Translation Validation for the LLVM Compiler

Kevin Jacobus de Vos

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisors: Prof. Nuno Claudino Pereira Lopes
Prof. José Carlos Alves Pereira Monteiro

Examination Committee

December 2020

Acknowledgments

I would like to thank my advisors José Monteiro and Nuno Lopes for their great help and guidance as without them I would not have been able to finish this work. Their support helped me learn and understand complex concepts and build algorithms that made great progress towards achieving the goals of this work.

I would also like to thank my family and advisors for encouraging and supporting me during this time, especially when I felt like giving up and when I was not making any progress.

Abstract

Compilers can have many bugs and miscompile, often resulting in unintended program behavior not specified by the original program. Translation validation is a technique that verifies the correctness of compilation given the source and target code. We extend Alive [1] to 1) reduce the SMT formula sizes generated to improve performance and 2) significantly increase analysis coverage with a new loop unrolling algorithm for loops written in Alive IR. Since the LLVM compiler is susceptible to miscompilation we do not use any loop unroll mechanisms available in LLVM. We evaluate our main contribution by measuring the increase of both *block* and *path* coverage for a large number of programs. We have been able to reduce Alive execution time with our SMT formula reduction algorithms and find multiple bugs in the LLVM compiler using our loop unrolling algorithm. Finding and fixing these bugs is essential to reducing the likelihood of miscompilation in LLVM.

Keywords: Compilers - Software verification - Translation validation - Loop unroll - Refinement - Undefined behavior

Resumo

Os compiladores podem ter bastantes erros e podem compilar incorretamente, muitas das vezes resultando em comportamento differente do especificado pelo programa original. *Translation Validation* é uma técnica capaz de verificar se as transformações aplicadas durante compilação são corretas. Estendemos o Alive [1] com o objetivo de 1) reduzir tamanhos das fórmulas SMT geradas e desta forma melhorar o desempenho do Alive, 2) aumentar substancialmente a cobertura de anàlise do Alive com um novo algorithmo de desenrolar ciclos para ciclos escritos em Alive IR. Dado que o compilador LLVM pode compilar programas incorretamente não utilizamos mecanismos de desenrolamento de ciclos disponíveis pelo LLVM. Avaliamos a nossa principal contribuição através da medida de cobertura de blocos e cobertura de caminhos para um grande número de programas. Foi possível reduzir o tempo de execução do Alive com os nossos algoritmos de redução de tamanho de fórmulas SMT e foi possível encontrar vários erros no compilador LLVM com o nosso algoritmo de desenrolamento de ciclos. Encontrar e resolver erros em compiladores como o compilador LLVM é essencial para reduzir a probabilidade de compilação incorreta.

Keywords: Compiladores - Verificação de software - Desenrolamento de loops - Refinação - Comportamento não definido

Contents

List of Tables

List of Figures

Chapter 1

Introduction

It is common to have programs execute in ways that developers did not intend them to. Often, the root cause of the program misbehaving ends up being the program itself, regardless of whether the blame is put on the program or on something else entirely. In reality however, compilers are not bug-free and can miscompile more often than we would expect. For example, Fig. 1.1 shows a bug in the *instcombine* LLVM optimization pass. Here the transformation is mathematically correct when using real arithmetics, but not when using the modular arithmetics that we see in computing. In particular, `b` is `x * 2`^{c} `/ d` before the transformation (source) and is `x / (d / 2`^{c}`)` after it (target). As a program transformation it introduces division by zero when having for instance `x = 0`, `c = 3` and `d = 7`, where `b = 0 / 7` in the source and `b = 0 / 0` in the target.

```
int a = x << c;          ⟹          int t = d / (1 << c);
int b = a / d;                      int b = x / t;
```

Figure 1.1: Incorrect transformation showcasing the subtleties of compiler math.

As we can see, just like almost any software, compilers are not bug-free. Compiler bugs become especially concerning when safety-critical applications are involved. Some of these bugs can be hard to find and can remain latent for a long time. Finding and resolving as many compiler bugs is thus especially important. Doing this is not trivial since compilers can be extremely large and complex to the point that verification through manual inspection becomes impractical and too difficult. Compiler correctness verification techniques such as *Translation Validation* (TV) [2] and *Compiler Verification* are helpful to resolve this issue and are in use by many different correctness verification tools [3–7]. Compiler fuzzing tools that randomly generate test cases are also interesting approaches to finding bugs. In particular, CSmith [3] is a randomized test-case generation tool and in a period of three years it was successful in finding more than 325 previously unknown bugs to compiler developers. Orion [4] created a practical implementation of *Equivalence Modulo Inputs* (EMI) for validating C compilers, and their extensive testing at the time has led to 147 confirmed bugs in GCC and LLVM. Athena [5] is also based on EMI and has found 72 new bugs in GCC.

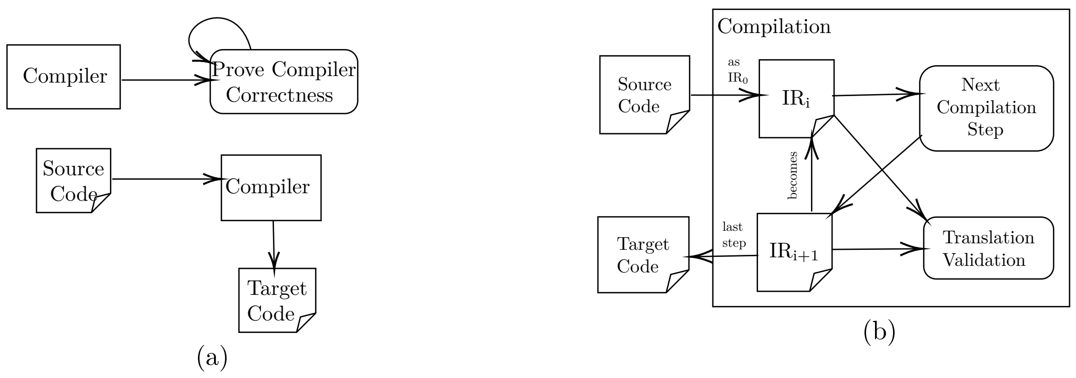


Figure 1.2: (a) *Compiler Verification*: The compiler is verified prior to compilation. (b) *Translation Validation* when performed at each step of compilation: during compilation, the source code is transformed to an intermediate representation (IR) at each compiler step. Then the previous $\mathrm{IR_i}$ and the new $\mathrm{IR_{i+1}}$ are fed into the translation validator.

Translation validation is a simpler approach over compiler verification, where the source and the target code of some transformation are compared and checked to see whether the target is a *refinement* of the source, meaning that the target only shows behavior that we would see from executing the source program. Moreover, since program optimization usually involves multiple smaller transformations, the accuracy of translation validation can be improved by also validating each pair of different program versions constructed during compilation at the cost of more time spent in analysis.

Compiler verification on the other hand checks whether individual optimizations are correct before any program is even compiled. Compiler verification proofs are created once and remain valid for all programs compiled and optimized by the compiler for a particular optimization.

Neither approach is better than the other as both are useful and have different use cases. For instance, we would opt for translation validation when either the compiler code base is not available or too complex. Similarly, compiler verification would be the better choice when the computational cost incurred with translation validation is too high or when we want to verify a transformation as correct regardless of the input values. Fig. 1.2 illustrates the differences between compiler verification and translation validation for each optimization step.

In this thesis we extend Alive [1], a compiler verification and translation validation tool for proving LLVM optimizations as correct even when under the presence of undefined behavior. Such programs become non-deterministic and behave in unpredictable ways.

Since current Alive's translation validation tool drastically increases program compilation time, it cannot be deployed for all users. As an effort to improve Alive's performance we present algorithms to reduce the sizes of correctness proofs built by Alive resulting in a reduction of time spent checking these proofs in the Z3 [8] SMT solver, which is the dominant factor in Alive's total execution time. Specifically, we find that Alive's representations of undefined behavior and semantics of phi instructions have a lot of redundancy that can be eliminated. Phi instructions assign values to variables depending on the history of program control-flow. Although Alive uses the Z3 SMT solver, other SMT solvers such as STP [9], MathSAT5 [10], CVC4 [11] and Boolector [12] exist with varying support on different SMT theories. The main goal of Alive is to identify as many bugs as possible in compilers such as LLVM that result in miscompilation, often with the consequence of programs misbehaving in real life situations. Ideally we would want to eliminate the chances of miscompilation, but sadly this is almost impossible to achieve. Instead, improving accuracy and coverage of Alive's program analysis is an ongoing effort.

```
int main() {
  int i = 0;
  while (i < 10) {
    foo(i++);
  }
  return i;
}
```

Figure 1.3: Simple program with a while loop written in C.

In addition to performance improvements, in this thesis we want to improve how Alive handles loops. Due to limitations to the *symbolic execution* program verification technique that Alive relies on to construct proofs, Alive cannot cover the body of most program loops. Consider the example program in Fig. 1.3, because Alive currently does not cover paths that do not reach some program exit or paths that visit the same instruction twice, the call to *foo* for instance is not covered by Alive's analysis. Doing so requires Alive's symbolic execution to visit `i < 10` twice and the return once.

As an additional step towards increasing the chances of finding bugs in LLVM we present a loop unrolling algorithm to solve this problem. This algorithm duplicates portions of program loops for a bounded number of times such that some loop instructions can be visited more than once in a controlled manner. This increases Alive's coverage while also avoiding limitations of *symbolic execution.* Loop unrolling can result in significant increases in program size, especially when we have deeply nested loops, therefore choosing the loop unrolling factor must be done with care.

1.1 Thesis Overview

This thesis is structured into four parts. We start by giving a brief summary of related work in Chapter 2. In Chapter 3 we present solutions to improve how loops are analyzed in Alive. Specifically, we extend Alive such that it can analyze more iterations of each loop and cover more program paths and regions of code that were previously ignored.

In Chapter 4 we make use of existing graph theory concepts and develop our own control flow graph (CFG) analysis algorithms for better construction of SMT formulas in Alive, reducing their size, complexity and computational cost incurred by running an SMT solver, which is a dominant factor in the overall program analysis time. Given that our improvements to loop analysis in Alive cause programs to increase significantly in size, reducing the execution time of Alive's analysis with these algorithms should be useful in mitigating the added cost of loop unrolling.

Lastly, we discuss our results obtained from both our SMT formula reduction and loop unroll algorithms in Chapter 5 which includes performance analysis, increase in Alive's coverage and bugs found in LLVM.

Chapter 2

Related Work

The notion of translation validation was initially presented in the work [2], where each run of the compiler was followed by a validation phase to verify whether the target correctly implemented the source program. Many works since then have explored this technique to validate either any optimization in general [13–17], or target specific optimizations such as instruction scheduling [18], lazy code motion (LCM) [19] and software pipelining [20].

Program analysis techniques such as *symbolic execution* used in [15, 18] have proven to be quite useful in translation validation. For instance Necula [15] uses symbolic execution to compute a simulation relation between two programs. A simulation relation is a witness that describes the conditions in which two programs are equivalent. Witnesses are independent of the optimization procedure and can be produced once, and are used as proofs that some transformation is correct.

2.1 Compiler Verification and Translation Validation

Alive [1] is a translation validation and compiler verification tool to aid in writing correct LLVM optimizations, automatically proving them correct even in the presence of LLVM's undefined behavior. Verification consists in checking that a transformation is correct for any input, where as validation only checks a specific transformation with one combination of the inputs. Rhodium [7] is a compiler verification tool that writes automated soundness proofs with facts obtained through data-flow analysis.

As for translation validation, many different tools have been developed in previous works, including:

- Whether the tool only supports validating programs to which only one transformation was applied [19–21], or combinations of transformations [22];
- Some only validate specific optimizations [18–20] while others validate any optimization [13, 21];
- The tool uses a simulation relation [15] or a more complex representation such as a graph to reason about program equivalence [13, 17];
- Whether or not the tool takes the possibility of and the different forms of undefined behavior into account in translation validation [13, 23]. For example, if the possibility of signed integer overflow in C or C++ is considered.

```
int f(a,t) {
   a = t + 1;
   a = t + 2;
   return 1;
}
```

(a)

```
int g(a,t) {
   a = t + 3;
   return 1;
}
```

(b)

Figure 2.1: An example of a transformation with the source code in (a) and the target code in (b).

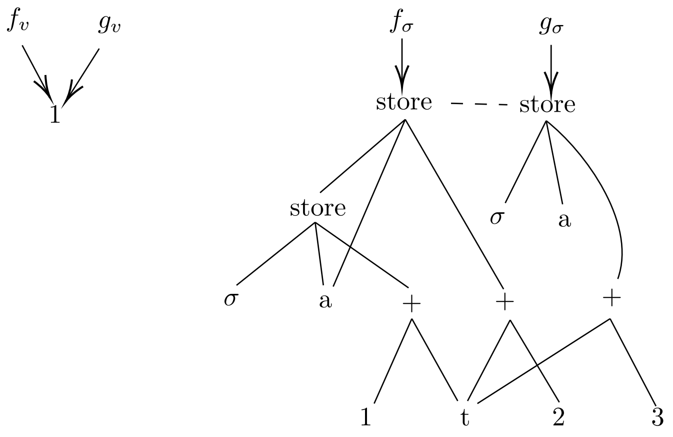


Figure 2.2: Combined Program Expression Graph (PEG). The dashed lines represent the equality relations generated in the equality saturation technique between graph vertices, and in this case, both f_σ and g_σ, representing the heap resulting of executing a function *f* and *g* respectively, are equivalent.

Peggy [16] constructs a purely functional representation of the return values and the heaps of both the source and target programs for a given transformation. An example of a PEG is shown in Fig. 2.2, for the program code shown in Fig. 2.1. The vertices $\mathtt{f}_\sigma$ and $\mathtt{g}_\sigma$ correspond to the heaps of the functions `f` and `g` respectively. Similarly, $\mathtt{f}_\mathtt{v}$ and $\mathtt{g}_\mathtt{v}$ represent the return values. Since both the PEG for the source and the one for the target are used in the same PEG space, we obtain a single combined PEG, where vertices are shared when possible. After the combined PEG is constructed, the authors proceed with adding equality relations between its vertices, based on axioms created through a technique called *equality saturation.* In this example, equality saturation builds the following equality relation:

$$store(store(\sigma, a, t + 1), a, t + 2) = store(\sigma, a, t + 3) \tag{2.1}$$

This relation is added into the PEG in the form of a dashed line. If the return values are equal, and if the program heaps are equivalent, then the tool developed by the authors concludes that the program transformation from *f* into *g* is correct. The equality saturation technique was developed in their previous work and can be seen in more detail in [24].

CompCert [6] takes an interesting approach by using both the compiler verification and translation validation approaches into one, where a verified translation validation algorithm is used when correctness proofs are too difficult to build with compiler verification.

CRELLVM [14] generates correctness proofs alongside translation validation. These proofs are witnesses of correctness of the compiler transformation and are independent of the optimization procedure. In this work, the witnesses are written in *extensible relational hoare logic* (ERHL) allowing them to be easily and quickly checked with an interactive proof assistant such as Coq, reducing the need to run translation validation more than once. Similarly to these proofs constructed in CRELLVM, a simulation relation is also a witness. Instead of using ERHL, Alive encodes program semantics into the SMT (*Satisfiability Modulo Theory*) language, and then correct program compilation is verified by using the Z3 SMT solver with the SMT formulas Alive generates as input.

The work [20] presents a translation validation tool that addresses the *software pipelining* loop optimization. Software pipelining overlaps multiple iterations of a loop to better exploit instruction-level parallelism. The authors in this work can decompose loops into a loop prolog, a steady state and an epilogue which significantly simplifies validation for this particular optimization.

The work in [19] constructs a translation validator verified in Coq for the *lazy code motion* (LCM) optimization. LCM may also include the loop invariant code motion transformation that moves invariant instructions out of loops. To validate this transformation, their work relies on *data-flow analysis* techniques to verify that the values assigned to variables moved to different points in the target code are maintained.

Both [19] and [20] validate specific loop optimizations. In practice, the number of different compiler optimizations that can be applied is large, and having a tool that can validate any given program transformation is very useful. Currently Alive can check most program transformations, however, it currently falls short in validating loop optimizations. Loops are difficult to validate as several program analysis techniques do not work well with them. *Symbolic execution* is an example of such a technique. Alive uses it to encode program semantics into the SMT language before passing it into an SMT solver. The drawback with symbolic execution is that information about the program state of each loop iteration must be kept in order to represent the semantics of a given program. When a loop has a sufficiently large number of iterations, this results in an extremely large amount of memory being used. In the context of Alive, this means that the encoding of program semantics written in the SMT language just becomes too large to be verified in a realistic amount of time. In this thesis we improve upon the current implementation of Alive by unrolling program loops for a bounded number of times, improving the accuracy of Alive's translation validation tool whilst also avoiding the drawbacks of techniques like symbolic execution on loops with a high enough number of iterations.

TVOC [22] is a translation validation tool that presents a new way of modelling loops by characterizing them by key features, allowing validation of loops without needing to deal with the limitations of some program analysis techniques. These features can be for instance the number of loops, the number of instructions in each loop and the set of values that each loop index variable can take. This way of representing loops significantly simplifies loop validation as we can quickly identify which loop optimizations were applied, by observing how these key features change from source to target in a compiler transformation. To illustrate the benefits of this representation of loop structures, consider the code in Fig. 2.3, where we observe that: (*i*) target code (b) is one loop shorter than the source (a); (*ii*) the target (b) has an additional instruction in the loop which corresponds to the instruction of the second loop in the source. These differences in terms of the number of loops and the number of instructions in each loop body closely resemble the loop fusion optimization, and thus validation of the transformation reduces to validating the application of this optimization. Knowing what optimization to validate and by having code specifically

```
for (int i = 0; i < 100; ++i) {
  a[i] := x + 5;
}
for (int i = 0; i < 100; ++i) {
  a[i] := a[i] * 2;
}
```

(a)

```
for (int i = 0; i < 100; ++i) {
  a[i] := x + 5;
  a[i] := a[i] * 2;
}
```

(b)

Figure 2.3: A loop fusion optimization example. Source code on the left (a) and target on the right (b).

built for validating certain optimizations can significantly simplify and improve the effectiveness of the TVOC validation process. One notable difficulty in validating programs with loops is knowing which loops in the source correspond to which loops in the target. TVOC does not deal with this issue and assumes that this mapping between loops is known beforehand.

Another aspect to consider when validating loop transformations is that these are often applied in a particular order. Because TVOC already has some information on the order in which compilers usually apply certain optimizations, TVOC can confidently guess this order. Afterwards, TVOC checks each guess by constructing intermediate versions of the program through *program synthesis*, a technique that can be used to generate code from formal specifications. With this TVOC is then able to validate the transformation by checking each subsequent pair of generated intermediate programs. If the validator concludes each subsequent pair as equivalent then the transformation is deemed correct.

Identifying the loop transformations applied on a given program can help improve the accuracy of validation of each program transformation. This unfortunately is not as scalable because separate checks dedicated for each compiler optimization are required. Alive does not follow this approach and instead builds a more thorough representation of program semantics that is independent of the applied compiler optimizations.

The work [13] is similar to Alive, where the range of optimizations that can be validated is large and unknown to the validator. The authors in [13] focus on incrementally building a representation of the correlations between points in the source and in the target programs of a given compiler transformation. Afterwards, constraints on program values and memory states, called the predicates, are associated with each correlation. The validator then only needs to check these predicates similarly to how a simulation relation is checked in Necula [15], without any knowledge about the optimizations performed by the compiler.

In contrast to most of the related work mentioned so far, [13] also supports optimizations with some potential undefined behavior. Alive takes this a step further and fully encodes undefined behavior of programs written in LLVM IR, which greatly increases the number of miscompilations that it can find.

2.2 Program Representations

To properly reason about a program's structure and semantics, we need clear representations of program code. These representations can have a variable level of abstraction. Programming languages as Java and

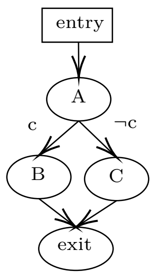
Python are high-level languages, while LLVM IR and Alive IR are low-level program representations.

2.2.1 Control Flow Graph

We define a control flow graph (CFG) as a pair (V, E), where V represents the graph vertices and E represents the directed edges between these vertices. A CFG gives a clear and visual representation of program structure and control flow. An example of a CFG is illustrated in Fig 2.4.

```
void foo(bool c) {
  A();
  if (c) {
    B();
  } else {
    C();
  }
  return;
}
```

(a)



(b)

Figure 2.4: A control flow graph (b) for the example program (a).

2.2.2 Programs in LLVM IR

Each program in LLVM IR is divided into basic blocks that represent sequences of instructions between control-flow altering instructions. For well defined programs, whenever control flow enters a basic block, all its instructions are executed. The vertices of a CFG represent basic blocks identified by some label while the edges represent the control flow of the program defined by LLVM IR instructions such as *br* for branches and *switch* for switches.

2.2.3 Intermediate Representations

Both Alive IR and LLVM IR are written in a Static Single Assignment (SSA) form where each new assignment to a variable initializes a fresh variable. SSA was originally created for compilers, but turned out to also be useful for program analysis and correctness verification. Verification conditions are another use of SSA, each representing some relationship between an aspect of the source and the target program. For example, one verification condition could state that one variable in the source is equal to some variable in the target. Note that since each variable modification will result in the initialization of a new variable in SSA, the value of each variable never changes, thus the verification condition can be checked at any point in time. TVOC [22] makes use of verification conditions and aggregates them to form inductive proofs of correctness of compilation. As an example of SSA, consider the two fragments of code in Fig 2.5.

```
x = 4;
y = x + 2;
x = y;
```

x_1 = 4;
y_1 = x_1 + 2;
x_2 = y_1;

Figure 2.5: Example illustrating the differences between code not SSA form on the left and the code in SSA form on the right.

With SSA we can also construct logical expressions that represent a sequence of instructions. An example of such an expression for the code fragment *b)* in Fig 2.5 would be:

$$x_1 = 4 \wedge y_1 = x_1 + 2 \wedge x_2 = y_1 \tag{2.2}$$

If we attempt to construct an expression from code not in SSA we would get:

$$x = 4 \wedge y = x + 2 \wedge x = y \tag{2.3}$$

Feeding the expression in Equation 2.3 into an SMT solver for proving would not work since x cannot represent two different values simultaneously. Program analysis techniques such as symbolic execution and bounded model checking make heavy use of SSA form.

In addition to SSA form, a program can also be written in *Static Single Information* (SSI) form which is an extension of SSA. SSI gives path-sensitive information in addition to flow-sensitive information with SSA by associating branch predicates with variable names at certain points in the program. SSI is useful for path-sensitive analysis techniques, an example of which is *range analysis*, a technique that maps integer variables to the possible ranges of values they can take throughout program execution. For instance, we can eliminate code that is not reachable unless a variable takes specific values. We can remove some array out of bounds checks and integer overflow detection checks, and we can improve optimizations such as constant propagation, where variables and arithmetic operations can be replaced by constants if these variables and arithmetic operands are proven to also be constant. An example of a code fragment in SSI form can be seen in Fig. 2.6.

```
y₀ = x₀ + w₀
if (y₀ < 0)
   y₁ = π(y₀)
   y₂ = -y₁
else
   y₃ = π(y₀)
y₄ = φ(y₂, y₃)
```

Figure 2.6: A code fragment in SSI form where the variables y_2 and y_3 are associated with the branch predicate $y_0 < 0$ through the π-function.

Gated Single Assignment (GSA) is another extension of SSA that represents loops in a higher level of abstraction. GSA provides three different pseudo-functions to represent control-flow dependent values within loops. Consider the example loop in Fig. 2.7. Note also that the `br` instruction can take either one or three operands. If only one operand is given then it is an unconditional jump. Otherwise, it takes in a condition as the first operand, and moves control-flow to the basic block indicated by the second operand if the condition is true, or to the basic block indicated by the third operand if false.

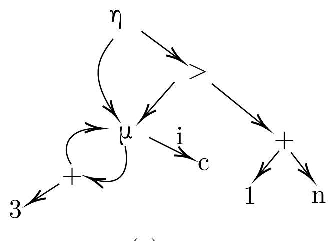
Here the phi instruction and the final assignment at loop exit are replaced by calls to two new pseudo-functions, μ and η. The code appears to be in SSA form, however the creation and assignment of new variables is abstracted by these two functions in GSA. The phi instruction in the SSA version selects one of the entries given to it depending on through which predecessor of `loop` was traversed.

```
  x0 = c
loop:
  xp = phi(x0, xk)
  b = xp * 2 > n + 1
  br b, loop1, exit
loop1:
  xk = xp + 3
  br loop
exit:
  x = xp
```

(a)

```
  x0 = c
loop:
  xp = μ(x0, xk)
  b = xp * 2 > n + 1
  br b, loop1, exit
loop1:
  xk = xp + 3
  br loop
exit:
  x = η(b, xp)
```

(b)



(c)

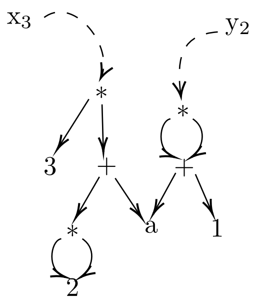
Figure 2.7: Code example not written in Gated SSA in (a), the same example but written in Gated SSA form in (b), and a shared value graph of this example in (c).

In addition to SSA, SSI, and GSA, many other intermediate representations exist for different use cases. For example, MLIR [25] is a multi-level intermediate representation highly influenced by LLVM and SIL [26] is a high-level SSA-form intermediate representation designed specifically for the Swift programming language.

```
B1:
  x1 = 2 * 2
  x2 = a + x1
  x3 = x2 * 3

B2:
  y1 = a + 1
  y2 = y1 * y1
```

(a)



(b)

Figure 2.8: Shared value graph on the right (b) with the corresponding code on the left (a), where B1 and B2 are two different basic blocks. The shown instructions are not written in any specific programming language.

The work in [17] proposes a translation validation tool that creates and evaluates value graphs, similar to the PEG's created in [16]. Value graphs can represent dependencies between variables in a program. To illustrate, consider the two sequences of instructions designated as B1 and B2 in Fig. 2.8. Consider also that we want to check whether the variable x_3 is equivalent to the variable y_2. To do this, we first create a value graph for each instruction sequence, and then combine them into a single shared value graph, where any vertex that is common to both graphs is shared in the resulting graph. As an example, the vertex representing a is shared between both variables in the shared value graph.

Once the shared graph is constructed, normalization operations are performed on the graph in order to reduce it. For example, the vertex representing the multiplication 2*2 in B1 can be replaced by a vertex representing the number 4 through constant folding. These rewrite rules are applied in sequence until they can no longer be applied, at which point we only need to check whether x_3 and y_2 point to the same graph vertex in order to conclude whether both variables have equivalent values. The main difference between this work and Peggy [16] lies in the fact that shared value graphs are reduced in size through rewrite rules, while PEG's are complemented with additional equivalence relations based on constructed equality relations. Both techniques create graphical representations of dependencies between program variables, and both can be used in translation validation.

2.3 Program Analysis Techniques

2.3.1 Model Checking

Model Checking [27] determines whether a given program p satisfies some given property m. In *Bounded Model Checking* (BMC) [28], the program is designated as the model and the property m is a logical formula over the states of p. The idea of bounded model checking is to analyze all the paths with all inputs at once up to some unfolding bound k. BMC creates a single formula that represents each unfolded program for each unfolding factor and then checks whether m is not satisfied in some unfolding. If it is not satisfied, then a counter example is produced explaining why. The formula created by BMC can be for instance a SAT formula or an SMT formula. Consider the program code (a) in Fig. 2.9 and the same code but with different levels of unfolding with `k = 0` in (b), `k = 1` in (c) and `k = 2` in (d). By unfolding

```
void foo(int a, int w) {
  while (a < 10) {
    a = a * 2;
    w = w + 1;
  }
  assert(a < 10);
}
```

(a)

```
void foo(int a, int w) {
  assume(a >= 10);
  assert(a < 10);
}
```

(b)

```
void foo(int a, int w) {
  if (a < 10) {
    a = a * 2;
    w = w + 1;
    assume(a >= 10);
  }
  assert(a < 10);
}
```

(c)

```
void foo(int a, int w) {
  if (a < 10) {
    a = a * 2;
    w = w + 1;
    if (a < 10) {
      a = a * 2;
      w = w + 1;
      assume(a >= 10);
    }
  }
  assert(a < 10);
}
```

(d)

Figure 2.9: Simple while loop in (a) without unfolding. Loop in (a) but with `k = 0` unfolding in (b), `k = 1` unfolding in (c) and `k = 2` unfolding in (d) for bounded model checking.

program (a) with `k = 1` unfolding as shown in (c), we obtain another program where the loop body is executed only once. If we construct an SMT formula describing both the program and the negation of the property m, we can feed this formula into an SMT solver and check whether the property always holds for some unfolding factor k:

$$a_0 < 10 \land a_1 = a_0 \times 2 \land w_1 = w_0 + 1 \land a_1 \geq 10 \land a_1 \geq 10 \tag{2.4}$$

Note the additional $\mathtt{a_1} \geq \mathtt{10}$ that corresponds to the negation of the property m in the final assert. Since this formula is satisfiable, the property m does not hold. Similarly, for `k = 0` unfolding we get:

$$a_0 \geq 10 \land a_0 \geq 10 \tag{2.5}$$

and for k = 2 unfolding:

$$a_0 < 10 \land a_1 = a_0 \times 2 \land w_1 = w_0 + 1 \land a_1 < 10 \ \land a_2 = a_1 \times 2 \land w_2 = w_1 + 1 \land a_2 \geq 10 \land a_2 \geq 10 \tag{2.6}$$

Since the formulas for all three unfolding factors are satisfiable, each one has at least one example for which the property m does not hold.

One problem with BMC is choosing the bound `k`. There is no easy solution because we can not be sure whether there exists a loop whose execution leads to a number of iterations greater than `k`. For example, if we have a loop that generates random numbers and exits when a specific one is found, it is possible to set a value of `k` for BMC while some execution of this loop may consist of `k+1` iterations. However, it is not possible to prove that some property `m` is satisfied for any execution of any program.

The work in [29] discusses the possibility of computing a *completeness threshold* (CT), a precomputed bound at least as high as the highest number of iterations of any loop. Unfortunately computing the CT is as hard as model checking.

CBMC [30] is an implementation of BMC for C and C++ programs. Some applications of CBMC include verification of ANSI-C embedded software [29] and program equivalence checking [31]. In [31], an ANSI-C program is transformed into a bit-vector logic equation consisting of only nested if instructions, assignments, assertions and branch targets. The program is then further transformed into two bit-vector equations, one forming the set of constraints and the other representing the properties to check. Finally, the resulting bit-vectors are translated into *Conjunctive Normal Form* (CNF) and fed into a SAT solver to test for satisfiability. In summary, the equivalence problem is reduced into a bit-vector logic decision problem and then tested for satisfiability with a SAT solver. Similarly, CBMC also constructs a formula $\mathtt{p} \land \neg\mathtt{m}$ and feeds it into a SAT solver.

2.3.2 Symbolic Execution

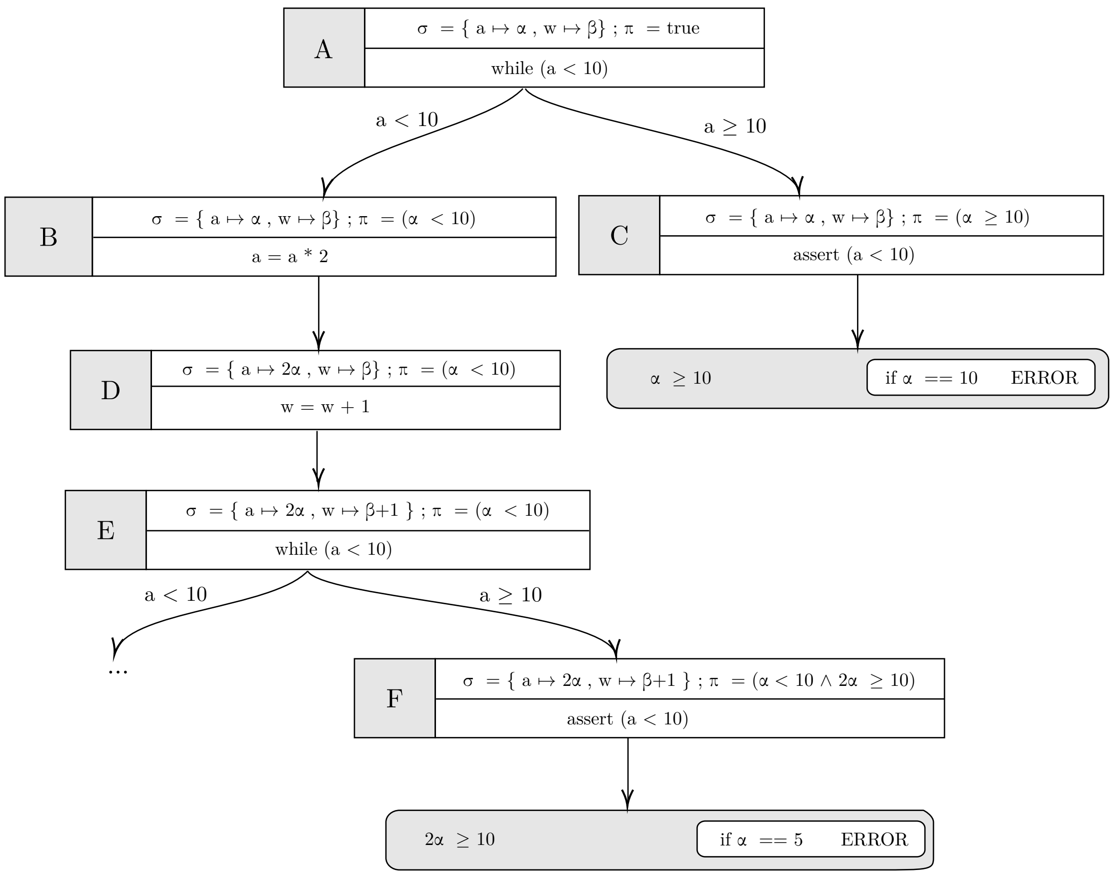
Symbolic execution (SE) [32] is a widely used program analysis technique used to test whether certain properties are violated in a program. In symbolic execution, each input variable is associated with a symbolic variable. Moreover, an execution engine steps through the program from instruction to instruction and builds a SAT formula as a conjunction of path constraints. An example of a path constraint could be the condition of an if statement. If the symbolic engine reaches a branch statement, the symbolic execution state is forked, creating a new state for each of the branch possibilities. Each new state will possess a new set of path constraints π and a new symbolic store σ, which is a mapping between variables and values or symbolic expressions. In particular, if a branch condition $\mathtt{a} = 4$ is satisfied, the $\mathtt{a} = 4$ constraint is added to the path constraints $\pi_{\mathtt{true}}$ of one child of SE, while $\mathtt{a} \neq 4$ is added to the path constraints of $\pi_{\mathtt{false}}$ of the other child. In the case that the execution engine encounters an assignment, the respective mapping for the assigned variable in the symbolic store σ is updated. Additionally, in symbolic execution a model checker can eventually be used to test whether the constraints of a given path are violated. If the model checker finds that the path constraints have been violated, the respective path is pruned, as the path will always be infeasible despite any new constraints that could be added to its path constraints π.

In summary, symbolic execution will simulate an increasing number of program paths simultaneously as new branches are encountered, thus SE is an incremental program analysis technique. This of course leads to a big problem when loops are involved, namely *state space explosion*. For each iteration of a loop the symbolic execution state can be forked more than once and lead to even more states with their own path constraints and symbolic stores. For this reason, maintaining an increasing number of different symbolic stores can incur in a heavy memory cost.

Besides state space explosion and memory concerns, symbolic execution also does not deal well with calls to libraries or system code. Some variations of symbolic execution can mock or ignore calls to code outside the program, but this affects the effectiveness of this technique. Moreover, the model checker that is usually based on *satisfiability modulo theory* (SMT) solvers will eventually test whether any constraints of a path have been violated, which can become quite a burden on the constraint solver when the path constraints become too much to handle.

The survey [33] discusses numerous different techniques towards mitigating the main challenges of symbolic execution. For instance, *selective symbolic execution* (S^2E) [34] is a technique where some paths of the software stack can be explored concretely. Instead of symbolic variables, program variables are associated with specific values similarly to normal program execution. With S^2E, the calls to system code and libraries can be executed concretely, which is a lot faster than doing it symbolically as the path to be taken is pre-defined implicitly by the concrete values assigned to the program variables during execution.

To better illustrate how symbolic execution works, Fig. 2.10 shows the symbolic execution tree for the program with the simple while loop seen previously in Fig. 2.9 (a). Each labeled box represents the state of the symbolic execution engine, each having one symbolic store σ, a set of path constraints π, and the next statement to execute. In state A, the engine encounters a while loop and the execution state is forked. The child that explores the new state with the path constraint $\alpha \geq 10$ brings the engine to an assert statement `assert(a < 10)` which tests a property `m`. Here the model checker tests whether `m` is violated in this path by checking the formula $\alpha \geq 10$ with for example an SMT solver. Note that a corresponds to the value of a when testing the loop condition in state A. In this path the formula is satisfiable, proven by showing the example $\alpha = 10$. As a consequence, the property `m` is violated at least once and the engine has found an error state. Similarly, if the path to B and then to F is tested first, we reach the assert statement with another error state. To see why, consider an example where we have $\alpha = 5$, with α corresponding to `a` having value 5 at state A, and `a = 10` after the first loop iteration, then $2\alpha \geq 10$ is satisfiable.



A
σ = { a ↦ α , w ↦ β} ; π = true
while (a < 10)
a < 10
a ≥ 10
B
σ = { a ↦ α , w ↦ β} ; π = (α < 10)
a = a * 2
C
σ = { a ↦ α , w ↦ β} ; π = (α ≥ 10)
assert (a < 10)
D
σ = { a ↦ 2α , w ↦ β} ; π = (α < 10)
w = w + 1
α ≥ 10
if α == 10 ERROR
E
σ = { a ↦ 2α , w ↦ β+1 } ; π = (α < 10)
while (a < 10)
a < 10
a ≥ 10
...
F
σ = { a ↦ 2α , w ↦ β+1 } ; π = (α < 10 ∧ 2α ≥ 10)
assert (a < 10)
2α ≥ 10
if α == 5 ERROR

Figure 2.10: Symbolic execution tree for the code in Fig. 2.9 (a). Only one iteration of the loop is shown, and two cases where the condition `a` < 10 is false.

2.3.3 Bounded Model Checking vs. Symbolic Execution

Both techniques can be used to check whether a property holds for a given program. The primary differences between bounded model checking and symbolic execution are: (*i*) BMC runs a single SAT formula in order to search for a counter example in executions of the loop unrolled program with an unrolling factor `k`. However, symbolic execution runs SMT queries in parallel, running new ones whenever the symbolic execution engine encounters a branch; (*ii*) BMC is bounded by an integer `k` while symbolic execution will keep running until all feasible paths have been explored; (*iii*) symbolic execution prunes paths that its model checker found infeasible while BMC is forced to keep searching; (*iv*) symbolic execution runs on symbolic inputs as opposed to BMC running with concrete inputs.

Because symbolic execution runs SMT queries in parallel and can prune infeasible paths early, symbolic execution is a more suitable choice towards analyzing programs with a higher number of paths as these paths are more likely to share unsatisfiable SMT prefixes.

Since BMC uses concrete values while symbolic execution uses symbolic variables, symbolic execution is more appropriate towards analyzing programs with undefined behavior, because BMC would be forced to test for every possible value as some operations can return any arbitrary value of a given type. In particular, a variable in LLVM IR may be marked as *undef*, a form of deferred undefined behavior that when read, is allowed to return any value within the range of values that the variable's type can represent.

2.4 Undefined Behavior

If a program produces different results at least once with the same inputs, then the program is non-deterministic. In order to correctly validate a program transformation with translation validation we need to also understand what the sources of non-determinism are and how they work. One type of non-deterministic program behavior is undefined behavior.

Undefined behavior can be classified into three types, (*i*) *immediate undefined behavior* leads to any kind of result when executed; (*ii*) *undef* returns an arbitrary value of a given type when read; (*iii*) *poison* is a stronger variant of undef as it also taints the control flow graph by propagating poison to subsequent instructions. In contrast to immediate UB, undef and poison are deferred, meaning that they do not immediately lead to the execution of UB. Deferred undefined behavior is useful as it allows for some optimizations such as speculative execution to exist. To illustrate the motivation for undef consider the code (a) in Fig 2.11.

```
int x;
if (cond) {
 x = f();
}

if (cond2) {
   g(x);
}
```

(a)

```
a = 2;
if (t != 0) {
 a = 1 / t;
}
```

(b)

```
y = 2 * x; ⇒ y = x + x;
```

(c)

Figure 2.11: A fragment of code motivating *undef* from [35] in (a), a snippet of code in (b) to show how *undef* could be problematic and an incorrect optimization illustrating one detail with undef in (c).

If the condition `cond` is false and `cond2` is true, then `x` is used before it is initialized. In particular for LLVM, an extra instruction is produced for every variable that is used before it is initialized. This can incur quite a performance cost in the long run. Instead, LLVM compiles it such that the uninitialized variables return undef when used, thereby not adding extra instructions and as a consequence avoiding the performance cost.

Undef is not a perfect solution. Consider the code (b) in Fig. 2.11. If `t` is undef, then every read to `t` returns an arbitrary value of its type. Therefore, it is possible that on the first read of `t` returns a value different than zero, and the second read returns zero. In that case, the code (b) in Fig. 2.11 will lead to division-by-zero which is immediate undefined behavior when executed. Moreover, duplicating usages of undef can be dangerous, consider code in (c) in Fig. 2.11 and assume that `x` and `y` are 1 byte-sized unsigned integers. If `x` is undef, then $\texttt{x} \in [0, 255]$ and thus the possible values of `y` would be any even number within [0, 254]. However, if we apply the optimization, the possible values of $\texttt{y} = \texttt{x} + \texttt{x}$ would be in [0, 255] as `x` can be read once as odd and once as even. Therefore, `y` after the optimization can have odd values but not before, thus the transformation is incorrect.

To see why poison is necessary, consider the LLVM IR optimization in Fig. 2.12. If `a` is `INT_MAX` and `b` is 1, we get `INT_MAX + 1 > INT_MAX`, which simplifies to `undef > INT_MAX` because `INT_MAX + 1` is signed integer overflow and considered undefined. This transformation is incorrect as no value of a given

```
%add = add %a, %b
%cmp = icmp sgt %add, %a
        ⇓
%cmp = icmp sgt %b, 0
```

Figure 2.12: An optimization for the LLVM IR that transforms `a + b > a` into `b > 0`.

```
t = x + 1;
if (t == y) {
  w = x + 1;
  foo(w);
}
```

(a)

```
t = x + 1;
if (t == y) {
  foo(y);
}
```

(b)

```
while (c) {
  if (c2) { foo(); }
  else { bar(); }
}
```

(c)

```
if (c2) {
  while (c) { foo(); }
} else {
  while (c) { bar(); }
}
```

(d)

```
if (freeze(c2)) {
  while (c) { foo(); }
} else {
  while (c) { bar(); }
}
```

(e)

Figure 2.13: The GVN optimization with source (a) and target (b), loop unswitching with source (c) and target (d), and a version of loop unswitching adjusted with freeze (e).

type can be greater than the highest value the type can represent. On the other hand if we do apply the optimization we get `b > 0` which simplifies to `1 > 0` which of course is true. In summary, we are getting different results before and after the optimization and thus this transformation is incorrect. To go around this problem, in the particular case for LLVM IR, the `nsw` attribute ("no signed wrap") is added to the *add* instruction in this optimization, which makes the instruction instead return poison on overflow. This allows us to still take advantage of the optimization by deferring the undefined behavior as poison, which can then be propagated to subsequent instructions.

The work [35] proposed a new 'freeze' instruction, now present in the LLVM IR. Freeze is an instruction that either returns an arbitrary value for a given type if the input is poison, or acts as a no-op. To illustrate the authors' motivation for the freeze instruction, consider the code examples in Fig. 2.13. Assuming that (*i*) `c` is false; (*ii*) `c2` is poison; (*iii*) branch-on-poison semantics is immediate UB, then for the *loop unswitching* transformation in (c), we do not branch on `c2` because `c` is false, therefore immediate UB does not occur. In (d) however, because `c2` is poison and since we branch on `c2`, we get immediate UB, therefore the target has different behavior than the source and the transformation is incorrect.

A similar case occurs in the *global value numbering* (GVN) transformation with source (a) and target (b). GVN finds expressions that are equivalent and substitutes them with one of the expressions designated as a representative, in this way removing redundant computations. Assuming branch-on-poison is not immediate UB, and `y` is poison whilst `w` is not, then the function foo is called with a poison argument in (b), but not in (a).

Since GVN and loop unswitching have conflicting branch-on-poison semantics, it would not be possible to apply both on the same code, unless the loop unswitching transformation wraps the outer if statement condition with the new freeze instruction, as shown in (e).

The work [23] presents a way to find and model undefined behavior as invariants, either obtained through a best-effort static analysis or deeper alias analysis when more complex code with memory accesses is involved. One assumption that some compilers take during compilation is the *out-of-bounds variable access* (OBVA) assumption. OBVA states that a program will not access a memory location beyond the region allocated to an object. Consider for instance a program that is vulnerable to array buffer overflow and the compiler applies the register allocation optimization. The register allocation optimization maximizes the use of registers by allocating local variables into registers instead of memory locations in the stack. If the OBVA assumption was not taken by the compiler, then there would be a difference in semantics between the source and the target, as buffer overflow can occur with contiguous memory locations in the stack (source), but not with registers (target). This way the authors soften the equality constraints to tune the translation validation process in accordance with the assumptions that compilers make during compilation.

2.5 Summary

In this chapter we have given an overview on different program verification techniques such as *compiler verification* and *translation validation*. We reviewed some program analysis techniques such as *alias analysis*, *symbolic execution* and *bounded model checking*. Each technique has its own merits and use cases, and all are widely used by many developed program verification tools today.

In addition to visiting different intermediate program representations, we have given an overview on program undefined behavior and how tricky it can make the process of verifying the correctness of compiler transformations.

Chapter 3

Loops

Loops allow the execution of the same sequence of code repeatedly and are common in programs in general. Alive has to be able to analyze them efficiently, however doing so is not an easy task, especially given the limitations of some program analysis techniques such as symbolic execution. As was discussed previously, symbolic execution with loops can result in the traversal of a very large number of blocks, leading to memory usage problems.

Because of these limitations, Alive currently only traverses each vertex in the loop at most once by analyzing a *directed acyclic graph* (DAG) version of the original program. A DAG is a graph without any cycles and with no back edges. As a consequence of ignoring back edges, Alive can miss some vertices or paths in its analysis of programs if a program exit is not reachable from these vertices in the DAG.

Definition 1. A vertex v is reachable from vertex u if there exists a path starting from u that traverses v.

Definition 2. For a given topological ordering of vertices in V, an edge $(u,\ v) \in E$ is a back-edge if the topological order of vertex v is lower than or equal to the topological order of vertex u.

Definition 3. For a given topological ordering of vertices in V, an edge $(u,\ v) \in E$ is a forward-edge if vertex v has a greater topological order than u.

Consider the simple example in Fig. 3.1, the path $entry \rightarrow A \rightarrow exit$ will be covered by Alive, however, the path $entry \rightarrow A \rightarrow B \rightarrow A \rightarrow exit$ will not as the edge $B \rightarrow A$ is replaced before symbolic execution. Later we will present an algorithm that will be capable of unrolling the program before Alive's symbolic execution, by duplicating vertices in a certain order, and consequently increasing both the *block coverage* and *path coverage* of Alive's analysis, two of the most important metrics by which our unrolling algorithm will be evaluated.

Definition 4. We say that a basic block b in some program p is covered by Alive if it is traversed by some covered path. A path is covered by Alive if it starts from program entry and reaches some program exit without traversing any back edges.

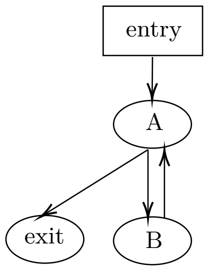


Figure 3.1: Simple loop example with A as the loop header. We have a total of four edges, from which only the edge (B,A) is a back-edge.

Definition 5. We define *block coverage* as the percentage of the total number of different basic blocks of a program *p* that are covered by Alive. Any duplicate *d* of some basic block *b* where *d* is inserted during loop unroll is considered the same block as *b*.

Definition 6. We define *path coverage* as the number of different paths of a program *p* covered by Alive.

We have *total block coverage* if every different basic block in a program *p* is covered by Alive. Total block coverage is not always possible, for instance if a program contains a basic block *b* with the *unreachable* LLVM IR instruction, and thus any path that traverses *b* will not reach a program exit.

Total path coverage is only possible to obtain for a program *p* if one of the following two conditions are met: (*i*) *p* has no loops or (*ii*) every loop in *p* does not run for some number of iterations that is greater than the unrolling factor *k* given to the loop unrolling algorithm, resulting in total path coverage only for the unrolled program *p'*.

The loop unrolling algorithm that we will present in Section 3.5 adds basic blocks without the guarantee that these will be covered by Alive. To evaluate whether loop unroll improved program coverage of Alive's analysis, metrics such as *block coverage* and *path coverage* are useful.

Before we can move on to providing solutions, we must first define what a loop is and what it consists of. We begin by defining a loop as a triple *ℓ = (h, B, X)*, consisting of a single loop header *h*, the loop body *B* and the loop exiting blocks *X*, which are included in the loop body *B*.

Definition 7. The loop header h of a loop $\ell = (h,\ B,\ X)$ in a program p with the CFG = $(V,\ E)$ is the loop entry vertex of ℓ such that:

$$\exists\ u, v, w \in V\ :\ u \notin \{h\} \cup B\ \wedge\ v, w \in \{h\} \cup B\ \wedge\ u, v \in E^{-1}[h] \wedge\ w \in E[h] \tag{3.1}$$

The loop header h must have at least two incoming edges from two different sources, one from within the loop and another from a vertex outside the loop. Moreover, it must also have at least one outgoing edge directed at a vertex within the loop. Note also that h is not always also a loop exiting block, it is possible to have multiple loop exiting blocks without any of them being the loop header.

Definition 8. The loop exiting blocks $e \in X$ of a loop $\ell = (h,\ B,\ X)$ satisfy:

$$\exists\ u \in E[e]\ :\ u \notin \{h\} \cup B \tag{3.2}$$

Definition 9. Any successor of a loop exiting block not in the loop is called a loop exit block.

Definition 10. A loop ℓ is *reducible* if it has exactly one loop entry.

Definition 11. A loop ℓ is *irreducible* if it has more than one loop entry.

To better understand the difference between reducible and irreducible loops, consider the two CFG's in Fig. 3.2. Note that the basic block *entry* is a program entry block and not a loop entry block.

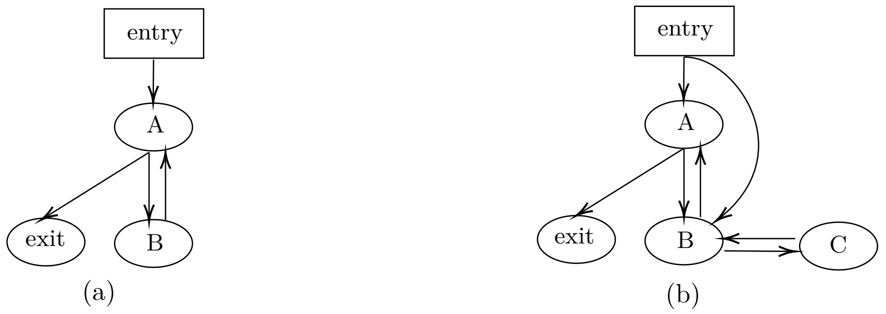


Figure 3.2: (a) A single reducible loop with A as header. (b) Two loops, one irreducible with header A and the second reducible loop with header B. The loop is irreducible because of the edge (entry, B) which is another loop entry to the loop in addition to the loop entry (A, B).

Definition 11 contradicts what we defined as a loop where it only has a single loop header. In the case for irreducible loops it would instead have a set of loop headers, however, in our approach we will designate a single header for each loop as the loop's representing vertex.

In this next section, we will go over an algorithm that allows us to identify reducible and irreducible loops in graphs. The results obtained through this algorithm will be the starting point of our own unroll algorithm. Unrolling will be vital in increasing Alive's block and path coverage.

3.1 Nested Loops

In addition to finding which vertices in a program make up a loop, we also want to look for how they are nested.

Definition 12. We define a loop nest $\mathcal{L}$ as a tree of loops $(\ell_0, (V, E))$ where the root ℓ_0 is the outer-most loop, V is the set of loops and E represents the nesting relationship between loops.

Definition 13. Given two loops ℓ_1 and ℓ_2 in some loop nest $\mathcal{L}$, we say that ℓ_2 is nested in ℓ_1 or ℓ_2 is a child loop of ℓ_1 if ℓ_2 is a descendant of ℓ_1 in $\mathcal{L}$. If $\ell_2 = (h_2, B_2, X_2)$ is nested in $\ell_1 = (h_1, B_1, X_1)$ then:

$$\forall\ u \in \{h_2\} \cup B_2 : u \in B_1 \tag{3.3}$$

Definition 14. We define the loop nesting depth d of a loop $\ell \in \mathcal{L}$ with $\mathcal{L} = (\ell_0, (V, E))$ as the path starting from ℓ_0 with a minimum number of edges $e \in E$ that need to be taken to reach ℓ.

Definition 15. We define the *degree* of a loop nest $\mathcal{L}$ as the highest loop depth of a loop $\ell \in \mathcal{L}$.

In theory loop nesting can involve an arbitrary number of loops. Knowing how loops are nested is necessary to minimize the number of times we unroll a given loop. If we unroll the outer-most loop ℓ_0 of a loop nest $\mathcal{L}$ first, then we are creating copies of other loops $\ell_i \in \mathcal{L} = (\ell_0, (V, E))$ nested in ℓ_0 that also need to be unrolled. If we instead unroll each loop in descending order of loop nesting depth d then we only need to unroll $|V|$ loops.

It is interesting to note that an irreducible loop ℓ_2 nested in another loop ℓ_1 does not guarantee that ℓ_1 would also be irreducible, since all entries to ℓ_2 that make it irreducible could be included in the body of ℓ_1. To illustrate this see Fig. 3.3.

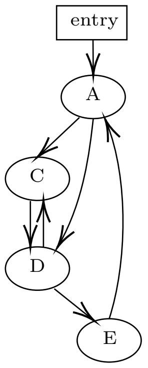


Figure 3.3: An irreducible loop ℓ_2 with header C nested in a reducible loop ℓ_1 with header A. ℓ_1 is reducible due to the single loop entry in A, but ℓ_2 is irreducible because of the edges (A,C) and (A,D).

3.2 Loop Identification

To reliably identify and classify loops in programs and see how they relate to one another, we use and extend on a loop identification algorithm developed in Havlak [36]. This algorithm can be viewed as a two-step process:

1) A *depth-first search* traversal (DFS) to divide the edges of a program $E(\text{p})$ into these two disjoint sets: the forward edges $\text{E}_\text{f}(\text{p})$ and the back edges $\text{E}_\text{b}(\text{p})$. An edge is classified either as a back-edge if the destination has already been visited during traversal, or forward edge otherwise. A control-flow graph (CFG) that can be partitioned into two sets and does not involve any irreducible loops is called a reducible control-flow graph.

If there exists a loop $\ell = (h, B, X)$ in the program, then finding it requires identifying at least one back-edge (u, v) where $u \in B \land h = v$.

Depending on the DFS traversal, a back-edge may or may not be found by the algorithm, leading to drastically different results depending on the *depth-first search spanning tree* DFST, obtained from visiting each vertex once with a DFS. If we would for example reorder the targets of some program vertex, even though the program has the exact same semantics, the DFS will find different back-edges and consequently find different loops. In addition to the variability of loop identification, the number of back-edges found can also depend on the DFS, causing some DFS traversal orders to find more loops than others, or even none at all.

Finding the back edge set $E_b(p)$ is key to finding the headers of each loop in addition to body and exiting vertices. Fortunately, the fact that results depend on the order of the DFS traversal is not true when programs only exhibit reducible loops, which are a lot more common than irreducible loops. Irreducible loops are rare and require an unstructured form of control flow. Such unstructured control-flow can be obtained by for example using the *goto* instruction in a language like C or C++.

2) The second and final step of the algorithm is to look at each identified loop header in reverse preorder, so the loop headers found last in the DFS are analyzed first, and building the vertex sets that contain all the vertices that are part of each loop. The preorder numbering is built in step 1 and represents an ordering between vertices such that descendants have a greater preorder than their ancestors.

3.3 Multiple Loops per Header

The algorithm discussed in Havlak [36] also presents an additional procedure called *fix_loops*. This procedure attempts to address one of the algorithm's limitations in which it cannot identify more than one loop for each vertex in the graph, as it chooses a single vertex as the unique representative and header of each loop. This is also why in our approach we also make this same convention of having only loops that don't share loop headers. The procedure *fix_loops* inserts new intermediate vertices when an irreducible loop is found, followed by a second DFS in order to update both the forward and backward edge sets and the preorder numbering with respect to the new graph. These extra vertices give additional choices for loop headers of irreducible loops and makes it possible to identify more loops.

In our case we have decided to not implement this procedure because the vast majority of programs that Alive analyzes have the same control-flow patterns. These programs have the same structure of basic blocks and edges that we would get from using for example the keywords *for*, *while*, *if* in the C programming language. In practice, only a minority of programs use the *goto* instruction with which more irregular control flow can be obtained. Such control flow may contain strangely structured loops where algorithms for loop identification or loop unrolling may not produce the best results.

We do believe that it would be a good idea to experiment with *fix_loops* and understand how much Alive can benefit from its implementation, but doing so would require a significant amount of time.

3.4 Loop Trees

In addition to running the algorithm to find loops, we also organize these loops into a *Loop Tree.* A loop tree is a graph $(v_0, V \times V)$, similar to a loop nest, but with two differences: (i) the root v_0 of the tree is the program entry and (ii) the vertices V are each loop's header with the possible exception of the root v_0, which may or may not be a loop header. An example of a loop tree can be seen in Fig. 3.4.

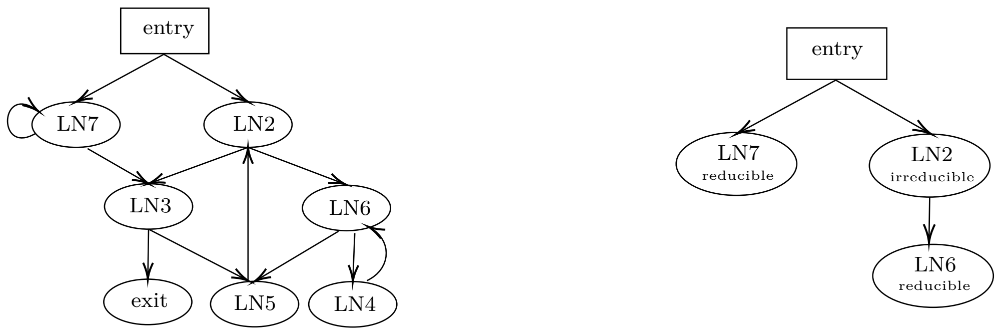


Figure 3.4: Example program CFG on the left with a Loop Tree represented on the right, obtained from running the loop identification algorithm presented in [36] on the program on the left. Since we have at least one irreducible loop, the loops identified may vary.

Organizing loops into a tree structure helps keep our algorithm simple and ensures that we can process each loop in a nested-first order obtained through traversing this tree with a reverse DFS, in order to minimize the number of times we run our loop unroll algorithm.

3.5 Loop Unrolling

In the previous sections we have discussed a loop identification algorithm capable of not only finding reducible and irreducible loops, but also gather information about how they are nested. We now move on to discussing our new approach to perform loop unroll on programs in order to increase both block and path coverage of Alive's analysis while also minimizing the number of vertices that need to be duplicated.

3.5.1 Reducible Loops

Before we dive into our approach, let's first go through a simple example with only reducible loops shown in Fig. 3.5. We designate ℓ_1 as the loop with header LN2 and ℓ_2 as the loop with the header LN6.

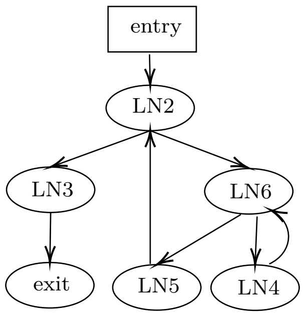


Figure 3.5: A control-flow graph with two nested reducible loops.

Here we can see that the loop ℓ_2 is nested in ℓ_1, since any path that reaches LN6 from *entry* must go through LN2. If we were to let Alive analyze this CFG without any prior unrolling, Alive would see a DAG like the one in Fig. 3.6.

Without any unrolling, only the following path is covered:

$$\text{entry} \rightarrow \text{LN2} \rightarrow \text{LN3} \rightarrow \text{exit}$$

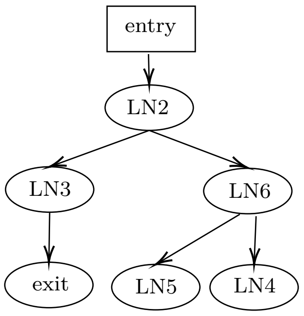


Figure 3.6: A control-flow graph with two nested reducible loops and without back edges.

Therefore the block coverage for this example without any prior unrolling is about 57 % (4/7). If we look back at the control-flow graph in Fig. 3.5, we see that the following uncovered path is both the shortest path required for LN5 to be covered and requires visiting LN2 twice.

$$\text{entry} \rightarrow \text{LN2} \rightarrow \text{LN6} \rightarrow \text{LN5} \rightarrow \text{LN2} \rightarrow \text{LN3} \rightarrow \text{exit}$$

Since any back-edges in Alive are replaced, to cover this path we must duplicate LN2 at least once. One method of doing this would be to transform the CFG into what we have in Fig. 3.7, where we have a duplicate of LN2 and where the original back-edge (LN5, LN2) was replaced with the edge (LN5, LN2').

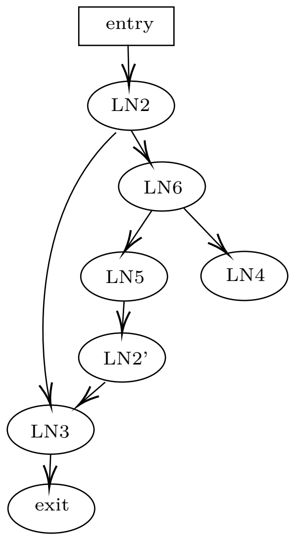


Figure 3.7: A control-flow graph with two nested reducible loops, where LN2 was duplicated.

Besides the previously covered path, we now also cover the path:

$$\text{entry} \rightarrow \text{LN2} \rightarrow \text{LN6} \rightarrow \text{LN5} \rightarrow \text{LN2'} \rightarrow \text{LN3} \rightarrow \text{exit}$$

With this new graph, we have improved block coverage by now covering the vertices LN6 and LN5, however, we still do not cover LN4. By applying the same reasoning as done for LN2 but on LN6 we obtain the graph in Fig. 3.8.

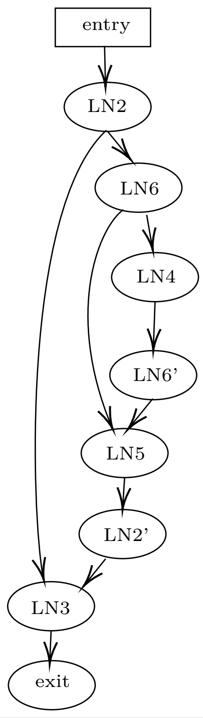


Figure 3.8: A control-flow graph unrolled with a factor k of 1.

We now have reached the point where the graph was unrolled such that the body of each loop is covered at least once. We call this having unrolled the program with an unroll factor k of at least 1. We now cover each and every basic block in the program at least once in some path covered by Alive, thus we achieved total block coverage for this example with the least number of blocks duplicated, in other words, the obtained graph is the optimal result for $k = 1$.

3.5.2 Loop Unroll Factor

Although we now cover all basic blocks of the program, we can still improve path coverage by increasing the unroll factor k. For $k = 2$, we would obtain the graph in Fig. 3.9 where each loop body would be covered at least twice.

It may seem tempting to further increase the unroll factor k, however the size of the resulting unrolled graph would grow significantly with nested loops. To illustrate, consider the case where we have three reducible loops, ℓ_1, ℓ_2, and ℓ_3, where ℓ_3 is nested in ℓ_2 and ℓ_2 is nested in ℓ_1. The loop nesting depth of loops ℓ_1, ℓ_2 and ℓ_3 are 0, 1 and 2 respectively. Assume also that we unroll loops in an inside-out order, where ℓ_3 is unrolled first, then ℓ_2 followed by ℓ_1. For a given unroll factor k, unrolling only ℓ_3 would result in having k iterations of the body of ℓ_3, which becomes the new loop body of ℓ_2. If we then unroll ℓ_2 with the same value of k, we obtain k iterations of its loop body, which now also already includes k iterations of the loop ℓ_3. Therefore unrolling ℓ_2 will lead to having a total of k^2 iterations of the body of ℓ_3. Applying the same process to ℓ_1 the final number of iterations of ℓ_3 becomes k^3. Generalizing, if we have some loop $\ell_i \in \mathcal{L}_j$ with a loop nesting depth of d, then the unrolled program will have at least k^{d+1} iterations of the loop ℓ_i. For large programs with a sufficiently high loop nesting degree, the resulting program may become too large, resulting in large SMT expressions that cannot be solved in a reasonable amount of time.

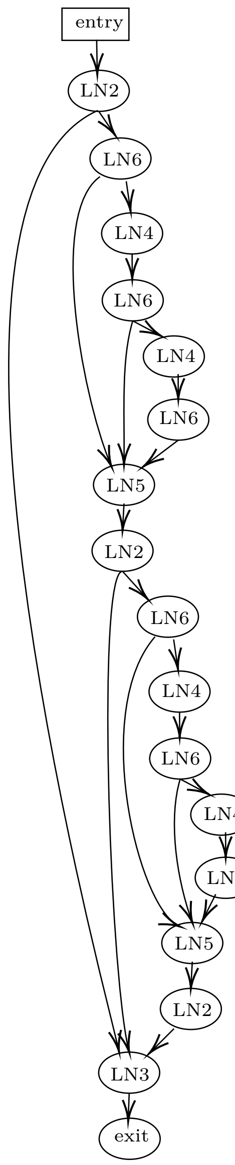


Figure 3.9: A control-flow graph unrolled with a factor k of 2.

3.5.3 Algorithm

A general overview of our algorithm is shown in Fig. 3.10. The algorithm starts by ordering all loop headers in topological order, such that a predecessor of some header h in LT always appears before h. Recall that in a loop tree any loop header h_2 with h_1 as predecessor is nested in h_1. For performance reasons we want to run our algorithm as less often as possible, therefore we should first unroll loops with highest nesting degree. To achieve this we can unroll the loops in reverse topological order, as described in lines 11-13, where each loop is unrolled separately in a call to LoopUnrollSingle.

The function LoopUnrollSingle shown in Fig. 3.11 describes how a loop is unrolled. This algorithm can be seen as two parts, duplication of the loop header and its edges, and duplication of vertices in the loop body and edges. The loop body is only duplicated for values of $k > 1$, since reaching the body again requires control flow to traverse an edge to the header or to a copy of the header. Moreover, the header is always the last vertex that we try to duplicate in a given iteration of loop unroll.

```
1  function BlockLoopUnroll($CFG$, $LT$, $k$)
2  input
3     $CFG : (V, E)$ - control flow graph as pair of vertices and edges
4     $L : \mathcal{P}(V)$ - loop header vertices
5     $LT : (v_0, L \times L)$ – Loop Tree
6     $k : \mathbb{N}_0$ - unroll factor
7  vars
8     $GetLoopset : L \rightarrow \mathcal{P}(L)$ - set of loop vertices given header
9     $stack : \mathcal{P}(L)$ - vertices pending processing
10 begin
11    $stack := Topsort(L)$ - $L$ in topological order
12    while $stack \neq \emptyset$ do
13       LoopUnrollSingle($CFG$, $GetLoopset(stack.pop())$, $k$)
```

Figure 3.10: Unroll of all loops in a program which are represented in a loop tree LT.

Variable *last* is a key component to pointing new edges to their correct destination. This variable keeps track of the last duplication of a given vertex, initially set to the identity function.

The function *dup* creates a copy of a given vertex and adds this vertex to the body of all parent loops in which the current loop is nested in.

After each iteration of the k loop, we take the previous iteration's loop as a reference to guide the creation of edges in the new iteration. Variable *hprev* designates either the loop header before the first iteration of loop unroll or the last duplication of the loop header before the new iteration.

The duplication of the header is simple, we first create a copy *h'* of the header *h*. Then we look at the edges entering and leaving *hprev* to create the new edges for *h'*. Lines 27-29 describe back edges to *hprev* being replaced with forward edges to *h'*. Lines 30-31 describe the edges leaving *h'*, either added as back-edges into the loop, or as a forward edges pointed at a vertex outside the loop. If we apply these steps on the program CFG described in Fig. 3.5 with $k = 1$ we obtain the CFG shown in Fig. 3.8.

If we have $k > 1$, then the algorithm also duplicates the body and its edges, consisting in three steps. First we duplicate each vertex without any edges in lines 14-16.

In the second step, we add new edges leaving each new body vertex pointing to either another vertex in the body or to a vertex outside the loop. No back edges are added in this step.

Lastly, we replace the back edges added in lines 30-31 with forward edges from the new header *hprev* into the new vertices of the loop. Note that *hprev* with $i = 1$ is *h'* from the previous unroll iteration. Running this algorithm with $k = 2$ on Fig. 3.5 results in the CFG shown in Fig. 3.9.

Loops with Empty Body

This algorithm assumes that each loop will have a non-empty body, therefore some preprocessing is needed. For each vertex u with at least one edge pointing to u, we insert a new vertex v representing the body of this loop. All edges (u, u) are replaced with edges (u, v) and one additional edge (v, u) is created.

```
1  function LoopUnrollSingle($CFG$, $L$, $k$)
2  input
3     $CFG$ : ($V$, $E$) - control flow graph as pair of vertices and edges
4     $L$ : ($V$, $\mathcal{P}(V)$, $\mathcal{P}(V)$) - loop header $h$, body $B$ and exiting blocks $X$
5     $k$ : $\mathbb{N}_0$ - loop unroll factor
6  vars
7     $last$ : $V \times V$
8  begin
9     $last := id$
10    $hprev := h$
11    $LoopSet := \{h\} \cup B$
12    for $i \leftarrow 0$ to $k$ - 1 do
13       if $i > 0$ then
14          for each $u \in B$ do
15             $u' := dup(u)$
16             $last[u] = u'$
17          for each $u \in B$ do
18             for each $v \in E[u]$ do
19                $E[last[u]] := E[last[u]] \cup \{last[v]\}$
20          for each $u \in (E[hprev] \cap LoopSet)$ do
21             $E^{-1}[u] := E^{-1}[u] \setminus \{hprev\}$
22             $E[hprev] := E[hprev] \cup \{last[u]\}$
23       if $i = k - 1 \wedge h \in X$ then
24          break
25       $h' := dup(h)$
26       $last[h] := h'$
27       for each $u \in (E^{-1}[hprev] \cap LoopSet)$ do
28          $E^{-1}[hprev] := E^{-1}[hprev] \setminus \{u\}$
29          $E[last[u]] := E[last[u]] \cup \{h'\}$
30       for each $u \in E[hprev]$ do
31          $E[h'] := E[h'] \cup \{last[u]\}$
32       $hprev := h'$
```

Figure 3.11: Loop unroll pseudocode to unroll a single loop L with an unroll factor of k.

Loop Headers and Loop Exiting Blocks

The header is always the last vertex we look at during each iteration of loop unroll, however it is not always duplicated. Consider the program CFG in Fig. 3.12.

In order to increase path coverage through loop unroll, any additional paths created must still traverse a loop exiting block to reach a program exit. Since A is not a loop exiting block, duplicating it will not result in an increase in path coverage. Lines 27-29 in LoopUnrollSingle skip duplication of the header if it is not a loop exiting block and if we are in the last iteration of loop unroll.

3.5.4 Instruction Patching

The algorithm presented in Section 3.5.3 unrolls program loops by adding new basic blocks and updating the control-flow jumps between them. However, these programs are written in SSA form, where new

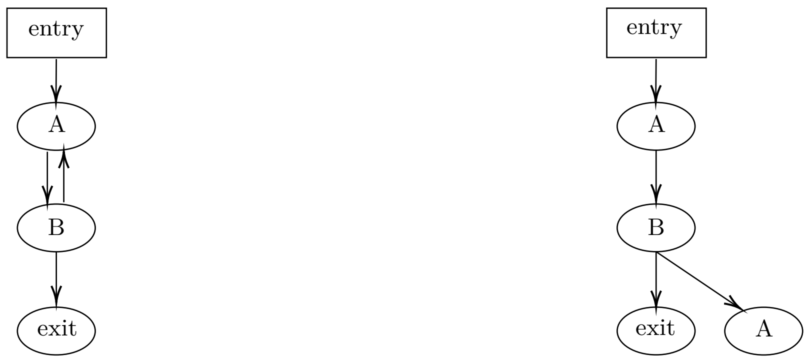


Figure 3.12: A program CFG on the left and a unrolled CFG on the right with $k = 1$ respective to the CFG on the left.

variables must be defined to represent changes to previously defined variables. For example, consider a semi-unrolled program with the control-flow graph shown in Fig. 3.13, where (a) is the DAG that Alive creates when given the program CFG in Fig. 3.5.

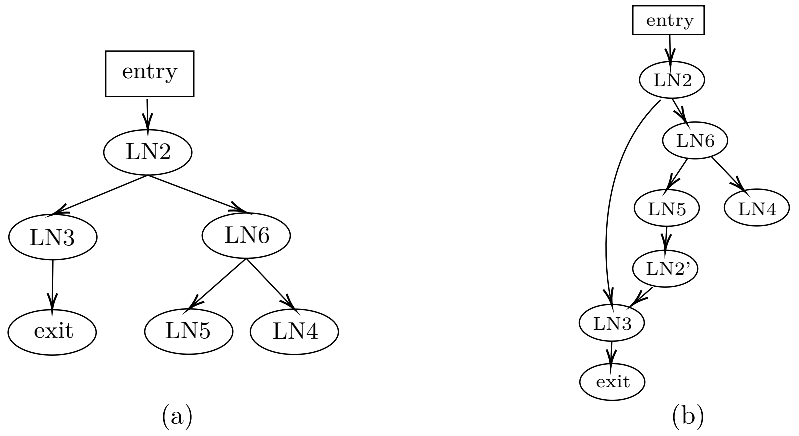


Figure 3.13: A program control flow graph (a) and its semi-unrolled control flow graph (b).

In this graph, only LN2 was duplicated, resulting in the insertion of the basic block LN2'. In (b), if some variable $\texttt{x}_1$ is defined in LN2, then a new variable $\texttt{x}_2$ is also defined in LN2', such that both $\texttt{x}_1$ and $\texttt{x}_2$ refer to the same variable `x` in (a). If we now have an instruction $\texttt{instr}_1$ in LN2, which takes in $\texttt{x}_1$ as an operand, then the corresponding instruction $\texttt{instr}_2$ in LN2' must take $\texttt{x}_2$ as an operand, with $\texttt{x}_2$ refering to the value of `x` in the new loop iteration.

In addition to updating the operands of instructions, another complication arises from unrolling loops written in SSA form. Consider now an instruction $\texttt{instr}_3$ in LN3, that takes in `x` as an operand. In (a), we know that `x` refers to the value defined for it in LN2. In (b) however, this same instruction should be able to take either $\texttt{x}_1$ from LN2, or $\texttt{x}_2$ from LN2' since LN3 in (b) now has LN2' as an additional predecessor compared to LN3 in (a). To differentiate between $\texttt{x}_1$ from LN2 and $\texttt{x}_2$ in LN3 in (b), we need to insert an additional phi instruction into LN3. Each instruction that defines a variable in the loop may require a phi instruction to be inserted in some of its successors that are outside of the loop. Moreover, the insertion of new phi instructions can require more phi instructions to be needed. For instance, a phi inserted into some basic block A outside of a loop ℓ_2 can result in the definition of a variable y in some other loop ℓ_1 that contains ℓ_2, meaning that ℓ_2 is nested in ℓ_1. Once ℓ_1 is unrolled, the definition of this new phi is duplicated, and may need to be differentiated with other new phi instructions in basic blocks outside of ℓ_1.

```
function LoopUnroll($CFG$, $LT$, $k$, $instrs$, $vars$)
input
    $CFG : (V, E)$ - control flow graph as pair of vertices and edges
    $L : \mathcal{P}(V)$ - loop header vertices
    $LT : (v_0, L \times L)$ – Loop Tree
    $k : \mathbb{N}_0$ – loop unroll factor
    $instrs : V \rightarrow \text{List}(Instr)$ – list of instructions for a vertex
    $vars : \mathcal{P}(Instr)$ – set of instructions that define variables
vars
    $addedPhis : V \times \mathcal{P}(Instr)$ – set of Phi instructions created for each vertex
    $vertices : \mathbb{N}_0 \times V$ – an array of vertices
    $phiRef : Instr \times Instr$ – original value that each phi decides
    $foriginal : V \times V$ – vertex from which another vertex was copied from
    $idupes : Instr \times V \times \text{List}(Instr)$ – list of instruction duplicates ordered topologically by vertex
begin
    $foriginal := id$
    $addedPhis : \lambda\ v\ . \rightarrow \emptyset$
    for each $v \in L$
        $\langle h, B, X\rangle :=$ GetLoop($v$)
        $exits := \emptyset$
        $loopset := \{\text{h}\} \cup B$
        for each $u \in loopset$
            for each $w \in E[u]$
                if $w \notin loopset$
                    $exits := exits \cup \{w\}$
        for each $u \in loopset$
            for each $instr \in instrs[u]$
                for each $e \in exits$
                    if $instr \in vars \wedge KnowsVar(e, instr)$
                        $addedPhis[u] := addedPhis[\text{u}] \cup \{\text{CreatePhi}(e, instr, instrs, phiRef, idupes, addedPhis)\}$
    BlockLoopUnroll($CFG$, $LT$, $k$, $foriginal$)
    $vertices :=$ Topsort($V$)
    for each $v \in vertices$
        for each $phi \in addedPhis[v]$
            for each $u \in E^*[v]$
                if $|E^{-1}[u]| > 1$
                    $predVals := \emptyset$
                    for each $w \in E^{-1}[u]$
                        $predVals := predVals \cup \{\text{GetUpdatedVal}(w, phiRef[phi])\}$
                    if $|predVals| > 1$
                        $addedPhis[u] := addedPhis[u] \cup \{\text{CreatePhi}(u, phiRef[phi], instrs, phiRef, idupes, addedPhis)\}$
    UpdatePhiEntries($vertices$, $phiRef$, $addedPhis$, $foriginal$, $idupes$)
    UpdateNonPhiInstructionOperands($vertices$, $instrs$, $idupes$)
    CleanupUnusedPhiInstrs($vertices$, $instrs$)
```

Figure 3.14: Loop unroll pseudocode to unroll all loops in a program and update program instructions.

The algorithm we implemented to update instruction operands as well as to add the necessary phi instructions into the unrolled program is shown in Fig. 3.14. In general, this algorithm does four important things. First, some phi instructions are inserted in basic blocks that are expected to need them. Afterwards, the program loops are unrolled with a call to the algorithm BlockLoopUnroll, presented in Fig. 3.10. Next, the remaining phi instructions that are needed are inserted, followed by a final cleanup of instructions that were left unused.

To check whether some phi instruction is needed in a given basic block, we need to traverse the control flow graph of the program and look at their instructions and their corresponding uses. Since this is too expensive to do for each instruction, we take a more optimistic approach by adding new phi instructions for loop variables without making most of these checks. This does mean that we could be adding more phi instructions than are needed. Fortunately, this is not an issue because any unused instruction can be easily and efficiently removed after the algorithm finishes.

When some basic block is duplicated, all its instructions are copied, therefore it is more efficient to divide the phi insertion logic into two parts, before and after calling the BlockLoopUnroll algorithm. Here a significant portion of the phi instructions are added to only the basic blocks that were present before loop unroll. Depending on the loop unrolling factor used, the number of new basic blocks inserted as a result of loop unrolling can be very large. In addition to performing less checks, we can also reduce the number of phi instructions we manually insert by taking advantage of the fact that the duplication of a given basic block also copies all its instructions. This means that if we duplicate a basic block with a new phi instruction, it will be present in the duplicate basic block for free. The remaining phi instructions that are needed due to these insertions are added after the call to BlockLoopUnroll.

To understand the algorithm in more detail, we have divided it here into seven different steps with their respective line numbers, accompanied by a brief explanation of the reasoning behind each step:

1. (ln. 24-27) We first identify the basic blocks that may need additional phi instructions. These blocks are designated as the loop *exits*. If any phi instruction is needed for a variable defined inside the loop, then they should be added to one or more of these basic blocks.

2. (ln. 28-32) For each variable defined by an instruction inside the loop, add a phi instruction into each of the loop's *exits* which have the basic block that defines each variable as an immediate predecessor. If any one of these phi instructions ends up unused, it will be removed in a later step. The function CreatePhi is shown in Fig. 3.15. In this section we use the terms *vertex* and *basic block* interchangeably.

```
function CreatePhi($v$, $instr$, $instrs$, $phiRef$, $idupes$, $addedPhis$)
    $phi$ := Phi($instr$)
    $instrs[e]$.pushFront($phi$)
    $addedPhis[u] := addedPhis[u] \cup \{instr\}$
    $phiRef[phi] := instr$
    $idupes[instr]$.pushBack($phi$)
    return $phi$
```

Figure 3.15: Create a phi instruction and initialize necessary data structures.

The *instrs* data structure is used to keep an ordered list of instructions present in each basic block. The array of sets *addedPhis* is used to keep track of all new phi instructions inserted in each basic block. The *phiRef* data structure keeps a reference to the variable that each new phi instruction decides, that is, given different redefinitions of some variable, choose the correct definition depending on the predecessor. Lastly, each instruction keeps a reference to its origin instruction, which is the instruction from which this one was copied from. The *idupes* data structure keeps an ordered list of these duplicates for each instruction in topological order. Both *phiRef* and *idupes* are needed to find the correct values to use in each entry of each new phi instruction. The KnowsVar function checks if any path reaching this block has seen the definition of a previous version of the given variable, preventing the algorithm from adding instructions that take variables as operands that are yet to be defined.

3. (ln. 33) The program is unrolled on a basic block level by calling our loop unrolling algorithm BlockLoopUnroll. Each new basic block has a copy of the instructions in the block it was copied from, with one minor difference. The instruction operands are kept the same, however, any variable defined by each of these instructions will have a new name. We leave the updating of instruction operands to be done after all phi instructions have been inserted.

4. (ln. 35-43) As we have mentioned earlier, when we have nested loops we may need to add additional phi instructions. For each of the already added phi instructions, we add additional phi instructions to basic blocks that need them. We identify these by checking whether there is more than one different redefinition for a variable among all its immediate predecessors. The function GetUpdatedVal finds the last definition of a variable for all paths reaching a given vertex *v*. If the variable passed in is a constant, it acts as a no-op. This function is shown in Fig. 3.16.

```
function GetUpdatedVal(instr, v, idupes)
   for i := idupes[instr].size() - 1 until 0; do
      val := idupes[instr][i]
      if KnowsVar(v, val)
         return val
   return instr
```

Figure 3.16: Get the most recent duplicate of a given instruction, else return reference instruction if none found.

5. (ln. 44) To keep our algorithm simple, each phi instruction that we insert will initially have no entries, which are the values to choose from for each predecessor of the basic block that contains the phi. In this step the correct entry for each immediate predecessor is added, which is shown in Fig. 3.17. For each phi instruction, we build a set *preds* containing the predecessors for which an entry is needed (ln. 5-7). Afterwards, we identify the value to use for each entry respective to each predecessor. We do this in two different ways, depending on whether the phi is new. If it is new, then we pick the reference variable in *phiRef*[*phi*], and call GetUpdatedVal to retrieve the correct value for the phi entry to create. Otherwise, we pick the reference variable for the origin of the predecessor (*foriginal*[*u*]), and call GetUpdatedVal on this variable and predecessor to obtain the right value. In our loop unrolling algorithm, each basic block also has a set origin, which is the block itself if it is not a duplicated block, or it is the basic block from which it was duplicated. Once the correct value is retrieved (ln. 10), we create and add the entry (ln. 11).

```
function UpdatePhiEntries(vertices, phiRef, addedPhis, foriginal, idupes)
  for each $v \in$ vertices
    for each instr $\in$ instrs[v]
      if instr is a Phi
        preds := $E^{-1}[v]$
        for each $\langle val, u \rangle \in$ instr.entries
          preds := preds $\setminus \{u\}$
        instr.entries.clear()
        for each $u \in$ preds
          val := instr $\in$ addedPhis[v] ? phiRef[instr] : instr.entries[foriginal[u]]
          instr.entries := instr.entries $\cup$ {(GetUpdatedVal(u, val, idupes), u)}
      else
        break
```

Figure 3.17: Update entries of each phi instruction.

6. (ln. 45) In this step we update the operands of each non phi instruction as shown in Fig. 3.18, after all needed phi instructions have been inserted.

```
function UpdateNonPhiInstructionOperands(vertices, instrs, idupes)
  for i := 0 until vertices.size() - 1; do
    for each instr $\in$ instrs[vertices[i]]
      if instr is not Phi
        for each op $\in$ instr.operands
          op := GetUpdatedVal(op, vertices[i], idupes)
```

Figure 3.18: Update the operands of each non-phi instruction.

To update the operands of each instruction, we only need to call GetUpdatedVal on each operand and replace it with the returned value.

7. (ln. 46) Lastly, we remove any phi instruction that is left unused. We show how to do this in Fig. 3.19.

```
function CleanupUnusedPhiInstrs(vertices, instrs)
  for i := vertices.size() - 1 until 0; do
    for j := instrs[vertices[i]].size() - 1 until 0; do
      if instrs[j] is Phi $\wedge$ GetUsers(instrs[j]) = $\emptyset$
        instrs[vertices[i]].erase(instrs[j])
```

Figure 3.19: Remove unused phi instructions.

The variable *vertices* is an array of vertices in topological ordering. We traverse the vertices in reverse topological order and check if there exists any use of the variable that each phi instruction defines. If no use is found then the instruction is removed. As a side effect, this also removes the instruction as a user of the operands it contained, which is the main reason why we traverse these vertices in reverse topological order, as we only need to visit each vertex once. The function GetUsers is originally part of Alive and gives a reference to each instruction that has the given input as an operand.

Finally, the necessary initialization of the data structures for this algorithm is encapsulated in the function dup, called in our BlockLoopUnroll algorithm. In this function we initialize the *foriginal*, *instrs*, *phiRef* and *idupes* data structures.

```
function dup($u$, $foriginal$, $instrs$, $idupes$)
    $u'$ := BasicBlock()
    $foriginal[u']$ := $u$
    for each $instr \in instrs[u]$
        $instr'$ := Instr($instr$)
        $instrs[u']$.pushBack($instr'$)
        if $instr$ is Phi
            $phiRef[instr]$ := $instr$
            $phiRef[instr']$ := $instr$
            $idupes[instr]$.pushBack($instr'$)
    AddToContainingLoopBodies($u$, $u'$)
    return $u'$
```

Figure 3.20: Duplication of a basic block and the necessary initialization of auxiliary data structures.

Critical Edges

Now that we know how the algorithm works, we must mention an an additional detail regarding *critical edges*. Critical edges are edges $(u,\ v)$ where at least one other edge $(w,\ z)$ exists such that $w = u \wedge z = v$. Critical edges cause problems in SSA because phi instructions only allow a single entry operand for each predecessor of the containing basic block. To distinguish between two values for the same predecessor, we need to add a basic block for each critical edge with a different name, splitting it into two edges. Our algorithm assumes that these edges do not appear and does not do critical edge splitting.

3.5.5 Optimality of Loop Unroll

Unrolling loops leads to a significant increase in the number of basic blocks that Alive needs to analyze, therefore loop unrolling is in itself a trade-off between sacrificing performance for better accuracy and coverage. It makes sense to evaluate the unrolled programs with respect to carefully chosen metrics and understand how to get the most out of loop unrolling, whilst also minimizing the number of basic blocks needed to be duplicated.

We consider loop unroll optimal for *total block coverage* if we cover all basic blocks in Alive's analysis with the minimum number of blocks duplicated.

Metrics

The metrics we use must be chosen carefully since we want them to give us as much useful information as possible about the programs before and after loop unroll. These metrics will indicate whether our unroll was or not optimal with respect to the obtained block coverage for the number of vertices that needed to be duplicated.

As mentioned before, we will evaluate our unroll algorithm based on two main metrics, block coverage and path coverage. The most important of the two is block coverage, as it is more likely to find bugs in basic blocks that Alive is not able to analyze prior to unrolling than it is to find bugs in analyzing the same blocks multiple times. For this reason, we should first attempt to cover as much of the program as possible before moving over to improving path coverage. Block coverage essentially allows Alive to work with a larger part of the program while path coverage provides useful information on program control flow.

It is important to note that total block coverage may be impossible for some programs, an example of this would be programs where parts of the program are reachable only after a certain number loop iterations have been executed, where the number of iterations required is greater than the defined unroll factor.

Another metric that is worth exploring would be the total number of blocks analyzed, meaning that we count multiple visits to the same block as different visits. However, it is clear that such a metric will not be as informative as either block or path coverage since we could potentially unroll a program an arbitrary number of times expecting an arbitrary amount of information, when in fact the blocks duplicated could have no issues to begin with.

Loop Structure

In this work we only focus on unrolling program loops with a fixed unrolling factor k given as input to Alive, using the same k for all programs that Alive encounters. Through experimentation and analysis of our results we will find an optimal value for k such that the cost-benefit relation is maximized. The disadvantage of having a fixed value for k is that it assumes that all programs show a similar cost-benefit relation. In reality this is not the case as some loops may benefit from a higher unroll factor than others.

With this in mind it would be beneficial in the future to study the structure of program loops before unroll and choose an appropriate value for the unroll factor k on a loop by loop basis, by estimating the number of iterations required of each loop for certain blocks to become reachable. Moreover, the more connected the vertices in a loop are with respect to each other the more likely it is that unrolling this particular loop with a higher unroll factor would lead to a larger number of paths gained per vertex duplicated, thus significantly increasing the quality of analysis for a minimum additional cost in analysis and SMT solving time.

3.5.6 Irreducible Loops and Node Splitting

Irreducible loops are loops with multiple entries that result from unstructured control flow. This kind of loops make many algorithms that rely on traversing the program CFG more complicated. As was seen in Section 3.2, irreducible loops are the cause for having more than one possible DFST for each CFG and can significantly affect the results of algorithms such as loop identification.

Node splitting is a technique that at the cost of adding more vertices can transform irreducible loops into reducible ones. An example of a simple application of this technique can be seen in Fig. 3.21. Here we have an irreducible loop (B, C), with either vertex as a possible loop entry because of vertex A. With node splitting, a copy C2 of vertex C is created, making the loop reducible without affecting program semantics.

Node splitting can sometimes also transform irreducible loops into multiple reducible ones, where each one can be unrolled separately therefore improving the coverage gain through loop unroll compared to unrolling the irreducible loop once. Unfortunately node splitting can lead to a dramatic increase in program size when we have larger and well connected loops, as making a copy of a vertex can lead to having more vertices of the loop being considered loop entries, each one also requiring node splitting.

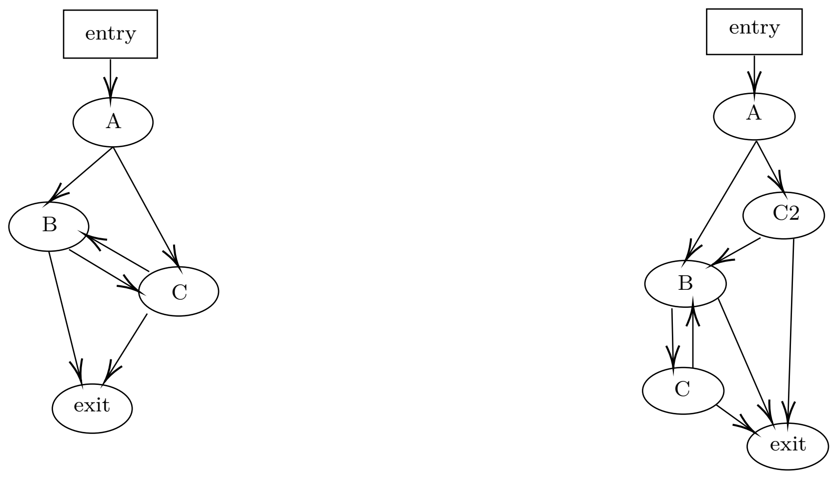


Figure 3.21: Node splitting technique applied on the graph on the left results in the graph on the right.

Some works such as [37] have been successful in mitigating program size explosion through the use of clever heuristics, however, in our case we would want to not only minimize code size increase, but also make sure that any duplicated vertices contribute to Alive's path coverage. In order to do so we would need to duplicate the minimum number of vertices of the loop such that a program exit is reachable from each and every new vertex.

Finding an optimal solution that meets these goals would bring little to no benefit since Alive rarely encounters programs with irreducible loops. It would be interesting to study optimal solutions to handling irreducible loops, however, the minimal added benefit to Alive's coverage does not justify the implementation of node splitting in our work.

3.6 Summary

In this chapter we made an overview of different concepts specific to program loops. We have presented an algorithm to unroll program loops written in Alive IR. We introduced the notions of *block coverage* and *path coverage* in Alive's analysis, which can be significantly improved through loop unrolling and with a careful choice of the loop unrolling factor. We went more in detail in how basic blocks were altered in loop unrolling and how program instructions were updated.

Lastly, some concerns regarding the unrolling of irreducible loops were mentioned, for which we think that the *node splitting* technique is useful. However, due to the rarity of irreducible loops, we do not expect that adding special treatment for irreducible loops in our loop unrolling algorithm will give much benefit to the effectiveness of Alive's analysis when validating a very large number of programs.

Chapter 4

SMT Formula Reduction

One of the main contributions of this work is the reduction of SMT formula sizes produced in Alive to mitigate the burden put on the SMT solver. Before we can dive into our approach, we must first understand how Alive's current program analysis works in a little more detail.

4.1 Current SMT Formulas in Alive

A program in LLVM IR is divided into basic blocks that consist of sequences of instructions in between control-flow jumps. Before symbolic execution, Alive runs a topological sort algorithm to sort the basic blocks topologically, such that predecessors appear before successors. Since symbolic execution simulates program execution, the predecessor before successor relation must be respected.

Definition 16. A list of program vertices is in topological order if each and every vertex is positioned after all its predecessors and before all its successors.

During symbolic execution, each instruction of each basic block is visited in sequence. Alive encodes each instruction into three SMT expressions, one for its value, one for poison and another for UB. The poison expression tells us if the value does not result from non-deterministic program behavior as defined in the LLVM IR specification, while UB represents the cases in which immediate undefined behavior occurs. An SMT expression is a set of logical constraints that can be evaluated to a true or false statement. These expressions encode the semantics of a program in a format that SMT solvers can reason with. For example, consider an `udiv` instruction in LLVM IR that takes in two operands: `%r = udiv exact %a, %b`. The encoded value expression will correspond to the unsigned division of `%a / %b`. The expression for poison will be true if for example `%a` is poison or if `%a` is not a multiple of `%b` since the *exact* keyword is present. The third expression indicates that the execution of this instruction results in immediate UB if `%b` is zero.

Expressions can reference other expressions. Here the expression for the `udiv` instruction references `a` and `b`, where either one of these also can reference other expressions.

During symbolic execution, Alive gathers expressions obtained from encoding instructions into a larger expression. If a program does not contain any branching instructions, the final expression will be a single chain of expressions connected with the logical and ($\wedge$) operator.

$$\text{expr}_1 \land \text{exp}_2 \land \text{expr}_3 \land ... \tag{4.1}$$

If the program has branching instructions, the formulas will include the logical or operator ($\lor$). For the program in Fig 2.4, vertex *exit* has two predecessors, therefore control-flow could have come from B or from C. The formula for this program would have the following structure:

$$((\text{entry} \land \text{A} \land \text{B}) \lor (\text{entry} \land \text{A} \land \text{C})) \land \text{exit} \tag{4.2}$$

Throughout this thesis, we will refer to all expressions that Alive creates for a given basic block by its label for readability. In practice, the basic block labels are not used in SMT expressions.

4.2 Motivation for ite Expressions

To reduce the redundancy of expressions created in Alive, *ite* expressions are ideal. Currently, Alive does not use *ite* expressions from the SMT language to represent the semantics of program undefined behavior. An *ite* is an if-then-else in the SMT language, which can be replaced by either the *then* or the *else* clause depending on the conditional expression given to it. The result of an *ite* expression is an unconstrained variable in SMT, a variable that is independent of the rest of the formula and allows for more efficient SMT solving.

The biggest advantage of *ite* expressions is that expressions such as UB formulas and path constraints can be divided and referenced by multiple *ite* expressions. This allows them to be shared among more program paths while also reducing the number of times each one is referenced. Furthermore, the structure of formulas constructed with *ite* expressions is much easier for us to read.

Lastly, *ite* expressions do come with the downside of making it more difficult to share the suffixes of program expressions that appear in its *then* and *else* clauses. This is due to these clauses being mutually exclusive, and thus common expressions between them cannot be easily shared. We must find better ways to also share these suffixes without sacrificing the benefits of easily sharing formula prefixes that *ite* expressions provide.

4.3 SMT Expression Sharing and Redundancy

Currently, when Alive encounters a basic block *b* with more than one predecessor, a disjunction expression *dis* of the predecessor formulas is created. After Alive fully analyzes a basic block *b*, the expression *expr* encoded from its instructions is used to create another formula *dis* $\lor$ *expr*, where *expr* is designated as the suffix. If the paths that reach *b* are not disjoint then the expressions obtained from encoding instructions in common basic blocks between predecessors of *b* will be referenced more than once. This redundancy is undesired and makes the formulas Alive creates for *b* larger.

Throughout this chapter we present four different versions of an algorithm for constructing smaller UB formulas in Alive in sequence, where each subsequent algorithm is an improvement of the previous.

The first algorithm *UBReduction* presented in Section 4.5 reconstructs UB formulas in Alive, and uses mainly *ite* expressions, allowing the prefix of these formulas to be easily shared. The issue with this implementation is that we are not considering suffix redundancy. To illustrate, consider the graph shown in Fig. 4.1.

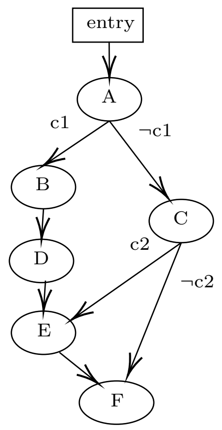


Figure 4.1: Simple program with two if statements, one in A and one in C.

The UBReduction algorithm produces the following formula:

$$\mathrm{entry} \land \mathrm{A} \land \mathrm{ite}(\mathrm{c}_1, \mathrm{B} \land \mathrm{D} \land \mathrm{E} \land \mathrm{F}, \mathrm{C} \land \mathrm{ite}(\mathrm{c}_2, \mathrm{E} \land \mathrm{F}, \mathrm{F})) \tag{4.3}$$

Alive on the other hand produces the following formula:

$$(((\mathrm{entry} \land \mathrm{A} \land \mathrm{B} \land \mathrm{D}) \lor (\mathrm{entry} \land \mathrm{A} \land \mathrm{C})) \land \mathrm{E}) \lor (\mathrm{entry} \land \mathrm{A} \land \mathrm{C})) \land \mathrm{F} \tag{4.4}$$

The prefix *entry* $\land$ A is only shared in Expression 4.3. The suffix F is shared in Expression 4.4, but referenced three times in Expression 4.3. Depending on the program given to Alive, either solution could produce a smaller formula than the other. In any case, both are susceptible to building formulas that can be extremely large and far from optimal. We deal with this problem by improving our algorithm to share both formula prefixes and suffixes by making use of the concept of *post-dominators*. This improvement is presented in Section 4.9 and in Section 4.10.

Sharing both prefixes and suffixes, however, is challenging. By creating *ite* expressions to share the prefix of a formula we may introduce more redundancy in the suffix of the formula, and vice versa. This is where introducing a heuristic becomes useful. In order to mitigate formula blowup as much as possible, analyzing the program and its control-flow graph before construction will allow us to decide on a block by block basis if adding an *ite* expression is a good idea. The better the heuristic, the more we can minimize formula sizes. We discuss this heuristic-based solution in Section 4.11.

4.4 Challenges with Building ite Expressions

As we have seen, *ite* expressions are useful in reducing some of the redundancy in SMT formulas. Sadly, *ite* expressions are more difficult to build. In order for Alive to use *ite* expressions instead, we must delay formula construction for the whole program until after Alive finishes symbolic execution on each basic block of the program. This is because when we construct an *ite* expression for some basic block b, we must already have the expressions for each successor of b.

To illustrate, consider the program shown in Figure 2.4. Instead of iterating the predecessors of `exit` when Alive's symbolic execution reaches it as is currently done in Alive, we instead build the *ite* expression for A. Recall that symbolic execution follows the control-flow of normal program execution, therefore when we reach A, we have not yet analyzed neither B nor C. For each *ite* we build for a given vertex u in a program, we need to have already analyzed all descendants of u, and as such we must delay the construction of *ite* expressions to only after Alive has finished symbolic execution on the program.

4.5 Basic UB Reduction Algorithm

Now that we understand the challenges involved in building correct expressions with the *ite* operator, let's move on to our algorithm. Since the construction of an *ite* requires that the expressions for its successors be already built, we construct the expressions in reverse topological order. The algorithm is shown in Figure 4.2.

```
1  function UBReduction($CFG$, $exprs$)
2  input
3      $E : V \times \langle c, V\rangle$ - set of edges
4      $CFG : (V, E)$ - graph with vertices $V$ and edges $E$
5      $v_0 : V$ - initial vertex
6      $exprs : V \nrightarrow$ SMT
7  vars
8      $visited : V \to$ bool
9      $stack : \mathcal{P}(V)$ - vertices with pending visit
10 begin
11     $stack := v_0$
12     $visited : \lambda\ v\ .\ \to$ false
13     while $stack \neq \emptyset$ do
14         $u := stack$.pop()
15         if not $visited(u)$ then
16             $visited(u) :=$ true
17             $stack$.push($u$)
18             for each $\langle c, v\rangle \in E(u)$ do
19                 $stack$.push($v$)
20         else
21             $e :=$ true
22             for each $\langle c, v\rangle \in E(u)$ do
23                 $e :=$ ite(c, $exprs$[v], $e$)
24             $exprs[u] := and(exprs[u], e)$
25     return $exprs[v_0]$
```

Figure 4.2: Base algorithm for constructing smaller SMT formulas in Alive.

To traverse the program in reverse topological order, we first add the vertices we see to a stack in visit order (ln. 14-19), adding the currently visited vertex first. Doing so will ensure that the second time we visit each vertex will be in reverse topological order.

Lines 21-24 show how the new expressions are created. We create a new variable e as an expression with a value of `true`, neutral to logical `and` operations. Then for each successor, we either assign e to have the expression of the first target, or wrap the current value of e in the `else` clause of an *ite*. Doing it this way ensures *ite* are created only when needed and are easily nested. The resulting number of *ite* for a given block u will then be equal to $|E(\mathrm{u})|$ - 1.

The algorithm does have an initial requirement where the map *exprs* is initialized during Alive's symbolic execution, before we run this algorithm. Initially, each entry of *exprs* corresponding to a vertex $v \in V$ will contain only those expressions that were obtained during symbolic execution of v. However during the execution of our algorithm, we will replace entries in *exprs* such that each entry will not only contain a reference to the expressions generated during symbolic execution of vertex v, but also those of all descendants of v in paths that reach a program exit. Since this process happens in reverse topological order, the new shorter expression produced by our algorithm will be kept in *exprs*$[n_0]$ once the algorithm has finished.

The intuition here is that we create *ite* expressions that wrap around other expressions until we reach the program entry v_0. For example, if we look back at the program in Fig. 4.1, when our algorithm reaches A, both entries for A and *entry* in *exprs* will only contain the expressions generated by Alive's symbolic execution of these vertices, while all other vertices will contain references to the expressions of their descendants. For example, *exprs*[C] will be:

$$\mathrm{C} \wedge \mathrm{ite}(\mathrm{c_2}, \mathrm{E} \wedge \mathrm{F}, \mathrm{F}) \tag{4.5}$$

and *exprs*[B] will be:

$$\mathrm{B} \wedge \mathrm{D} \wedge \mathrm{E} \wedge \mathrm{F} \tag{4.6}$$

Using *exprs* this way helps us to efficiently create *ite* expressions a program vertex v by only looking at the immediate descendants of v. In this case for v := A, we have:

$$\mathrm{A} \wedge \mathrm{ite}(\mathrm{c_1}, \mathrm{exprs[B]}, \mathrm{exprs[C]}) \tag{4.7}$$

which becomes:

$$\mathrm{A} \wedge \mathrm{ite}(\mathrm{c_1}, \mathrm{B} \wedge \mathrm{D} \wedge \mathrm{E} \wedge \mathrm{F}, \mathrm{C} \wedge \mathrm{ite}(\mathrm{c_2}, \mathrm{E} \wedge \mathrm{F}, \mathrm{F})) \tag{4.8}$$

In the next few sections we will present some changes to this algorithm. Although it produced good results, these can still be improved. For example, in the expression above, the expressions created by Alive for vertex F appear three times, even though every single program path that reaches program exit will always include F. We eliminate most of this redundancy by extracting repeated references to sub-expressions from the *ite* that do not depend on the *ite* condition. In this case F is included regardless whether either $\mathrm{c_2}$ and $\mathrm{c_1}$ is satisfied, and we can instead end up with the following formula for A:

$$\mathrm{A} \wedge \mathrm{ite}(\mathrm{c_1}, \mathrm{B} \wedge \mathrm{D} \wedge \mathrm{E}, \mathrm{C} \wedge \mathrm{ite}(\mathrm{c_2}, \mathrm{E}, \mathrm{true})) \wedge \mathrm{F} \tag{4.9}$$

4.6 Rewriting Expressions in Z3

After having presented the algorithm that creates smaller SMT formulas with *ite* expressions in Section 4.5, we could ask the question: Why reconstruct the expressions for the whole program and not just modify what needs to be changed? There are two main reasons for doing so:

(*i*) **The structure of the overall formula is modified**. If we use *ite* expressions for a basic block's successors instead of simple disjunctions of its predecessors, we obtain a completely different formula structure, which we cannot create through modification of the formulas that Alive currently builds.

(*ii*) **Modifying expressions in Z3 is expensive**. During symbolic execution, Alive creates expressions for the Z3 SMT solver, where each one may reference other expressions when its result depends on them. Expressions are also represented as an *ast* (*abstract syntax tree*) in Z3, which can be thought of as a tree of references, where its hash code is the unique identifier for the expression. This kind of hashing is called *structural hashing*, where the structure of the *ast* also influences the hash code. Modifying an expression e would invalidate all other expressions that have a reference to e, forcing Z3 to recalculate hash codes, therefore we cannot modify expressions as this is too expensive. Instead, in our algorithm we opt for a reverse topological visit order which ensures that the sub-expressions are built just before they are needed.

4.7 Formula Size Explosion

Alive's symbolic execution does not traverse the same graph as the program's control flow graph. Due to limitations of symbolic execution back edges must be ignored. Essentially, Alive is traversing a *directed acyclic graph* (DAG) version of the program. Unfortunately, creating a tree of *ite* expressions from a DAG can result in an exponential growth in the number of *ite* expressions. Consider the CFG in Fig. 4.3 that shows a worst-case example.

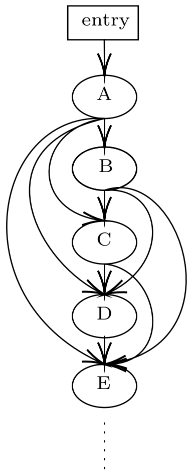


Figure 4.3: A generalization of a DAG with maximal number of directed edges. Each vertex has a directed edge to all its descendants.

Consider also the generalized topological ordering of vertices corresponding to the program graph in this figure:

$$[\text{A, B, C, D, E, ...}] \tag{4.10}$$

We can write the topological ordering as a sequence of vertices such that each vertex has an index ranging from 0 to |N| - 1, where n_i precedes n_{i+1}. The number of directed edges for a vertex n_i would be `N-i-1` and there would be no edge from a vertex n_j to a vertex n_k such that $k < j$.

If we keep adding vertices where each vertex n_i has a directed edge to every other vertex n_j, where `i < j`, meaning the DAG has a maximal number of directed edges, then the number of directed edges will grow exponentially as the number of vertices grows linearly. As a consequence, the number of *ite* expressions would also grow exponentially since for each vertex with `s` successors we would have `s-1` *ite* expressions. Fortunately, in practice we rarely encounter any DAG with a maximal or close to maximal number of edges that is sufficiently large to become problematic, as such graphs with unstructured control flow require the use of rarely used instructions like `goto` in C.

4.8 Dominators and Post-dominators

The *dominator* and *post-dominator* vertex relationships are extremely useful as they can be computed efficiently and provide relevant information about program control flow. We will be using these concepts heavily to further improve our solution for creating smaller SMT formulas.

Definition 17. In a DAG, a vertex *u* dominates a vertex *v* if any path reaching *v* must visit *u*.

Definition 18. In a DAG, a vertex *u* post-dominates a vertex *v* if any path starting from *v* reaching some program exit must visit *u*.

To have a better understanding regarding these definitions and the difference between *dominators* and *post-dominators*, consider the following statements about the *dominance* and *post-dominance* relationship between vertices of the program in Fig. 4.1.

4.8.1 Dominance

- A dominates C because any path reaching C must pass through A.
- C does not dominate E because there is another path *entry* $\rightarrow$ A $\rightarrow$ B $\rightarrow$ D $\rightarrow$ E reaching E that does not pass through C.
- F is dominated by A because there does not exist a path reaching F that does not pass through A.

4.8.2 Post-dominance

- A post-dominates *entry* because there are no paths from *entry* to the program exit F that do not pass through A.

- D does not not post-dominate A because there exists a path *entry* → A → C → F that reaches the program exit F without passing through F.
- F post-dominates A because any path from A reaching the program exit F passes through F.

The changes we make to the basic algorithm presented in Section 4.5 will make heavy use of these two concepts. Specifically, we will be building a *dominator tree* with an algorithm presented in [38], while the *post-dominance* relationship will be computed when needed during formula construction.

4.8.3 Dominator Trees

For us to be able to improve our SMT formula reduction algorithm, building a dominator tree provides useful information that can be looked up efficiently.

Definition 19. A dominator tree is a tree of vertices where the descendants of any vertex u are dominated by u.

Definition 20. The immediate dominator of a vertex u is the parent of the vertex u in the dominator tree.

The algorithm for building the dominator tree is presented in [38], and will be useful for giving dominator information needed by our SMT formula reduction algorithms. An example of a program CFG with its respective dominator tree is shown in Fig. 4.4.

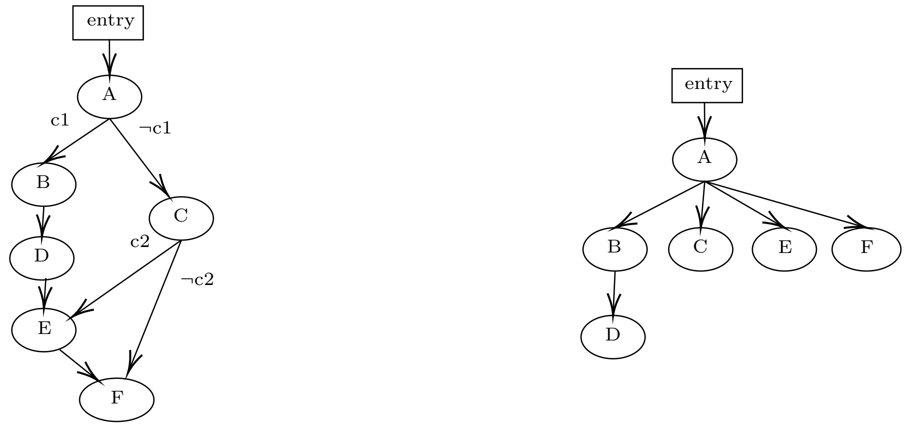


Figure 4.4: An example program CFG to the left (reproduced from Fig 4.1) with the respective dominator tree on the right, where A dominates B, C, E and F, while D is dominated by B.

4.8.4 Dead Vertex Removal

In addition to implementing the algorithm, we also had to make a few changes to Alive to cover certain edge cases. For example consider the program represented in Fig. 4.5.

Vertex D is not reachable from entry and has no predecessors. Vertex C has two predecessors which are A and D. With this in mind, neither A nor D dominate C, therefore C has no dominator. This happens because there is no common dominator between the predecessors of C, and results from a control flow graph containing unreachable vertices. In such graphs the definition for *dominance* is not valid.

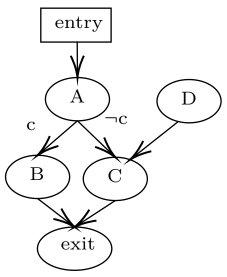


Figure 4.5: A program with an *if-then-else* statement. Vertex D has no predecessors and is designated as a dead basic block.

Unreachable vertices do not contribute to the program as they are never covered by Alive, therefore we can safely remove them before running any algorithm that relies on *dominators* and *post-dominators*.

4.9 UBReduction with Post-dominators

The algorithm UBReduction presented in Section 4.5 is able to create smaller SMT formulas, however, we can do better.

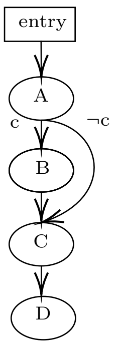


Figure 4.6: A CFG that describes a program with an if statement without an else statement. We have a fork in vertex A, and a merge in C.

If we run our basic algorithm on the program shown in Fig. 4.6 we will obtain the following formula:

$$\mathrm{A} \wedge \mathrm{ite}(\mathrm{c}, \mathrm{B} \wedge \mathrm{C} \wedge \mathrm{D}, \mathrm{C} \wedge \mathrm{D}) \tag{4.11}$$

In this formula C ∧ D is repeated because it is referenced in both paths created at the fork A that join at merge C. This expression is referenced twice in this *ite* even though it is independent of the jump condition *c*. A better expression would be the following:

$$\mathrm{A} \wedge \mathrm{ite}(\mathrm{c}, \mathrm{B}, \mathrm{true}) \wedge \mathrm{C} \wedge \mathrm{D} \tag{4.12}$$

By identifying that C ∧ D is common between all paths reaching C, we can extract it from the ite and instead have it be referenced only once. Only expressions built for a merge vertex are shared among the incoming paths. Recall that an expression built for a vertex consists of the expressions encoded from that vertex and all its descendants.

To remove this redundancy, we only need to make a minor adjustment to our initial algorithm UBReduction. Whenever we visit a merge *m*, we look up its immediate dominator *idom* and check whether *m postdom idom*, meaning that *m* post-dominates *idom*. If this condition is satisfied, we can move the expressions built for *m* into *idom*:

$$\begin{gathered}\text{exprs[idom]} := \text{exprs[idom]} \land \text{exprs[m]} \\ \text{exprs[m]} := \text{true}\end{gathered} \tag{4.13}$$

If *m postdom idom* then any path starting from *idom* will reach *m* regardless of any control flow altering instructions in paths between *m* and *idom*. For this reason, the expressions built for *m* will always be included for any path starting from *idom*. By moving the expressions of *m* to *idom*, we add a single reference for all paths at once. Afterwards, we replace the expressions stored for *m* with *true*, therefore preventing the moved expressions from being referenced again outside *idom*. We will designate these two steps as the *move* operation. The updated algorithm with these changes is presented in Fig. 4.7. In line 26 we look up the immediate dominator of the merge vertex, in line 27 we check whether the merge vertex post dominates its immediate dominator, and in lines 28-29 we apply the *move* operation.

```
1  function UBReductionPostdom($CFG$, $DT$, $exprs$)
2  input
3     $E : V \times \langle c, V\rangle$ - set of edges
4     $CFG : (V, E)$ - graph with vertices $V$ and edges $E$
5     $DT : (V, ET)$ - dominator tree
6     $v_0 : V$ - initial vertex
7     $exprs : V \nrightarrow$ SMT
8  vars
9     $visited : V \to$ bool
10    $stack : \mathcal{P}(V)$ - vertices with pending visit
11 begin
12    $stack := \{v_0\}$
13    $visited : \lambda\ v\ .\ \to$ false
14    while $stack \neq \emptyset$ do
15       $u := stack$.pop()
16       if not $visited(u)$ then
17          $visited(u) :=$ true
18          $stack$.push($u$)
19          for each $\langle c, v\rangle \in E(u)$ do
20             $stack$.push($v$)
21       else
22          $e :=$ true
23          for each $\langle c, v\rangle \in E(u)$ do
24             $e :=$ ite(c, $exprs$[v], $e$)
25          $exprs[u] := and(exprs[u], e)$
26          $idom := DT$.getIdom($u$)
27          if $postdom(u, idom)$ then
28             $exprs[idom] := and(exprs[idom], exprs[u])$
29             $exprs[u] :=$ true
30    return $exprs[v_0]$
```

Figure 4.7: Improved version of the UBReduction algorithm by making use of the post-dominance relation between vertices.

4.10 UBReduction with Conditional Post-dominators

The improvement to the algorithm discussed in the previous section promises to eliminate significant amounts of redundancy. It is unfortunately not very effective as it assumes that *m postdom idom* is commonly satisfied. In practice this condition may never satisfy for larger programs, as it takes just a single path from *idom* to not reach *m* for the formula to keep its redundant expressions. Instead, we construct a guard expression that represents the paths from *idom* that do not reach *m*. We represent a path by an SMT expression with only the path constraints.

For the program in Fig. 4.1, we see that vertex E does not post-dominate A because of the path *entry* → A → C → F, therefore the guard condition for this merge vertex becomes $\neg c_1 \wedge \neg c_2$. Similarly, an expression representing all the paths that do reach E starting from A would be the negation of this expression: $\neg(\neg c_1 \wedge \neg c_2)$.

```
function UBReductionCondPostdom($CFG$, $DT$, $exprs$)
input
    $E : V \times \langle c, V \rangle$ - set of edges
    $CFG : (V, E)$ - graph with vertices $V$ and edges $E$
    $DT : (V, ET)$ - dominator tree
    $v_0 : V$ - initial vertex
    $exprs : V \nrightarrow \text{SMT}$
vars
    $visited : V \to \text{bool}$
    $stack : \mathcal{P}(V)$ - vertices pending visit
begin
    $stack := \{v_0\}$
    $visited : \lambda\ v\ .\ \to \text{false}$
    while $stack \neq \emptyset$ do
        $u := stack.\text{pop}()$
        if not $visited(u)$ then
            $visited(u) := \text{true}$
            $stack.\text{push}(u)$
            for each $\langle c, v \rangle \in E(u)$ do
                $stack.\text{push}(v)$
        else
            $e := \text{true}$
            for each $\langle c, v \rangle \in E(u)$ do
                $e := \text{ite}(c, exprs[v], e)$
            $exprs[u] := and(exprs[u], e)$
            $idom := DT.\text{getIdom}(u)$
            $guard := postdom(u, idom)$
            $exprs[idom] := and(exprs[idom], \neg\ guard \implies exprs[u])$
            $exprs[u] := \text{true}$
    return $exprs[v_0]$
```

Figure 4.8: Improved version of the UBReduction algorithm by making use of a conditional post-dominance relation between vertices.

This allows us to perform this *move* only for a subset of the paths starting from A by moving the expression $(\neg(\neg c_1 \wedge \neg c_2)) \Rightarrow \text{exprs}[E]$ instead of just exprs[E]:

$$A \wedge \text{ite}(c_1, B \wedge D, C) \wedge ((\neg(\neg c_1 \wedge \neg c_2)) \Rightarrow E\ ) \wedge F \tag{4.14}$$

The updated algorithm with this guarded *move* is shown in Fig. 4.8. For a vertex *u* and its immediate dominator *idom*, the function *postdom* was changed to now return an expression, called the *guard*, which represents the paths from *idom* that do not reach *u*. If *u* post-dominates *idom*, then we get the same result as the one from UBReductionPostdom.

4.11 UBReduction with Heuristics

UBReductionPostdom and UBReductionCondPostdom aim to reduce redundancy in the suffix of the *ite* expressions by taking advantage of the concepts of dominators and post-dominators. Unfortunately, the former rarely triggered the *move* operation while UBReductionCondPostdom triggered it too much, often trading an amount of redundancy with a larger amount of path constraints. For the best solution, we must find a balance in the trade-off between choosing to keep the redundancy or to eliminate it and add a guard to the moved expressions. In other words, to decide whether it is worth sharing the suffix of created formulas at the cost of an additional expression based on path constraints.

Consider the program with two program exits N2 and *m* shown in Fig. 4.9. Because of the path A → N1 → N2, *m* does not post-dominate A. However, it is clear that building a guard expression for this program would significantly reduce the formula size, since the number of paths starting from A that do not reach *m* is very small compared to the amount of redundancy we can eliminate.

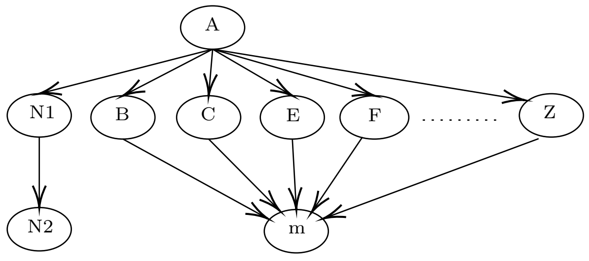


Figure 4.9: *m* does not post-dominate A because of a single path, resulting in all successors of A except N1 to have a reference to the expressions in *m*.

In order to make this decision on any program, we define a heuristic based on two metrics. The first metric aims to measure the size of the current vertex redundancy by counting the number of UB expressions created from encoding instructions in each basic block. The second metric measures the size of the added path constraints by counting the number of conditional jumps between basic blocks in each path between a merge *m* and its immediate dominator *idom*. If *pathSize* > *ubSize*, we decide to not apply the *move*. The pseudocode for the final version of our algorithm is presented in Fig. 4.10. The data structure *exprcount* contains expressions created for encoding the undefined behavior semantics of each basic block's instructions and those of all its descendants. The function *postdom* in line 31 returns a guard and an additional estimation of the number of expressions in the guard expression. In line 32 we decide whether or not to perform the *move* operation.

```
function UBReductionHeuristic($CFG$, $DT$, $exprs$, $exprcount$)
input
    $E$ : $V \times \langle c, V\rangle$ - set of edges
    $CFG$ : $(V, E)$ - graph with vertices $V$ and edges $E$
    $DT$ : $(V, ET)$ - dominator tree
    $v_0$ : $V$ - initial vertex
    $exprs$ : $V \nrightarrow$ SMT
    $exprcount$ : $V \times \mathbb{N}_0$ - estimated number of exprs added for each vertex
vars
    $visited$ : $V \rightarrow$ bool
    $stack$ : $\mathcal{P}(V)$ - vertices pending visit
    $ubcount$ : $V \times \mathbb{N}_0$
begin
    $stack := \{v_0\}$
    $visited : \lambda\ v\ .\ \rightarrow$ false
    $ubcount : \lambda\ n\ .\ \rightarrow 0$
    while $stack \neq \emptyset$ do
        $u := stack$.pop()
        if not $visited(u)$ then
            $visited(u) :=$ true
            $stack$.push($u$)
            for each $\langle c, v\rangle \in E(u)$ do
                $stack$.push($v$)
        else
            $e :=$ true
            for each $\langle c, v\rangle \in E(u)$ do
                $e :=$ ite(c, $exprs$[v], $e$)
                $ubcount[u] := ubcount[u] + exprcount[v]$
            $exprs[u] := and(exprs[u], e)$
            $idom := DT$.getIdom($u$)
            $\langle guard, pathSize\rangle := postdom(u, idom)$
            if not $pathSize > ubcount[u]$ then
                $exprs[idom] := and(exprs[idom], \neg guard \implies exprs[u])$
                $exprs[u] :=$ true
    return $exprs[v_0]$
```

Figure 4.10: Heuristic based version of the UBReductionCondPostdom algorithm.

4.12 Reduction of Phi SMT Formulas

A phi instruction in LLVM IR must always appear at the start of a basic block, before any other non-phi instructions. It selects a value depending on through which predecessor control flow passed through. As an example, consider the CFG in Fig 4.11.

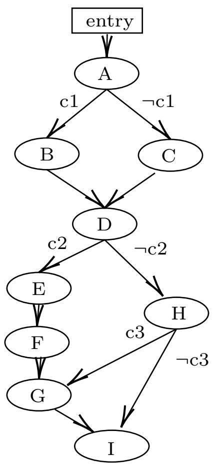


Figure 4.11: A control flow graph similar to the one in Fig. 4.1, but with an additional conditional jump in A and a merge in D.

Assume also that vertex G has the following phi instruction:

$$\%\text{result} := \text{phi i1 } [\ 0,\ \%\text{F}\ ],\ [\ 1,\ \%\text{H}\ ] \tag{4.15}$$

The value assigned to `result` is 0 if control-flow entered G through F, or 1 if through H. A phi instruction must have as many entries as there are predecessors, and one basic block may have more than one phi instruction.

In the current implementation of Alive, the SMT expressions created for a phi instruction in some vertex u consist of a logical disjunction of the path constraints of predecessors of u. Since the path expressions of each predecessor include expressions since program entry, then the encoded expressions for phi instructions will contain a lot of unnecessary information.

To better illustrate the problem, consider the expression obtained from encoding the above phi instruction:

$$\text{ite}((\text{c}_1 \vee \neg\text{c}_1) \wedge \neg\text{c}_2 \wedge \text{c}_3, 1, 0) \tag{4.16}$$

Note that this expression represents the worst-case as there exists a simpler alternative. A simpler expression would consist in having c_2 in the condition of the *ite* instead of $\neg\text{c}_2 \wedge \text{c}_3$. Obtaining one or the other depends on many factors and is out of our control.

Here the expression $(\text{c}_1 \vee \neg\text{c}_1)$ is irrelevant to the decision in distinguishing between the entries in the phi instruction, since both paths that branch out from A can reach either one of the predecessors of the phi. Ideally, for a single phi instruction we want the smallest path expression containing a minimum amount of information on control flow to decisively select between values of its entries. In this case, just c_2 would be sufficient for distinguishing control-flow in G between predecessor vertices F and H. The expression that our algorithm builds would be the following:

$$\text{ite}(\neg\text{c}_2, 1, 0) \tag{4.17}$$

Our approach to this problem is a simple adaptation of the algorithm UBReduction and is shown in Fig. 4.12.

```
 1 function PhiReduction($CFG$, $merge$, $idom$, $phientries$, $jumpconds$)
 2 input
 3     $CFG : (V, E)$ - control flow graph as pair of vertices and edges
 4     $merge : V$ - vertex containing phi
 5     $idom : V$ - immediate dominator of $merge$
 6     $phientries : V \times$ SMT - values for each phi entry
 7     $jumpconds : V \times V \times V \to$ SMT - jump conditions of each edge
 8 vars
 9     $exprs : V \nrightarrow$ SMT
10     $visited : V \to$ bool
11     $stack : \mathcal{P}(V)$ - vertices pending visit
12     $dsts : V \times V \to \mathcal{P}(V)$ - whitelist of vertices
13 begin
14     $visited : \lambda\ v\ . \to$ false
15     for each $u \in E^{-1}[merge]$ do
16         $stack$.push($u$)
17         $exprs[u] := phientries[u]$
18     while $stack \neq \emptyset$ do
19         $u := stack$.pop()
20         if not $visited[u] \lor u \neq idom$ then
21             $visited[u] :=$ true
22             for each $v \in E^{-1}[u]$ do
23                 $dsts[v] := dsts[v] \cup \{u\}$
24     for each $u \in V$ do
25         $visited[u] :=$ false
26     $stack := \{idom\}$
27     while $stack \neq \emptyset$ do
28         $u := stack$.pop()
29         if not $visited[u] \lor u \neq idom$ then
30             $visited[u] :=$ true
31             for each $v \in E[u]$ do
32                 $stack$.push($v$)
33         else
34             for each $v \in dsts[u]$ do
35                 $exprs[u] := ite(jumpconds[u][v], exprs[v], exprs[u])$
36     return $exprs[idom]$
```

Figure 4.12: SMT formula reduction algorithm for phi instructions.

Lines 15-23 show a bottom-up traversal starting from all predecessors of *merge* until *idom*, the immediate dominator of *merge*. The goal here is to filter out any vertices that do not belong to a path reaching *merge* starting from *idom*. Since *idom* dominates *merge*, no ancestor of *idom* will be visited in this traversal.

Lines 27-35 show the process of creating *ite* expressions bottom-up, similar to what is done in the UBReduction algorithm presented in Section 4.5. Intuitively, we are creating *ite* expressions that reference other expressions, which we can visualize as a tree gaining a new root on each vertex visit. The final expression that distinguishes between the phi entries with little to no redundancy will therefore be the *ite* constructed for *idom*.

4.13 Summary

In this chapter we gave an overview of how Alive currently generates the encoding of the semantics of program undefined behavior and how it can be improved. We took advantage of graph theory concepts such as *dominators* and *post-dominators* in our developed algorithms. These algorithms can build less redundant SMT formulas that represent the semantics of program undefined behavior and the semantics of phi instructions. The construction of smaller SMT formulas are a means towards better performance by reducing the time spent in the SMT solver, which is a dominant factor in total Alive's analysis time.

Chapter 5

Results

In this chapter, we evaluate our loop unrolling algorithm in terms of block coverage, path coverage and total execution time. In addition to this, we also present two bugs found in LLVM. Secondly, we evaluate whether our SMT formula reduction algorithms reduced formula sizes and SMT solving time.

All results shown were obtained in a Linux environment through WSL 2 (Windows Subsystem for Linux) on Windows 10 with an Intel i5 6600 CPU and 32 GB of RAM. Each test run has a default timeout of 10 seconds per Z3 query which is respected most of the time, apart from some situations where Z3 is not able to exit early due to program safety reasons.

5.1 Loop Unrolling

To evaluate the benefits of our loop unrolling algorithm in Alive, we ran four different benchmarks, namely `ph7`[1], `gzip`[2], `oggenc`[3] and `bzip2`[4], each with different loop unroll factors k. To deal with functions that either take too long to analyze or never finish, we added a timeout of 15 minutes for each function. We also ran loop unrolling on the LoopVectorization and LoopUnroll LLVM test sets with unrolling factors of 0, 16 and 32, which serve as good benchmarks to evaluate our algorithm. We measured the average block coverage of each benchmark for functions that had loops, we measured the impact that loop unrolling had on total execution time and we show measurements of other metrics such as path coverage, timeouts and number of functions with unsatisfied preconditions.

For obtaining these results, we used the versions of LLVM and Z3 corresponding to the Github commit hashcodes `e2dd86bbfcb` and `44679d8f5b` respectively. The loop unrolling algorithm we implemented extends Alive with version `798f2f7150`.

The average block coverage for each benchmark is shown in Fig. 5.1. The current version of Alive does not unroll loops, as if `k = 0`. Observing these results, we can confirm that loop unroll did drastically improve Alive's block coverage, with for instance `gzip` jumping from 54 % with no loop unroll to 89.6 % with `k = 1`. Moreover, the largest gains in block coverage occur with a loop unrolling factor of `k = 1`, as

[1]https://github.com/symisc/PH7
[2]http://people.csail.mit.edu/smcc/projects/single-file-programs/gzip.c
[3]http://people.csail.mit.edu/smcc/projects/single-file-programs/oggenc.c
[4]http://people.csail.mit.edu/smcc/projects/single-file-programs/bzip2.c

is expected since for most programs with regular control flow, where the body of each loop is ignored in Alive's coverage without loop unroll, mainly due to limitations of symbolic execution. The increase in block coverage for loop unroll with loop unroll factors `k >= 2` are not as significant and lead to a dramatic increase in program size. Recall that with `k = 1`, our loop unrolling algorithm only duplicates the header of each loop, while for `k >= 2` we duplicate the body of each loop `k - 1` times and the header of each loop `k` times. Some programs such as `oggenc` show a reduced block coverage with greater unrolling factors. This is due to an increased number of timeouts in functions that reported good coverage results with lower unrolling factors. In Fig. 5.2 we have the block coverage results for the LoopVectorize and LoopUnroll LLVM test sets.

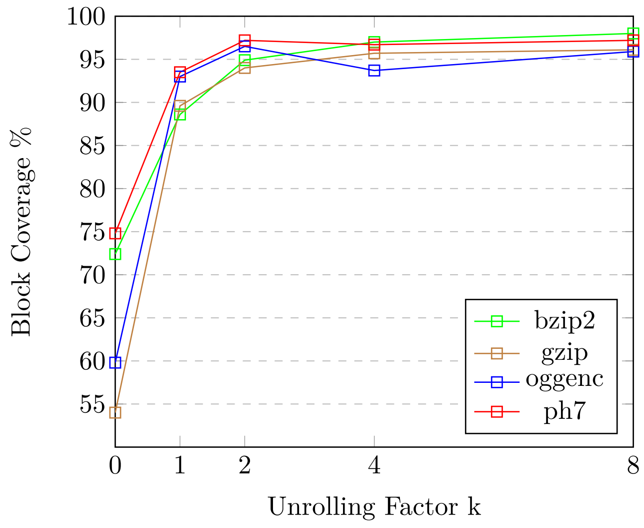


Figure 5.1: Average block coverage measured after loop unroll on four different benchmarks.

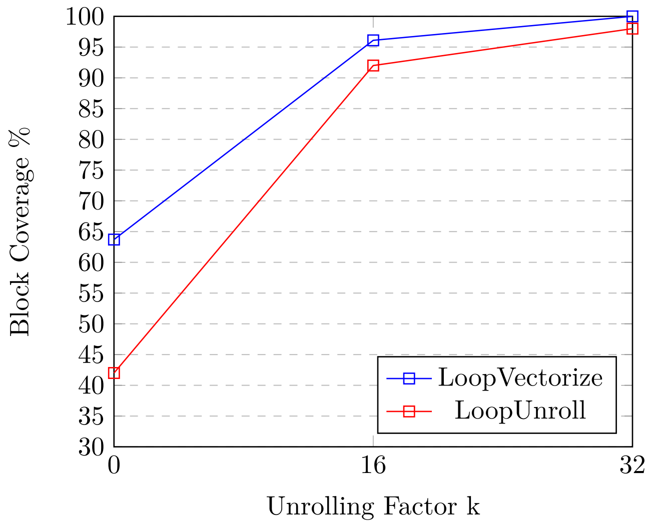


Figure 5.2: Average block coverage measured for the LoopVectorize and LoopUnroll test sets.

In Fig. 5.3 we can see the total execution time of the unrolled benchmarks. Since the number of basic blocks duplicated is significantly greater with `k >= 2`, we see a larger increase in execution time. Programs `gzip` and `bzip` are much smaller than `ph7` and `oggenc` and thus take much less time to run, for which loop unrolling had a lesser impact on performance. For reference, on our test setup the programs

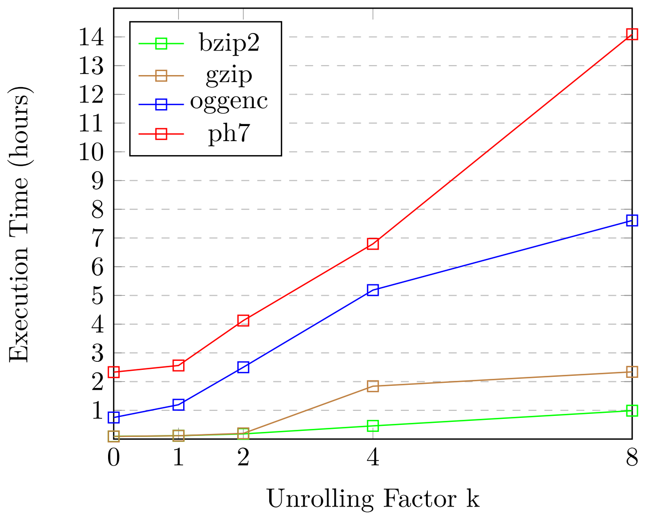


Figure 5.3: Total execution time in hours of loop unroll of the four different benchmarks.

`gzip` and `bzip2` take about 10 minutes to analyze with Alive without loop unroll, while `oggenc` and `ph7` take approximately an hour. In contrast to `k = 2`, loop unroll with `k = 1` shows a dramatic increase in block coverage with a relatively small impact on performance, therefore it makes sense to have Alive default to `k = 1` loop unroll.

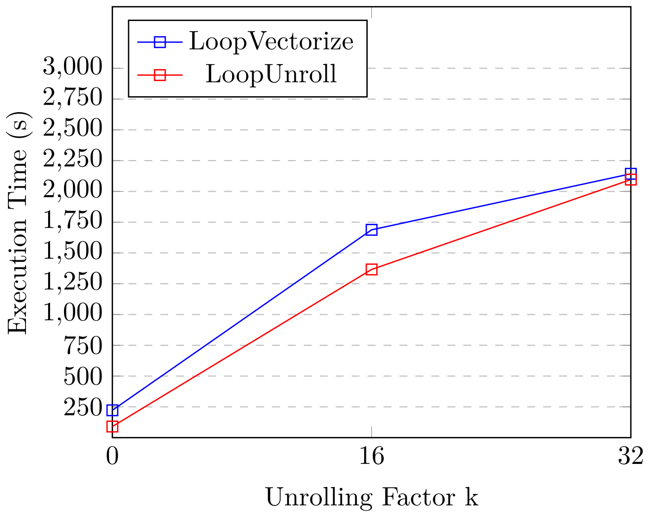


Figure 5.4: Total execution time in seconds through loop unroll of two LLVM test sets, namely loop vectorization and loop unroll.

The execution times for the LoopVectorize and LoopUnroll LLVM test sets with an unrolling factor of 16 and of 32 are shown in Fig. 5.4. Although program size increase is exponential as we increase the unrolling factor, the increases in execution time do not reflect the true results because of the use of timeouts.

Lastly, we observe the impact loop unrolling in Alive had on path coverage, timeouts and on checking preconditions in Tables 5.1 and 5.2. Overall, we see that the average number of paths covered in Alive increases drastically with higher loop unrolling factors due to a much larger increase in program size. This increase in the number of paths covered allows Alive to analyze programs more accurately. For example, Alive can consider an additional program path in analysis which allows for having more accurate value range representations of some program variables, and as a consequence allows Alive to consider more program behavior in its analysis.

	k = 0			k = 16			k = 32		
	Paths	Pre	Timeo.	Paths	Pre	Timeo.	Paths	Pre	Timeo.
LoopVectorize	1	341	72	5506132	247	182	68158496	193	192
LoopUnroll	1	77	16	43819259	47	30	276298252	39	31

Table 5.1: Average number of paths covered (Paths) over all functions in LoopVectorize and LoopUnroll; total number of times Alive reports transformations with false preconditions (Pre); the total number of timeouts (Timeo.) with loop unroll factors 0, 16, 32 with 0 corresponding to no loop unroll.

Preconditions are conditions that are checked by Alive before verification. If at least one is not satisfied then Alive does not verify the program and considers it incorrect. For instance, if a program does not have a path that Alive can cover. Recall that not all program paths are covered by Alive, since it does not visit any basic block twice. For some programs, loop unrolling can reduce the number of cases in which this happens, since some back edges are transformed into forward edges and some blocks are duplicated. The larger we set the unrolling factor, the more likely it is to have less of this kind of unsatisfied preconditions, as observed in our results in Table 5.2. In general, the number of timeouts should increase as we increase the loop unrolling factor, but there may be some exceptions. For instance, larger SMT queries may include more restrictions on the search space and push Z3 into solving some queries faster by letting Z3 take better search decisions early.

	k = 0			k = 1			k = 2		
	Paths	Pre	Timeo.	Paths	Pre	Timeo.	Paths	Pre	Timeo.
bzip2	1070	11	30	1137	10	33	6636	10	28
gzip	641	11	25	863	9	31	2193167	9	35
oggenc	191	13	220	383	11	241	19261129	11	243
ph7	43561	3	669	694782	3	685	50630040	3	695

	k = 4			k = 8		
	Paths	Pre	Timeo.	Paths	Pre	Timeo.
bzip2	2858158	10	25	566914555	6	27
gzip	183261696	8	37	411536806	7	40
oggenc	105819941	8	220	288820771	5	224
ph7	56210522	3	698	252313279	2	698

Table 5.2: Average number of paths covered (Paths) over all functions in each benchmark, total number of times Alive reports transformations with false preconditions (Pre); the total number of timeouts (Timeo.) for four different benchmarks with loop unroll factors 0, 1, 2, 4 and 8, where 0 corresponds to no loop unroll.

Overall, our results show significant increases in block coverage and path coverage, allowing for Alive's analysis to consider larger fractions of code for each program. Moreover, we have also seen a reduction in the number of unsatisfied preconditions when using higher values for the unrolling factor *k*, which in turn reduces the number of functions skipped, and Alive will be able to cover a larger amount of code in its analysis and thus further improving the chances of detecting miscompilation.

5.2 Bugs Found in LLVM

Overall, our loop unroll algorithm has enabled Alive to flag two LLVM tests as incorrect, which were reported to LLVM. The first bug involves the LoopReroll LLVM optimization, and the second involves LoopVectorization and ScalarEvolution.

5.2.1 LoopReroll/basic.ll

LoopReroll does the opposite of loop unroll, where the number of loop iterations is increased while the number of instructions in each loop iteration is decreased. LoopReroll is an optimization with the purpose of code size reduction. To illustrate how it works, consider Fig. 5.5.

```
for (int i = 0; i < 500; i+=3) {
  x[i] = foo();                        for (int i = 0; i < 1500; ++i) {
  x[i+1] = foo();                        x[i] = foo();
  x[i+2] = foo();                      }
}
```

Figure 5.5: Example 1 of LoopReroll with the original on left and the optimized one on the right.

The problem we found is that LoopReroll incorrectly reorders memory instructions when memory overlap is possible. Specifically, it can swap the order between a store and a load instruction to the same memory position and consequently alter the semantics of the program.

```
for (int i = 0; i < N; i+=2) {
  int a = p[i];                        for (int i = 0; i < 2*N; ++i) {
  int b = p[i+1];                        int a = p[i];
  p[i+1] = a + 1;                        p[i+1] = a + 1;
  p[i+2] = b + 1;                      }
}
```

Figure 5.6: Example 2 of LoopReroll with the original on left and the optimized one on the right.

An illustration of the problem is shown in the example transformation in Fig. 5.6. In one iteration of the source, we first read `p[i+1]`, then write to it, while in two iterations of the target we write first to `p[i+1]` in the first iteration, then in the second iteration we read from `p[i+1]` followed by a write to `p[i+2]`. Since this reordering of memory instructions changes the semantics of the program, the target will show different behavior in comparison with the source program and is therefore an incorrect transformation. Alive reported this test as incorrect when it is loop unrolled with an unroll factor of at least 1. A report for this bug was created at [39].

5.2.2 LoopVectorize/pr30654-phiscev-sext-trunc.ll

LoopVectorize is an optimization that condenses scalar instructions of multiple iterations of a loop into SIMD instructions using vectors. It does this by creating an additional loop based on vectors, while also keeping the original loop intact. Depending on the number of loop iterations, either the vectorized or the scalar loop is executed, and possibly both. For example, if we have a loop that will execute 7 scalar instructions, then with a loop vectorization factor of 4 the transformed program will execute one iteration of the vectorized loop and after that execute three iterations of the scalar loop.

LoopVectorize can also reference the ScalarEvolution optimization pass, which analyzes how variables that depend on a loop induction variable change over time, creating SCEV expressions for some of them. A SCEV expression for a loop induction variable `i` that starts with value 0 and increments by 1 on each iteration is represented as the SCEV {0,+,1}. This representation helps LLVM gain relevant information such as estimating the number of iterations for a given loop or for instance the final values of variables that depend on the loop induction variable.

Scalar evolution can also replace SCEVs with simpler ones through rewriting by making assumptions. These assumptions need to be checked during run-time before entering the vectorized loop, done by adding a new basic block just before the vectorized loop.

We found that SCEVExpander that generates instructions to check these assumptions can sometimes introduce true undefined behavior when a branch on poison is inserted. Recall that poison is a form of deferred undefined behavior, created from instructions such as *add* with the *nsw* flag. The *nsw* flag specifies that poison is returned if integer overflow occurs on the *add* instruction. Poison is also propagated as the result of some instructions that have poison as an input. If a control-flow altering instruction such as a *branch* receives poison as the jump condition, the program becomes non-deterministic and shows true undefined behavior. A report for this bug was created at [40].

5.3 Reduction of UB SMT Formulas

We evaluate the four different versions of our SMT formula reduction algorithm in terms of execution time and memory. The first version (v_1) is the current version of Alive without any changes and will serve as our baseline. UBReduction (v_2) corresponds to our basic implementation in Section 4.5, the versions UBReductionCondPostdom (v_3) and UBReductionHeuristic (v_4) use the post-dominator relationship described in Sections 4.10 and 4.11 respectively, with the latter being heuristic-driven.

Execution time is divided into three separate parts: Alive's analysis time to create formulas, SMT solving time with Z3, and total execution time. For obtaining the results in this section we also force Alive to check all Z3 queries created by not stopping as soon as some query is either found satisfiable or times out. Since Alive creates more than one SMT query, forcing Alive to check all of them independently of the version used allows for a fair comparison of execution time.

We run our benchmarks by running each different version of our implementation separately on three different programs, namely `bzip2`, `gzip` and `oggenc`. The versions we used for LLVM and Z3 correspond to the Github commit hashcodes `941005f51a` and `1f48eabea5` respectively. Our implementation of this algorithm extends Alive with base version `eadbbd9e28`.

For each benchmark, Alive checks each program function individually. These results are then summed and shown in Table 5.3 with execution times in seconds (s), and the memory allocated for formulas in Z3 in megabytes (MB) in Table 5.4.

Observing the results obtained, the algorithms did not lead to a notable additional cost in Alive-only

	Alive-only			SMT-only			Total		
	bzip2	gzip	oggenc	bzip2	gzip	oggenc	bzip2	gzip	oggenc
v_1	765.20	133.68	905.95	422.48	656.78	4935.92	1187.68	790.46	5841.87
v_2	761.72	133.34	913.58	422.85	658.07	5231.29	1184.57	791.41	6144.87
v_3	759.95	131.66	923.18	421.66	652.10	5236.29	1181.61	783.76	6159.47
v_4	766.13	131.04	929.97	421.78	661.24	5010.50	1187.91	792.28	5940.47

Table 5.3: Execution times for the four versions and the three different benchmarks measured in seconds (s).

	bzip2	gzip	oggenc
v_1	276.616	223.936	2821.428
v_2	276.506	224.109	2800.481
v_3	276.790	223.162	2740.787
v_4	276.869	223.048	2781.686

Table 5.4: Memory usage for created formulas for four versions and three different benchmarks measured in megabytes (MB).

analysis time. Results also show that the SMT solving time did not reduce as much as we would have hoped for. This could be due to the formulas we changed not being a large enough fraction of the total formula, or Z3 is doing a particularly good job in efficiently ignoring SMT redundancy.

The Z3 routine *Z3_get_estimated_alloc_size()* gives an estimation of the total memory allocated for the SMT formulas created and was used to obtain the memory usage results presented here. Given that only expressions referenced in the formula are relevant, and since our changes affect only the formulas that encode program undefined behavior, this metric only gives a rough idea of the reduction in formula sizes compared to the current implementation of Alive.

5.4 Reduction of Phi SMT Formulas

In addition to reducing the SMT formula sizes for undefined behavior semantics, we also improved how the semantics of phi instructions are represented in SMT through an algorithm we presented in Section 4.12. This algorithm is also designed to reduce the time spent checking proofs with an SMT solver.

	v_0	v_1	Speedup
bzip2	373.24	359.81	1.037 (-3.73%)
gzip	410.51	399.09	1.029 (-2.86%)
oggenc	3138.42	2961.84	1.060 (-5.96%)

Table 5.5: Execution times for the current version of Alive v_0, and PhiReduction v_1 for reduction of phi formula sizes. The versions of LLVM, Z3 and Alive used are the same as those in Section 5.3.

Observing these results, we conclude that the algorithm did improve the total execution time. The algorithm reduces the prefixes of SMT formulas encoded for phi instructions by replacing them with expressions containing only the necessary information about program control flow to decide between the phi entries. The `oggenc` benchmark shows a larger reduction in execution time likely due to it containing much larger functions where more control-flow instructions are present in paths reaching phi instructions. This reduction in execution time is expected to grow as program sizes increase.

Chapter 6

Conclusions and Future Work

In this document we presented two main contributions as a means to improve Alive in terms of both performance and analysis coverage. Moreover, we have been able to find and report two loop-related bugs in LLVM.

We presented a new loop unrolling algorithm for Alive IR that has shown to significantly improve Alive's block coverage. We have seen an increase of roughly 30% for most tests we performed, some even reaching improvements of up to 40%. These increases in block coverage are large improvements to how much code Alive can cover for programs with loops. This significantly increased the chances of Alive finding compiler bugs.

We presented two algorithms for reducing the sizes of SMT expressions in Alive through the use of useful graph theory concepts such as dominators and post-dominators. Specifically, we reduced the sizes of SMT formulas needed to represent undefined behavior and those representing the semantics of phi instructions as an effort to reduce the time spent in the Z3 SMT solver, which is a dominant factor in Alive's total execution time. Although no performance decrease was observed with smaller SMT formulas for undefined behavior likely due to how well Z3 performs, we did see some notable improvement when reducing the size of SMT expressions that represent the semantics of phi instructions.

Lastly, we found that the LoopReroll LLVM optimization pass incorrectly reordered memory instructions when the memory positions they accessed could overlap with one another across loop iterations. We also found that some uses of SCEVExpander in the LoopVectorize optimization pass could sometimes introduce true undefined behavior. Both incorrect transformations introduced undesired program behavior in the compiled program that we would never observe from running the unoptimized program.

For future work, it would be interesting to explore better techniques to increasing analysis coverage of programs containing irreducible loops in a practical and beneficial manner. It would also be interesting to find methods for choosing values for the unrolling factor k that better suit each individual loop, further improving Alive's analysis coverage while minimizing the number of basic blocks inserted during loop unroll. Furthermore, we expect that the integration of a loop unrolling algorithm into Alive will enable the detection of more loop-related bugs in the future as a step forward towards less compiler bugs and lower chances of program miscompilation.

Bibliography

[1] N. P. Lopes, D. Menendez, S. Nagarakatte, and J. Regehr. Provably correct peephole optimizations with Alive. *SIGPLAN Not.*, 50(6):22–32, June 2015. ISSN 0362-1340. URL `http://doi.acm.org/10.1145/2813885.2737965`.

[2] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '98, pages 151–166, Berlin, Heidelberg, 1998. Springer-Verlag. ISBN 3-540-64356-7. URL `http://dl.acm.org/citation.cfm?id=646482.691453`.

[3] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. *SIGPLAN Not.*, 46(6):283–294, June 2011. ISSN 0362-1340. URL `http://doi.acm.org/10.1145/1993316.1993532`.

[4] V. Le, M. Afshari, and Z. Su. Compiler validation via equivalence modulo inputs. *SIGPLAN Not.*, 49 (6):216–226, June 2014. ISSN 0362-1340. URL `http://doi.acm.org/10.1145/2666356.2594334`.

[5] V. Le, C. Sun, and Z. Su. Finding deep compiler bugs via guided stochastic program mutation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 386–399, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3689-5. URL `http://doi.acm.org/10.1145/2814270.2814319`.

[6] X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009. ISSN 0001-0782. URL `http://doi.acm.org/10.1145/1538788.1538814`.

[7] S. Lerner, T. Millstein, E. Rice, and C. Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 364–377, New York, NY, USA, 2005. ACM. ISBN 1-58113-830-X. URL `http://doi.acm.org/10.1145/1040305.1040335`.

[8] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3-540-78799-2, 978-3-540-78799-0. URL `http://dl.acm.org/citation.cfm?id=1792734.1792766`.

[9] V. Ganesh. *Decision Procedures for Bit-vectors, Arrays and Integers.* PhD thesis, Stanford, CA, USA, 2007. AAI3281841.

[10] A. Cimatti, A. Griggio, B. Schaafsma, and R. Sebastiani. The MathSAT5 SMT Solver. In N. Piterman and S. Smolka, editors, *Proceedings of TACAS*, volume 7795 of *LNCS*. Springer, 2013.

[11] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovi'c, T. King, A. Reynolds, and C. Tinelli. CVC4. In G. Gopalakrishnan and S. Qadeer, editors, *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, July 2011. URL `http://www.cs.stanford.edu/~barrett/pubs/BCD+11.pdf`. Snowbird, Utah.

[12] A. Niemetz, M. Preiner, and A. Biere. Boolector 2.0 system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 9:53–58, 2014 (published 2015).

[13] M. Dahiya and S. Bansal. Black-box Equivalence Checking across compiler optimizations. In *Asian Symposium on Programming Languages and Systems*, pages 127–147. Springer, 2017.

[14] J. Kang, Y. Kim, Y. Song, J. Lee, S. Park, M. D. Shin, Y. Kim, S. Cho, J. Choi, C.-K. Hur, and K. Yi. CRELLVM: Verified credible compilation for LLVM. *SIGPLAN Not.*, 53(4):631–645, June 2018. ISSN 0362-1340. URL `http://doi.acm.org/10.1145/3296979.3192377`.

[15] G. C. Necula. Translation validation for an optimizing compiler. *SIGPLAN Not.*, 35(5):83–94, May 2000. ISSN 0362-1340. URL `http://doi.acm.org/10.1145/358438.349314`.

[16] M. Stepp, R. Tate, and S. Lerner. Equality-based translation validator for LLVM. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 737–742, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-22109-5. URL `http://dl.acm.org/citation.cfm?id=2032305.2032364`.

[17] J.-B. Tristan, P. Govereau, and G. Morrisett. Evaluating value-graph translation validation for LLVM. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 295–305, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0663-8. URL `http://doi.acm.org/10.1145/1993498.1993533`.

[18] J.-B. Tristan and X. Leroy. Formal verification of translation validators: A case study on instruction scheduling optimizations. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 17–27, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781595936899. doi: 10.1145/1328438.1328444. URL `https://doi.org/10.1145/1328438.1328444`.

[19] J.-B. Tristan and X. Leroy. Verified validation of lazy code motion. *SIGPLAN Not.*, 44(6):316–326, June 2009. ISSN 0362-1340. URL `http://doi.acm.org/10.1145/1543135.1542512`.

[20] J.-B. Tristan and X. Leroy. A simple, verified validator for software pipelining. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 83–92, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-479-9. URL `http://doi.acm.org/10.1145/1706299.1706311`.

[21] C. Barrett, Y. Fang, B. Goldberg, Y. Hu, A. Pnueli, and L. Zuck. TVOC: A Translation Validator for Optimizing Compilers. In *International Conference on Computer Aided Verification*, pages 291–295. Springer, 2005.

[22] B. Goldberg, L. Zuck, and C. Barrett. Into the loops: Practical issues in translation validation for optimizing compilers. *Electron. Notes Theor. Comput. Sci.*, 132(1):53–71, May 2005. ISSN 1571-0661. URL `http://dx.doi.org/10.1016/j.entcs.2005.01.030`.

[23] M. Dahiya and S. Bansal. Modeling undefined behaviour semantics for checking equivalence across compiler optimizations. In *Haifa Verification Conference*, pages 19–34. Springer, 2017.

[24] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation: A new approach to optimization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 264–276, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-379-2. URL `http://doi.acm.org/10.1145/1480881.1480915`.

[25] MLIR: Multi-level intermediate representation for compiler infrastructure. URL `https://github.com/tensorflow/mlir`.

[26] SIL: Swift intermediate language. URL `https://github.com/apple/swift/blob/master/docs/SIL.rst`.

[27] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, Berlin, Heidelberg, 1982. Springer-Verlag. ISBN 3-540-11212-X. URL `http://dl.acm.org/citation.cfm?id=648063.747438`.

[28] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, pages 193–207, Berlin, Heidelberg, 1999. Springer-Verlag. ISBN 3-540-65703-7. URL `http://dl.acm.org/citation.cfm?id=646483.691738`.

[29] L. Cordeiro, B. Fischer, and J. Marques-Silva. SMT-based bounded model checking for embedded ANSI-C software. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 137–148, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3891-4. URL `https://doi.org/10.1109/ASE.2009.63`.

[30] CBMC Bounded Model Checker for C and C++ programs. `https://www.cprover.org/cbmc/`. Accessed: 2019-11-16.

[31] E. Clarke and D. Kroening. Hardware verification using ANSI-C programs as a reference. In *Proceedings of the 2003 Asia and South Pacific Design Automation Conference*, ASP-DAC '03, pages 308–311, New York, NY, USA, 2003. ACM. ISBN 0-7803-7660-9. URL `http://doi.acm.org/10.1145/1119772.1119831`.

[32] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976. ISSN 0001-0782. URL `http://doi.acm.org/10.1145/360248.360252`.

[33] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3):50:1–50:39, May 2018. ISSN 0360-0300. URL `http://doi.acm.org/10.1145/3182657`.

[34] V. Chipounov, V. Kuznetsov, and G. Candea. The S2E platform: Design, implementation, and applications. *ACM Trans. Comput. Syst.*, 30(1):2:1–2:49, Feb. 2012. ISSN 0734-2071. URL `http://doi.acm.org/10.1145/2110356.2110358`.

[35] J. Lee, Y. Kim, Y. Song, C.-K. Hur, S. Das, D. Majnemer, J. Regehr, and N. P. Lopes. Taming undefined behavior in LLVM. *SIGPLAN Not.*, 52(6):633–647, June 2017. ISSN 0362-1340. URL `http://doi.acm.org/10.1145/3140587.3062343`.

[36] P. Havlak. Nesting of reducible and irreducible loops. *ACM Trans. Program. Lang. Syst.*, 19(4): 557–567, July 1997. ISSN 0164-0925. doi: 10.1145/262004.262005. URL `https://doi.org/10.1145/262004.262005`.

[37] J. Janssen and H. Corporaal. Making graphs reducible with controlled node splitting. *ACM Trans. Program. Lang. Syst.*, 19(6):1031âĂŞ1052, Nov. 1997. ISSN 0164-0925. doi: 10.1145/267959.269971. URL `https://doi.org/10.1145/267959.269971`.

[38] K. D. Cooper, T. J. Harvey, and K. Kennedy. A simple, fast dominance algorithm. Technical report, 2006.

[39] LLVM bug report for loopreroll/basic.ll. `https://bugs.llvm.org/show_bug.cgi?id=47658`, . Accessed: 12-10-2020.

[40] LLVM bug report for loopvectorize/pr30654-phiscev-sext-trunc.ll. `https://bugs.llvm.org/show_bug.cgi?id=47769`, . Accessed: 12-10-2020.