BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

STATIC ANALYSIS AND BUG FINDING IN SOFTWARE
STATICKÁ ANALÝZA A VYHLEDÁVÁNÍ CHYB V PROGRAMECH

PHD THESIS PROPOSAL
TEZE DISERTAČNÍ PRÁCE

AUTHOR Ing. VIKTOR MALÍK
AUTOR PRÁCE

SUPERVISOR prof. Ing. TOMÁŠ VOJNAR, Ph.D.
ŠKOLITEL

BRNO 2018

Abstract

The main research topic of the future PhD thesis described in this report is a formal verification of programs working with dynamically-allocated memory. We develop novel methods for static analysis of reachable shapes of the program heap that can be incorporated into formal verification methods based on SMT solving and invariant inference. Our solution is designed to be easily combined with other verification techniques, which allows us to analyse complex properties of programs. We explore and develop these combinations and also work towards a better scalability of verification of heap-manipulating programs. This work describes the theoretical background and the state-of-the-art of this research, summarizes goals of the PhD thesis, and presents the results achieved so far.

Abstrakt

Hlavnou témou výzkumu v rámci budúcej dizertačnej práce popísanej v tejto správe je formálna verifikácia programov pracujúcich s dynamicky alokovanou pamäťou. Tento projekt predkladá nové metódy statickej analýzy tvaru haldy programu, ktoré môžu byť začlenené do existujúcich metód formálnej verifikácie založených na SMT solvingu a invariantoch programov. Riešenie je navrhnuté tak, aby bolo jednoducho kombinovateľné s inými prístupmi k verifikácii, čo umožňuje analýzu zložitých vlastností programov. V rámci práce tiež skúmame a rozvíjame možnosti takýchto kombinácii a ďalej pracujeme na zlepšení škálovateľnosti verifikácie programov pracujúcich s dynamickou pamäťou. Tento text popisuje teoretické základy na ktorých výzkum stavia, definuje ciele dizertačnej práce a prezentuje doterajšie výsledky.

Keywords

formal verification, static analysis, shape analysis, dynamic data structures, heap analysis, abstract interpretation, k-induction, program invariants, templates of invariants, 2LS, static single assignment form

Klíčová slova

formálna verifikácia, statická analýza, analýza tvaru haldy, dynamické dátové štruktúry, abstraktná interpretácia, k-indukcia, invarianty programu, šablóny invariantov, 2LS, SSA forma

Reference

MALÍK, Viktor. *Static Analysis and Bug Finding in Software.* Brno, 2018. PhD thesis proposal. Brno University of Technology, Faculty of Information Technology. Supervisor prof. Ing. Tomáš Vojnar, Ph.D.

Static Analysis and Bug Finding in Software

Declaration

I declare that I have prepared this dissertation thesis proposal independently, under the supervision of Tomáš Vojnar. Additional information was provided by Peter Schrammel. I listed all of the literary sources and publications that I have used.

. . . .
Viktor Malík
May 31, 2019

Contents

Chapter 1

Introduction

Nowadays, a large amount of new software is produced every day. Assuring its quality is often more difficult and more time consuming than creating the software itself. However, this task is important, especially in mission-critical software, where each bug can have serious consequences.

Traditionally, software quality is assured by extensive testing. Even though testing techniques are currently very advanced, they still lack an important characteristic—they are generally not able to cover all paths through a program and therefore to assure complete correctness of software. On the other hand, methods of formal verification based on static analysis have this property since they inspect programs on the level of the source code. Nonetheless, these are usually complicated and advanced techniques that struggle from many problems and are still subject to extensive research.

Currently, there is a large number of tools for formal verification available, designed to analyse various properties of programs. However, most of the tools are typically very narrowly focused on a single area of analysis. They usually fail to analyse complex properties of real-life programs (e.g. verifying termination of programs using numerical and pointer variables at the same time) while still being able to scale for realistic programs. In order to overcome these limitations, one needs to develop a technique that combines multiple static analysis approaches together, allowing it to explore various different properties of programs. One of the methods that seems promising from this point of view is the *k*I*k*I method proposed in [10]. This method is based on using an SMT solver to compute program invariants and to reason about program properties. The method has been developed by researchers at the University of Oxford and is implemented in a framework for formal verification of C programs called 2LS.

When it comes to analysis of programs written in low-level programming languages (such as C), one of the most challenging tasks for static analysers is analysis of the contents of the program heap. This is because memory on the heap is dynamically allocated, it has an unbounded size, and typically it is linked through pointers into complex graphs, which makes it difficult to be analysed without executing the program. The task of analysing the shape (and subsequently the properties) of the heap graphs is usually referred to as shape analysis since its goal is to discover reachable shapes of dynamic data structures (such as linked lists or trees) on the heap of the analysed program. Practical programs very often work with dynamically allocated memory and hence supporting shape analysis is essential for tools that are intended to be used on real-life software. As for the verification method implemented in the 2LS framework, possibilities of integrating shape analysis into it have not yet been explored.

With respect to the above, the main research topic of my PhD thesis is integration of shape analysis into the specific verification method used by the 2LS framework. It very much differs from what is common in other tools that perform shape analysis. These are usually based on some form of abstract interpretation that symbolically executes a given program and uses widening/abstraction to avoid generation of infinitely many reachable program configurations. On the other hand, the method in [10] differs in multiple aspects. First, it represents programs by logical formulae, which allows it to use an SMT solver to reason about program properties. Second, its computational loop combines multiple analysis techniques, such as k-induction and a rather specific form of abstract interpretation, with a notion of program invariants based on so-called templates. Integrating shape analysis into this framework requires a specific solution.

This obstacle, however, brings an important advantage. The created shape analysis needs to have a form similar to other analyses already present in the framework (or to any analysis that will be integrated in the future). Thanks to this requirement, it is straightforward to compose various analyses together, which can enable reasoning about interesting program properties that other, single-purpose methods are not able to handle well. From the point of view of shape analysis, a particularly interesting one is its combination with analysis of numerical values in the program, which could bring possibilities of analysing properties of the program heap such as contents of dynamically-allocated structures, lengths of lists, heights of trees, or offsets of pointers. Exploring and developing these combinations, that could help overcoming limits of the current shape analysers, is an essential part of my research.

Furthermore, the 2LS framework supports interprocedural analysis that allows one to analyse each function of a program separately and hence to reduce the complexity of the overall verification. This is particularly important for the method to scale on realistic programs that often contain thousands or millions of lines of code. A combination of shape analysis with interprocedural analysis is yet another interesting problem that will be a target of the research within the PhD thesis described by this report. Analysing functions separately is particularly difficult when dealing with heap-manipulating programs since one needs to model an existing state of the heap at the beginning of each function.

This work is organised as follows. First, in Chapter 2, we introduce the approach to formal verification of programs based on invariant inference using templates that we intend to build on. Then, in Chapter 3, we give an overview of existing shape analysis methods since shape analysis is the main topic of my PhD thesis. Goals of the thesis are formulated in detail in Chapter 4. We have already proposed and published an initial solution to integration of shape analysis into the underlying framework and we introduce it in Chapter 5. The chapter also contains a description of the performed experiments whose results show that our analysis can be combined with analysis of other program properties, which allows 2LS to reason about, e.g., contents of dynamic data structures on the program heap.

Chapter 2

Program Verification using Template-Based Invariant Inference

In this chapter, we describe an automatic program verification framework called 2LS that combines multiple analysis approaches together: namely abstract interpretation, bounded model checking, and k-induction. The method approximates the program semantics using logical formulae which allows it to use an SMT solver to reason about properties of programs. In order to approximate sets of all reachable program states, a well-known concept of inductive invariants is used. To infer such invariants, 2LS introduces a novel algorithm reducing the problem from the second-order logic to the first-order logic. Inductive invariants can be used to describe invariants of loops and of functions in the analysed program and subsequently to prove and to refute program properties.

The rest of this chapter is organised as follows. Section 2.1 presents basic principles of the verification techniques used. Then, the core algorithm combining these techniques is outlined in Section 2.2. After that, we describe in detail the most important parts of the framework: a rather specific internal program representation based on the static single assignment form (Section 2.3) and an algorithm for inference of inductive invariants (Section 2.4).

2.1 Overview of the Used Techniques

The program verification approach that underlies 2LS is based on effectively combining three different static analysis techniques together. These are namely *abstract interpretation*, *bounded model checking (BMC)*, and *k-induction*. They are all based on some form of approximation of the set of all program states reachable by the analysed program. Two basic forms of approximation are common, namely *over-* and *under-approximation*.

Over-approximating the set of all reachable program states allows the analysis to soundly verify validity of program properties—if a property holds for a superset of all reachable states, it must hold for all states of the program as well. On the other hand, under-approximating reachable program configurations is typically used to discover violations of program properties—if a property is violated in a subset of all reachable program states, it must be violated in some reachable state, too.

All of the mentioned techniques are well-established in program verification. However, 2LS uses rather specific forms of these methods, especially of the abstract interpretation. This is caused by the fact that the analysed program is viewed as a logical formula and an automatic SMT solver is used to compute program invariants and to reason about program properties.

In this section, we formally define the standard approach to these techniques. To ease formalisation, we view the analysed program as a transition system. We show how concrete semantics of a program is defined within this representation and how individual techniques approximate this semantics.

2.1.1 Concrete Semantics of Programs

The *concrete semantics* of a program formalises the set of all possible behaviours of the program. For transition systems, this can be defined as a set of all states that the program can get into during its execution [16].

A *program state* $\boldsymbol{x}$ is the current value of all program variables (including the program counter) and related memory (i.e. the stack and the heap). Let S be a set of program states, and let the *transition relation* $\tau \subseteq S \times S$ define for each state a set of its possible successors in the program execution.

Assume a sequence of sets of states $S_0 S_1 \dots S_k$ such that $\forall 0 \leq i < k : (S_i, S_{i+1}) \in \tau$. We denote $S_k = \tau^k(S_0)$ the set of states *reachable from* S_0 *after* k *execution steps*. If I is the set of all possible initial states of a program, then the set of *all reachable states* S_r is the least fixed point of τ starting from I defined as:

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

With respect to the above, S_r defines the concrete semantics of the analysed program.

2.1.2 Programs As Logical Formulae

Since it is generally very hard to automatically analyse C programs directly, they are usually translated into a so-called *internal representation*. This is typically a simpler form of the program preserving the original concrete semantics that is easier to be used for further analysis. A very common representation are e.g. *control flow graphs (CFG)* used by the two most spread C compilers `gcc` [23] and `clang` [35].

However, another form of program representation that is recently getting more and more popular in fields of program analysis and verification are *logical formulae*. The reason for this is that there exist strong automatic SAT (e.g. [21, 6]) and SMT (e.g. [39]) solvers and it is very advantageous to use them to automatically reason about programs. This kind of representation is also used in the verification framework 2LS. Thus, we adapt the formalisation from the previous section to use logical formulae.

A state of a program is described by a logical interpretation of logical variables corresponding to each program variable. A set of states can be described using a formula—states in the set are defined by models of the formula. Given a vector of variables $\boldsymbol{x}$, a predicate $\mathit{Init}(\boldsymbol{x})$ is the predicate describing the initial states. A transition relation is described as a formula $\mathit{Trans}(\boldsymbol{x}, \boldsymbol{x}')$. From these, it is possible to determine the set of reachable states as the least fixed-point of the transition relation starting from the states described by $\mathit{Init}(\boldsymbol{x})$ as already shown in Formula 2.1. This is, however, difficult to compute, so instead an *inductive invariant* is used. A predicate *Inv* is an inductive invariant if it has the property:

$$\forall \boldsymbol{x}, \boldsymbol{x}'.(\mathit{Inv}(\boldsymbol{x}) \land \mathit{Trans}(\boldsymbol{x}, \boldsymbol{x}') \implies \mathit{Inv}(\boldsymbol{x}')). \tag{2.2}$$

An inductive invariant defined as above is a description of a fixed-point of the transition relation. However, it is not guaranteed to be the least one, nor to include $\mathit{Init}(\boldsymbol{x})$. Moreover, there are predicates which are inductive invariants, but are not sufficient to be used for proving any properties of the source program (such as the predicate *true*, which describes the complete state space) [10]. That is why it is useful to compute such invariants that approach the least fixed-point, so that it is enough to use them to check a given property.

A verification task does often require showing that the set of all reachable states does not intersect with the set of error states denoted $\mathit{Err}(\boldsymbol{x})$. Using the concept of inductive invariants and existential second-order quantification ($\exists_2$), we can formalise it as:

$$\begin{aligned}\exists_2 \mathit{Inv}.\ \forall \boldsymbol{x}, \boldsymbol{x}'.\ &(\mathit{Init}(\boldsymbol{x}) \implies \mathit{Inv}(\boldsymbol{x})) \land \\ &(\mathit{Inv}(\boldsymbol{x}) \land \mathit{Trans}(\boldsymbol{x}, \boldsymbol{x}') \implies \mathit{Inv}(\boldsymbol{x}')) \land \\ &(\mathit{Inv}(\boldsymbol{x}) \Rightarrow \neg \mathit{Err}(\boldsymbol{x})).\end{aligned} \tag{2.3}$$

2.1.3 Abstract Interpretation

Abstract interpretation is a static analysis technique that soundly approximates the concrete semantics of programs using a so-called *abstract semantics.* Generally, the set of all reachable states is not computable. However, since it is usually needed to reason about a certain program property only, to prove this property it is sufficient to approximate program states as elements of a simpler domain, called the *abstract domain.*

Having the *concrete domain* P of program states, we create an abstract domain Q. An element of the abstract domain, called an *abstract value*, corresponds to an element from the concrete domain, which is typically a set of concrete program states. Along with the abstract domain, we define two functions [17]:

- The *concretisation function* defines a mapping from an abstract value to a value of the concrete domain. Formally $\gamma : Q \to P$ and $\gamma(q)$ is a concrete value represented by q.
- The *abstraction function* defines a mapping from a concrete value to an abstract value from the abstract domain. Formally $\alpha : P \to Q$ and $\alpha(p)$ is the most precise abstract value in Q whose concretisation contains p.

An abstract interpretation I of a program is then a tuple [18]:

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

where

- Q is the abstract domain (along with well-defined abstraction and concretisation functions),
- $\top \in Q$ is the supremum of Q,
- $\bot \in Q$ is the infimum of Q,
- $\sqcup : Q \times Q \to Q$ is the *join operator*, $(Q, \sqcup, \top)$ is a complete semilattice,

- $(\sqsubseteq) \subseteq Q \times Q$ is an ordering on $(Q, \sqcup, \top)$ defined as $x \sqsubseteq y \Leftrightarrow x \sqcup y = y$,
- $\tau^{\#} : \mathit{Instr} \times Q \to Q$ defines the interpretation of *abstract transformers*.

The framework of abstract interpretation allows to approximate the original program semantics by computing the fixpoint of $\tau^{\#}$ in the abstract domain. The result is one abstract value (i.e. one abstract state) for each execution point of the source program. In case multiple abstract values are obtained (because of multiple execution paths entering the program location), these are accumulated into one using the join operator. The properties of the analysed program are then checked in the computed abstract values.

In order to be sound in proving program properties, an abstract value must describe at least (but not precisely) all concrete states that are reachable in the given program location. This property is ensured using a *Galois connection* between the concrete and abstract domains. We say that $(P, \leq, Q, \sqsubseteq)$ is a Galois connection if and only if $(P, \leq)$ and $(Q, \sqsubseteq)$ are partially ordered sets, and there is the following relation between abstraction and concretisation functions [18]:

$$\begin{aligned} &\forall p \in P, q \in Q : \\ &\qquad p \leq \gamma(q) \Leftrightarrow \alpha(p) \sqsubseteq q. \end{aligned} \tag{2.5}$$

Since the computed abstract value is an over-approximation of the set of all reachable concrete program states, abstract interpretation may generate a *false positive*. It is a situation when a property does not hold for the computed abstract semantics, but it holds for the set of all reachable program states. This incoherence is usually caused by the fact that an abstract value represents multiple concrete program states and may represent also states that are not reachable in the real program.

The analysis approach of 2LS uses inductive invariants instead of directly computing the fixed point of the transition relation. Abstract interpretation is used here in such way that the inductive invariant is computed in a chosen abstract domain. Hence, it describes a property that holds for a superset of reachable program states. Such property (and any other property logically implied by it) thus holds for the set of all reachable program states, too.

2.1.4 Bounded Model Checking

Bounded Model Checking (BMC) [7] is a static analysis technique that is in a way complementary to abstract interpretation. While the latter is based on an over-approximation to soundly prove program properties, BMC relies on under-approximation to find (mainly) property violations.

BMC is based on checking only program paths whose length is bounded by a certain integer k. To this end, it uses an *unwinding (unfolding) of the transition relation.* For a constant $k \in \mathbb{N}$, we introduce the k-th unwinding $T[k]$ representing k steps of the transition transition relation:

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

Using the unwound transition relation as defined in Formula 2.6, it is possible to under-approximate the concrete program semantics by $\mathit{Init}(\boldsymbol{x}_0) \wedge T[k]$. This formula describes a subset of all reachable program states since only prefixes of program paths are considered. This makes BMC useful for finding property violations (i.e. bugs) in the original program.

To formalise the BMC problem, we first introduce a predicate $P[k]$ in Formula 2.7 describing k states being error-free.

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

Using $T[k]$ and $P[k]$, we formally define Bounded Model Checking as a problem of picking a bound k and solving Formula 2.8 [10].

$$\exists \boldsymbol{x}_0, \ldots, \boldsymbol{x}_k . \mathit{Init}(x_0) \wedge T[k] \wedge \neg P[k+1] \tag{2.8}$$

If the formula is satisfiable, a property is violated in some state reachable after at most k steps. Moreover, the model of the formula represents a concrete counterexample (a witness to the property violation). On the other hand, an unsatisfiability of the formula does not necessarily imply that the property always holds since it might be violated after more than k steps. This means that BMC is generally unsound and it may produce so-called *false negatives*. It is a situation complementary to the false positive described in the previous section—a property holds for the k-th unwinding, but it does not hold for some l-th unwinding with $l > k$.

Incremental Bounded Model Checking

One of the main limitations of BMC is a need to specify an unwinding bound. This is often avoided by using a technique called *incremental bounded model checking* [24]. It uses repeated BMC checks with bound starting at 0 and being increased in a linear manner. In each step, this allows to assume that there are no errors in previous states and hence simplifies the problem to solving Formula 2.9.

$$\exists \boldsymbol{x}_0, \ldots, \boldsymbol{x}_k . \mathit{Init}(x_0) \wedge T[k] \wedge P[k] \wedge \neg \mathit{Err}(\boldsymbol{x}_k) \tag{2.9}$$

2.1.5 k-Induction

The *k-induction* technique [46] can be viewed as an extension of incremental Bounded Model Checking that is capable of proving program properties. Similarly to BMC, it uses unwinding of the transition relation which allows it to find witnesses of property violations. This approach is extended by computing a so-called *k-inductive invariant*. It is a generalisation of the concept of inductive invariants introduced in Section 2.1.2. A k-inductive invariant KInv is a predicate having the property:

$$\forall \boldsymbol{x}_0, \ldots, \boldsymbol{x}_k . K[k] \wedge T[k] \Rightarrow \mathit{KInv}(\boldsymbol{x}_k) \tag{2.10}$$

where

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

Similarly to the inductive invariant described in Section 2.1.2, if a k-inductive invariant is implied by the k-th unwinding of the transition relation starting from the initial state, it represents a fixed-point of the transition relation. Therefore, it can be used to soundly verify a program property. Formally, a system is safe if and only if there is a k-inductive invariant *KInv* that satisfies the property:

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

Moreover, k-inductive invariants have the following properties:

- Every inductive invariant is a 1-inductive invariant.
- Every k-inductive invariant is a $(k+1)$-inductive invariant.
- Showing that k-inductive invariant exists implies that an inductive invariant exists.
- k-inductive invariant is not necessarily an inductive invariant, usually a corresponding inductive invariant is much more complex.

Generally, finding a k-inductive invariant is simpler than finding an inductive invariant. However, it increases the complexity of the formula being checked (since unwinding of the transition relation significantly increases the formula size) [10]. In addition, it still remains to be a hard problem and hence it is useful to combine k-induction with abstract interpretation by computing k-inductive invariants in some abstract domain. Such combination is implemented by the kIkI algorithm described in the following section.

2.2 kIkI Algorithm

In Section 2.1, we presented three common techniques that are widely used for formal analysis and verification of programs. Each of the approaches is suitable for a different purpose and also has different limitations. Generally, advantages and disadvantages of individual techniques can be summarized as follows:

Bounded Model Checking is suitable for finding violations of program properties, while providing counterexamples. However, only a small part of true properties can be proven since programs are explored up to a given bound only.

k-induction is capable of proving true properties as well as providing counterexamples for a part of property violations. However, it requires k-inductive invariants which are typically expensive to be computed.

Abstract Interpretation is designed to prove true properties of programs by computing over-approximative invariants in some abstract domain. However it suffers from a large number of false positives that cannot be distinguished from real violations of properties.

In order to make use of each technique's strengths and overcome their limitations in the same time, 2LS combines them together in a special algorithm called kIkI (k-Invariants and k-Induction). The basic structure of the algorithm is illustrated by Figure 2.1 [10].

Initially, k is set to 1. At the beginning, initial program states are checked whether they contain errors. Then, kIkI computes a k-invariant (*KInv*) in some chosen abstract domain (i.e. using abstract interpretation). The computed invariant is checked whether it is sufficient to prove safety. In case a violation of the property is found, BMC is used to check if the error state is truly reachable within k steps in the original program. In case it is not reachable, the counterexample can be spurious and hence the procedure is repeated with an increased k. In each iteration, kIkI adds assumptions that the checked property holds for all previous states (this is expressed by the predicate $P[k]$).

Even though kIkI eliminates most of the limitations of the three approaches, there still remains a need to specify some maximal k, otherwise it might not terminate. In case the maximal k is reached and the property was neither proved nor refuted, the algorithm ends with an inconclusive result.

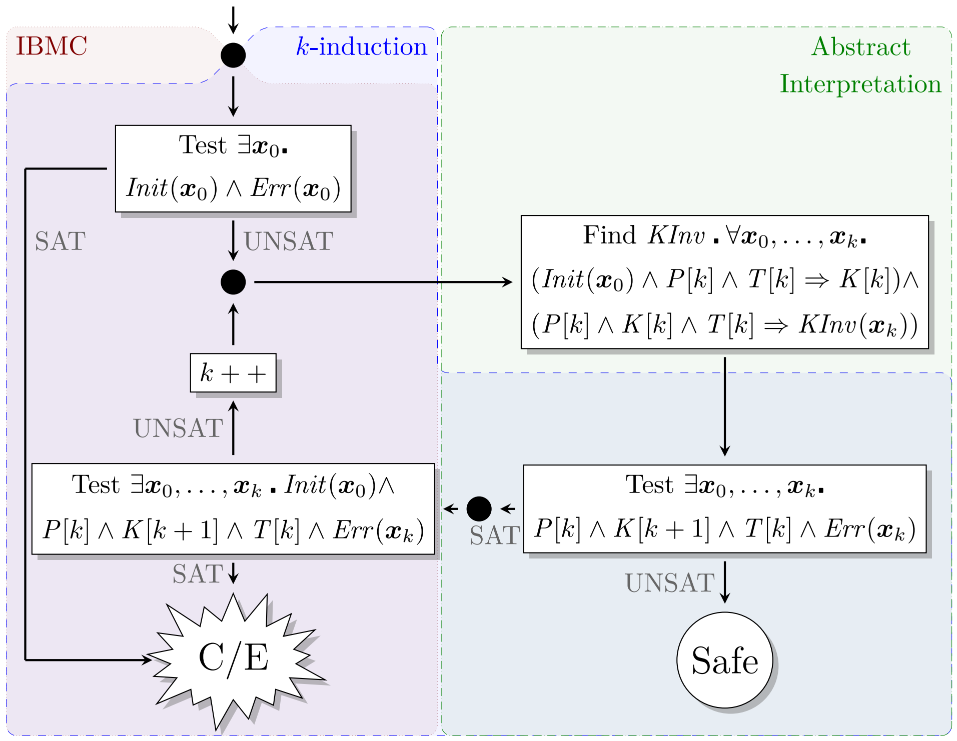

IBMC
k-induction
Abstract Interpretation
Test $\exists \boldsymbol{x}_0 .$ $\mathit{Init}(\boldsymbol{x}_0) \wedge \mathit{Err}(\boldsymbol{x}_0)$
SAT
UNSAT
Find $\mathit{KInv} \, . \, \forall \boldsymbol{x}_0, \ldots, \boldsymbol{x}_k .$ $(\mathit{Init}(\boldsymbol{x}_0) \wedge P[k] \wedge T[k] \Rightarrow K[k]) \wedge$ $(P[k] \wedge K[k] \wedge T[k] \Rightarrow \mathit{KInv}(\boldsymbol{x}_k))$
$k++$
UNSAT
Test $\exists \boldsymbol{x}_0, \ldots, \boldsymbol{x}_k \, . \, \mathit{Init}(\boldsymbol{x}_0) \wedge$ $P[k] \wedge K[k+1] \wedge T[k] \wedge \mathit{Err}(\boldsymbol{x}_k)$
SAT
Test $\exists \boldsymbol{x}_0, \ldots, \boldsymbol{x}_k .$ $P[k] \wedge K[k+1] \wedge T[k] \wedge \mathit{Err}(\boldsymbol{x}_k)$
SAT
UNSAT
C/E
Safe

Figure 2.1: The kIkI algorithm [10]

2.2.1 Incremental Solving

In order for kIkI to be efficient, it is based on a so-called *incremental solving* [28]. This technique aims at checking whether satisfiability of a problem is preserved when the clause set is incremented with new clauses. Instead of re-solving the whole problem, the information from the original problem is used to speed up the solution of the new one. The original problem (before adding the clauses) is thus considered satisfiable, and only the impact of the new clauses is checked.

In kIkI , this concept is used as follows. Instead of giving the whole $P[k]$ to the solver in each iteration of the algorithm, incremental solving allows to give only $\neg \mathit{Err}(\boldsymbol{x}_{k-1})$, since $P[k-1]$ is already in the clause set of the solver from the previous iteration and $P[k] = P[k-1] \wedge \neg \mathit{Err}(\boldsymbol{x}_{k-1})$. Analogously, $\mathit{Trans}(\boldsymbol{x}_{k-1}, \boldsymbol{x}_k)$ can be given instead of the whole $T[k]$ in each iteration.

2.3 Representation of Programs Using Logical Formulae

In order to seamlessly combine various analysis techniques together, 2LS uses logical formulae to represent analysed programs. In this section, we show how a program can be translated into a quantifier-free first-order formula that over-approximates its concrete semantics. The transformation is based on *static single assignment* (*SSA*) form. It is a well-known concept of an intermediary program representation that is usable in combination with an automatic solver. We first define the general principles of the SSA form in Section 2.3.1 and then we present specific modifications adopted by 2LS in Section 2.3.2. At the end, in Section 2.3.3, we give an example of a translation of a program into the SSA form.

2.3.1 Single Static Assignment Form

Generally, single static assignment form [2] is an intermediary program representation satisfying the property that each variable is assigned at most once. A translation into the SSA form thus involves separating each variable v into several variables v_i. Every time some program location i of the original program contains an assignment to v, it is replaced by an assignment to v_i. Every R-value usage of v (i.e. every occurrence of v at the right-hand side of an assignment or in a condition) is replaced by the appropriate variable v_j where j is the last program location in which v was assigned before the given use.

In order to fulfil the single assignment property, it is required that for each program location j and each variable v, there is a unique v_i such that there are no assignments to v between i and j. This is achieved by introducing additional assignments at join points of the translated program. These are called Φ (*phi*) *nodes* and have a form of an assignment $x = \Phi(y, z)$. This expression means that x is assigned the value of y if the control reaches this program location via the first entering edge, and x is assigned the value of z if the control reaches the node via the second entering edge. For simplicity, we may assume that each join point joins exactly two program branches.

The logical formula corresponding to the original program is then a conjunction of SSA formulae for all program statements. For an acyclic code, this formula represents exactly the *strongest post condition* of running the code. In 2LS, the standard SSA form is made acyclic even for programs containing loops by over-approximating their effect. Thanks to this, the corresponding logical formula implicitly encodes the transition relation $\mathit{Trans}(\boldsymbol{x}, \boldsymbol{x}')$. Moreover, when the formula is used in the abstract interpretation procedure, it removes the need to explicitly define abstract transformers. Details of the loop approximation are described in the following section.

2.3.2 SSA Form Used in 2LS

The SSA form used in 2LS extends the general concepts introduced in the previous section. The most important extensions are a specific encoding of control-flow information, and an over-approximation of loops and function calls.

Encoding of Control-Flow Information

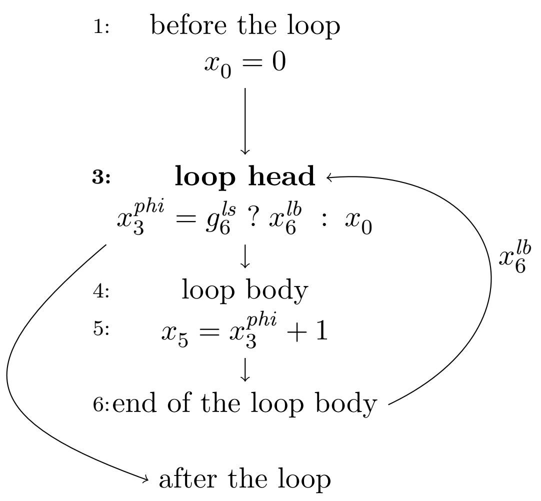
A logical formula obtained from the standard SSA form, as described above, implicitly encodes the data-flow among variables by enhancing the single assignment property. However, control-flow is lost after the transformation, and hence it must be encoded explicitly. To this end, special variables called *guards* are introduced. In particular, for each program location i, a Boolean variable g_i is introduced, and its value encodes whether the program location is reachable.

Over-Approximation of Loops

In order to be able to use the generated formula in an SMT solver, the SSA form used by 2LS is made acyclic by cutting loops at the end of the loop body. Then, the value of each variable x at the loop head is represented using a Φ *variable* x^{phi} whose value is defined by a non-deterministic choice between the value coming from before the loop, and the value coming from the end of the loop. The latter value is represented by a newly introduced unconstrained *loop-back* variable x^{lb}. An example of this conversion is given in Figure 2.2.

```
1 unsigned x = 0;
2
3 while (x < 10)
4 {
5     ++x;
6 }
```

(a) A loop in C



(b) Encoding of the loop in the SSA form

Figure 2.2: Conversion of loops in the SSA form used in 2LS

The loop has been cut at the end of its body: instead of passing the version of x from the end of the loop body (x_5) back to the Φ node in the loop head, a free "loop-back" variable x_6^{lb} is passed. This way, the SSA form remains acyclic. The choice of the value of x in the Φ node is made non-deterministically using the free Boolean "loop-select" variable g_6^{ls}.

Since x_6^{lb} and g_6^{ls} are free variables, this representation is an over-approximation of the actual program traces. The precision can be improved by constraining the value of x_6^{lb} by means of a *loop invariant*, which is then inferred during the analysis. A loop invariant for the variable x describes a property that holds for x at the end of the loop body, after each iteration of the loop. In the program in Figure 2.2, it describes a property that holds for x_5 and hence can be assumed to hold for x_6^{lb} as well.

For example, a common property that is computed when dealing with numerical variables are the lower and the upper bound of their value. For the given example, such invariant for x_6^{lb} could be:

$$x_6^{lb} \geq 1 \land x_6^{lb} \leq 10. \tag{2.13}$$

Function Inputs and Outputs

When dealing with real-world programs, it is usually essential to perform inter-procedural analysis (i.e. to analyse each function of the program separately). To facilitate such analysis, 2LS introduces special sets of variables in the SSA form of each function f. These are in particular:

- ${\boldsymbol{x}^{in}}_f$ denoting the set of *input parameters* of the function. The set includes all variables entering f from the caller function (i.e. formal parameters of the function and global variables[1]). When analysing f separately, their value is unconstrained.
- ${\boldsymbol{x}^{out}}_f$ denoting the set of *output parameters* of the function. The set includes all SSA variables that are changed within f and that can be used by the caller function (i.e. the return value and the changed global variables).

Over-Approximation of Function Calls

To handle inter-procedural analysis correctly, function calls are over-approximated in the SSA form used in 2LS. An invocation of a function f in a program location i is replaced by a *function placeholder* predicate $f_i({\boldsymbol{x}^{a\_in}}_i, {\boldsymbol{x}^{a\_out}}_i)$ where ${\boldsymbol{x}^{a\_in}}_i$ and ${\boldsymbol{x}^{a\_out}}_i$ are sets of input and output arguments of the call, respectively. These are analogous to ${\boldsymbol{x}^{in}}_f$ and ${\boldsymbol{x}^{out}}_f$ described in the previous section and describe same variables from the view of the caller function:

- ${\boldsymbol{x}^{a\_in}}_i$ denotes the set of *input arguments* of the call. The set includes all SSA variables that are passed to the function (i.e. the actual arguments of the call and the SSA instances of global variables in the program location i).
- ${\boldsymbol{x}^{a\_out}}_i$ denotes the set of *output arguments* of the call. The set includes fresh SSA variables that represent values changed by the called function. Their value is initially unconstrained and thus the function placeholder represents an over-approximation of the actual function effect. Its value can be later constrained with means of an invariant which is usually denoted as a *function summary.*

2.3.3 Example of a Program Transformation into the SSA Form

To better understand conversion of a C program, we give an example in Figure 2.3. Line 2 is the entry location of the program. It is always reachable, therefore the guard g_2 is set to *true*. The definition of x is done at line 3. The head of the loop contains a Φ node (line 6) and since it is directly reachable from the beginning of the `main` function, its guard g_5 is the same as the guard of the entry point (g_2). The guard g_7 at line 7 expresses that the loop body is only reachable if the loop head is reachable (g_5) and if the loop condition is true ($x_6^{phi} < 10$). line 8 sets the new value of x. The guard g_{10} at line 10 captures the fact that the location after the loop is reachable when the loop condition is false. Finally, line 11 requires x to be equal to 10 once the assertion is reachable (i.e. once g_{10} is true).

[1]Currently, 2LS does not support dealing with dynamically allocated memory, which is why we do not consider objects on the heap.

```c
void main()
{
   unsigned x = 0;


   while (x < 10)
   {
      ++x;
   }

   assert(x == 10);
}
```

(a) The C program

$$
\begin{aligned}
&g_2 = \mathit{true}\\
&x_3 = 0\\
&g_5 = g_2\\
&x_6^{phi} = (g_9^{ls}\,?\,x_9^{lb} : x_3)\\
&g_7 = (x_6^{phi} < 10) \wedge g_5\\
&x_8 = 1 + x_6^{phi}\\
&g_{10} = \neg(x_6^{phi} < 10) \wedge g_5\\
&x_6^{phi} = 10 \vee \neg g_{10}
\end{aligned}
$$

(b) The corresponding SSA

Figure 2.3: Conversion from a C program to SSA

2.4 Template-Based Program Verification

The most important phase of the presented k*I*k*I* algorithm is inference of a k-inductive invariant *KInv* using the abstract interpretation approach. This problem, which can be expressed in (existential fragment) of second-order logic, is reduced to a problem expressible in quantifier-free first-order logic using so-called *templates*. This reduction enables 2LS to use an SMT solver for automated inference of *loop invariants* and *function summaries*. These are then used to check various properties of the analysed program. The whole concept is focused on finite state systems since 2LS uses bit-vectors to analyse software [10].

2.4.1 Invariant Inference via Templates

In order to exploit the power of the kIkI algorithm, 2LS uses a solver-based approach to computing inductive invariants. Formally, search for a 1-inductive invariant is expressed by Formula 2.3. This is extended to a k-inductive invariant in Formula 2.12. For simplification purposes, the following explanation will refer to 1-inductive invariants, however, the presented concepts can be easily applied to k-inductive invariants as well, by using $T[k]$ instead of *Trans* and $K[k]$ instead of *Inv*.

To directly handle Formula 2.3 by a solver, 2LS would need to handle second-order logic quantification. Since a suitably general and efficient second order solver is not currently available, the problem is reduced to one that can be solved by an iterative application of a first-order solver. This reduction is done by restricting the form of the inductive invariant *Inv* to $\mathcal{T}(\boldsymbol{x}, \boldsymbol{\delta})$ where $\mathcal{T}$ is a fixed expression (a so-called *template*) over program variables $\boldsymbol{x}$ and template parameters $\boldsymbol{\delta}$. This restriction corresponds to the choice of an abstract domain in abstract interpretation—a template only captures the properties of the program state space that are relevant for the analysis. This reduces the second-order search for an invariant to a first-order search for the template parameters:

$$
\begin{aligned}
\exists \boldsymbol{\delta}.\ \forall \boldsymbol{x}, \boldsymbol{x}'.\ (\mathit{Init}(\boldsymbol{x}) \implies \mathcal{T}(\boldsymbol{x}, \boldsymbol{\delta})) \wedge\\
(\mathcal{T}(\boldsymbol{x}, \boldsymbol{\delta}) \wedge \mathit{Trans}(\boldsymbol{x}, \boldsymbol{x}') \implies \mathcal{T}(\boldsymbol{x}', \boldsymbol{\delta})).
\end{aligned}
\tag{2.14}
$$

Although the problem is now expressible in first-order logic, the formula contains quantifier alternation, which poses a problem for current SMT solvers. This is solved by iteratively checking the negated formula (to turn $\forall$ into $\exists$) for different choices of constants $\boldsymbol{d}$ as candidates for template parameters $\boldsymbol{\delta}$ (to remove $\exists\boldsymbol{\delta}$). For a value $\boldsymbol{d}$, the template formula $\mathcal{T}(\boldsymbol{x}, \boldsymbol{d})$ is an invariant if and only if Formula 2.15 is unsatisfiable.

$$\begin{aligned} \exists \boldsymbol{x}, \boldsymbol{x}'.\ &\neg(\mathit{Init}(\boldsymbol{x}) \implies \mathcal{T}(\boldsymbol{x}, \boldsymbol{d})) \lor \\ &\neg(\mathcal{T}(\boldsymbol{x}, \boldsymbol{d}) \land \mathit{Trans}(\boldsymbol{x}, \boldsymbol{x}') \implies \mathcal{T}(\boldsymbol{x}', \boldsymbol{d})) \end{aligned} \tag{2.15}$$

From the abstract interpretation point of view, $\boldsymbol{d}$ is an abstract value, i.e. it represents (*concretises to*) the set of all program states $\boldsymbol{x}$ that satisfy the formula $\mathcal{T}(\boldsymbol{x}, \boldsymbol{d})$. The abstract values representing the infimum $\bot$ and supremum $\top$ of the abstract domain denote the empty set and the whole state space, respectively: $\mathcal{T}(\boldsymbol{x}, \bot) \equiv \mathit{false}$ and $\mathcal{T}(\boldsymbol{x}, \top) \equiv \mathit{true}$ [10].

Formally, the concretisation function γ is same for each abstract domain:

$$\gamma(\boldsymbol{d}) = \{\boldsymbol{x} \mid \mathcal{T}(\boldsymbol{x}, \boldsymbol{d}) \equiv \mathit{true}\}. \tag{2.16}$$

As for the abstraction function, it is essential to find the most precise abstract value representing a concrete program state. Thus:

$$\alpha(\boldsymbol{x}) = \min(\boldsymbol{d}) \text{ such that } \mathcal{T}(\boldsymbol{x}, \boldsymbol{d}) \equiv \mathit{true}. \tag{2.17}$$

Since the abstract domain forms a partially ordered set with ordering $\sqsubseteq$ and $\mathcal{T}(\boldsymbol{x}, \top) \equiv \mathit{true}$, existence of such a minimal value $\boldsymbol{d}$ is guaranteed.

The algorithm for the invariant inference takes an initial value of $\boldsymbol{d} = \bot$ and iteratively solves the below quantifier-free formula (corresponding to the second disjunct in Formula 2.15) using an SMT solver:

$$\mathcal{T}(\boldsymbol{x}, \boldsymbol{d}) \land \mathit{Trans}(\boldsymbol{x}, \boldsymbol{x}') \land \neg(\mathcal{T}(\boldsymbol{x}', \boldsymbol{d})). \tag{2.18}$$

If the formula is unsatisfiable, then an invariant has been found, otherwise the model of satisfiability is returned by the solver. The model represents a counterexample to the current instance of the template being an invariant. The value of the template parameter $\boldsymbol{d}$ is though refined by joining with the obtained model of satisfiability using the domain-specific join operator [10].

Incremental Solving Similarly to the kIkI algorithm itself, invariant inference makes use of the incremental solving technique, described in Section 2.2.1. Here, since $\mathit{Trans}(\boldsymbol{x}, \boldsymbol{x}')$ does not change, it is sufficient to provide current instances of the template formula (i.e. $\mathcal{T}(\boldsymbol{x}, \boldsymbol{d}) \land \neg\mathcal{T}(\boldsymbol{x}', \boldsymbol{d})$) in each iteration of solving of Formula 2.18.

2.4.2 Guarded Templates

Traditionally, analyses based on abstract interpretation use forms of control-flow graphs (CFG) to represent the analysed programs. In a CFG, a computed invariant can be directly bound to certain program state in which it is computed and for which it holds. Since 2LS uses logical formulae generated from the SSA form, invariants cannot be used directly. Instead, so-called *guarded templates* are used.

A guarded template has a form $G \Rightarrow \mathcal{T}(\boldsymbol{x}, \boldsymbol{d})$ where G is a conjunction of SSA guards that are associated with the definition of variables whose values the invariant constrains. This guarantees that the invariant can be applied for some program run if and only if the variables that it describes are defined in the given run.

In case an invariant describes multiple different variables defined in different program locations, it can be split into multiple parts where each part has its own guard defined.

2.4.3 Loop Invariants

Loop invariants are used to constrain values of loop-back variables defined in Section 2.3.2. These variables represent abstractions of values of program variables returning from the end of the loop body to the loop head. Hence, a loop invariant must describe a property of a variable that holds at the end of the loop body, after each iteration of the loop.

Formally, let L be the set of all loops in a program and let $\boldsymbol{x}_l$ be the set of all loop-back variables of some loop $l \in L$. A loop invariant Inv^l is a projection of an inductive invariant Inv (describing the set of all states reachable in a program) onto the set of variables $\boldsymbol{x}_l$. The loop invariant is expressed in the form of a guarded template. A guarded template for Inv^l has the form:

$$(g_{lh} \wedge g_{lh}^{ls}) \Rightarrow \mathcal{T}(\boldsymbol{x}_l, \boldsymbol{\delta}) \tag{2.19}$$

where lh is the program location of the head of the loop l. The guard g_{lh} expresses that l is reachable from the beginning of the program and the guard g_{lh}^{ls} is a free loop-select variable driving the choice between values of variables coming from the loop head and from the end of the loop body (for details see Section 2.3.2). If $(g_{lh} \wedge g_{lh}^{ls})$ is equal to *true*, the loop l is reached, and the loop-back variables of l are defined and hence the loop invariant Inv^l constraining their value can be used.

Example

We now illustrate the procedure of computing a loop invariant of the program given in Figure 2.3. Here, the set of all loop-back variables $\boldsymbol{x}_l = \{x_9^{lb}\}$. We use the template polyhedra domain [44], particularly its subclass for the *interval abstract domain* [17]. Using this domain, we compute for each variable x an interval in which all possible values of x lie. Hence, the template has the form:

$$\mathcal{T}(\{x_9^{lb}\}, (d_1, d_2)) \;\equiv\; x_9^{lb} \geq d_1 \wedge x_9^{lb} \leq d_2 \tag{2.20}$$

where d_1 and d_2 are template parameters whose values are to be inferred during the analysis. The template form expresses the fact that all values of x_9^{lb} lie in the interval $[d_1, d_2]$.

We now show how iterative solving of Formula 2.18 allows 2LS to infer values of d_1 and of d_2. The transition relation $\mathit{Trans}(\boldsymbol{x}, \boldsymbol{x}')$ is expressed by the SSA form given in Figure 2.3. Formula 2.18 is repeatedly solved until it is unsatisfiable. We assume that $\mathit{Trans}(\boldsymbol{x}, \boldsymbol{x}')$ is already proven by the solver to be satisfiable (since it describes an acyclic program) and that in every iteration we only solve current instances of the invariant (since an incremental solver is used).

In Formula 2.18, two instances of a template are used. The instance $\mathcal{T}(\boldsymbol{x}, \boldsymbol{d})$ describes a loop invariant and hence we define its guarded form to be:

$$(g_5 \wedge g_9^{ls}) \Rightarrow \mathcal{T}(\{x_9^{lb}\}, (d_1, d_2)) \tag{2.21}$$

where g_5 guards the reachability of the loop and g_9^{ls} is the loop-select variable corresponding to x_9^{lb}.

The second instance of the template $\mathcal{T}(\boldsymbol{x}', \boldsymbol{d})$ describes the same loop invariant for the program state $\boldsymbol{x}'$ obtained after execution of the transition relation from the state $\boldsymbol{x}$. For the variable x in the analysed loop, this corresponds to the SSA variable x_8 (i.e. the state of x at the end of the loop body). Its guarded form is hence:

$$(g_5 \wedge g_7) \Rightarrow \mathcal{T}(\{x_8\}, (d_1, d_2)) \tag{2.22}$$

where g_7 guards the reachability of the definition of x_8.

We now describe particular iterations of solving of Formula 2.18.

1. As stated in Section 2.4.1, the initial value of the template parameter is $\boldsymbol{d} = \perp$ and $\mathcal{T}(\boldsymbol{x}, \perp) \equiv \mathit{false}$. The formula to solve is hence:

$$(g_5 \wedge g_9^{ls}) \Rightarrow \mathit{false} \wedge \neg((g_5 \wedge g_7) \Rightarrow \mathit{false}). \tag{2.23}$$

   The only possibility to satisfy the formula is that $(g_5 \wedge g_9^{ls})$ is evaluated to false. Since $g_5 = \mathit{true}$, g_9^{ls} must be equal to *false*.

   In such case, $x_3 = 0$ is chosen as the value of x_6^{phi} and subsequently $x_8 = 1$. This value represents the value of x at the end of the loop body and thus it is used to refine the current invariant. Both d_1 and d_2 are updated to 1 and the current invariant is:

$$x_9^{lb} \geq 1 \wedge x_9^{lb} \leq 1. \tag{2.24}$$

2. In the second iteration, we use the previously computed invariant in Formula 2.18. The formula to solve is:

$$\begin{gathered}(g_5 \wedge g_9^{ls}) \Rightarrow (x_9^{lb} \geq 1 \wedge x_9^{lb} \leq 1) \wedge \\ \neg((g_5 \wedge g_7) \Rightarrow (x_8 \geq 1 \wedge x_8 \leq 1)).\end{gathered} \tag{2.25}$$

   In order to satisfy this formula, the solver must choose 1 as the value of x_9^{lb} and hence the value of $x_8 = 2$. Using this value to refine the template invariant causes the template parameter d_2 to be updated to 2. The current instance of the invariant is:

$$x_9^{lb} \geq 1 \wedge x_9^{lb} \leq 2. \tag{2.26}$$

3. Analogously to the previous step, values $3, 4, 5, \ldots$ are consecutively found as models of satisfiability and they are used to update the value of d_2. This continues until the guard g_7 is *false* and the second conjunct of the solved formula cannot be evaluated to *true* (and Formula 2.18 is unsatisfiable).

   Since $g_7 = (x_6^{phi} < 10) \wedge g_5$, such situation occurs for the first time when $x_9^{lb} = 10$. Hence, this is last iteration and the final computed invariant is:

$$x_9^{lb} \geq 1 \wedge x_9^{lb} \leq 10. \tag{2.27}$$

   The invariant can be then used to prove that the assertion $x_6^{phi} = 10 \vee \neg g_{10}$ always holds and that the analysed program is correct.

2.4.4 Function Summaries

Even though loop invariants are sufficient for intra-procedural analysis, additional concepts must be used when analysing functions of the original program separately. In 2LS function placeholders and function summaries are used.

A *function placeholder* $f_i(\boldsymbol{x}^{a\_in}, \boldsymbol{x}^{a\_out})$, as described in Section 2.3.2, over-approximates the effect of a call to function f in a program location i (since 2LS does not handle recursive programs, we assume that i is in function other that f). It is a predicate parametrised by two sets of variables $\boldsymbol{x}^{a\_in}$ and $\boldsymbol{x}^{a\_out}$ denoting the input and the output arguments of the call, respectively. Values of output arguments are initially unconstrained and can be constrained with means of a function summary.

A *function summary* abstracts the behaviour of a function. In other words, it describes how a function f transforms its formal inputs $(\boldsymbol{x}^{in}{}_f)$ into outputs $(\boldsymbol{x}^{out}{}_f)$. In 2LS, a function summary is described by an inductive invariant over the sets of variables $\boldsymbol{x}^{in}{}_f$ and $\boldsymbol{x}^{out}{}_f$. Formally, given a computed inductive invariant Inv approximating the set of all states reachable by a function f, input and output variables $\boldsymbol{x}^{in}{}_f$ and $\boldsymbol{x}^{out}{}_f$, and a predicate $Init_f(\boldsymbol{x})$ describing the initial states of the function f, a summary of f is a predicate Sum_f such that:

$$\begin{aligned}\forall \boldsymbol{x}, \boldsymbol{x}' : (\boldsymbol{x}^{in}{}_f \subseteq \boldsymbol{x} \wedge Init(\boldsymbol{x}) \wedge \\ Inv(\boldsymbol{x}') \wedge \boldsymbol{x}^{out}{}_f \subseteq \boldsymbol{x}') \implies Sum_f(\boldsymbol{x}^{in}{}_f, \boldsymbol{x}^{out}{}_f).\end{aligned} \tag{2.28}$$

The first line of the implication antecedent expresses the fact that the initial states of the function depend on the set of input parameters $\boldsymbol{x}^{in}{}_f$. Using the computed invariant for the output variables (second line), we define a summary $Sum(\boldsymbol{x}^{in}{}_f, \boldsymbol{x}^{out}{}_f)$ of the function. The summary can be then used to constrain a function call placeholder $f_i(\boldsymbol{x}^{a\_in}{}_i, \boldsymbol{x}^{a\_out}{}_i)$ by replacing formal input and output variables $\boldsymbol{x}^{in}$ and $\boldsymbol{x}^{out}$ in $Sum(\boldsymbol{x}^{in}, \boldsymbol{x}^{out})$ by actual values of inputs and outputs $\boldsymbol{x}^{a\_in}{}_i$ and $\boldsymbol{x}^{a\_out}{}_i$, respectively [13].

Example

To better illustrate inter-procedural analysis, we show an example of analysis of a program containing a function call. The source of the program and SSA forms of both present functions are given in Figure 2.4.

```c
int inc(int x)
{
  int y = x + 1;
  return y;
}

void main()
{
   int a = 0;
   int b = inc(a);
   assert(b == 1);
}
```

(a) The C program

```
1
2  $g_2 = true$
3  $y_3 = x_1 + 1$
4  $rv_4 = y_3$
5
```

(b) The SSA form of `inc`

```
7
8  $g_8 = true$
9  $a_9 = 0$
10 $inc_{10}(\{a_9\}, \{b_{10}\})$
11 $b_{10} = 1$
```

(c) The SSA form of `main`

Figure 2.4: Example of a program containing a function call

The SSA form of the function `inc` contains a variable x_1 that corresponds to the input value of the function parameter x. Also, a special variable rv_4 denoting the return value of the function is introduced. The sets of input and output variables of the function can be defined as follows:

$$\begin{aligned}\boldsymbol{x}^{in}{}_f &= \{x_1\} \\ \boldsymbol{x}^{out}{}_f &= \{rv_4\}\end{aligned} \tag{2.29}$$

The SSA form of `main` contains a function call placeholder $inc_{10}(\{a_9\},\{b_{10}\})$ that approximates the effect of the function call. During the analysis, we first compute summary of `inc` and then we use it to constrain the function placeholder and to prove that the assertion holds.

Computing Summary We compute the summary predicate $Sum_{inc}(\{x_1\},\{rv_4\})$ using the template-based synthesis of inductive invariants. Similarly to the previous example, we use the abstract template polyhedra domain [44] but now we use its subclass for the *zones abstract domain*. Here, instead of computing an interval for each individual variable, we compute for each pair of variables an interval in which their difference lies. In case of the function `inc`, important variables are x_1 and rv_4 and hence we specify the form of the template as follows:

$$\mathcal{T}(\{x_1,rv_4\},(d_1,d_2)) \equiv (rv_4 - x_1) \geq d_1 \wedge (rv_4 - x_1) \leq d_2 \tag{2.30}$$

Using the algorithm for invariant inference, we compute $d_1 = d_2 = 1$. Hence, the summary $Sum_{inc}(\{x_1\},\{rv_4\})$ is as follows:

$$Sum_{inc}(\{x_1\},\{rv_4\}) = (rv_4 - x_1) \geq 1 \wedge (rv_4 - x_1) \leq 1 \tag{2.31}$$

Constraining Function Call Placeholder After the summary is computed, it can be used to constrain the value of the function call placeholder $inc_{10}(\{a_9\},\{b_{10}\})$. This is done by replacing input and output parameters of the function by input and output arguments of the call. In the example, x_1 and rv_4 are replaced by a_9 and b_{10}, respectively:

$$inc_{10}(\{a_9\},\{b_{10}\}) = (b_{10} - a_9) \geq 1 \wedge (b_{10} - a_9) \leq 1. \tag{2.32}$$

Using the constraint given in Formula 2.32, 2LS can prove that the assertion $b_{10} = 1$ always holds and that the program is thus correct.

Chapter 3

Existing Methods for Shape Analysis

Shape analysis is a technique of static analysis aimed at discovering shapes of data structures dynamically allocated on the program heap. Such structures usually include various forms of linked lists (singly or doubly linked, circular, nested, etc.), trees, or more complicated structures such as skip-lists.

Unlike stack and static memory that can be abstracted by a finite set of named variables occuring in the analysed program, heap data is potentially unbounded and seemingly arbitrary. This poses a challenge in terms of used heap abstractions and makes shape analysis an interesting and widely explored research topic. In this chapter, we give an overview of some of the current approaches to shape analysis. For a more complete survey, we refer to [32].

We split the described methods into multiple groups based on the models they use to abstract the shape of the heap. The first two groups, using namely various kinds of logics, automata, and graphs are store-based, i.e., they describe the heap explicitely. On the contrary, the approaches in last group are inspired by storeless semantics.

3.1 Logic-based Methods

One of the first approaches to shape analysis is based on a so-called *three-value logic* [43]. This logic introduces a new value *unknown* to the traditional boolean values *true* and *false*. The approach is based on abstract interpretation and the value *unknown* is used to express the fact that some elements may or may not be in a relation after an abstraction is done. The method is rather generic but usually requires some manual intervention to be sufficiently scalable.

Another approach uses a so-called *Pointer Assertion Logic* [38] to verify data structures that can be described by graph types. The technique is highly modular, however, it is semi-automated only—it requires explicit loop and function call invariants.

A different group of shape analysis techniques uses *separation logic* [41]. It is an extension of *Hoare logic* developed specifically for reasoning about programs manipulating heap. It builds on *Hoare triples*, which is a mechanism to describe how a program state changes after execution of a piece of code. A triple has a form $\{P\}C\{Q\}$, where P, Q are predicates (often called a *precondition* and a *postcondition*) and C is a command. The predicate P is assumed to hold before the execution of C, and Q is assumed to hold afterwards. Separation logic extends the predicate logic by several new operators and symbols: *emp* (a constant representing an empty heap), $e \mapsto e'$ (an operator expressing the fact that the heap contains a single cell at address e which maps to the value e'), $p * q$ (an operator expressing that the heap can be separated into two parts where p holds for one and q holds for the other), and $p \mathrel{-\!\!*} q$ (an operator expressing that if the heap is extended by a disjoint part in which p holds, then q will hold after the extension).

There are many fully-automated tools based on separation logic such as Space Invader [48] and SLAyer [3]. Separation logic is also applied in practice, e.g. in combination with *bi-abduction* [11] in the Facebook Infer tool [22]. However, the tool misses support for low-level features such as block operations and advanced pointer arithmetics and handles simple data structures only (mainly lists).

Another verifier based on separation logic is $\mathtt{S2}_{\mathtt{td}}$ [34]. It implements a satisfiability procedure for separation logic extended by user-defined inductive predicates. Although is is able to handle very complex data structures (e.g. trees with linked leaves), it is rather fragile and can fail on quite simple structures if they are not handled by the program in a suitable way. Moreover, a support for combinations with other abstract domains is not very advanced and the approach has a problem with reliable diagnostics of discovered possible errors.

Also, more recently, automation of separation logic using SMT solvers by reduction to effectively propositional logic has been proposed by [40, 29, 30].

3.2 Methods Using Automata and Graphs

Another group of shape analysis tools describes the state of the heap using various forms of automata and graphs. One of such tools is Predator [20] which uses *symbolic memory graphs (SMGs)* [33]. These are designed as an abstract domain for the framework of abstract interpretation. SMGs model the heap with byte-precision and use summary nodes to represent abstractions of linked lists of unbounded length. They are designed to handle low-level manipulation of dynamic data structures. The usability of the approach is confirmed by mutliple wins in the heap-related categories of the International Competition on Software Verification (SV-COMP). However, the approach has not yet been extended to tree-like structures and it is missing a combination with other abstract data domains.

A different approach based on graphs uses *tree automata* and *regular tree model checking* [8] and is implemented, e.g., in the Forester tool [25]. The approach uses automata over words and trees to describe the shape of the heap and a tree-automata-based abstraction to over-approxmate the set of reachable heap configurations. The abstraction can be refined by counterexample-guided refinement. Combining these approaches with reasoning about value properties is not easy as shown in the works [1, 27] that extended Forester with reasoning about finite data and a specialised support for handling ordered list segments.

3.3 Methods Using Storeless Semantics

All of the above approaches are store-based, i.e. they explicitly describe the state of the heap using some logic or graphs. On the contrary, methods based on *storeless semantics* [31] use pointer access paths to describe reachable shapes on the heap [14, 42, 37, 9]. A *pointer access path* does not concretely express the heap state, it only describes which dynamic objects are reachable from a pointer. Using a set of access paths for each pointer, one can efficiently describe the shape of (the reachable part of) the heap. These approaches usually use abstract interpretation over control-flow graphs and their support of dealing with the data content is limited [37]. However, pointer access paths proved the most suitable for our purposes and our work is heavily inspired by them.

Chapter 4

Thesis Goals

The main topic of my PhD thesis is research of shape analysis in the context of program verification using template-based invariant inference. During my work, I intend to build on rich expertise in the fields of shape analysis that we have in the VeriFIT research group (e.g. two of the tools mentioned in Chapter 3, namely Predator [20] and Forester [25], were developed within our group). I want to develop an approach of integrating shape analysis into the 2LS framework and then explore possibilities that will open. Since the framework allows for simple combination of multiple verification approaches together, we could be able to analyse complex properties of real-world programs that use (among other features) dynamically allocated memory. Overall, we may summarize the goals for this thesis as follows:

1. **Development of a new abstract shape domain.** One of the main problems of the current shape analysers is the difficulty of their combination with analyses of other program properties. We address this problem by proposing a novel abstract shape domain suitable for verification approaches using invariant inference based on templates, such as the one implemented in the 2LS framework. Since all domains in 2LS share the common form of templates, a combination with other analyses already present in the framework is then straightforward. Currently, we are not aware of any abstract shape domain having the required form of templates, therefore its design and implementation is the first goal of my thesis. It will be inspired by existing approaches to shape analysis—we have found the methods based on storeless semantics (Section 3.3) the most suitable. The domain should be able to describe dynamic structures of various shapes, with the main focus on the linked lists.

2. **Combination of the shape domain with other abstract domains.** After implementing the novel abstract shape domain for the 2LS framework, we focus on possibilities of its combination with other abstract domains. Here, the most interesting is a combination with abstract domains for analysis of numerical values. This could allow us to analyse complex properties of programs, such as contents of dynamic data structures, length and sortedness of linked lists, or even complicated low-level heap manipulations using pointer arithmetic and structure offsets (i.e. using a numerical domain not on data stored in dynamically linked structures but on the low-level features of pointers themselves).

3. **Introduction of methods for interprocedural shape analysis.** Another common problem of software verification approaches in general is a limited scalability.

The methods being used are often very advanced and have a high computational complexity and therefore struggle to scale on large real-world programs. For example, the method used by 2LS is exponential in the number of commands of the analysed program, since it relies on solving the satisfiability (SAT) problem, which is generally NP-complete. One of the possible solutions to this problem is interprocedural analysis, i.e. analysing each function of the program separately. This is often done using a concept of *summaries*—a summary describes how a function transforms its inputs into outputs. In the 2LS framework, summaries are supported for programs without side-effects (i.e. programs not manipulating heap) in the way described in Section 2.4.4. Due to this, the next goal of my thesis is to adapt the proposed shape analysis for using summaries. This is, however, a rather difficult task since we will need to come with a way to describe changes to parts of the existing heap that are reachable from the function parameters. There are, basically, two suitable approaches to this problem. The first one is computing a calling context for every function call. The calling context describes the shape of the heap in time of the function call and can be used to simplify analysis of the called function. However, the problem is that when the function is called in a different context, it must be re-analysed, which partially eliminates the benefits of using interprocedural analysis. The other possible approach is to analyse each function individually, without being sensitive to its calling context. Here, we could build on the principle of bi-abduction [11], which is based on inferring assumptions about the state of the heap at the function beginning.

4. **Application of shape analysis outside of safety verification.** The proposed shape domain could be used not only to verify safety, but also to analyse different properties of programs working with dynamic data structures. These may include (non)termination of programs (there is already a support for termination and non-termination analysis in 2LS), inference of resource bounds, or combination with concurrency.

Chapter 5

Template-Based Verification of Heap Manipulating Programs

We propose a novel technique of shape analysis suitable for analysis engines that perform automatic invariant inference using an SMT solver such as the one described in Chapter 2. We have published this work at the *2018 Formal Methods in Computer Science (FMCAD)* conference and the following text is based on this paper [36]. The author of this report has contributed most of the main ideas of the paper as well as did most of the implementation and experimental work and contributed also a significant part of the writing.

One of the main advantages of the template-based verification is that it uses a unified form of abstract domains (templates) and delegates semantic reasoning to an SMT solver. This makes it straightforward to compute invariants describing both shape and value properties of data structures, which is more difficult when combining domains that are based on different principles.

The described approach assumes non-recursive programs with all function calls inlined.

Contributions The contributions of the work described in [36], which form the contents of Sections 5.1–5.5, are as follows:

1. We propose a novel abstract template domain for reasoning over heap-allocated data structures such as singly- and doubly-linked lists using a template-based parameter synthesis engine.

2. We show how we can build product and power domain combinations of our heap domain with structural domains (e.g. trace partitioning) and value domains such as template polyhedra that capture the contents of data structures.

3. We implement our abstract heap domain in the 2LS verification tool for C programs. We demonstrate the power of the proposed domain on benchmarks, which require combined reasoning about the shape and contents of data structures, showing that other tools, which performed well in SV-COMP, cannot handle these examples.

These contributions represent a contribution to the first two points of the thesis goals described in Chapter 4. A further research in these areas is, however, needed.

Running example To better illustrate the concepts and methods described in the following text, we use the program in Listing 5.1 as a running example. It creates a singly-linked

```
 1 typedef struct node {
 2   int val;
 3   struct node *next;
 4 } Node;
 5
 6 int main() {
 7   Node *p, *list = malloc(sizeof(Node));
 8   Node *tail = list;
 9   *list = {.next = NULL, .val = 10};
10   while (__VERIFIER_nondet_int()) {
11     int x = __VERIFIER_nondet_int();
12     if (x < 10 || x > 20) continue;
13     p = malloc(sizeof(Node));
14     *p = {.next = NULL, .val = x};
15     tail→next = p; tail = p;
16   }
17   while (1) {
18     for (p = list; p!= NULL; p = p→next) {
19       assert(p→val <= 20 && p→val >= 10);
20       if (p→val < 20) p→val++;
21       else p→val /= 2;
22     }
23   }
24 }
```

Listing 5.1: A running example

list, each node containing a value between 10 and 20 (Lines 7–15). The list is afterwards traversed repeatedly and the value of each node is either incremented by 1 or halved (Lines 16–22). We add an assertion that, in every iteration, the value of each node stays between 10 and 20. The goal of the analysis is to prove that the assertion always holds. This requires an analysis capable of reasoning about unbounded linked data structures and numerical content of their nodes at the same time.

To prove this property we have to infer that the value of the `val` field of the dynamic objects allocated in Line 7 and 13 is always in the range $[10, 20]$. With the help of our technique, we will infer an invariant for the loop on Line 10 that states the following:

1. `tail` may point to the sets of `Node` objects created in Line 7 and 13. We denote these sets ao_7 and ao_{13}, respectively.

2. The `next` field of ao_7 may point to ao_{13} or to `null`. Its `val` field has a value in the interval [10,10].

3. The `next` field of ao_{13} may point to ao_{13} or `null`. However, its `val` field has a value in the interval [10,20]. This means that ao_{13} abstracts a set of `Node` objects whose `val` fields have values in the interval [10,20].

For the loop in Line 18, we infer the invariant that the `val` fields of ao_7 and ao_{13} must both be in the interval [10,20], which implies that the property holds.

5.1 Abstract Memory Operations in the SSA form

We now propose a representation of heap memory and operations over it, designed to be used within the approach laid out in Chapter 2. The proposal respects the fact that the considered SSA form is an acyclic program representation, over-approximating reachable values of variables used in loops.

5.1.1 Abstract Memory Representation

Under our assumption of fully inlined, non-recursive programs, *static memory objects* correspond simply to a finite set Var of *program variables*: we do not need to consider the stack. We let $PVar, SVar \subseteq Var$, $PVar \cap SVar = \emptyset$, be the sets of variables of *pointer* and *structure type*, respectively. A linked data structure in C is typically defined using a `struct` type, which groups together named *fields* for the payload data and the link pointers (see Lines 1–4 in Listing 5.1). We use Fld to denote the finite set of fields used in the given program. Let $PFld \subseteq Fld$ be the set of all pointer-typed fields.

Abstract Dynamic Objects

We use *abstract dynamic objects* to represent *dynamic memory objects*, i.e. those that are allocated using `malloc` (or some of its variants) on the heap. An abstract dynamic object represents a set of concrete dynamic objects allocated at the same *allocation site* i, e.g. by the same `malloc` call located at Line i in Listing 5.1. However, a single abstract dynamic object is not sufficient to represent *all* concrete dynamic objects allocated by a given `malloc`. The reason for this is that the program may use several independent objects created at an allocation site at the same time. Typically, this issue is solved by the analysis algorithm materialising dynamic objects on-demand. We take a different approach and statically over-approximate the maximum number n_i of concrete objects required (see the next section below). Hence, we use a *set* $AO_i = \{ao_i^k \mid 1 \leq k \leq n_i\}$ of abstract dynamic objects for that purpose. We let $AO = \cup_i AO_i$ and require $Var \cap AO = \emptyset$ and $AO_i \cap AO_j = \emptyset$ for $i \neq j$. The set of all objects of our program abstraction is then $Obj = AO \cup Var$.

Pairs consisting of an abstract dynamic object and a field, i.e. elements of the set $AO \times Fld$, represent an abstraction of the appropriate *fields* of all the represented concrete objects. We use the “dot” notation to represent such pairs: e.g. $ao_i.next$ denotes the abstraction of the $next$ field of all the concrete dynamic objects represented by ao_i.

We define $Ptr = PVar \cup ((SVar \cup AO) \times PFld)$ to be the set of all *pointers* of the given program abstraction. Pointers can be assigned addresses of objects. Since we currently do not support pointer arithmetic, the only addresses that we consider are *symbolic addresses* of static and dynamic objects together with the special address null. The symbolic address of an abstract dynamic object ao_i is an abstraction of the symbolic addresses of the concrete dynamic objects represented by ao_i. To get the address of both static and dynamic objects, we use the &-operator. Hence, the set $Addr$ of addresses that we consider is defined as $Addr = \{\&o \mid o \in Obj\} \cup \{\mathsf{null}\}$.[1]

Pre-Materialisation

As mentioned above, instead of materialising dynamic objects on-demand, we pre-materialise a sufficient number n_i of them for each allocation site i and encode them into our SSA rep-

[1]We currently assume that addresses of newly allocated objects are fresh. Hence, we can miss behaviours where some memory space is recycled while some pointers are still pointing to it, which is undefined according to the C standard, but sometimes used in practice. If that was a problem, we could, e.g., extend our preliminary static analysis to detect objects that can possibly be in that form and add them among possible returns from the allocation.

resentation. In order for this abstraction to be sound, it is sufficient that the number n_i equals the maximal number of distinct concrete objects allocated at i that are simultaneously pointed to by some pointer at any location of the analysed program.

For each allocation site i, we compute the number n_i as follows. First, using a standard static may-alias analysis, we over-approximate, for each program location j, the set P_j^i of all pointer expressions of the source program that *may point* to some object allocated at i. These might be pointer variables from $PVar$, pointer-typed fields of static objects from $SVar \times PFld$, or pointer-typed fields of dynamic objects accessed through dereferences of pointers—i.e. elements of $PVar \times PFld$. For simplicity, we assume that all chained dereferences of the form $p \to f_1 \to f_2$ with $f_1, f_2 \in PFld$ are broken into two expressions using an intermediate variable. Overall, $P_j^i \subseteq PVar \cup ((SVar \cup PVar) \times PFld)$. Next, we compute the *must-alias relation* $\sim_j$. For each pair of pointers p and q and for each program location j, $p \sim_j q$ iff p and q must point to the same concrete dynamic object at j. Finally, we partition the set P_j^i into equivalence classes by $\sim_j$, and n_i is given by the maximal number of such classes at any j.

5.1.2 Operations over the Abstract Memory Representation

After defining the abstract representation of dynamic memory, we describe the way we encode typical heap-manipulating operations. These include allocation, reading, writing, and deallocation of dynamic memory.

Dynamic Memory Allocation

We represent a call to `malloc` at program location i by a non-deterministic choice among the addresses of objects from the set AO_i. Hence, a statement p = `malloc`(...) at i is translated to the formula

$$p_i = g_{i,1}^{os} \,?\, \&ao_i^1 : (g_{i,2}^{os} \,?\, \&ao_i^2 : (\dots (g_{i,n_i-1}^{os} \,?\, \&ao_i^{n_i-1} : \&ao_i^{n_i}))) \tag{5.1}$$

where $g_{i,j}^{os}$, $1 \leq j < n_i$ are free Boolean variables, so-called *object-select guards*.

Example In Listing 5.1, two calls of `malloc` occur on Lines 7 and 13. For Line 7, a single abstract dynamic object ao_7 is created as there is just one concrete object allocated.[2] The `malloc` on Line 13 must be represented by two objects ao_{13}^1 and ao_{13}^2 as, e.g. on Line 14, variables `tail` and `p` may point to different concrete objects allocated by this `malloc` call. Specifically, the statement on Line 13 will be translated into the equality $p_{13} = g_{13}^{os} \,?\, \&ao_{13}^1 : \&ao_{13}^2$. Abstract dynamic objects ao_{13}^1 and ao_{13}^2 then collectively represent all concrete dynamic objects allocated in the loop.

Reading through Dereferenced Pointers

We handle expressions of the form $p \to f$ for $p \in PVar$, $f \in Fld$ appearing on the right-hand side of assignments or in conditions as follows. We first perform a *may-points-to analysis*, which over-approximates for each pointer $p \in Ptr$ and each program location i the set of objects from Obj that p may point to at i. Using the result of the analysis, we can replace

[2]In fact, we should write ao_7^1, but we omit the superscript when a single abstract object suffices for the given `malloc`. Likewise for the object-select guards below.

the pointer dereference $p \rightarrow f$ by a choice among the values of the field f of the objects possibly pointed to by p.

To facilitate the replacement, we introduce purely logical *dereference variables*. Assume that at program location i there appears an R-expression $p \rightarrow f$ and that the pointer p may point to a set of objects $O \subseteq \mathit{Obj}$ at i. We replace the use of $p \rightarrow f$ by using a fresh variable $\mathit{drf}(p).f_i$ whose value is defined by the formula

$$\begin{gathered}\bigwedge_{o \in O} \big(p_j = \&o \Longrightarrow \mathit{drf}(p).f_i = o.f_k\big) \wedge \\ \Big(\bigwedge_{o \in O} p_j \neq \&o\Big) \Longrightarrow \mathit{drf}(p).f_i = o_\perp\end{gathered} \tag{5.2}$$

where p_j, $o.f_k$ are the relevant versions of the concerned variables at program location i and $o_\perp$ denotes a special "unknown object" (a result of a dereference of an unknown or invalid (null) address).[3]

Example We give the translation of the assignment $p = p \rightarrow \mathit{next}$ from Line 18 in Listing 5.1. Since the assignment is executed at the end of each loop iteration, we define its program location to be Line 22. At this program location, p may point to the set of objects $\{ao_7, ao^1_{13}, ao^2_{13}\}$. Hence, the assignment will be represented by the following formula:

$$\begin{aligned}&p_{22} = \mathit{drf}(p).\mathit{next}_{22} \wedge \\ &p^{\mathit{phi}}_{18} = \&ao_7 \Rightarrow \mathit{drf}(p).\mathit{next}_{22} = ao_7.\mathit{next}^{\mathit{phi}}_{18} \wedge \\ &\bigwedge_{l=1,2} \Big(p^{\mathit{phi}}_{18} = \&ao^l_{13} \Rightarrow \mathit{drf}(p).\mathit{next}_{22} = ao^l_{13}.\mathit{next}^{\mathit{phi}}_{18}\Big) \wedge \\ &\Big(p^{\mathit{phi}}_{18} \neq \&ao_7 \wedge \bigwedge_{l=1,2} p^{\mathit{phi}}_{18} {\neq} \&ao^l_{13}\Big) \Rightarrow \mathit{drf}(p).\mathit{next}_{22} = o_\perp.\end{aligned} \tag{5.3}$$

The first conjunct represents the transformed assignment, and the following conjuncts define the value of the dereference variable. The value of p entering program location 22 is the value from the loop head p^{phi}_{18}. If it equals the address of ao_7, ao^1_{13}, or ao^2_{13}, the value of $\mathit{drf}(p).\mathit{next}_{22}$ is $ao_7.\mathit{next}^{\mathit{phi}}_{18}$, $ao^1_{13}.\mathit{next}^{\mathit{phi}}_{18}$, or $ao^2_{13}.\mathit{next}^{\mathit{phi}}_{18}$, otherwise, it equals $o_\perp$.

As an optimisation, if the dereference variable is once created and the value of the concerned expression does not change, we reuse the existing dereference variable. Second, when dealing with a statement like $v = p \rightarrow f$, the use of the dereference variable may seem unnecessary as one can plug v_i instead of $\mathit{drf}(p).f_i$ into the formula defining the value of $\mathit{drf}(p).f_i$. This can be done, but, as explained below, the use of dereference variables can give us more precision when dealing with sequences of reading and writing operations.

Writing through a Dereference

When writing into an abstract dynamic object ao_i, we need to respect the fact that only one concrete object abstracted by ao_i is actually written to, and the others keep the original value. Hence, we need to make a join of the original and the new value. We again use dereference variables to facilitate the transformation.

[3] A dereference of the form $*p$ for a non-structured object can be handled analogously, just without the field f in the above formula.

Assume that at program location i, we have an assignment $p \to f = v$, $p \in PVar$, $f \in Fld$, $v \in Var$, and that p may point to a set of objects $O \subseteq Obj$ at the entry to i.[4] We replace the L-expression $p \to f$ by a fresh variable $drf(p).f_i$ whose value is defined by the value of v, i.e. we assert that $drf(p).f_i = v_l$ where v_l is the version of v valid at program location i. We then use $drf(p).f_i$ to update the value of the field f of the referenced object, using the formula

$$\bigwedge_{o \in O} \left(o.f_i = (p_j = \&o \wedge g_i^{os}) \,?\, drf(p).f_i : o.f_k \right) \tag{5.4}$$

where p_j, $o.f_k$ are the relevant versions of the variables p and $o.f$ at program location i.[5] The formula expresses the fact that $o.f_i$ gets updated if p equals the address of o, otherwise its value remains unchanged; k is the last program location before i where the value of $o.f$ was changed. The object-select guard g_i^{os}, which is a freshly introduced unconstrained Boolean variable, enforces that the value of field f is changed in only one of the concrete objects abstracted by o while it remains unchanged in the other objects abstracted by o. If o is not allocated in a loop (and hence representing a single instance), g_i^{os} may be omitted.

Example For illustration, the assignment `tail->next=p` from Line 15 of Listing 5.1 will be translated into the following formula:

$$\begin{aligned}
&(drf(list).next_{15} = p_{13}) \wedge \\
&ao_7.next_{15} = (list_{10}^{phi} = \&ao_7) \,? \\
&\qquad\qquad drf(list).next_{15} \;:\; ao_7.next_{10}^{phi} \wedge \\
&\bigwedge_{l=1,2} (ao_{13}^{l}.next_{15} = (list_{10}^{phi} = \&ao_{13}^{l} \wedge g_{15}^{os}) \,? \\
&\qquad\qquad\qquad drf(list).next_{15} \;:\; ao_{13}^{l}.next_{10}^{phi}).
\end{aligned} \tag{5.5}$$

As mentioned above, the use of dereference variables may increase the precision of our analysis. This happens in particular when we write into an abstract object through some pointer and later read the written value back through the same pointer (or a pointer aliased with it) without any change of the pointers and the concerned value in between. Then, we get back exactly the value that we wrote, which would otherwise not happen due to the joins involved.

Memory Free

Since the `free` operation has no effect on the heap reachability itself, we defer its discussion to Section 5.3 devoted to checking memory safety.

5.2 An Abstract Domain for Heap Analysis

We will now work towards our template-based abstract domain suitable for reasoning about properties of heap-manipulating programs, starting from a base shape domain and refining it. We will show that, due to the fact that all domains in the considered approach are based on templates, the new domain can be easily combined with other domains, e.g. for inferring properties about numerical data of data structures.

[4] More complex assignments can be transformed into this form.

[5] A write to a dereference of the form $*p$ to a non-structured object can be handled analogously, omitting field f from the formula.

5.2.1 Base Abstract Shape Domain

In the considered approach, an abstract domain needs to have the form of a *template*—a fixed, parametrised, quantifier-free first-order logic formula describing the desired property of a program. As described in Section 2.4, templates are used to efficiently compute *loop invariants* of the analysed program. These are used to constrain values of the *loop-back variables* that are used in the SSA-based program encoding to over-approximate values returning from the end of the loop to the loop head. Hence, a loop invariant describes a property that holds for some program variables at the end of the loop body after any iteration of the loop. Hence, we limit our shape domain to the set Ptr^{lb} of all *loop-back pointers*. Let L be the set of all loops in the program. Since there is one loop-back pointer variable for each pointer variable and each loop, we define $Ptr^{lb} = Ptr \times L$. We denote elements $(p, l) \in Ptr^{lb}$ by p_i^{lb} where i is the program location of the end of the loop l. Intuitively, the value of p_i^{lb} is an abstraction of the value of the pointer p coming from the end of the body of the loop l. The property that our base shape domain describes is the *may-point-to* relation between the set Ptr^{lb} and the set $Addr$.[6]

The template of our base shape domain has the form:

$$\mathcal{T}^S \equiv \bigwedge_{p_i^{lb} \in Ptr^{lb}} \mathcal{T}^S_{p_i^{lb}}(d_{p_i^{lb}}). \tag{5.6}$$

It is a conjunction of so-called *template rows* $\mathcal{T}^S_{p_i^{lb}}$, each row corresponding to one loop-back pointer from the set Ptr^{lb}. A template row $\mathcal{T}^S_{p_i^{lb}}(d_{p_i^{lb}})$ describes the may-point-to relation for the loop-back pointer p_i^{lb}. The parameter $d_{p_i^{lb}} \subseteq Addr$ of the row (a so-called *abstract value of the row*) specifies the set of all addresses from the set $Addr$ that p may point to at the location i. The template row can thus be expressed as the quantifier-free formula

$$\mathcal{T}^S_{p_i^{lb}}(d_{p_i^{lb}}) \equiv \bigvee_{a \in d_{p_i^{lb}}} p_i^{lb} = a. \tag{5.7}$$

Abstract values of template rows corresponding to pointer fields of abstract dynamic objects allow the domain to describe unbounded linked paths in the heap, such as list segments.

Example In Listing 5.1, a list segment is created by the first loop. Objects in the segment are linked through the pointer field `next`, and they are represented by the abstract dynamic objects ao^1_{13} and ao^2_{13}. In our base shape domain, the shape of this segment will be described by an invariant for the first loop, specifically by the two template rows for $ao^1_{13}.next^{lb}_{16}$ and $ao^2_{13}.next^{lb}_{16}$. They will give us the formula

$$\bigwedge_{l=1,2} \mathcal{T}^S_{ao^l_{13}.next^{lb}_{16}}(\{\&ao^1_{13}, \&ao^2_{13}, \mathsf{null}\}) \tag{5.8}$$

where the rows $\mathcal{T}^S_{ao^l_{13}.next^{lb}_{16}}$ are the formulae

$$\begin{aligned} ao^l_{13}.next^{lb}_{16} &= \&ao^1_{13} \vee \\ ao^l_{13}.next^{lb}_{16} &= \&ao^2_{13} \vee \\ ao^l_{13}.next^{lb}_{16} &= \mathsf{null}. \end{aligned} \tag{5.9}$$

[6]Note that unlike the previously mentioned point-to relations, this relation is computed not just syntactically but using the considered abstract semantics.

These formulae say that the `next` fields of both ao^1_{13} and ao^2_{13} may either point to one of the objects themselves or to null. This describes an unbounded linked path in the heap composed of objects abstracted by ao^1_{13} or ao^2_{13} and terminated by null.

5.2.2 Shape Domain with Symbolic Loop Paths

Unfortunately, base shape templates are not precise enough for many heap-manipulating programs. One often needs to allow the invariant of a loop to be able to distinguish which loops were or were not executed while reaching the given loop. This can, e.g. distinguish which objects were allocated and can hence be processed in the given loop.

To deal with the above problem, we introduce the concept of *symbolic loop paths* and compute different invariants for different paths. Since we use loop-select guards to express the control flow through the loops (see Section 2.3.2), a symbolic loop path is simply a conjunction of loop-select guards.[7] Let G^{ls} be the set of all loop-select guards of all loops in a program. A symbolic loop path π is then formally defined as $\pi = \bigwedge_{g \in G^{ls}} l_g$ where l_g is a literal of the variable g, i.e. either g or $\neg g$. We use Π to denote the set of all symbolic loop paths of a given program. A *shape template extended with symbolic loop paths* is then given by the formula

$$\mathcal{T}^L \equiv \bigwedge_{\pi \in \Pi} \pi \Longrightarrow \mathcal{T}^G_\pi \tag{5.10}$$

where the $\mathcal{T}^G_\pi$ formulae are base shape templates as defined in Section 5.2.1. Here, $\pi_\perp$ is a special path containing negative literals only. On that path no loop invariants are computed since it corresponds to a program path where no loops were executed.

Example We now show invariants for the pointer p for the second loop of the program in Listing 5.1. Using our (trace-insensitive) shape domain, the corresponding template row would be

$$\mathcal{T}^G_{p^{lb}_{22}}\left(\{\&ao^1_{13}, \&ao^2_{13}, \mathsf{null}\}\right). \tag{5.11}$$

In other words, p would be understood as possibly pointing to ao^1_{13} or ao^2_{13} even on paths where they were not allocated. However, symbolic loop paths allow us to obtain two different invariants depending on the execution of the first loop (for simplicity, we only provide the appropriate template row).

$$g^{ls}_{16} \wedge g^{ls}_{22} \Rightarrow \mathcal{T}^G_{p^{lb}_{22}}(\{\&ao^1_{13}, \&ao^2_{13}, \mathsf{null}\}) \tag{5.12}$$

$$g^{ls}_{16} \wedge g^{ls}_{22} \Rightarrow \mathcal{T}^G_{p^{lb}_{22}}(\{\mathsf{null}\}) \tag{5.13}$$

Invariant 5.12 corresponds to the case when the body of the first loop is executed and invariant 5.13 corresponds to the case when the body of the first loop is not executed.

5.2.3 Combinations of Domains

The true power of the template-based verification approach lies in the simplicity of domain combinations. Since templates are general logical formulae, they can be easily composed, forming abstract domains capable of describing more complex properties of programs while relying on the solver to do the heavy-lifting on the combination of the domain operations and the mutual reduction of their abstract values.

[7] The notion of symbolic loop paths can be easily generalised to program path sensitivity by including branches of conditional statements too.

Power Templates

The definition of shape templates with symbolic loop paths shows one way how a complex template can be formed from a simpler one. In this case, the template parameter, i.e. the abstract value, maps particular symbolic loop paths to sets of parameters of the original shape template. In fact, the shape domain could be replaced by any other abstract domain. The symbolic paths template can hence be viewed as a *power template*—in the sense of power domains [19]—which assigns to each element of the base domain an element of the exponent domain.

Product Templates

From the perspective of program analysis, a very interesting possibility is the combination of the shape domain with an abstract domain capable of describing values of variables of non-pointer types, e.g. numerical variables (such as the well-known interval or octagon domains). The simplest way to achieve such a combination is to use a *Cartesian product template* that combines templates of different kinds to be used independently side-by-side. The proposed shape template with loop-back guards $\mathcal{T}^G$ from Section 5.2.2 can be combined with a template for analysis of numerical values $\mathcal{T}^V$ by simply taking their conjunction, i.e. $\mathcal{T}^G \wedge \mathcal{T}^V$. This not only allows us to analyse programs that use pointer and numerical variables simultaneously, but also to reason about the contents of data structures on the heap. We achieve this by analysing numerical fields of abstract dynamic objects using the value part of the template.

In addition, we use this product template as the inner template of the template with symbolic loop paths, forming an even stronger abstract domain: $\mathcal{T}^{LV} \equiv \bigwedge_{\pi \in \Pi} \pi \Longrightarrow \mathcal{T}^G_\pi \wedge \mathcal{T}^V_\pi$. Using this domain for the running example allows us to analyse the shape and the contents of the linked list at the same time, obtaining the invariants described in the Chapter 5 introduction that enable us to prove the given property of interest.

5.3 Memory Safety Analysis

Apart from checking user-defined assertions, we can also verify memory safety. This includes a number of properties: (1) pointer dereferencing safety, (2) `free` safety, and (3) absence of memory leaks.

5.3.1 Dereferencing a null Pointer

Since our invariants are over-approximating the reachable program states, we can soundly verify *may* (or better called *must-not*) properties. To check dereferences of null, for each expression $*p$ occurring in a program location i, we verify the assertion $p_j \neq \mathsf{null}$ where p_j is the version of p valid at i.

5.3.2 `Free` Safety

`Free` safety includes the absence of dereferencing a freed pointer and freeing an already freed pointer (a so-called "double free"). To prove absence from these errors, we introduce a new special variable fr initialised to null, which is then non-deterministically set to the address of the object to be freed in a `free` call. We replace each call of the form `free(p)` at program location i by a formula $fr_i = g^{fr}_i ? p_j : fr_k$, where p_j and fr_k are the versions of p

Table 5.1: Comparison of 2LS using the proposed method with the previous version of the tool over the SV-COMP benchmark.

	RS-Control		**RS-Heap**		**MS-Heap**		**MS-Linked**		**MS-Other**	
	cpu (s)	score	cpu (s)	score	cpu (s)	score	cpu (s)	score	cpu (s)	score
2LS	252	64	41	106	17.5	59	107	7	29	46
2LS-old	1400	45	53	-161	190	-194	96	-182	23	46

and fr, respectively, valid in i, and g_i^{fr} is a free Boolean variable (a so-called *free guard*). Treating fr as a standard pointer-typed variable allows us to over-approximate the set of all freed addresses with the help of our shape domain. Then, in each program location i where either $*p$ or `free(p)` occurs, we can check for the assertion $p_j \neq fr_k$ to prove `free` safety (here, p_j and fr_k are again versions of p and fr, respectively, valid at i).

Even though this approach is sound, it is often too imprecise. Freeing one of the concrete objects represented by an abstract objec does not mean that all the represented objects were freed and that it is not safe any more to dereference/free the abstract object. To improve precision, we modify the representation of `malloc` calls. At each allocation site i, we add one more object ao_i^{co} to the set $\{ao_i^k\}$. The object can be chosen as the result of the allocation non-deterministically like any other ao_i^k, but it is guaranteed to be allocated only once (by an additional condition checking that, upon its allocation, no loop-back pointer can point to it). Hence, ao_i^{co} represents a concrete object. Then, for each allocation site i, we only allow $\&ao_i^{co}$ to be assigned to fr. The checks for free safety described above are done on concrete objects only, avoiding possible imprecision stemming from dealing with multiple objects represented by a single abstract object which would join the possibly different values of these objects. Also, as ao_i^{co} represents an arbitrary concrete object allocated at i, if safety can be proven for it, it can be assumed to hold for any other object allocated at i.

5.3.3 Memory Leaks

Using fr, we then check whether some ao_i^{co} object may be not freed at the end of the program. This allows us to discover memory leaks—if there is a leak, it must be possible to show it on some concrete object. Unfortunately, as we do not track the sequencing of abstract objects representing a set of objects allocated at an allocation site (even when they form a list segment), our analysis typically sees that ao_i^{co} may be skipped in the deallocation loops. Because of this, absence of memory leaks can only be proven for programs without loops (or with loops that can be fully unwound).

5.4 Implementation

We implemented[8] the proposed shape domain within the 2LS framework [45] that uses the template-based verification method described in Chapter 2. We extended the SSA form generated by the framework to handle dynamic memory allocation. 2LS is based on the CPROVER framework [15], which includes an SMT solver based on reduction to propositional logic. We used Glucose 4.0 as the back-end solver in our experiments. We let 2LS inline all functions before running our analysis. For combination with numerical

[8]Available at https://github.com/diffblue/2ls/releases/tag/2ls-0.7.

Table 5.2: Comparison of 2LS with other tools on examples combining unbounded data structures and their stored data.

	2LS	**CPA-Seq**	**Predator**	**Forester**	**Symbiotic**	**UAutomizer**
Calendar	2.88	timeout	false	unknown	timeout	timeout
Cart	23.70	timeout	false	unknown	timeout	timeout
Hash Function	3.65	8.51	unknown	unknown	unknown	timeout
MinMax	5.14	timeout	false	unknown	timeout	timeout
Packet Filter	431.00	timeout	timeout	unknown	unknown	timeout
Process Queue	6.62	7.68	timeout	unknown	timeout	timeout
Quick Sort	18.20	3.50	timeout	unknown	unknown	5.75
Running Ex.	1.24	timeout	timeout	unknown	timeout	unknown
SM1	0.53	timeout	0.31	false	timeout	timeout
SM2	0.55	5.41	false	false	timeout	14.50

domains described in Section 5.2.3, we use the template polyhedra domain that is already a part of 2LS. Our approach handles any sequential C program, however, invariants are not inferred for array contents and memory manipulation using pointer arithmetic.

5.5 Experiments

We performed the experiments to show how our approach improves the performance of 2LS and also how it compares to other state-of-the-art software verification tools.[9] We used BenchExec [5] to run the experiments with time limit set to 900 s and memory limit to 15 GB. The first comparison was done on the subcategories of the SV-COMP benchmarks [47] related to memory safety, particularly *ReachSafety-ControlFlow*, *ReachSafety-Heap*, *MemSafety-Heap*, *MemSafety-LinkedLists*, *MemSafety-Others*. Tasks in *ReachSafety* are checked for reachability of an error condition, tasks in *MemSafety* for absence of invalid pointer dereference, invalid free, and memory leaks. We compared our implementation to the version of 2LS from SV-COMP'17 without the proposed shape analysis.

The results are shown in Table 5.1. The proposed method significantly improves the performance of the tool. Due to missing heap analysis support, the old version of 2LS often reported wrong results and therefore it had a negative score in three subcategories. 2LS with our analysis obtained a positive score in all subcategories and it is also faster in some of them.

Although the results show an improvement, we are still unable to compete with the best tools of SV-COMP'18 in the heap categories. This is mainly because our analysis does not yet support pointer arithmetic and is not yet expressive enough to handle various kinds of trees or nested lists.

However, the main purpose of our work was to extend possibilities of analysing combined shape and value properties of programs. To evaluate, we performed an experiment comparing our tool with the leaders of SV-COMP'18 in the heap-related categories, on tasks combining manipulation of unbounded data structures with a need to reason about the data stored in these structures. All these tasks[10] are correct programs created by our

[9]All tools, benchmarks, and results are available here: https://pschrammel.bitbucket.io/schrammel-it/research/2ls/fmcad18_exp.tar.xz.

[10]See https://github.com/diffblue/2ls/tree/2ls-0.7/regression/heap-data.

team, since no such programs are part of the SV-COMP benchmarks yet. For each task, we verify that no error state is reachable. The results of the evaluation are shown in Table 5.2. Numbers in the table represent CPU time in seconds needed for the analysis of the example. The value *unknown* means that the tool was not able to analyse the task.

On these benchmarks, 2LS outperforms the other tools significantly. Even tools specialised in shape analysis, *Forester* [25] and *Predator* [20], often report unknown, timeout or even find a false error. This is probably caused by their inability to reason about the data stored in the lists. More general tools such as *Symbiotic* [12] or *Ultimate Automizer* [26] often time out since they probably lack an efficient abstraction for combination of shape and value properties. *CPAChecker* [4] (in the *CPA-Seq* configuration from SV-COMP'18) solved four tasks but times out on the rest.

Chapter 6

Future Work

This work describes the research topic of the PhD thesis of the author which is a development of a new shape analysis method suitable for the context of program verification using template-based invariant inference, such as the one implemented in the 2LS framework. The first part of the text presents theoretical concepts of the underlying approach, outlines the current state-of-the-art of the shape analysis, and defines goals of the thesis.

The second part of the text presents the first results of our research. We have proposed a simple shape abstract domain capable of describing the state of the heap that can be easily combined with other analyses present in the 2LS framework. Experiments show that this combination allows us to analyse properties of programs that other analysers are not able to handle. However, this work still has a lot of limitations (e.g. see Section 5.3.3).

In the following work, we intend to extend the proposed domain so that it is able to better describe properties of various shapes on the heap, with main focus on the linked lists. Moreover, we want to explore further possibilities of domain combinations that could allow us to analyse complex properties of heap structures, such as pointer offsets, list sortedness, or relations between contents of dynamic data structures. Even though the main focus of our work will be on programs with various kinds of lists, we will also consider its generalisation to the much more complex case of trees and their various extensions.

To go even further, we intend to examine combination of shape analysis with less traditional analyses, e.g. with termination analysis or with resource bounds analyses or to explore verification of concurrent heap-manipulating programs which in the context of analysis in 2LS is a completely new area.

Bibliography

[1] Abdulla, P. A.; Holík, L.; Jonsson, B.; et al.: Verification of Heap Manipulating Programs with Ordered Data by Extended Forest Automata. In *Proceedings of the 11th International Symposium on Automated Technology for Verification and Analysis*, *LNCS*, vol. 8172. Springer. 2013. pp. 224–239.

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

[3] Berdine, J.; Cook, B.; Ishtiaq, S.: SLAyer: Memory Safety for Systems-Level Code. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, *LNCS*, vol. 6806. Springer. 2011. pp. 178–183.

[4] Beyer, D.; Keremoglu, M. E.: CPAchecker: A Tool for Configurable Software Verification. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, *LNCS*, vol. 6086. Springer. 2011. pp. 184–190.

[5] Beyer, D.; Löwe, S.; Wendler, P.: Benchmarking and Resource Measurement. In *Proceedings of the 22nd International SPIN Symposium on Model Checking of Software*, *LNCS*, vol. 9232. Springer. 2015. pp. 160–178.

[6] Biere, A.: CaDiCaL, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2017. In *Proc. of SAT Competition 2017 – Solver and Benchmark Descriptions*. University of Helsinki. 2017. pp. 14–15.

[7] Biere, A.; Cimatti, A.; Clarke, E.; et al.: Symbolic Model Checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer. 1999. pp. 193–207.

[8] Bouajjani, A.; Habermehl, P.; Rogalewicz, A.; et al.: Abstract Regular Tree Model Checking of Complex Dynamic Data Structures. In *Proceedings of the 13th International Static Analysis Symposium*, *LNCS*, vol. 4134. Springer. 2006. pp. 52–70.

[9] Brain, M.; David, C.; Kroening, D.; et al.: Model and Proof Generation for Heap-Manipulating Programs. In *Proceedings of the 23rd European Symposium on Programming*. Springer. 2014. pp. 432–452.

[10] 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.

[11] Calcagno, C.; Distefano, D.; O'Hearn, P. W.; et al.: Compositional shape analysis by means of bi-abduction. In *Proceedings of the 36th ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM. 2009. pp. 289–300.

[12] Chalupa, M.; Vitovská, M.; Strejcek, J.: SYMBIOTIC 5: Boosted Instrumentation - (Competition Contribution). In *Proceedings of the 24th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, *LNCS*, vol. 10806. Springer. 2018. pp. 442–446.

[13] Chen, H.; David, C.; Kroening, D.; et al.: Synthesising Interprocedural Bit-Precise Termination Proofs. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*. ACM. 2015. pp. 53–64.

[14] Chong, S.; ; Rugina, R.: Static Analysis of Accessed Regions in Recursive Data Structures. In *Proceedings of the 10th International Static Analysis Symposium*. Springer. 2003. pp. 463–482.

[15] Clarke, E.; Kroening, D.; Lerda, F.: A Tool for Checking ANSI-C Programs. In *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, *LNCS*, vol. 2988. Springer. 2004. pp. 168–176.

[16] Cousot, P.: Abstract Interpretation Based Formal Methods and Future Challenges. In *Informatics: 10 Years Back, 10 Years Ahead*, edited by R. Wilhelm. Berlin, Heidelberg: Springer. 2001. pp. 138–156.

[17] Cousot, P.; Cousot, R.: Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*. Dunod, Paris, France. 1976. pp. 106–130.

[18] 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 SIGPLAN Symposium on Principles of Programming Languages*. ACM. 1977. pp. 238–252.

[19] Cousot, P.; Cousot, R.: Systematic Design of Program Analysis Frameworks. In *Proceedings of the 6th ACM SIGPLAN Symposium on Principles of Programming Languages*. 1979. pp. 269–282.

[20] Dudka, K.; Peringer, P.; Vojnar, T.: Byte-Precise Verification of Low-Level List Manipulation. In *Proceedings of the 20th International Static Analysis Symposium*. Springer. 2013. pp. 215–237.

[21] Eén, N.; Sörensson, N.: An Extensible SAT-solver. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing*. Berlin, Heidelberg: Springer. 2004. pp. 502–518.

[22] Facebook: The Facebok Infer. 2019.
Retrieved from: https://fbinfer.com

[23] Free Software Foundation: GCC, the GNU Compiler Collection. 2019.
Retrieved from: https://gcc.gnu.org/

[24] Günther, H.; Weissenbacher, G.: Incremental Bounded Software Model Checking. In *Proceedings of the 2014 International SPIN Symposium on Model Checking of Software*. SPIN 2014. New York, NY, USA: ACM. 2014. pp. 40–47.

[25] Habermehl, P.; Holík, L.; Rogalewicz, A.; et al.: Forest Automata for Verification of Heap Manipulation. In *Proceedings of the 23rd International Conference on Computer Aided Verification*. Springer. 2011. pp. 424–440.

[26] Heizmann, M.; Chen, Y.; Dietsch, D.; et al.: Ultimate Automizer and the Search for Perfect Interpolants - (Competition Contribution). In *Proceedings of the 24th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 2018. pp. 447–451.

[27] Holík, L.; Hruška, M.; Lengál, O.; et al.: Counterexample Validation and Interpolation-Based Refinement for Forest Automata. In *Proceedings of the 18th International Conference on Verification, Model Checking, and Abstract Interpretation*, *LNCS*, vol. 10145. Springer. 2017. pp. 288–309.

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

[29] Itzhaky, S.; Banerjee, A.; Immerman, N.; et al.: Modular reasoning about heap paths via effectively propositional formulas. In *ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM. Proceedings of the 41st 2014. pp. 385–396.

[30] Itzhaky, S.; Bjørner, N.; Reps, T. W.; et al.: Property-Directed Shape Analysis. In *Proceedings of the 26th International Conference on Computer Aided Verification*, *LNCS*, vol. 8559. Springer. 2014. pp. 35–51.

[31] Jonkers, H. B. M.: Abstract storage structures. In *Algorithmic Languages*. IFIP. 1981. pp. 321–343.

[32] Kanvar, V.; Khedker, U. P.: Heap Abstractions for Static Analysis. *ACM Computing Surveys*. vol. 49, no. 2. 2016: pp. 29:1–29:47.

[33] Laviron, V.; Chang, B. E.; Rival, X.: Separating Shape Graphs. In *Proceedings of the 19th European Symposium on Programming*, *LNCS*, vol. 6012. Springer. 2010. pp. 387–406.

[34] Le, Q. L.; Sun, J.; Chin, W.-N.: Satisfiability Modulo Heap-Based Programs. In *Proceedings of the 28th International Conference on Computer Aided Verification*. 2016. pp. 382–404.

[35] LLVM project: The LLVM Compiler Infrastructure. 2019.
Retrieved from: http://llvm.org/

[36] Malík, V.; Hruska, M.; Schrammel, P.; et al.: Template-based verification of heap-manipulating programs. In *Proceedings of the 18th Formal Methods in Computer-Aided Design*. 2018. pp. 103–111.

[37] Matosevic, I.; Abdelrahman, T. S.: Efficient Bottom-up Heap Analysis for Symbolic Path-based Data Access Summaries. In *Proceedings of the International Symposium on Code Generation and Optimisation*. ACM. 2012. pp. 252–263.

[38] Møller, A.; Schwartzbach, M. I.: The Pointer Assertion Logic Engine. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM. 2001. pp. 221–231.

[39] de Moura, L.; Bjørner, N.: Z3: An Efficient SMT Solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer. 2008. pp. 337–340.

[40] Piskac, R.; Wies, T.; Zufferey, D.: Automating Separation Logic Using SMT. In *Proceedings of the 25th International Conference on Computer Aided Verification*, *LNCS*, vol. 8044. Springer. 2013. pp. 773–789.

[41] Reynolds, J. C.: Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings of the seventeenth Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society. 2002. pp. 55–74.

[42] Rinetzky, N.; Bauer, J.; Reps, T.; et al.: A semantics for procedure local heaps and its abstractions. In *Proceedings of the 32nd ACM SIGPLAN Symposium on Principles of Programming Languages*. 2005. pp. 296–309.

[43] Sagiv, M.; Reps, T.; Wilhelm, R.: Parametric Shape Analysis via 3-valued Logic. In *Proceedings of the 26th POPL*. ACM. 1999. pp. 105–118.

[44] Sankaranarayanan, S.; Sipma, H. B.; Manna, Z.: Scalable Analysis of Linear Systems Using Mathematical Programming. In *Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation*. Berlin, Heidelberg: Springer. 2005. pp. 25–41.

[45] 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.

[46] Sheeran, M.; Singh, S.; Stålmarck, G.: Checking Safety Properties Using Induction and a SAT-Solver. In *Proceedings of the 3rd Formal Methods in Computer-Aided Design*. Berlin, Heidelberg: Springer. 2000. pp. 127–144.

[47] Software Verification Competition: Benchmarks. 2017.
Retrieved from: `https://github.com/sosy-lab/sv-benchmarks/`

[48] Yang, H.; Lee, O.; Berdine, J.; et al.: Scalable Shape Analysis for Systems Code. In *Proceedings of the 20th International Conference on Computer Aided Verification*, *LNCS*, vol. 5123. Springer. 2008. pp. 385–398.