Bachelor of Science Thesis

Efficient Computation of Weakest Preconditions

Michael Beaumont

September 30, 2015

Chair for Software Modeling and Verification
RWTH Aachen University

First Reviewer:
Prof. Dr. Thomas Noll

Second Reviewer:
Prof. Dr. Ir. Joost-Pieter Katoen

Supervisors:
Tim Lange
Dr. Martin Neuhäußer

Abstract

Predicate transformers are logical formula generators used in verification algorithms such as CEGAR. Much of the runtime of these algorithms is spent checking logical formulas using SMT solvers and thus the size and quality of those formulas is of utmost importance to achieving efficient verification. We investigate improvements to predicate transformers that decrease the size of the generated verification conditions and examine integrating the improved predicate transformers into a practical verification framework. The results of our benchmarks are ultimately inconclusive but show potential for drastically reducing verification time.

Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe. Sämtliche Zitate wurden eindeutig als solche gekennzeichnet.

Aachen, den 30. September 2015 _____

Michael Beaumont

Contents

List of Tables and Figures

1 Introduction

Computers and automated systems are responsible for the most safety-critical systems in the world. They have a reputation as being more reliable than human-controlled systems. However, every computer system is ultimately built and maintained by humans. As applications grow in complexity, it becomes difficult to ensure that they behave as intended. In real-world situations, programs may encounter conditions never considered during their design. In order to truly guarantee reliability of these systems, they must be prepared to handle these conditions and still function correctly. Unseen edge cases and countless potential circumstances render testing an incomplete and insufficient method of verification.

The desired behavior of programs can be expressed as *correctness properties*, which are requirements such as something a program will never do or something that will always be true. For example, a program ultimately designed to measure the power output of an engine should never report a negative number.

With the advent of the field of computer science, verification emerged as a method for determining whether a program satisfies a correctness property. As the complexity of computerized systems has grown, so too have the capabilities of verification methods. Ideally, verification should be an automatic examination of correctness properties. An algorithm for automatic verification does not execute the program but rather decides, without human intervention, whether a property holds by analysis of a representation of the program. Additionally, verification takes into account *all* possible scenarios under which the program runs, no matter how unlikely.

At any time, a program has a state. A state is nothing more than a description of what information is known to the program at that time. Correctness properties are typically conditions on the state of a program during or after execution. For example, the correctness property for the aforementioned case would be that the known power output must never contain a negative number.

Because verification is based on absolute and confident decisions about correctness properties, programs along with correctness properties are typically analyzed using mathematical models. The semantics of program execution and the semantics of properties of programs are given interpretations in these models that can be investigated using algorithms that in turn make correctness decisions.

As early as 1975, the field of automatic verification for software was relatively well explored and that year, Edsger W. Dijkstra proposed a framework for formally verifying desired properties of programs [Dij75]. To this end, he introduced a language to represent programs and a method for verifying that language, the guarded command language and predicate transformers. In short, the guarded command language describes the semantics with which a program transforms a given state. Given a correctness property expressed as a logical formula, a predicate transformer transforms that formula into one guaranteeing the formula holds after executing the program. This logical formula can then be analyzed to decide whether the property holds.

A general problem plagues the field of verification, the state-space explosion. Programs are complex and can act in any number of ways. From a starting state, there are potentially a prohibitively large number of states a program can end in, and these must all be considered in order to properly verify. Specifically, with each additional boolean variable the number of

states double, causing an exponential growth of the number of states. Variables taking on more complex values make this growth even more dramatic. This massive number of states can potentially prevent decisions from ever being made due to slowdown and memory issues.

Counterexample guided abstract refinement (CEGAR) was developed as a general purpose verification method in order to minimize this problem. The algorithm attempts to abstract and thereby simplify a program and then verify the abstraction, refining the abstraction if it is not sufficient. CEGAR makes use of predicate transformers in the course of doing this.

Unfortunately, predicate transformers as introduced by Dijkstra suffer from a related problem in that the logical formulas they output can grow exponentially in the size of the input programs.

The focus of this thesis is chiefly the predicate transformers. Firstly, we discuss the background underpinning verification of programs, including SMT solvers and the language used to abstract programs, the GCL. Then we move on to a motivation for predicate transformers and a motivation for their application to verification.

Subsequently, the problems with classic predicate transformers are described and demonstrated in order to motivate a new interpretation of programs and the predicate transformer operation, introduced by Flanagan and Saxe [FS01] and also explored by Rustan and Leino [RL05]. We then explore this interpretation in a detailed way from a different perspective and prove the equivalence of both the new interpretation of programs and the new predicate transformers to their classic counterparts. The new predicate transformers bring the advantage of better asymptotic behavior versus the classic ones, which is also analyzed and proven in the thesis.

Finally, we move on to the CEGAR algorithm. We briefly go through the theory behind CEGAR as used in a practical verification framework. The uses of the predicate transformers and where the new version can be applied are then examined. We discuss and show the correctness of the changes necessary for incorporating the new predicate transformers before analyzing their performance characteristics with a series of benchmarks.

2 Background

Although verification is concerned with checking real-world programs, verification itself is based on making sound conclusions about mathematical models. Therefore, an understanding of the foundations on which the remainder of the thesis is built is crucial.

2.1 Logic and SMT

Structuring problems in a logical form allows expression in a unambiguous and mathematical form. Therefore, this section introduces a few concepts on which the rest of the thesis will rely.

A logic is a mathematical structure that includes a universe, or domain of objects, along with symbols. The universe is the collection of objects over which we want to make statements. The symbols can be combined according to given rules, the *syntax*, to form expressions called formulas. Symbols include the logical symbols such as boolean connectives and variables, which serve as placeholders for objects in the universe, as well as the non-logical symbols like constants and functions. The set of logical symbols is usually fixed over the type of logic while the set of non-logical symbols, the *signature*, varies in connection with how the logic is used. The *semantics* of the logic define the meaning of non-logical symbols [KS08].

The syntax of the logic consists of rules about how to construct valid formulas. Boolean connectives, for instance, usually have two operands that must be boolean values. The application of a boolean connective to its operands is also a boolean value.

The semantics for a logic are defined by what *meaning* is given to the symbols. The non-logical symbols change their meaning according to a given *interpretation*. An interpretation is an additional mathematical object that provides meaning to the non-logical symbols, a universe of objects, and assignments for the variables if necessary. This includes which objects belong to a relation as well as the interpretation of functions. Given an interpretation, a formula can be evaluated to true or false using the assignments of the variables, the constants, the functions and the relations.

Logic provides a framework for expressing problems mathematically. Decision problems, questions that are answered with a yes or no answer, often appear in verification. A ubiquitous and fundamental decision problem is the satisfiability problem.

Definition 1. Satisfiability Problem (SAT)

Given is a formula φ over some specific logic. The SAT problem asks whether there is an interpretation M such that M makes φ true (written $M \models \varphi$), also called a *model*.

Example 1. Propositional Logic

In propositional logic, there are only the typical logical symbols, $\{\neg, \wedge, \vee, \Rightarrow, true, false\}$ and the variables.

An example formula would be $X \wedge Y \wedge (Z \vee \neg X)$.

The SAT problem asks whether this formula is satisfiable, which the interpretation, $\{X = true, Y = true, Z = true\}$, confirms to be true.

Propositional logic is relatively simple. This is because propositional logic considers solely propositions, i.e. variables which represent either *true* or *false* and so the only objects are *true* and *false*. Many practical questions about safety of systems can nonetheless be expressed as formulas in propositional logic and as satisfiability questions.

Example 2. A circuit, accepting some input and giving some output, can be expressed as a boolean formula.

The circuit equivalence checking problem asks whether two circuits give the same output for some input. The $\oplus$ operator returns true if its two operands are not equal. The disjunction of $\oplus$ applied to each bit of the output is true if any of the two output bits are not equal.

If this disjunction is satisfiable, the two circuits are not equivalent [Mar08].

Of course, any decision problem can be posed but usefulness comes only through a method for answering the problem. The question of whether or not an algorithm exists that can make the decision is based on how complex the problem is.

Definition 2. Decidability

A decision problem is decidable if and only if there exists an always-terminating algorithm which always gives a correct answer to the problem.

Because it is so straightforward, propositional logic has a very convenient property: the satisfiability problem is decidable. A simple algorithm for this would be to simply enumerate all possible combinations of assignments to variables of the formula in question, the truth-table method. This algorithm has a time complexity of $O(2^n)$ however, that is, the time required grows exponentially, rendering it theoretically impractical for real-world programs.

However, given that the satisfiability problem has found so many practical applications, much time and research has been invested into improving the algorithm used to decide satisfiability. SAT solvers are applications that take advantage of these improvements and accept a propositional formula as input to decide whether it is satisfiable.

Propositional logic is useful in some cases, but its simplicity keeps it from being able to capture many other practical problems. For example, problems about the natural numbers require more than $true$ or $false$.

Definition 3. First order logic (FO)

FO is an extension of propositional logic. It adds relations, functions, equality and allows for a different universe. Additionally, variables can be quantified over. The quantifiers are $\forall$ and $\exists$, respectively universal and existential quantification. These ask whether a formula is true for all possible instances of a variable or for at least one. A variable in a formula not bound by a quantifier is called *free*, otherwise it is *bound* [KS08].

An example of an FO formula is: $\forall x.\, x > 0$

The signature of a logic can have any arbitrary meaning in a logic. However, their semantics can be constrained using a *theory*, or a set of formulas with no free variables. Given a theory, deductions can be made about other, novel formulas.

Definition 4. Satisfiability Modulo Theories (SMT)

Given are a theory $\mathbb{T}$ and a formula φ. From the set of all models of $\mathbb{T}$ is there a model M such that $M \models \varphi$?

Figure 2.2.1: Grammar of the guarded command language (GCL) [FS01]

$$
\begin{array}{lll}
\langle s \rangle & ::= & \langle id \rangle := \langle expr \rangle \\
 & | & \textbf{assert } \langle expr \rangle \\
 & | & \textbf{assume } \langle expr \rangle \\
 & | & \langle s \rangle \ ; \ \langle s \rangle \\
 & | & \langle s \rangle \ \square \ \langle s \rangle \\
\langle expr \rangle & \in & \mathbb{D}
\end{array}
$$

Example 3. Quantifier-free SAT with equality and uninterpreted functions (QFEUF)

For QFEUF, formulas are assumed to be quantifier-free. Equality is assumed to be an equivalence relation in FO and congruence is given in the theory by $\forall u \forall v.\, u = v \Rightarrow f(u) = f(v)$ for all functional symbols f. The SAT problem for this theory is whether some quantifier-free formula is satisfiable by a model of the theory.

An example formula for this theory would be: $u = v \wedge v \neq w \wedge f(u) = f(w)$

A program called an SMT solver accepts logical formulas and attempts to decide their satisfiability given some set of theories. The SMT solver knows how to use the theories to solve for satisfiability. For example, an SMT solver knows how to decide problems in (QFEUF) because it understands the rules of equality and congruence and so will try to discover if the formula can be satisfied while still following these rules.

Because of the nature of the SAT problem as well as added complexity from extending the logic, the theory in which a problem is posed has a significant effect on the speed with which and even whether satisfiability can be determined. Quantifiers, for example, increase the difficulty significantly, and this specifically becomes important for predicate transformers.

2.2 GCL

Programming languages differ in how they express programs semantically and syntactically. While logic allows us to formally and uniformly express questions, it is not clear how to do this for programs. One approach is to use a guarded command language.

A method for verifying programs should not be restricted to only one language if possible but rather should be applicable for any program. Nevertheless it would certainly be advantageous to only have to reason about one fixed language. Each high-level programming language, being designed for specific purposes, has its own idiosyncrasies and special cases. Attempting to handle each one separately would complicate any method for verification. A better approach is to write an algorithm for one, more general language. If verification is restricted to this *intermediate* language then programs can be translated into it and the translation can be used to verify the original program. Adding support for another language would simply mean writing an additional translator, rather than modifying any algorithm involved in verification.

Dikjstra introduced the initial version of the guarded command language in [Dij75], the grammar of the GCL used in this thesis is shown in Fig. 2.2.1. It fulfills the requirements of being a relatively simple language free of peculiarities, encapsulating only the very fundamentals of programs.

Definition 5. Program domain and state

Each GCL program has a fixed mathematical domain denoted in general as $\mathbb{D}$. A state ω is a mapping from variables to elements of the GCL domain $\mathbb{D}$. The set of potential states for a program is denoted Ω.

Example 4. Unsigned bit vectors

BV_{u32} is a program domain consisting of all 32 bit binary numbers, interpreted as unsigned. The usual mathematical operations are defined such that when the results are greater than 32 bits, any extra bits are truncated, such that results are again in BV_{u32}

Definition 6. Paths and execution on states

$S(\omega)$ is the set of possible states resulting from executing S with initial state ω.

A *path* is the sequence of semantic rules used to go from one state to another or intuitively, a single execution of a program on a state.

In a slight abuse of notation, we write $\omega \models \varphi$ to mean that assigning the free variables in φ with their values from ω gives a true formula.

There are a minimum of language statements and they are built inductively. The := operation behaves like a typical assignment statement and alters the program state. The last two atomic statement types are **assert** and **assume**. These take an expression which has a value in the program's domain $\mathbb{D}$ as an argument. Both statements have the ability to influence the execution state of the program. When the operand of an **assert** statement is false, the execution *goes wrong*. Otherwise, execution continues. If the operand of **assume** is false, execution can simply not continue [RL05].

There are two ways to compose statements together. The first is the sequence statement $S\,;T$, which creates a new statement where first S is executed and then T. The second way is the choice statement, $S \,\square\, T$. Execution of a choice statement is defined as choosing exactly one of the statements non-deterministically and executing that statement. Exactly which statement is chosen is not predictable, a type of semantics known as *demonic* non-determinism [BW89].

Execution of compositional statements is dependent on their substatements. A sequence statement goes wrong if either substatement goes wrong. If execution is blocked through an assume statement, the second statement is never executed. Assume statements therefore allow for control flow, which would otherwise be absent. Choice statements going wrong is more complicated due to non-determinism. A choice statement only definitely goes wrong if every substatement goes wrong. Otherwise, this is dependent on which path is chosen.

The precise operational semantics are given in Fig. 2.2.2. A statement and state evaluate to a state and an execution status, either successful ($\top$) or gone wrong ($\bot$).

Such a limited choice of statements would appear to restrict what we can express in our language. In fact, by combining choice statements and **assume**, we can recreate constructs like **if** as follows:

$$\textbf{if } e \textbf{ then } S \textbf{ else } T \equiv (\textbf{assume } e;\, S) \,\square\, (\textbf{assume } \neg e;\, T)$$

Execution is simply blocked if the operand of the **assume** statement evaluates to false. Although this statement has a non-deterministic choice, some determinism has been regained. One and only one substatement will be executed given the value of e from the program state.

Very important to note is that these GCL programs are necessarily acyclic. No construct is available that lets us return to an earlier point of execution. This property makes statically analyzing GCL programs relatively straightforward. It also makes it intuitively clear that we can easily transform a certain subset of programs from high-level languages into GCL.

Figure 2.2.2: Operational semantics of the GCL

$$\text{assert-f}\ \frac{e\ false}{(\textbf{assert}\ e, \omega) \Downarrow (\omega, \bot)} \qquad \text{assert-t}\ \frac{e\ true}{(\textbf{assert}\ e, \omega) \Downarrow (\omega, \top)}$$

$$\text{assume}\ \frac{e\ true}{(\textbf{assume}\ e, \omega) \Downarrow (\omega, \top)} \qquad \text{assign}\ \frac{}{(x := e, \omega) \Downarrow (\omega[x \to e], \top)}$$

$$\text{seq-1}\ \frac{(S_1, \omega) \Downarrow (\omega', \top) \qquad (S_2, \omega') \Downarrow (\omega'', s)}{(S_1\,;S_2, \omega) \Downarrow (\omega'', s)}$$

$$\text{seq-2}\ \frac{(S_1, \omega) \Downarrow (\omega', \bot)}{(S_1\,;S_2, \omega) \Downarrow (\omega', \bot)}$$

$$\text{choice-1}\ \frac{(S_1, \omega) \Downarrow (\omega', s)}{(S_1 \square S_2, \omega) \Downarrow (\omega', s)} \qquad \text{choice-2}\ \frac{(S_2, \omega) \Downarrow (\omega', s)}{(S_1 \square S_2, \omega) \Downarrow (\omega', s)}$$

GCL was developed as a language for verification, not for programming. This means that despite the defined semantics of the language, GCL programs are never executed. They are used to *model* real programs.

2.3 Predicate Transformers

Predicate transformers provide the link between verification and the formalities introduced in the previous two sections. SMT solvers allow for automated decisions about logical formulas. In order to move from programs to logical formulas, however, we need predicate transformers.

Definition 7. Predicate Transformer
Predicate transformers are functions $p : GCL \times \mathcal{L} \to \mathcal{L}$ that map a program and a logical formula to another logical formula [BM07].

Definition 8. Verification Condition (VC)
Verification conditions are logical formulas expressing some condition on program state. These are typically generated by predicate transformers [BM07].

Verification conditions can be used in three different ways. The first is to ask whether a specific program state satisfies the VC and evaluate the formula using the program state, assigning the values of the state to the corresponding variables. Secondly, we can ask whether all possible program states satisfy the formula, i.e whether the formula is *valid*. Lastly, we can ask whether a VC is *satisfiable* by some state.

Predicate transformers are found in the context of Hoare triples, which are combinations of a precondition, a program and a postcondition denoted $\{P\}S\{Q\}$ [BM07]. For example, the precondition $x > 0$, the program $x := x - 1$, and the postcondition $x \geq 0$ make up a Hoare triple. The triple is a property of the program that can be read: if the precondition holds for a state, after executing S the postcondition should hold. We will see how they can be proven using predicate transformers.

Figure 2.3.1: The translation from GCL to weakest precondition [FS01]

P	$wp(P, Q)$
$x := e$	$Q[x \to e]$
assert e	$e \wedge Q$
assume e	$e \Rightarrow Q$
$S\,;T$	$wp(S, wp(T, Q))$
$S \square T$	$wp(S, Q) \wedge wp(T, Q)$

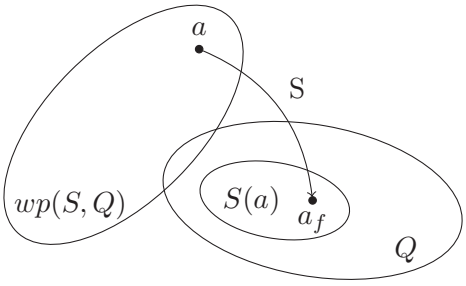


Figure 2.3.2: The relationship between weakest preconditions and program execution [BM07]

Informally, predicate transformers can be considered translators of the GCL semantics over logical formulas. They are calculated by traversing the GCL program to the atomic statements and inductively constructing a logical formula using those statements and the input formula.

2.3.1 Weakest Precondition

The first predicate transformer is the weakest precondition (WP). The weakest precondition describes program states from which a predicate holds post program execution.

The essence of the weakest precondition is the following [BM07]:

$$\text{If } a \models wp(S, Q) \text{ then } a' \models Q, \text{ for all } a' \in S(a)$$

That is, execution of S on a will always result in Q as shown in Fig. 2.3.2. It has the name *weakest* precondition because it is simply the least restrictive logical formula satisfying these conditions. If some other predicate O exists such that all satisfying states end up in Q after executing S, then $O \Rightarrow wp(S, Q)$. Therefore, the WP can be used to verify the Hoare triple $\{P\}S\{Q\}$ by checking the validity of the formula $P \Rightarrow wp(S, Q)$ [BM07].

Figure 2.3.1 shows the translation from GCL into WP. Each translation from statement to WP reflects GCL execution. The assignment state, like its effect on a program state, simply substitutes its right-hand side for all instances of the left-hand side in the predicate. Assert statements are enforced and lead to the WP evaluating to $false$ if they are not true. Assume statements are allowed to evaluate to $false$ and what might happen afterwards is ignored. If they evaluate to $true$, verification continues on the remaining program.

The assume statement can be used to generate so-called *miraculous* statements [BW89]. Consider the program in Fig. 2.3.3. This is called a miraculous statement because $wp(S, false) \neq false$. The idea is that the execution function is not a total function with arbitrary use of assume

Figure 2.3.3: Miraculous GCL program

```
  assume x > 0;
 []
  assume x < 0;
```

Figure 2.3.4: Simple example GCL program

```
x := a + b;
assert x > a;
  assume x >= 0;
  y := 1
 []
  assume x < 0;
  y := -1
x := x*y;
```

statements. For example, the execution of S on state $a = \{x \to 0\}$ does not result in a final state, the execution blocks and therefore, the state vacuously satisfies $wp(S, false)$.

Because the weakest precondition is a statement about all possible executions, the end state must be in Q no matter which path is taken. The algorithm therefore uses a conjunction over the weakest preconditions for each path. As a sequence statement is analyzed, the WP of the last statement is used as the postcondition for the WP of the preceding statements. To understand this, consider that state a satisfying the WP of the last statement simply guarantees that executing the last statement with a satisfies the postcondition. Thus, the preceding statements must end up in such a state a in order for the entire sequence to guarantee the postcondition.

We can examine a small program (Fig. 2.3.4) to show the WP translation. This program calculates the absolute value of $a + b$ as long as $b > 0$. The VC $wp(S, x > 0)$ will verify this:

$$
\begin{aligned}
wp(S, x > 0) &\equiv wp(x := a + b; \textbf{assert } x > a; C; x := x * y, x > 0) \\
&\equiv wp(x := a + b; \textbf{assert } x > a; C; wp(x := x * y, x * y > 0)) \\
&\equiv wp(x := a + b; \textbf{assert } x > a; C, x * y > 0) \\
&\equiv wp(x := a + b; \textbf{assert } x > a, wp(C, x * y > 0)) \\
wp(C, x * y > 0) &\equiv wp(C_1, x * y > 0) \wedge wp(C_2, x * y > 0) \\
wp(C_1, x * y > 0) &\equiv wp(\textbf{assume } x >= 0; y := 1, x * y > 0) \\
&\equiv wp(\textbf{assume } x >= 0, wp(y := 1, x * y > 0)) \\
&\equiv wp(\textbf{assume } x >= 0, x * 1 > 0) \\
&\equiv x \geq 0 \Rightarrow x * 1 > 0 \\
wp(C_2, x * y > 0) &\equiv wp(\textbf{assume } x < 0; y := -1, x * y > 0) \\
&\equiv wp(\textbf{assume } x < 0, wp(y := -1, x * y > 0)) \\
&\equiv wp(\textbf{assume } x < 0, x * -1 > 0) \\
&\equiv x < 0 \Rightarrow x * -1 > 0
\end{aligned}
$$

Figure 2.3.5: The translation from GCL to weakest liberal precondition [RL05]

P	$wlp(P, Q)$
$x := e$	$Q[x \to e]$
assert e	$e \Rightarrow Q$
assume e	$e \Rightarrow Q$
$S\,;T$	$wlp(S, wlp(T, Q))$
$S \square T$	$wlp(S, Q) \wedge wlp(T, Q)$

$$
\begin{aligned}
wp(C, x * y > 0) &\equiv (x \geq 0 \Rightarrow x * 1 > 0) \wedge (x < 0 \Rightarrow x * -1 > 0) \\
wp(S, x > 0) &\equiv wp(x := a + b; \textbf{assert } x > a, wp(C, x * y > 0) \\
&\equiv wp(x := a + b, wp(\textbf{assert } x > a, wp(C, x * y > 0)) \\
&\equiv wp(x := a + b, x > a \wedge wp(C, x * y > 0)) \\
&\equiv a + b > a \wedge ((a + b \geq 0 \Rightarrow a + b * 1 > 0) \\
&\wedge (a + b < 0 \Rightarrow a + b * -1 > 0))
\end{aligned}
$$

2.3.2 Weakest Liberal Precondition

If only successfully terminating states are important and errors are permissible or handled otherwise, a variation called weakest liberal preconditions (WLP) can be used. WLPs treat assert statements as assume statements, the construction is shown in Fig. 2.3.5.

Lemma 1.

$$wp(P, Q) \wedge wlp(P, R) \equiv wp(P, Q \wedge R)$$

Proof.

$$
\begin{aligned}
&P = x := e: && wp(S, Q) \wedge wlp(S, R) \equiv Q[x \to e] \wedge R[x \to e] \equiv (Q \wedge R)[x \to e] \\
&P = \textbf{assume } e: && wp(S, Q) \wedge wlp(S, R) \equiv (e \Rightarrow Q) \wedge (e \Rightarrow R) \equiv (e \Rightarrow (Q \wedge R)) \\
&P = \textbf{assert } e: && wp(S, Q) \wedge wlp(S, R) \equiv (e \wedge Q) \wedge (e \Rightarrow R) \equiv e \wedge P \wedge (\neg e \vee R) \\
& && \equiv e \wedge Q \wedge R \equiv wp(S, Q \wedge R) \\
&P = S \square T: && wp(P, Q) \wedge wlp(P, R) \\
& && \equiv wp(S, Q) \wedge wp(T, Q) \wedge wlp(S, R) \wedge wlp(T, R) \\
& && \equiv wp(S, Q) \wedge wlp(S, R) \wedge wp(T, Q) \wedge wlp(T, R) \\
& && \equiv wp(S, Q \wedge R) \wedge wp(T, Q \wedge R) && \text{IH} \\
& && \equiv wp(P, Q \wedge R) \\
&P = S; T: && wp(P, Q) \wedge wlp(P, R) \equiv wp(S, wp(T, Q)) \wedge wlp(S, wlp(T, R)) \\
& && \equiv wp(S, wp(T, Q) \wedge wlp(T, R)) && \text{IH} \\
& && \equiv wp(S, wp(T, Q \wedge R)) && \text{IH} \\
& && \equiv wp(P, Q \wedge R))
\end{aligned}
$$

□

Figure 2.3.6: The translation from GCL to weakest existential precondition

P	$wep(P, Q)$
$x := e$	$Q[x \to e]$
assert e	$e \wedge Q$
assume e	$e \wedge Q$
$S\,;T$	$wep(S, wep(T, Q))$
$S \,\square\, T$	$wep(S, Q) \vee wep(T, Q)$

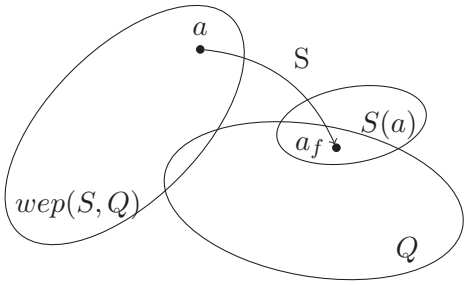


Figure 2.3.7: The relationship between weakest existential preconditions and program execution

An important connection between WLP and WP is the following equivalence.

Theorem 1.

$$wp(S, Q) \equiv wp(S, true) \wedge wlp(S, Q)$$

Proof. Follows from Lem. 1 with $R = false$. □

$wp(S, true)$ is simply the condition that S not go wrong (every state satisfies $true$) while $wlp(S, Q)$ is satisfied when all terminating states satisfy Q. Thus if both are satisfied, the program does not go wrong and terminates only in Q states.

2.3.3 Weakest Existential Precondition

The weakest existential precondition is defined in Fig. 2.3.6. This predicate transformer, instead of ensuring Q for every execution path, describes whether Q will be true at the end of at least one of the possible paths.

The essence of the weakest existential precondition is the following:

$$\text{If } a \models wep(S, Q) \text{ then } a' \models Q, \text{ for some } a' \in S(a)$$

Thus Q is reachable from a through execution of S as shown in Fig. 2.3.7.

Because *some* path through the program *must* lead to a state in Q, the WEP for each path in a choice statement is put into a disjunction. For a state to satisfy the WEP, it *must* end in that state and thus some path must not go wrong or block, hence the conjunction for both assume and assert statements.

We can examine an example calculation of a WEP for the program in Fig. 2.3.4:

$$
\begin{aligned}
wep(S, x \leq 0) &\equiv wep(x := a + b; \textbf{assert } x > a; C; x := x * y, x \leq 0) \\
&\equiv wep(x := a + b; \textbf{assert } x > a; C; wep(x := x * y, x * y \leq 0)) \\
&\equiv wep(x := a + b; \textbf{assert } x > a; C, x * y \leq 0) \\
&\equiv wep(x := a + b; \textbf{assert } x > a, wep(C, x * y \leq 0)) \\
wep(C, x * y \leq 0) &\equiv wep(C_1, x * y \leq 0) \vee wep(C_2, x * y \leq 0) \\
wep(C_1, x * y \leq 0) &\equiv wep(\textbf{assume } x >= 0; y := 1, x * y \leq 0) \\
&\equiv wep(\textbf{assume } x >= 0, wep(y := 1, x * y \leq 0)) \\
&\equiv wep(\textbf{assume } x >= 0, x * 1 \leq 0) \\
&\equiv x \geq 0 \wedge x * 1 \leq 0 \\
wep(C_2, x * y \leq 0) &\equiv wep(\textbf{assume } x < 0; y := -1, x * y \leq 0) \\
&\equiv wep(\textbf{assume } x < 0, wep(y := -1, x * y \leq 0)) \\
&\equiv wep(\textbf{assume } x < 0, x * -1 \leq 0) \\
&\equiv x < 0 \wedge x * -1 \leq 0 \\
wep(C, x * y \leq 0) &\equiv (x \geq 0 \wedge x * 1 \leq 0) \vee (x < 0 \wedge x * -1 \leq 0) \\
wep(S, x \leq 0) &\equiv wep(x := a + b; \textbf{assert } x > a, wep(C, x * y \leq 0) \\
&\equiv wep(x := a + b, wep(\textbf{assert } x > a, wep(C, x * y \leq 0)) \\
&\equiv wep(x := a + b, x > a \wedge wep(C, x * y \leq 0)) \\
&\equiv a + b > a \wedge ((a + b \geq 0 \wedge a + b * 1 \leq 0) \\
&\quad \vee (a + b < 0 \wedge a + b * -1 \leq 0))
\end{aligned}
$$

Theorem 2.

$$wep(P, Q) \equiv \neg wlp(P, \neg Q)$$

Proof.

$P = x := e$: $wep(P, Q) \equiv Q[x := e] \equiv \neg\neg Q[x := e] \equiv \neg wlp(P, \neg Q)$

$P = \textbf{assume } e$:

$P = \textbf{assert } e$: $wep(P, Q) \equiv E \wedge Q \equiv \neg\neg E \wedge \neg\neg Q \equiv \neg(\neg E \vee \neg Q)$
$\equiv \neg(E \Rightarrow \neg Q) \equiv \neg wlp(P, \neg Q)$

$P = S\,;T$: $wep(P, Q) \equiv wep(S, wep(T, Q))$
$\equiv wep(S, \neg wlp(T, \neg Q))$ IH
$\equiv \neg wlp(S, \neg\neg wlp(T, \neg Q))$ IH
$\equiv \neg wlp(S, wlp(T, \neg Q)) \equiv \neg wlp(P, \neg Q)$

$P = S \square T$: $wep(P, Q) \equiv wep(S, Q) \vee wep(T, Q)$
$\equiv \neg wlp(S, \neg Q) \vee \neg wlp(T, \neg Q)$ IH
$\equiv \neg(wlp(S, \neg Q) \wedge wlp(T, \neg Q)) \equiv \neg wlp(P, \neg Q)$

□

Figure 2.3.8: The translation from GCL to strongest postcondition

S	$sp(S, P)$
$x := e$	$\exists v.(x = e[v \to x]) \wedge P[v \to x]$
assert e	$e \wedge P$
assume e	$e \Rightarrow P$
$S\,;T$	$sp(T, sp(S, P))$
$S \,\square\, T$	$sp(S, P) \vee sp(T, P)$

Th. 2 shows that the two transformers WLP and WEP can be described as dual to one another in the sense in [BW89]. The duality is the connection between miraculous statements and non-terminating statements as well as the connection between demonic and *angelic* determinism, so called because the safe path will be chosen if there is one. To be precise, the state a which satisfies $wlp(S, false)$ because $S(a)$ is empty, does not satisfy the *dual* verification condition, $wep(S, true)$, because it goes wrong. Additionally, the demonic non-determinism of choice enforced by the conjunction in $wlp(S, false)$ is instead enforced as angelic by the disjunction in $wep(S, true)$.

2.3.4 Strongest Postcondition

The final predicate transformer is the strongest postcondition, pictured in Fig. 2.3.8. Given a program and a formula, the strongest postcondition is the formula most strongly characterizing a set of states after executing the program with those states. The strongest postcondition conforms to the following [BM07]:

$$\text{If } a \models sp(S, P) \text{ then } a_0 \text{ exists such that } a_0 \models P, \text{ and } a \in S(a_0)$$

Thus $sp(S, P)$ is reachable from P through execution of S as shown in Fig. 2.3.9.

The connection between strongest postcondition and weakest precondition is expressed through the following property which allows the SP to be used to verify a Hoare triple as well as the WP [BM07]:

Proposition 1.

$$P \Rightarrow wp(S, Q) \Leftrightarrow sp(S, P) \Rightarrow Q \tag{2.1}$$

Proof. The property can be shown by structural induction on the individual statement translations. □

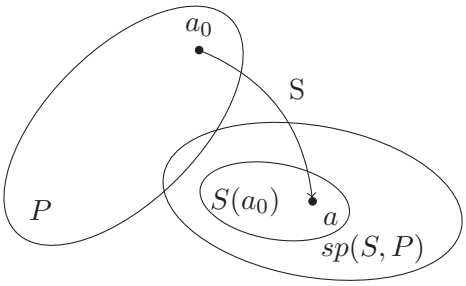


Figure 2.3.9: The relationship between strongest postconditions and program execution [BM07]

3 Efficient Predicate Transformers

When verifying real programs a serious drawback with the resulting formulas can quickly make itself known [FS01]. Consider the isolated choice statement in Fig. 2.3.4. When calculating the weakest precondition, we calculate $wp(C_1, Q) \wedge wp(C_2, Q)$. Notice that the postcondition is now represented twice in the weakest precondition. Because the postcondition is doubled, the size of the weakest precondition is now at least twice as long as the original postcondition. In this case, the effect is not significant but consider that a WP resulting from such a choice statement might be passed again as postcondition because it lies in a sequence. With each choice statement causing a duplication, the resulting formula size is exponential in the number of choice statements.

Generating the WP for the first statement with the choice's WP as the condition, we finally get $(wp(C_1, Q) \wedge wp(C_2, Q))[x \to a + b]$ for the entire program. The assignment is translated as substitutions for x because by changing the state of the program, the assignment statement invalidates the previous uses of x in the WP.

By replacing x with its new value, $a + b$, the size of this assignment statement has an effect on both conjuncts and has increased the total size by twice what it would for a single path. This effect will occur with any sequence of statements which reference previous variables. The more references and the more assignments throughout the program, the more duplication occurs and this too can grow exponentially [FS01].

In the worst case, these two effects compound each other and chains of assignment statements are replicated into each branch of a choice statement, which perhaps more choice statements follow. Because the exponential explosion can be so drastic, it can potentially prevent a weakest precondition from ever being constructed [FS01].

The root cause of the blowup is this need to duplicate and alter the postcondition due to choices and assignments. What is essentially happening is that as the WP algorithm moves backwards from the last statements, the history of changes to the program state must be constantly replicated into postconditions. Each path of a choice statement must therefore alter a unique copy of the postcondition. In order to prevent exponential explosion, we need to avoid these duplications and alterations of the predicate. For that to be possible, we must ensure some additional properties of our program.

Consider Fig. 2.3.4 again. As we generate $wp(x := x * y, x > 0)$, we need to introduce new information, namely that $x := x * y$. The classic WP algorithm does this by substituting the new value for x into the predicate. Upon encountering a choice statement, two weakest preconditions are calculated separately for each path, with separate copies of the postcondition, so that updates in those paths are made on their own versions of the predicate. These updates to the state come solely through assignments.

However, another way to enforce updates like this in logic is given in Lem. 2.

Lemma 2. $Q[x \to e] \equiv \forall x'.\, (x' = e) \Rightarrow Q[x \to x']$ if $x' \notin e$ and x', e not bound in Q.

Proof. $\Leftarrow$ Let a model M for the right-hand side exist. The semantics of the universal qualifier dictate that if we set the free variable x' in $(x' = e) \Rightarrow Q$ such that it is equal to e, then Q must

be true. Because the variable x' does not occur in e, substituting e in for x' gives an equivalent formula, satisfying the left-hand side.

$\Rightarrow$ Let a model M not satisfying the right-hand side exist. The semantics of the universal qualifier dictate that there exists a value x' such that $(x' = e) \Rightarrow Q[x \to x']$ is false. Because of the transitivity of equality, then we can set x' to be e and the formula is also false, thus the right-hand side is false. We can substitute e for x' in $Q[x \to x']$ to get an equivalent formula and so the left-hand side is also not satisfied. □

Trying to utilize this equivalence without minding the conditions on x' leads to problems (e.g. using $\forall x.\,(x = x * y) \Rightarrow x > 0$ in place of $x > 0[x \to x * y]$). Such a subformula does not maintain the desired semantics. In this specific case, y must be equal to 0 in any satisfying model. The problem is that x occurs already in the right side of the newly introduced equality. For an assignment which assigns a new value to an existing variable, this equality will break the intended translation. But the approach is encouraging, because it allows the postcondition to maintain its size. To avoid collisions we can refer to the new value by a new name x_1, called an *intermediate* variable, and then replace, without blowing up the length, x with x_1 in the predicate. It is important that occurrences of x be replaced with x_1, because the post condition refers to the value of the variable after making the assignment, and thus it must use the new value x_1. So as a tentative new rule for assignments we have:

$$\forall x_1.\,(x_1 = e) \Rightarrow Q[x \to x_1]$$

in place of $Q[x \to e]$. For this specific assignment the rule gives:

$$wp(x := x * y, x > 0) = \forall x_1.\,(x_1 = x * y) \Rightarrow x_1 > 0$$

This new rule has the form $e \Rightarrow Q$, the same one created by translating an assume statement into a weakest precondition. Semantically, assignment statements can thus be viewed as statements that introduce a very specific type of assumption into the rest of the program.

Though assume also allows introducing this type of assumption, one must be careful, because each assignment must receive its own variable, lest there be collisions. It makes no sense to assume two potentially contradictory values for a variable. This is the fundamental requirement of this approach.

This worked easily enough in the case of a single statement but it is not immediately obvious how to apply it to the general case, especially given a choice statement. We can first transform the program to use solely intermediate variables.

As a first attempt for choice statements, we could rewrite the choice from Fig. 2.3.4 as pictured in Fig. 3.0.1, creating a new variable for each assignment. If we wish to calculate $wp(C, y > 0)$, then we will need to refer to the renamed variable in $y > 0$ and replace every occurrence of y in the predicate with the renamed variable. If we choose either y_1 or y_2 though, the changes of the other path are lost. Choosing y_1 to get $wp(C, y_1 > 0)$ for instance, the postcondition no longer considers the second path.

3.1 Dynamic Static Assignment Form

In order to use the new assignment rule for all programs, we must be able to transform all types of statements to use intermediate variables. We begin by considering a similar form used

Figure 3.0.1: Simple choice rewrite attempt

```
  assume x >= 0;
  y_1 := 1
 []
  assume x < 0;
  y_2 := -1
```

Figure 3.1.1: Choice with phi instruction

```
  assume x >= 0;
  y_1 := 1
 []
  assume x < 0;
  y_2 := -1
y_3 := phi((C_1, 1), (C_2, -1))
```

in compiler construction called *single static assignment* form (SSA). All variables in an SSA program are assigned exactly one time.

As we saw in Fig. 3.1.2 however, this does not conform with branching constructs. In different executions of the program, different branches might be taken. The semantics require that there must be a way to unify these different values under a single name for use after a branch.

The solution for SSA programs is a construct called a *phi* instruction. Given that which branch is taken depends on runtime information, there is no way to statically know which will be chosen. The *phi* instruction is inserted after the branch and simply represents the value of the variable after some branch. The SSA structure of the program is thus maintained [AWZ88]. Specifically, for each variable whose value has been changed in some branch, a new variable is created at the point where branches reconverge and assigned the value of the phi instruction.

Often used in compilers for analysis and code transformations, SSA is an intermediate form and is not executed as such. The *phi* instruction is therefore more of a formality. It is removed on translation to machine code and replaced with the equivalent imperative assignments.

If we were to put our GCL program into SSA form, we would then want to complete the weakest precondition translation. Unfortunately the semantics of the phi instruction can not easily be expressed in a logical formula.

Therefore, the algorithm used for GCL programs differs slightly from SSA. Multiple definitions of the same variable are actually allowed provided that they occur "parallel" or in separate paths of a choice statement. For each variable altered in a choice statement, we define a new intermediate variable at the end of *every* branch of the choice statement. This variable points simply to the most recent use of the variable in each branch. This is shown in Alg. 1. With this change, uses of the variable after the choice statement refer to the correct value of the variable no matter which path was taken. The resulting transformation is referred to as *Dynamic Static Assignment* (DSA) form.

Definition 9. Variable Map

Variable maps are functions $\sigma : Vars \to Vars$ mapping variables to variables.

$$\begin{aligned} values(\sigma) &= \{\sigma(x) \mid \sigma(x) \neq x\} \\ keys(\sigma) &= \{x \mid \sigma(x) \neq x\} \end{aligned}$$

Algorithm 1 Branch Merge

$$\begin{aligned} merge(\sigma_S, \sigma_T, \sigma') &= (P_S, P_T, \sigma') \\ \text{where } D &= \{x \in keys(\sigma_S) \cup keys(\sigma_T) \mid \sigma_S(x) \neq \sigma_T(x)\} \\ \sigma_D &= \{x \to x_C \mid x \in D \wedge x_C \text{ is a new variable}\} \\ P_S &= \{\textbf{assume } x_C = \sigma_S(x); \mid x \in keys(\sigma_D)\} \\ P_T &= \{\textbf{assume } x_C = \sigma_T(x); \mid x \in keys(\sigma_D)\} \\ \sigma' &= \{x \to \sigma_D(x) \mid x \in keys(\sigma_D)\} \cup \{x \to \sigma_S(x) \mid x \notin keys(\sigma_D)\} \end{aligned}$$

Algorithm 2 DSA

S	$dsa(P, \sigma)$
assert e	$(\textbf{assert } \sigma(e), \sigma)$
assume e	$(\textbf{assume } \sigma(e), \sigma)$
$x := e$	$(\textbf{assume } x_{i+1} = \sigma(e), \sigma[x \to x_{i+1}])$ where x_i is the current substitution for x
$S \,\square\, T$	$((S'; P_S) \,\square\, (T'; P_T), \sigma')$ where $(S', \sigma_S) = dsa(S, \sigma)$, $(T', \sigma_T) = dsa(T, \sigma)$, $(P_S, P_T, \sigma') = merge(\sigma_S, \sigma_T)$
$S\,;T$	$(S'; T', \sigma'')$ where $(S', \sigma') = dsa(S, \sigma)$, $(T', \sigma'') = dsa(T, \sigma')$

Lemma 3. A bijection exists between programs and their DSA versions for executions, i.e. between $(P, \omega) \Downarrow (\omega', s)$ and $(P_p, \omega) \Downarrow (\omega'^p, s)$ where $(P_p, \sigma) = dsa(P, [x \to x])$. Additionally, $\sigma(\omega')(x) = \omega'^p(x)$ and $\sigma^{-1}(\omega'^p)(x) = \omega'(x)$ for all $x \in values(\sigma)$.

Proof. The bijection and correspondence between final states can be proven by induction over the semantic execution operation (cf. Fig. 2.2.2). □

The connection between the VC of a program and the VC of its DSA version is formalized in the following proposition:

Lemma 4. $wp(S, Q) \equiv wp(S_p, \sigma(Q))$ where $(S_p, \sigma) = dsa(S)$ and Q does not refer to variables $\in vars(S_p) \setminus vars(S)$. The same applies to *wlp* and *wep*.

Proof. Let state $\omega \models wp(S, Q)$. For all states ω' with $(S, \omega) \Downarrow (\omega', s)$, $\omega' \models Q$. Thus, $\sigma(\omega') \models \sigma(Q)$ because Q contains no intermediate variables. Because there is a bijection between derivations, then equivalently, for all states ω'^p with $(S_p, \omega) \Downarrow (\omega'^p, s)$, $\omega'^p \models \sigma(Q)$ because $\sigma(Q)$ contains no variables without intermediate variables. This is equivalent to $\omega \models wp(S_p, \sigma(Q))$. □

Figure 3.1.2: Second rewrite attempt

```
  assume x >= 0;
  y_1 := 1;
  y_f := y_1
 []
  assume x < 0;
  y_2 := -1;
  y_f := y_2
x_1 := x * y_f
```

Now consider the example program rewritten in this way (Fig. 3.1.2). We introduce another new variable y_f and again modify the GCL to give it the value of y_1 in the first path and y_2 in the second and use only y_f after the choice statement.

Now we can calculate $wp(S', x > 0)$ using the new assignment rule.

$$
\begin{aligned}
wp(S', x > 0) &\equiv wp(C', wp(x_1 := x * y_f, x > 0)) \\
&\equiv wp(C', \forall x_1.\, (x_1 = x * y) \Rightarrow x_1 > 0)) \\
\varphi &= \forall x_1.(x_1 = x * y_f) \Rightarrow (x_1 > 0) \\
wp(C', \varphi) &\equiv wp(C'_1, \varphi) \wedge wp(C'_2, \varphi) \\
wp(C'_2, \varphi) &\equiv wp(\textbf{assume } x >= 0; \textbf{assume } y_1 = 1; \textbf{assume } y_f = y_1, \varphi) \\
&\quad wp(\textbf{assume } x >= 0; \textbf{assume } y_1 = 1, \forall y_f.\, (y_f = y_1) \Rightarrow \varphi) \\
&\quad wp(\textbf{assume } x >= 0, \forall y_1.\, (y_1 = 1) \Rightarrow \forall y_f.\, (y_f = y_1) \Rightarrow \varphi) \\
&\quad \forall y_1 y_f.\, ((x \geq 0) \Rightarrow (y_1 = 1) \Rightarrow (y_f = y_1) \Rightarrow \varphi) \\
wp(C'_1, \varphi) &\equiv wp(\textbf{assume } x < 0; \textbf{assume } y_2 = -1; \textbf{assume } y_f = y_2, \varphi) \\
&\quad wp(\textbf{assume } x < 0; \textbf{assume } y_2 = -1, (y_f = y_2) \Rightarrow \varphi) \\
&\quad wp(\textbf{assume } x < 0, (y_2 = -1) \Rightarrow (y_f = y_2) \Rightarrow \varphi) \\
&\quad \forall y_2 y_f.\, ((x < 0) \Rightarrow (y_2 = -1) \Rightarrow (y_f = y_2) \Rightarrow \varphi) \\
wp(C', \varphi) &\equiv ((x \geq 0) \Rightarrow (y_1 = 1) \Rightarrow (y_f = y_1) \Rightarrow \varphi) \\
&\quad \wedge ((x < 0) \Rightarrow (y_2 = -1) \Rightarrow (y_f = y_2) \Rightarrow \varphi) \\
&\equiv ((x \geq 0) \Rightarrow (y_1 = 1) \Rightarrow (y_f = y_1)) \\
&\quad \wedge ((x < 0) \Rightarrow (y_2 = -1) \Rightarrow (y_f = y_2)) \Rightarrow \varphi
\end{aligned}
\tag{3.1}
$$

3.2 Passive Weakest Preconditions

Looking again at Eq. (3.1), we have two *identical* copies of the postcondition and have maintained the WP semantics. By putting a program in DSA form the program always satisfies an interesting property. When a DSA program satisfies $wlp(S, Q)$, either it goes wrong or blocks, or the postcondition is true unchanged [RL05]. This is formalized as an equivalence with wlp in Th. 3. First, we treat assignments in the DSA program as assume statements. A DSA program with assignments replaced with assume is called *passified*. The WLP $wlp(S, Q)$ can then be calculated

without duplicating or otherwise altering Q. Second, all the universal quantifiers introduced through the new assignment rule can be pulled to the front of the formula.

Theorem 3. WLP for DSA programs

$$wlp(S,Q) \equiv \forall \overline{x}.\, wlp(S_p, false) \vee Q$$

$$\text{where } \begin{array}{ll} S_p & = S \text{ with } x := e \text{ replaced with } \textbf{assume } x = e \\ \overline{x} & = \{x \mid (x := e) \in S\} \end{array}$$

Proof.

$P = \textbf{assume } e$

$P = \textbf{assert } e:$ $\quad wlp(P,Q) \equiv (e \Rightarrow Q) \equiv (\neg e \vee Q) \equiv wlp(P, false) \vee Q$

$P = x := e:$

$$\begin{aligned} & wlp(P,Q) \equiv Q[x \to e] \equiv \forall x.\, (x = e) \Rightarrow Q && \text{Lem. 2} \\ & \equiv \forall x.\, (x \neq e) \vee Q \equiv \forall x.\, (x \neq e \vee false) \vee Q \\ & \equiv \forall x.\, wlp(\textbf{assume } x = e, false) \vee Q \end{aligned}$$

$P = S\,;T:$

$$\begin{aligned} & wlp(P,Q) \equiv wlp(S;T,Q) \\ & \equiv wlp(S, wlp(T,Q)) \\ & \equiv wlp(S, \forall \overline{x}_T.\, wlp(T_p, false) \vee Q) && \text{IH} \\ & \equiv \forall \overline{x}_S.\, (wlp(S_p, false) \vee \forall \overline{x}_T.\, (wlp(T_p, false) \vee Q)) && \text{IH} \\ & \equiv \forall \overline{x}_S.\, \forall \overline{x}_T.\, (wlp(S_p, false) \\ & \qquad\qquad \vee (wlp(T_p, false) \vee Q)) && \overline{x}_T \text{ not in } wlp(S, false) \\ & \equiv \forall \overline{x}_T.\, \forall \overline{x}_S.\, ((wlp(S_p, false) \vee wlp(T_p, false)) \vee Q) \\ & \equiv \forall \overline{x}_T.\, \forall \overline{x}_S.\, (wlp(S_p, wlp(T_p, false)) \vee Q) && \text{IH} : S \text{ and } T \text{ already passified} \\ & \equiv \forall \overline{x}_T.\, \forall \overline{x}_S.\, (wlp(S_p; T_p, false) \vee Q) \end{aligned}$$

$P = S \,\square\, T:$

$$\begin{aligned} & wlp(P,Q) \equiv wlp(S,Q) \wedge wlp(T,Q) \\ & \equiv \forall \overline{x}_S.\, (wlp(S_p, false) \vee Q) \wedge \forall \overline{x}_T.\, (wlp(T, false) \vee Q) && \text{IH} \\ & \equiv \forall \overline{x}_{ST}.\, (wlp(S_p, false) \wedge wlp(T_p, false)) \vee Q && \forall \text{ distributes over } \wedge \\ & \text{where } \overline{x}_{ST} = x \in \overline{x}_S \cup \overline{x}_T \\ & \equiv \forall \overline{x}_{ST}.\, wlp(P_p, false) \vee Q \end{aligned}$$

□

Theorem 4. WP for DSA programs

$$wp(P,Q) = \forall \overline{x}.\, wp(P_p, true) \wedge (wlp(P_p, false) \vee Q)$$

Proof. From Th. 1, we know $wp(P,Q) = wp(P, true) \wedge wlp(P,Q)$.

$P = \textbf{assume } e:$ $\quad wp(P,Q) \equiv (e \Rightarrow true) \equiv (\neg e \vee true) \equiv true$

Algorithm 3 $wlp(P, false)$ for passive programs

P	$wlp(P, false)$
assert e	$\neg e$
assume e	$\neg e$
$S \square T$	$wlp(S, false) \wedge wlp(T, false)$
$S\,;T$	$wlp(S, false) \vee wlp(T, false)$

Proof. Th. 3 with $Q = false$. □

$P = \textbf{assert}\ e$:
$wp(P, Q) \equiv (e \wedge true) \equiv (\neg e \wedge true) \equiv \neg e$

$P = x := e$:
$wp(P, Q) \equiv wp(P, true) \wedge wlp(P, Q)$
$\equiv true \wedge \forall \overline{x}.\, wlp(P_p, false) \vee Q$
$\equiv (x \neq e \vee true) \wedge \forall \overline{x}.\, wlp(P_p, false) \vee Q$
$\equiv (x = e \Rightarrow true) \wedge \forall \overline{x}.\, wlp(P_p, false) \vee Q$
$\equiv wp(\textbf{assume}\ x = e, true) \wedge \forall \overline{x}.\, wlp(P_p, false) \vee Q$
$\equiv \forall \overline{x}.\, wp(P_p, true) \wedge wlp(P_p, false) \vee Q$

$P = S \square T$:
$wp(P, Q) \equiv wp(P, true) \wedge wlp(P, Q)$
$\equiv wp(P, true) \wedge \forall \overline{x}_{ST}.\, wlp(P_p, false) \vee Q$ — Th. 3
$\equiv wp(S, true) \wedge wp(T, true) \wedge \forall \overline{x}_{ST}.\, wlp(P_p, false) \vee Q$
$\equiv \forall \overline{x}_S.\, wp(S_p, true) \wedge true \wedge \forall \overline{x}_T.\, wp(T_p, true) \wedge true$ — IH
$\wedge \forall \overline{x}_{ST}.\, wlp(P_p, false) \vee Q$
$\equiv \forall \overline{x}_{ST}.\, (wp(S_p, true) \wedge wp(T_p, true))$
$\wedge \forall \overline{x}_{ST}.\, (wlp(P_p, false) \vee Q)$
$\equiv \forall \overline{x}_{ST}.\, wp(P_p, true) \wedge (wlp(P_p, false) \vee Q)$

$P = S\,;T$:
$wp(P, Q) \equiv wp(P, true) \wedge wlp(P, Q)$
$\equiv wp(P, true) \wedge \forall \overline{x}_{ST}.\, wlp(P_p, false) \vee Q$ — Th. 3
$\equiv \forall \overline{x}_S.\, wp(S_p, true) \wedge (wlp(S_p, false) \vee wp(T, true))$ — IH with S
$\wedge \forall \overline{x}_{ST}.\, wlp(P_p, false) \vee Q$
$\equiv \forall \overline{x}_S.\, wp(S_p, true) \wedge (wlp(S_p, false) \vee \forall \overline{x}_T.\, wp(T_p, true))$ — IH with T
$\wedge \forall \overline{x}_{ST}.\, wlp(P_p, false) \vee Q$
$\equiv \forall \overline{x}_{ST}.\, wp(S_p, true) \wedge (wlp(S_p, false) \vee wp(T_p, true))$ — *
$\wedge \forall \overline{x}_{ST}.\, wlp(P_p, false) \vee Q$
$\equiv \forall \overline{x}_{ST}.\, wp(S_p, wp(T_p, true)) \wedge wlp(P_p, false) \vee Q$ — IH: S and T already passified
$\equiv \forall \overline{x}_{ST}.\, wp(S_p\,;T_p, true) \wedge (wlp(P_p, false) \vee Q)$
$\equiv \forall \overline{x}_{ST}.\, wp(P_p, true) \wedge (wlp(P_p, false) \vee Q)$

□

Algorithm 4 $wp(P, true)$ for passive programs

P	$wp(P, true)$
assert e	e
assume e	$true$
$S \square T$	$wp(S, true) \wedge wp(T, true)$
$S\,;T$	$wp(S, true) \wedge (wlp(S, false) \vee wp(T, true))$

Proof. Th. 4 with $Q = true$. For sequence, see Eq. (*) in the proof of Th. 4. □

3.2.1 Efficient Predicate Transformers

We now have the two main constituents of an algorithm for generating weakest preconditions to avoid exponential growth. The algorithm works in two steps, first translating a program into DSA and then passive form. Subsequently, we use the new WP algorithm to generate a verification condition. Because every intermediate variable is new, we can always extract the universal quantifiers from subformulas and bring them to the front. We also use the substitution returned by *dsa* on the postcondition so that all references to variables are made to their last intermediate version. We call the new algorithms efficient predicate transformers.

WEP has not been discussed at all in the context of passified programs. One could go through the previous proofs with an equivalence similar to Th. 3, namely that

$$\forall Q.wep(S, Q) \equiv \neg wlp(S, false) \wedge Q$$

Informally, if some path to Q exists, then the program does not always block or go wrong and Q is true. However, with Th. 2, we can simplify the calculation by negating the postcondition and then calculating and negating the WLP on that postcondition. The final algorithm is pictured in Alg. 5. To note specifically the new efficient predicate transformers, we prefix their name with e, e.g. *ewep*.

The goal of our algorithm was to have a new function, $ewp(P, Q)$ which accepts a program and a condition and returns a formula *equivalent* to $wp(P, Q)$. In the verification framework, weakest preconditions are generated and integrated into larger formulas and decisions. The verification framework uses an SMT solver to decide satisfiability of formulas. Therefore, any problem we decide about these VCs must ultimately be in terms of satisfiability.

The basic WP algorithm produces formulas without quantifiers. All program variables are represented in the formula as free variables. No other free variables are present. In the course of passifying the program new variables are introduced representing values intermediate between initial and final values. When generating a weakest precondition on the resulting passified program, these intermediate variables naturally show up in the final logical formulas as quantifier-bound variables. As discussed in section 2, deciding satisfiability with quantifiers adds considerable complexity to the decision process.

If we simply ignore the quantifiers, the newly introduced variables are thus free in our formula. Any model of the formula must include an assignment for them. For the sake of simplicity we can consider a WLP formula generated by the new algorithm given to the SMT solver in a check for satisfiability.

Algorithm 5 Efficient Predicate Transformers (EPT)

1: $(P, \sigma) \leftarrow dsa(S, [x \rightarrow x])$
2: $P_p \leftarrow passify(P)$
3: $\overline{x} \leftarrow values(\sigma)$
4: **if** EXISTENTIAL **then**
5: $\quad Q \leftarrow \neg Q$
6: **end if**
7: $wlp \leftarrow wlp(P_p, false) \vee \sigma(Q)$
8: **if** LIBERAL **then**
9: $\quad$ **return** $\forall \overline{x}.\, wlp$
10: **else**
11: $\quad wp \leftarrow \forall \overline{x}.\, wp(P_p, true) \wedge wlp$
12: **end if**
13: **if** EXISTENTIAL **then**
14: $\quad$ **return** $\neg wlp$
15: **else**
16: $\quad$ **return** wlp
17: **end if**

Figure 3.2.1: Program guaranteed to fail

```
x = y + 1;
assert x = y
```

Recall these have the form $wlp(S, false) \vee Q$. Now within the $wlp(S, false)$ formula, which represents the formula either blocking or going wrong, there are potentially expressions such as $(x_i \neq val)$ where x_i is free.

$$wlp(S, false) = \cdots \vee (x_i \neq val) \vee \ldots$$

If our SMT solver assigns x_i a value not equal to val, this subexpression evaluates to true. Within $wlp(S, false) \vee Q$, the left-hand side is true and the entire formula evaluates to true, satisfying the formula. Our SMT solver can therefore satisfy the formula by satisfying such inequalities. What is happening is that the assumes transformed from assignments are not being satisfied, a situation semantically equal to blocking execution. This then allows the WLP to be satisfied. This may not cause problems for a VC which is otherwise satisfiable but consider the GCL program in Fig. 3.2.1. The passified program is in Fig. 3.2.2. The efficient $wlp(S, true)$ is:

$$x_1 = y + 1 \Rightarrow x_1 = y$$

and is obviously satisfiable. The classic algorithm however, gives:

$$y + 1 = y$$

which is of course unsatisfiable.

The problem stems from the fact that x_1 is no longer constrained by the universal quantifier. The intention of introducing it was to simply use it as a predetermined value. In fact, it

Figure 3.2.2: Program guaranteed to fail passified

```
assume x_1 = y + 1;
assert x_1 = y
```

should not even be free at all. Ideally though, efficient verification conditions should be expressible in quantifier-free logic. Keeping efficient VCs quantifier-free requires treating the cases of satisfiability and validity separately.

3.2.2 Validity

For efficient predicate transformers we receive a VC of the following form:

$$\forall \overline{x}.\, \varphi_{wp} \text{ where } \varphi_{wp} \text{ is quantifier-free}$$

The SMT solver can only decide problems of satisfiability and thus in order to check for validity we negate the formula and check for satisfiability.

$$\neg \forall \overline{x}.\, \varphi_{wp} \equiv \exists \overline{x}.\, \neg \varphi_{wp}$$

This formula is in a class of formulas called Σ_1 because it is a quantifier-free formula prefixed only by existential quantifiers.

Definition 10. Skolemization
For a formula $\varphi \in \Sigma_1$, skolemization introduces a new formula which is satisfiable if and only if φ is (equi-satisfiable). This is done by replacing each existentially quantified variable with a constant [KS08].

Proposition 2. $\exists \overline{x}.\, \neg \varphi_{wp}$ and $\neg \varphi_{wp}$ can be made equi-satisfiable through skolemization.

The SMT solver can now handle this equi-satisfiable formula and give us an answer to our original query. The model will have additional variable assignments but they can be ignored.

For weakest existential preconditions the situation is reversed, due to the equivalence in Th. 2. That is, deciding satisfiability of a WEP can be done with a Σ_1 formula, because it is already the negation of a WLP formula.

3.2.3 Satisfiability

When checking for satisfiability, our formula maintains the form:

$$\forall \overline{x}.\varphi_{wp}$$

We would need a method for transforming this formula into one with only free variables that is equi-satisfiable. In the original formulation for efficient predicate transformers, assign statements are transformed into assumes. Assumes are allowed to be false by weakest preconditions but this is not a problem because when checking for validity, we are also verifying those states where the assumes are true. For satisfiability on the other hand, any state that blocks satisfies the WP, thus assume statements will not suffice.

In the spirit of Lem. 2, we can attempt to use another formula to create quantifier-free WPs for satisfiability by beginning with existentially quantified formulas for assignments.

Figure 3.2.3: Program causing issues with WP satisfiability

```
x_1 := 1 [] x_1 := −1
```

Lemma 5. $\forall x'.(x' = e) \Rightarrow Q \equiv \exists x'.(x' = e) \wedge Q$

Proof. $\Rightarrow$ Let a model M for the left-hand side exist. The semantics of the universal qualifier dictate that if we set the free variable x in $(x = e) \Rightarrow Q$ such that it is equal to e, then this formula must be true. Model M must therefore also satisfy the right-hand side, because the value for x in the left-hand side exists and satisfies Q. M satisfies the right-hand side.

$\Leftarrow$ Let a model M exist which does not satisfy the left-hand side. The semantics of the universal qualifier dictate that if we set the free variable x in $(x = e) \Rightarrow Q$ such that it is equal to e, then this Q must be false. Therefore, no x with $x = e$ exists so that Q is satisfied because of the transitivity of equality. M does not satisfy the right-hand side. □

In analog to using assume for the implication in Lem. 2, we could translate $x := e$ as **assert** $x = e$ because the right hand side of the lemma has a conjunction. However, a problem arises upon encountering non-determinism. Consider the program in Fig. 3.2.3 and the calculation $wp(S, Q)$.

$$\begin{aligned} wp(S, Q) &\equiv wp(x_1 := 1, Q) \wedge wp(x_1 := -1, Q) \\ &\equiv \exists x_1.\,((x_1 = 1) \wedge Q) \wedge \exists x_1.\,((x_1 = -1) \wedge Q) \end{aligned}$$

If we set $Q = true$, the WP is satisfiable. Looking at the above formula, we get $\exists x_1.\,((x_1 = 1) \wedge true) \wedge \exists x_1.\,((x_1 = -1) \wedge true)$. Existential quantifiers can not in general be distributed over conjunction. If we try to pull the existential quantifier out of the formula like with the universal quantifier we get the unsatisfiable formula: $\exists x_1.\,(x_1 = 1 \wedge x_1 = -1)$.

We could try a different transformation and put the equalities together in a disjunction: $\exists x_1.\,(x_1 = 1 \vee x_1 = -1) \wedge Q$. But if we calculate $wp(S, x_1 = 1)$, which is unsatisfiable, the new version returns $\exists x_1.\,((x_1 = 1 \vee x_1 = -1) \wedge x_1 = 1)$, which is satisfiable.

There appears to be no way translate the weakest precondition into an equivalent Σ_1 formula. The problem lies in the fact that using assert for assignments enforces the assignments in the path across the entire formula. This causes problems with real non-determinism.

We claim that this is not a problem, however. When checking a VC for satisfiability, the idea is usually to check for a *bad* postcondition Q. In this case, it is not typically useful to ask whether a state exists such that execution will *always* result in a bad Q but rather such that execution will *sometimes* result in Q. This is precisely the semantics provided by WEP, the dual of the WLP.

3.3 Complexity

In order to analyze the final complexity of the formulas resulting from the new algorithm, we need to first analyze the complexity of DSA.

Definition 11. Size of formulas and programs

$|\varphi|$ is defined inductively. All constants and variables have size 1 while operators have size 1 plus the sizes of their operands.

$|P|$ is also defined inductively:

$$\begin{aligned} |x := e| &= 2 + |e| \\ |\textbf{assume } e| &= 1 + |e| \\ |\textbf{assert } e| &= 1 + |e| \\ |S\,;T| &= 1 + |S| + |T| \\ |S \,\square\, T| &= 1 + |S| + |T| \end{aligned}$$

3.3.1 DSA

Proposition 3. $|dsa(P)| \in O(|P|^2)$

Proof. First of all, substituting a variable for another variable does not change the size of the substituted term.

The two suffixes returned from *merge* are in $O(vars)$ with $vars$ the number of vars in the substitutions, because each suffix has the same number of intermediate assignments as the number of variables with unequal values.

Next, it is not difficult to see that DSA atomic statements do not change from their size and DSA sequences are equal to exactly the sum of the sizes of their DSA substatements.

$$\begin{aligned} &P = \textbf{assert } e \\ &P = \textbf{assume } e: && |dsa(P,\sigma)| = |P| \in O(|P|^2) \\ &P = S\,;T: && |dsa(S\,;T,\sigma)| = 1 + |dsa(S,\sigma)| + |dsa(T,\sigma)| \in O(|P|^2) \end{aligned}$$

First, the worst-case blowup will be shown to be in $O(|P|^2)$. Consider any GCL statement S with size l. The increase in size from S to S_p is due only to assignments, thus a worst case statement must contain only choice, sequence and assignments. At a merge, new assignments created by converting the substatements to DSA do not affect the number of intermediate assignments added at the merge. That is, all assignments added when merging come from real assignments in the original program. Thus, the maximum number of variables appended to each choice while merging is clearly l.

Now assume simply that the number of choice statements in the program, and thus the number of merges, is in $O(l)$. Now although each substatement of the choice statement is smaller than l, assume still that for all choices in the program, the number of added variables in each merge is nonetheless $2l$. The number of merges being in $O(l)$, this brings the total increase in size to $O(l^2)$.

Now, we will demonstrate a worst case where the blowup is in fact in $\Theta(|P|^2)$. Consider the following program:

$$x_1 := e_1 \,\square (x_2 := e_2 \,\square (\dots (x_n := e_n) \dots))$$

Let s be the size of one assignment statement, let every expression be the same size for simplicity. Note that the size of the program is $l = (2 + s) * n + (n - 1)$.

The base choice statement has two assignments as substatements. Merging them causes two merge assignments to be created on each side, one for each variable. The final DSA size is therefore the original program plus the 4 extra merge assignments, $2s + 1 + 4 * 3 = 2s + 13$. Each parent choice statement has a size of 1 plus the size of the merged substatements. In order to merge the assignment from the left-hand assignment, one merging intermediate assignment must be added to the right side. All changed variables $(n - 1)$ in the right substatement must have corresponding assignments in the passified left-hand side as well, plus one for merging the left-hand side assignment.

The recurrence relation for passification of this statement is therefore:

$$T(n) = 1 + (s + 3n) + (T(n-1) + 3) = T(n-1) + 3n + 4 + s$$
$$T(2) = 2s + 13$$

which has the following sum as a solution:

$$\sum_{i=3}^{n}(3i + 4 + s) + 2s + 13 = 3\sum_{i=3}^{n} i + \sum_{i=3}^{n} s + \sum_{i=3}^{n} 4 + 2s + 13$$
$$= 3(\frac{n(n+1)}{2} - 3) + sn + 4(n-2) + 13 \in O(n^2) = O((sn)^2) \in O(l^2)$$

Proof.

$$T(2) = 7 = \Sigma_{i=3}^{2}(3 + i) + 7$$
$$T(n) = T(n-1) + n + 3 = (\Sigma_{i=3}^{n-1}(3 + i) + 7) + n + 3 = \Sigma_{i=3}^{n}(3 + i) + 7$$

□

□

As one can see however, depending on the size of the expressions, the increase in size due to passification may well be dwarfed by the size of the original GCL, and [FS01] report that it is usually linear.

3.3.2 Efficient Predicate Transformers

There are three components to the final formula returned by the efficient predicate transformers: $wlp(P, false), wp(P, true)$, and Q. Since the final formula is a concatenation of these, we can analyze them separately.

First, we see that Q does not change its size in relation to the program size.

Next, $wlp(P, false)$ is independent of $wp(P, true)$, but the latter uses the former for sequence statements (Alg. 4). We will first analyze $wlp(P, false)$.

Proposition 4. $wlp(P, false) \in O(|P|)$

Proof.

$P =$ **assert** e

$P =$ **assume** e : $\quad wlp(P, false) \equiv \neg e$

$\quad |\neg e| \in O(|P|)$

The compositional statements make only recursive calls to $wlp(P, false)$ and add a constant size each time:

$P = S \,\square\, T$: $\quad wlp(P, false) \equiv wlp(P, false) \wedge wlp(P, false)$

$\quad |wp(P, Q)$

$\quad = 1 + |wlp(S, false)| + |wlp(T, false)| \in O(|P|)$

$P = S\,;T$: $\quad wlp(P, false) \equiv wlp(P, false) \vee wlp(P, false)$

$\quad |wp(P, Q)|$

$\quad = 1 + |wlp(S, false)| + |wlp(T, false)| \in O(|P|)$

□

Proposition 5. $|wp(P, true)| \in O(|P|^2)$

Proof. First for the atomic statements and choice:

$P =$ **assert** e

$P =$ **assume** e : $\quad wp(P, true) \equiv e$

$\quad |e| \in O(|P|^2)$

$P = S \,\square\, T$: $\quad wp(P, true) \equiv wp(S, true) \wedge wp(T, true)$

$\quad |wp(P, Q)|$

$\quad = 1 + |wp(S, true)| + |wp(T, true)| \in O(|P|^2)$

The sequence statement is similar to the choice statement in the proof about DSA. The worst-case is in $O(|P|^2)$, because the maximum number of sequence statements is in $O(|P|)$ and each time $wp(S, true) \wedge (wlp(S, false) \vee wp(T, true))$ is calculated. Because $wlp(S, false)$ was shown to have size $O(|P|)$, the result is $O(|P|^2)$.
We will demonstrate a worst case where the final WP formula is in fact in $\Theta(n^2)$. Let s be the size of one assume statement, let every expression be the same size for simplicity. Consider the following passive program:

$$(\ldots((\textbf{assume}\ x_1 = e_1\,;\textbf{assume}\ x_2 = e_2)\,;\textbf{assume}\ x_3 = e_3)\,;\ldots(\textbf{assume}\ x_n = e_n))$$

The base sequence statement has two assumptions as substatements.

$$\begin{aligned} & wp(\textbf{assume } x_1 = e_1 \,;\textbf{assume } x_2 = e_2, true) \\ & \equiv wp(\textbf{assume } x_1 = e_1, true) \\ & \wedge (wlp(\textbf{assume } x_1 = e_1, false) \vee wp(\textbf{assume } x_2 = e_2, true)) \\ & \equiv x_1 = e_1 \wedge (x_1 \neq e_1 \vee x_2 = e_2) \\ |\varphi_2^{wlp}| &= 3s + 2 \end{aligned}$$

Now for a sequence statement above the base:

$$\begin{aligned} & wp(S_{i-1} \,;\textbf{assume } x_i = e_i, true) \\ & \equiv wp(S_{i-1}, true) \wedge (wlp(S_{i-1}, false) \vee wp(\textbf{assume } x_i = e_i, true)) \\ & \equiv wp(S_{i-1}, true) \wedge (wlp(S_{i-1}, false) \vee x_i = e_i)) \\ |\varphi_i^{wp}| &= 2 + |\varphi_{i-1}^{wp}| + |\varphi_{i-1}^{wlp}| + s \end{aligned}$$

Given that φ_i^{wp} was already shown to have a size in $O(|P|)$, this gives a recurrence relation similar to the one for the DSA transformation above.

$$\begin{aligned} T(n) &= 2 + s * (n-1) + T(n-1) + s = T(n-1) + sn + 2 \\ T(2) &= 3s + 2 \end{aligned}$$

Analog to the previous recurrence relation, this proves $T(n) \in \Theta(|P|^2)$. □

Combining these two propositions, we can see that the efficient weakest precondition of a program with size l has worst-case size in $O(l^4)$ while the efficient weakest liberal and existential preconditions have worst-case size $O(l^2)$.

Because for every WP calculation for a sequence $S\,;T$, the $wlp(S, false)$ is calculated twice, once for $wp(S\,;T, true)$ and once directly, terms can appear a quadratic number of times in the final WP formula. These terms can be replaced with a constant because they are identical. The constant can be constrained with a top level set of conjunctions, e.g.:

$$(w_1 = wlp(S, false)) \wedge (w_2 = wlp(T, false)) \Rightarrow wp(S\,;T, true) \wedge (wlp(S\,;T, false) \vee Q)$$

This removes the need to calculate $wlp(S, false)$ in $wp(S\,;T, true)$ and places a term of constant size in its place, bringing the final formula size down to a linear bound. However, the results in [FS01] show that this actually almost always creates extra work for the solver, to the point where its more efficient not to rename any formulas. This will not be discussed further due to the fact that the conditions under which predicate transformers are used in the verification framework make it unnecessary in any case.

4 Verification Framework

Having established the theoretical advantages of the efficient predicate transformers algorithm, we can consider using it in within a verification framework. The verification framework used in this thesis is intended to be a practical tool and can thus analyze strictly more types of programs than weakest preconditions alone can. The framework uses an intermediate language related to GCL called the intermediate verification language (IVL). IVL has the constructs of GCL but it can express unrestricted control flow by means of labels and goto, including unbounded looping.

Weakest preconditions can be very straightforwardly applied if we have an acyclic program. Assert statements act as conditions that must hold at given parts of the program. If we generate the weakest precondition of this program with any given predicate, we can then check the *validity* of the weakest precondition. If it is valid, then all assert statements hold in the program in all cases and post-execution, our predicate as well.

Of course, GCL-like programming languages are not of very much practical use. Even the most basic programs typically have unbounded loops. Therefore, more powerful verification algorithms that can deal with such applications are necessary.

4.1 CEGAR

Counterexample guided abstract refinement (CEGAR) is primarily based on creating an abstraction of a program and progressively refining this abstraction as needed in order to prove something about the program [Cla+00].

Abstractions are descriptions of program states using a finite set of predicates, which are simply boolean-valued formulas with variables from the program. Every abstract state is a member of the abstract domain.

Definition 12. Abstract Domain

For a set of predicates, $P = \{p_1, \ldots, p_n\}$, the abstract domain is the set of all possible conjunctions of predicates: $Abs(P) = \{\bigwedge Q \mid Q \subseteq P\}$

A formula over predicates is known as a *region* while the set of possible predicates for a region is known as the *precision* [Bey+09].

The idea of an abstraction is to represent the potentially very complex state of a program through a limited number of boolean formulas. This abstraction is reversible through concretization. Instead of verifying the concrete program, the simplified abstraction is validated instead.

Definition 13. Abstraction

An abstraction function is a function from concrete states to the abstract domain:

$\alpha : \mathcal{P}(\Omega) \to Abs(P)$

The concretization function is the corresponding function from an abstraction to concrete states:

$\gamma : Abs(P) \to \mathcal{P}(\Omega)$

Important is the requirement that α over-approximates concrete states, i.e. for every set of states $S \subseteq \gamma(\alpha(S))$ and that concretization does not lose information, i.e. for every abstraction $\alpha(\gamma(T)) \equiv T$ [CC77].

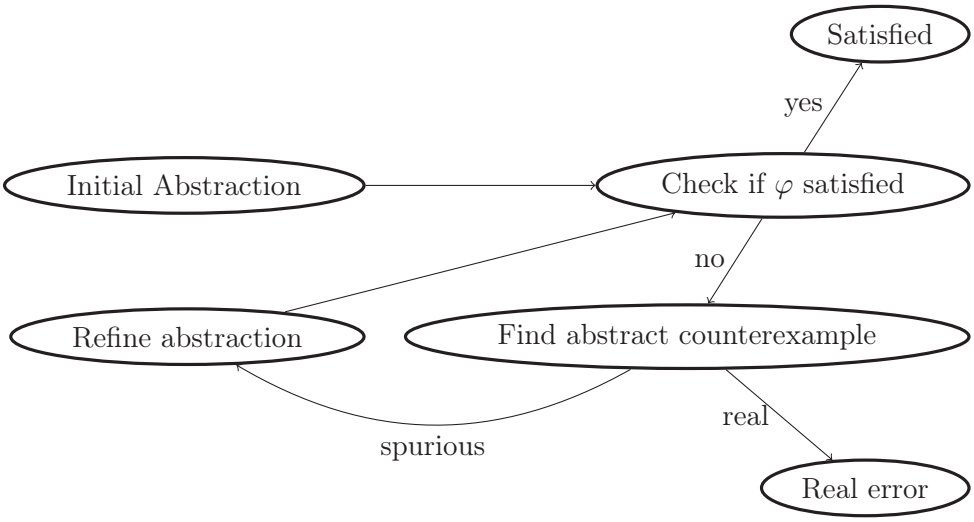


Figure 4.1.1: Main loop of CEGAR

CEGAR verifies IVL programs, but the programs are first transformed into a control flow graph.

Definition 14. Control Flow Graph (CFG)
The CFG is a directed graph whose vertices represent the basic blocks of the program and whose edges are GCL program statements. Vertices are either normal or error locations.

There are multiple ways to translate the IVL program into a CFG. Generally, each statement is transformed into an edge. Choices are represented by two outgoing edges in the CFG. Large block encoding (LBE) is another way to represent the program CFG. In short, edges can be combined together into one edge such that the semantics of the CFG remain the same [Bey+09]. With LBE, each edge may have a very large statement as its label. This will play a role in the performance of the efficient predicate transformers.

Definition 15. Abstract Transition
An abstract transition is a tuple (S, q) consisting of a GCL statement S and an abstract state $q \in Abs(P)$:

Definition 16. Abstract Reachability Tree (ART)
The ART is a directed tree (V, E) with abstract transitions as edges. The nodes correspond to a node from the CFG and are either marked or unmarked. More than one ART node may correspond to one CFG node.
Intuitively, the ART is the CFG unrolled and abstracted.

The main loop of CEGAR is pictured in Sec. 4.1.
In line 1-2 CEGAR begins with the control flow graph corresponding to the input program, the initial abstraction $P_0 = true, false$, and the initial abstract reachability tree (ART) generated from the initial locations of the CFG.
In line 3 an unmarked node in the ART is chosen and then marked. Assuming it is not an error location, its outgoing edges must be unrolled and added to the ART. CEGAR then moves onto a new unmarked node.

Algorithm 6 CEGAR

```
1: P ← {true, false}
2: while ∃n in ART with n not marked do
3:   if n not error loc then
4:     expand ART with outgoing edges CFG node of n
5:   else
6:     if ∃m pivot node then
7:       Refine abstraction
8:     else
9:       Real counterexample found
10:    end if
11:  end if
12: end while
```

Execution moves to line 6 if the chosen node is an error location. We now have a path in the ART from an initial node to an error location. This represents an abstract counterexample. At this point the search begins for a node which can not be reached by a concrete state, known as the pivot node. If no pivot node is found, a real counterexample has been found. If a pivot node is found, then the counterexample was spurious. The abstraction is refined using the information gained from the pivot node. The abstract states of the nodes along the counterexample are then recalculated using the new predicates and CEGAR continues at line 2.

4.2 Predicate Transformers

Predicate transformers play a role in CEGAR in several places. In fact, much of the time of the CEGAR loop is spent checking satisfiability of formulas. Thus, the quality of those formulas should theoretically play a significant role in the speed of the CEGAR algorithm.

A notable difference is that in CEGAR, errors are represented by error nodes in the CFG. Thus, asserts do not appear in the GCL used in the verification framework. This is advantageous because the WLP can be used at all times, since without asserts it is equivalent to the WP.

Predicate transformers were initially created and used in isolation. Either programs were acyclic or loops would be unrolled to some maximum degree. At this point, the program would be passified and then the weakest precondition was generated and checked.

With CEGAR, however, a program is initially represented as a control flow graph, because of the cyclic properties inherent to IVL. A control flow graph can not unfortunately be passified as such, because it may contain backedges. When the CFG is unrolled, those transitions can be used multiple times and thus variables may be used at some edge e_2 in the ART after they have been modified in edge e_1 corresponding to a CFG edge c_1 *dominated* by e_2's CFG edge in the control flow graph.

Figure 4.2.1: Cyclic IVL program

```
init:
  assume x > 0
  y := x*y
  x := x-1
  goto init, end

end:
  assume x <= 0
  assert y = fact(x)
```

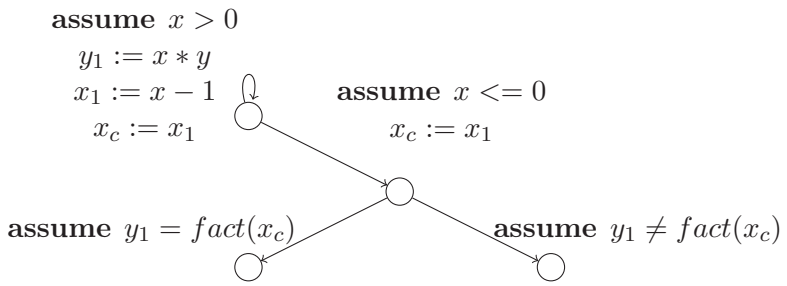


Figure 4.2.2: DSA CFG

Lemma 6. CFG with backedges can not be properly passified

Proof. Consider the IVL program in Fig. 4.2.1. Trying to passify the edge leaving the initial node, the assignment statement would be transformed to **assume** $x_1 = x - 1$ resulting in the CFG in Sec. 4.2. Attempting to unroll this might result in the ART in Fig. 4.2.3, for example.

This breaks the rules of passification however, because the sequence from the initial node to the final is no longer passive. When executing the edge statement again, a potentially contradicting value is assumed for x_1, breaking the invariant of a passified program.

□

Therefore, the program can not simply be passified at the beginning of the CEGAR loop. Instead, each time the weakest precondition algorithm is called, passification is done on the input program, even if that program or parts of it have been passified before.

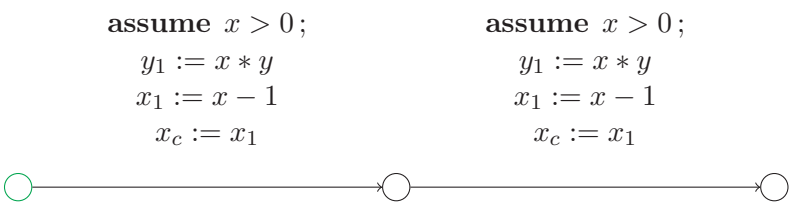
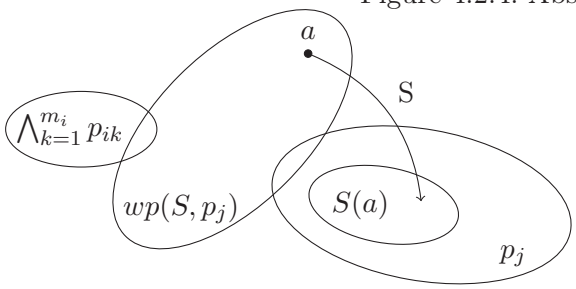


Figure 4.2.3: Unrolled ART

Figure 4.2.4: Abstract post relationship



4.2.1 ART Unrolling

The first place where CEGAR takes advantage of weakest precondition is when unrolling the outgoing edges of a newly marked node in the ART. Given a node in the ART, the outgoing edges of the corresponding CFG node, along with the ART node's abstraction, each form an abstract transition.

Definition 17. Abstract Post

The abstract post operator $post : GCL \times Abs \times Preds \to Abs$ takes an abstract transition (S, q) and the set of predicates. The operator determines the predicates that remain true after execution of S, given the predecessor region q [BPR01]. This is pictured in Alg. 7.

Note, this is not necessarily the strongest abstraction possible, but it is easier to compute.

Algorithm 7 Abstract Post

```
1: P = {p_1, ..., p_r}
2: q = ⋀_{i=1}^{m} p_i
3: abs ← true
4: for j = 1 to r do
5:    if (⋀_{i=1}^{m} p_i) ⇒ wlp(S, p_j) is valid then
6:       abs ← (abs ∧ p_j)
7:    end if
8: end for
9: return abs
```

The abstract post operator can be calculated with either strongest postconditions or weakest preconditions (cf. Prop. 1). The relationship check in line 7 is shown in Fig. 4.2.4, it decides whether all states satisfying $\bigwedge_{k=1}^{m_i} p_{ik}$ also satisfy $wp(S, p_j)$ and thus whether the successor states of S satisfy p_j.

In order to make sure we can use efficient weakest liberal preconditions here, we need to figure out what form the WLP takes in the final query to the SMT solver when using *ewlp*. We can replace the query, negated for validity, with the following transformations:

$$\neg\left(\left(\bigwedge_{k=1}^{m_i} p_{ik}\right) \Rightarrow wlp(S, p_j)\right) \equiv \neg\left(\left(\bigwedge_{k=1}^{m_i} p_{ik}\right) \Rightarrow ewlp(S, p_j)\right) \equiv \neg\left(\left(\bigwedge_{k=1}^{m_i} p_{ik}\right) \Rightarrow \forall\overline{x}.\,\varphi\right)$$

$$\equiv\neg\left(\left(\neg\bigwedge_{k=1}^{m_i} p_{ik}\right) \vee \forall\overline{x}.\,\varphi\right) \equiv \neg\forall\overline{x}.\left(\left(\neg\bigwedge_{k=1}^{m_i} p_{ik}\right) \vee \varphi\right) \equiv \exists\overline{x}.\left(\bigwedge_{k=1}^{m_i} p_{ik}\right) \wedge \neg\varphi$$

This formula can therefore be skolemized as per Definition 10 and handed to the SMT solver.

4.2.2 Pivot Node Search

The second place where the weakest precondition comes into play is the search for the pivot node. In the pivot finding operation, the verification framework needs to check whether a certain state is reachable at all.

Given an error location, there is a path from the initial node of the ART to the error node. The pivot node is found by beginning at the end of the path and searching iteratively backwards until the node is found from which the abstract counterexample does not correspond to a real counterexample. For each node a, the set of concrete predecessor states given the GCL label coming into that node is calculated. The intersection of this set with the region of the source node in the ART describes those state that might also lead to a.

Algorithm 8 Pivot Node Search

```
1: Counterexample = {a_1, ..., a_n}
2: bad ← true
3: for i = n to 1 do
4:    bad ← wep(edge(a_i), bad)
5:    if ∅ = bad ∩ region(a_i) then
6:       return a_i
7:    end if
8: end for
```

In order to calculate the set of predecessor states, the verification framework takes advantage of weakest existential preconditions. The set of concrete predecessor states are defined as those which *might* lead to the region in question. Given an edge, E, and the region, r, in question, this is calculated using $wep(E, r)$.

In order to check whether a region is empty, the formula describing the region is checked for satisfiability. The intersection of two regions is calculated using conjunction and thus the formula has the form: $\varphi_{region} \wedge wep(edge(a_i), bad)$.

Because the WEP when checked for satisfiability can be transformed to an equi-satisfiable formula, this formula can be directly given to the SMT solver.

Another issue arises when doing an iterative search. Each *wep* calculation is used in the *wep* calculation for the following step. For performance reasons, the results of SMT queries should be able to be cached. This way, identical queries do not need to be given multiple times. In order for this to work, the translation must be pure, i.e. it must always return the exact same output for a given input. Therefore, the first intermediate for a variable x always has the same name across any weakest precondition calculation.

However, this causes problems when using efficient WEPs as postconditions for subsequent efficient WEP calculations. Consider the following sequence of calculations:

$$1 : ewep(x := x + 2, x > 0) = vc_0 = ((x_1 = x + 2) \Rightarrow x_1 > 0))$$
$$2 : ewep(x := x + 2, vc_0)$$

At step 2, the statement is first passified and then the WEP is generated:

$$dsa(x := x + 2) = x_1 := x + 2$$
$$ewep(x_1 := x + 2, vc_0) = (x_1 = x + 2) \Rightarrow ((x_1 = x_1 + 2) \Rightarrow x_1 > 0)$$

The variable x in vc_0 is mapped to the current substitution for that variable, x_1, causing a semantic conflict. The solution is for the pivot finder calculation to maintain a counter variable which is understood by the WEP calculation such that the new variable names in the *wep* calculation are unique in every step of the pivot search and do not collide with previous WEPs. This maintains the purity of the transformation modulo the counter variable as well as prevents the collision problem from occurring. The new DSA program and WEP are:

$$dsa(x := x + 2) = x_{1_1} := x + 2$$
$$ewep(x_1 = x + 2, vc_0) = (x_{1_1} = x + 2) \Rightarrow ((x_1 = x_{1_1} + 2) \Rightarrow x_1 > 0)$$

4.2.3 Predicate Refinement

When a spurious counterexample is found, the predicate abstraction must be refined such that there is no longer a counterexample along that path. Precisely how new predicates are extracted is outside the scope of this thesis. But given the pivot node, the remainder of the spurious counterexample and the new predicates, the new abstraction for each node on the remainder can be computed using the abstract post function. The function is computed just as in Alg. 7 for a node on the path and its outgoing edge, except this time for already existing ART nodes. There are new predicates in P such that the abstraction for the next node may differ from its previous abstraction.

5 Results

The performance of the new predicate transformers in CEGAR is evaluated using the verification framework on a set of test programs, both C and IVL.

There are two versions of the new predicate transformers, a one-step and a two-step algorithm. The two-step algorithm is the algorithm as explained in Ch. 3. The DSA version of a program is first calculated and then the efficient predicate transformers generate the verification condition. Another way to do this is to simply do the DSA transformation on the fly, never creating assignments or assumes but rather translating directly to the logical formula. The results show that the one-step algorithm is marginally but consistently quicker, perhaps because no intermediate data structure needs to be created.

The speedup and slowdown can be further pinpointed by looking at both points of use of EPT, i.e. abstract post or focus operator. In some cases it appears that using EPT for abstract post seems to make the most difference while in others it appears to be for the focus operator. What causes one or the other to have a larger effect is not clear. However, to get a more general idea of the effects, the presented benchmarks use either no EPT or EPT everywhere.

The first comparison is shown in Table 5.1, a set of 59 benchmarks where CEGAR terminated either with the new or with the old predicate transformers. Columns with a $'$ indicate the results using the new algorithm. The data points are respectively: the time spent in SMT queries, the time spent generating verification conditions, the size of the VCs and finally the size of the new as a percentage of the size of the old. Highlighted rows are particularly illustrative and are discussed further on.

Table 5.1: CEGAR with MathSAT, incremental and with LBE max 10000000

Program	t_{SMT}	t'_{SMT}	t_{wp}	t'_{wp}	$\|\varphi\|$	$\|\varphi'\|$	$\frac{\|\varphi'\|}{\|\varphi\|} * 100$
array_max-1_tuc	40.272	67.46	1.328	1.444	2385	109	4.57
array_max-2_tuc	189.48	282.852	11.844	7.392	949368	335	0.0353
bist_cell_tuc	0.016	0.16	0.504	0.004	6297362	252	0.004 00
bubblesort-1_tuc	0.632	4.804	0.028	0.02	43	48	112
bubblesort-2_tuc	44.272	209.12	0.908	1.352	118	80	67.8
bubblesort_init-1_tuc	0.084	0.348	0.004	0.004	19	33	174
bubblesort_init-2_tuc	1.22	6.088	0.068	0.024	829	86	10.4
bubblesort_init-3_tuc	6.04	29.336	0.552	0.264	84437	110	0.130
flpy_simpl4.cil_64_fuc	0.024	0.076	0	0	21	47	224
flpy_simpl4.cil_fuc	0.016	0.08	0	0.004	22	47	214
kbf_smpl2.cil_fuc	0	0	0	0	1	36	3600
kbf_smpl2_w.cil_fuc	0	0	0	0	1	33	3300
kundu-1_fuc	4	6.784	0.12	0.324	18	18	100
kundu-2_fuc	12.368	22.16	0.488	0.944	16	18	113

Program	t_{SMT}	t'_{SMT}	t_{wp}	t'_{wp}	$\lvert\varphi\rvert$	$\lvert\varphi'\rvert$	$\frac{\lvert\varphi'\rvert}{\lvert\varphi\rvert} * 100$
kundu_tuc	343.956	OOM	27.152	OOM	3252	OOM	0
pc_sfifo_1_tuc	0.484	1.808	0.044	0.024	28	38	136
pc_sfifo_2_tuc	8.848	45.672	2.152	1.172	13193	67	0.508
pc_sfifo_3_tuc	0.064	0.112	0.004	0	12	20	167
s3_clnt_1.cil_fuc	4.868	11.256	2.144	0.38	377	156	41.4
s3_clnt_2.cil_fuc	4.868	27.168	4.128	0.836	925	199	21.5
s3_clnt_3.cil_fuc	3.472	30.584	3.7	0.904	1168	151	12.9
s3_clnt_4.cil_fuc	2.724	25.42	1.456	0.688	808	164	20.3
s3_srvr_1.cil_fuc	3.792	11.32	0.296	0.304	22	43	195
s3_srvr_2.cil_fuc	6.9	22.32	0.42	0.56	24	45	188
smpl_arr_inv-1_tuc	0	0	0	0	295	262	88.8
smpl_arr_inv-2_tuc	0	0	0	0	424	349	82.3
smpl_arr_inv-3_tuc	0	0	0	0	592	462	78.0
smpl_arr_inv-4_tuc	0	0	0.004	0	773	575	74.4
smpl_arr_inv-5_tuc	0	0	0	0	993	714	71.9
smpl_arr_inv-6_tuc	0.004	0	0	0.004	1226	853	69.6
token_ring.1_tuc	1.424	5.328	4.236	1.256	59770	44	0.0736
transmitter.1_fuc	0.06	0.192	0.012	0.02	678	27	3.98
transmitter.2_fuc	0.452	2.48	2.06	0.104	2340211	101	0.004 32
transmitter.3_fuc	1.476	4.928	83.288	0.168	1.98×10^{9}	277	1.40×10^{-5}
transmitter.4_fuc	0.128	9.86	116.56	0.34	1.83×10^{11}	2392	1.30×10^{-6}
byte_add_fuc	10.08	3	1.52	0.184	130	72	55.4
gcd_1_tuc	4.868	5.088	0	0	31	42	135
gcd_4_tuc	202.272	183.492	0.008	0.032	13	17	131
interleave_bits_tuc	140.336	OOM	1.124	OOM	358	OOM	0
jain_1_tuc	0.052	0.068	0	0	9	19	211
jain_2_tuc	0.136	0.176	0	0.004	15	28	187
modulus_tuc	14.524	13.98	0	0	16	21	131
num_conv_2_tuc	0.008	0.004	0	0	15	18	120
s3_1_fuc.BV.c.cil	41.288	143.612	82.608	3.644	9228	165	1.79
s3_3_fuc.BV.c.cil	6.252	42.12	4.292	1.256	628	147	23.4
soft_float_2_tuc	1.012	1.168	0	0.004	23	33	143
soft_float_4_tuc	35.112	32.228	0.1	0.28	137	134	97.8
soft_float_5_tuc	0.012	0.016	0	0	6	13	217
diamond_fuc2	0.012	0.016	0	0	17	28	165
nested_arrays_tuc	0	0	0	0	68	62	91.2
simple_add_tuc	0	0	0	0	3	6	200
structarray_tuc	0	0.004	0	0	68	62	91.2
counter_fuc	0.06	0.068	0	0	8	12	150
gcd_minus_fuc	2.036	1.716	0	0	26	29	112
loop_byte2_fuc	0.012	0.024	0	0	8	15	188
loop_byte3_fuc	0.02	0.024	0	0.012	9	17	189
loop_byte_fuc	0.008	0.008	0	0	8	16	200
loop_fuc	2.152	9.208	0.02	0.08	21	25	119

Program	t_{SMT}	t'_{SMT}	t_{wp}	t'_{wp}	$\lvert\varphi\rvert$	$\lvert\varphi'\rvert$	$\frac{\lvert\varphi'\rvert}{\lvert\varphi\rvert} * 100$
loop_tuc	5.116	15.524	0.028	0.076	24	28	117
Avg. ($\lvert\varphi\rvert > 100$)	15.69	39.03	12.88	0.86	7.43×10^9	328	30.87

For test cases where the original size of the formula was less than about 100, the new algorithm does not bring much of an improvement in size, sometimes creating a larger formula. In these cases, the original algorithm does not exhibit the exponential behavior and thus the overhead of the additional assignments dominates.

However, with sizes above 100, the new algorithm decreases the size regularly, sometimes drastically. Additionally, time spent generating the VCs decreases as well. For cases with original VC sizes above 100, the new VCs are on average 30.87% smaller and the time needed to generate formulas decreases by 94%.

On the other hand, time spent in the SMT solver generally increases when using the new algorithm. Why exactly this is the case is difficult to say. CEGAR uses the SMT solver in incremental mode, so that information gained about predicates can be kept and used for following queries. The solver essentially uses two sets of heuristics in incremental mode and the second set, for handling incremental queries, is quite a bit weaker than the first. Therefore, it is entirely possible that the structure created with efficient predicate transformers does not lend itself to these heuristics, forcing the SMT solver to do a lot of preprocessing. Two of the benchmarks also end in an out of memory error for the EPT algorithm. This may potentially be an issue with the passification process.

Some benchmarks show improvements with EPT:

- `transmitter.*_fuc`: The entire family of benchmarks responds very well to using EPT. Indeed, CEGAR was not able to terminate using the classic predicate transformers on `terminator.4`, although information was collected about what CEGAR had done up to the timeout point. CEGAR with EPT terminates. Much time was spent generating the VCs with the classic algorithm and they are indeed enormous. EPT cuts their size down quite dramatically, very quickly calculates them and terminates.

- `gcd_4`: The new algorithm causes a small but noticeable speedup, although the overall size only barely decreases.

Other benchmarks highlight the drawbacks:

- `bubblesort-2_tuc`: This benchmark shows a drastic increase in time spent in the SMT solver when using EPT but the change in average formula size is quite small.

- `s3_*_fuc`: Here, the size and speed gains from using EPT were not enough to offset the extra time spent querying the solver.

- `array_max`: Another example of where the tradeoff between smaller VC size and SMT time ultimately does not benefit the overall performance of CEGAR.

Two factors have a large effect on the performance of predicate transformers. The first is large-block encoding. Because without large-block encoding each edge of the CFG is a minimal GCL statement, each predicate transformer has only that minimal GCL statement as input. This results in a small corresponding verification condition, where the growth problems of the

classic predicate transformers can not have a noticeable effect. Thus the overhead of using the new predicate transformers can have a larger influence on the difference between the classic and new predicate transformers. When large-block encoding is turned on, however, abstract post and pivot node searching operations have to work with much larger edge labels. This lets the differences between the two versions become apparent.

Although the formulas themselves are indeed smaller, the results show that efficient predicate transformers do not in general bring a performance benefit to CEGAR. The lack of performance gains is perhaps most attributable to the performance of the SMT solver itself. The time spent querying and waiting plays a very large role in performance of verification methods like CEGAR. SMT solvers typically have a vast range of heuristics and simplifications they take advantage of. The performance of these factors is typically not very predictable and thus the SMT solver is usually considered a black box.

6 Conclusion

We aimed to reduce the runtime of a verification framework by improving the logical formulas generated by predicate transformers. First, a collection of predicate transformers was introduced. We explained the relationship between them and motivated their adoption as a verification technique.

A drawback of predicate transformers that can cause them to produce exponentially large formulas was then demonstrated. The size of these formulas has a significant effect on how long it takes to verify programs and even if they can be verified. Next, we looked into the origins of this explosive growth.

We then presented a way to avoid this drawback by first transforming programs to an equivalent form. For this form of program, a modified algorithm for predicate transformers can generate formulas without exponential blowup. The equivalence of the program transformation as well as the predicate transformers was proven.

Finally, focus was shifted to the CEGAR algorithm and we explored the theoretical basis for the algorithm itself as well as how it can take advantage of predicate transformers. The performance differences between classic and efficient predicate transformers in the CEGAR algorithm were presented and then examined. We offered commentary and interpretations for the outcome of using the new predicate transformers.

In conclusion, the new predicate transformers do not bring a universal performance benefit to CEGAR but can help quite dramatically in some cases. To justify using the new algorithm in general, more analysis must be done and we provide some orientation for future exploration.

6.1 Future Work

Directions for further investigation include looking for heuristics that might be able to predict if and perhaps specifically where EPTs should be used in CEGAR. The factors at play include both the structure and size of the GCL commands and predicates as well as the structure of the program CFG but how these factors interact is unclear. For example, LBE and EPT could potentially be used in concert such that they complement each other but it is not apparent how to find the best combination.

Furthermore, the reasons for the time discrepancies between classic verification conditions and EPT verification conditions in the SMT solver can be explored. The internals of SMT solvers are complex and not particularly well documented. Nevertheless, this aspect has the most potential to bring additional performance benefits.

Bibliography

[AWZ88] B. Alpern, M. N. Wegman, and F. K. Zadeck. "Detecting Equality of Variables in Programs". In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '88. San Diego, California, USA: ACM, 1988, pp. 1–11. ISBN: 0-89791-252-7. DOI: 10.1145/73560.73561. URL: http://doi.acm.org/10.1145/73560.73561.

[Bey+09] D. Beyer et al. "Software model checking via large-block encoding". In: *Formal Methods in Computer-Aided Design, 2009. FMCAD 2009*. Nov. 2009, pp. 25–32. DOI: 10.1109/FMCAD.2009.5351147.

[BM07] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007. ISBN: 3540741127.

[BPR01] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. "Boolean and Cartesian Abstraction for Model Checking C Programs". In: Springer, 2001, pp. 268–283.

[BW89] R. J. R. Back and J. von Wright. "Combining Angels, Demons and Miracles in Program Specifications". In: *Theoretical Computer Science* 100 (1989).

[CC77] Patrick Cousot and Radhia Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '77. Los Angeles, California: ACM, 1977, pp. 238–252. DOI: 10.1145/512950.512973. URL: http://doi.acm.org/10.1145/512950.512973.

[Cla+00] Edmund Clarke et al. "Counterexample-Guided Abstraction Refinement". English. In: *Computer Aided Verification*. Ed. by E.Allen Emerson and AravindaPrasad Sistla. Vol. 1855. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2000, pp. 154–169. ISBN: 978-3-540-67770-3. DOI: 10.1007/10722167_15. URL: http://dx.doi.org/10.1007/10722167_15.

[Dij75] Edsger W. Dijkstra. "Guarded Commands, Nondeterminacy and Formal Derivation of Programs". In: *Commun. ACM* 18.8 (Aug. 1975), pp. 453–457. ISSN: 0001-0782. DOI: 10.1145/360933.360975. URL: http://doi.acm.org/10.1145/360933.360975.

[FS01] Cormac Flanagan and James B. Saxe. "Avoiding Exponential Explosion: Generating Compact Verification Conditions". In: *Symposium on Principles of Programming Languages*. ACM, 2001, pp. 193–205.

[KS08] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. 1st ed. Springer Publishing Company, Incorporated, 2008. ISBN: 3540741046, 9783540741046.

[Mar08] Joao Marques-silva. "Practical applications of boolean satisfiability". In: *In Workshop on Discrete Event Systems (WODES*. IEEE Press, 2008.

[RL05] K. Rustan and M. Leino. “Efficient Weakest Preconditions”. In: *Inf. Process. Lett.* 93.6 (Mar. 2005), pp. 281–288. ISSN: 0020-0190. DOI: 10.1016/j.ipl.2004.10.015. URL: http://dx.doi.org/10.1016/j.ipl.2004.10.015.