Validation of Interprocedural Optimizations

Amir Pnueli\textsuperscript{a,1}, Ganna Zaks\textsuperscript{a,1}

\textsuperscript{a} Computer Science Department
New York University
New York, USA

Abstract
Translation validation is an approach of ensuring compilation correctness in which each compiler run is followed by a validation pass that proves that the target code produced by the compiler is a correct translation of the source code. We present a framework for translation validation of compiler optimization run that targets reactive procedural programs. Our algorithm is automatic and accommodates most classical interprocedural optimizations such as global constant propagation, inlining, tail-recursion elimination, interprocedural dead code elimination, dead argument elimination, and cloning.

Keywords: compiler verification, translation validation, interprocedural optimizations, program equivalence, deductive verification, formal methods

1 Introduction
The effort of program correctness verification is extensive. First, the programmer examines the code and tests it, usually with compiler optimizations turned off. Then, numerous verification tools and techniques can be applied to verify that the code satisfies the desired properties. After all the rigorous checks are complete, it is compiled by an optimizing compiler and released. Nevertheless, our verification effort should not stop here. Compilers are quite large applications, which are bound to have bugs. At present, the GCC Bug Database contains 3217 reported bugs. Clearly, it is highly desirable to ensure that the transformations performed by a compiler preserve the semantics of a program.

The methodology of compiler verification can be categorized by its intended customers. Compiler writers are interested in methods that lead to creation of a self-certifying compiler and may assume full knowledge of the inner workings of a particular compiler. Another group interested in compiler verification are compiler users who may need to work with a black box and require tools that insist on minimal compiler cooperation. Good examples of such tools are presented in [11], [16,17,8], and [14]. The tools are based on the technique of translation validation.

\textsuperscript{1} Email: \{amir, ganna\}@cs.nyu.edu
they check the result of each compilation against the source program, and rely on heuristics and available compiler annotations, usually the debug information, to detect the transformations that take place. The frameworks assume minimal cooperation from the compiler; however, they are also well suited for development of a self-certifying compiler, thus, contributing to the first direction stated above as well. Translation validation algorithms can be generally viewed as a special case of program equivalence checking. Furthermore, since the input programs are infinite state systems, deductive techniques are applied. Specifically, [16] is a generalization of the Floyd method [7] in which a set of assertions is associated with the locations of the source and target programs. Next, a set of verification conditions is generated. The validity of the verification conditions implies that the assertions hold and that the target is a correct translation of the source. To the best of our knowledge, the existing translation validation approaches are not capable and were not designed to deal with interprocedural optimizations. For example, in [11] two executions are considered the same if both lead to the same sequence of function calls and returns. [16] does not model programs with procedures and only considers deterministic programs with no intermediate inputs and outputs.

The main contribution of this paper is a novel translation validation algorithm capable of checking correctness in presence of interprocedural optimizations. Specifically, our framework is an extension of [16] to programs with procedures. In contrast to [16], which used transition systems as the formal model, we rely on transition graphs, which capture not only conditions and assignments but also procedure calls and I/O operations. The notion of correct translation has also been generalized - the target program $T$ is a correct translation (refinement) of program $S$ if every observation of $T$ is also an observation of $S$. An observation of a program is similar to a program computation. However, it only captures the essential information, for example, the values of the variables used in I/O instructions. Finally, the paper presents methodology for generation of auxiliary invariants used for verification of context sensitive copy propagation and presents Interprocedural Translation Validation algorithm that, in addition to the transformations covered by [16], is strong enough to handle most, if not all, of the interprocedural optimizations described in literature [10,4] and performed by optimizing compilers (GCC, ORC, LLVM [2]), like global constant propagation, inlining, tail-recursion elimination, interprocedural dead code elimination, dead argument elimination, and cloning. The main restriction of [16] and, consequently, of the extended approach is that they assume that there exists a mapping from the loops(cut-points) of $T$ to the loops of $S$. However, most of the classical compiler optimizations such as constant folding, induction variable optimizations, branch optimizations, common subexpression elimination, inlining, tail recursion elimination, and others, preserve this property. Rules for loop reordering transformations can be additionally applied to verify transformations such as loop interchange, fusion, tiling, and others, see [17] for further details.

The paper is organized as following. Section 2 presents our formal model, the notion of correct translation, and inductive assertion networks. Section 3 describes the Translation Validation algorithm. Section 4 and Section 5 present the generation of the auxiliary invariants and the translation verification conditions, respectively. We give a comprehensive example in Section 6 and conclude in Section 7.
2 Preliminaries

2.1 Formal Model of Transition Graphs

Our model is similar to that presented in [13] for verification of procedural programs. A program (application) $A$ consists of $m + 1$ procedures: $\text{main}, f_1, \ldots, f_m$, where $\text{main}$ represents the main procedure, and $f_1, \ldots, f_m$ are procedures which may be called from $\text{main}$ 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 $f_i := (\vec{y}, N_i; E_i)$ with variables $\vec{y}$, nodes (locations) $N_i = \{r^i = n^i_0, n^i_1, n^i_2, \ldots, n^i_k = t^i\}$ and a set of labeled edges $E_i$. 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 $g(E(\vec{y}), \vec{u})$ denotes a call to procedure $g(\vec{x}_g; \&\vec{z}_g)$, passing input parameters $E(\vec{y})$ by value and $\vec{u}$ by reference.
- Read and write instructions are denoted by $\text{read}(\vec{u})$ and $\text{write}(\vec{u})$. They are used to express the interaction of the procedure with the outside world; e.g. I/O.

Consider a pair of nodes $(i, j)$ connected by either procedure call, read, or write edge. With no loss of generality, we assume that this edge is the only edge connecting $i$ and $j$. Note that deterministic and non-deterministic branching can be expressed through the use of the guarded assignment instruction. The transition graphs represent a deterministic system when, for every node $i$, the guards of all edges departing from $i$ are mutually exclusive.

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 cut-points $C$ to be a set of program locations such that:

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

The choice of $C$ 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 [8]. Each procedure (or function) whose implementation is given is represented by a transition graph. We choose the set $C$ of a procedure $f_i$ to be
the set of nodes for the corresponding transition graph. For every pair of locations
\( n, m \) in \( C \), if there exists a path \( \pi \) from \( n \) to \( m \), which does not pass through any
other location from \( C \), we add edge \( (n, m) \) to the graph and label it by the instruction
that summarizes the effect of executing the path \( \pi \). Each call to a procedure
whose implementation is hidden can be modeled by read/write instructions. If a
hidden procedure is stateless and does not perform I/O operations (for example,
\textit{pow} function in C), the call is modeled by uninterpreted functions. Note that global
variables and functions also can be efficiently modeled in this framework.

For example, the procedure \textit{MAIN} depicted on Fig. 1 reads in a natural number
\( A \) and writes out the expression \( 3 \times A! + 5 \). It calls a recursive procedure \textit{FAC}
to compute the factorial. \textit{FAC} takes argument \( X \) by value and \( Z \) by reference and
computes \( Z \times X! \), which is returned to the caller by reference.

\begin{verbatim}
read (A) (B, C) := (1, 5)
write (B + C)
\end{verbatim}

Fig. 1. Transition graphs for the program that on input \( A \), outputs \( 3 \times A! + 5 \).

2.2 States and Computations

We denote by \( \vec{d} = (\vec{x}; \vec{z}; \vec{w}) \) 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 \( \text{ret} \). 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 \textit{balanced}. See
our technical report [?] for the formal definition. We define a set of computations
of program \( A \) to be the set of computations of \textit{main}.

2.3 Correct Translation

The notion of correct translation used in this work is based on the general notion of
refinement between source(abstract) program \( S \) and target(concrete) program \( T \).
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 \( S \) and the target program \( T \), must produce the same output.

Given a computation, we define \( V_s \) - the set of \textbf{observable variables} at a state
\( s = \langle n, \vec{d} \rangle \), to be the minimal set satisfying the following two conditions. First,
if \( s \) is a state immediately after transition \( \text{read}(\vec{u}) \), \( V_s \supseteq \vec{u} \). Second, if \( s \) is a
We say that the target program $\phi_t$ for each procedure $f$ with the entry location $r$, we denote $\phi_r$ by $p_f$. The input predicate $p_f = p_f(\vec{X}, \vec{Z}; \vec{x}, \vec{z})$ imposes constraints only on the input variables of the module. Since we assume that the main module main does not have input parameters, $p_0 = true$.

• Similarly, we denote $\phi_t$, the assertion associated with the exit location of $f$, by $q_f$. The output predicate $q_f = q_f(\vec{X}, \vec{Z}; \vec{z})$ is the procedure summary: it specifies the relation between the input and output values.

2.4 Inductive Assertion Network

We introduce virtual variables $\vec{X}$ and $\vec{Z}$ to represent the values of the input variables $\vec{x}$ and $\vec{z}$ at the procedure entry and denote the extended vector of variables by $\vec{Y} = (\vec{X}, \vec{Z}, \vec{x}, \vec{z}, \vec{y})$. An assertion network associates an assertion $\phi_l$ with each program location $l$. 

• For each procedure $f$ with the entry location $r$, we denote $\phi_r$ by $p_f$. The input predicate $p_f = p_f(\vec{X}, \vec{Z}; \vec{x}, \vec{z})$ imposes constraints only on the input variables of the module. Since we assume that the main module main does not have input parameters, $p_0 = true$.

• Similarly, we denote $\phi_t$, the assertion associated with the exit location of $f$, by $q_f$. The output predicate $q_f = q_f(\vec{X}, \vec{Z}; \vec{z})$ is the procedure summary: it specifies the relation between the input and output values.
The assertions at all other locations $\varphi_l(\vec{Y})$ may depend on any of the variables.

For each edge of the transition graph $e$ connecting node $i$ to node $j$, we form **verification conditions**, which represent different edge types:

- **Guarded Assignment:** If $e$ is an assignment edge labeled by $c \rightarrow [\vec{u} := E(\vec{y})]$, 
  $$\forall \mathcal{C}_e : \ \varphi_i(\vec{Y}) \land c(\vec{y}) \rightarrow \varphi_j(\vec{Y})[\vec{u} \mapsto E(\vec{y})],$$
  where $\varphi_j(\vec{Y})[\vec{u} \mapsto E(\vec{y})]$ is obtained from $\varphi_j(\vec{Y})$ by replacing variables in $\vec{u}$ by the corresponding expressions in $E(\vec{y})$.

- **Read:** If $e$ is a read edge labeled by $\text{read}(\vec{u})$, 
  $$\forall \mathcal{C}_e : \ \varphi_i(\vec{Y}) \rightarrow \varphi_j(\vec{Y})[\vec{u} \mapsto \vec{u'}],$$
  where $\vec{u'}$ is a vector of fresh variables. Intuitively, the assertion $\varphi_j$ must hold for all possible inputs.

- **Write:** If $e$ is a write edge labeled by $\text{write}(\vec{u})$, 
  $$\forall \mathcal{C}_e : \ \varphi_i(\vec{Y}) \rightarrow \varphi_j(\vec{Y}).$$

- **Procedure call:** We associate the following two conditions with a procedure call $f(E(\vec{y}), \vec{u})$, which calls the procedure with signature $f(\vec{x}_f; & \vec{z}_f)$:
  $$\mathcal{C}_{\text{call}} : \ \varphi_i(\vec{Y}) \rightarrow p_f(E(\vec{y}), \vec{u}; E(\vec{y}), \vec{u})$$
  $$\mathcal{C}_{\text{return}} : \ \varphi_i(\vec{Y}) \land q_f(E(\vec{y}), \vec{u}; \vec{z}_f) \rightarrow \varphi_j(\vec{Y})[\vec{u} \mapsto \vec{z}_f]$$
  Note that $p_f$ and $q_f$ are the input and output predicates of $f$. Thus, $\mathcal{C}_{\text{call}}$ checks that the assertion associated with the location before the call, $\varphi_i$, implies the input predicate of the callee. $\mathcal{C}_{\text{return}}$ checks that the assertion at the location reached immediately after the procedure return is implied by the output predicate and $\varphi_i$. The conditions generally use variables of the caller procedure with the only exception of the variables passed by reference $\vec{z}_f$. This exception allows to disregard the old information about the variables passed by reference, stored by $\varphi_i(\vec{Y})$, and instead rely on $q_f$.

An assertion network $\mathcal{N} = \{\varphi_0, \ldots, \varphi_n\}$ for a program $\mathcal{A}$ is said to be **inductive** if all the verification conditions for all edges in $\mathcal{A}$ are valid. Network $\mathcal{N}$ is said to be **invariant** if for every execution state $\langle l; \vec{d} \rangle$ occurring in a computation, $d \models \varphi_l$. That is, on every visit of a computation of node $l$, the visiting data state satisfies the corresponding assertion $\varphi_l$ associated with $l$.

**Claim 1** Every inductive network is invariant.

### 3 Interprocedural Translation Validation Algorithm

The Interprocedural Translation Validation algorithm is an extension of the rule Validate [17] to reactive procedural programs. Given two procedural programs $\mathcal{S}$ and $\mathcal{T}$, the algorithm generates a proof that the target program $\mathcal{T}$ is a correct translation of the source program $\mathcal{S}$. Let $C^T$ and $C^S$ denote the sets of nodes (cut-points) of $\mathcal{T}$ and $\mathcal{S}$ respectively. We follow the five steps below to check if $\mathcal{T} \subseteq \mathcal{S}$.

**Step 1:** Establish control abstraction $\kappa : C^T \rightarrow C^S$, mapping the target nodes to the source nodes, such that $r$ is the initial location (root of the main module) of $\mathcal{T}$ if and only if $\kappa(r)$ is the initial location of $\mathcal{S}$. The mapping $\kappa$ is total but does not have to be neither surjective nor injective. For example, we allow a non-surjective
mapping to handle a situation when a loop is eliminated as part of dead code elimination. Optimizations such as inlining result in a non-injective control abstraction. Note that the control abstraction not only specifies the mapping between the program locations but also imposes many-to-one correspondence between target and source procedures. For example, consider target procedure $g^T$ with the root node $r$ and the tail node $t$. $g^T$ corresponds to source procedure $G^S$ with the root node $\kappa(r)$ and the tail node $\kappa(t)$.

**Step 2:** Construct sets of target and source auxiliary assertions that form inductive networks $N^T = \{\phi^T_0, \ldots, \phi^T_{|C^T|}\}$ and $N^S = \{\phi^S_0, \ldots, \phi^S_{|C^S|}\}$ for programs $T$ and $S$, respectively. Form verification conditions showing that the networks are invariant, following rules from Section ?? . Add the generated conditions to the set of verification conditions $\mathcal{VC}$.

**Step 3:** Let $V^S$ and $V^T$ denote the sets of variables that belong to programs $S$ and $T$, respectively. Form data abstraction $D = \{\alpha_0, \ldots, \alpha_{|C^T|}\}$ by defining each $\alpha_l(V^S; V^T)$ as a conjunction of equalities of the form $E(v^S) = E(v^T)$ at each target node $l \in C^T$. The data abstraction must be valid at the initial location of $T$: $\alpha_r = \text{true}$. Intuitively, the data abstraction maps the values of target variables at location $l$ to the values of source variables at location $\kappa(l)$.

**Step 4:** Form Translation Verification Conditions, presented in Section 5, for every edge of the target program and add them to the set of verification conditions $\mathcal{VC}$. If there exists an edge of the target program that does not contribute a verification condition, generate ERROR.

**Step 5:** Establish validity of the conditions in $\mathcal{VC}$; generate ERROR otherwise. The ERROR signifies that either an error in translation is detected or we ran into a transformation that is not currently supported.

The methods for data abstraction construction and generation of inductive assertion networks are presented in [6]. Construction of $D$ is based on refining a candidate data abstraction, obtained from the compiler annotations. Generally, each invariant is based on the set of reachable definitions (definitions that must hold at a particular location). We use the data abstraction and invariants constructed by these methods as the foundation and show how to extend them to the interprocedural setting when necessary. To construct the control abstraction, we first generate the set of source cut-points $C^S$ such that they satisfy the minimal requirements stated in Section 2.1. Then, we rely on the compiler annotations to assist in computation of the control abstraction $\kappa$ and $C^T$. Finally, we check the $C^T$ for completeness with respect to the requirements of Section 2.1. The compiler annotations that the methods depend on are also required for debugging, so they are provided by most compilers.

### 4 Strengthening the Source Inductive Network

The inductive network $N^S = \{\phi^S_0, \ldots, \phi^S_{|C^S|}\}$ has to be augmented so that it incorporates the information essential to proving interprocedural optimizations. We are going to use [15] as our interprocedural dataflow analysis algorithm. The algorithm is precise and has an efficient representation for the internal data that we can use to our advantage. In this section, we show how to generate the source invariant net-
work that is strong enough for context sensitive copy constant propagation. Linear constant propagation can be handled in a similar fashion.

As a first try, it appears that any precise solution to the interprocedural constant-propagation problem should suffice. For example, $\varphi_l$ should be extended with conjunct $x = 17$ if $x$ always evaluates to constant 17 at location $l$. However, the resulting network $N^S$ may not be inductive. Fortunately, the fixpoint based dataflow analysis algorithm not only provides a solution, but also finds a fixpoint for the corresponding set of dataflow equations. We are going to use the information about the fixpoint itself to strengthen our network so it would be inductive.

Let $V$ be the finite set of program variables. Let $L = Z_+^\top$ be the integer constant propagation lattice. We denote the meet operator by $\cap$. The set $\mathit{Env}(V, L)$ of environments is the set of functions from $V$ to $L$. A mapping $T : \mathit{Env}(V, L) \mapsto \mathit{Env}(V, L)$ is called an environment transformer. A transformer $T$ is distributive iff for every variable $v \in V$, $(T(\cap_i \mathit{env}_i))(v) = \cap_i (T(\mathit{env}_i))(v)$. The algorithm in [15] essentially computes a transformer $T_{(r_k, t_k)}$ between the root of each procedure $P_k$ and every location in $L_k$. Note that the transformer $T_{(r_k, t_k)}$ between the root and the tail of $P_k$ is essentially a procedure summary that is represented in our framework by the invariant $q_k$.

Since $T$ needs to operate on functions with infinite domains, the following succinct representation for distributive transformers is used in [15]. Every distributive transformer $T$ can be represented using a set of functions $F_T = \{f_{v, v'} \mid v, v' \in V \cup \{A\}\}$, each of type $L \mapsto L$. Function $f_{v, v'}$ captures the effect that the value of variable $v$ in the argument environment has on the value of $v'$ in the result environment; if $v'$ does not depend on $v$, then $f_{v, v'} = \lambda x.\top$. Function $f_{\Lambda, v'}$ is used to represent the effects of on the variable $v$ that are independent of the argument environment. For any symbol $v'$, the value $T(\mathit{env})(v')$ can be determined by taking the meet of the values of $|V| + 1$ individual function applications: $T(\mathit{env})(v') = f_{\Lambda, v'} \cap (\cap_{v \in V} f_{v, v'}((\mathit{env})(v)))$. Since we are only concerned with constant copy propagation, all the functions in $F_T$ will be either identities or constants.

**Example 1** Consider the example in Fig. 2. Below is the list of environment transformers computed by [15] for procedure $\text{foo}$. We omit all the functions that evaluate to top $f_{v, v'} = \lambda x.\top$.

$$
F_{(2,2)} = \{ f_{x,x} = \lambda x.1, f_{c,c} = \lambda x.1, f_{y,y} = \lambda x.1, f_{z,z} = \lambda x.1 \}
$$

$$
F_{(2,3)} = \{ f_{c,c} = \lambda x.1, f_{y,y} = \lambda x.1, f_{y,z} = \lambda x.1 \}
$$

$$
F_{(2,4)} = \{ f_{c,c} = \lambda x.1, f_{c,z} = \lambda x.1, f_{y,z} = \lambda x.1 \}
$$

Given all the dataflow facts (constants) and the transformer represented by $F_{(i,j)}$, we follow the following rules to compute an invariant $\varphi_l$ at location $l$ of $P_k$:

- We ignore all functions of the form $f_{v, v'} = \lambda x.\top$.
- For each variable $v'$ that is not set to $\bot$ by $f_{(\Lambda, v')} \in F_{(r_k, t_k)}$ we add the following conjunct to $\varphi_l$:

$$
\bigvee_{f_{v, v'} \in F_{(r_k, t_k)}} v' = f_{v, v'}(V), \text{ where } V \text{ represents the value of } v \text{ at the procedure entry.}
$$

We use disjunction to model the effect of the meet operator. In our example, we
use fictitious variables $X$, $C$, $Y$, $Z$ to store the the initial values of $x$, $c$, $y$, $z$.

- We also add the conjunct $x = \text{const}$ if $x$ was determined to evaluate to constant $\text{const}$ at location $l$. We need this addition since $T_{(rk,l)}$ does not propagate the information from the callers.

The resulting invariants, denoted in Fig. 2 by curly brackets, form an inductive network. For example, let’s show that the return verification condition for call edge $(1,5)$ of our example holds.

\[
\forall C_{\text{ret}}: \varphi_1 \land \varphi_4[(C,Y) \mapsto (5,y_m)] \implies \varphi_5[z_m \mapsto z] \iff y_m = 5 \land c = 5 \land (z = 5 \lor z = y_m) \land c = 5 \implies z = 5
\]

5 Translation Verification Conditions

Similarly to the verification conditions used to prove the assertion network inductive, Translation Verification Conditions prove that the data abstraction is inductive on the computations of the target program. They also ensure that source and target observations match given the consistent input. We first give a recipe of generating translation verification conditions when the structure of the transformed program is preserved: for every edge of the target program $e^T$ connecting nodes $i$ and $j$, there exist the corresponding source edges $e^S$ between nodes $\kappa(i)$ and $\kappa(j)$:

- **Guarded Assignment**: If the target edge $e^T$ is a guarded assignment edge of $T$; and $\kappa(i), \kappa(j)$ are also connected by one or more assignment edges in $S$, we generate the following conditions.

\[
\alpha_i \land \varphi^S_i \land \varphi^T_i \land \rho_{cT} \rightarrow (\bigvee_{e^S \in \text{Edges}(\kappa(i),\kappa(j))} c_{eS}),
\]

\[
\alpha_i \land \varphi^S_i \land \varphi^T_i \land \rho_{cT} \land (\bigvee_{e^S \in \text{Edges}(\kappa(i),\kappa(j))} \rho_{eS}) \rightarrow \alpha_j.
\]

In the formulas above, for an edge $e \in \{e^S,e^T\}$ labeled by $c \rightarrow [\bar{u} := E(\bar{y})]$, $c_e$ stands for the condition $c$ and $\rho_e$ for the expression $c \land (\bar{u}' = E(\bar{y}')) \land \bar{v}' = \bar{u}$, where $\bar{v}'$ are all variables of $\alpha_j$ with the exception of those in $\bar{u}$. The first implication checks
that whenever the target transition is enabled, at least one of the corresponding source transitions is also enabled. The second verification condition checks that the data abstraction is preserved by the matching target and source transitions. Invariants $\phi_i^S$ and $\phi_i^T$ are used to strengthen the left-hand-side of the implication.

- **Read**: If $e^T$ and $e^S$ are both labeled by read instructions $\text{read}(u^T)$ and $\text{read}(u^S)$,
  \[ \alpha_i \land \phi_i^T \land \phi_{i(k)}^S \land (u^T = u^S) \rightarrow \alpha_j. \]

- **Write**: If $e^T$ and $e^S$ are both write edges, labeled by $\text{write}(E^T)$ and $\text{write}(E^S)$,
  \[ \alpha_i \land \phi_i^T \land \phi_{i(k)}^S \rightarrow \alpha_j \land (E^T = E^S). \]

Read and write verification conditions ensure that the data mapping implies matching source and target output given the consistent input.

Fig. 3. **Call Verification Conditions**: procedure $G^S$ calls procedure $F^S$ in the source program and procedure $f^T$ calls procedure $F^T$ in the target program.

- **Procedure Call**: If both $e^T$ and $e^S$ are call edges labeled by $f^T( E_g^T; u_g^T )$ and $F^S( E_g^S; u_g^S )$, respectively, where $f^T$ is mapped to $F^S$, we generate Call Verification Conditions presented in Fig. 3, which check that the data abstraction is preserved by stepping through the procedure calls. Similarly to the call conditions of Section ??, the $\mathcal{VC}_{\text{call}}$ condition checks that the data mapping holds at the entry to the procedure; and the $\mathcal{VC}_{\text{ret}}$ condition guarantees that it holds after the procedure return. The right-hand-sides of the implications are strengthened by the auxiliary invariants of the source and target systems; recall that $q_i^S$ and $q_i^T$ are the output predicates of $F^S$ and $F^T$, respectively. If procedure $f^T$ is not mapped to procedure $F^S$, $\text{Error}$ should be generated.

Inlining and Tail-Recursion Elimination (TRE) introduce situations in which the source code contains a call edge that corresponds to a subgraph in the target. In this case, we prove the translation by “stepping into” the procedure call on the
source. Let \( e^T = (i, a) \) be an unconditional assignment edge of the target such that there exists a source call edge \((\kappa(i), \kappa(j))\), labeled by \( F^S(E_G^S; \bar{u}_G^S) \); \( \kappa(a) \) is the entry node of \( F^S \); and there exists the corresponding node \( b \) in the target such that \( \kappa(b) \) is the exit node of the procedure \( F^S \). If \((b, j)\) is an unconditional assignment of \( T \), proceed with inlining verification conditions; otherwise, consider TRE.

\[ \begin{align*}
5.1 & \text{ Inlining} \\
F^S(\text{in} \colon \bar{x}_p^S; \bar{z}_p^S) & \quad g^T(\text{in} \colon \bar{x}_q^T; \bar{z}_q^T) \\
\xymatrix{ & \kappa(a) \ar[r] & \kappa(b) \ar[l] \ar[d] \\
\kappa(i) & 
F^S(E_G^S; \bar{u}_G^S) & \kappa(j) & \cdots \\
& \ar[r] & \ar[u] & \\
\xymatrix{ & a & l & b \ar[l] \\
i & \ar[r] & \ar[u] & \ar[r] & j \\
\alpha_i(\bar{y}_G^S; \bar{y}_b^S) & \alpha_i((v_G^S, \bar{x}_p^S, \bar{z}_p^S); \bar{y}_b^S) & \alpha_b((v_G^S, \bar{x}_p^S, \bar{z}_p^S); \bar{y}_b^S) & \alpha_j(\bar{y}_G^S; \bar{y}_b^S) \\
& [\bar{y}_b^S := EC^T_g] & [\bar{y}_b^S := ER^T_g] & \cdots \\
& \ar[r] & \ar[u] & \\
\mathcal{VC}_{\text{call}}: \alpha_i(\bar{y}_G^S; \bar{y}_b^S) & \wedge \varphi^T(\bar{y}_b^S) & \wedge \varphi^S(\bar{u}_G^S; \bar{u}_G^S) & \rightarrow \alpha_i((v_G^S, \bar{x}_p^S, \bar{z}_p^S); \bar{y}_b^S) \\
\mathcal{VC}_{\text{ret}}: \alpha_b((v_G^S, \bar{x}_p^S, \bar{z}_p^S); \bar{y}_b^S) & \wedge \varphi^T(\bar{y}_b^S) & \wedge \varphi^S(\bar{u}_G^S; \bar{u}_G^S) & \wedge q^T_{\bar{F}}(E_G^S, \bar{u}_G^S; \bar{z}_p^S) & \rightarrow \alpha_j(\bar{y}_G^S; \bar{y}_b^S) \\
\text{where } \bar{y}_G^S = \bar{y}_G^S \setminus \bar{u}_G^S \\
\end{align*} \]

Fig. 4. Inlining Verification Conditions: a call to procedure \( F^S \) is inlined.

Consider a case when a source call edge \( e^S = (\kappa(i), \kappa(j)) \), labeled by \( F^S(E_G^S; \bar{u}_G^S) \), has been inlined. Suppose that the target locations \( i \) and \( j \) belong to some procedure \( g^T \). To simplify this presentation, we assume that there is no nested inlining, so \( e^S \) belongs to \( G^S \) such that \( g^T \) is mapped to \( G^S \). The target procedure should contain unconditional assignment transitions \((i, a)\) and \((b, j)\) that correspond to the call to and return from procedure \( F^S \) on the source. Assume, \((i, a)\) is labeled by \([\bar{y}_b^S := EC^T_g]\) and \((b, j)\) is labeled by \([\bar{y}_b^S := ER^T_g]\), as depicted in Fig. 4.

Define a set of target locations \( L \subset C^T \) such that it includes all locations on every path from \( i \) to \( j \). Note that all the locations in this set will be mapped to the nodes of the source procedure \( F^S \). It is required that \( \alpha_l, l \in L \) does not depend on \( \bar{u}_G^S \), the variables whose references are passed to \( F^S \). However, we do allow the dependence on the corresponding formal parameters. This restriction comes from the fact that \( \bar{u}_G^S \) may change during the execution of \( F^S \), but, for efficiency reasons, the verification conditions from Section 5 work over variables of one procedure at a time. Inlining Verification Conditions, presented in Fig. 4, are generated for each pair of target locations \((i, j)\) that correspond to the inlined call edge \((\kappa(i), \kappa(j))\). The \( \mathcal{VC}_{\text{call}} \) condition checks the data abstraction associated with locations \( a \) and \( \kappa(a) \), which are reached after the assignment on the target and the call on the source; the \( \mathcal{VC}_{\text{ret}} \) condition checks that the data abstraction holds at locations \( j \) and \( \kappa(j) \) - after the corresponding assignment and the return. This ensures that both of the target edges, \((i, a)\) and \((b, j)\), contribute a condition to the set \( \mathcal{VC} \).
5.2 Tail Recursion Elimination

A call edge \((i, t)\) of a procedure \(f(\vec{x}; \& \vec{z})\) is a **TRE candidate** if it is a recursive call labeled by \(f(E(y); \vec{z})\) and \(t\) is the tail node of procedure \(f\). Note that the formal input parameters \(\vec{z}\) are passed as the actual parameters in the tail call. Let \(e^T = (i, r)\) be an unconditional assignment edge of the target procedure \(g^T\) such that there exists a TRE candidate source edge \((\kappa(r), \kappa(t))\) labeled by \(G^S(E^S(SG_{\vec{y}^S}(\vec{z}^S), \vec{z}^S))\), where the target procedure \(g^T\) is mapped to the source procedure \(G^S\). Under these conditions, we guess that TRE optimization occurred and generate TRE Verification Condition, shown in Fig. 5. The condition checks that the data abstraction holds at the entry to the procedure: after a call on the source and the assignment on the target are performed. There is no target edge that corresponds to the return from the recursive call on the source and, consequently, the exit verification condition is not generated. Next, we explain why the requirements of the correct translation, as defined in Section 2.3, are still satisfied. Consider a source computation \(\sigma_S\) that contains \(m\) recursive calls to \(G^S\) and the corresponding target computation \(\sigma_T\):

\[
\begin{align*}
\sigma_S = & \quad \ldots \quad \langle \kappa(r), D^0_r \rangle \quad \longrightarrow \quad \ldots \quad \longrightarrow \quad \langle \kappa(i), D^0_i \rangle \quad \overset{e^S}{\longrightarrow} \\
& \quad \ldots \quad \langle \kappa(r), D^{m-1}_r \rangle \quad \longrightarrow \quad \ldots \quad \longrightarrow \quad \langle \kappa(i), D^{m-1}_i \rangle \quad \overset{e^S}{\longrightarrow} \\
& \quad \langle \kappa(r), D^m_r \rangle \quad \longrightarrow \quad \ldots \quad \longrightarrow \quad \langle \kappa(t), D^m_t \rangle \\
& \quad \overset{\text{ret}}{\longrightarrow} \langle \kappa(t), D^{m-1}_{r} \rangle \quad \ldots \quad \overset{\text{ret}}{\longrightarrow} \langle \kappa(t), D^1_t \rangle \quad \overset{\text{ret}}{\longrightarrow} \langle \kappa(t), D^0_t \rangle \quad \ldots
\end{align*}
\]

\[
\begin{align*}
\sigma_T = & \quad \ldots \quad \langle r, d^0_r \rangle \quad \longrightarrow \quad \ldots \quad \longrightarrow \quad \langle i, d^0_i \rangle \quad \overset{e^T}{\longrightarrow} \\
& \quad \ldots \quad \langle r, d^{m-1}_r \rangle \quad \longrightarrow \quad \ldots \quad \longrightarrow \quad \langle i, d^{m-1}_i \rangle \quad \overset{e^T}{\longrightarrow} \\
& \quad \langle r, d^m_r \rangle \quad \longrightarrow \quad \ldots \quad \longrightarrow \quad \langle t, d^m_t \rangle \\
& \quad \ldots
\end{align*}
\]

\(D^k_r\) and \(d^k_i\) denote source and target data interpretation, where \(k\) stands for the recursion level in the source program and the iteration level in the target program.
First, we want to show that $\alpha_t(\vec{z}_S; \vec{z}_T)$ holds. Validity of the verification conditions generated for all target edges that end in $t$ prove that $\alpha_t(D_m^t; d_m^t)$ holds: $\alpha_t$ holds before we take the very first return transition in the source. Note that the only source variables effecting $\alpha_t(\vec{z}_S; \vec{z}_T)$ are the formal parameters passed by reference that match the actual parameters used for the tail call. Therefore, popping the stack does not change $\vec{z}_S$, and $\alpha_t$ is preserved by the return transitions.

Second, we show that for every target observation, there exists a stuttering equivalent source observation. Consider the observations of the source and target programs $o_S$ and $o_T$ that can be obtained by applying the observation function $O$ to the computations $\sigma_S$ and $\sigma_T$:

$$o_S = \ldots \xrightarrow{\top} O((\kappa(r), D_r^0)) \rightarrow \ldots \rightarrow O((\kappa(i), D_i^0)) \xrightarrow{\top} \ldots$$

$$o_T = \ldots \xrightarrow{\top} O((r, d_r^0)) \rightarrow \ldots \rightarrow O((i, d_i^0)) \xrightarrow{\top} \ldots$$

The verification conditions that we generate for each target edge ensure that for every target transition in $\sigma_T$ there exists a corresponding source transition in $\sigma_S$. Furthermore, the I/O transitions (and the associated data) match. Thus, the source observation $o_S$ can be obtained from the target observation $o_T$ by adding exactly $m$ pairs $\xrightarrow{\top} \top$.

6 Example

The target program depicted in Fig. 6 is obtained from the source program shown earlier in Fig. 1 after TRE is applied to the procedure $fac$, the value of constant $c$ is propagated, and the computation of the expression $a * 3$ is moved due to instruction
scheduling. Note our notation: we use capital letters to denote the source variables
and procedure names; we use the lowercase counterparts in the target program.
Let us apply the Translation Validation algorithm from Section 3 to prove that the
target is the correct translation of the source.
**Step 1:** The control abstraction \( \kappa \) is identity.
**Step 2:** The inductive assertion network \( N^S \) associates assertion \((C = 3)\) with
locations \( l \in \{3, 4, 5\} \) and assertion true with \( l \in \{1, 2, 6, 7, 8\} \). All the assertions in
\( N^T \) are true. We omit the set of the verification conditions that prove inductiveness
of \( N^S \) and \( N^T \) since they are straightforward.
**Step 3:** The following data abstraction is generated:
\[
\begin{align*}
\alpha_1 & : \text{true} \\
\alpha_2 & : (A = a) \\
\alpha_3 & : (A \ast 3 = k) \land (B = b) \\
\alpha_4 & : (A \ast 3 = k) \land (B = b) \\
\alpha_5 & : \text{true} \\
\alpha_6 & : (Z = z) \land (X = x) \\
\alpha_7 & : (Z = z) \land (X = x) \\
\alpha_8 & : (Z = z)
\end{align*}
\]
**Step 4:** Below we list selected translation verification conditions from the set \( \mathcal{VC} \).
We are going to omit the invariants that evaluate to true:
\[
\begin{align*}
\text{Read} & : \mathcal{VC}_{(1,3)} : \alpha_1 \land (A = a) \rightarrow \alpha_2 \iff \text{true} \land (a = A) \rightarrow (a = A) \\
\text{Assign} & : \mathcal{VC}_{(2,3)} : \alpha_2 \land \rho_{(2,3)T} \rightarrow \epsilon_{(2,3)S} ; \alpha_2 \land \rho_{(2,3)T} \land \rho_{(2,3)S} \rightarrow \alpha_3' \iff \\
& (A = a) \land ((b' = 1) \land (k' = a \ast 3) \land (a' = a)) \rightarrow \text{true}; \\
& (A = a) \land ((b' = 1) \land (k' = a \ast 3) \land (a' = a)) \land \\
& ((B' = 1) \land (C' = 5) \land (A' = A)) \rightarrow ((A' \ast 3 = k') \land (B' = b')) \\
\text{Call} & : \mathcal{VC}_{(3,4)} : \mathcal{VC}_{\text{call}} : \alpha_3 \land \varphi_{3}^{S} \rightarrow \alpha_6[(X, Z) \mapsto (A \ast 3, B); (x, z) \mapsto (k, b)]; \\
\text{Write} & : \mathcal{VC}_{(4,5)} : \alpha_4 \land \varphi_{4}^{S} \land \alpha_8 \rightarrow \alpha_4'[B \mapsto Z; b \mapsto z] \iff \\
& ((A \ast 3 = k) \land (B = b)) \land (C = 5) \rightarrow ((A \ast 3 = k) \land (B = b)); \\
& ((A \ast 3 = k) \land (B = b)) \land (C = 5) \land (Z = z) \rightarrow ((A \ast 3 = k) \land (Z = z)) \\
\text{TRE} & : \mathcal{VC}_{(7,6)} : \alpha_7 \rightarrow \alpha_6[\{X \mapsto (X - 1); x \mapsto (x - 1)\}] \iff \\
& ((Z = z) \land (X = x)) \rightarrow ((Z = z) \land (X - 1 = x - 1))
\end{align*}
\]
**Step 5:** We use an automatic theorem prover, such as [3,1], to check the generated
conditions for validity. Since all the conditions in \( \mathcal{VC} \) are valid, we conclude the
correctness of translation.

7 Conclusion and Future Work

We presented the novel framework for automatic translation validation of reactive
programs in presence of interprocedural optimizations. Since all the translation
validation approaches mentioned in Section 1 deal with infinite state systems, they
cannot hope to have a complete method for proving correct translation in general.
However, because the focus is only on compiler optimizations, the number of false
alarms can be drastically minimized or even eliminated. Intuitively, since we are aware of the analysis used by the optimizing compilers, we are optimistic in creation of a strong enough set of auxiliary invariants.

We are currently developing a tool that verifies the optimizations performed by LLVM compiler and uses CVC3 [1] as the back end validity checker. In addition, the framework has yet to be extended to incorporate more language features such as dynamic memory allocation and exceptions.

Another long-term goal of ours is development of a self-certifying compiler or program transformation engine that would allow third parties to specify the desired program transformations. The need for verification is even more obvious here. Therefore, an interesting research direction is construction of the transformation specification language that provides enough information for automatic translation validation. This idea is similar to [9]. However, since our approach is translation validation and we do not insist that all the transformations are provably correct for all the programs, we hope to be more flexible and capture a larger set of program transformations.

References

Appendices

A States and Computations

For simplicity, we assume that all variables of a module range over the same domain $D$ (say, the integers). We denote by $\overrightarrow{d} = (\overrightarrow{d}^x; \overrightarrow{d}^z; \overrightarrow{d}^w)$ a tuple of $D$-values, which represents an interpretation (i.e., an assignment of values) of the module variables.

Definition 2 A state of module $f$ is a pair $\langle l; \overrightarrow{d} \rangle$ consisting of a node $l$ and a data interpretation $\overrightarrow{d}$.

Definition 3 A $(\xi, \zeta)$-computation of module $f$ is a maximal sequence of states and their labeled transitions:

$$\sigma : \langle r; (\xi, \zeta, \overrightarrow{\top}) \rangle \xrightarrow{\lambda_1} \langle l_1; \overrightarrow{d}_1 \rangle \xrightarrow{\lambda_2} \langle l_2; \overrightarrow{d}_2 \rangle \ldots$$

The tuple $\overrightarrow{\top}$ denotes uninitialized values. At the first state of the computation, the location is $r$, the entry location of $f$; the values of input variables $\overrightarrow{x}$ and $\overrightarrow{z}$ are set to $\xi$ and $\zeta$, respectively, and the local variables $\overrightarrow{w}$ are not initialized. Labels in the transitions are either names of edges in the program or the special label $\text{ret}$. Each transition in a computation must be justified by one of the following cases:

- **Guarded Assignment:** There exists an edge $e$ from node $l$ to node $l'$ in the program $\mathcal{A}$ (not necessarily in $f_i$) labeled by $c \rightarrow [\overrightarrow{u} := E(\overrightarrow{y})]$ such that $\overrightarrow{d} \models c$ and $\overrightarrow{d}' = (\overrightarrow{d}$ with $\overrightarrow{u} = E(\overrightarrow{d}))$, i.e., $\overrightarrow{d}'$ is obtained from $\overrightarrow{d}$ by replacing the values corresponding to the variables $\overrightarrow{u}$ by $E(\overrightarrow{d})$.

- **Read:** There exists an edge $e$ in the program $\mathcal{A}$ from node $l$ to node $l'$ labeled by $\text{read}(\overrightarrow{u})$ such that $\overrightarrow{d}_{\overrightarrow{v}} = \overrightarrow{d}_{\overrightarrow{v}}$, where $\overrightarrow{v} = \overrightarrow{y} \setminus \overrightarrow{u}$. $\overrightarrow{d}_{\overrightarrow{v}}$ is obtained from $\overrightarrow{d}$ by restricting it only to the values that correspond to the variables $\overrightarrow{v}$. The values of all variables but the ones in $\overrightarrow{u}$ are preserved by the read transition.

- **Write:** There exists an edge $e$ in the program $\mathcal{A}$ from node $l$ to node $l'$ labeled by $\text{write}(\overrightarrow{u})$. Since a write instruction does not change the values of the variables, $\overrightarrow{d}' = \overrightarrow{d}$.

- **Procedure call:** To justify transition $\langle l; \overrightarrow{d} \rangle \xrightarrow{e} \langle r^k; (E(\overrightarrow{d}), \overrightarrow{d}_{\overrightarrow{u}}, \overrightarrow{\top}) \rangle$, there must exist a call edge $e = (l, l')$ in the program $\mathcal{A}$ labeled by $f_k(E(\overrightarrow{y}), \overrightarrow{u})$, as depicted in Fig. A.1. The location of the new state $r^k$ is the first location in the called procedure $f_k$. $E(\overrightarrow{d})$ and $\overrightarrow{d}_{\overrightarrow{u}}$ are the values of the input variables $\overrightarrow{x}_k$ and $\overrightarrow{z}_k$ on entry to $f_k$. We assume that the working variables are uninitialized.

- **Procedure return:** Finally we consider transition $\langle t^k; (\xi'_k, \zeta'_k, \eta_k) \rangle \xrightarrow{\text{ret}} \langle l'; \overrightarrow{d}' \rangle$. To justify such a transition, there must exist a module $f_k$ (the module from which we return), such that $t^k$ is the terminal location of $f_k$, and we should be able to identify a suffix of the current computation of the form

$$\langle l; \overrightarrow{d} \rangle \xrightarrow{e} \langle r^k; (\xi_k, \zeta_k, \overrightarrow{\top}) \rangle \xrightarrow{\epsilon_1} \ldots \xrightarrow{\epsilon_m} \langle t^k; (\xi'_k, \zeta'_k, \eta_k) \rangle \xrightarrow{\text{ret}} \langle l'; \overrightarrow{d}' \rangle$$
such that the segment $\hat{\sigma}$ is balanced (has an equal number of calls and returns).

We also require that $e$ is a procedure call edge from node $l$ to node $l'$ labeled by $f_k(E(\vec{y}), \vec{u})$ and $\vec{d}' = (\vec{d} \text{ with } \vec{u} = \zeta'_k)$.

Computations of $\textit{main}$ constitute the set of computations of a procedural program $A$.

## B From Programs to Transition Graphs

In this section, we describe how to construct the formal model of a program specified in a standard imperative languages such as C. We also show how to represent the basic language features like global variables and function calls. At present, the framework has yet to be extended to incorporate some language features with most noticeable omissions of dynamic memory allocation and exceptions.

### B.1 Module Nodes as Cut-Points

A set of cut-points is a set of program locations $C$ such that:

- At least one location in each loop belongs to $C$.
- For every procedure, both procedure entry and exit belong to $C$.
- For every procedure call edge $(i,j)$, locations $i$ and $j$ belong to $C$.
- The locations right before and after each read/write operation belong to $C$.

The choice of cut-point set can be generalized not to require at least one cut-point per each loop but to ensure that the transitions between every pair of cut-points are computable \[8\].

Each procedure used in the program, whose implementation is given, is represented by a transition graph. We choose the set of cut-points of a procedure $f_k$ to be the set of nodes for the corresponding transition graph. If there exists a path $\pi$ from cut-point $i$ to cut-point $j$, which does not pass through any other cut-point, we add edge $(i,j)$ to the graph and label it by the instruction that summarizes the effect of executing the path $\pi$. Each call to a procedure whose implementation is hidden can be modeled by read/write instructions. If a hidden procedure is stateless and does not perform I/O operations (for example, $\textit{pow}$ function in C), the call is modeled by uninterpreted functions.
B.2 Global Variables

The global variables are modeled using input parameters. Global variable $v$ is represented by variable $v_k$ in procedure $f_k$. Later in the presentation, we may drop index $k$ when it’s clear from the context. First, for each procedure $f_k$ except for the main module $main$, we compute the set of global variables $M$ that may be modified by the procedure or any of its children. We also compute the set of global variables $U$ whose value may be used by $f_k$ or its children such that $U \cap M = \emptyset$. Second, we add variables of $M$ to the list of input parameters $\vec{z}_k$, which are passed by reference. We add variables of $U$ to the list of input parameters $\vec{x}_k$, which are passed by value. Third, we modify each call to the procedure adding $M$ and $U$ to the argument list. Finally, we add all the global variables to the set of local variables of $main$ and move their initialization inside the scope of $main$.

B.3 Functions

A function call is modeled by a procedure call followed by an assignment:

$$n := f(E(\vec{y}), u) \implies f(E(\vec{y}), (u, n')); \quad n := n'$$

where $n'$ is a fresh variable. The following modifications should be applied to the module $f$. The list of input variables $\vec{z}_f$ should be extended with variable $result_f$, representing the return value of the function. Each edge $(i, j)$ labeled $return E(\vec{y}_f)$ should be replaced by an assignment edge from $i$ to $t_f$ (the exit location of the module $f$) labeled by $result_f := E(\vec{y}_f)$. 