1.1 is known that the initial value of the template parameter d is $\perp$. Also from Formula 4.4 is defined that $\mathcal{T}(x, \perp) \equiv false$. Taking these two facts and including it into loop invariant 4.11 the following formula is obtained:

$$(guard_2 \wedge guard_4^{ls}) \Rightarrow false \wedge \neg((guard_2 \wedge guard_3) \Rightarrow false) \tag{4.13}$$

Since the right side of the implication is always false for the whole formula to be satisfiable left side also needs to be false. But as $guard_2$ is true, the only option is for $guard_4^{ls}$ to be false. This implies that i_2^{phi} is assigned the value of i_1 which is 10. Subsequently to i_3 the value 9 is assigned. Variable i_3 is the variable holding the value of i at the end of the loop body. Therefore, this value is used to refine the current invariant and since both d_1 and d_2 are not defined both are set to this value. Hence, after the first iteration the current invariant is following formula:

$$i_4^{lb} \geq 9 \wedge i_4^{lb} \leq 9 \tag{4.14}$$

- In the second iteration the current invariant is applied to Formula 4.7 and it has form:

$$\begin{aligned}(guard_2 \wedge guard_4^{ls}) \Rightarrow (i_4^{lb} \geq 9 \wedge i_4^{lb} \leq 9) \wedge \\ \neg((guard_2 \wedge guard_3) \Rightarrow (i_3 \geq 9 \wedge i_3 \leq 9)\end{aligned} \tag{4.15}$$

  In order to understand this formula it may be split into two separate formulae which both need to be true. In the first conjunct left side of implication is true so does the left side need to be true. Only possible way how to make this formula satisfiable is to select 9 as the value of the variable i_4^{lb}. The second part is negated and hence the inside formula needs to be false – and since the left side of the implication is true, the only way how to make the whole implication false is to find a value of i_3 for which it does not hold. In this case it is number 8 to comply with $Trans(x, x')$.

  Since a model of satisfiability was found the invariant can be updated. Since $i_4^{lb} = 9$, then i_3 needs to equal 8, which is different from 9. Therefore the invariant is modified to:

$$i_4^{lb} \geq 8 \wedge i_4^{lb} \leq 9 \tag{4.16}$$

- Other iterations do not differ much from second iteration, only the value of i_4^{lb} is always decreased in the same manner.

- The iteration ends once a model of satisfiability cannot be found. This happens when $guard_3$ becomes false and therefore the second conjunct is always false and therefore the whole formula is unsatisfiable. Since $guard_3$ is conjunction of $guard_2$, which is always true, and $cond_2$, for it to become false the condition needs to be false. This happens when i_4^{lb} reaches value 0. After this iteration the final computed invariant has form:

$$i_4^{lb} \geq 0 \wedge i_4^{lb} \leq 9 \tag{4.17}$$

Chapter 5

Strategy Solver

2LS uses abstract domains (in the form of templates) to analyze various properties of programs. Each abstract domain specifies a template form for computing invariants, a domain of abstract values (i.e. of values of the template parameter) and an algorithm for joining a model of satisfiability returned from the solver and the current instance of the template.

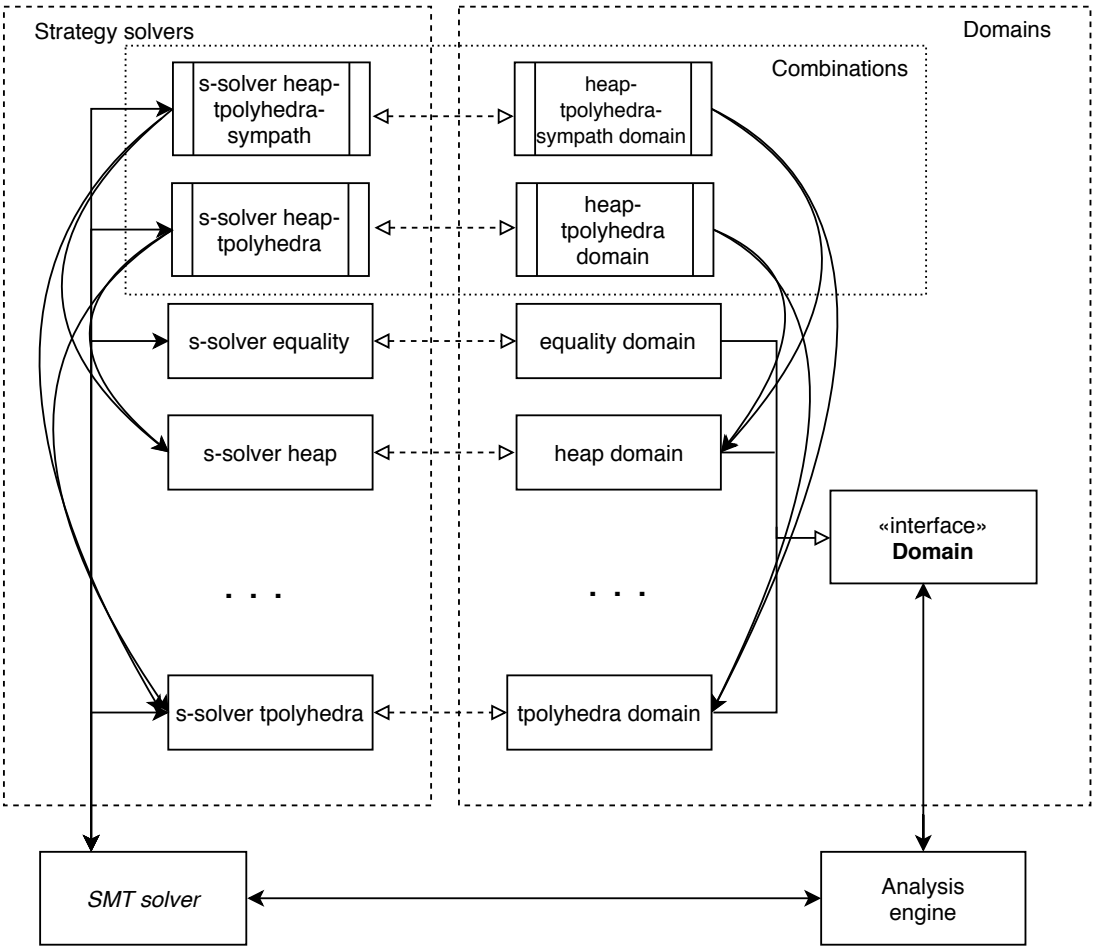


Figure 5.1: Structure of 2LS from the solvers point of view

An instance of the template (i.e. template $\mathcal{T}(x, d)$ with some concrete value d of the parameter) needs to be translated into formula that SMT solver understands. Also the opposite process is needed for converting the model of satisfiablity from SMT solver into a logical formula. For both of these tasks, a so-called *strategy solver* is used. In the current implementation, there is one strategy solver for each domain. The structure of domains and corresponding strategy solvers is displayed in Figure 5.1.

Creating a new domain therefore means creating the domain itself and then creating a strategy solver for this domain. It has been noted, that all strategy solvers have a very similar logic. That led to the idea of creating a single strategy solver, that could suit all domains. The specific behaviour of domains, that is in the current implementation encapsulated in strategy solvers, can be moved into domain itself. This behaviour then can be accessible using unified domain interface.

It is also worth mentioning that there are combination domains with corresponding strategy solvers. Such domains combine more simple domains to unveil the power of 2LS approach. To these domains is devoted Chapter 6. This chapter only focuses on simple domains.

5.1 Domain Interface

In order to make a generic solver possible, each domain must implement some basic interface. To create this interface, domains were analyzed and it was found out that there are two types of domains – a set based and a row based domains. In the set based domain, all rules to be checked are stored in a set and are checked with a SMT solver one by one. The row based domains use a vector to store all the rules and all rules are checked at once with the SMT solver. To work around this difference, during the iterative solving, in the each iteration, the first item in the set-based domains is taken and inserted into a one-item vector. Than it is possible to look at this as a row-based domain.

In the following part the type `exprt` means expression, that an SMT solver understands and `valuet` represents the current invariant. Each interface must implement following methods:

- `exprt to_pre_constraints(valuet)` - Get a formula expressing the current state of the domain invariant. This represents $\mathcal{T}(x, d)$ from Equation 4.7.
- `void make_not_post_constraints(valuet, *exprt)` - Get the formula expressing negation of the current invariant instance after execution of the transition relation. This represents $\neg(\mathcal{T}(x', d))$ from Equation 4.7.
- `vector<exprt> get_required_values(row)` - Get a vector of variables, those values are needed from the SMT solver. This method is needed since domains do not interact with SMT solver directly but only through strategy solver. However domain needs to know about the model of satisfiablity. And since this model is often very large and domain needs to know what value has been assigned to a one specific variable this method is used. Domain specifies list of variables for which wants to know values in the current model of satisfiability.
- `void set_values(vector<exprt>)` - Set values of requested variables. A vector of values that the SMT solver assigned to variables required by `get_required_values` is given as a parameter. This is counterpart to the previous method.

- `bool edit_row(row, valuet, bool)` - For each satisfiable row (in set-based domains it is the first item in set) this method is called. It gives opportunity to the domain to edit its internal state. Return true if want to refine the invariant more in the next iteration or false, if the invariant cannot be refined any more.

There are also methods, which are not mandatory for the domain to implement. When not overridden, default implementation is used instead. Domains can use these methods to adjust their internal state in different phases of the invariant inference algorithm.

- `exprt initiliaze_solver()` - This method is called only when a new solver is created. Domains may want to initialize the solver. When domain does not require any initialization steps the default behaviour is to do nothing.
- `void pre_iterate_init(valuet)` - This method is called on every cycle of the iterative solving exactly once at the beginning of each iteration. It is mostly designed for domains to prepare internal state for the next iteration.
- `bool has_something_to_solve()` - Ask the domain whether there is anything to solve and therefore it makes sense to continue. Called at the beginning of each iteration. The sole purpose of this method is to inform strategy solver that the domain does not have anything that would need solving.
- `bool not_satisfiable(valuet, bool)` - If no model of satisfiablity for the current invariant has been found, this method is called. Domain should return true if wants to refine the invariant more in the next iteration (assuming that it edits the invariant before next iteration) or false, if the invariant cannot be refined any more.
- `exprt make_permanent(valuet)` - Enables domain to write clauses into SMT solver at the end of each iteration. The returned formula from this method is made permanent within the SMT solver (i.e. will be considered to hold for all following iteration).
- `void post_edit()` - After calling `make_permanent` method, last edits in each iteration can be made by calling this method.

5.2 Generic Solver

Using the proposed interface from section 5.1 it is possible to create a generic strategy solver. The first step in a generic solver is to let the domain know, that there is going to be a new iteration of the solving process, since some domains may want to make some initialization. This can be simply done by calling `preIterateInit` method. Then the domains is inquired about the state – whether the invariant can be refined more or not. This is mostly useful for a set based domains, where empty set of "to-be-refined" rules means the solving can be halted and there is no need to start another iteration of the solving process. This functionality is in the `somethingToSolve` method, which default behaviour is to return true. If the condition was satisfied the main solving part can be executed.

On line 3 in Algorithm 1 a new context is created in the SMT solver. Since there is only one instance of SMT solver in 2LS shared for all analyses. Also it is incremental solver which keeps previous states and those are considered to always holds. However since this iterative process is based on trying and refining, it may happen that in the solver a non-solvable formula would end up and all following computations would not be solvable.

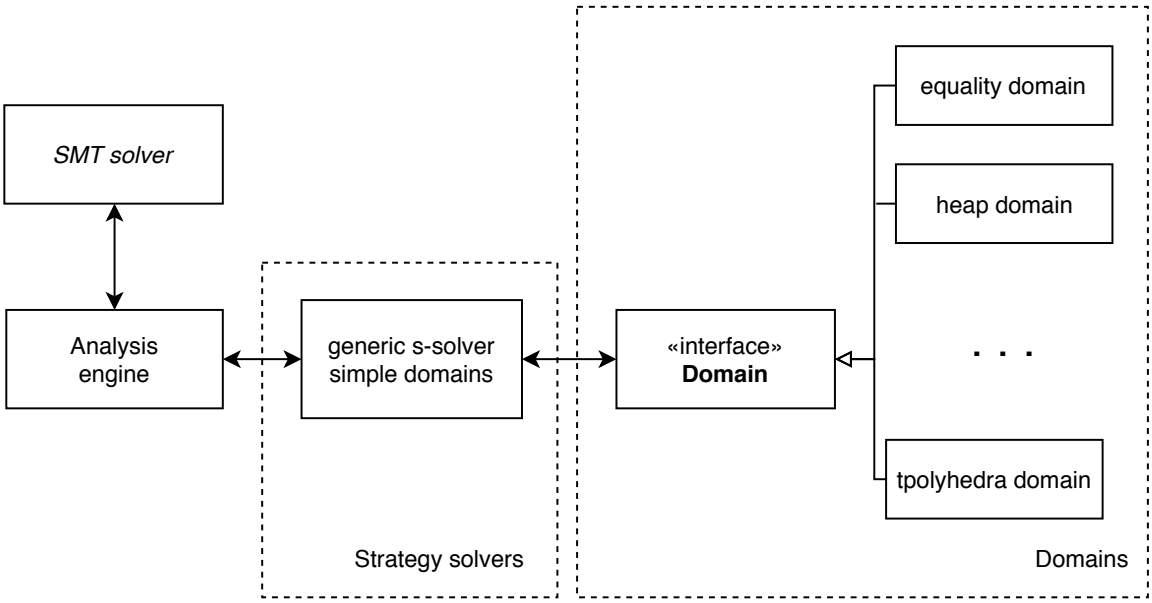


Figure 5.2: Structure of 2LS from the solvers point of view with generic solver

Algorithm 1: Algorithm of the generic solver

```
1  domain.preIterateInit();
2  if domain.hasSomethingToSolve() then
3      smtSolver.newContext();
4      smtSolver ← domain.toPreConstraints();
5      smtSolver ← domain.makeNotPostConstraint();
6      if smtSolver is satisfiable then
7          for row in domain.rows do
8              if smtSolver.get(row) is True then
9                  values ← domain.getRequiredValues(row);
10                 solverValues ← smtSolver.get(values);
11                 domain.setValues(solverValues);
12                 domain.editRow(row);
13             end
14         end
15     else
16         domain.notSatisfiable();
17     end
18     smtSolver.popContext();
19     smtSolver ← domain.makePermanent();
20     domain.postEdit();
21 end
```

Therefore 2LS enables to create a new layer and all new changes are stored in this layer. In this way it is easy to remove all added changes.

First step in the main solving part of the algorithm is to insert into the SMT solver current invariant. This invariant can be obtained from the domain with the method `toPreConstraints`. As explained in the Section 2.2 also negated version of the next step of the program is needs to be written into the SMT solver. This form can be provided by the `makeNotPostConstraint` method. In this step the SMT solver can be executed and checked if the form has model of satisfiablity. If there is such model, it means the current domain state is not yet invariant and further refining is needed. That is done by iterating through all rows in row based domains or all lines in lines based domains and checking with the SMT solver if are satisfiable. If so, then the domain invariant is altered adequately. If no such model was found by the SMT solver, then domain is informed about this state by calling `notSatisfibale` methods. Usually this means end of the invariant refining, however domain may want to continue. This usually happens with set-based domains if there are more items to be refined.

Finally the new context in the SMT solver is removed and SMT solver is therefore in the state it was before the solving. However domain may want to make some expression invariant for all further checks and therefore may write into the SMT solver an expression with method `makePermanent`. The last step, contrary to `preIterateInit`, is method `postEdit` which enables domains to clean-up right before the strategy solver finishes.

The algorithm of the generic solver is shown in Algorithm 1.

This proposed solver can replace all strategy solvers of migrated domains. Here migrating domain means moving some functionality from the domain's strategy solver into methods implementing the proposed interface. Structure of 2LS from the solvers point of view with the generic solver is displayed in Figure 5.2.

Chapter 6

Combination Strategy Solvers

The 2LS tool supports also solving inference of invariants in more than one abstract domain. This feature provides a possibility to analyse complex properties of programs, such as those that require reasoning in multiple abstract domains at the same time. As a concrete example, an abstract data structure such as a linked list can be assumed, where modifications of the list are done based on the numeric value of each individual item.

Introduction of such a new analysis can be view as a new abstract domain (and in practise it is a new domain) for any new combination of existing domains. Since 2LS uses logical formulae as program representation, introduction of such new complex analyses is rather simple as it mostly requires just composition of logical formulae produced by other, already implemented, domains.

This chapter proposes generic solvers for some kinds of domain combinations. Section 6.1 introduces to two possible approaches for combining domains, namely *product domains* and *domains with symbolic paths*. Since new domains are introduced into 2LS, there is also need for a strategy solver for every new combination domain. Similarly to Section 5.2, where a generic solution for a strategy solver for simple domains was introduced, in Section 6.2 a generic solution for each type of new domain is described. Such a generic solution can work with any number of domains and also any types of domains – these can be dynamically picked at the start of the analysis. This makes it very easy to use for different combinations and to analyse new properties without a need to implement any new domains.

6.1 Combinations of Domains

In case when more than one abstract domain is used in a single analysis, it is referred to it as a combination of domains and a special strategy solvers are needed. In theory, the simplest solution is to take two domains, iteratively solve the first one and then the second one. In such approach a shared SMT solver is used so the second domain is effected by the first domain. At the end a disjunction of both results is considered.

However, this basic approach does not bring any significant improvement that can be achieved with smarter combining of domains. In 2LS there are two specific kinds of domain combinations – *Product Domains* and *domains with symbolic paths*. These two types are described in the following subsections.

6.1.1 Product Domains

A product domain is a way of combining any amount of different domains to make the analysis much more interesting. This can be done simply by using a *Cartesian product* of different domain templates – domains are used side-by-side while reinforcing each other (in each iteration of a domain solving, invariants computed so far in other domains are used as constraints, allowing to infer stronger invariants). The current implementation of 2LS contains, for example, combination of heap and interval domains, which enables 2LS to analyse pointer and numerical data in a single analysis and what is more important, which enables analysis of the numeric data on the heap.

Product domains are universally combinable and the result of such combination for templates $\mathcal{T}^1, \mathcal{T}^2 \ldots \mathcal{T}^n$ is their conjunction $\mathcal{T}^1 \wedge \mathcal{T}^2 \wedge \cdots \wedge \mathcal{T}^n$.

In each iteration of solving such domain combination, the strategy solver needs to run one iteration of a corresponding solver for each participating domain in the context of all invariants found by all other solvers. This approach can be nicely demonstrated on combination of heap and interval abstract domains. First, one iteration of the heap strategy solver is run in the context of the invariant from the interval domain. Then, one iteration of the interval strategy solver is run in the context of the invariant from the heap domain. Not only that this makes it possible to analyse more complex properties of analyzed programs, but also it can find invariants faster and more efficiently.

6.1.2 Domains Combination over Symbolic Paths

While product domains enabled the analysis to combine more than two domains into one stronger analysis, this type of domains combination enables combining a single domain with the domain of symbolic paths. The domain can be a product domain so technically multiple domains are combinable as well.

The core idea of this approach lies in solving for each symbolic path in the program an invariant in the chosen domain. As an example from 2LS can be used a heap domain with symbolic loop paths, where the abstract value maps particular symbolic loop paths to sets of parameters of the heap domain.

The combination template can be view as a power template, where the domain of symbolic paths acts as a base domain and the other domain acts as an exponent domain [12]. The template then assigns to each symbolic path a template of the other domain.

Solving such domain means iteratively going through all not yet explored symbolic paths and finding invariant for each one.

6.2 Generic Combination Solver

In the previous text were introduced two types of domains combination with short explanation of the corresponding solvers. In the current implementation, two combination domains exist – one for each mentioned type. Introducing a new combination domain means creating a new domain and a corresponding strategy solver. Both of these steps are rather simple and very similar to other combination domains.

To ease the process of creating a new combination a generic solution is proposed. Having just one generic domain and one generic strategy solver for each type of domain combination would enable 2LS to use combination of any domains, which can be selected at a run time. This way 2LS can support many new domains without any additional work.

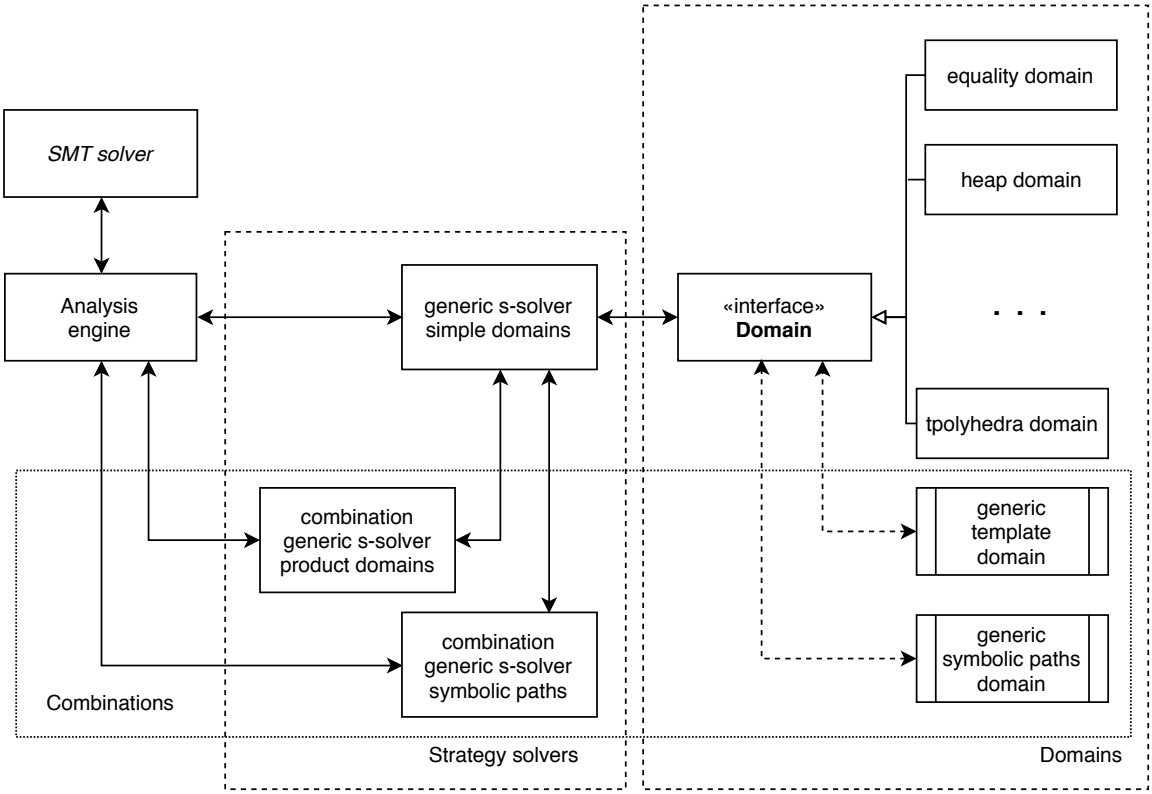


Figure 6.1: Structure of 2LS from the solvers point of view with generic combination solver

Equally to introduction of a generic strategy solver for simple domains in Section 5.2, in this section introduction of two more generic strategy solvers is presented – one for each type of domain combination. Structure of 2LS with generic combination solvers is shown in Figure 6.1.

6.2.1 Generic Combination Solver for Product Domains

The basic idea behind generic combination strategy solver for product domains is running one iteration of a specific strategy solver for one abstract domain in the edited context of invaraints of other domains. The proposed algorithm is shown in Algorithm 2.

The whole iterative solving loops as long as at least one domain refined its invariant. This is controlled with the variable `updated`. Each iteration of the solving process consists of a loop through all domains. In the product domains combination there can be any number of domains and they can be of any type.

In this loop, firstly a new context for the solver is created (new context is just creating of a new layer, which is normally considered to be part of the solver, but any changes done in this layer can be easily removed by popping the context). To the new context, current invariants from all abstract domains are written. It is advised not to write the invariant of the current domain, as this invariant is written into the solver in the strategy solver of the given domain (see Algorithm 1).

Then, it is possible to run one iteration of the strategy solver for the current abstract domain, which, thanks to the current edited context, is run in the context of invariants from other solvers' templates. After this step, the solvers context can be popped.

Algorithm 2: Iterative algorithm of the generic combination solver for product domains

```
1  updated = True;
2  while updated do
3  |   updated = False;
4  |   for domain in domains do
5  |   |   smtSolver.newContext();
6  |   |   for d in domains- domain do
7  |   |   |   smtSolver ← d.toPreConstraints();
8  |   |   end
9  |   |   lastUpdated = solvers [domain ].iterate();
10 |   |   smtSolver.popContext();
11 |   |   updated = updated or lastUpdated;
12 |   end
13 end
```

This approach needs 3 sets:

- **Set of all domains** - The solver needs to know all domains that take part in the analysis. These domains can be held by a generic domain, which is able to provide list of all domains to a strategy solver.
- **Set of all templates** - For each domain a template for this domain is needed to be know. Similarly to the previous set, this information also can be held by a generic domain in the similar fashion.
- **Set of all strategy solvers** - The last set that is needed for this combination is set of all strategy solvers. This information may be kept by a generic strategy solver itself.

It is also very important that it is known which template is for which domain and with which solver is this domain being solved. This may be done in many ways, but in this work a very simple approach was selected and that being order of items in vector. First domain in the set of domains has template on the first position in the set of templates and is solved by the first strategy solver.

6.2.2 Generic Combination Strategy Solver over Symbolic Paths

A generic strategy solver for combination of domains over symbolic paths combines only two domains. The base domain provides symbolic paths and the exponent domain solves invariants for these paths.

To support domain combination with symbolic paths in a generic way, all domains must be able to provide some functionality. All of this functionality is directly connected with symbolic loop paths. Following methods need to be added:

- `symbolic_patht get_symbolic_path()` - Get the current symbolic path used in the last iteration.
- `void restrict_to_sympath(symbolic_paths)` - Restrict the domain to only compute invariants in the symbolic path given as a parameter.

- `void undo_restriction()` - Remove the symbolic path restriction. This is the opposite of `restrict_to_sympath`.
- `void eliminate_sympath(vector<symbolic_patht>)` - Restrict the domain to avoid computing invariants in the given symbolic paths. This is used to prevent exploration of paths for which an invariants has been already computed.

Even though that there is no need for sets of domains, templates and solvers, the algorithm is a bit more complex compared to product templates. The main idea of the algorithm is shown in Algorithm 3.

Algorithm 3: Iterative algorithm of the generic combination solver with symbolic paths

```
1  baseDomain.iterate();
2  symbolicPath = baseDomain.getSymbolicPath();
3  while symbolicPath do
4  |   exponentDomain.restrictToPath(symbolicPath);
5  |   while exponentDomain.iterate() do
6  |   end
7  |   exponentDomain.undoRestriction();
8  |   baseDomain.eliminateSymbolicPath(symbolicPath);
9  |   baseDomain.iterate();
10 |   symbolicPath = baseDomain.getSymbolicPath();
11 end
```

Chapter 7

Implementation

This chapter describes the implementation part of this work. Firstly in Section 7.1 is described the current architecture of 2LS tool with explanation of the most important steps of program analysis in the tool with focus on those parts, which later on were altered and enhanced.

Following in sections 7.2 and 7.3 the result of implementation and integration of generic (introduced in Section 5.2) and generic combination (introduced in Section 6.2) solvers are described. These sections contain the most important implementation details, steps that were needed to be done as well as description of the main changes to the existing code.

In the given sections is also explained how creation of new abstract domains differ with these new changes.

7.1 2LS Architecture

2LS has rather complicated structure in which many components and steps take part during program analysis. To make it clearer how the tool works and where the changes introduced by this work fell under, in this section the architecture of this tools is shown and explained. The main steps performed by 2LS are depicted in Figure 7.1 [15].

The architecture of 2LS can be divided into three main parts – *front end*, *middle end* and *back end*. Many of these steps use components from CPROVER infrastructure [6] upon which 2LS is build.

Following is description of the main parts and steps of the program analysis.

7.1.1 Front End

The command-line front end firstly configures 2LS according to user-supplied parameters. In this first step user may specify all available settings such as frontend options (like type of endians, architectures or bit-widths), middle end options that affect type of analyses that are going to be performed (for example checking of overflows, divisions by zeros or memory leaks) and back end options containing mostly selection of abstract domains to be used.

All possible options are available through `-help` switch. The only required parameter is the C program to be analyzed.

After configuring the tool, the input program is parsed (by using off-the-shelf C preprocessor, such as `gcc -E`) and translated into GOTO program. The parser for converting into GOTO representation comes from CPROVER framework.

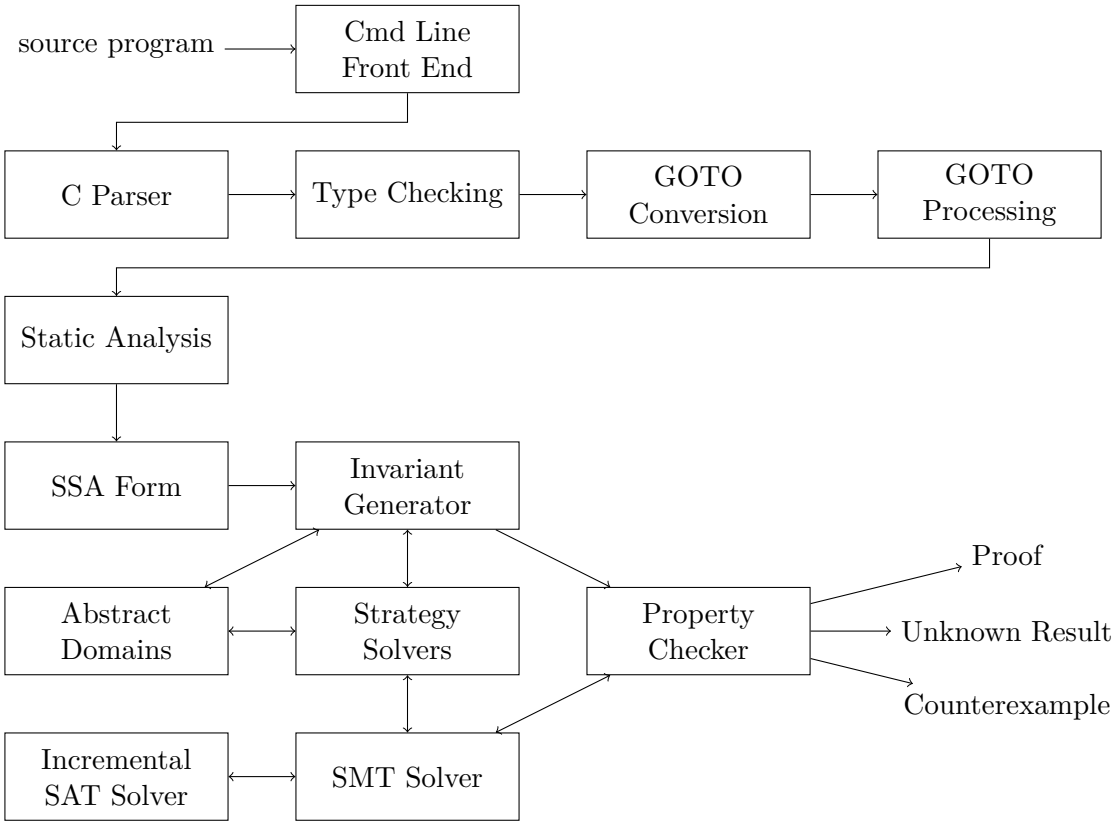


Figure 7.1: The architecture of 2LS

A GOTO program is an intermediate representation of the source program in a form of a control flow graph. On this representation further tasks are performed such as functions inlining or constants propagation. Also all non-linear control flows, such as loops and jumps, are translated into a equivalent guarded goto statements. In the end assertions that guard against invalid pointer operations or memory leaks are inserted.

7.1.2 Middle End

The middle end of 2LS is the main part where the most of analyses happen. Firstly a various static analyses are performed in preparation for conversion to a SSA form. These include *object analysis* and *assignment analysis* to collect all objects needed by a function and to determine program locations where objects are assigned.

After all static analyses are performed, the GOTO representation of the source program is converted into the static single assignment (SSA) form as described in the Section 3.1.

As in the SSA form loops have been cut at the back edges to the loop head and function calls havocked, this representation is over-approximation of the GOTO program. Subsequently 2LS refines this over-approximation by computing invariants. These are computed by abstract domains, which are selected by a command line options. In the current version of 2LS following domains are implemented:

- **Equalities domain**: A domain for analysis of equality (or disequality) of two variables. For each x,y of program variables test $x == y$.

- **Heap domain**: A domain for analysis of the shape of the domain through access paths.
- **Lexicographic ranking domain**: A domain for analysis of the program termination.
- **Polyhedra domains**: Domains used for analysis of numeric variables – integers and floats, both signed and unsigned. There are 3 implemented types of polyhedra domains:
  - **Interval domain**: A domains where for each variable x is found constant C for which $x < C$ holds true.
  - **Octagonal domain**: A domains where for each variables x,y is found constant C for which $x + y < C$ holds true.
  - **Zones domain**: A domains where for each variables x,y is found constant C for which $x - y < C$ holds true.
- **Combination domains**: A special case domains that combine two or more of the above domains to provide a stronger analysis and to check properties not only all of the domains but also properties that cannot be analysed by a single domain. Two types of such combinations exists:
  - **Product domains**: A domain where any number of domains can be used. All domains are analysed in the context of all others. This type of combination domain is introduced in 6.1.1.
  - **Power domains**: A domain that combines two domains – base and exponent. This type of combination domain is introduced in 6.1.2.

The last step in the program analysis is the checking if all assertions (user-supplied or generated ones) hold true for the computed invariants. This step is performed by solving the formula that represents the source program (which is created from the SSA form) together with the computed invariants in the SMT solver. Negations of program assertions are checked for satisfiability. If all negations are unsatisfiable, the source program is valid. If there is a satisfiable negation of a assertion it may mean two things – either the source program contains an error or the invariants were too weak and due to over-approximation the result may be wrong and therefore the result is unknown.

7.1.3 Back End

There is need for a back end SMT solvers in 2LS for both invariants inferecne and property checking. Since 2LS needs incremental solvers and the support for incremental SMT solvers is still lagging behind incremental SAT solvers, 2LS uses external instance of SAT solver. It is possible to use Glucose 4.0[1] or MiniSAT 2.2.0[2].

All the functionality that makes it possible to use incremental SAT solvers instead of SMT solvers is provided by CPROVER framework.

[1] http://www.labri.fr/perso/lsimon/glucose/#glucose-4.0
[2] http://minisat.se/Main.html

7.2 Generic solver integration

In Section 5.2 a generic strategy solver for simple domains was proposed. In this section the implementation process and result of the introduction of this solver is described.

To support a generic solver, following two steps needed to be done:

- **Implementing a needed domain interface** - First step in introducing a generic solver was enhancing domain interface as described in Section 5.1. This means declaring these new methods in `src/domains/domain.h`. In this interface some methods were required to be implemented by derived classes (so-called `pure virtual methods`). All pure virtual methods needed to be implemented in all derived classes, which means editing each and every domain and declaring and defining these methods. All other methods were given a default implementation in `src/domains/domain.cpp`.
- **Implementing the solver itself** - Next step was to implement Algorithm 1. In 2LS this means introducing a new file `src/domains/strategy_solver.cpp` (along with `src/domains/strategy_solver.h` into the tool.

After domain interface and the generic strategy solver itself have been implemented it was possible to migrate simple domains into using the generic strategy solver. Such process is individual for each domain but in general a switch from a specific to generic solver consists of following 4 steps:

- **Understanding the strategy solver** - First step in each migration was understanding the strategy solver for the domain being migrated and check how it can be fitted into the generic solver.
- **Implementing needed methods from domain interface** - Generic solver was designed to fit any domains needs and therefore enables the domains at different places of the solving process to execute any needed code. This is done by implementing a non-required methods of the domain interface. Such events may be initializing some values before solving starts, selecting values from solver or writing formulae into SMT solver. Therefore if domain needs such actions, they needed to be implemented in this step. Of course that all pure virtual methods had to be implemented as well.
- **Switching into a generic solver** - Once domain implements everything it needs a generic strategy solver can be used. This means just simply using a generic strategy solver instead of the specif one. In 2LS this is done in `src/domains/ssa_analyzer.cpp` where strategy solvers are created.
- **Removing the specific strategy solver** - Since the specific strategy solver for the domain is not used anymore and its functionality is replaced by the generic solver this unused solver can be dropped. It is simply done by removing files that implement this solver.

In this work six strategy solvers were replaced by the generic solver. From strategy solver that deal with simple domains only three were not migrated. All three solvers implement a different solving approach for interval domains. The proposed and implemented generic solution cannot replace this three solvers by design. Most likely another generic strategy solver for these solvers would be needed, but this was not in the scope of this work.

By migrating these domains twelve files were removed from the base of 2LS (six strategy solvers together with corresponding header file). However the code base did not get

much smaller (1814 lines removed and 1658 lines added) since most of the functionality from removed strategy solvers is still being implemented by domains (like join algorithm). However purpose of this change was mainly to make it easier to implement new domains, since now only the domain is needed to be implemented without implementing the strategy solver logic as well.

7.3 Generic combination solver integration

Similarly to the previous section, in this section is a description of implementation of a generic strategy solvers for combination domains that were introduced in sections 6.2.1 and 6.2.2. Although work on this two generic solvers was separate the process and changes are rather similar. Following steps were done in process of introducing these solvers:

- **Alter domain interface** - As described in Section 6.2.1 there were some further modification needed to be introduced into the domain interface. This interface in the end is used by both generic solvers for combination domains.
- **Implement generic domains** - While the generic solver for simple domains was working with the existing domains through the domain interface and therefore no new domains were needed, strategy solvers for combination domains need a special domain. Therefore for each type of combination a new generic domain has been created. These domains are generic templates that take domain types as parameters. In the 2LS tool this meant adding two new files, specifically `src/domains/combination_domain.cpp` as a generic domain for product domains and `src/domains/combination_domain_sympath.cpp` for combination over symbolic paths.
- **Implement generic solvers** - Equally as in the process of adding a generic solver for simple domains also in this process the next step is implementation of the strategy solver. For the strategy solve for product templates it was implementation of Algorithm 2 and for combination over symbolic paths an existing algorithm was refactored to accept domains as arguments. Each new solver was created in a separate file, and that `src/domains/combination_solver.cpp` and `src/domains/combination_solver_sympath.cpp` respectively.
- **Migrate to a generic solver** - Having both generic domain and generic strategy solver it was possible to move existing implementations of combinations into a generic versions. In current state of 2LS there is one implementation for each type of combination – always combining heap and interval domain. Process of switching into using a generic solution means mostly just correctly initializing generic domain and generic solver at the beginning of the analysis.
- **Remove unused domain and solver** - Finally when both combination domains were moved into a generic solution, unused domains and strategy solvers could have been removed. This meant removing two domains and two solvers, specifically `src/domains/strategy_solver_heap_tpolyhedra.cpp`, `src/domains/strategy_solver_heap_tpolyhedra_sympath.cpp`, `src/domains/heap_tpolyhedra_domain.cpp` and `src/domains/heap_tpolyhedra_sympath_domain.cpp`.

While introduction of a generic strategy solver for simple domains meant reduction in code base (mostly in number of files), with introduction of these solvers no such reduction happened. This was due to fact that two new solvers were introduced as well as two new domains and equal number of domains and solvers for a specific domain combination was removed. However, these generic solvers and domains are usable for any combination of domains and therefore no new domains and solvers are needed to be introduced in order to tests with different domain combination. This change therefore makes it very easy to experiment with different combination of domains and exploring their potential.

Chapter 8

Results and Experiments

This work introduced generic solutions for three different types of strategy solvers. Althought this work removed a lot of redundancy from the existing code, the main purpose of these changes was to ease the process of creating a new abstract domain.

To prove that these changes were effective and that it is easier to create a new domain, various experiments were conducted. Even though introducing a new simple domain should be rather simple with this new approach, no experiments were done in this area. The reason is that a new domain needs a complex abstract value definition and a join algorithm. Both of these parts are non-trivial and go largely beyond this work. However, introducing a new domain combination should also become an easy task with changes introduced in this work. Experiments with new domain combinations were performed and results are described in Section 8.1.

Since in this work a large amount of existing code was modified and a new code was introduced, it is necessary to validate that these new changes did not affect the existing implementation in a bad way. This was checked with regression tests that exist in 2LS. Results of this checks are presented in Section 8.2. As a more complex set of tests, a benchmark from the International Competition on Software Verification 2018 (SV-COMP 2018) can be used. On this benchmark also a new combinations were tried. These results are described in Section 8.3.

8.1 Experiments with new domain combination

Since this work made it very simple to combine any domains, an experiment was performed to demonstrate that it is possible to create a new domain combination that allows to verify programs which could not be correctly verified before. As an example, a program in Algorithm 4 can be used.

This program randomly (`__VERIFY_nondet_int` assumes any random integer) adds in a loop into one of two counters – a or b until one of them reaches a predefined maximum value. If both counters are equal, one of them id decremented by one. After the end of loop, program checks that the sum of the counters does not eqaul twice the maximum value (i.e. that not both `a` and `b` are equal to the maximum value). Such program could be verified using various abstract domain. For example, when this program is analysed with interval abstract domain, it calculate the following invariant (in a simple version without guards and without single assignment variables):

$$a >= 0 \;\&\&\; a <= 10 \;\&\&\; b >= 0 \;\&\&\; b <= 10 \tag{8.1}$$

This invariant is correct, however it is not sufficient to prove the assertion as according to it, both values can reach up to 10.

The same program can be also analysed in an equality domain in which equalities and disequalities between pairs of variables are found. This domain can find the following invariant:

$$a \neq b \tag{8.2}$$

This invariant is also correct but absolutely cannot verify the assert as it only knows that these two variables are never equal at the end of the loop body.

In this stage, a generic strategy solver for product domain combination can be used to combine these two domains. Making it possible to run this domain combination was as easy as adding a new condition into the 2LS command line arguments parser that when both `-equalities` and `-intervals` switches are used, an instance of this new generic strategy solver is created and it is initialized with equalities and interval domains.

After introducing this small change introduced and analysing the same program in this new domain, the found invariant has the form:

$$a >= 0 \;\&\&\; a <= 10 \;\&\&\; b >= 0 \;\&\&\; b <= 10 \;\&\&\; a! = b \tag{8.3}$$

With this invariant, 2LS could not find any possibility when $a + b$ would be equal to 20 or more and thus correctly verified that this program is always correct.

Algorithm 4: A running example

```
1  #define MAX 10;
2  void main()
3      a = 0;
4      b = 1;
5      while a < MAX && b < MAX do
6          if __VERIFY_nondet_int() then
7              a ++;
8          else
9              b ++ ;
10         end
11         if a == b then
12             if __VERIFY_nondet_int() then
13                 a --;
14             else
15                 b --;
16             end
17         end
18     end
19     assert(a + b < 2 * MAX);
20     return 0;
```

8.2 2LS Regression Tests

In 2LS there is a rather large set of regression tests, that are used to validate that the program behaves the same after new changes were introduced.

This set consists of 9 categories. Each category tests a different type of analysis such as analyses of heap or of termination. In each category there are multiple tests, each consisting out of two files – one containing the source program to be analysed and the second one describing with what arguments 2LS should be run with and what the expected result is.

In total, there are 383 tests but only 268 are really tested, since others are either not yet implemented or they contain a known bug. Since this work reworked a lot of code, it was very important that the results are the same before and after this work. The results confirming this can be seen in Table 8.1.

Category	**Tasks**	**Correct results**	
		Before our work	**After our work**
Non-termination	43	31	**31**
Termination	129	90	**90**
kIkI	36	30	**30**
Preconditions	8	6	**6**
Interprocedural	47	30	**30**
Invariants	86	61	**61**
Heap	19	7	**7**
Heap date	11	9	**9**
Memsafety	4	4	**4**

Table 8.1: A comparison of results of the regression tests

8.3 Running on SV-COMP 2018 tests

The other set of tests that was used for testing, comes from the International Competition on Software Verification. The specific tests come from the 7th year of this competition.

	2LS	
	Before our work	**After our work**
Number of tasks	390	**390**
Correct results	252	**252**
Correct true	172	**172**
Correct false	80	**80**
Incorrect results	7	**7**
Incorrect true	0	**0**
Incorrect false	7	**7**
Inconclusive	131	**131**
Score	312	**312**
CPU time per finished tasks (s)	4.9	**5.1**

Table 8.2: A comparison of results of the SV-COMP 2018 tests

A subset of all SV-COMP tests was selected with focus on covering different tested properties, such as heap, termination, or loops. Comparison of results before and after introduced changes can be seen in Table 8.2.

As seen in this table, results before and after are the same, which means that with these new changes, no capabilities of this tool were broken. The only change is the time needed for the test run, where a 4% increase is seen. This is most likely to the generic concept which results in more function calls.

Chapter 9

Conclusion

In this work, three generic strategy solvers have been proposed. The first one is generic solution for iterative invariant inference for simple domains. The other two are generic strategy solvers for two types of domains combination – product domains and domains with symbolic paths. For making these changes possible, a new abstract domain interface needed to be designed as well.

The proposed domain interface has been implemented and all domains adjusted to comply with it. Then simple domains were migrated into using a generic strategy solver. This migration was executed on all domains except the ones that make use of a binary search while refining its invariants. Also both implemented domain combinations were moved into using the corresponding combination strategy solver.

Although the migration removed a lot of duplicate code and shrunk the code base by 12 files and a few hundred lines of code, the main advantage of this work is the fact that adding new domains or experimenting with new domain combinations becomes much easier.

To prove that it is very simple to experiment with new domain combinations, experiments were conducted. A simple program was found which can be verified only by a domain combination which was not yet implemented in 2LS tool. The only change that was needed to be made was editing how 2LS parses command line arguments and teaching it to accept new argument for using this new domain combination. After that, 2LS could correctly verify the given program. In the future, a more generic parsing of command line arguments could be introduced which would remove even this step.

Bibliography

[1] CPROVER.
Retrieved from: http://www.cprover.org

[2] Alpern, B.; Wegman, M. N.; Zadeck, F. K.: Detecting Equality of Variables in Program. In *Proceedings of the 15th ACM SIGPLAN symposium on Principles of programming languages*. ACM. 1988. pp. 1–11.

[3] Biere, A.; Cimatti, A.; Clarke, E. M.: Bounded model checking. *Advances in Computers*. vol. 58. 2003: pp. 117–148.

[4] Bjørner, N.; Gurfinkel, A.; McMillan, K.; et al.: Horn Clause Solvers for Program Verification. In *Lecture Notes in Computer Science*, vol. 9300. Springer. 2015. pp. 24–51.

[5] 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*, vol. 9291. Springer. 2015. pp. 145–161.

[6] Clarke, E.; Kroening, D.; Lerda, F.: A Tool for Checking ANSI-C Programs. In *Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, vol. 2988. Springer. 2004. pp. 168–176.

[7] Cousot, P.; Cousot, R.: Static determination of dynamic properties of programs. In *Proceedings of the 2nd International Symposium on programming*. Dunod. 1976. pp. 106–130.

[8] Cousot, P.; Cousot, R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM. 1977. pp. 238–252.

[9] Donaldson, A. F.; Haller, L.; Kroening, D.; et al.: Software Verification Using k-Induction. In *Lecture Notes in Computer Science*, vol. 6887. Springer. 2011. pp. 351–368.

[10] Een, N.; Sörensson, N.: Temporal Induction by Incremental SAT Solving. *Electronic Notes in Theoretical Computer Science*. vol. 89. 2003: pp. 543–560.

[11] Hooker, J. N.: Solving the incremental satisfiability problem. *The Journal of Logic Programming*. vol. 15, no. 1&2. 1993: pp. 177–186.

[12] Malík, V.; Hruska, M.; Schrammel, P.; et al.: Template-Based Verification of Heap-Manipulating Programs. In *Proceedings of the 18th Conference on Formal Methods in Computer-Aided Design (FMCAD 2018)*. 2018. pp. 103–111.

[13] Malík, V.: *Abstrakce dynamických datových struktur s využitím šablon*. Master's Thesis. Vysoké učení technické v Brně, Fakulta informačních technologií. 2017. Retrieved from: http://www.fit.vutbr.cz/study/DP/DP.php?id=19901

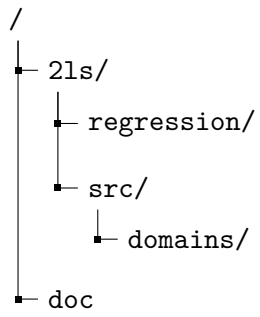
[14] Martiček, Š.: *Syntéza důkazů nekonečnosti běhu programů s využitím šablon*. Master's Thesis. Vysoké učení technické v Brně, Fakulta informačních technologií. 2017. Retrieved from: http://www.fit.vutbr.cz/study/DP/DP.php?id=13436

[15] 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*, vol. 9636. Springer. 2016. pp. 905–907.

Appendix A

Content of the CD

The attached CD contains source codes of 2LS with all changes mentioned in this work. Also on the CD is this text. The main directory structure of the CD is the following:



The main directory contains two subdirectories – `2ls` and `doc`. Subdirectory `doc` contains this text both ad LaTeXsource code and the PDF version.

The `2ls` subdirectory contains all source codes. The structure of this directory is complex. The main part which this work has affected is in `src/domains` subdirectory. This directory contains abstract domains and strategy solvers including all the new ones.