CoVaC: Compiler Validation by Program Analysis of the Cross-Product

Anna Zaks and Amir Pnueli

New York University, New York, {ganna,amir}@cs.nyu.edu

Abstract. The paper presents a deductive framework for proving program equivalence and its application to automatic verification of transformations performed by optimizing compilers. To leverage existing program analysis techniques, we reduce the equivalence checking problem to analysis of one system - a cross-product of the two input programs. We show how the approach can be effectively used for checking equivalence of consonant (i.e., structurally similar) programs. Finally, we report on the prototype tool that applies the developed methodology to verify that a compiler optimization run preserves the program semantics. Unlike existing frameworks, CoVaC accommodates absence of compiler annotations and handles most of the classical intraprocedural optimizations such as constant folding, reassociation, common subexpression elimination, code motion, dead code elimination, branch optimizations, and others.

1 Introduction

Compilers, especially optimizing compilers, are quite large applications, which are bound to have bugs. At present, the GCC Bug Database contains over 3 thousand reported bugs some of which may alter the behavior of programs being compiled. This is highly undesirable, especially in safety critical and high-assurance software, where the effort of program correctness verification is extensive. First, the developers manually examine code and test it. Then, numerous verification tools and techniques are applied to verify that the source code satisfies the desired properties. After all the rigorous checks are complete, the program is compiled by an optimizing compiler and released. Clearly, the verification effort should not stop here - it is highly advisable to ensure that the transformations performed by a compiler preserve the semantics of a program.

That is precisely the goal of Translation Validation (TV) [16] - it ensures that optimizing transformations preserve program semantics. In essence, instead of attempting verification of a given compiler, each compiler run is followed by a validation pass that automatically checks that the target code produced by the compiler is semantically equivalent to the source code. A good question is: "Can this goal be achieved?" The problem of program equivalence is undecidable. However, since the focus is only on compiler optimizations, the number of false alarms can be drastically minimized or even eliminated, intuitively, due to the fact that we are aware of the analyses used by the optimizing compilers, and since those analyses are mechanical in nature.

In this paper, we present a Compiler Verification by Program Analysis of the Cross-Product framework (CoVaC) - a novel translation validation approach, in which one constructs a *comparison system* - a cross-product of the source and target programs. The input programs are equivalent if and only if the comparison system satisfies a certain specification. This allows us to leverage the existing methods of proving properties of a single program instead of relying on program analysis and proof rules specialized to translation validation, used by the existing frameworks [21, 13, 18]. CoVaC is not tailored to validation of compiler transformations - it targets program equivalence in general; for example, it can be applied to validation of language-based security properties [4].

The CoVaC framework can be used in various settings, and, while the check for specification conformance is expected to be the same, the construction of the comparison system may diverge. For example, compiler writers may use translation validation for creation of a self-certifying compiler and, thus, may assume full knowledge of the inner workings of a particular compiler. In this case, the compiler itself may output the comparison system. The second part of this paper pursues the other extreme - it describes a method for automatic generation of the comparison system, and thus, a translation validation algorithm, which accommodates no compiler cooperation. To the best of our knowledge, the existing translation validation frameworks which handle a comparable set of optimizations at least to some degree rely on compiler assistance. The lack of compiler dependency makes it possible to develop a general purpose verification tool that can be used to verify the transformations performed by different compilers. Such tool would be especially useful to compiler users who may have to work with a particular existing compiler. Additionally, this methodology can be of service to compiler developers to facilitate testing of immature compilers.

In order to make the validator of non-cooperative compilers feasible and effective, we currently restrict the set of transformations under consideration to intraprocedural optimizations in which each loop in the target program corresponds to a loop in the source program; we refer to such input systems as *consonant*. Many of the classical compiler optimizations such as constant folding, reassociation, induction variable optimizations, common subexpression elimination, code motion, branch optimizations, register allocation, instruction scheduling, and others fall into this category. These optimizations are usually referred to as structure preserving [21]. Finally, this paper reports on a prototype tool CoVaC that applies the developed framework to verification of optimizing transformations performed by LLVM 1.9 [12, 2] - a very aggressive open-source compiler.

In summary, this paper makes the following contributions. First, it presents a novel deductive framework for checking equivalence of infinite state programs. Second, it defines the notion of consonance and shows how the method can be effectively applied to consonant programs. The presented algorithm does not rely on any additional input; thus, it can be used to verify compilations while treating the compiler as a black-box. Due to lack of space, the paper does not contain proofs and only briefly discusses the implementation details. For a full version, the reader is referred to our technical report [20].

The rest of the paper is organized as follows. Section 2 introduces our formal model and defines the notion of correct translation. We describe the general framework for establishing program equivalence in Section 3. Section 4 presents the algorithm for comparison system construction, which requires no compiler cooperation. An example is presented in Section 5; and Section 6 focuses on the experimental results. We discuss the related work and conclude in Section 7.

2 Formal Model and the Notion of Correct Translation

2.1 Transition Graphs

Our model is similar to that presented in [15] for verification of procedural programs.

A program (application) $\mathcal{A}$ consists of $m+1$ procedures: *main*, $f_1, \ldots, f_m$, where *main* represents the main procedure, and $f_1, \ldots, f_m$ are procedures which may be called from *main* or from other procedures. We use $f_i(\vec{x},\ \&\vec{z})$ to denote the signature of a procedure. Here, call-by-value parameter passing method is used for $\vec{x}$, and call-by-reference is used for $\vec{z}$. A procedure may return a result by means of $\vec{z}$ variables. We use $\vec{y}$ to denote the typed variables of a module. $\vec{y} = (\vec{x};\ \vec{z};\ \vec{w})$, i.e. the variables in $\vec{y}$ are partitioned into $\vec{x}$, $\vec{z}$, and $\vec{w}$, where $\vec{x}$ and $\vec{z}$ are the *input* parameters and $\vec{w}$ denotes the *local* variables of the module.

Each procedure is presented as a *transition graph* $f_i := (\vec{y},\ \mathcal{N}_i,\ \mathcal{E}_i)$ with variables $\vec{y}$, nodes (locations) $\mathcal{N}_i = \{r^i = n_0^i,\ n_1^i,\ n_2^i,\ \ldots,\ n_k^i = t^i\}$ and a set of labeled edges $\mathcal{E}_i$. It must have a distinct root node r^i as its only entry point, a distinct tail node t^i as its only exit point, and every other node must be on a path from r^i to t^i. Nodes of the graph are connected by directed edges labeled by instructions. There are four types of instructions: guarded assignments, procedure calls, reads, and writes. Consider a procedure $f_i(\vec{x};\ \&\vec{z})$ with $\vec{y} = (\vec{x},\ \vec{z},\ \vec{w})$. Let $\vec{u}$ include variables from $\vec{y}$; and $E(\vec{y})$ be a list of expressions over $\vec{y}$.

- A *guarded assignment* is an instruction of the form $c \rightarrow [\vec{u} := E(\vec{y})]$, where guard c is a boolean expression. When the assignment part is empty, we abbreviate the label to a pure condition $c?$.
- *Procedure call* instruction $f_k(E(\vec{y}), \vec{u})$ denotes a call to module $f_k(\vec{x}_f;\ \&\vec{z}_f)$, passing input parameters $E(\vec{y})$ by value and $\vec{u}$ by reference.
- *Read* and *write* instructions are denoted by $read(\vec{u})$ and $write(\vec{u})$. They are used to express the interaction of the procedure with the outside world; e.g. I/O instructions.

The guards of read, write, and procedure call instructions always evaluate to *true*. A transition graph is *deterministic* when, for every node n, the guards of all edges departing from n are mutually exclusive. A transition graph is *non-blocking* when, for every node, the disjunction of the guards evaluates to *true*. In this work, we only consider deterministic non-blocking systems.

Transition graphs can be used to model programs written in procedural languages. In order to construct a formal model of a program, we first choose a set of program locations Υ such that:

- At least one location in each loop belongs to Υ.
- For every procedure, both procedure entry and exit belong to Υ.
- The locations before and after read, write, and procedure call belong to Υ.

The choice of Υ can be generalized not to require at least one location per each loop as long as we can ensure that the transitions between every pair of locations are computable [10]. Each procedure (or function) whose implementation is given is represented by a transition graph. We choose the set Υ of a procedure f_i to be the set of nodes for the corresponding transition graph. For every pair of locations $n,\ m$ in Υ, if there exists a path π from n to m, which does not pass through any other location from Υ, we add edge (n, m) to the graph and label it by the instruction that summarizes the effect of executing the path π.

2.2 States and Computations

We denote by $\vec{d} = (\vec{d^x};\ \vec{d^z};\ \vec{d^w})$ a tuple of values, which represents an interpretation (i.e., an assignment of values) of the module variables $\vec{y} = (\vec{x};\ \vec{z};\ \vec{w})$. A *state* of a module f is a pair $\langle n; \vec{d}\rangle$ consisting of a node n and a data interpretation $\vec{d}$. A $(\vec{\xi}, \vec{\zeta})$-*computation* of module f is a maximal sequence of states and labeled transitions:

$$\sigma : \langle r;\ (\vec{\xi}, \vec{\zeta}, \vec{\top})\rangle \xrightarrow{\lambda_1} \langle n_1;\ \vec{d_1}\rangle \xrightarrow{\lambda_2} \langle n_2;\ \vec{d_2}\rangle\ \ldots$$

The tuple $\vec{\top}$ denotes uninitialized values. At the first state of the computation, the location is r, the entry location of f; the values of input variables $\vec{x}$ and $\vec{z}$ are set to $\vec{\xi}$ and $\vec{\zeta}$, respectively, and the local variables $\vec{w}$ are not initialized. Labels of the transitions are either labels of edges in the program or the special label *ret*. Each transition must be justified by either an intra-procedural transition, a call transition, or a return transition such that the call and return transitions are *balanced*. See our technical report [20] for the formal definition.

We use $Cmp(f)$ to denote the computations of a transition graph f. We define a set of computations of a procedural program $\mathcal{A}$, denoted $Cmp(\mathcal{A})$, to be the set of computations $Cmp(main)$.

2.3 Correct Translation

In this work, we are only concerned with intraprocedural optimizations, so for simplicity, we are going to assume that the corresponding procedures of $\mathcal{S}$ and $\mathcal{T}$ have the same names; we are going to use a superscript notation to differentiate between the source and target procedures. We define the correctness of translation via equivalence of program behaviors that can be observed by the user. Intuitively, given the same input both, the source program $\mathcal{S}$ and the target program $\mathcal{T}$, must produce the same output; we also observe the values of input and output parameters of every procedure.

Given a computation, we define V_s - the set of *observable variables* at a state $s = \langle n, d\rangle$, to be the minimal set satisfying the following conditions:

- If s is a state immediately after transition $read(\vec{u})$, $V_s \supseteq \vec{u}$.
- If s is a state immediately before transition $write(\vec{u})$, $V_s \supseteq \vec{u}$.

- If $n = r$ is the entry node of procedure $f(\vec{x},\ \&\vec{z})$, $(V_s \supseteq \vec{x}) \wedge (V_s \supseteq \vec{z})$.
- If $n = t$ is the exit node of procedure $f(\vec{x},\ \&\vec{z})$, $V_s \supseteq \vec{z}$.

Above, we use $V_s \supseteq \vec{u}$ to denote $V_s \supseteq \{v : \forall v \text{ in } \vec{u}\}$.

We associate *observation function* $\mathcal{O}$ with each program, mapping the source and target states and transition labels into a common domain. The observation function needs to ensure that read and write transitions of the source and target computations match. Formally, given a state $s = \langle n, d\rangle$, an observation function $\mathcal{O}(s)$ is defined as following. Let V_s be the set of observable variables at s. If $V_s = \emptyset$ then $\mathcal{O}(s) = \perp$, else $\mathcal{O}(s) = \vec{d}_{V_s}$. We obtain $\vec{d}_{V_s}$ by restricting $\vec{d}$ only to the values that correspond to the variables in V_s. Given a transition label λ, an observation function $\mathcal{O}(\lambda)$ is defined as follows. If λ is a label of a transition that is a read, a write, a call to procedure g, or a return from procedure g, $\mathcal{O}(\lambda)$ is equal to $read$, $write$, $call_g$, or ret_g, respectively. Otherwise, $\mathcal{O}(\lambda) = \perp$.

An *observation* of a computation σ, denoted $o(\sigma)$, is obtained by applying the observation function $\mathcal{O}$ to each state and each transition label in σ. That is, for $\sigma : s_1 \xrightarrow{\lambda_1} s_2 \xrightarrow{\lambda_2} s_3 \ldots$, we get $o(\sigma) : \mathcal{O}(s_1) \xrightarrow{\mathcal{O}(\lambda_1)} \mathcal{O}(s_2) \xrightarrow{\mathcal{O}(\lambda_2)} \mathcal{O}(s_3) \ldots$.

Definition 1 *Computations σ and σ' are* **stuttering equivalent**, *denoted $\sigma \sim_{st} \sigma'$, if their observations $o(\sigma)$, $o(\sigma')$ only differ from each other by finite sequences of pairs $\perp \xrightarrow{\perp}$ or $\xrightarrow{\perp} \perp$.*

Stuttering equivalence is used to ensure that even though the programs may have to execute a different number of instructions to get to an observable state, the difference is always finite. Our assumption is that the user is not time-sensitive so this finite delta cannot be observed. For example, $\beta \sim_{st} \beta'$:

$o(\beta):\ \perp \xrightarrow{read} (5, 22) \xrightarrow{\perp} \perp \xrightarrow{\perp} \perp \xrightarrow{\perp} (110) \xrightarrow{write} \perp$

$o(\beta'):\ \perp \xrightarrow{read} (5, 22) \xrightarrow{\perp} \perp \xrightarrow{\perp} (110) \xrightarrow{write} \perp \xrightarrow{\perp} \perp$

In both computations, first two numbers: 5 and 22, are read; and then, after a finite number of steps, their product: 110, is written out.

Definition 2 *We say that procedure $f_{\mathcal{T}}$ is* **a correct translation** *of procedure $f_{\mathcal{S}}$ if, for every $(\vec{\xi}, \vec{\zeta})$-computation $\sigma_{\mathcal{T}}$ in $Cmp(f_{\mathcal{T}})$, there exists a $(\vec{\xi}, \vec{\zeta})$-computation $\sigma_{\mathcal{S}}$ in $Cmp(f_{\mathcal{S}})$ such that $\sigma_{\mathcal{T}} \sim_{st} \sigma_{\mathcal{S}}$, and vice versa. Program $\mathcal{T}$ is a correct translation of program $\mathcal{S}$ if $main_{\mathcal{T}}$ is a correct translation of $main_{\mathcal{S}}$.*

3 Equivalence Checking by Program Analysis of the Cross-Product $\mathcal{S} \boxtimes \mathcal{T}$.

In this section, we show that the problem of establishing correct translation is equivalent to construction of a cross-product (comparison) system $\mathcal{C} = \mathcal{S} \boxtimes \mathcal{T}$ and checking if $\mathcal{C}$ satisfies a set of correctness conditions. Our framework is general enough for establishing translation correctness of deterministic systems in presence of a wide set of intraprocedural transformations and can be easily extended to cope with interprocedural optimizations as well. Later in the paper, we present application of the method to proving translation of consonant systems. However,

the general framework can be used to reason about translation correctness in presence of structure modifying optimizations such as loop transformations [22].

3.1 Comparison Graphs

Assume we are given two programs, $\mathcal{S}$ and $\mathcal{T}$. For each pair of the corresponding source and target procedures, $f^S = (\vec{y}^S,\ \mathcal{N}^S,\ \mathcal{E}^S)$ and $f^T = (\vec{y}^T,\ \mathcal{N}^T,\ \mathcal{E}^T)$, the *comparison transition graph* $f = (\vec{y},\ \mathcal{N},\ \mathcal{E}) = f^S \boxtimes f^T$ is defined by the set of rules below. f represents a simultaneous execution of f^S and f^T. The collection of comparison graphs for all procedures constitutes the comparison program $\mathcal{C} = \mathcal{S} \boxtimes \mathcal{T}$.

Rule 1 *(Structural Requirement)*

1. *The variables of the comparison graph $\vec{y} = (\vec{x}, \vec{z}, \vec{w})$ are defined as follows: $\vec{x} = \vec{x}^S \circ \vec{x}^T$; $\vec{z} = \vec{z}^S \circ \vec{z}^T$; and $\vec{w} = \vec{w}^S \circ \vec{w}^T$, where $\vec{v} \circ \vec{u}$ denotes concatenation of two vectors.*
2. *Each node of f is a pair of source and target nodes: $\mathcal{N} \subseteq \mathcal{N}^S \times \mathcal{N}^T$. Let r^S, t^S and r^T, t^T denote the exit and entry nodes of f^S and f^T respectively. Then $r = \langle r^S, r^T \rangle$ and $t = \langle t^S, t^T \rangle$ are the entry and exit nodes of f.*
3. *Each edge of the graph $e = (\langle n^S, n^T \rangle,\ \langle m^S, m^T \rangle) \in \mathcal{E}$, labeled by a pair of instructions $\langle op^S;\ op^T \rangle$, should be justified by one of the following:*
   - *$(n^S, m^S) \in \mathcal{E}^S$ and it is labeled by op^S; $(n^T, m^T) \in \mathcal{E}^T$ and it is labeled by op^T; and op^S and op^T are instructions of the same type (either both reads, writes, assignments, or calls to procedures with the same name).*
   - *$(n^S, m^S) \in \mathcal{E}^S$, labeled by assignment op^S; $n^T = m^T$; and $op^T = \epsilon$.*
   - *$(n^T, m^T) \in \mathcal{E}^T$, labeled by assignment op^T; $n^S = m^S$; and $op^S = \epsilon$.*

   Where, ϵ stands for assignment true?, which represents an idle transition.

Since the edges of a comparison graph are labeled by the same type instructions, reads and writes of the two systems are performed in sync.

A *composed transition* $\langle n;\ \vec{d} \rangle \xrightarrow{e^S;\ e^T} \langle n';\ \vec{d'} \rangle$ is interpreted as a sequential composition of the source and target transitions with one exception. Let e^S and e^T be labeled by $read(\vec{u}^S)$ and $read(\vec{u}^T)$. Then, the transition is enabled only if $\vec{d'_{u^S}} = \vec{d'_{u^T}}$, where $\vec{d'_{u^S}}$ and $\vec{d'_{u^T}}$ are obtained from $\vec{d'}$ by restricting it to the values that correspond to the variables $\vec{u}^S$ and $\vec{u}^T$. Thus, we require that the values fed into the source and target reads are equal. Given σ in $Cmp(f)$, we use $\sigma{\uparrow_S}$ to denote a path obtained by projection of σ onto the states and transitions related to module f^S.

Rule 2 *There does not exist σ in $Cmp(f)$ such that $\sigma{\uparrow_S}$ or $\sigma{\uparrow_T}$ contains an infinite sequence of ϵ-transitions.*

The following claim follows directly from Rule 1 and Rule 2: $\forall \sigma \in Cmp(f)\ :\ (\exists \sigma^S \in Cmp(f^S)\ :\ \sigma^S \sim_{st} \sigma{\uparrow_S}) \wedge (\exists \sigma^T \in Cmp(f^T)\ :\ \sigma^T \sim_{st} \sigma{\uparrow_T})$; i.e. every computation of the comparison graph has the corresponding computations in both source and target.

In addition, we should ensure the reverse of the previous claim: the computations of the comparison graph represent all the computations of the input systems. We say that computations of an input system, say $Cmp(f^S)$, are *covered* by $Cmp(f)$ when the following condition holds: $\forall \sigma^S \in Cmp(f^S),\ \exists \sigma \in Cmp(f)$: σ^S differs from $\sigma\uparrow_S$, by only finite sequences of (padding) ϵ-transitions. The notion of coverage is stronger then stuttering equivalence, so it follows that $\sigma^S \sim_{st} \sigma\uparrow_S$.

Rule 3 *Computations of f^S and computations of f^T are covered by $Cmp(f)$.*

Note that following Rule 3, not all edges of the input graphs have to be in the comparison graph, which allows us to disregard the unreachable states of the input systems. Now as we have defined a comparison graph, let's consider what properties it should satisfy in order to guarantee the correctness of translation.

Definition 3 *A comparison graph $f = f^S \boxtimes f^T$, is a* witness *of correct translation if for every $((\vec{\xi}\circ\vec{\xi}),\ (\vec{\zeta}\circ\vec{\zeta}))$-computation of f, its target and source projections have equal observations.* Note, we restrict the computations under consideration to those in which the input parameters are initialized with the same values.

Theorem 1 *Target function f^T is* a correct translation *of source function f^S if and only if there exists a* witness *comparison graph $f = f^S \boxtimes f^T$. In addition, if f^T is a correct translation of f^S then every comparison graph $f = f^S \boxtimes f^T$ is a witness of correct translation.* (Refer to [20] for the proof of the theorem.)

Thus, in order to determine the correctness of translation, it is sufficient to construct a comparison graph and check if it is, indeed, a witness. Fig. 1 depicts an example of a witness comparison graph. For example, let σ be the $(\vec{\top}; (5, 5))$-computation of $f(\&(Y, y))$ defined by user input 10 then

$$o(\sigma\uparrow_S) = o(\sigma\uparrow_T) = \bot \xrightarrow{\bot} \bot \xrightarrow{read} (10) \xrightarrow{\bot} (300) \xrightarrow{write} \bot$$

Fig. 1. A comparison transition graph for $f(\&(Y, y)) = f^S(\&Y) \boxtimes f^T(\&y)$. We use capital variables to denote the variables of the source and their lower case counterparts for the target.

3.2 Witness Verification Conditions

An assertion network $\Phi = \{\varphi_n | n \in locations\ of\ \mathcal{C}\}$ for a program $\mathcal{C}$ is said to be **invariant** if for every execution a state $\langle n; \vec{d}\rangle$ occurring in a computation, $d \models \varphi_n$. That is, on every visit of a computation of node n, the visiting data state satisfies the corresponding assertion φ_n associated with n.

Suppose a comparison program $\mathcal{C} = \mathcal{S} \boxtimes \mathcal{T}$ and its invariant network have been constructed. The rules presented below can be used to generate `Witness Verification Conditions` for $\mathcal{C}$. Whenever the verification conditions are valid, all the transition graphs that constitute $\mathcal{C}$ are witnesses of correct translation,

so we can apply Theorem 1 to safely conclude that the translation is correct; otherwise, we report that the translation is erroneous.

- For a write edge (n, m) labeled by ($write(\vec{u}^S)$; $write(\vec{u}^T)$):
$$\varphi_n \ \rightarrow \ (\vec{u}^S = \vec{u}^T).$$
- For a call edge $e = (n, m)$ labeled by ($g^S(E^S, \vec{u}^S)$; $g^T(E^T, \vec{u}^T)$), we check that the call arguments are equal:
$$\varphi_n \ \rightarrow \ (E^S = E^T) \wedge (\vec{u}^S = \vec{u}^T)$$
- If n is the exit node of the comparison transition graph $f^S \boxtimes f^T$, where $f^S(\vec{x}^S; \&\vec{z}^S)$ and $f^T(\vec{x}^T; \&\vec{z}^T)$, we check if the values of the variables passed by reference are equal:
$$\varphi_n \ \rightarrow \ (\vec{z}^S = \vec{z}^T).$$

Claim 1 *Let $main^S$ and $main^T$ be the main procedures of $\mathcal{S}$ and $\mathcal{T}$ respectively. A comparison graph $main = main^S \boxtimes main^T$ is a witness of correct translation and, consequently, $\mathcal{S}$ is a correct translation of $\mathcal{T}$ if all the* `Witness Verification Conditions` *associated with $\mathcal{C}$ are valid.*

Note that since we are checking that the procedure input parameters are equivalent, the invariant generation algorithm can be intraprocedural. Essentially, one can apply an assume-guarantee reasoning, where f is checked to be a a witness, assuming that all the callees of a procedure f are witnesses themselves.

The presented conditions do not constitute an inductive proof of translation correctness: it is assumed that the assertions in Φ are indeed system invariants. The extra requirement that has to be satisfied in case such a proof is desirable is that the invariant assertion network should be *inductive* [8, 20]. The availability of such a proof increases the level of confidence and allows third-party verification. In addition, it is required if one is to employ invariant generation techniques that may introduce *false positives*, such as probabilistic algorithms [9]. Automatic theorem provers such as YICES[3], CVC3[1], can be utilized to independently check the validity of the proof.

The rest of this paper describes a method for comparison system construction. However, generation of program invariants and checking their correctness are essential ingredients for solving a translation validation problem. Here, one of the main advantages of our approach comes into play. Since we have reduced the translation validation problem to analysis of a single system, any existing technique out of a vast body of work on invariant generation can be used. From our experiments, we found that, among others, global value numbering [19] and assertion checking - a static program verification technique based on computation of weakest-precondition [6], are quite effective in this setting. Refer to our technical report [20] for a detailed discussion.

4 Comparison Graph Construction

We have developed a construction algorithm for *consonant* input programs (i.e., structurally similar programs). This restriction allows for effective application of our methodology to verification of optimizing compilers even in absence of

compiler annotations. The comparison system $\mathcal{C} = \mathcal{S} \boxtimes \mathcal{T}$ is just a collection of all graphs $f = f^S \boxtimes f^T$, where f^S and f^T are the corresponding procedures from $\mathcal{S}$ and $\mathcal{T}$. Thus, it suffices to present a construction algorithm for a procedure f.

4.1 Consonant Transition Graphs

We are going to use a transition graph $f^S = (\vec{y}^S,\ \mathcal{N}^S,\ \mathcal{E}^S)$ with entry and exit nodes r^S and t^S to define several notions, which apply to both f^S and f^T. Each node of f^S belongs to one of the following categories: read, write, call, branch, unconditional assignment, and exit; denoted rd, wt, cl, br, ua, tl respectively. Specifically, we say that a node $n \in \mathcal{N}^S$ is a read node, written $\tau(n^S) = rd$, if $\exists\ (n^S, m^S) \in \mathcal{E}^S$ and (n^S, m^S) is labeled by a read instruction. Similarly, we define write, call, branch, and unconditional assignment nodes; $\tau(t^S) = tl$. Intuitively, the type of node n depends on the type of the edges outgoing from n. The node types are well defined due to the fact that the graphs are deterministic and read, write, and call edges are implicitly conditioned on *true*. Given a transition graph f^S, we define a set of *cut points*, denoted $\mathcal{P}^S$, to be a subset of graph nodes such that $\mathcal{P}^S = \{\ n^S\ :\ n^S \in \mathcal{N}^S\ \wedge\ \tau(n^S) \neq ua\ \}$. Adding all branch nodes, not only loop heads, to the cut point set allows us to have different granularity, depending on the choice of the transition graph nodes (see Section 2.1). Finer granularity improves efficiency; but it is not always applicable: the input programs have to be consonant modulo the chosen cut point set. Every computation σ^S defines a corresponding sequence of cut points, which can be obtained from σ^S by first selecting the nodes of each subsequent state and then removing nodes that are not in $\mathcal{P}^S$ from that sequence.

Definition 4 *We say that graphs f^S and f^T are* consonant *if there exists a partial map $\kappa\ :\ \mathcal{P}^S \mapsto \mathcal{P}^T$ such that $\forall\ \sigma^S, \sigma^T :\ \sigma^S \in Cmp(f^S),\ \sigma^T \in Cmp(f^T)$ the following holds: if σ^S and σ^T are defined by the same input sequence, and $n^S_0, n^S_1, \ldots$ and $n^T_0, n^T_1, \ldots$ are the cut point sequences defined by σ^S and σ^T, then $(\kappa(n^S_i) = n^T_i) \wedge (\tau(n^S_i) = \tau(n^T_i))$. Such map is called a* control abstraction.

Our comparison graph construction is going to discover the control abstraction by composing the corresponding nodes. Surprisingly many compiler optimizations preserve consonance of programs. For example, code motion, constant folding, reassociation, common subexpression elimination, dead code elimination, instruction scheduling, branch optimizations all fall into this category. On the other hand, loop reordering transformations such as tiling and interchange are not covered by the method presented below.

4.2 Algorithm `compose`

Fig. 2 presents pseudocode for the `compose` algorithm that iteratively constructs a comparison graph $f = f^S \boxtimes f^T$ for consonant input transition graphs f^S and f^T. We start the construction with a node $n_0 = \langle n^S_0,\ n^T_0 \rangle$, where n^S_0 and n^T_0 are the entry nodes of f^S and f^T, respectively. The new node n_0 is added to the `WorkList`, which is our discovery frontier: if a node `n` is placed into `WorkList`, it

means that, potentially, more edges outgoing from `n` may be discovered. At each iteration, we remove a node n from the `WorkList` and apply `matchEdges` function to construct a list of newly discovered outgoing edges. The end nodes of the edges, denoted $\mathtt{n}_e$, are placed into `WorkList`. Even though we always discover a new edge, $\mathtt{n}_e$ could have been added to f at some previous iteration and may also have successors in f. In that case, all its eventual successors must also be added to `WorkList`. Intuitively, if a new path leading to a node is added, that node has to be processed again since more outgoing edges could be discovered. The function `matchEdges` may fail, returning `NULL`. This happens when we cannot construct a comparison system satisfying the requirements from Section 3.1.

```
//Initialization:
n0:=CompNode(n0^S, n0^T); C.Nodes:={n0}; C.Edges:={}; WorkList := {n0};
//Iteration:
while( ! WorkList.isEmpty()) {
    n := WorkList.removeElement();
    MatchList := matchEdges(n,S,T);
    if(MatchList == NULL) ABORT;
    while(! MatchList.isEmpty()){
        e_new := MatchList.removeElement();
        n_e = NewCEdge.toNode();
        C.Nodes.add(n_e);  //unlike the edge, n_e may not be new*
        C.Edges.insert(e_new);
        WorkList.add(n_e);
        WorkList.add(getDescendants(n_e));
    }
}
```

Fig. 2. Algorithm `compose` that constructs the comparison transition graph $f = f^S \boxtimes f^T$. * Procedure *add* does not add duplicate items to a collection.

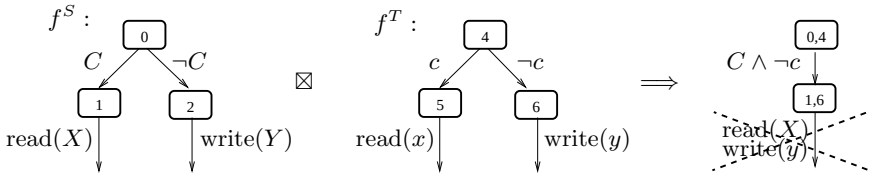
Matching the source and target edges (`matchEdges`): Below is a set of rules used to add new edges to the comparison graph:

- **Matching edges of the same type**: Given a node $\langle n^S, n^T \rangle$, we match the outgoing edges if and only if $\tau(n^S) = \tau(n^T)$.
- **Adding ϵ-transitions**: If $n^S \notin \mathcal{P}^S$ (implying $\tau(n^S) = ua$), we match the source assignment edge with an ϵ-transition on the target. The case of $n^T \notin \mathcal{P}^T$ is handled analogously.
- **Raising error**: If none of the rules are applicable to a node $\langle n^S, n^T \rangle$, `matchEdges` returns `NULL`, and the construction of $\mathcal{C}$ is aborted.

We always match read, write, and function call edges if both systems can take such a step. Guarded assignment edges can also be composed with each other; but we require that either both systems are currently at a branch node (or a loop head depending on the desired granularity) or none. Since the input systems are consonant, this condition allows to align the corresponding source and target cut points. The case when only one of systems has reached a cut point is covered

via ϵ-transitions, so that it can wait for the other system to catch up. Note that since ϵ is only composed with unconditional assignments, it is guaranteed that the comparison system does not contain an ϵ-cycle, so the wait always is finite. Finally, we fail when both systems are at cut point nodes n^S, n^T but $\tau(n^S) \neq \tau(n^T)$. For example, one system is ready to read while the other is about to execute a procedure call.

Branch Alignment: Consider the case when $\tau(n^S) = \tau(n^T) = br$. By the first rule of `matchEdges`, the outgoing edges should be matched. However, there is an obvious efficiency problem with simply taking all possibilities (i.e., cartesian product) when we consider two nodes with multiple outgoing assignment edges. Such straightforward approach may lead to a number of edges in f being quadratic in the number of edges in the input graphs. More importantly, if we mismatch the branches, unreachable nodes could be introduced into the graph, which may lead to further misalignment down the road. In particular, read, write, and function call edges may get out of sync. Consider the example in the figure below. Suppose $C = c$, $X = x$, and $Y = y$. Then f^T is a correct translation of f^S. However, if we compose edges $(0, 1)$ and $(4, 6)$ just relying on the fact that they are both conditional assignments ($\tau(0) = \tau(4) = br$), the algorithm presented so far will raise an error when examining the newly added unreachable node $\langle 1, 6\rangle$. Thus, there is a need for comprehensive branch matching. One such method is presented below; in addition to resolving the misalignment issue, it usually constructs a comparison graph linear in the size of the input graphs.



Assume that we have an algorithm *InvGen*(f_k) that, given a partially constructed graph f_k, obtained after the k^{th} iteration of `compose`, outputs a *set of invariants* $\{\ \varphi_n^k \mid n \in \textit{locations of } f_k\}$. We will use these invariants to facilitate the edge matching at iteration $k + 1$ so that the composed edges that would introduce infeasible paths are ruled out. Let $\mathcal{E}_n^S$ represent the set of source edges outgoing from n^S s.t. each edge $e^S \in \mathcal{E}_n^S$ is labeled by $c^S \ \rightarrow\ [\vec{u^S} := E^S(\vec{y})]$. Similarly, we define $\mathcal{E}_n^T$.

A pair $(e^S, e^T) \in \mathcal{E}_n^S \times \mathcal{E}_n^T$ *is matched if and only if*
- it does not yet belong to the comparison graph and
- $(\varphi_n^k \ \wedge\ c^S \ \wedge\ c^T)$ *is satisfiable.*

We only want to add an edge if there exists an execution through f_k in which e^S and e^T are enabled simultaneously. An important question to ask is how the decision made using a partially constructed graph f_k relates to the fully constructed graph f. Let φ_n^{fix} be the invariant which can be obtained by running *InvGen* on the fully constructed comparison graph f. Invariant φ_n^k is an under-approximation of φ_n^{fix}, meaning, for some assertion Φ_n^k, $\varphi_n^k \ =\ (\varphi_n^{fix} \wedge \Phi_n^k)$.

Lemma 1 *No spurious predictions are possible: if the match* (e^S, e^T) *is made with* φ_n^k*, it also complies with* φ_n^{fix}*.* As a practical consequence, algorithm `compose` never has to remove any of the previously added edges; thus, it never backtracks. The converse does not hold: we may discover more matches with invariant φ_n^l, where $l : l > k$ is a later iteration of algorithm `compose`. For this reason, the algorithm adds n_e and its decedents to the `WorkList` (see Fig. 2).

Theorem 2 The following are properties of algorithm `compose`:

- *Termination:* algorithm `compose` terminates.
- *Soundness:* if algorithm `compose` succeeds, the resulting graph $f = f^S \boxtimes f^T$ satisfies all of the requirements presented in Section 3.1.
- *Completeness:* if f^T and f^S are consonant, `compose` succeeds in construction of a comparison graph $f = f^S \boxtimes f^T$ given a strong enough *InvGen*.

Note that the completeness of the algorithm is conditional on strength of *InvGen* algorithm used for branch matching. As we show in [20], even for consonant graphs the invariant may need to be strong enough to express reachability, which is undecidable for infinite state systems. All hope is not lost: it is usually feasible to construct the invariants sufficient for our particular application - verification of compiler transformations, intuitively, due to the fact that compilers base their decisions on automated reasoning. The proof of the theorem and the discussion on the *InvGen* used in practice can be found in [20].

5 Example

In this section, we present an example that demonstrates application of `compose` algorithm to comparison system construction along with the generated invariants and `Witness Verification Conditions`. Consider Fig. 3. The first two graphs depict the *source* transition graph and the *target* obtained from the source after constant copy propagation, if simplification, loop invariant code motion, reassociation, and instruction scheduling. Cut point nodes are denoted by double circles. We use capital letters to denote the source variables and their lowercase counterparts for the target. MEM and mem denote the memory heaps. The procedure first reads in two elements - one is stored in register K and the other one in memory at address A. Then, ten elements of the array, stored starting at address P, are being assigned to. Finally, the first element of the array is printed out. We assume that the addresses of the array elements do not overlap with A.

After the third iteration of the algorithm `compose` (from Section 4), we obtain graph $\mathcal{C}_3$ (Comparison 3), which is constructed as following. On the first iteration, an assignment of the source is matched up with an ϵ-transition on the target. On the second iteration, node $\langle 1, 0\rangle$ is considered, and since both procedures are ready to execute reads, the composed read edge is added. Next, we examine node $\langle 2, 1\rangle$. Since only the source procedure has reached a cut point, it waits for the target system to catch up by taking an ϵ-transition. On the fourth iteration, node $\langle 2, 2\rangle$ is considered for the first time and the algorithm $InvGen(\mathcal{C}_3)$ returns $\varphi^3_{\langle 2,2\rangle} : (I = i = 1)$, which is used to align the branches of the loop and obtain

$\mathcal{C}^4$ (Comparison 4). However, $\varphi^3_{\langle 2,2\rangle} \ \wedge \ (I \geq 10) \ \wedge \ (i \geq 10)$ is unsatisfiable. Thus, the matching of the loop exit edges is ruled out by the invariant. At the end of the fourth iteration, node $\langle 2,2\rangle$ is added to the `WorkList` again. Notice that $\varphi^3_{\langle 2,2\rangle}$ does not hold in system $\mathcal{C}^4$ since I and i are updated in the loop, so $InvGen(\mathcal{C}^4)$ widens the invariant, resulting in $\varphi^4_{\langle 2,2\rangle} : \ (I = i)$, which allows to match up the loop exit edges ($\varphi^4_{\langle 2,2\rangle} \ \wedge \ (I \geq 10) \ \wedge \ (i \geq 10)$ is satisfiable). Finally, we match the write edges and obtain $\mathcal{C} = \mathcal{C}_6$ (Comparison 6).

After the comparison system is constructed, we generate the `Witness Verification Condition` to check that both systems write out the same values (following the rules from Section 3.2):

$$\varphi^{fix}_{\langle 3,3\rangle} \rightarrow (MEM[P] = mem[p]).$$

An *inductive* invariant network for the comparison program is presented below. The validity of the verification condition directly follows from $\varphi^{fix}_{\langle 3,3\rangle}$.

$$\begin{aligned}
&\varphi^{fix}_{\langle 0,0\rangle} : \ (MEM = mem) \ \wedge \ (A = a) \ \wedge \ (P = p) \ \wedge \ (A \notin [P..P+9]) \\
&\varphi^{fix}_{\langle 1,0\rangle} : \ (I = 1) \ \wedge \ (C = 5) \ \wedge \ \varphi^{fix}_{\langle 0,0\rangle} \\
&\varphi^{fix}_{\langle 2,1\rangle} : \ (K = k) \ \wedge \ \varphi^{fix}_{\langle 1,0\rangle} \\
&\varphi^{fix}_{\langle 2,2\rangle} : \ (I = i) \ \wedge \ (u = (MEM[A] + C) * K) \ \wedge \ (C = 5) \ \wedge \ \varphi^{fix}_{\langle 0,0\rangle} \\
&\varphi^{fix}_{\langle 3,3\rangle} : \ (MEM = mem) \ \wedge \ (P = p)
\end{aligned}$$

$\varphi^{fix}_{\langle 0,0\rangle}$ asserts our assumptions that at the entry to the programs, the memory heaps of the source and target are the same; the corresponding address variables of the two systems are equal; and the address variable $A(a)$ does not overlap with the addresses of the elements of the source(target) array. The most interesting invariant is $\varphi^{fix}_{\langle 2,2\rangle}$, which asserts that, after each loop iteration, the source and target heaps stay equivalent. It's validity is ensured by the following facts: the addresses at which memory is updated are equal (due to $I = i \ \wedge \ P = p$); the expressions stored at those addresses are equal (since $u = (MEM[A] + C) * K \ \wedge \ I = i$); $MEM[A]$ is not altered by the loop (because $A \notin [P..P + 10]$).

6 CoVaC Tool

We have developed a prototype tool CoVaC based on the presented methodology and used it to verify the optimizations performed by LLVM compiler [2]. The tool has been developed in C++ and uses LLVM data structures for program representation and parsing. It's current line count is at approximately 7,000.

One of the main focuses of the tool is balancing precision and efficiency. To achieve this balance, CoVaC generally utilizes a two-phase strategy: first, it applies fast lightweight analyses, and then, resorts to deep and precise analyses. A good illustration of the two-phase approach is expression equivalence checking, which is one of the central subgoals of our framework. We employ a fast but imprecise value numbering algorithm during the first phase. When value numbering is inconclusive, we resort to assertion checking - a static program verification technique based on computation of weakest-precondition [6].

We have tested the tool performance on a set of C programs compiled from the CoVaC feature tests, the LLVM test suite, and third party implementation of several algorithms (such as in-place heapsort, binary search, print first N primes, etc.), with the total line count of approximately 1K, which corresponds to 2K lines of LLVM bytecode. On average, when validating highly optimized code: 0.53 optimizations per line, CoVaC spends 1 second per every 41 lines, assuming no compiler collaboration. The most time is spent on calls to assertion checker, which is dispatched once per every 8 lines. The prototype's performance provides a strong evidence that a practical validator can be constructed, especially taking into account that, unlike compiler, the tool is used few times per program's lifetime. The most notable exception is application of the framework to compiler testing. However, in such a case, CoVaC can be called after every optimization path rather then after a complete run, and verification of lightly optimized code is much faster since value numbering is sufficient for resolving most of the equivalence checks.

7 Related Work and Conclusion

Our approach can be seen as application of bisimulation equivalence [14] to translation validation and checking equivalence of infinite state programs. Good examples of existing general translation validation frameworks are [7, 22, 17], [18], and [13]; all present program analysis and proof rules specialized to program equivalence checking. [7, 22, 17] and [18] rely on compiler debug information to guide their effort. [13] attempts to eliminate the dependency; however, the compiler annotations are the essential part of its heuristics for branch matching. Even though CoVaC can benefit from compiler annotations in the same way as the existing approaches, it does not require any. [17] focuses on handling interprocedural optimizations; and tools presented in [22] and [18] provide additional rules for loop reordering optimizations (loop interchange, fusion, etc.), which do not preserve consonance. Similar extentions can be incorporated into our general framework; however, their application to non-cooperative compilers is a topic of future research. [11] describes a complete method (no false alarms) for translation validation specialized to register allocation and spilling. For a comprehensive survey on compiler verification in general, refer to [5].

We presented a framework for checking program equivalence based on construction of a cross-product system, which reduces the problem to verification of a single program and allows for utilization of the existing of-the-shelf program analyses and tools. In particular, we have shown how the CoVaC framework can be applied to verification of non-cooperative compilers and used it on practice to validate a wide range of optimizations performed by an aggressive modern compiler, LLVM[2]. Many interesting questions remain. For example, we plan to extend our method to support interprocedural optimizations. We are also interested in investigating application of the CoVaC framework to development of a self-certifying compiler and validation of language-based security properties, specifically, checking conformance with information flow policies [4].

References

1. CVC3: An Automatic Theorem Prover for Satisfiability Modulo Theories (SMT). http://www.cs.nyu.edu/acsys/cvc3/.
2. The LLVM Compiler Infrastructure Project. http://llvm.org/.
3. The yices smt solver.
4. G. Barthe, P. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *Computer Security Foundations Workshop (CSFW)*, 2004.
5. Maulik A. Dave. Compiler verification: a bibliography. *SIGSOFT Softw. Eng. Notes*, 2003.
6. E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.
7. Y. Fang. *Translation Validation of Optimizing Compilers*. PhD thesis, New York University, 2005.
8. R.W. Floyd. Assigning meanings to programs. In *Symposia in Applied Mathematics*, volume 19:19-32, 1967.
9. S. Gulwani and G. Necula. Global value numbering using random interpretation. In *POPL*, pages 342–352. ACM, January 2004.
10. Y. Hu, C. Barrett, B. Goldberg, and A. Pnueli. Validating more loop optimizations. In *Proceedings of the* 4^{th} *International Workshop on Compiler Optimization meets Compiler Verificaiton*, 2005.
11. Y. Huang, B. R. Childers, and M. Soffa. Catching and identifying bugs in register allocation. In *SAS*, 2006.
12. C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization*, 2004.
13. G. Necula. Translation validation for an optimizing compiler. In *PLDI*. ACM Press, 2000.
14. D. Park. Concurrency and automata on infinite sequences. In *Theoretical Computer Science*, pages 167–183, 1981.
15. A. Pnueli. Verification of procedural programs. In *We Will Show Them! Essays in Honour of Dov Gabbay, Volume Two*, pages 543–590. College Publications, 2005.
16. A. Pnueli, M. Siegel, and E. Singerman. Translation validation. *Lecture Notes in Computer Science*, 1384:151+, 1998.
17. A. Pnueli and A. Zaks. Translation validation of interprocedural optimizations. In *Proceedings of the 4th International Workshop on Software Verification and Validation (SVV 2006)*.
18. X. Rival. Symbolic transfer function-based approaches to certified compilation. In *POPL*, 2004.
19. L. Simpson. *Value-Driven Redundancy Elimination*. PhD thesis, Rice University, 1996.
20. A. Zaks and A. Pnueli. Covac: Compiler validation by program analysis of the cross-product. Technical report, NYU, 2007. http://cs.nyu.edu/acsys/publications.html.
21. L. Zuck, A. Pnueli, Y. Fang, and B. Goldberg. Voc: A methodology for the translation validation of optimizing compilers. In *Journal of Universal Computer Science*, 2003.
22. L. Zuck, A. Pnueli, B. Goldberg, C. Barrett, Y. Fang, and Y. Hu. Translation and run-time validation of loop tranformations. In *Formal Methods in System Design*, volume 27(3), pages 335–360, 2005.

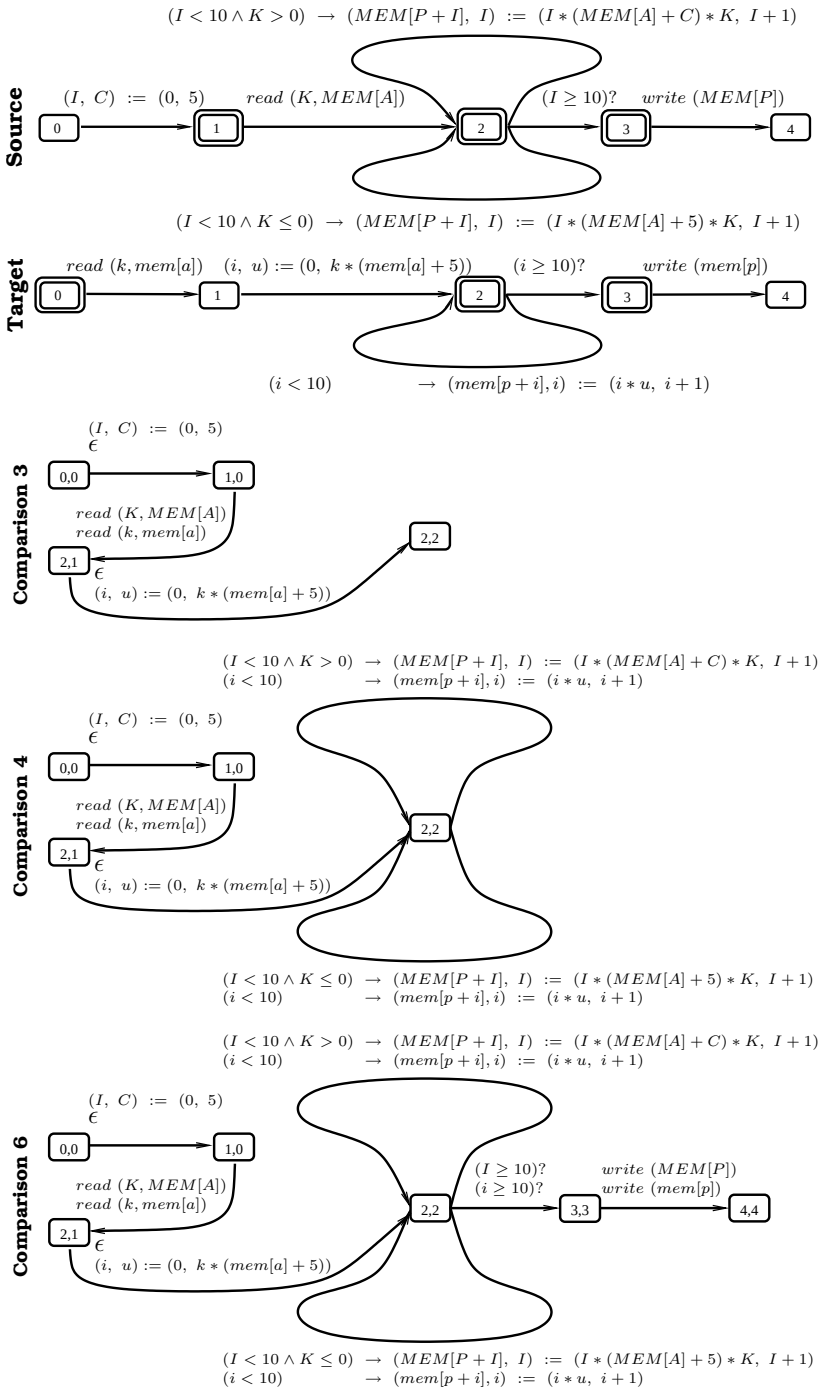


Fig. 3. Source and Target are the input transition graphs: the programs before and after the optimization, respectively. Next, we depict the comparison graphs obtained at the three stages of the comparison graph construction.