MŰEGYETEM 1782

Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Compiler Optimizations for Software Verification

TDK Thesis

Author:

Gyula Sallai

Advisors:

Tamás Tóth
Ákos Hajdu

2016

Contents

Kivonat

Ahogy a beágyazott rendszerek egyre inkább életünk szerves részévé válnak, biztonságos és hibamentes működésük egyre kritikusabb a felhasználók és a gyártók számára egyaránt. A tesztelési módszerekkel ellentétben a formális verifikációs technikák nem csak a hibák jelenlétét, hanem hiányát is képesek bizonyítani, ezáltal kiváló eszközzé téve őket biztonságkritikus rendszerek verifikációjához. Ennek egy módja a már elkészült forráskód formális modellé alakítása és a modell ellenőrzése a hibás állapotok bekövetkezhetősége szempontjából.

Sajnos a forráskódból formális modellt előállító eszközök gyakran állítanak elő kezelhetetlenül nagy és komplex modelleket, így téve ellenőrzésüket rendkívül bonyolulttá és időigényessé. Munkámban bemutatok egy olyan komplex folyamatot, amely képes forráskódból formális modellt előállítani. A folyamat részeként fordítótervezésben gyakran használt optimalizációs algoritmusokat (konstans propagálás, halott kódrészletek törlése, ciklus kihajtogatás, függvények „inline”-olása) alkalmazok a bemeneten, illetve egy program szelető algoritmus segítségével több egyszerűsített modellt állítok elő egy nagy probléma helyett. Az optimalizációk hatékonyságát és hatását mérésekkel demonstrálom.

Abstract

As embedded systems are becoming more and more common in our lives, the importance of their safe and fault-free operation is becoming even more critical. Unlike testing, formal verification can not only prove the presence of errors, but their absence as well, making it suitable for verifying safety-critical systems. Formal verification may be done by transforming the already implemented source code to a formal model and querying the resulting model's properties on the reachability of an erroneous state.

Sadly, source code to formal model transformations often yield unmanageably large and complex models, resulting in an extremely high computational time for the verifier. In this work I propose a complex workflow which provides a source code to formal model transformation, with program size reduction optimizations. These optimizations include constant propagation, dead branch elimination, loop unrolling, function inlining, extended with a program slicing algorithm which splits a single large problem into multiple smaller ones. Furthermore, I provide benchmarks to demonstrate the usability and efficiency of this workflow and the effect of the optimization algorithms on the formal model.

Chapter 1

Introduction

Starting from the end of the late century, modern human society has an over increasing reliance on embedded computer systems. These systems are now present in almost every aspect of our lives: we have them in our washing machines, cars and medical equipment, etc. As the reliance upon them grows, we also have a greater need to prove their fault-free behavior, as even the smallest erroneous operation can cause a considerable damage to property or – in a worse case – human lives. Several otherwise preventable accidents can be credited to the lack of sufficient testing and verification. With these in mind we have the natural desire for a reliable, mathematically precise proof regarding the system's proper operation.

For satisfactory verification of a system, we can use a modeling formalism with formal semantics to model the system's behavior and query that model's properties. Such queries usually target reachability (e.g. whether an erroneous state can be reached). However, designing and defining a model for a project can be rather difficult and in many cases the financial and time constraints do not make it possible. Many projects start right at the implementation phase without sufficient planning and modeling. In the domain of embedded systems, implementation is usually done in C. While there are many tools that can be used to generate C code from a formal model, the reverse transformation (model from a source code) is far less supported.

Another difficulty with this process is the size of the state space of the model generated from the source code. As most verification algorithms have a rather demanding computational complexity (usually operating in exponential time), the resulting model may not admit efficient verification. A way to resolve this issue is to reduce the size of the generated model. During program-to-model transformation this can be done by applying some optimization passes on the input program to simplify it, thus simplifying the model output as well.

The project presented in this work proposes a transformation workflow from C programs to a formal model, known as control flow automaton. The workflow enhances this transformation procedure by applying some common optimization transformations used in compiler design. These optimizations are constant propagation, dead branch elimination, loop unrolling, and function inlining. Their application results in a simpler model, which is then split into several smaller, more easily verifiable chunks using the so-called program slicing technique. This allows the model checker to handle multiple small problems, instead of a single large one. For evaluation reasons, a simple bounded model checker algorithm was implemented as a verifier.

Chapter 2 offers some background information on model checking and data dependency analysis structures commonly used in compiler design, such as use-define chains, program dependence graphs, dominator trees, and call graphs. Chapter 3 describes a verification compiler, that is a compiler built for software verification support. This requires discussing the construction of the previously mentioned dependency structures and the implemented transformation algorithms. Chapter 4 discusses the implementation details of this compiler and the verifier. Chapter 5 evaluates and measures the effect of these transformations on some verification tasks from the Competition on Software Verification. Finally, Chapter 6 concludes my work and offers some possible extensions for future improvement.

Chapter 2

Background

This chapter gives a brief introduction to the theoretical background of the algorithms and structures used later in this work. Section 2.1 describes the theory of the formal verification techniques applied and Section 2.2 discusses some common formal representations of programs suitable for dependency and control flow analysis.

2.1 Formal verification

Formal verification is an approach for program verification. Unlike other methods, formal verification techniques are not only able to prove the presence of errors, but are capable of proving their absence as well. This section presents the theoretical background of the tools and structures used for formal verification of programs. Section 2.1.1 offers a description of the formal model used throughout this work, Section 2.1.2 describes bounded model checking, a well-known algorithm for formal verification of programs.

2.1.1 Control flow automata

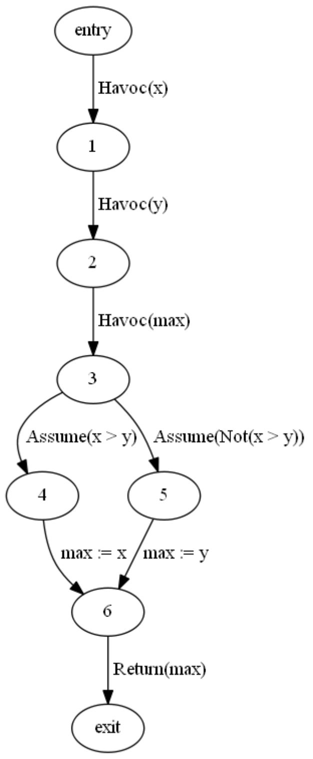
Control flow automata [4] aim to formally model program flow. The formal semantics and the automaton-like description makes the formalism suitable for formal verification.

Definition 1 (Control flow automaton). A *control flow automaton* (CFA) is a 4-tuple (L, E, ℓ_0, ℓ_q) where

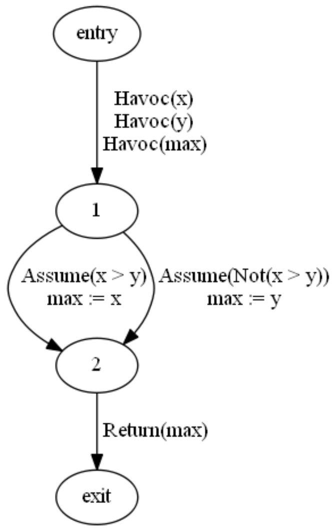
- $L = \{\ell_0, \ell_1, \dots, \ell_n\}$ is a set of locations, representing values of the program counter,
- $E \subseteq L \times \mathit{Ops} \times L$ is a set of edges, representing control flow, with the operations performed when a particular path is taken during execution,
- ℓ_0 is the distinguished entry location,
- ℓ_q is the designated exit location. ▪

There are two common ways to encode control flow in a CFA. In the case of *single-block encoding* the CFA contains a location for every value of the program counter. In the case of *large-block encoding*, a block of sequential execution paths are merged into a single edge, resulting in a smaller graph.

Example 1. *The control flow automaton shown in Figure 2.1a models the behavior of the max function. It represents each program counter value in its own location, thus it is single-block encoded. Figure 2.1 shows the same model with large-block encoding. The operation Havoc(X) means that the variable X is assigned to a nondeterministic value. The operation Assume(φ) means that a branching decision was taken with φ being the boolean expression satisfying the branch condition of the given path.*



(a) A single-block encoded CFA.



(b) A large-block encoded CFA.

Figure 2.1: A single-block encoded and a large-block encoded CFA.

2.1.2 Model checking

Model checking is a technique used for automatically proving a certain property of a formal model by systematically exploring its state space [14]. In our case, the formal model is a control flow automaton and the required property is the unreachability of an erroneous state. This notion can be formalized by inserting a special location into the control flow automaton, called the *error location* ℓ_e. Thus an instance of the *model checking problem* is a pair (A, ℓ_e) where $A = (L, E, \ell_0, \ell_q)$ is a control flow automaton and $\ell_e \in L$ is the designated *error* location. The problem targets whether there exists an executable program path π from ℓ_0 to ℓ_e in A. If such path exists, it is a counterexample for the property that can be reported to the programmer, otherwise the property holds.

Checking for the graph-theoretical reachability of the error location yields false-positive results. It might be possible that a path $l_0 \implies l_e$ exists, however, it is not evident whether that erroneous control flow path can be executed for any inputs. Therefore model checking requires checking the semantic interpretation of the input CFA and only reporting a counterexample if it is actually a viable error path.

There are various approaches for software verification. Abstraction based model checkers operate by ”hiding” certain details of the program. If the abstraction fails to model the original system precisely (by hiding too much information), then the abstraction is refined, until it faithfully represents the original input model [12]. Several software verification tools use this approach, such as CPAChecker [6], BLAST [22], SLAB [9] and UFO [2]. Partial order reduction [19] is mostly used for verifying concurrent programs. If a program runs multiple threads whose paths to a certain state overlap in some order, then many of these orderings can be represented with only one of them. Abstract interpretation methods [16] prove the unreachability of a certain state (in our case, the error state), by iteratively extending an over-approximation of the set of reachable states. Bounded model checkers [7], such as CBMC [13] use an approach which searches for error paths within a given maximal length and reduces them to mathematical formulas. From these approaches, I focus on bounded model checking in my work, which is presented in the following sections.

Mathematical logic in model checking

Propositional logic is a branch of mathematical logic. The basic elements of the logic are *propositional variables* (e.g. P and Q). A *formula* φ is constructed from propositional variables and *logical connectives* such as $\top$ (*true*), $\bot$ (*false*), $\neg$ (*negation*), $\wedge$ (*conjunction*), $\vee$ (*disjunction*) and $\rightarrow$ (*implication*). An *interpretation* assigns a truth value to every propositional variable.

The *boolean satisfiability problem* (SAT) is the problem of deciding whether a formula φ is satisfiable, i.e. whether there is an interpretation which yields that φ is true. Despite the problem's NP-completeness [15], modern SAT solvers in cases can handle models with millions of formulas [23]. However, describing a computer program using merely propositional variables and truth symbols is a rather complicated issue, resulting in impractically large models.

First order logic (FOL) extends propositional logic with *variables*, *function symbols*, *predicate symbols* and *quantifiers*. A first order interpretation is based on a *domain*, which is a set that may contain any abstract objects (such as numbers, animals or teapots). Function symbols are interpreted as functions over the domain, and predicate symbols are interpreted as relations over the domain. First order logic is undecidable [11, 29].

However, decidability can be achieved by semantically restricting first order logic to a class of interpretations. The *satisfiability modulo theories* (SMT) problem [8] is the problem of deciding whether a first order logic formula φ is satisfiable in a combination of certain (usually quantifier free) theories. In our case, the theory (and interpretation) used is the theory of integers, which interprets $+$ as the well-known integer addition, $\leq$ as the usual total order over integers, etc. Besides often being decidable, first order theories enable reasoning over data structures commonly used in computer programs (e.g. arrays), thus are convenient to describe program semantics.

Bounded model checking

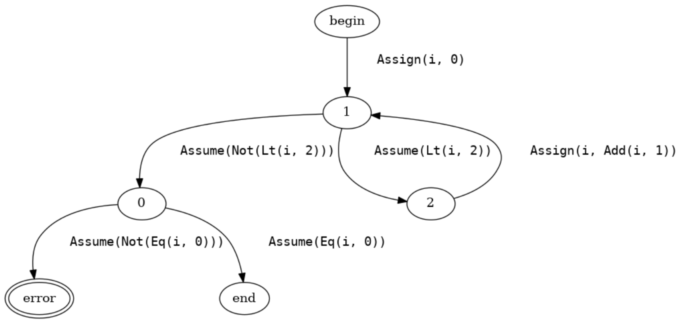
A bounded model checker traverses the state space, searching for an error path with the length of n (the bound – hence the name *bounded* model checker), starting from 0. If such path is present, its feasability is checked by the SMT solver.

In order to do this, the path has to be reduced to an SMT problem. In particular, assignments need to replaced by a equivalence predicates. This requires having different variables for each occurrences of a program variable on the left-hand side. This can be done tracking indices for our variables and incrementing it every time a new assignment to that variable is encountered. References to the previous value of the variable are resolved by referencing to the lower index version that variable.

If the solver finds that the SMT problem is satisfiable, then the error path is feasible, thus the input program contains a faulty behavior. In that case, the program path serves as a counterexample an the analysis terminates.

In case the problem is unsatisfiable, the next error path can be analyzed. If no more error paths exist with the length of n, then the model checker raises the bound to $n + 1$. If the bound exceeds k, the algorithm terminates.

Example 2. *Consider the input CFA in Figure 2.2a. The automaton describes a program which increases the variable i's value until it reaches* 2*. On the other hand, the assertion at the end of the program requires i to be equal to* 0*. Figure 2.2b shows the SMT representation of the error path* ($begin \rightarrow 1 \rightarrow 2 \rightarrow 1 \rightarrow 2 \rightarrow 1 \rightarrow 0 \rightarrow error$).



(a) A control flow automaton.

$$
\begin{gathered}
i_1 = 0 \\
i_1 < 2 \\
i_2 = i_1 + 1 \\
i_2 < 2 \\
i_3 = i_2 + 1 \\
\neg(i_3 < 2) \\
\neg(i_3 = 0)
\end{gathered}
$$

(b) The SMT representation of the 7 length error path in **a**.

Figure 2.2: An example control flow graph.

Generally, bounded model checking is not complete: it may not find all possible error paths (due to the maximum bound). On the other hand, a bounded model checker can be sound, that is, it does not report false-positives. This however requires the faithful (i.e. bit-precise) encoding of the program.

An overview of the bounded model checking algorithm can be seen in Algorithm 1.

2.2 Program representations

This sections describes some data structures suitable for representing a program. Each presented structure has a different purpose: some of them are convenient tools for language agnostic program representation, others merely describe dependency relations of a given program.

2.2.1 The C language

The C language was designed in 1970s by Dennis Ritchie at AT&T Bell Labs. Despite its age, it is still the most widely used programming language in the world of embedded

Algorithm 1: Bounded Model Checking on Control Flow Automatons.

```
Input: A control flow automaton $(L, E, l_0, l_q)$
Input: An error location $\ell_e \in L$
Input: A maximum bound $k \geq 0$
Output: A counterexample CEX($\pi$) or UNKNOWN
1  $n := 0$
2  while $n \leq k$ do
3      foreach error path $\pi = (\ell_0 \Longrightarrow \ell_e)$ such that $|\pi| = n$ do
4          $\varphi :=$ the SMT representation of $\pi$
5          if $\varphi$ is satisfiable then
6              return CEX($\pi$)
7          end
8      end
9      $n := n + 1$
10 end
11 return UNKNOWN
```

systems[1]. Although it was never designed for embedded use, C's simple syntax, relatively rich standard library and low-level architecture made its position dominant in systems programming, with compiler support for almost every microprocessor architecture.

2.2.2 Abstract syntax trees

In order to easily analyze the syntax of a parsed source code, compilers usually build a graph (more precisely a tree) describing the syntactic structure of the given program [1]. In this tree each instruction and expression is represented by a node, and a node's children are its subexpressions or contained instructions. The resulting graph is called *Abstract Syntax Tree*, *AST* for short. The corresponding AST for the code shown in Listing 2.1 can be seen in Figure 2.3.

```c
int main(void) {
    int x = 2;

    for (int i = 0; i < 3; i = i + 1) {
        x = x + 1;
    }

    assert(x == 5);

    return 0;
}
```

Listing 2.1: A simple C program.

2.2.3 Control flow graphs

Control flow graphs (*CFG* for short) [3] are language-agnostic intermediate representations of computer programs. Almost every compiler uses them, and they are the main tools for compiler optimizations. In order to reduce the graph's size and to create a safe window for local optimizations, it is usual to merge linear instructions into a single entity, called

[1] http://spectrum.ieee.org/static/interactive-the-top-programming-languages-2016

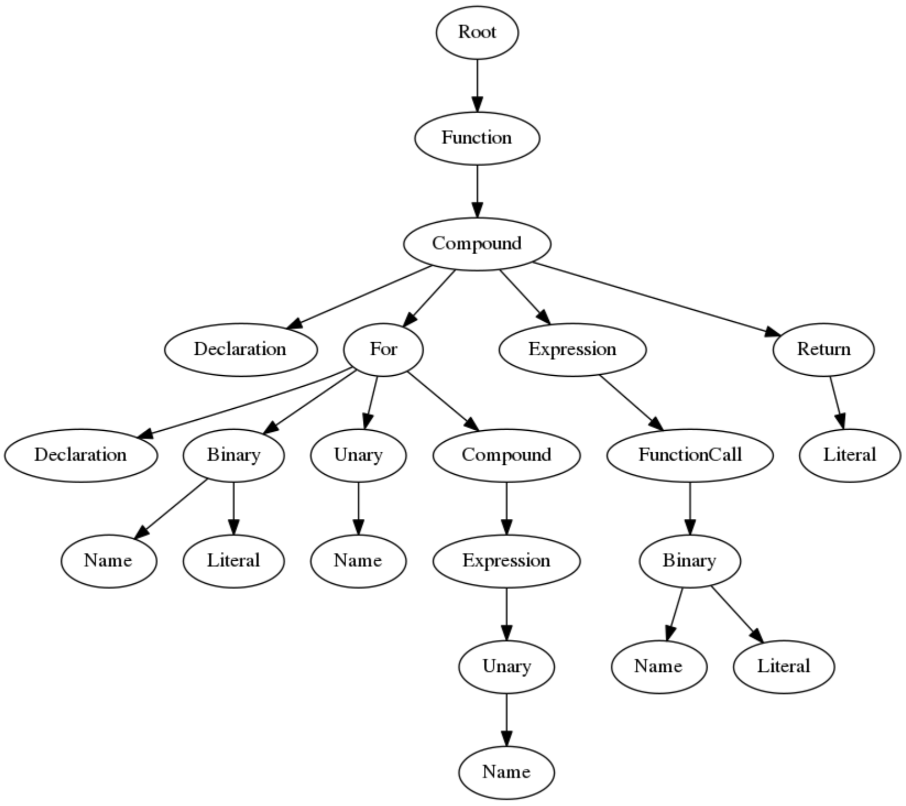


Figure 2.3: AST representation of Listing 2.1.

the *basic block*. A basic block may only contain sequential instructions of a program and have only one entry point (the first instruction) and one exit point (the last instruction). However, it may have multiple predecessors and successors, and may even be its own successor.

Definition 2 (Control flow graph). A *control flow graph* (CFG) is a tuple (B, E, b_e, b_q), where

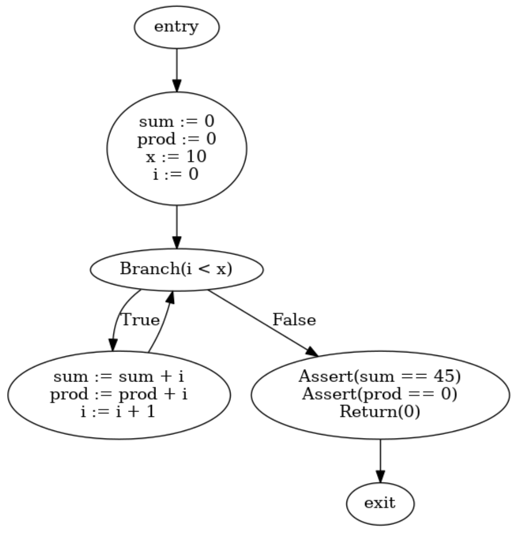
- $B = \{b_1, b_2, ..., b_n\}$ is a set of basic blocks,
- $E = \{(b_i, b_j), (b_k, b_l), ...\}$ is a set of directed edges, each representing a possible execution path in the program,
- the marked $b_e \in B$ is the special entry block, which all execution paths must begin from, and
- $b_q \in B$ is the special exit block, which must be pointed to by edges from terminating blocks. ▪

Throughout this work, for a control flow graph G, the notation $\mathsf{pred}(G)$ marks the set of G's predecessors, $\mathsf{succ}(G)$ marks the set of G's successors.

Example 3. *An example is shown in Figure 2.4. Figure 2.4a shows a simple C program, Figure 2.4b shows its corresponding control flow graph. Notice the edges on the labels: in many cases, it is useful to augment the edges, and store information on the reason why a particular control path was chosen. In this case, the labels show how a path will be taken depending on the result of the branching comparison.*

```c
int main(void) {
    int sum = 0;
    int prod = 0;
    int x = 10;
    int i = 0;

    while (i < x) {
        sum = sum + i;
        prod = prod * i;

        i = i + 1;
    }

    assert(sum == 45);
    assert(prod == 0);

    return 0;
}

```

(a) A simple C program.



(b) The control flow graph of the program shown in **(a)**.

Figure 2.4: An example control flow graph.

2.2.4 Flow graph dominators

It may not be a trivial task to recognize the control structures present within a control flow graph, as it requires the recognition of several patterns, which may span through multiple blocks. Luckily, the theory of flow graph dominators offers structures for easier recognition of loops and branches [1]. The dominator relation shows us which instructions (nodes) will always be executed before reaching to a certain point of the input program. This information will later be used to find control dependency relations.

Definition 3 (Dominator). Let P and Q be two nodes of a flow graph. P is said to *dominate* Q (denoted as $P \,\mathsf{dom}\, Q$) if all paths from the entry node to Q contains P. ▪

As it can be observed, according to this definition every node dominates itself, and the entry node dominates all nodes of the graph. If we wish to determine which instructions depend on the execution of a particular branch, we need to introduce the following definition.

Definition 4 (Post-dominator). Let P and Q be two nodes of a flow graph. P is said to *post-dominate* Q (denoted as $P \,\mathsf{pdom}\, Q$) if all paths from Q to the exit node contains P. ▪

The (post-) dominator relation can be organized into a structure which allows us to perform easy and efficient queries on the dominator information. This structure is a tree, called the *dominator tree*. In order to build this tree, we will need to the following definitions.

Definition 5 (Immediate dominator). Let P and Q be two nodes of a flow graph. P *immediately dominates* Q (denoted as $P \,\mathsf{idom}\, Q$) if and only if:

- $P \,\mathsf{dom}\, Q$,
- every other dominator of Q dominates P. ▪

Definition 6 (Immediate post-dominator). Let P and Q be two nodes of a flow graph. P *immediately post-dominates* Q (denoted as $P\ \mathsf{ipdom}\ Q$) if and only if:

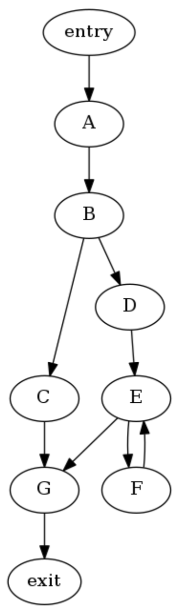
- $P\ \mathsf{pdom}\ Q$,
- every other post-dominator of Q post-dominates P. ▪

The above definitions imply that a node may only have a single (post-) dominator. This observation allows the (post-) dominator tree to be indeed a tree.

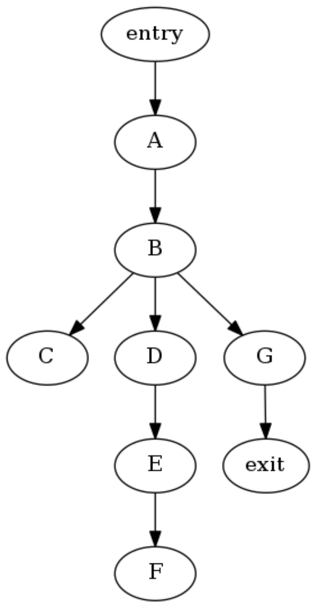
Definition 7 (Dominator tree). Let G be a flow graph, with a vertex set of V. A dominator tree is a graph with the vertex set V, and includes the edge $P \to Q$ between the nodes P, Q if and only if $P\ \mathsf{idom}\ Q$. ▪

Definition 8 (Post-dominator tree). Let G be a flow graph, with a vertex set of V. A post-dominator tree is a graph with the vertex set V, and includes the edge $P \to Q$ between the nodes P, Q if and only if $P\ \mathsf{ipdom}\ Q$. ▪

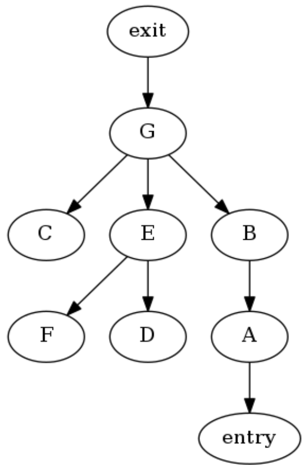
Example 4. *Consider the flow graph shown in Figure 2.5a. Figure 2.5b shows its dominator tree, Figure 2.5c shows its post-dominator tree. As it can be observed, if node Q is a descendant of node P, in the (post-) dominator tree, then P* dom *Q (P* pdom *Q) holds.*



(a) A flow graph.



(b) The dominator tree of **a**.



(c) The post-dominator tree of **a**.

Figure 2.5: An example on (post-) dominator trees.

2.2.5 Use-define chains

During optimization, it is often useful to know whether an instruction writes variables later read by another instruction. This relation is captured by the definition of flow dependency [21, 27].

Definition 9 (Flow dependency). Let G be an input program. Let P and Q be instructions in G.

1. Instruction P *unambiguously defines* variable V if P assigns a value to V.
2. Instruction P *ambiguously defines* variable V if P performs an operation which may or may not change the value of V.
3. Q *uses* V if Q reads the value of V.

If P ambiguously or unambiguously defines V and Q uses V afterwards, without V being redefined by another instruction, then Q *flow depends* on P. ▪

An example on ambiguous definitions is a function call with side-effects. As we need to make safe and conservative decisions during optimization, we must also treat ambiguous definitions as if they were unambiguous.

Definition 10 (Definition, use). Let G be an input program. Let I be the set of instructions in G. Let $P, Q \in I$. Let V be a variable in G.

1. The pair $d = (P, V)$ is a *definition* in G, if P ambiguously or unambiguously defines V. In this case, instruction P *generates* d.
2. The pair $u = (Q, V)$ is a *use* in G, if Q uses V.
3. Instruction P *kills* the definition d if $d = (X, V)$ where $X \in I, X \neq P$, and P unambiguously defines V. ▪

In order to find the flow dependency relation, we need to satisfy the restriction "without being redefined" of Definition 9. A redefinition of a variable occurs when an instruction after the original definition but prior the use defines the same variable. Such redefinitions kill the original definition, making it unnecessary to include them in the flow dependency relation. However, if a redefinition is done after a branching decision, we must make sure to check the other branching paths for redefinitions. If not all branching paths of a branch contains a redefinition, then the effects original definition may be just as valid. Furthermore, if instruction P only ambiguously defines V, then we may not be certain if the effects of the original definition were invalidated, thus we must place a restriction on definition killing not to include unambiguous definitions.

Definition 11 (Reaching definition). Let G be an input program. Let $d = (P, V)$ be a definition, $u = (Q, V)$ be a use in G. The definition d *reaches* u if there is a control flow path between P and Q where d is not killed along that path. ▪

Definition 11 formalizes the notion of "without being redefined by another instruction" in Definition 9. If there exists a definition $d = (P, V)$ that is a reaching definition of $u = (Q, V)$, then Q flow depends on P [18]. With this knowledge, we can build a structure containing every reaching definition of every use in our program. Such structure is called a use-define chain [1].

Definition 12 (Use-define chain). Let G be a program. Let $D = \{d_1, d_2, \ldots, d_k\}$ be the set of P's definitions, and $U = \{u_1, u_2, \ldots, u_n\}$ the set of its uses. The *use-define chain* of P is a set of pairs $\{(u_1, D_1), (u_2, D_2), \ldots, (u_n, D_n)\}$, where $D_i \subseteq D$ is the set of definitions reaching u_i. ▪

Example 5. *Consider the subgraph describing a code snippet shown in Figure 2.6. The definition set of the snippet is* $D = \{(i_1, x), (i_2, y), (i_4, y), (i_5, u)\}$*. The use set is* $U = \{(i_2, x), (i_3, x), (i_5, x), (i_5, y)\}$*. The use-define information of Figure 2.6 is*

$$(i_2, x) \to \emptyset,$$
$$(i_3, x) \to \{(i_1, x)\},$$
$$(i_5, x) \to \{(i_1, x)\},$$
$$(i_5, y) \to \{(i_2, y), (i_4, y)\}.$$

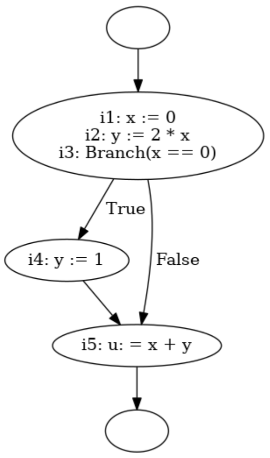


Figure 2.6: A control flow graph.

2.2.6 Program dependence graphs

A program dependence graph [18] is a program representation which explicitly shows data and control dependency relations between two nodes in a control flow graph. The control dependencies show if a branch decision in a node affects whether another instruction gets executed or not. Data dependencies tell which computations must be done in order to have all required arguments of an instruction. Program dependence graphs are constructs which allow easy and efficient querying on these properties.

In order to formalize the notion of control dependency, we shall define it using the theory of flow graph dominators, described in Section 2.2.4.

Definition 13 (Control dependency). Let G be a control flow graph. Let X and Y be nodes in G. Y is *control dependent* on X if and only if:

- there exists a directed path P from X to Y with any $Z \in P$ $(Z \neq X, Z \neq Y)$ post-dominated by Y $(Y \;\mathsf{pdom}\; Z)$,
- X is not post-dominated by Y. ▪

A program dependence graph is the result of the union of a *control dependence graph* and a *flow dependence graph*. Control dependence graphs contain the edge $X \to Y$ if Y is control dependent on X. Flow dependence graphs are merely the graph representations of the use-define information shown in Section 2.2.5, therefore they contain the edge $P \to Q$ if Q is flow dependent on P.

Definition 14 (Program dependence graph). A *program dependence graph* is a 3-tuple (V, C, F), where

- $V = \{v_1, v_2, \dots v_n\}$ is a set of instructions,

- $C \subseteq V \times V$ is a set of control dependency edges and
- $F \subseteq V \times V$ is a set of flow dependency edges.

The edge $(v_i, v_j) \in C$ if v_j is control dependent on v_i. The edge $(v_m, v_u) \in F$ if v_u is flow dependent on v_m. ▪

For practical and implementation reasons, the definition of control dependency is extended in this work with another special case: if a node X is not control dependent on any nodes according to Definition 13, it shall be control dependent on the entry node. This allows us to always treat connected graphs in the algorithms presented later on.

Example 6. *The program dependence graph of the program shown in Figure 2.4a can be seen in Figure 2.7. Solid lines represent control dependency, dashed lines represent flow dependency.*

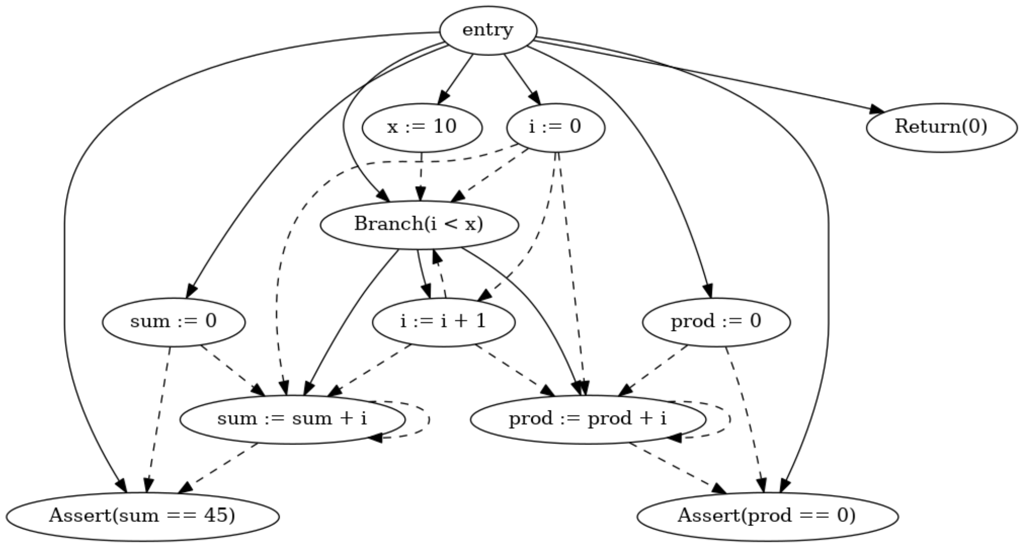


Figure 2.7: Program dependency graph of the program and CFG shown in Figure 2.4.

2.2.7 Call graphs

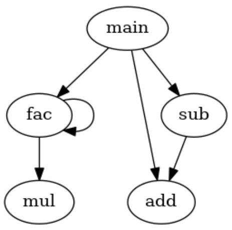
A call graph [28] is a program representation whose vertices represent program functions and its edges show whether a function calls another. This structure describes the interprocedural communication of a program and the basic relationships of its procedures. Naturally, this structure is not complete – it is rather complicated (if not practically impossible) to include the function calls in an externally defined function. However, describing the calling graph of a module and its internal functions provides a convenient structure for basic interprocedural analysis.

Definition 15 (Call graph). A *call graph* of a program G is a pair (P, E) where $P = \{p_1, p_2, \ldots, p_n\}$ is the set of G's procedures, and $E \subseteq P \times P$ is a set of directed edges. An edge $(p_u, p_v) \in E$ denotes that procedure p_u calls procedure p_v within G. ▪

Example 7. *The call graph of the program shown in Figure 2.8a can be seen in Figure 2.8b. Notice the* `fac` *function in the call graph: recursive functions reference themselves. If none of the included functions are recursive then the resulting call graph is directed acyclic graph.*

```c
int add(int x, int y) { return x + y; }
int sub(int x, int y) { return add(x, -y); }
int mul(int x, int y) { return x * y }
int fac(int x) {
    if (x == 0)
        return 1;

    return mul(x, fac(x - 1));
}

int main(void) {
    int sum = add(10, 20);
    int diff = sub(35, sum);
    int f = fac(diff);

    return 0;
}
```

(a) A C program using multiple functions.



(b) The call graph of the program shown in **(a)**.

Figure 2.8: A program and its call graph

2.3 Summary

This section summarizes the purpose of the program representations and dependency relations presented in the previous sections. Table 2.1 offers an overview of the presented structures.

Table 2.1: Summary of program representations.

Name	**Short description**	**Function**	**Section**
C source code	Widely used imperative programming language	The primary input format for the implemented program	2.2.1
Abstract syntax tree (AST)	Tree representation of the syntactic structure of a source code	Used for generating the control flow graph representation	2.2.2
Control flow graph (CFG)	A language-agnostic program representation	The basis of most transformations and syntactic analyses	2.2.3
(Post-) dominator tree (DT)	Structure for querying the dominator relation of a flow graph	Branch and loop recognition, control dependency description	2.2.4
Use-define chain (UD-chain)	A structure which contains all reaching definitions of a variable use	Data dependency description	2.2.5
Program dependency graph (PDG)	A structure which explicitly shows all control and data dependencies of a program	Overall dependency analysis of a program	2.2.6
Control flow automaton (CFA)	Language-agnostic control flow representation with formal automaton-like semantics	Used as an output of the transformation workflow and as the input of the verifier	2.1.1

Chapter 3

A Verification Compiler

The aim of the project presented in this work is to provide algorithms for transforming C source code to a formal model for verification. As real-life computer programs are rather complex and formal verification has a large computational complexity, it is desirable to simplify the resulting formal model. Such simplification can be done by applying some commonly used compiler optimization techniques on the input program. Such optimizations aim to reduce the program's size and complexity.

An overview of the presented workflow can be seen in Figure 3.1. The compiler receives a C source code as an input and at the end of the workflow, it outputs a control flow automaton, suitable for verification. The first step, shown in Section 3.1.1 is to parse the given C source code and produce an abstract syntax tree from it. Then the abstract syntax tree is transformed into a control flow graph, used for transformations and optimizations (Section 3.1.2). Finally, the control flow graph is transformed into a control flow automaton, as described in Section 3.1.7.

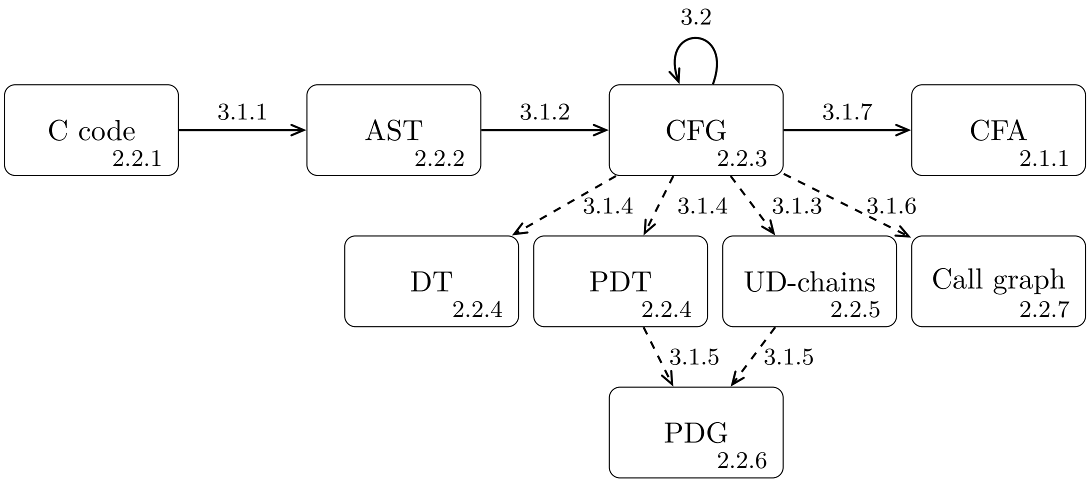


Figure 3.1: Transformation workflow.

Many transformation and optimization algorithms require some information about the program's dependency relations. While extracting these relations from the control flow graph is rather difficult, several helper structures (presented in Section 2.2) exist to allow easier querying on a program's properties. Most of these structures are built from the control flow graph, sometimes with the help of other dependency structures. Section 3.1.4 describes the building of dominator and post-dominator trees. Use-define information calculation is presented in Section 3.1.3. The program dependency graph from a program can be built by merging the control dependency and data dependency information available from post-dominator trees and use-define chains, this procedure is discussed in Section 3.1.5. Call graph construction is described in Section 3.1.6.

The optimizations presented in Section 3.2 aim to reduce the control flow graph's size and complexity. They all operate with control flow graphs being their input and output as well. A summary of presented transformations and their required helper structures (apart from the control flow graph) are shown in Table 3.1.

Table 3.1: A summary of the transformations presented in this work.

Optimization name	**Helper structures**	**Section**
Constant propagation	Use-define chains	3.2.1
Dead branch elimination		3.2.2
Loop unrolling	Dominator tree	3.2.4
Function inlining	Call graph	3.2.5
Program slicing	Program dependence graph	3.2.3

3.1 Building program representations

This section presents techniques and algorithms used to build the program representations introduced in Section 2.2.

3.1.1 C code parsing

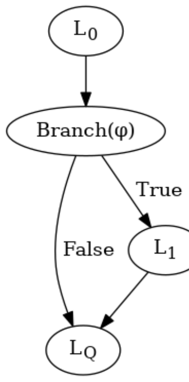
Source code parsing is the process of analyzing an input program's syntactic structure. Usually supported by a lexing phase, which transforms character sequences into tokens. A token is a character string with an assigned meaning. As an example, all the characters in an integer constant are grouped together in a single token, with the assigned meaning being the fact that they represent an integer literal. With this information, the parser can recognize certain patterns and structures following a set of rules. This set of rules is the program's syntax. As a program's syntax is described hierarchically, the recognized patterns can be arranged into a *parse tree*. The parse tree then can be simplified to an abstract syntax tree, introduced in Section 2.2.2.

Currently only a small subset of the C language is supported. The current implementation only allows the usage of control structures (such as **if-then-else**, **do-while**, **switch**, **while-do**, **break**, **continue**, **goto**) and non-recursive functions. Types are only restricted to integers and booleans. Arrays and pointers are not supported at the moment.

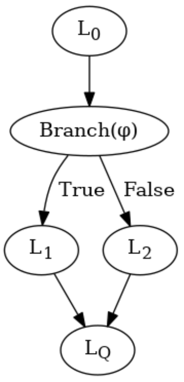
3.1.2 Building control flow graphs

Control flow graphs are built from the abstract syntax tree. For each control structure in the abstract syntax tree the CFG building algorithm will build a suitable flow graph representation and then append them together according to the control structure nesting in the input program. The transformation can be done by following a set of rules, each describing how a particular control structure can be transformed into its appropriate CFG representation. An overview of the transformation rules of common control structures is shown in Figure 3.2.

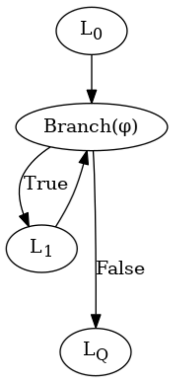
```
$L_0$
if ($\varphi$) {
    $L_1$
}
$L_Q$
```



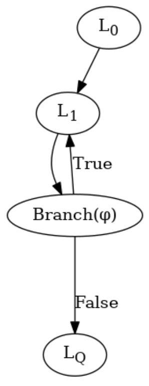
```
$L_0$
if ($\varphi$) {
    $L_1$
} else {
    $L_2$
}
$L_Q$
```



```
$L_0$
while ($\varphi$) {
    $L_1$
}
$L_Q$
```



```
$L_0$
do {
    $L_1$
} while ($\varphi$);
$L_Q$
```



```
$L_0$
switch ($X$) {
    case $\chi_1$: $L_1$ break;
    case $\chi_2$: $L_2$
    case $\chi_3$: $L_3$ break;
    case $\chi_4$:
    case $\chi_5$: $L_4$ break;
    case $\chi_6$: $L_5$ break;
    default: $L_6$ break;
}
$L_Q$
```

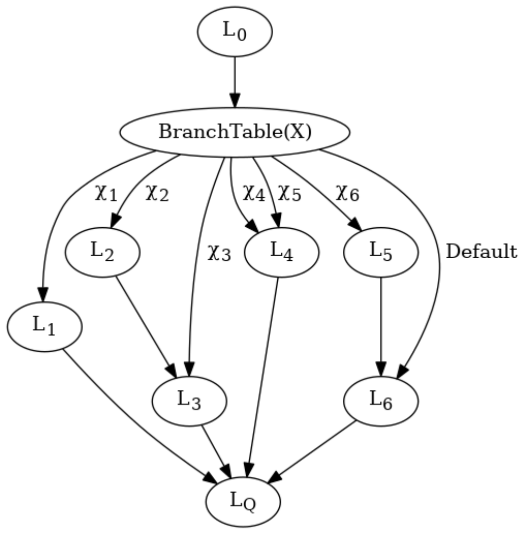


Figure 3.2: CFG construction rules.

For the sake of allowing a somewhat uniform and safe input CFGs for the transformation algorithms, the notion of *normalized control flow graph* is introduced. During implementation this prevents several erroneous input cases. A control flow graph is normalized if

1. its basic blocks are maximal (it does not contain sequences split through multiple blocks),

2. it has no branching statements with all edges pointing to the same node and
3. it does not contain unreachable blocks (in the graph theoretical sense).

3.1.3 Calculating use-define information

This section presents the algorithms used for computing use-define chains from control flow graphs.

Computing reaching definitions

The reaching definitions information can be calculated by solving the data flow equation

$$out[S] = gen[S] \cup (in[S] \setminus kill[S])$$

where $gen[S]$ is the set of definitions generated within the instruction S, $kill[S]$ is the set of definitions killed in S. The set $in[S]$ contains all definitions which are not killed before reaching S. The result set, $out[S]$ is a set of all definitions which are alive at the exit point of S [1].

For easier and faster computation, the reaching definitions problem is described here with basic blocks. Using basic blocks for this task saves memory and computation time, for the price of somewhat slower querying for local definitions.

In order to calculate $out[S]$ for a basic block S, we need to compute several sets.

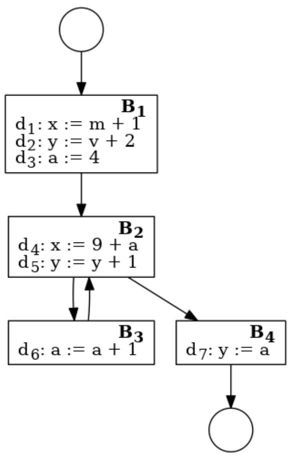
1. As $gen[S]$ is the set of all definitions generated within S, it is straightforward to fill this set up with the right values.
2. The set $kill[S]$ contains all definitions killed within the block S. If a definition $d = (P, V)$ is within the block S, all definitions of V which are not in S are killed within S.
3. The set $in[S]$ is the set of "alive" definitions reaching the entry point of S. Due to this definition, $in[S]$ is the union of the definitions arriving from all of S's predecessors.

Example 8. *Consider the control flow graph shown in Figure 3.3 with its already computed gen and kill sets. For the initial block (B_1), the set of its incoming definitions $in[B_1] = \varnothing$. As the set difference of $in[B_1]$ and $kill[B_1]$ is empty as well, the set of definitions leaving B_1 is $out[B_1] = gen[B_1] = \{d_1, d_2, d_3\}$.*

The next block B_2 redefines x and y, thus its kill set contains all definitions of x and y which are not in B_2. Its $in[B_2]$ set contains all the definitions coming from B_1, also the definition of a in B_3, meaning that the reaching definitions of B_2 are $in[B_2] = \{d_1, d_2, d_3, d_6\}$. The definitions leaving from B_2 are $out[B_2] = \{d_3, d_4, d_5, d_6\}$.

Following this logic, we can determine the $in[B]$ and $out[B]$ sets for each block B. Doing so yields the sets $in[B_3] = \{d_3, d_4, d_5, d_6\}$, $out[B_3] = \{d_4, d_5, d_6\}$, $in[B_4] = \{d_3, d_4, d_5, d_6\}$, and $out[B_4] = \{d_3, d_4, d_6, d_7\}$.

Algorithm 2 describes the computation of reaching definitions [1]. The algorithm computes the *in* and *out* sets for each basic block B, then propagates this information through the basic blocks. The algorithm halts if there are no more definitions to propagate, thus failing to change any *out* sets. As this algorithm propagates changes forward, better performance may be reached by arranging the blocks into depth-first order.



$$
\begin{aligned}
gen[B_1] &= \{d_1, d_2, d_3\}\\
kill[B_1] &= \{d_4, d_5, d_6, d_7\}\\[1em]
gen[B_2] &= \{d_4, d_5\}\\
kill[B_2] &= \{d_1, d_2, d_7\}\\[1em]
gen[B_3] &= \{d_6\}\\
kill[B_3] &= \{d_3\}\\[1em]
gen[B_4] &= \{d_7\}\\
kill[B_4] &= \{d_4, d_5\}
\end{aligned}
$$

Figure 3.3: A control flow graph and its *gen* and *kill* sets.

Algorithm 2: Reaching definitions.

```
Input: A control flow graph $G$ with $kill[B]$ and $gen[B]$ already calculated for each
       basic block $B$.
Output: The sets $in[B]$ and $out[B]$ for each basic block $B$.
1 for each block $B$ do $out[B] := gen[B]$;
2 while changes in any $out[B]$ set occur do
3     for each block $B$ do
4         $in[B] := \bigcup_{P \in \mathsf{pred}(B)} out[P]$
5         $out[B] := gen[B] \cup (in[B] \setminus kill[B])$
6     end
7 end
```

Local reaching definitions

While the reaching definition algorithms presented operate on basic blocks, it is often required to find reaching definitions for a particular instruction. Let B be a basic block, containing the instruction P. To find $in[P]$, we need to find $in[B]$ then walk along the (sequential) instructions within B, applying the appropriate data flow equation for each instruction.

Building use-define chains from reaching definitions information

After having $in[S]$ and $out[S]$ computed for each statement S, we can transform this information into use-define chains. For each use u_S in the input program, we find the D_S set of its reaching definitions. Naturally, $d \in D_S$ if and only if $d \in in[S]$.

3.1.4 Calculating dominator relation

The dominator relation is calculated by using a data flow equation somewhat similar to the one presented in Section 3.1.3. The data flow equation is based on the observation that if $p_1, p_2, \ldots, p_k$ are the predecessors of a block n, then node $d \neq n$ dominates n if and only if $d\,\mathsf{dom}\,p_i$ for every $1 \leq i \leq k$ [1]. This means that finding the set of n's dominators

$D(n)$ requires finding the dominator set $D(p_i)$ of every $p_i \in \mathsf{pred}(n)$. This can be described by the equation

$$D(n) = \{n\} \cup \bigcap_{p \in \mathsf{pred}(n)} D(p)$$

which means the every node's dominator set contains the node itself and the intersection of its predecessors' dominators. Solving this equation for all nodes yields the complete dominator relation.

The algorithm begins by setting an initial approximation for each node. For every node n, except the entry node n_0, the initial approximation is the whole node set. For the entry node n_0, the initial value is $D(n_0) = \{n_0\}$, as due to the dominator relation's definition the entry node is only dominated by itself. This will serve as a basis for further refinement. In each step, we refine the dominator relation by excluding nodes from $D(n)$ which do not dominate the node n's predecessors.

Algorithm 3: Dominator computing algorithm.

```
  Input: A flow graph $G$ with the vertex set $N$, and an entry node $n_0 \in N$
  Output: The dominator relation $D(n)$ for each $n \in N$
1 $D(n_0) := \{n_0\}$
2 for each node $n \in N \setminus \{n_0\}$ do $D(n) := N$;
3 while changes in any $D(n)$ set occur do
4     for each node $n \in N \setminus \{n_0\}$ do
5         $D(n) := \{n\} \cup \bigcap_{p \in \mathsf{pred}(n)} D(p)$
6     end
7 end
```

Because in the algorithm the dominator information is propagated forward, it is beneficial performance-wise to arrange the nodes into depth-first order, just like in the case of the computation of reaching definitions.

In order to build the dominator tree from the dominator relation, we need to find for each node n its immediate dominator. Due to the definition of immediate dominators (Definition 5), this requires traversing the set $D(n)$ for each n, and finding the node d which is dominated by every node $v \in D(n) \setminus \{d, n\}$. When such a node is found, it is set to be the parent of n in the dominator tree.

While there are other, faster and more widely used algorithms available [25], the complexity of their implementation compared to the efficiency gained was deemed unfavorable.

Another issue arises when we wish to compute the post-dominator relation of a flow graph. Luckily, this can be easily nullified by finding the dominator relation of the reverse of the input graph (i.e. the graph with the same vertex set but with each edge pointing in the reverse direction).

Let G be a flow graph with the entry node n_0 and exit node n_q. Let G' be is G's reverse. If a node n is post-dominated by a node d in G, then all $n \to n_q$ paths must contain d. As the exit node of G' is n_0 and all paths are reversed, all $n_0 \to n$ paths will contain d, thus d will dominate n in G'. Using this observation, we can easily find the post-dominator relation of a graph, by merely using its reverse as the input for the algorithms presented above.

3.1.5 Building program dependence graphs

As discussed in Section 2.2.6, program dependence graphs are merely a union of a control dependence graph and a flow dependence graph. In order to build the PDG, we only need to build these two structures and merge them together.

A difficulty of this approach is the problem of the different vertex sets: the dominator relation and control dependence graphs can be built using basic blocks, but the flow dependency relation is only sensible in the context of instructions. In order to save computational time, the control dependency relation is calculated using basic blocks. After that calculation, the basic blocks of the input CFG are split into individual nodes for data flow analysis.

Ferrante and others [18] offer a simple and fast algorithm for control dependency calculation. Let S consist of all edges (A, B) in the control flow graph such that B is not an ancestor of A in the post-dominator tree (B may not post-dominate A). Let L denote the least common ancestor of A and B in the post-dominator tree. L may only be A, or A's parent in the post-dominator tree (see [18] for proof). This gives us two cases.

1. If L is the parent of A, then all nodes in the post-dominator tree on the path between L to B, including B, but not L, should be marked control dependent on A.
2. If $L = A$, then all nodes in the post-dominator tree on the path from A to B, including A and B, should be marked control dependent on A.

This means that given a (A, B) edge in the CFG, we can achieve the desired effect by traversing backwards in the post-dominator tree from B to A's parent, marking all nodes visited before A's parent as control dependent on A.

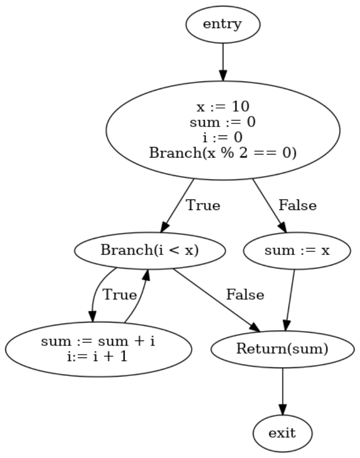
After finding the control dependency relation, the PDG building algorithm splits the basic blocks and reapplies the relation on the individual instructions. If a block Y (with the instruction set of $\{y_1, y_2, \ldots, y_n\}$) is control dependent on block X (containing the instructions $\{x_1, x_2, \ldots, x_t\}$), then all instructions $y_i \in Y$ will be control dependent on x_t. If a block was not control dependent on any blocks, the algorithm marks all of its instructions control dependent on the entry node.

Example 9. *Consider the control flow graph shown in Figure 3.4a. Its control dependency graph is shown in Figure 3.4b. The control dependency graph of the same program following the block splitting is shown in Figure 3.5.*

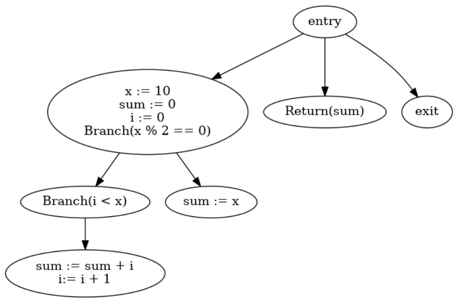
Following the construction of the control dependence graph, extending it with the required flow dependency edges is a simple matter. This requires querying the use-define information for each instruction and finding its reaching definitions. If an instruction X has a reaching definition for instruction Y, then we add a $X \to Y$ edge to the graph. Doing this for all instructions will yield the finished program dependency graph.

3.1.6 Call graph construction

Call graphs are built by the analysis of multiple control flow graphs and function declarations. The process takes a set of control flow graphs (representing function definitions) and function declarations as an input. This set will serve as the vertex set of the result call graph. The call graph construction algorithm examines every instruction of every function



(a) A control flow graph.



(b) The control dependency graph of **a**.

Figure 3.4: A control dependency graph example.

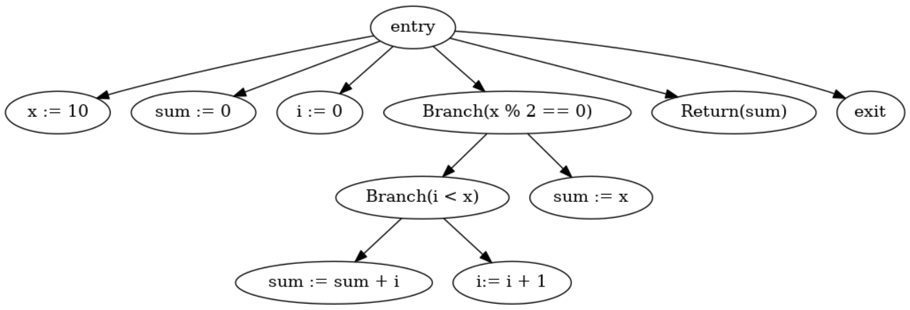


Figure 3.5: The control dependency graph in Figure 3.4, after block splitting.

definition, searching for function calls. If a call was found in a function A to a function B, an $A \to B$ edge is inserted into the call graph.

Naturally, this process cannot find function calls in a function which has no definition, furthermore it may not recognize calls to functions referenced by function pointers. This means that the resulting call graph might not represent every interprocedural call of the program.

3.1.7 Transforming a CFG to a CFA

At the end of the workflow, the (optimized) control flow graph is transformed into a control flow automaton. The transformation algorithm is rather straightforward: in the case of single-block encoding, each instruction is transformed into a location, with the instruction's effect (e.g. `x := 5`) being placed on the location's outgoing edges. Branching statements are transformed to locations with multiple outgoing edges, each marked with an **`assume`** statement, containing the satisfying branch condition formula.

Transformation to large-block encoding can be done by creating a location for every basic block of the program, and marking its outgoing edges with the appropriate statement lists.

3.2 Optimization algorithms

This section presents a list of optimization techniques commonly used by compilers and code analysis tools. Each optimization attempts to reduce the program size in order to simplify the formal model being used as the verifier input.

3.2.1 Constant propagation and folding

Constant propagation and folding are techniques present in almost every compiler. Constant folding provides a way to simplify every expression of the input program, while constant propagation allows the substitution of constants in place of variables with a known value.

Constant folding

Complex expressions with constant operands may be simplified during compile time by recognizing and evaluating them. Consider the following expression:

$$(120 - 15 \cdot 8) \cdot x$$

A constant folding algorithm traverses the tree describing this expression, computes and merges nodes with known values and replaces the entire expression with an equivalent but simpler one, in this case the literal 0. Figure 3.6 shows this process on the above example.

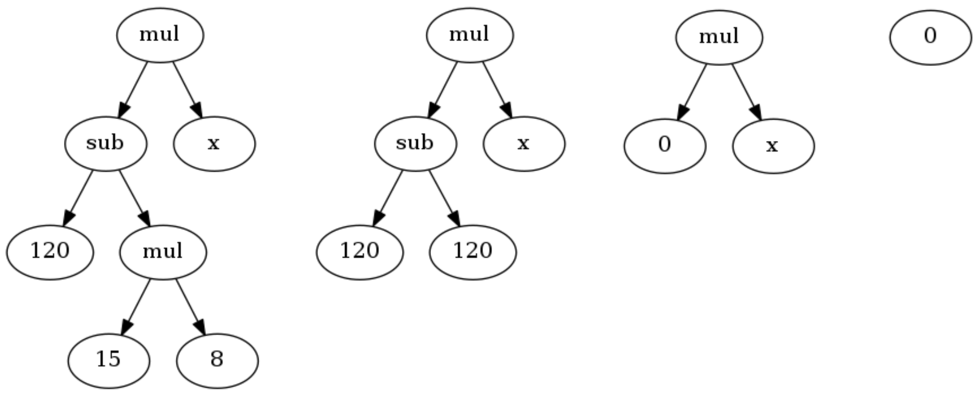


Figure 3.6: Constant folding example.

There are several rules which may be used to simplify the expression even further. For example, it is known that anything multiplied by zero results in zero. Some of the applied mathematical axioms are (where E denotes an expression with an unknown value):

$$E + 0 = 0 + E = E$$
$$E \cdot 0 = 0 \cdot E = 0$$
$$E \cdot 1 = 1 \cdot E = E$$
$$E \vee \textbf{true} = \textbf{true} \vee E = \textbf{true}$$
$$E \wedge \textbf{false} = \textbf{false} \wedge E = \textbf{false}$$
$$E/1 = E$$

Constant folding provides a way to simplify each expression in the program. Further analysis and constant propagation techniques can be applied to allow even greater simplification.

Local constant propagation

The aim of this transformation is to substitute constants in place of variables whose values are known during compile time. This requires determining if a variable has a constant value (i.e. yields the same value for every execution of the program), and replacing each occurrence of this variable with its constant value. Supported by constant folding, this optimization may significantly reduce the size of an expression. Furthermore, applying constant propagation for branching statements, it may even detect branches which could never be executed.

Example 10. *The code snippet shown in Figure 3.7a presents a primitive C function which could be improved by constant propagation. The result of the transformation (in conjunction with constant folding) can be seen in Figure 3.7b.*

```c
int calculate(int p) {
    int x = 13;
    int y = 9;
    int z = 3;

    return x * y - 39 * z + x * p;
}
```

(a) A simple C program snippet, with some constant variables.

```c
int calculate(int p) {
    int x = 13;
    int y = 9;
    int z = 3;

    return 13 * p;
}
```

(b) The program in **(a)**, after constant propagation and folding.

Figure 3.7: A constant propagation example.

Local constant propagation runs inside basic blocks only. Basic blocks provide safe windows for this optimization, because between a block's entry and exit points, only sequential instructions may change a variable's value. This makes it impossible for a variable to have an ambiguous value from multiple different sources. Local constant propagation thus only propagates between a particular block's entry and exit points.

To achieve this, the algorithm keeps track of each variable defined within the block in a table. If an assignment's right-hand side is a constant, we put its variable (the left-hand side) into the table along with the constant. If the right side is an expression of unknown value, we delete the defined variable from the table. With this structure, we can introduce new rules into the constant folding algorithm, telling it to replace occurrences of the variables contained in the table with their assigned constant value. Algorithm 4 offers a description of this procedure.

Global constant propagation

Global constant propagation allows folding rules to be propagated between basic blocks. This is not a trivial task: several assignments may be able to reach a single basic block and we must make sure to only propagate constant definitions which do not override other reaching definitions of the block being optimized.

This is achievable by utilizing the use-definition information described in Section 3.1.3. A global constant propagator enhances local propagation by querying the UD-chain for solely

Algorithm 4: Local Constant Propagation.

```
Input: A basic block $B$
Output: A revised basic block $B'$
for $i < B$ instruction count do
    $I := B[i]$
    if $I$ is an assignment then
        $L :=$ The variable defined by $I$
        $R :=$ The right-hand side of $I$

        if $R$ is constant then $M[L] := R$
        else $M[L] := \varnothing$
    end
    $B'[i] :=$ constant_fold($I$, $M$)
end
```

reaching constant assignments at the beginning of each block. For all such assignments we insert a new rule into the table containing the constant definition information.

3.2.2 Dead branch elimination

Dead branch elimination is the process of recognizing and deleting unreachable parts of code. Normally used in conjunction with constant propagation, dead branch elimination removes instruction and execution paths which were discovered as inviable. Such path maybe the *true* path of a branch comparison which is always failing.

In most compilers, dead code elimination consists of two slightly different procedures: dead instruction elimination and dead branch elimination. Dead instruction elimination is the process of finding *dead variables* which are written but never read again, then removing every instruction writing them. On the other hand, dead branch elimination removes whole control structures which were deemed unreachable. Dead instruction elimination is not required in this project as program slicing (presented in Section 3.2.3) will remove dead instructions from the code as a side-effect.

Dead branch elimination is done by iterating through all branching comparisons in the input program and finding those which have a literal as condition (for example: `Branch(False)`). Then the algorithm deletes all branching paths that cannot be taken according to the branch criteria constant.

3.2.3 Program slicing

Program slicing is a technique first described by Mark Weiser [30]. He suggested that while debugging a complex program, a programmer only pays attention to a smaller subset of the entire source code. This subset contains only the instructions and variables relevant to the problem being debugged. Attempting to formalize this practice, Weiser gave the following definition for program slices:

Definition 16 (Program slice). A *program slice* P' of a program P *with respect to the criteria* of (S, V) is an executable subset of P, producing the same output and assigning the same values to the variables $V = \{V_1, \ldots, V_n\}$ as the original program P in its statement S. ∎

```
 1  int i = 0;
 2  int sum = 0;
 3
 4  while (i < 11) {
 5      sum = sum + i;
 6      i = i + 1;
 7  }
 8
 9  printf("i=%d\n", i);
10  printf("sum=%d\n", sum);
11
```

(a) A simple C program snippet.

```
1  int i = 0;
2
3  while (i < 11) {
4      i = i + 1;
5  }
6
7  printf("i=%d\n", i);
8
```

(b) Slice of **(a)** with the criteria of $(9, \{i\})$.

Figure 3.8: A slicing example.

Example 11. *The example shown in Figure 3.8a presents a simple C code snippet which calculates the sum of the natural numbers less than* 10*, and then outputs it, followed by the output of the loop counter variable. A slice of this program with respect to the criteria of* $(9, i)$ *is shown in Figure 3.8b. As it can be observed, the only statements preserved are those relevant to the criteria output.*

During program size reduction, we shall use program slices to split the input program into smaller chunks, each being a slice for a single assertion in the input program. This will result in smaller verifiable problems instead of a single large one. Considering that the model checkers usually operate in exponential time, this may enable a reduction in verification running time.

Calculating slices using program dependence graphs

The problem of computing program slices can be represented by a reachability problem in the program dependence graph, as it was described by Ottenstein and Ottenstein [26]. The main idea is to find the criteria node in the PDG and walk backwards along the edges, finding all reachable nodes from this node. Nodes visited during this procedure should be included in the slice, all others can be disposed.

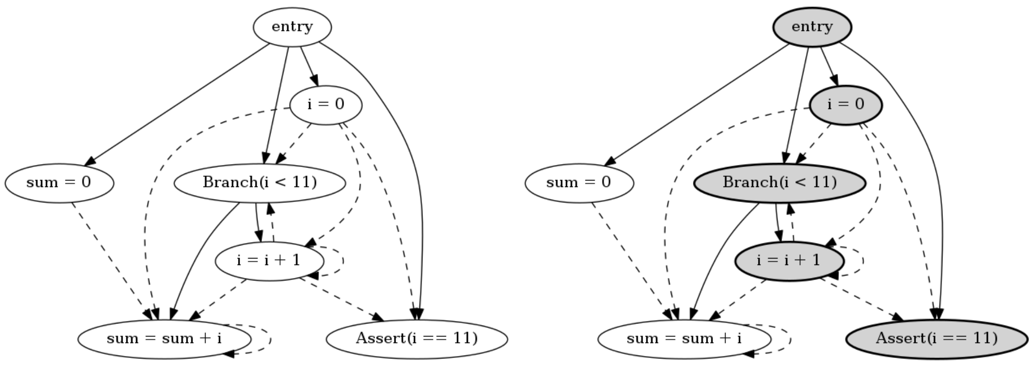


Figure 3.9: Slicing on the assert node using a program dependence graph. Solid lines represent control dependence, dashed lines show data dependence, filled nodes are those backwards reachable from the assert node in the PDG.

As the program dependence graph explicitly shows dependency relation of every node, this method will include all required nodes in the slice, while discarding all unambiguously irrelevant nodes. An example of this process can be seen in Figure 3.9. In this example, the slicer works with the criteria of the assert statement and its variable *i*. As this assertion makes no use of the value of the variable *sum*, all of its assignments are sliced away and are no longer present in the slice. Naturally, all assignments to i are needed and kept. The branch condition is also kept because the incremental assignment's execution depends on it.

Extracting slices from the control flow graph

The algorithm presented in the previous section allows us to mark instructions (nodes) in a control flow graph unneeded for a particular slice. However, due to the blocked structure of the control flow graphs used, additional measures are needed to extract the slice from the original CFG.

There are two cases for how the removal of a particular instruction i can effect the original control flow graph. If i is marked unneeded in its block B, but there are other instructions which are not marked like so, then i can be simply removed from B. If all of B's instruction were deemed irrelevant for the slice, then the whole block B need to be deleted.

Deleting a block requires rewiring all of their incoming edges into the targets of their outgoing edges. This is not always trivial: if a block marked for deletion is actually a branching one, while its parent is a simple jumping instruction, then this rewiring requires more effort than merely replacing the head of a particular edge.

To solve this problem, we shall search for entire unneeded *regions*, made up from the subgraph of multiple unneeded blocks. These regions are required to have exactly one exit point, and at least one entry point. After finding such a region, an algorithm rewires the entry points into the exit point and removes all blocks from this region. It repeats this procedure until it is capable of finding any regions.

These regions are found by placing all unneeded blocks into a queue and polling the queue each block a time. Then the algorithm traverses lastly polled block's children and parents. If a parent was not marked unneeded, then we shall call it an entry point. If a child was not marked unneeded, then we shall mark it as the exit point. There can be only one exit point for a connected subgraph of unneeded blocks: if such a subgraph S had connections to several needed blocks, then their execution would have depended on at least one block in S, meaning that needed blocks were control dependent on unneeded blocks, which is impossible.

3.2.4 Loop optimization

This section presents the method of loop recognition in a flow graph, and describes a loop optimization, known as loop unrolling.

Recognition of loops

For convenience, this work only discusses the recognition and optimization of *natural loops*. Natural loops are loop constructs which contain a header that dominates all nodes in the loop. Furthermore it contains at least one edge back to the header, called the *back edge*. In order to satisfy this definition, a loop must have only one entry point (its header), because otherwise the header would not dominate all nodes in the loop.

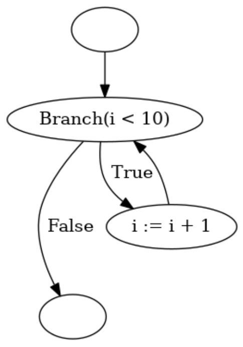
Most program loops in practice fall into the above category. It can be easily seen, that if a program only uses **if-then-else**, **while**, **do-while**, **continue**, and **break** statements, all loops in the program will be natural loops, because these control structures do not make jumping into the middle of a loop possible. Even most of the programs using **goto** statements conform to this restriction. Flow graphs which do not contain jumps into the middle of loops are called *reducible flow graphs* [20].

As natural loops can only contain one header and must contain at least one back edge, recognizing them requires finding edges whose heads dominate their tails. An edge $n \to h$, is a back edge if $h \,\mathsf{dom}\, n$. In that case n is the header of the loop. Determining whether a node is a loop header is done by examining whether it dominates one of its predecessors. Finding all nodes in a loop (with the header h) can be done by searching for all nodes v which are dominated by h, but also can reach h (i.e. there is a directed path from v to h).

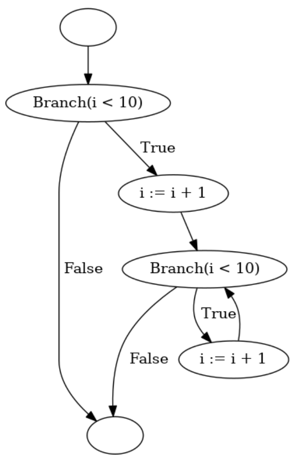
Loop unrolling

Loop unrolling is a technique for partially eliminating loops from a control flow graphs. As loops introduce a great complexity into the program, their (partial) elimination may reduce the model checker's running time, even at the cost of increased input size. For a loop with the entry condition φ, loop unrolling is done by inserting a branching node with the branch condition φ before the loop header. If the branch condition is not satisfied, the control flow jumps to the loop's exit. If the branch condition was satisfied, then the control flow goes to a copy of the loop's body. This copy then jumps to the initial loop header. This operation can be repeated arbitrary times, resulting in a ”more unrolled” loop, with the cost of a significantly greater graph.

Example 12. *Consider the control flow graph shown in Figure 3.10a. Figure 3.10b shows this graph after loop unrolling with the depth of* 1*.*



(a) A control flow graph.



(b) The graph in **a**, after loop unrolling.

Figure 3.10: Loop unrolling example.

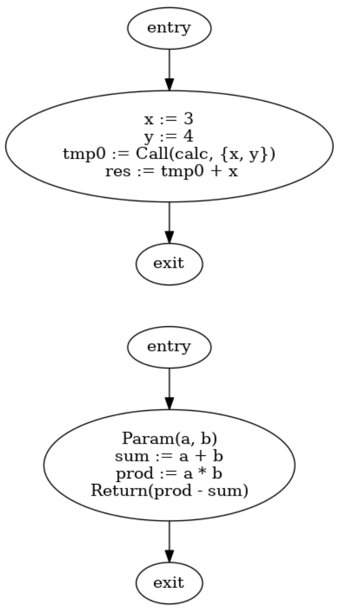
3.2.5 Function inlining

Function inlining is the procedure of replacing a function call with the callee's body. In the compiled code, this eliminates the overhead of the function call with the cost of an increased target size. In this work, function inlining is used to eliminate all inlineable function calls from the input program, as function calls can act as "black boxes". A model checker algorithm may extract more critical information from the entire inlined function body from merely just a function call.

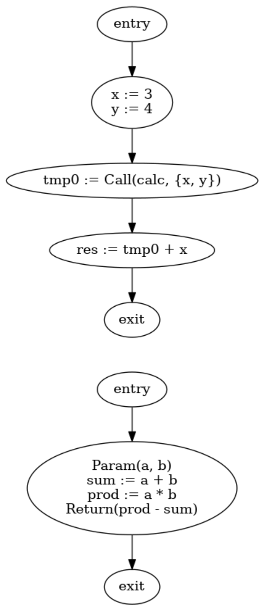
Inlining is done by recognizing inlinable functions in the program. A function is inlineable if its definition is available and that definition does not contain a recursive call. After finding such functions, the algorithm orders them in the ascending order of their degree in the call graph. This ordering optimizes the results by allowing functions containing no inlineable function calls to be the first to be inlined. After building this queue, the algorithm polls a procedure P contained within and traverses backwards from its appropriate node in the call graph, finding all procedures which call P.

After finding all such procedures, we search for the exact location of the calls and perform the actual inlining there. This requires splitting the block in which the function call is placed and replacing the call node with the entire callee body. The function parameters are replaced with the actual arguments of the function call, and the callee's return value (if it exists) is copied to the call site.

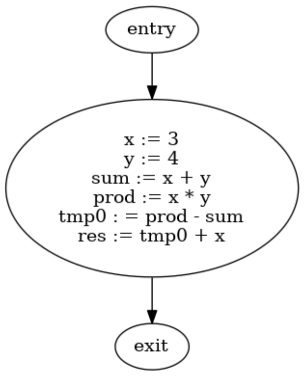
Example 13. *Consider the control flow graphs shown in Figure 3.11. The example contains two functions:* `main` *and* `calc`*. In Figure 3.11a we can see their control flow graphs. Figure 3.11b shows* `main` *after the block splitting. Finally, the resulting graph is shown in Figure 3.11c.*



(a) Control flow graphs of functions `main` (top) and `calc` (bottom).



(b) The control flow graphs shown in Figure 3.11, after block splitting.



(c) The resulting control flow graph of `main`, after the function inlining.

Figure 3.11: The function inlining procedure.

Chapter 4

Implementation

This chapter describes relevant implementation details of the developed project. Section 4.1 gives an overview of the overall architecture, Section 4.2 describes the technologies used for C source code parsing. Section 4.3 describes the architecture and implementation of the data structures and algorithms presented in Section 3.1 and Section 3.2. Finally, 4.4 describes the implemented bounded model checker.

4.1 Architecture

An overview of the program architecture can be seen in Figure 4.1. The system has three main components: the parser, the optimizer and the verifier. The parser handles C source code parsing and the intermediate representation generation. The optimizer module is responsible for running optimizing transformations, performing program slicing and constructing the control flow automaton used as the verifier input. The verifier component (implemented as a bounded model checker) performs the verification on its input model.

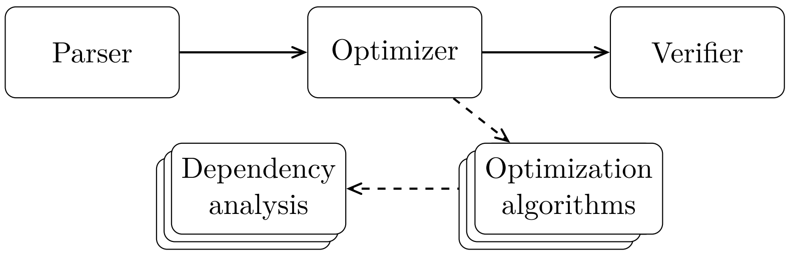


Figure 4.1: Architecture of the implemented program.

The optimizer may run a configurable number of optimization passes. Optimizations may require a certain dependency information, therefore they can query the dependency structures of the program at any time.

All components are implemented in Java[1], with additional dependencies to certain Eclipse[2] libraries. The program also makes use of the `theta` formal verification framework, developed at the Fault Tolerant Systems Research Group of Budapest University of Technology and Economics. It defines several formal tools (mathematical languages, formal models) and algorithms. It also provides a set of utilities for convenience, such as expression

[1]`https://www.java.com/en/` (version 8)
[2]`http://www.eclipse.org/`

representations and interfaces to SAT/SMT solvers, which are used in the project's implementation. The work discussed here extends this framework with an interface and toolset for C code verification.

4.2 C code parsing

C code parsing is done by using an external library, the parser component of the Eclipse CDT plug-in. CDT (short for "C Developer Tools") offers several tools to ease C/C++ development, such as syntax highlighting, content assist and some simple static analysis tools. To support these features, the plug-in contains a parser library which can be easily referenced from any Java project.

The plug-in handles the lexing and parsing phases and returns an abstract syntax tree. However, during implementation several problems arose during processing. The most serious one was the way the CDT ASTs handled traversals. The framework places restrictions on the traversal paths in the trees, namely traversals can only be preorder or postorder, however sometimes it is needed to traverse through a custom way.

To solve this, CDT's AST was transformed into a custom AST representation. This representation is relatively simpler (contains less classes and nodes, filters out some redundancy) and allows more flexible usage. This new custom representation also allows for greater modularity, as it would be easy to change the parser service in the future. All later AST transformations make use of this custom representation.

4.3 Optimizer implementation

This section describes the implementation details of the optimizer module. The optimizer is built on three main components: the intermediate representation, a set of dependency analysis tools and a configurable number of optimizer algorithms.

4.3.1 Intermediate language implementation

The class diagram in Figure 4.2 offers an overview of the intermediate representation architecture. The class `GlobalContext` represents a compilation unit, with all its functions and global variables. A procedure/function is described by a `Function` class, which contains a list of its basic blocks. A `BasicBlock` instance contains multiple instructions (`IrNode` instances). There are two types of instructions: terminating and non-terminating. Terminating instructions (`TerminatorIrNode`), such as branches, unconditional jumps and return statements end the instruction list of a basic block and provide information about the possible control flow paths leading to the next block. Non-terminator instructions (`NonTerminatorIrNode`) are simple statements which do not affect control flow.

Several instructions make use of the `theta` framework's expression (`Expr`) interfaces. Several expression classes implement this interface, such as classes for representing basic arithmetic operators, variable references and comparisons. Variable and global variable declarations (`VarDecl`) are also represented with `theta` appropriate classes.

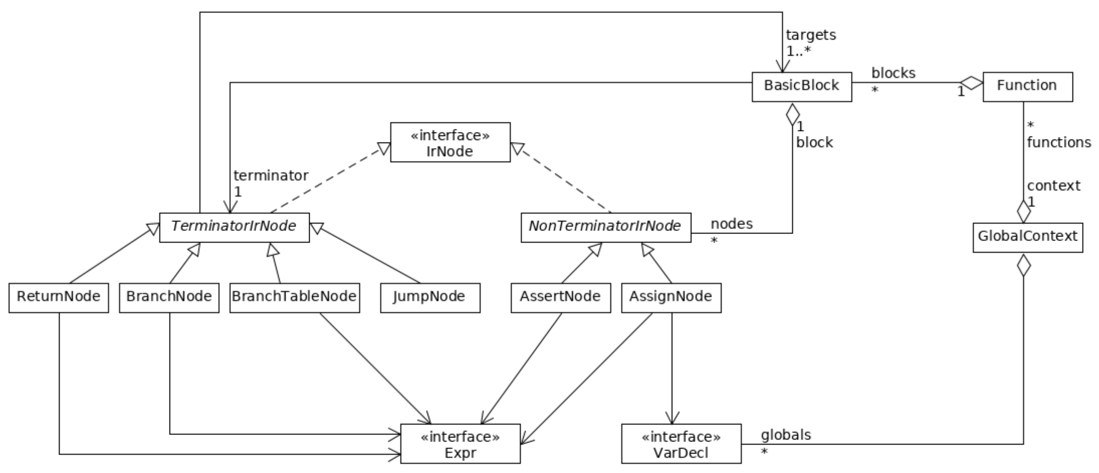


Figure 4.2: Class diagram of the intermediate representation.

4.3.2 Dependency graphs

Dependency analysis is performed by constructing several helper structures for dependency queries, as it was discussed in Section 3.1. Such structures have practically no common functionality or signature, therefore there is no shared parent interface for them. While having no public constructors, each dependency class offers a static method for building its required dependency relation and returning a new instance of the required class. Table 4.1 summarizes the implemented dependency classes.

Table 4.1: Dependency classes and the structures they implement.

Class name	**Data structure**	**Description**
`ProgramDependency`	Program dependence graph	Requires a UD-chain and a post-dominator tree
`DominatorTree`	(Post-) dominator relation	Both relations implemented in the same class
`LoopInfo`	Loop headers and bodies	Loop recognition and description
`UseDefineChain`	Use-define information	Use-define chains, reaching definitions
`CallGraph`	Call graph	Interprocedural information, function description

4.3.3 Transformations

Figure 4.3 shows the class diagram of the optimization module. The core element of the transformation workflow is the `Optimizer` class. This class handles and runs optimization passes and it is responsible for converting the optimized control flow graph (a `Function` instance) into a control flow automaton (an instance of the `CFA` interface – also provided by the theta framework).

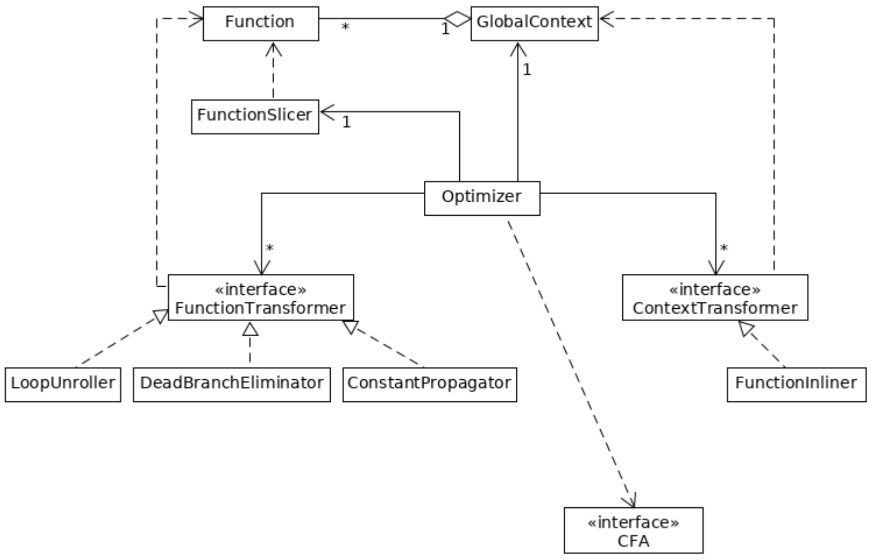


Figure 4.3: Class diagram of the Optimizer module.

Optimizations are configurable, an arbitrary number of them can be registered into the optimizer. Optimizations can be:

- function optimizations (`FunctionTransformer`) operating on a single function and
- context optimizations (`ContextTransformer`) operating on the whole context and all of its functions.

Currently implemented function optimizations are the constant propagator (operating both locally and globally), dead branch eliminator and the loop unroller. The function inlining transformation is a context optimization, as it needs to operate on all functions.

The optimizer also handles program slicing. As currently only function slicing is supported, the optimizer uses a `FunctionSlicer` instance to perform this operation on a function. After finishing with the optimization and transformation passes, the optimizer generates a list of control flow automata from each extracted slice. These smaller slices then later will be used as the verifier's input.

4.4 Bounded model checker

Currently the verifier is implemented as a simple bounded model checker algorithm, as described in Section 2.1.2. The model checker is implemented as a `BoundedModelChecker` class, which handles error path search and SMT transformation. The resulting SMT formula is checked with the Z3 SMT Solver [17], whose solution will either serve as a counterexample if the formula was satisfiable. If the formula was proved to be unsatisfiable, the algorithm will keep searching for an error path until it reaches its maximum bound.

The verifier operates on a collection of control flow automata, with each CFA being a slice extracted from the input program. If a CFA was deemed faulty, then the whole program is reported as erroneous.

Chapter 5

Evaluation

This chapter evaluates the effect of the transformations presented in Chapter 3 on the size of the model and the performance of verification. As it was discussed in Section 4.4, the verifier is implemented as a bounded model checking algorithm. However, the verification algorithm can be replaced with a more efficient verification technique at any time. Section 5.1 describes the benchmarking environment and the chosen optimizations, Section 5.2 presents the actual benchmarking results. Finally, Section 5.3 summarizes the results and makes conclusions on the efficiency of the presented transformations.

5.1 Benchmark environment

This evaluation focuses on two types of measurements: the size of the control flow automata used as the verifier input and the results of a benchmarking session on the verifier run time. The size of an automaton is currently measured by three factors: the number of its locations and edges and the depth of its depth-first spanning tree. The performance benchmarking was done by measuring the run time of the verifier on every input CFA.

Due to the slicing operation, a single input model may get split into several smaller slices. In this evaluation, each of these slices is benchmarked. However, it must be noted that during real-life model checking, it is sufficient to find one failing assertion among all slices for reporting that the input program is faulty. Furthermore, as the bounded model checking algorithm is not complete, it cannot determine whether a model is actually error-free. Therefore the verifier will report `UNKNOWN` or `TIMEOUT` (depending whether it reached the bound first or its process was just killed because of timeout) if it could not find any errors in the model. As this limitation makes checking error-free models a time-consuming operation with little benefit, all input programs included in this benchmark have at least one failing assertion.

The verifiable input programs can be separated in three categories. The first category is a collection of simple, easily verifiable models, while the other two are verification tasks are from the annual Competition on Software Verification (SV-COMP) [5]. The verification task categories are listed below.

trivial Used initially for implementation testing, this test suite contains some trivially verifiable tasks, such as no-operations, primitive locking mechanisms, summarization and greatest common divisor algorithms.

locks An SV-COMP task set aiming at control flow structures verification. Each task describes a locking mechanism with integer variables and simple **if-then-else** structures.

ECA Another SV-COMP task set, ECA (Event-Condition-Action) systems describe event driven reactive systems. The events are represented by nondeterministic integer variables, the conditions are simple **if-then-else** statements. While syntactically simple, verifying these programs requires verifying some of the largest models in the competition repertoire.

All measurements were performed with the following configuration:

- Intel(R) Core(TM) i7-3632QM CPU @ 2.20GHz,
- 16 GB RAM,
- Arch Linux with Linux Kernel 4.7.6-1-ARCH 2016 x86_64 GNU/Linux,
- Java 8.

5.2 Benchmark results

This section presents the benchmark results for each benchmark category. Unless noted otherwise, the measurement results of each category are shown in a table with four columns. The first column is test's name, the second contains its measurement results without any optimizations. The third and four columns show the measurement results with using slicing and the full optimization set, respectively. Columns containing the measurement results are also split into several subcolumns. The legend of the subcolumn labels is shown in Table 5.1.

Table 5.1: Subcolumn labels and their associated meanings.

Label	Description
#	CFA model index (for sliced programs)
L	CFA location count
E	CFA edge count
S	Verification result, `F` stands for failure, `T` for **`TIMEOUT`**, `U` for **`UNKNOWN`**
R	Verification run time (in milliseconds)

The full optimization set contains slicing, dead branch elimination and constant propagation. Loop unrolling is omitted due to the fact that it produces significantly larger models with more redundant information. These models posed a challenge to the bounded model checker algorithm. However it must be noted, that back edge reduction side effect of this transformation may be helpful for other model checking algorithms.

The trivial task set

The trivial task set consist of several simple programs, some of them containing only a single assertion. The measurement results of this set are shown in Table 5.2. As the input programs are already small, the slicing operation only results in a modest reduction in program size. Furthermore, because of the input models' size, the running times of the verifier are not comparable: their differences are well within the margin of error.

Table 5.2: Benchmarks results for the **trivial** program set.

Name	**No optimization**					**Slicing**						**Full optimization**					
	L	**E**	**D**	**S**	**R**	**#**	**L**	**E**	**D**	**S**	**R**	**#**	**L**	**E**	**D**	**S**	**R**
gcd0	11	11	7	F	2	0	11	11	7	F	2	0	11	11	7	F	3
ca-ex-const	11	13	7	F	49	0	3	2	1	U	1	0	3	2	1	U	1
						1	8	8	4	F	22	1	8	8	4	F	23
						2	6	6	3	F	12	2	6	6	3	F	14
ca-nop	3	2	1	F	1	0	3	2	1	F	1	0	3	2	1	F	1
ca-ex	12	14	8	F	2	0	5	4	3	F	2	0	5	4	3	F	1
						1	10	10	6	F	6	1	10	10	6	F	3
						2	8	8	5	F	2	2	8	8	5	F	2
ca-lock	8	8	5	F	3	0	8	8	5	F	2	0	8	8	5	F	2

The locks task set

A verification task in the **locks** section contains multiple assertions. This allows the creation of many smaller slices, which all can be verified. While all slices are measured and benchmarked in this operation, in real life scenarios it is sufficient to prove that at least one slice is erroneous.[1] Due to the bounded model checker's limitations, slices which do not contain an error path always time out.

Table 5.3 shows a benchmark summary of two verification tasks. Each task was split into several slices, with most slices having only a portion of the original program's size. In each case, the smallest slice's node and edge count is 90% smaller than the original program's. On average, the slicer algorithm reduces node and edge count both by 75%. Other optimization algorithms only reduce both numbers only slightly.

As discussed previously, all running time lengths are so short that their meaningful comparison is impossible. It must also be noted that the spanning tree depth may alternate because of the nature of the depth-first search algorithm. As optimizations change the program's structure and there is no restriction about which nodes will the DFS select during traversal, the depth of a smaller graph may actually be larger.

The ECA task set

The ECA task set is special in the sense that a program is split into several different verification tasks. Multiple files define the same program, but with a different error condition each time. As a single verification task contains only one assertion, the slicer algorithm yields a single slice for each input program. Because only the error conditions differ for each task, these slices are extremely similar.

Despite the fact that ECA only uses primitive language constructs, the resulting model is huge and complex, even after optimization transformations. The models contain if-else constructs inside a loops. This yields an exponential number of possible error paths. The

[1]This also allows efficient parallelization: each slice can be verified on a separate thread.

Table 5.3: Benchmarks results for the **locks** program set.

Name	No optimization					Slicing						Full optimization					
	L	**E**	**D**	**S**	**R**	**#**	**L**	**E**	**D**	**S**	**R**	**#**	**L**	**E**	**D**	**S**	**R**
locks15	145	207	116	F	57	0	59	79	47	F	57	0	58	77	48	F	53
						1	108	165	71	F	53	1	107	163	78	F	54
						2	42	60	26	T	–	2	41	58	34	T	–
						3	40	57	29	T	–	3	39	55	30	T	–
						4	38	54	30	T	–	4	37	52	26	T	–
						5	36	51	30	T	–	5	35	49	28	T	–
						6	34	48	30	T	–	6	33	46	26	T	–
						7	32	45	21	T	–	7	31	43	26	T	–
						8	30	42	23	T	–	8	29	40	18	T	–
						9	28	39	21	T	–	9	27	37	22	T	–
						10	26	36	20	T	–	10	25	34	22	T	–
						11	24	33	18	T	–	11	23	31	17	T	–
						12	22	30	19	T	–	12	21	28	11	T	–
						13	20	27	17	T	–	13	19	25	12	T	–
						14	18	24	15	T	–	14	17	22	15	T	–
						15	14	18	11	T	–	15	13	16	10	T	–
						16	14	18	12	T	–	16	13	16	11	T	–
locks14	136	194	106	F	52	0	56	75	44	F	34	0	55	73	46	F	52
						1	105	161	68	F	59	1	104	159	74	F	60
						2	40	57	28	T	–	2	39	55	29	T	–
						3	38	54	30	T	–	3	37	52	20	T	–
						4	36	51	24	T	–	4	35	49	27	T	–
						5	34	48	22	T	–	5	33	46	21	T	–
						6	32	45	26	T	–	6	31	43	23	T	–
						7	30	42	25	T	–	7	29	40	16	T	–
						8	28	39	18	T	–	8	27	37	25	T	–
						9	26	36	22	T	–	9	25	34	13	T	–
						10	24	33	15	T	–	10	23	31	14	T	–
						11	22	30	13	T	–	11	21	28	12	T	–
						12	20	27	17	T	–	12	19	25	16	T	–
						13	18	24	15	T	–	13	17	22	15	T	–
						14	14	18	12	T	–	14	13	16	11	T	–
						15	14	18	11	T	–	15	13	16	10	T	–

bounded model checking algorithm is rather ineffective for such problems and thus, it cannot handle these models in a reasonable amount of time, resulting in a timeout in all cases.

The programs of this category contain two functions: an entry function (**main**) and a computation function which contains all program logic except event polling. In order to handle the system as a whole, function inlining is enabled for the ”no optimization” category as well.

The measurement results are shown in Table 5.4. As it can be seen, the slicing algorithm extracted slices with the same size for every error condition. The size reduction of the slicer is considerable, node size is reduced by 21%, edge count is reduced by 17%. Further optimizations reduce these numbers only slightly.

Table 5.4: Benchmarks results for the **ECA** program set.

Name	**Inlining only**					**Slicing and inlining**						**Full optimization**					
	L	**E**	**D**	**S**	**R**	**#**	**L**	**E**	**D**	**S**	**R**	**#**	**L**	**E**	**D**	**S**	**R**
eca0-label32	391	459	97	T	–	0	309	377	83	T	–	0	307	374	81	T	–
eca0-label20	391	459	97	T	–	0	309	377	83	T	–	0	307	374	81	T	–
eca0-label33	391	459	97	T	–	0	309	377	83	T	–	0	307	374	81	T	–
eca0-label21	391	459	97	T	–	0	309	377	83	T	–	0	307	374	81	T	–
eca0-label44	391	459	97	T	–	0	309	377	83	T	–	0	307	374	81	T	–

5.3 Summary

The program slicing operation is able to notably reduce program size, resulting in up to 90% reduction (both in node and edge count) in some extreme cases. As proving only one slice's faultiness is enough for failing models, this may enable a considerable speed up if the erroneous slice is found early. However, for error-free models, checking multiple slices instead of one large program can introduce an overhead. This could be mitigated by checking the smaller slices in parallel operations.

The other optimization algorithms (constant propagation, dead branch elimination) reduce the input program's size only modestly. This is probably due to the fact that the SV-Comp entry programs are already simplified and the slicing algorithm is able to cast away dead instructions and branches (as they are not relevant for the slicing criteria). The effect of these optimizations on more real-life programs and without slicing support is a target of further evaluation.

The measurement results clearly show the limitations of the implemented bounded model checking algorithm. The verifier timed out on almost all realistically sized programs, allowing only small programs to be verified. Due to the small running time, most running time measurement differences are within the margin of error. As the verification method is completely replaceable and this bounded model checking was merely implemented for the workchain's completeness, this is not a large issue. However, further investigation is required for runtime evaluation. Future evaluations may replace the bounded model checker with a CEGAR-based [12] or a symbolic [10] algorithm.

Chapter 6

Conclusion

This work presented a transformation workflow for generating optimized formal model representations of C programs. I examined several program analysis methods and structures used in compiler design: data flow computation for use-define chain construction, dominator relation and dominator trees, program dependence graphs, call graphs (Section 2.2 and Section 3.1). Furthermore I adapted and implemented four optimization algorithms (constant propagation, dead branch elimination, function inlining, loop unrolling) along with a program slicing algorithm, all used for program model simplification (Section 3.2). For benchmarking purposes I implemented a verifier component with a bounded model checking algorithm (Section 2.1.2 and Section 4.4). The developed project was built as modular components, therefore any module can be replaced or extended for further improvement.

The evaluation of the above methods (Chapter 5) showed that program slicing is promising technique for program size reduction especially for verification. While the other optimization methods proved to be much less efficient, their program size reduction may still be valuable. As the runtime evaluation proved to be difficult because of the bounded model checker's performance, further evaluation is in order with other, more effective verification algorithms.

Future work. This project has several opportunities for future improvements and feature additions.

- Extending the support for more features of the C language. Such features could be arrays, pointers, structs, etc.
- As only a subset of all possible optimization algorithms were implemented, the list can be extended with other transformations, such as interprocedural program slicing.
- LLVM [24] is a compiler infrastructure framework, which provides a language-agnostic intermediate representation (*LLVM IR*) with multiple optimization algorithms and has a frontend for several programming languages, such as C, C++, Fortran, Swift, or Rust. Adding support for the LLVM IR would extend the range of supported languages dramatically and would also implicitly add multiple fine-tuned optimizations into the workflow.
- Currently the workflow does not contain any traceability information, therefore finding the counterexample in the original code requires manual traceback. Support can be added for a traceability model which is able to report the erroneous operation's location in the original program.

Acknowledgements

First of all, I would like to thank my supervisors, Tamás Tóth and Ákos Hajdu for their support and guidance. Their help and ideas were indispensable during both research and development. I also would like to thank András Vörös for introducing me to this project and constantly following my progress.

This work was partially supported by IncQuery Labs Ltd. and MTA-BME Lendület Research Group on Cyber-Physical Systems.

List of Figures

List of Tables

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[2] Aws Albarghouthi, Yi Li, Arie Gurfinkel, and Marsha Chechik. Ufo: A framework for abstraction- and interpolation-based software verification. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification: 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, pages 672–678. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[3] Frances E. Allen. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*, pages 1–19, New York, NY, USA, 1970. ACM.

[4] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani. Software model checking via large-block encoding. In *Formal Methods in Computer-Aided Design*, pages 25–32. IEEE, 2009.

[5] Dirk Beyer. Reliable and reproducible competition results with benchexec and witnesses (report on SV-COMP 2016). In Marsha Chechik and Jean-François Raskin, editors, *Tools and Algorithms for the Construction and Analysis of Systems: 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9636 of *Lecture Notes in Computer Science*, pages 887–904. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[6] Dirk Beyer and Stefan Löwe. Explicit-state software model checking based on CEGAR and interpolation. In *Fundamental Approaches to Software Engineering*, volume 7793 of *Lecture Notes in Computer Science*, pages 146–162. Springer, 2013.

[7] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.

[8] Aaron R Bradley and Zohar Manna. *The calculus of computation: Decision procedures with applications to verification.* Springer, 2007.

[9] Ingo Brückner, Klaus Dräger, Bernd Finkbeiner, and Heike Wehrheim. Slicing abstractions. In *International Symposium on Fundamentals of Software Engineering*, volume 4767 of *Lecture Notes in Computer Science*, pages 17–32. Springer, 2007.

[10] Jerry R Burch, Edmund M Clarke, Kenneth L McMillan, David L Dill, and Lain-Jinn Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science*, pages 428–439. IEEE, 1990.

[11] Alonzo Church. A note on the Entscheidungsproblem. *The Journal of Symbolic Logic*, 1(1):40–41, 1936.

[12] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003.

[13] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[14] EdmundM. Clarke. The birth of model checking. In *25 Years of Model Checking*, volume 5000 of *Lecture Notes in Computer Science*, pages 1–26. Springer, 2008.

[15] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[16] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages, Los Angeles*, pages 238–252, New York, NY, 1977. ACM.

[17] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[18] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[19] Patrice Godefroid, Doron Peled, and Mark Staskauskas. Using partial-order methods in the formal validation of industrial concurrent programs. *IEEE Transactions on Software Engineering*, 22(7):496–507, 1996.

[20] M. S. Hecht and J. D. Ullman. Characterizations of reducible flow graphs. *J. ACM*, 21(3):367–375, July 1974.

[21] Matthew S. Hecht. *Flow Analysis of Computer Programs.* Elsevier Science Inc., New York, NY, USA, 1977.

[22] Thomas A Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 58–70. ACM, 2002.

[23] Matti Järvisalo, Daniel Le Berre, Olivier Roussel, and Laurent Simon. The international SAT solver competitions. *AI Magazine*, 33(1):89–92, 2012.

[24] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[25] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, January 1979.

[26] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, SDE 1, pages 177–184, New York, NY, USA, 1984. ACM.

[27] Karl Joseph Ottenstein. *Data-flow Graphs As an Intermediate Program Form.* PhD thesis, West Lafayette, IN, USA, 1978. AAI7905807.

[28] B. G. Ryder. Constructing the call graph of a program. *IEEE Trans. Softw. Eng.*, 5(3):216–226, May 1979.

[29] Alan Mathison Turing. On computable numbers, with an application to the Entscheidungsproblem. *Journal of Math*, 58:345–363, 1936.

[30] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.