Unité Mixte de Recherche 5104 CNRS - INPG - UJF
Centre Equation
2, avenue de VIGNATE
F-38610 GIERES
tel : +33 456 52 03 40
fax : +33 456 52 03 50
http://www-verimag.imag.fr

Proving Inter-Program Properties

Andrei Voronkov, Iman Narasamdya

Verimag Research Report n^o TR-2008-13

September 2008

Proving Inter-Program Properties

Andrei Voronkov, Iman Narasamdya

The University of Manchester
voronkov@cs.man.ac.uk
Verimag
Iman.Narasamdya@imag.fr

September 2008

Abstract

We develop foundations for proving properties relating two programs. Our formalization is based on a suitably adapted notion of program invariant for a single program. First, we give an abstract formulation of the theory of program invariants based on the notion of assertion function: a function that assigns assertions to program points. Then, we develop this abstract notion further so that it can be used to prove properties between two programs. We describe two applications of the theory. One application is in the translation validation for optimizing compilers, and the other application is in the certification of smart-card application in the framework of Common Criteria. The latter application is part of an industrial project conducted at Verimag laboratory.

Keywords: Assertion Function, Invariant, Translation Validation, Common Criteria Certification

Reviewers: Laurent Mounier

Notes:

How to cite this report:

@techreport { ,
title = { Proving Inter-Program Properties},
authors = { Andrei Voronkov, Iman Narasamdya},
institution = { Verimag Research Report },
number = {TR-2008-13},
year = { },
note = { }
}

1 Introduction

Techniques for proving properties between two programs have become important in the area of program verification. The verification of a program consists of proving that the program satisfies a given specification. The specification is usually written in a formal language such as first-order or temporal logic. However, in some cases, like software evaluation and certification, the formal specification itself is often not available, and we are only given a model of the specification. This model is essentially a program written in a simple language. To prove the correctness of our program, we first formulate the property relating the program and the model. For example, our program is correct with respect to the model if they perform the same sequence of function calls when both of them are run on the same input. Such a property between two programs is called *inter-program property* throughout this report.

Inter-program properties describe relationships between two programs. A relationship between two programs includes a mapping between locations and a relationship between variables of the two programs. Moreover, inter-program properties also involve run-time behaviors of the two programs. Consider the following two programs:

P	P'
$\mathsf{i} := 0$	$\mathsf{i}' := 0$
while $(\mathsf{i} < 100)$ **do**	**while** $(\mathsf{i}' < 100)$ **do**
$\quad \mathsf{i} := \mathsf{i} + 1$	$\quad \mathsf{i}' := \mathsf{i}' + 2$
q :	q' :
od	**od**
return i	**return** i'

We want to prove that P and P' are semantically equivalent. That is, for every pair of runs of both programs, one run is terminating if and only if so is the other, and if the runs are terminating, they return the same value. We first assert $\mathsf{i} = \mathsf{i}'$ at q and q'. Then, we argue that P and P' are equivalent with the following reasoning. From the entries of P and P', by taking two iterations of the loop in P and a single iteration of the loop in P', one can reach q and q' such that the values of i and i' coincide. From q and q', by knowing that the values of i and i' coincide (or the equality $\mathsf{i} = \mathsf{i}'$ holds), then there are two possibilities depending on the values of i and i'. One possibility is follow the same paths as before and reach q and q' again such that the values of i and i' coincide. The other possibility is exit the loops and the values of i and i' remain coincide. These two possibilities show that both runs of P and P' are either terminating or non-terminating. The second possibility shows that on termination, both runs return the same value.

The notion of semantic equivalence is an example of inter-program property. Such a notion is heavily used in compiler verification, particularly in translation validation approach and certifying compilers. In the translation validation approach [11], for each compilation, one proves that the source and the target programs are semantically equivalent. Particularly in a certifying compiler, the compiler must produce a *certificate* certifying such an equivalence.

One might be interested in the notion of safe implementation. For example, a program is a safe implementation of another program if the sequence of observable behaviors performed by the former program is a subsequence of that of the latter program. Consider the programs P and P' above and imagine that there is a function call $\mathsf{f}(\mathsf{i})$ at q and q'. Let function calls and return values be the only observable behaviors. P and P' are no longer equivalent because both perform different sequences of function calls. Nonetheless, one can prove that P' is a safe implementation of P.

Standard techniques for proving properties of a single program have been addressed for four decades [4, 5]. However, although there have been many kinds of inter-program property used in program verification, there is no adequate basis for describing inter-program properties formally such that a rigorous standard is establish for certificates and proofs about such properties. We propose in this report an abstract theory of inter-program properties. The theory is based on the notion of *assertion function*: a function that assigns assertions to program points. For example, in the above program P we can assert that $\mathsf{i} \leq 100$ at q by defining an assertion function I such that I maps q to $\mathsf{i} \leq 100$.

The formalization of our theory is based on a suitably adapted notion of program invariant for a single program. We introduce the notion of extendible assertion function as a constructive notion for describing

and proving program invariants. An assertion function I of a program is extendible if for every run of the program reaching a point p_1 on which I is defined and the assertion defined at p_1 holds, we can always *extend* the run so that it reaches a point p_2 on which I is defined and the assertion at p_2 holds. For example, suppose that we define an assertion function I of the program P above such that, on the entry and exit of P, I is defined as true, and on q, I is defined as $\mathsf{i} \leq 100$. The function I is extendible because if a run reaches q such that $\mathsf{i} \leq 100$ holds, then we can extend the run either to reach q again or to reach the exit of P, and the assertions defined at those points will also hold.

We develop further the notion of extendible assertion function so that it can be used to prove inter-program properties. To this end, we consider the two programs as a *pair of programs* with disjoint sets of variables. For example, to assert that $\mathsf{i} = \mathsf{i}'$ at q and q' in the programs P and P' above, we define an assertion function I of (P, P') such that I maps (q, q') to $\mathsf{i} = \mathsf{i}'$. We will show in this report that meta properties that hold for the case of a single program also hold for the case of a pair of programs. Furthermore, since we are interested in a kind of certificate, we develop a notion of verification condition as a notion of certificate. A verification condition itself is a set of assertions. A certificate can be turned into a proof by proving that all assertions in the verification condition are valid.

In this report we discuss two prominent applications of the theory of inter-program properties. The first application is translation validation. We focus the application on the translation validation for optimizing compilers. We can show that the notion of extendible assertion function can capture inter-program properties used in all existing works on translation validation for optimizing compilers. In Section 5 we will discuss its application to our previous work on finding basic block and variable correspondence [8] and briefly mention how our notion of weakly extendible assertion function and the corresponding notion of verification condition can be used to certify other approaches.

The other application is in software certification. We describe an industrial project for certifying smart-card applications at Verimag laboratory. In this project, we show that, using our theory, we can provide certificates that certify properties between different models of a specification in the framework of Common Criteria [1].

In summary, the contributions of this report are the following:

- A theory of inter-program properties as an adequate basis for describing and proving properties relating two programs.
- Applications of the theory in compiler verification and in software certification.

The outline of this report is as follows. We first describe the main assumptions used in the theory of inter-program properties. We then develop a theory of properties of a single program. We call such properties intra-program properties. Then, we develop the theory further so that it can be used to prove inter-program properties. Having the theory of inter-program properties, we then discuss two applications of the theory in translation validation and in Common Criteria certification.

2 Main Assumptions

Our formalization will be based on standard assumptions about programs and their semantics. We assume that a program consists of a finite set of *program points*. For example, a *program point* of a program P can be the entry or the exit of a sequence of statements (or a *block*) in P. We denote by $\mathbf{Point}_P$ the set of program points of P. A *program-point flow graph of* P is a finite directed graph whose nodes are the program points of P. In the sequel, we assume that every program P we are dealing with is associated with a program-point flow graph, denoted by $\mathbf{G}_P$.

We assume that every program has a unique *entry point* and a unique *exit point*. Denote by $entry(P)$ and $exit(P)$, respectively, the entry and the exit point of program P. We assume that the program-point flow graph contains no edge into the entry point and no edge from the exit point.

We describe the run-time behavior of a program as sequences of configurations. A *configuration* of a program run consists of a program point and a mapping from variables to values. Such a mapping is called a *state*. The variables used in a state do not necessarily coincide with variables of the program. For example, we may consider *memory* to be a variable. Formally, a configuration is a pair (p, σ), where p is a program

point and σ is a state. A configuration (p, σ) is called an *entry configuration for* P if $p = entry(P)$, and an *exit configuration for* P if $p = exit(P)$. For a configuration γ, we denote by $pp(\gamma)$ the program point of γ and by $state(\gamma)$ the state of this configuration.

We assume that the semantics of a program P is defined as a transition relation $\mapsto_P$ with transitions of the form $(p_1, \sigma_1) \mapsto_P (p_2, \sigma_2)$, where p_1, p_2 are program points, σ_1, σ_2 are states, and (p_1, p_2) is an edge in the program-point flow graph of P.

DEFINITION 2.1 (Computation Sequence,Run) A *computation sequence of a program* P is either a finite or an infinite sequence of configurations

$$(p_0, \sigma_0), (p_1, \sigma_1), \ldots, \tag{1}$$

where $(p_i, \sigma_i) \mapsto_P (p_{i+1}, \sigma_{i+1})$ for all i. A *run* R *of a program* P *from an initial state* σ_0 is a computation sequence (1) such that $p_0 = entry(P)$. A run is *complete* if it cannot be extended, that is, it is either infinite or terminates at an exit configuration.

For two configurations γ_1, γ_2, we write $\gamma_1 \overset{*}{\mapsto}_P \gamma_2$ to denote that there is a computation sequence of P starting at γ_1 and ending at γ_2. We say that a computation sequence is *trivial* if it is a sequence of length 1.

We introduce two restrictions on the semantics of programs. First, we assume that programs are deterministic. That is, for every program P, given a configuration γ_1, there exists at most one configuration γ_2 such that $\gamma_1 \mapsto_P \gamma_2$. Second, we assume that, for every program P and for every non-exit configuration γ_1 of P's run, there exists a configuration γ_2 such that $\gamma_1 \mapsto_P \gamma_2$, that is, a complete run may only terminate in an exit configuration. Our results can easily be generalized by dropping these restrictions. Indeed, one can view a non-deterministic program as a deterministic program having an additional input variable x whose value is an infinite sequence of numbers, these numbers are used to decide which of non-deterministic choices should be made. Further, if a program computation can terminate in a state different from the exit state, we can add an artificial transition from this state to the exit state. After such a modification we can also consider arbitrary non-deterministic programs.

Further, we assume some *assertion language* in which one can write *assertions* involving variables and express properties of states. For example, the assertion language may be some first-order language. The set of all assertions is denoted by $\mathbf{Assertion}$. We will use meta variables α, ϕ, φ, and ψ, along with their primed, subscript, and superscript notations, to range over assertions. We write $\sigma \models \alpha$ to mean an assertion α is true in a state σ, and also say that σ *satisfies* α, or that α *holds at* σ. We say that an assertion α is valid if $\sigma \models \alpha$ for every state σ. We will also use a similar notation for configurations: for a configuration (p, σ) and assertion α we write $(p, \sigma) \models \alpha$ if $\sigma \models \alpha$. We also write $\sigma \not\models \alpha$ to mean an assertion α is false in σ, or σ does not satisfy α. We assume that the assertion language is closed under the standard propositional connectives and respects their semantics, for example $\sigma \models \neg\alpha$ if and only if $\sigma \not\models \alpha$. We call an assertion *valid* if it is true in all states.

To ease the readability we introduce the following notation: for all assertions α, α_1, and α_2, and for every state σ,

$\alpha_1 \wedge \alpha_2$	for	α, where $\sigma \models \alpha$ if and only if $\sigma \models \alpha_1$ and $\sigma \models \alpha_2$
$\alpha_1 \vee \alpha_2$	for	α, where $\sigma \models \alpha$ if and only if $\sigma \models \alpha_1$ or $\sigma \models \alpha_2$
$\neg\alpha_1$	for	α, where $\sigma \models \alpha$ if and only if $\sigma \not\models \alpha_1$
$\alpha_1 \Rightarrow \alpha_2$	for	α, where $\sigma \models \alpha$ if and only if $\sigma \models \alpha_2$ whenever $\sigma \models \alpha_1$

3 Intra-Program Properties

In this section we introduce the notion of program invariant for a single program and some related notions that make it more suitable to present inter-program properties later.

3.1 Program Invariants

We introduce the notion of assertion function that associates program points with assertions. An *assertion function* for a program P is a partial function

$$I : \mathbf{Point}_P \to \mathbf{Assertion}$$

mapping program points of P to assertions such that $I(entry(P))$ and $I(exit(P))$ are defined. The notion of assertion function generalizes the notion of program invariant: one can consider I as a collection of invariants associated with program points. The requirement that I is defined on the entry and exit points is purely technical and not restrictive, for one can always define $I(entry(P))$ and $I(exit(P))$ as $\top$, that is, an assertion that holds at every state.

Given an assertion function I, we call a program point p *I-observable* if $I(p)$ is defined. A configuration (p, σ) is called I-observable if so is its program point p. We say that a configuration $\gamma = (p, \sigma)$ *satisfies* I, denoted by $\gamma \models I$, if $I(p)$ is defined and $\sigma \models I(p)$. We will also say that I is defined on γ if it is defined on p and write $I(\gamma)$ to denote $I(p)$.

DEFINITION 3.1 (Program Invariant) Let I be an assertion function of a program P. The function I is said to be a *program invariant* of P if for every run

$$\gamma_0, \gamma_1, \dots$$

of the program such that $\gamma_0 \models I$ and for all $i \geq 0$, we have $\gamma_i \models I$ whenever I is defined on $pp(\gamma_i)$. □

In other words, an assertion function is an invariant if and only if for every program run from an entry configuration satisfying I, every observable configuration of this run satisfies I too.

This notion of invariant is useful for asserting that a program satisfies some properties, including partial correctness of a problem. Recall that a program P is *partially correct* with respect to a precondition φ and a postcondition ψ, denoted by $\{\varphi\}P\{\psi\}$, if for every run of P from a configuration satisfying φ and reaching an exit configuration, this exit configuration satisfies ψ. Likewise, a program P is *totally correct* with respect to a precondition φ and a postcondition ψ, denoted by $[\varphi]P[\psi]$, if every run of P from a configuration satisfying φ terminates in an exit configuration and this exit configuration satisfies ψ.

THEOREM 3.2 *Let P be a program and φ, ψ be assertions. Let I be an assertion function for P such that $I(entry(P)) = \varphi$ and $I(exit(P)) = \psi$. If I is an invariant, then $\{\varphi\}P\{\psi\}$. If, in addition, I is only defined on the entry and the exit points, then I is an invariant if and only if $\{\varphi\}P\{\psi\}$.*

PROOF. Suppose that I is an invariant of P and $\gamma_1 \overset{*}{\mapsto}_P \gamma_2$, where γ_1 is an entry configuration and γ_2 is an exit configuration, and $\gamma_1 \models \varphi$. Then $\gamma_1 \models I$. Using this and the fact that γ_2 is I-observable, we obtain $\gamma_2 \models I$, that is, $\gamma_2 \models \psi$.

Now suppose that I is only defined on the entry and the exit points, and $\{\varphi\}P\{\psi\}$. Consider any complete run of P from a configuration γ_1 that satisfies φ. We have to show that every I-observable configuration of this run also satisfies I. It is obvious that $\gamma_1 \models I$. But the only observable state of this run different from γ_1 may be an exit configuration γ_2, in which case, and by our restrictions on programs, the run terminates at this configuration, then by $\{\varphi\}P\{\psi\}$, we have $\gamma_2 \models \psi$, that is, $\gamma_2 \models I$. □

One can provide a similar characterization of loop invariants using our notion of invariant.

3.2 Extendible Assertion Functions

Our notion of invariant is not immediately useful for *proving* that a program satisfies some properties. For proving, we need a more constructive characterization of relations between I and P than just those expressed by program runs. We introduce the notion of extendible assertion function that provides such a characterization.

DEFINITION 3.3 Let I be an assertion function of a program P. I is *strongly extendible* if for every run

$$\gamma_0, \dots, \gamma_i$$

of the program such that $i \geq 0$, $\gamma_0 \models I$, $\gamma_i \models I$, and γ_i is not an exit configuration, there exists a finite computation sequence

$$\gamma_i, \dots, \gamma_{i+n}$$

such that

1. $n > 0$,
2. $\gamma_{i+n} \models I$, and
3. for all j such that $i < j < i + n$, the configuration γ_j is not I-observable.

The definition of *weakly-extendible* assertion function is obtained from this definition by dropping condition 3. □

EXAMPLE 3.4 Let us give an example illustrating the difference between the two notions of extendible assertion functions. Consider the following program P:

```
i := 0
j := 0
while (i < 100) do
  if (i > j) then j := j + 1
  else i := i + 1
  fi
q :
od
```

Define an assertion function I of P such that $I(entry(P)) = \top$ and $I(q) = I(exit(P)) = (\mathsf{i} = \mathsf{j})$, and $I(p)$ is undefined on all program points p different from q and the entry and exit points. Then I is weakly extendible but not strongly extendible. To show that I is weakly extendible, it is enough to observe the following properties:

1. From an entry configuration, in two iterations of the loop, one reaches a configuration with the program point q in which $\mathsf{i} = \mathsf{j} = 1$;
2. For every $v < 100$, from a configuration with the program point q in which $\mathsf{i} = \mathsf{j} = v$, in two iterations of the loop, one can reach a configuration in which $\mathsf{i} = \mathsf{j} = v + 1$;
3. For every $v \geq 100$, from a configuration with the program point q in which $\mathsf{i} = \mathsf{j} = v$, one can reach an exit configuration in which $\mathsf{i} = \mathsf{j} = v$.

To show that I is not strongly extendible, it is sufficient to note that, from any entry configuration, after one iteration of the loop, one can reach a configuration with the program point q in which $\mathsf{i} = 1$ and $\mathsf{j} = 0$ and so $\mathsf{i} = \mathsf{j}$ does not hold. □

Using the same arguments as in the proof of Theorem 3.2, we can show that weakly-extendible functions are sufficient for proving partial correctness:

THEOREM 3.5 *Let I be a weakly-extendible assertion function of a program P such that $I(entry(P)) = \varphi$ and $I(exit(P)) = \psi$. Then $\{\varphi\}P\{\psi\}$, that is, P is partially correct with respect to the precondition φ and the postcondition ψ.* □

On the other hand, strongly-extendible assertion functions serve as invariants, as the following theorem shows:

THEOREM 3.6 *Every strongly-extendible assertion function I of a program P is also an invariant of P.*

PROOF. We have to show that, for every run $\gamma_0, \gamma_1, \ldots$ of P such that $\gamma_0 \models I$ and every I-observable configuration γ_i of this run, we have $\gamma_i \models I$. We will prove it by induction on i. When $i = 0$, the statement is trivial. Suppose $i > 0$. Take the greatest number j such that $0 \leq j < i$ and γ_j is I-observable. Such a number exists since γ_0 is I-observable. By the induction hypothesis, we have $\gamma_j \models I$. By the definition of strongly-extendible assertion function, we have that there exists an $n > 0$ and a run $\gamma_0, \ldots, \gamma_j, \ldots, \gamma_n$ such that $\gamma_n \models I$ and all configurations between γ_j and γ_n are not I-observable. Note that both γ_i and γ_n are the first I-observable configurations after γ_j in their runs. By the assumption that our programs are deterministic, we obtain $\gamma_i = \gamma_n$, so $\gamma_i \models I$. □

The following theorem shows that for terminating programs there is a closer relationship between strongly-extendible assertion functions and invariants:

THEOREM 3.7 *Let an assertion function I be an invariant of P such that $I(entry(P)) = \alpha$. Let P terminate for every entry configurations satisfying α, that is, every run on an entry configuration satisfying α is finite. Then I is strongly extendible.*

PROOF. Take any run $\gamma_0, \ldots, \gamma_i$ of P such that $\gamma_0 \models I$, $\gamma_i \not\models I$, and γ_i is not an exit configuration. Extend this run to a run $\gamma_0, \ldots, \gamma_{i+n}$ that satisfies the conditions of Definition 3.3. To this end, first extend the run to a complete run

$$R = \gamma_0, \ldots, \gamma_i, \gamma_{i+1}, \ldots.$$

Let us show that R contains a configuration γ_{i+n} with $n > 0$ on which I is defined. Such a configuration exists since R is finite and I is defined on the exit configuration of R. Take the smallest n such that I is defined on γ_{i+n}. Since n is the smallest, I is undefined on all configurations between γ_i and γ_{i+n} in R. Since I is an invariant, we have $\gamma_{i+n} \models I$. □

The condition on programs to be terminating is not very constructive. We will introduce other sufficient conditions on assertion functions which, on the one hand, will guarantee that an invariant is also strongly or weakly extendible, and on the other hand, make our notion of invariant similar to more traditional ones [7]. To this end, we will use paths in the program-point flow graph $\mathbf{G}_P$. Such a path is called *trivial* if it consists of a single point. To guarantee that an invariant I of a program P is strongly extendible, we require that I must be defined on certain program points such that those points break all cycles in $\mathbf{G}_P$. That is, every cycle in $\mathbf{G}_P$ contains at least one of these points. We introduce the notion of covering set to describe this requirement.

DEFINITION 3.8 (Covering Set) Let P be a program and C be a set of program points in P. We say that C *covers* P if $entry(P) \in C$ and every infinite path in $\mathbf{G}_P$ contains a program point in C. An assertion function I is said to *cover* P if the set of I-observable program points covers P. □

Any set C that covers P is often called a *cut-point set* of P.

THEOREM 3.9 *Let I be an invariant of P. If I covers P, then I is strongly extendible.*

PROOF. Take any run $\gamma_0, \ldots, \gamma_i$ of P such that $\gamma_0 \models I$, $\gamma_i \not\models I$ and γ_i is not an exit configuration. We have to extend this run to a run $\gamma_0, \ldots, \gamma_{i+n}$ satisfying the conditions of Definition 3.3. To this end, first extend this run to a complete run $R = (\gamma_0, \ldots, \gamma_i, \gamma_{i+1}, \ldots)$. Let us show that R contains a configuration γ_{i+n} with $n > 0$ on which I is defined. Indeed, if R is finite, then the last configuration of R is an exit configuration, and then I is defined on it. If R is infinite, then the path $pp(\gamma_{i+1}), pp(\gamma_{i+2}), \ldots$ is infinite, hence contains a program point on which I is defined. Take the smallest positive n such that I is defined on γ_{i+n}. Since n is the smallest, I is undefined on all configurations between γ_i and γ_{i+n} in R. Since I is invariant, we have $\gamma_{i+n} \models I$. □

EXAMPLE 3.10 Consider again the program P of Example 3.4. Define an assertion function I_1 of P such that I_1 is defined only on the entry and the exit points, $I_1(entry(P)) = \top$ and $I_1(exit(P)) = (\mathsf{i} = \mathsf{j})$. I_1 is an invariant of P, but does not cover P since it is undefined on all points in the loop. Nevertheless, I_1 is strongly extendible.

Let us now define another assertion function I_2 such that $I_2(entry(P)) = \top$, $I_2(exit(P)) = (\mathsf{i} = \mathsf{j})$, $I_2(q) = (\mathsf{i} > \mathsf{j} \Rightarrow \mathsf{i} = \mathsf{j} + 1)$, and I_2 is undefined on all other points. I_2 is an invariant of P and also strongly extendible. Moreover, I_2 covers P. □

3.3 Verification Conditions

Our next aim is to define a notion of verification condition as a collection of formulas and use these verification conditions to prove properties of programs. We want to define it in such a way that a verification condition guarantees certain properties of programs. To this end, we use the notions of precondition and liberal precondition for programs and paths in program-point flow graphs.

DEFINITION 3.11 (Weakest Liberal Precondition) An assertion φ is called the *weakest liberal precondition* of a program P and an assertion ψ, if

1. $\{\varphi\}P\{\psi\}$, and
2. for every assertion φ' such that $\{\varphi'\}P\{\psi\}$, the assertion $\varphi' \Rightarrow \varphi$ is valid.

In general, the weakest liberal precondition may not exists. If it exists, we denote the weakest liberal precondition of P and ψ by $wlp_P(\psi)$.

In a similar way, we introduce the notion of a weakest liberal precondition of a path $\pi = (p_0, \ldots, p_n)$ in the flow graph. An assertion φ is called a *precondition* of the path π and an assertion ψ, if, for every state σ_0 such that $\sigma_0 \models \varphi$, there exist states $\sigma_1, \ldots, \sigma_n$ such that

$$(p_0, \sigma_0) \mapsto (p_1, \sigma_1) \mapsto \ldots \mapsto (p_n, \sigma_n)$$

and $\sigma_n \models \psi$. An assertion φ is called the *weakest precondition* of π and ψ, denoted by $wp_\pi(\psi)$, if it is a precondition of π and ψ, and, for every precondition φ' of π and ψ, the assertion $\varphi' \Rightarrow \varphi$ is valid.

An assertion φ is called a *liberal precondition* of the path π and an assertion ψ, if, for every sequence $\sigma_0, \ldots, \sigma_n$ of states such that

$$(p_0, \sigma_0) \mapsto (p_1, \sigma_1) \mapsto \ldots \mapsto (p_n, \sigma_n),$$

and $\sigma_0 \models \varphi$, we have $\sigma_n \models \psi$. An assertion φ is called the *weakest liberal precondition* of π and ψ, denoted by $wlp_\pi(\psi)$, if it is a liberal precondition of π and ψ, and, for every liberal precondition φ' of π and ψ, the assertion $\varphi' \Rightarrow \varphi$ is valid. □

Later in proving the correctness of verification condition, we find that the following property of weakest liberal precondition is useful:

COROLLARY 3.12 *Let $\pi = p_0, \ldots, p_n$ be a path, and φ and ψ be assertions. Suppose that there exists a sequence $\sigma_0, \ldots, \sigma_n$ of states such that*

$$(p_0, \sigma_0) \mapsto (p_1, \sigma_1) \mapsto \ldots \mapsto (p_n, \sigma_n),$$

$\sigma_0 \models \varphi$ and $\varphi \Rightarrow wlp_\pi(\psi)$ is valid. Then $\sigma_n \models \psi$.

PROOF. Since $\sigma_0 \models \varphi$ and $\varphi \Rightarrow wlp_\pi(\psi)$ is valid, we have $\sigma_0 \models wlp_\pi(\psi)$. Since $wlp_\pi(\psi)$ is the weakest liberal precondition for π and ψ, we have $\sigma_n \models \psi$. □

Another useful property of weakest preconditions and weakest liberal precondition is that the weakest liberal precondition can be expressed in terms of the weakest precondition.

THEOREM 3.13 *Let π be a path and ψ be an assertion. Then $wlp_\pi(\psi)$ is equivalent to $wp_\pi(\psi) \vee \neg wp_\pi(\top)$.*

PROOF. Let $\pi = (p_0, \ldots, p_n)$. We have to show that, for every state σ, $\sigma \models wlp_\pi(\psi)$ if and only if $\sigma \models wp_\pi(\psi) \vee \neg wp_\pi(\top)$.

($\Rightarrow$) Suppose that $\sigma \models wlp_\pi(\psi)$ for some state σ. Suppose further that there exists a sequence $\sigma_0, \ldots, \sigma_n$ of states such that $\sigma_0 = \sigma$ and

$$(p_0, \sigma_0) \mapsto (p_1, \sigma_1) \mapsto \ldots \mapsto (p_n, \sigma_n).$$

Since the weakest liberal precondition is a liberal precondition, by the definition of liberal precondition, we have $\sigma_n \models \psi$. Hence, by the definition of precondition, the state σ satisfies some precondition φ of π and ψ. By the definition of weakest precondition, the assertion $\varphi \Rightarrow wp_\pi(\psi)$ is valid, and thus we have $\sigma \models wp_\pi(\psi)$.

Suppose that there is no such a sequence $\sigma_0, \ldots, \sigma_n$. By the definition of precondition, any precondition of π and $\top$ is equivalent to $\bot$, and so is $wp_\pi(\top)$. Thus, we have $\sigma \models \neg wp_\pi(\top)$.

($\Leftarrow$) Suppose $\sigma \models wp_\pi(\psi)$ for some state σ. Then, there exist states $\sigma_0, \ldots, \sigma_n$ such that $\sigma = \sigma_0$,

$$(p_0, \sigma_0) \mapsto (p_1, \sigma_1) \mapsto \ldots \mapsto (p_n, \sigma_n),$$

and $\sigma_n \models \psi$. By the definition of liberal precondition, σ satisfies some liberal precondition φ of π and ψ. By the definition of weakest liberal precondition, the assertion $\varphi \Rightarrow wlp_\pi(\psi)$ is valid. Hence, we have $\sigma \models wlp_\pi(\psi)$.

Suppose now that $\sigma \models \neg wp_\pi(\top)$. Since every state satisfies $\top$, the relation $\sigma \models \neg wp_\pi(\top)$ means that there is no sequence $\sigma_0, \ldots, \sigma_n$ of states such that $\sigma = \sigma_0$ and

$$(p_0, \sigma_0) \mapsto (p_1, \sigma_1) \mapsto \ldots \mapsto (p_n, \sigma_n).$$

By the definition of liberal precondition, the state σ satisfies any liberal precondition of π and any assertion. Thus, the state σ also satisfies any liberal precondition φ of π and ψ. By the definition of weakest liberal precondition, the assertion $\varphi \Rightarrow wlp_\pi(\psi)$ is valid. Hence, we have $\sigma \models wlp_\pi(\psi)$. □

We have so far not imposed any restrictions on the programming languages in which programs are written. However, to provide certificates or verification conditions for program properties, we need to be able to compute the weakest and the weakest liberal precondition of a given path and an assertion.

DEFINITION 3.14 (Weakest Precondition Property) We say that a programming language has the *weakest precondition property* if, for every assertion ψ and path π, the weakest precondition for π and ψ exists and moreover, can effectively be computed from π and ψ. □

In the sequel we assume that our programming language has the weakest precondition property. Note that Theorem 3.13 implies that in any such language, given a path π and an assertion ψ, one can also compute the weakest liberal precondition for π and ψ.

Next, we describe the verification conditions associated with assertion functions. Such verification conditions form *certificates* for program properties described by the assertion functions. Let I be an assertion function. A path $p_0, \ldots, p_n$ in $\mathbf{G}_P$ is called *I-simple* if $n > 0$ and I is defined on p_0 and p_n and undefined on all program points $p_1, \ldots, p_{n-1}$. We will say that the path is *between* p_0 and p_n.

DEFINITION 3.15 Let I be an assertion function of a program P such that the domain of I covers P. The *strong verification condition* associated with I is the set of assertions

$$\{I(p_0) \Rightarrow wlp_\pi(I(p_n)) \mid \pi \text{ is an } I\text{-simple path between } p_0 \text{ and } p_n\}.$$

Note that the strong verification condition is always finite.

□

THEOREM 3.16 *Let I be an assertion function of a program P whose domain covers P and $\mathbb{S}$ be the strong verification condition associated with I. If every assertion in $\mathbb{S}$ is valid, then I is strongly extendible.*

PROOF. Take any run $\gamma_0, \ldots, \gamma_i$ of P such that $\gamma_0 \models I$, $\gamma_i \models I$ and γ_i is not an exit configuration. Using arguments of the proof of Theorem 3.9, we extend this run to a run $\gamma_0, \ldots, \gamma_{i+n}$ such that I is defined on γ_{i+n} but undefined on $\gamma_{i+1}, \ldots, \gamma_{i+n-1}$. It remains to prove that $\gamma_{i+n} \models I$.

Consider the run $\gamma_i, \ldots, \gamma_{i+n}$ and denote the program point of each configuration γ_j in this run by p_j and the state of γ_j by σ_j. Then the path $\pi = (p_i, \ldots, p_{i+n})$ is simple and we have $\sigma_i \models I(p_i)$. The assertion

$$I(p_i) \Rightarrow wlp_\pi(I(p_{i+n}))$$

belongs to the strong verification condition associated with I, hence valid, so by Corollary 3.12 we have $\sigma_{i+n} \models I(p_i)$, which is equivalent to $\gamma_{i+n} \models I$. □

Note that this theorem gives us a sufficient condition for checking partial correctness of the program: given an assertion function I defined on a covering set, we can generate the strong verification condition associated with I. This condition by Theorem 3.16 guarantees that I is strongly extendible, hence also weekly extendible. Therefore, by Theorem 3.5 guarantees partial correctness. Moreover, the strong verification condition is simply a collection of assertions, so if we have a theorem prover for the assertion language, it can be used to check the strong verification condition.

One can reformulate the notion of verification condition in such a way that it will guarantee weak extendibility. For every path π, denote by $start(\pi)$ and $end(\pi)$, respectively, the first and the last point of π.

DEFINITION 3.17 Let I be an assertion function of a program P and Π a set of paths in $\mathbf{G}_P$ such that for every path π in Π both $start(\pi)$ and $end(\pi)$ are I-observable. For every program point p in P, denote by $\Pi|p$ the set of paths in Π whose first point is p.

The *weak verification condition* associated with I and Π consists of all assertions of the form

$$I(start(\pi)) \Rightarrow wlp_\pi(I(end(\pi))),$$

where $\pi \in \Pi$ and all assertions of the form

$$I(p) \Rightarrow \bigvee_{\pi \in \Pi|p} wp_\pi(\top),$$

where p is an I-observable point.

The first kind of assertion in this definition is similar to the assertions used in the strong verification condition, but instead of all simple paths we consider all paths in Π. The second kind of assertion expresses that, whenever a configuration at a point p satisfies $I(p)$, the computation from this configuration will inevitably follow at least one path in Π. This informal explanation is made more precise in the following theorem.

THEOREM 3.18 *Let I and Π be as in Definition 3.17 and $\mathbb{W}$ be the weak verification condition associated with I and Π. If every assertion in $\mathbb{W}$ is valid, then I is weakly extendible.*

PROOF. In the proof, whenever we denote a configuration by γ_i, we use p_i for the program point and σ_i for the state of this configuration, and similarly for other indices instead of i.

Take any run $\gamma_0, \ldots, \gamma_i$ of P such that $\gamma_0 \models I$, $\gamma_i \models I$ and γ_i is not an exit configuration. Since p_i is I-observable, the following assertion belongs to $\mathbb{W}$:

$$I(p_i) \Rightarrow \bigvee_{\pi \in \Pi|p_i} wp_\pi(\top),$$

and hence it is valid. Since $\gamma_i \models I$, we have $\sigma_i \models I(p_i)$, then by the validity of the above formula we have

$$\sigma_i \models \bigvee_{\pi \in \Pi|p_i} wp_\pi(\top).$$

This implies that there exists a path $\pi \in \Pi|p_i$ such that $\sigma_i \models wp_\pi(\top)$. Let the path π have the form $p_i, \ldots, p_{i+n}$. Then, by the definition of $wp_\pi(\top)$, there exist states $\sigma_{i+1}, \ldots, \sigma_{i+n}$ such that

$$(p_i, \sigma_i) \mapsto (p_{i+1}, \sigma_{i+1}) \mapsto \ldots \mapsto (p_{i+n}, \sigma_{i+n}).$$

Using that $\pi \in \Pi$ and repeating arguments of Theorem 3.16 we can prove $\sigma_{i+n} \models I(p_{i+n})$. □

4 Inter-Program Properties

In this section we develop further the notion of extendible assertion function so that it can be used to prove *inter-program properties*. Given a pair (P, P') of programs, we assume that they have disjoint sets of variables. A configuration is a tuple $(p, p', \hat{\sigma})$, where $p \in \mathbf{Point}_P$, $p' \in \mathbf{Point}_{P'}$, and $\hat{\sigma}$ is a state mapping from all variables of both programs to values. A state can be considered as a pair of states: one for the variables of P and one for the variables of P'. In the sequel, such a state $\hat{\sigma}$ is written as (σ, σ'), where σ is for P and σ' is for P'. Similarly, the configuration $(p, p', \hat{\sigma})$ can be written as (p, p', σ, σ').

Similar to the case of a single program, we say that a configuration $\gamma = (p, p', \sigma, \sigma')$ is called an *entry configuration for* (P, P') if $p = \mathit{entry}(P)$ and $p' = \mathit{entry}(P')$, and an *exit configuration for* (P, P') if $p = \mathit{exit}(P)$ and $p' = \mathit{exit}(P')$. We overload the functions pp and state to deal with such configurations, that is, $pp(\gamma) = (p, p')$ and $\mathit{state}(\gamma) = (\sigma, \sigma')$. We introduce two new functions on configurations, ps_1 and ps_2, such that, on γ, $ps_1(\gamma) = (p, \sigma)$ is a configuration of P and $ps_2(\gamma) = (p', \sigma')$ is a configuration of P'.

The transition relation $\mapsto$ of a pair (P, P') of programs contains two kinds of transition:

$$(p_1, p', \sigma_1, \sigma') \mapsto (p_2, p', \sigma_2, \sigma'),$$

such that $(p_1, \sigma_1) \mapsto (p_2, \sigma_2)$ is in the transition relation of P, and

$$(p, p'_1, \sigma, \sigma'_1) \mapsto (p, p'_2, \sigma, \sigma'_2),$$

such that $(p_1, \sigma_1) \mapsto (p_2, \sigma_2)$ is in the transition relation of P'.

Having the notion of transition relation for pairs of programs, the notions of computation sequence and run can be defined in the same way as in the case of a single program. That is, a computation sequence of (P, P') is a finite or infinite sequence

$$\gamma_0, \gamma_1, \ldots$$

of configurations such that $\gamma_i \mapsto \gamma_{i+1}$ for all i. A run from an initial state $\hat{\sigma}$ is a computation sequence such that $\gamma_0 = (p_0, p'_0, \hat{\sigma})$ is an entry configuration. One can observe that, for any pair (σ, σ') of states, there can be many runs of (P, P') from (σ, σ'). The following theorem then shows that if any of those run is terminating, then all runs are terminating, and they terminate at the same configuration.

LEMMA 4.1 *Let (P, P') be a pair of programs. If a run of (P, P') from an entry configuration γ is terminating at an exit configuration γ', then all runs of (P, P') from γ are terminating at γ'.*

PROOF. Let $R = \gamma_0, \ldots, \gamma_k$ be a terminating run of (P, P') such that $\gamma_0 = \gamma$ and $\gamma_k = \gamma'$. Denote by $R|P$ the subsequence

$$\gamma_{i_0}, \ldots, \gamma_{i_m}$$

of R such that, for all $l = 0, \ldots, m-1$, the transition $ps_1(\gamma_{i_l}) \mapsto ps_1(\gamma_{i_{l+1}})$ is a transition in P. This means that the run of P from the entry configuration $ps_1(\gamma_{i_0})$ terminates at the exit configuration $ps_1(\gamma_{i_m})$. Similarly for $R|P'$, we have the subsequence

$$\gamma_{j_0}, \ldots, \gamma_{j_n}.$$

We also have

$$\begin{array}{rcl} \gamma_0 & = & (pp(ps_1(\gamma_{i_0})), pp(ps_2(\gamma_{j_0})), \\ & & \mathit{state}(ps_1(\gamma_{i_0})), \mathit{state}(ps_2(\gamma_{j_0}))) \\ \gamma_k & = & (pp(ps_1(\gamma_{i_m})), pp(ps_2(\gamma_{j_n})), \\ & & \mathit{state}(ps_1(\gamma_{i_m})), \mathit{state}(ps_2(\gamma_{j_n}))). \end{array} \tag{2}$$

Assume that there is a non-terminating run R' of (P, P') from γ. Then, $R'|P$ or $R'|P'$ is infinite. Without loss of generality, suppose that

$$R'|P = \gamma'_0, \gamma'_1, \ldots$$

is infinite. That is, the run of P from $ps_1(\gamma'_0)$ is non-terminating. However, since $ps_1(\gamma'_0) = ps_1(\gamma_{i_0})$ and P is deterministic, the run of P from $ps_1(\gamma'_0)$ must terminate. This contradicts the existence of R'.

Moreover, the run of P from $ps_1(\gamma_{i_0})$ must terminate at $ps_1(\gamma_{i_m})$. Using the same argument, we can show that the run of P' from $ps_1(\gamma_{j_0})$ must terminate at $ps_1(\gamma_{j_n})$. By equalities 2, it follows that all runs of (P, P') from γ are terminating at γ'. □

We will show later that the above lemma allows us to preserve meta properties of the abstract notions introduced in the previous section when these notions are developed further for the case of a pair of programs.

An *assertion function* of a pair (P, P') of programs is a partial function

$$I : \mathbf{Point}_P \times \mathbf{Point}_{P'} \to \mathbf{Assertion}$$

mapping pairs of program points of P and P' to assertions such that I is defined on $(entry(P), entry(P'))$ and $(exit(P), exit(P'))$.

Given an assertion function I, we call a pair of program points (p, p') *I-observable* if $I(p, p')$ is defined. Let $\gamma = (p, p', \sigma, \sigma')$ be a configuration. Then, γ is I-observable if so is the pair of program points (p, p'). We also say that γ *satisfies* I, denoted by $\gamma \models I$, if I is defined on (p, p') and $(\sigma, \sigma') \models I(p, p')$. We will also say that I is defined on γ if it is defined on (p, p') and write $I(\gamma)$ to denote $I(p, p')$.

The notions of partial and total correctness for the case of single programs can be adapted for the case of pairs of programs. A pair (P, P') of programs is partially correct with respect to a precondition φ and a postcondition ψ, denoted by $\{\varphi\}(P, P')\{\psi\}$, if for every run of (P, P') from a configuration satisfying φ and reaching an exit configuration, this exit configuration satisfies ψ. A pair (P, P') of programs is totally correct with respect to a precondition φ and a postcondition ψ, denoted by $[\varphi](P, P')[\psi]$, if every run of (P, P') from a configuration satisfying φ terminates in an exit configuration and this exit configuration satisfies ψ.

Unlike in the case of a single program, for a pair of programs, there is no notions of invariant and strongly-extendible assertion function. The transition relation of a pair of programs has no synchronization mechanism. For example, one program in a pair can make as many transitions as possible, while the other program in the same pair stays at some program point without making any transition. Thus, it is not useful to have the notions of invariant and strongly-extendible assertion functions.

The notion of weakly-extendible assertion function is better suited for describing inter-program properties. Weakly-extendible assertion functions for a pair of programs can be defined in the same way as in the case of a single program.

DEFINITION 4.2 Let I be an assertion function of a pair (P, P') of programs. I is *weakly extendible* if for every run

$$\gamma_0, \ldots, \gamma_i$$

of (P, P') such that $i \geq 0$, $\gamma_0 \models I$, $\gamma_i \models I$, and γ_i is not an exit configuration, there exists a finite computation sequence

$$\gamma_i, \ldots, \gamma_{i+n}$$

of (P, P') such that

1. $n > 0$, and
2. $\gamma_{i+n} \models I$.

□

EXAMPLE 4.3 Let us illustrate the notion of weakly-extendible assertion function for a pair of programs. Consider the following two programs P and P':

P	P'
$\mathsf{i} := 0$	
$\mathsf{j} := 0$	
while $(\mathsf{j} < 100)$ **do**	$\mathsf{i}' := 0$
if $(\mathsf{i} > \mathsf{j})$ **then** $\mathsf{j} := \mathsf{j} + 1$	$\mathsf{j}' := 0$
else $\mathsf{i} := \mathsf{i} + 1$	**while** $(\mathsf{j}' < 100)$ **do**
fi	$\mathsf{i}' := \mathsf{i}' + 1$
q :	$\mathsf{j}' := \mathsf{j}' + 1$
od	q' :
	od

Define an assertion function I of (P, P') such that

$$\begin{array}{lcl} I(entry(P), entry(P')) & = & \top \\ I(q, q') & = & \varphi \\ I(exit(P), exit(P')) & = & \varphi, \end{array}$$

where

$$\varphi = (\mathsf{i} = \mathsf{i}') \wedge (\mathsf{j} = \mathsf{j}') \wedge (\mathsf{i} = \mathsf{j}).$$

The function I is weakly extendible due to the following properties:

1. From an entry configuration of (P, P'), by taking a computation sequence consisting of two iterations of the loop of P and one iteration of the loop of P', one reaches a configuration with program points (q, q') in which φ holds.
2. For every $v < 100$, from a configuration with the program points (q, q') in which $\mathsf{i} = \mathsf{i}' = \mathsf{j} = \mathsf{j}' = v$, by taking a computation sequence consisting of two iterations of the loop of P and one iteration of the loop of P', one again reaches a configuration with program points (q, q') in which $\mathsf{i} = \mathsf{i}' = \mathsf{j} = \mathsf{j}' = v + 1$.
3. For every $v \geq 100$, from a configuration with the program points (q, q') in which $\mathsf{i} = \mathsf{i}' = \mathsf{j} = \mathsf{j}' = v$, one can reach an exit configuration in which $\mathsf{i} = \mathsf{i}' = \mathsf{j} = \mathsf{j}' = v$.

□

Concerning the sufficiency of weakly-extendible assertion functions for proving partial correctness, we obtain the same result as in the case of a single program, as stated by the following theorem:

THEOREM 4.4 *Let I be an assertion function of a pair (P, P') of programs such that*

$$\varphi = I(entry(P), entry(P')) \text{ and } \psi = I(exit(P), exit(P')).$$

If the assertion function I is weakly extendible, then $\{\varphi\}(P, P')\{\psi\}$, that is, (P, P') is partially correct with respect to the precondition φ and postcondition ψ.

PROOF. Suppose that I is weakly extendible and $\gamma \overset{*}{\mapsto}_{(P,P')} \gamma'$, where γ is an entry configuration and γ' is an exit configuration, and $\gamma \models \varphi$. It follows that $\gamma \models I$. By Lemma 4.1, all runs of (P, P') from γ terminate at γ'.

Consider any complete run

$$R = \gamma_0, \ldots, \gamma_m$$

of (P, P') from γ, that is, $\gamma = \gamma_0$ and $\gamma_m = \gamma'$. We need to prove that $\gamma_m \models \psi$. Take the largest number j such that γ_j is not the exit configuration γ' and $\gamma_j \models I$. Such a configuration exists since $\gamma_0 = \gamma$ and $\gamma \models I$. Since I is weakly extendible, there exists a computation sequence

$$\gamma_j, \ldots, \gamma_{j+n}$$

such that $\gamma_{j+n} \models I$. Now, since j is the largest one, we have $\gamma_{j+n} = \gamma_m$, and thus $\gamma_m \models I$. It follows by the definition of I that $\gamma_m \models \psi$, as required. □

Similar to the properties of a single program, the verification conditions associated with inter-program properties use the notion of path. However, since the flow graphs of the two programs in a pair of programs are considered disjoint, the notion of path for pairs of programs needs to be elaborated. A *path* π of a pair (P, P') of programs is a finite or infinite sequence

$$(p_0, p'_0), (p_1, p'_1), \ldots$$

of pairs of program points such that, for all $i \geq 0$, either

- (p_i, p_{i+1}) is an edge of $\mathbf{G}_P$ and $p'_i = p'_{i+1}$, or
- (p'_i, p'_{i+1}) is an edge of $\mathbf{G}_{P'}$ and $p_i = p_{i+1}$

A path $\hat{\pi}$ of (P, P') can be considered as a trajectory in a two dimensional space where the axes are paths of P and P'. We denote such a path $\hat{\pi}$ by (π, π'), where π and pi' are the axes of the space, π is a path of P and π' is a path of P'.

Having the notion of path for a pair of programs, the notions of precondition and liberal precondition for paths of a pair of programs can be defined in the same way as in the case of a single program. In fact, the weakest precondition of a path of a pair of programs may be derived from the paths of the single programs.

THEOREM 4.5 *Let (π, π') be a path of a pair (P, P') of programs. Let ψ be an assertion such that ψ is equivalent to $\psi_1 \wedge \psi_2$, where ψ_1 contains only variables from P and ψ_2 contains only variables from P'. Then, $wp_{(\pi,\pi')}(\psi)$ is equivalent to $wp_\pi(\psi_1) \wedge wp_{\pi'}(\psi_2)$.*

PROOF. Let $(\pi, \pi') = (p_0, p'_0), \ldots, (p_k, p'_k)$ Suppose there is a pair (σ_0, σ'_0) of states that satisfies $wp_{\pi\cdot\pi'}(\psi)$. Then there is a sequence of $(\sigma_1, \sigma'_1), \ldots, (\sigma_k, \sigma'_k)$ such that

$$(p_0, p'_0, \sigma_0, \sigma'_0) \mapsto (p_1, p'_1, \sigma_1, \sigma'_1) \ldots \mapsto (p_k, p'_k, \sigma_k, \sigma'_k)$$

and $(\sigma_k, \sigma'_k) \models \psi$, which also means $(\sigma_k, \sigma'_k) \models \psi_1 \wedge \psi_2$. By the disjointness of sets of variables of P and P', we have $\sigma_k \models \psi_1$ and $\sigma'_k \models \psi_2$.

By the construction of (π, π'), we have

$$(p_{i_0}, \sigma_{i_0}) \mapsto \ldots \mapsto (p_{i_m}, \sigma_{i_m})$$

such that $\pi = p_{i_0}, \ldots, p_{i_m}$, $\sigma_{i_0} = \sigma_0$, and $\sigma_{i_m} = \sigma_k$. Similarly, we have

$$(p'_{j_0}, \sigma'_{j_0}) \mapsto \ldots \mapsto (p'_{j_n}, \sigma'_{j_n})$$

such that $\pi' = p'_{j_0}, \ldots, p'_{j_n}$, $\sigma'_{j_0} = \sigma'_0$, and $\sigma'_{j_n} = \sigma'_k$. It follows that $\sigma_0 \models wp_\pi(\psi_1)$ and $\sigma'_0 \models wp_{\pi'}(\psi_2)$. Consequently, $(\sigma_0, \sigma'_0) \models wp_\pi(\psi_1) \wedge wp_{\pi'}(\psi_2)$, as required. □

We can define the verification condition associated with weakly extendible assertion functions similarly to the case of a single program.

DEFINITION 4.6 Let I be an assertion function of a pair (P, P') of programs and Π a set of non-trivial paths of the pair of programs such that for every path π in Π both $start(\pi)$ and $end(\pi)$ path are I-observable. For every pair (p, p') of program points, denote by $\Pi|(p, p')$ the set of paths in Π whose first pair of points is (p, p').

The *weak verification condition* associated with I and Π consists of all assertions of the form

$$I(start(\pi)) \Rightarrow wlp_\pi(I(end(\pi))),$$

where $\pi \in \Pi$ and all assertions of the form

$$I(p, p') \Rightarrow \bigvee_{\pi \in \Pi|(p,p')} wp_\pi(\top),$$

where (p, p') is an I-observable point, and p is not the exit point of P. □

THEOREM 4.7 *Let I and Π be as in Definition 4.6 and $\mathbb{W}$ be the weak verification condition associated with I and Π. If every assertion in $\mathbb{W}$ is valid, then I is weakly extendible.*

PROOF. In the proof, whenever we denote a configuration by γ_i, we use (p_i, p'_i) for the program points and (σ_i, σ'_i) for the states of this configuration, and similarly for other indices instead of i. Take any run $\gamma_0, \ldots, \gamma_i$ of (P, P') such that $\gamma_0 \models I$, $\gamma_i \models I$ and γ_i is not an exit configuration. Since (p_i, p'_i) is I-observable, the following assertion belongs to $\mathbb{W}$:

$$I(p_i, p'_i)) \Rightarrow \bigvee_{\pi \in \Pi | (p_i, p'_i)} wp_\pi(\top),$$

and hence it is true. Since $\gamma_i \models I$, we have $(\sigma_i, \sigma'_i) \models I(p_i, p'_i)$, then by the validity of the above formula we have

$$(\sigma_i, \sigma'_i) \models \bigvee_{\pi \in \Pi | (p_i, p'_i)} wp_\pi(\top).$$

This implies that there exists a path $\pi \in \Pi|(p_i, p'_i)$ such that $(\sigma_i, \sigma'_i) \models wp_\pi(\top)$. Let the path π have the form

$$(p_i, p'_i), \ldots, (p_{i+n}, p'_{i+n}).$$

Then, by the definition of $wp_\pi(\top)$, there exist pairs of states $(\sigma_{i+1}, \sigma'_{i+1}), \ldots, (\sigma_{i+n}, \sigma_{i+n})$ such that

$$\begin{array}{l}(p_i, p'_i, \sigma_i, \sigma'_i) \mapsto (p_{i+1}, p_{i+1}, \sigma_{i+1}, \sigma'_{i+1}) \mapsto \ldots \\ \ldots \mapsto (p_{i+n}, p'_{i+n}, \sigma_{i+n}, \sigma'_{i+n}).\end{array}$$

Using that $\pi \in \Pi$, it follows that $(\sigma_{i+n}, \sigma'_{i+n}) \models I(p_{i+n}, p'_{i+n})$. □

The notion of weak verification condition is the cornerstone of our theory of inter-program properties. The notion of weak verification condition forms a suitable notion of certificate about properties involving two programs.

5 Translation Validation

Translation validation [11] is an approach to compiler verification. In this approach, instead of proving the correctness of a compiler for all source programs, one proves that, *for a single source program*, the program and the result of its compilation, or the target program, are semantically equivalent. Translation validation approach has mainly been used in the verification of optimizing compilers, for example in [12, 10, 15, 13, 8]. In the case of optimizing compilers, the target program is obtained by applying optimizing transformations to the source program. Both source and target programs are usually in the same language. In the sequel we focus the application of our theory on the translation validation for optimizing compilers.

In translation validation one first has to define formally the correctness property between the source and the target programs. A typical correctness property in translation validation is semantic equivalence. An example of informal definition of semantic equivalence is as follows: a source program P and a target program P' are semantically equivalent if, for every pair of runs of both programs on the same input, (1) both runs perform the same sequence of function calls, (2) one run is terminating if and only if so is the other, and (3) on termination both runs return the same value. Having the correctness property, usually one then defines a notion of correspondence between two programs. The semantic equivalence is then established by finding some correspondences between the programs. Both the correctness property and the notion of correspondence are inter-program properties. If we can show that such properties can be captured by our notion of extendible assertion function, then we can provide certificates or proofs for those properties.

5.1 Basic-Block and Variable Correspondences

We start our discussion with our translation validation work described in [8, 9]. In our work we introduce the notion of basic-block and variable correspondence. The equivalence between two programs is established by finding certain basic-block and variable correspondences.

Denote by $\mathbf{InVar}_P$ the set of input variables of a program P. In the sequel, given two programs P and P', we assume that there always exists a one-to-one correspondence In between $\mathbf{InVar}_P$ and $\mathbf{InVar}_{P'}$. We also say that the runs R and R' are on the same input if, let states σ_0 and σ'_0 be the initial states of, respectively, R and R', we have $\sigma_0(x) = \sigma'_0(In(x))$ for all $x \in \mathbf{InVar}_P$.

We first define the notion of program equivalence. Denote by $\mathbf{InVar}_P$ and $\mathbf{ObsVar}_P$ the sets of, respectively, input variables and observable variables of a program P. The source program P and the target program P' are *semantically equivalent* if there exist a one-to-one correspondence In between $\mathbf{InVar}_P$ and $\mathbf{InVar}_{P'}$ and a one-to-one correspondence Obs between $\mathbf{ObsVar}_P$ and $\mathbf{ObsVar}_{P'}$, such that for every pair of runs

$$\begin{array}{rcl} R & = & (p_0, \sigma_0), (p_1, \sigma_1), \ldots \\ R' & = & (p'_0, \sigma'_0), (p'_1, \sigma'_1), \ldots \end{array}$$

of, respectively, P and P', and $\sigma_0(x) = \sigma'_0(In(x))$ for all $x \in \mathbf{InVar}_P$, the following conditions hold:

- R is terminating (or finite) if and only if so is R';
- if R and R' are terminating with, respectively, states σ and σ', then $\sigma(y) = \sigma(Obs(y))$ for all $y \in \mathbf{ObsVar}_P$.

A block in a program is a sequence of statements in the program. A block is basic if it is maximal, and it can only be entered at the beginning and exited at the end of the block.

Let us assume that the program points being considered in a programs consist of the entry point of each basic block in the program, such that the point is denoted by the basic block itself. A run can be defined as a sequence

$$(\beta_0, \sigma_0), (\beta_1, \sigma_1), (\beta_2, \sigma_2), \ldots,$$

where, for all $i \geq 0$, the point β_i is the entry point of basic block β_i. For any run R and any sequence $\bar{b}$ of basic blocks, we denote by $R|\bar{b}$ the subsequence of R consisting only of configurations whose program points are the entry points of basic blocks in $\bar{b}$.

Given two programs P and P', let $\bar{b} = b_1, \ldots, b_m$ and $\bar{b}' = b'_1, \ldots, b'_m$ be sequences of distinct basic blocks of, respectively, P and P', and let $\bar{x} = x_1, \ldots, x_n$ and $\bar{x}' = x'_1, \ldots, x'_n$ be sequences of distinct variables of, respectively, P and P'. There is a *basic-block and variable correspondence* between $(\bar{b}, \bar{x})$ and $(\bar{b}', \bar{x}')$ if for every two runs

$$\begin{array}{rcl} R & = & (\beta_0, \sigma_0), (\beta_1, \sigma_1), \ldots \\ R' & = & (\beta'_0, \sigma'_0), (\beta'_1, \sigma'_1), \ldots \end{array}$$

of, respectively, P and P' on the same inputs, let

$$\begin{array}{rcl} R|\bar{b} & = & (\beta_{i_0}, \sigma_{i_0}), (\beta_{i_1}, \sigma_{i_1}), \ldots \\ R'|\bar{b}' & = & (\beta'_{i'_0}, \sigma'_{i'_0}), (\beta'_{i'_1}, \sigma'_{i'_1}), \ldots, \end{array}$$

then $R|\bar{b}$ and $R|\bar{b}'$ are of the same length and the following conditions hold: for all k

1. $\beta_{i_k} = b_j$ if and only if $\beta'_{i'_k} = b'_j$ for all j, and
2. $\sigma_{i_k+1}(x_l) = \sigma'_{i'_k+1}(x'_l)$ for all l.

In the sequel, we often call $\bar{b}, \bar{b}'$ sequences of *control blocks* and $\bar{x}, \bar{x}'$ sequences of *control variables*. We assume that every program has a unique start block and a unique exit block. The entry of start block is the program's entry point, while the exit of exit block is the program's exit points. The start block of a program is also a control block, and it always corresponds to the start block of the other program.

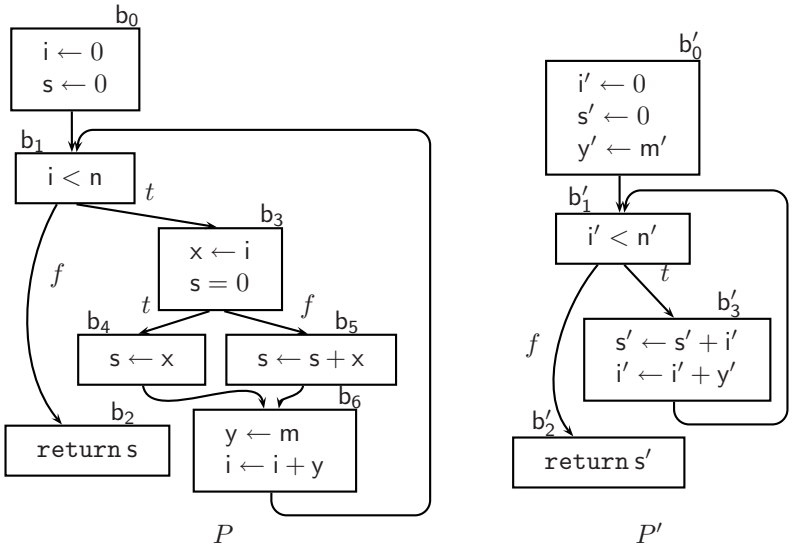

Figure 1: P' is an optimized version of P.

EXAMPLE 5.1 Let us give an example of basic-block and variable correspondence. Consider the programs P and P' depicted in Figure 1. The variables m, n are the input variables of P, and their primed counterparts are the input variables of P' such that $In(\mathsf{m}) = \mathsf{m}'$ and $In(\mathsf{n}) = \mathsf{n}'$. P' is obtained from P by fusing the branches that are represented by blocks b_4 and b_5, and by moving the assignment instruction $\mathsf{y} \leftarrow \mathsf{m}$ out of the loop.

There is a basic-block and variable correspondence between $(\mathsf{b}_6, (\mathsf{s}, \mathsf{i}, \mathsf{m}, \mathsf{n}))$ and $(\mathsf{b}'_3, (\mathsf{s}', \mathsf{i}', \mathsf{m}', \mathsf{n}'))$. For every two runs of P and P' on the same input, the block b_6 is visited as many times as the block b'_3 is visited, and on each corresponding visits, at the exits of the blocks, the values of $\mathsf{s}, \mathsf{i}, \mathsf{m}, \mathsf{n}$ coincide with the values of their primed counterparts. With the same reasoning, it is easy to see that there is a basic-block and variable correspondence between $((\mathsf{b}_6, \mathsf{b}_2), (\mathsf{s}, \mathsf{i}, \mathsf{m}, \mathsf{n}))$ and $((\mathsf{b}'_3, \mathsf{b}'_2), (\mathsf{s}', \mathsf{i}', \mathsf{m}', \mathsf{n}'))$. □

Establishing program equivalence can be accomplished by finding basic-block and variable correspondences between the exit blocks and between the observable variables.

THEOREM 5.2 *Let P and P' be programs, and b_t and b'_t be the exit blocks of P and P', respectively. Let Obs be the one-to-one correspondence between $\mathbf{ObsVar}_P$ and $\mathbf{ObsVar}_{P'}$, where $\mathbf{ObsVar}_P = \{x_1, \ldots, x_n\}$. P and P' are semantically equivalent if and only if there is a basic-block and variable correspondence between*

$$(b_t, (x_1, \ldots, x_n)) \text{ and } (b'_t, (Obs(x_1), \ldots, Obs(x_n)))$$

□

Note that in the above example there is a correspondence between

$$((\mathsf{b}_6, \mathsf{b}_2), (\mathsf{s}, \mathsf{i}, \mathsf{m}, \mathsf{n})) \text{ and } ((\mathsf{b}'_3, \mathsf{b}'_2), (\mathsf{s}', \mathsf{i}', \mathsf{m}', \mathsf{n}')).$$

By the definition of basic-block and variable correspondences, it obviously follows that there is a correspondence between $(\mathsf{b}_2, \mathsf{s})$ and $(\mathsf{b}'_2, \mathsf{s}')$. Since b_2 and b'_2 are the exit blocks and the only observable variables are s and s', we can conclude that the programs P and P' in the above example are equivalent.

The verification of basic-block and variable correspondences has been described in detail in [9]. For presentation in this section, suppose that one finds a basic-block and variable correspondence between $(\bar{b}, \bar{x})$ of a program P and $(\bar{b}', \bar{x}')$ of a program P', where $\bar{b} = b_1, \ldots, b_m$, $\bar{b}' = b'_1, \ldots, b'_m$, $\bar{x} = x_1, \ldots, x_n$,

and $\bar{x}' = x'_1, \ldots, x'_n$. Assume that a path is a sequence of program points, where each point is the *exit* of a basic block and is denoted by the basic block itself. Given a sequence $\bar{b}$ of basic blocks, a path $\pi = \beta_0, \ldots, \beta_k$ is $\bar{b}$-simple if β_0 and β_k are in $\bar{b}$, or β_0 is the start block and β_k is in $\bar{b}$, but none of $\beta_1, \ldots, \beta_{k-1}$ are in $\bar{b}$. Assume further that every program has a unique variable ρ representing program counter, and for every basic block β in the program, the value of ρ is updated with β by the assignment $\rho \leftarrow \beta$ at the entry of β.

The verification condition associated with a basic-block and variable correspondence consists of two parts: conjecture preservation and simulation relation. A conjecture is a set of assertions. Consider again the basic-block and variable correspondence between $(\bar{b}, \bar{x})$. Let ρ, ρ' be program counters of, respectively, programs P and P', and $\mathcal{C}$ be a conjecture in the verification condition. For two blocks β, β', we often write $\rho(\beta, \beta')$ as a shorthand for $\rho = \beta \wedge \rho' = \beta'$. For every pair β, β' of corresponding control blocks, we require that the assertion $\bigwedge \mathcal{C} \wedge \rho(\beta, \beta') \Rightarrow \bigwedge_{k=1}^{n} x_k = x'_k$ is valid. For all $i = 1, \ldots, m$ and for all $j = 1, \ldots, m$, let

$$\begin{array}{lcl} \Pi_{b_j, b_i} & = & \{\pi^1_{b_j, b_i}, \ldots, \pi^c_{b_j, b_i}\} \\ \Pi_{b'_j, b'_i} & = & \{\pi^1_{b'_j, b'_i}, \ldots, \pi^d_{b'_j, b'_i}\} \end{array}$$

be the sets of, respectively, all $\bar{b}$-simple paths between b_j and b_i and all $\bar{b}'$-simple paths between b'_j and b'_i. The verification condition associated with conjecture preservation consists of the following assertions: for all $k = 1, \ldots, c$ and for all $l = 1, \ldots, d$,

$$\bigwedge \mathcal{C} \wedge \rho(b_j, b'_j) \Rightarrow wlp_{\pi^k_{b_j, b_i}}(wlp_{\pi^l_{b'_j, b'_i}}(\bigwedge \mathcal{C})).$$

The verification condition associated with simulation relation consists of the following assertions: for all $k = 1, \ldots, c$,

$$\bigwedge \mathcal{C} \wedge \rho(b_j, b'_j) \wedge wp_{\pi^k_{b_j, b_i}}(\top) \quad \Rightarrow \quad \bigvee_{l=1}^{d} wp_{\pi^l_{b'_j, b'_i}}(\top),$$

and for all $l = 1, \ldots, d$,

$$\bigwedge \mathcal{C} \wedge \rho(b_j, b'_j) \wedge wp_{\pi^l_{b'_j, b'_i}}(\top) \quad \Rightarrow \quad \bigvee_{k=1}^{c} wp_{\pi^k_{b_j, b_i}}(\top).$$

EXAMPLE 5.3 Let us consider again the programs P and P' in Example 5.1. The programs are depicted in Figure 1. In this example we show the verification condition associated with the basic-block and variable correspondence between $((\mathsf{b}_6, \mathsf{b}_2), (\mathsf{s}, \mathsf{i}, \mathsf{m}, \mathsf{n}))$ and $((\mathsf{b}'_3, \mathsf{b}'_2), (\mathsf{s}', \mathsf{i}', \mathsf{m}', \mathsf{n}'))$.

Let b_s and b'_s be the start blocks of, respectively, P and P'. That is, b_s is the predecessor of b_0 and b'_s is the predecessor of b'_0. Let φ be an assertion equivalent to $\mathsf{m} = \mathsf{m}' \wedge \mathsf{n} = \mathsf{n}'$. The conjecture $\mathcal{C}$ in the verification condition consists of the following assertions:

$$\begin{array}{l} \rho(b_s, b'_s) \Rightarrow \varphi, \\ \rho(\mathsf{b}_6, \mathsf{b}'_3) \Rightarrow \varphi \wedge \mathsf{s} = \mathsf{s}' \wedge \mathsf{i} = \mathsf{i}' \wedge \mathsf{y}' = \mathsf{m}', \text{ and} \\ \rho(\mathsf{b}_2, \mathsf{b}'_2) \Rightarrow \varphi \wedge \mathsf{s} = \mathsf{s}' \wedge \mathsf{i} = \mathsf{i}'. \end{array}$$

The first assertion above describes the input condition. The second and third assertions describe the correspondence between corresponding control variables at the corresponding control blocks. Note that in the second assertion the conjunction $\mathsf{y}' = \mathsf{m}'$ is a loop invariant that is crucial for proving the correspondence.

Having the conjecture, we can generate assertions associated with conjecture preservation and simulation relation for the following pairs of sets of simple paths:

- $\Pi_{\mathsf{b}_6, \mathsf{b}_6} = \{\pi^{\mathsf{b}_4}_{\mathsf{b}_6, \mathsf{b}_6}, \pi^{\mathsf{b}_5}_{\mathsf{b}_6, \mathsf{b}_6}\}$ and $\Pi_{\mathsf{b}'_3, \mathsf{b}'_3} = \{\pi_{\mathsf{b}'_3, \mathsf{b}'_3}\}$;
- $\Pi_{\mathsf{b}_6, \mathsf{b}_2} = \{\pi_{\mathsf{b}_6, \mathsf{b}_2}\}$ and $\Pi_{\mathsf{b}'_3, \mathsf{b}'_2} = \{\pi_{\mathsf{b}'_3, \mathsf{b}'_2}\}$;
- $\Pi_{b_s, \mathsf{b}_6} = \{\pi^{\mathsf{b}_4}_{b_s, \mathsf{b}_6}, \pi^{\mathsf{b}_5}_{b_s, \mathsf{b}_6}\}$ and $\Pi_{b'_s, \mathsf{b}'_3} = \{\pi_{b'_s, \mathsf{b}'_3}\}$; and
- $\Pi_{b_s, \mathsf{b}_2} = \{\pi_{b_s, \mathsf{b}_2}\}$ and $\Pi_{b'_s, \mathsf{b}'_2} = \{\pi_{b'_s, \mathsf{b}'_2}\}$,

where $\pi^{b_3}_{b_1, b_2}$ denotes a path from b_1 to b_2 via b_3. □

The notion of basic-block and variable correspondence can be captured by the notions of extendible assertion function and weak verification condition. That is, given a basic-block and variable correspondence between programs P and P', we can define an extendible assertion function $\hat{I}$ and a set $\hat{\Pi}$ of paths of (P, P'), such that if all assertions in the weak verification condition associated with $\hat{I}$ and $\hat{\Pi}$ are valid, then so are all assertions in the verification condition associated with the correspondence.

Consider a basic-block and variable correspondence between $(\bar{b}, \bar{x})$ of P and $(\bar{b}', \bar{x}')$ of P'. Let $\mathbb{V}$ be the verification condition associated with the correspondence, such that $\mathcal{C}$ is the conjecture in $\mathbb{V}$. We define the assertion function $\hat{I}$ of (P, P') that expresses $\mathcal{C}$. For simplicity, since we are interested in the correspondence of control variables at the exits of corresponding control blocks, we say that $\hat{I}$ is defined on a pair (β, β') of control blocks to mean that $\hat{I}$ is defined on a pair $(exit(\beta), exit(\beta))$ of the exits of β and β. The function $\hat{I}$ is defined as follows:

$$\hat{I}(entry(P), entry(P')) = \rho(entry(P), entry(P')) \wedge \bigwedge \mathcal{C}$$

and for every pair β, β' of corresponding control blocks

$$\hat{I}(\beta, \beta') \quad = \quad \rho(\beta, \beta') \wedge \bigwedge \mathcal{C}.$$

Note that $\hat{I}$ can be defined on other pairs of programs points.

Next, recall that a path $\hat{\pi}$ of (P, P') can be considered as a trajectory in a two dimensional space where the axes are a path π of P and a path π' of P'. We denote such a path $\hat{\pi}$ by (π, π'). We now impose some requirements on the set $\hat{\Pi}$ of paths of (P, P'). First, the set $\hat{\Pi}$ includes all simple paths in the sets of simple paths used to generate the verification condition $\mathbb{V}$, that is,

$$\begin{array}{l}\hat{\Pi} \supseteq \{(\pi_{\beta_1,\beta_2}, \pi_{\beta'_1,\beta'_2}) \mid \\ \qquad \exists \Pi_{\beta_1,\beta_2}, \Pi_{\beta'_1,\beta'_2}. \pi_{\beta_1,\beta_2} \in \Pi_{\beta_1,\beta_2} \wedge \pi_{\beta'_1,\beta'_2} \in \Pi_{\beta'_1,\beta'_2}\},\end{array}$$

where Π_{β_1,β_2} is the set of all $\bar{b}$-simple paths from β_1 to β_2 and $\Pi_{\beta'_1,\beta'_2}$ is the set of all $\bar{b}'$-simple paths from β'_1 to β'_2. Second, for every other path (π, π') in $\hat{\Pi}$, π and π' are not prefixes of any $\bar{b}$-simple and $\bar{b}'$-simple path, and neither the nontrivial prefixes of π are $\bar{b}$-simple paths nor the nontrivial prefixes of π' are $\bar{b}'$-simple paths.

Having the function $\hat{I}$ and the set $\hat{\Pi}$, one can prove a basic-block and variable correspondences by proving the weak verification condition associated with $\hat{I}$ and $\hat{\Pi}$.

THEOREM 5.4 *Let $\mathbb{V}$, $\hat{I}$, and $\hat{\Pi}$ be as defined above, and $\mathbb{W}$ be the weak verification condition associated with $\hat{I}$ and $\hat{\Pi}$. Then, if all assertions in $\mathbb{W}$ are valid, then all assertions in $\mathbb{V}$ are valid.*

PROOF. We first prove the conjecture preservation of $\mathbb{V}$. Take any two pairs (β_1, β'_1) and (β_2, β'_2) of corresponding control blocks, such that there exist a $\bar{b}$-simple path π_{β_1,β_2} from β_1 to β_2 and a $\bar{b}'$-simple path $\pi_{\beta'_1,\beta'_2}$ from β'_1 to β'_2. We need to prove that the assertion

$$\bigwedge \mathcal{C} \wedge \rho = \beta_1 \wedge \rho' = \beta'_1 \Rightarrow wlp_{\pi_{\beta_1,\beta_2}}(wlp_{\pi_{\beta'_1,\beta'_2}}(\bigwedge \mathcal{C})) \tag{3}$$

is valid. Since the assertion $\bigwedge \mathcal{C} \wedge \rho = \beta_1 \wedge \rho' = \beta'_1$ is equivalent to $\hat{I}(\beta_1, \beta'_1)$ and we assume that ρ and ρ' are updated immediately preceding the exit blocks, the assertion $wlp_{(\pi_{\beta_1,\beta_2}, \pi_{\beta'_1,\beta'_2})}(\hat{I}(\beta_2, \beta'_2))$ is equivalent to $wlp_{\pi_{\beta_1,\beta_2}}(wlp_{\pi_{\beta'_1,\beta'_2}}(\bigwedge \mathcal{C}))$. Because the assertion

$$\hat{I}(\beta_1, \beta'_1) \Rightarrow wlp_{(\pi_{\beta_1,\beta_2}, \pi_{\beta'_1,\beta'_2})}(\hat{I}(\beta_2, \beta'_2))$$

is valid, so is the assertion (3).

For the simulation relation of $\mathbb{V}$, we prove it by contradiction. Assume that there are two pairs (β_1, β'_1) and (β_2, β'_2) of corresponding control blocks such that, without loss of generality, there is a $\bar{b}$-simple path π_{β_1,β_2}, but the assertion

$$\bigwedge \mathcal{C} \wedge \rho = \beta_1 \wedge \rho' = \beta'_1 \wedge wp_{\pi_{\beta_1,\beta_2}}(\top) \Rightarrow \bigvee_{\pi \in \Pi_{\beta'_1,\beta'_2}} wp_{\pi}(\top)$$

is not valid. Recall that the set $\Pi_{\beta_1',\beta_2'}$ is the set of all $\bar{b}'$-simple paths from β_1' to β_2'. Since the assertion $\bigwedge \mathcal{C} \wedge \rho = \beta_1 \wedge \rho' = \beta_1'$ is equivalent to $\hat{I}(\beta_1, \beta_1')$, it means that there is a pair (σ, σ') of states satisfying $\hat{I}(\beta_1, \beta_1')$ such that there is a computation sequence of P that starts from the exit of β_1 and state σ, and follows the path π_{β_1,β_2}, but there is no computation sequence of P' that starts from the exit of β_1' and state σ', and follows any path in $\Pi_{\beta_1',\beta_2'}$.

By the requirements imposed on $\hat{\Pi}$, the path π_{β_1,β_2} is only paired with some path in $\Pi_{\beta_1',\beta_2'}$. It means that at the exits of β_1 and β_2, the computation sequence from the states (σ, σ') that satisfy $\hat{I}(\beta_1, \beta_1')$ cannot follow any path in $\hat{\Pi}$. This is a contradiction since all assertions in $\mathbb{W}$ are valid. □

EXAMPLE 5.5 Consider again the programs P and P' in Figure 1. We want to verify the basic-block and variable correspondence between $(\mathsf{b}_6, (\mathsf{s}, \mathsf{i}, \mathsf{m}, \mathsf{n}))$ and $(\mathsf{b}_3', (\mathsf{s}', \mathsf{i}', \mathsf{m}', \mathsf{n}'))$. The verification condition $\mathbb{V}$ associated with the correspondence consists of a conjecture $\mathcal{C}$ that includes the following assertions:

$$\rho(b_s, b_s') \Rightarrow \mathsf{m} = \mathsf{m}' \wedge \mathsf{n} = \mathsf{n}', \text{ and}$$
$$\rho(\mathsf{b}_6, \mathsf{b}_3') \Rightarrow \mathsf{m} = \mathsf{m}' \wedge \mathsf{n} = \mathsf{n}' \wedge \mathsf{s} = \mathsf{s}' \wedge \mathsf{i} = \mathsf{i}' \wedge \mathsf{y}' = \mathsf{m}'.$$

The pairs of sets of simple paths considered in generating $\mathbb{V}$ are: $(\Pi_{\mathsf{b}_6,\mathsf{b}_6}, \Pi_{\mathsf{b}_3',\mathsf{b}_3'})$ and $(\Pi_{b_s,\mathsf{b}_6}, \Pi_{b_s',\mathsf{b}_3'})$. These sets of paths are defined as in Example 5.3.

We define the assertion function $\hat{I}$ as follows:

$$\begin{array}{lcl} \hat{I}(b_s, b_s') & = & \rho(b_s, b_s') \wedge \bigwedge \mathcal{C} \\ \hat{I}(\mathsf{b}_6, \mathsf{b}_3') & = & \rho(\mathsf{b}_6, \mathsf{b}_3') \wedge \bigwedge \mathcal{C} \\ \hat{I}(\mathsf{b}_2, \mathsf{b}_2') & = & \top \end{array}$$

Next, we define the set $\hat{\Pi}$ as the union of the cross products of the following pairs of sets: $(\Pi_{\mathsf{b}_6,\mathsf{b}_6}, \Pi_{\mathsf{b}_3',\mathsf{b}_3'})$, $(\Pi_{b_s,\mathsf{b}_6}, \Pi_{b_s',\mathsf{b}_3'})$, $(\Pi_{b_s,\mathsf{b}_2}, \Pi_{b_s',\mathsf{b}_2'})$, and $(\Pi_{\mathsf{b}_6,\mathsf{b}_2}, \Pi_{\mathsf{b}_3',\mathsf{b}_2'})$. It can be proved that all assertions in the weak verification condition $\mathbb{W}$ associated with $\hat{I}$ and $\hat{\Pi}$ are valid. By Theorem 5.4, all assertions in $\mathbb{V}$ are also valid. It means that there exists a basic-block and variable correspondence between $(\mathsf{b}_6, (\mathsf{s}, \mathsf{i}, \mathsf{m}, \mathsf{n}))$ and $(\mathsf{b}_3', (\mathsf{s}', \mathsf{i}', \mathsf{m}', \mathsf{n}'))$. □

Note that in the above example the function $\hat{I}$ is defined on the pair $(\mathsf{b}_2, \mathsf{b}_2')$ although none of the blocks are control blocks. Moreover, the set $\hat{\Pi}$ above includes the pair $(\pi_{b_s,\mathsf{b}_2}, \pi'_{b_s,\mathsf{b}_2'})$ of simple paths but none of them are used to generate the verification condition $\mathbb{V}$. One can actually prove a larger correspondence, that is between $((\mathsf{b}_6, \mathsf{b}_2), (\mathsf{s}, \mathsf{i}, \mathsf{m}, \mathsf{n}))$ and $((\mathsf{b}_3', \mathsf{b}_2'), (\mathsf{s}', \mathsf{i}', \mathsf{m}', \mathsf{n}'))$, and thus $\hat{I}$ can be defined only on pairs of control blocks and $\hat{\Pi}$ includes only pairs of paths used to generate $\mathbb{V}$. By the following theorem, it follows that there exists a basic-block and variable correspondence between $(\mathsf{b}_6, (\mathsf{s}, \mathsf{i}, \mathsf{m}, \mathsf{n}))$ and $(\mathsf{b}_3', (\mathsf{s}', \mathsf{i}', \mathsf{m}', \mathsf{n}'))$.

THEOREM 5.6 *Let P and P' be programs. If there is a basic-block and variable correspondence between $(\bar{b}, \beta, \bar{x})$ of P and $(\bar{b}', \beta', \bar{x})$ of P', or there is a basic-block and variable correspondence between $(\bar{b}, \bar{x}, y)$ of P and $(\bar{b}', \bar{x}, y')$ of P', then there is a basic-block and variable correspondence between $(\bar{b}, \bar{x})$ and $(\bar{b}, \bar{x})$.*

Adding a pair of control blocks or a pair of control variables into a basic-block and variable correspondence often results in a non basic-block and variable correspondence. In such a case, to apply the above theorem, one can always translate programs into SSA form [2], and modify the correspondence according to the variable renaming that occurs during the translation.

In the following example we will show that we can prove a basic-block and variable correspondence using the notions of extendible assertion function and weak verification condition although the verification condition associated with the correspondence cannot be generated.

EXAMPLE 5.7 Consider the programs P and P' in Figure 2. The one-to-one correspondence In between the input variables maps N to N'. There is a basic-block and variable correspondence between $((b_s, \mathsf{b}_6), \mathsf{j})$ and $((b_s', \mathsf{b}_0'), \mathsf{N}')$, and we want to verify this correspondence. The verification condition associated with the

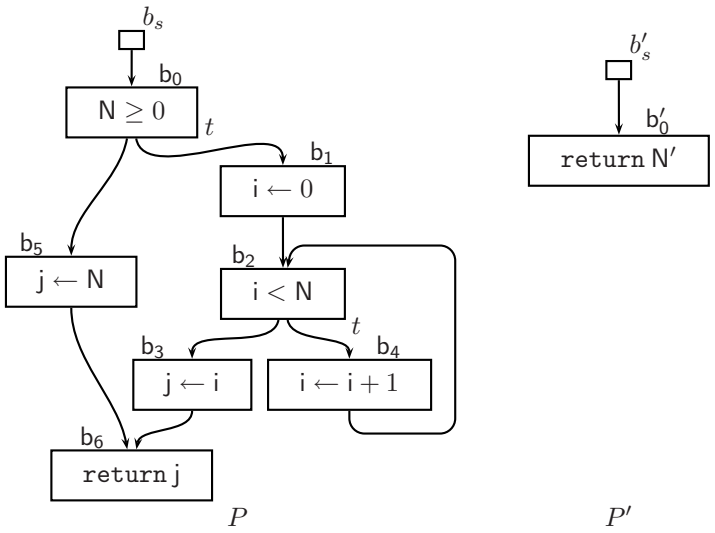

Figure 2: P' is an optimized version of P.

correspondence cannot be generated because there are infinitely many simple paths from b_s to b_6. Adding new pairs of control blocks is not possible because all blocks in P' are already control blocks.

We can prove the correspondence using the notions of extendible assertion function and weak verification condition. Define the assertion function $\hat{I}$ as follows:

$$\begin{array}{rcl} \hat{I}(b_s, b'_s) & = & \mathsf{N} = \mathsf{N}', \quad \hat{I}(\mathsf{b}_6, \mathsf{b}'_0) = \mathsf{j} = \mathsf{N}', \\ \hat{I}(\mathsf{b}_4, b'_s) & = & \mathsf{N} = \mathsf{N}' \wedge \mathsf{N} \geq 0 \wedge \mathsf{i} \leq \mathsf{N}. \end{array}$$

Let the set $\hat{\Pi}$ of paths of (P, P') consist of the following paths:

$$\begin{array}{l} (\pi^{\mathsf{b}_5}_{b_s,\mathsf{b}_6}, \pi_{b'_s,\mathsf{b}'_0}), (\pi^{\mathsf{b}_2,\mathsf{b}_3}_{b_s,\mathsf{b}_6}, \pi_{b'_s,\mathsf{b}'_0}), (\pi_{b_s,\mathsf{b}_4}, \pi_{b'_s}), \\ (\pi_{\mathsf{b}_4,\mathsf{b}_4}, \pi_{b'_s}), (\pi_{\mathsf{b}_4,\mathsf{b}_6}, \pi_{b'_s,\mathsf{b}'_0}), \end{array}$$

where π_b denotes a trivial path consisting only of a single point b. One can prove that all assertions in the weak verification condition associated with $\hat{I}$ and $\hat{\Pi}$ are all valid. Moreover, from $\hat{I}$ and $\hat{\Pi}$, one can reason that there is a basic-block and variable correspondence between $((b_s, \mathsf{b}_6), \mathsf{j})$ and $((b'_s, \mathsf{b}'_0), \mathsf{N}')$. □

5.2 Proof Rule VALIDATE

In this section we discuss how our notion of extendible assertion function can capture inter-program properties described by the proof rule VALIDATE in [15]. The proof rule consists of several steps. First, establish a *control abstraction* κ between programs P and P'. The abstraction is a mapping from $CP_{P'}$ to CP_P, where $CP_{P'}$ is a cut-point set of P' and, additionally, includes the exit block of P'. The set CP_P can be defined similarly. The abstraction κ must map the entry and the exit of P to, respectively, the entry and the exit of P'. Second, for each point p' in $CP_{P'}$, form an intra-program assertion $\alpha_{p'}$ referring only to variables in P'. Next, establish a *data abstraction* δ, which is an assertion relating variables in P and variables in P'.

A path in a program can be considered as a transition relation containing the conditions that enable the path to be traversed and the data transformation effected by the path. For example, consider the program P is Example 1. The path consisting of the instructions

$$\mathsf{i} \ := \ \mathsf{i} + \mathsf{y}; \mathsf{i} < \mathsf{n}; \mathsf{x} \ := \ \mathsf{i}'$$

describes the transition relation

$$\mathsf{i}^* = \mathsf{i} + \mathsf{y} \wedge \mathsf{i}^* < \mathsf{n} \wedge \mathsf{x}^* = \mathsf{i}^*.$$

For simplicity, we denote by π the transition relation described by a path π.

Similar to the proof technique for verifying basic-block and variable correspondences, the proof rule VALIDATE generates assertions that express simulation relation. That is, for each pair (p_1, p_1') of program points such that $\kappa(p_1') = p_1$ and there is a $CP_{P'}$-simple path $\pi_{p_1',p_2'}$ from p_1' to p_2' in the flow graph of P', let $\Pi_{\kappa(p_1'),\kappa(p_2')}$ be the set of all simple paths from $\kappa(p_1')$ to $\kappa(p_2')$ in P, one proves that the following assertion is valid:

$$\alpha_{p_1'} \wedge \delta \wedge \pi_{p_1',p_2'} \Rightarrow \exists V_P^*.(\bigvee_{\pi_{\kappa(p_1'),\kappa(p_2')} \in \Pi_{\kappa(p_1'),\kappa(p_2')}})\pi_{\kappa(p_1'),\kappa(p_2')}) \wedge \delta^* \wedge \alpha_{p_2'}^*, \tag{4}$$

where V_P^* is a sequence of starred version of some variables in P, and δ^* and $\alpha_{p_2'}^*$ are obtained from δ and $\alpha_{p_2'}$ by replacing all variables updated in $\pi_{p_1',p_2'}$ and $\pi_{\kappa(p_1'),\kappa(p_2')}$ by their starred counterparts.

In [15] the correctness of data abstraction δ is proved separately. Essentially, for every two simple paths from p_1' to p_2' and from $\kappa(p_1')$ to $\kappa(p_2')$, one proves that the assertion

$$\alpha_{p_1'} \wedge \delta \wedge \pi_{p_1',p_2'} \wedge \pi_{\kappa(p_1'),\kappa(p_2')} \Rightarrow \delta^* \wedge \alpha_{p_2'}^* \tag{5}$$

is valid.

The notions of extendible assertion function and weak verification condition can capture the program properties described by the rule VALIDATE. First, define an assertion function $\hat{I}$ from the abstractions and intra-program assertions in the rule. Then, reuse the simple paths in the proof rule to generate the weak verification condition associated with the function. Let the assertion function $\hat{I}$ of (P, P') be defined as follows: for every point p' in $CP_{P'}$,

$$\hat{I}(\kappa(p'), p') = \alpha_{p'} \wedge \delta,$$

and $\hat{I}$ is undefined on other pairs of points. Define a set $\hat{\Pi}$ of paths of (P, P') as follows:

$$\hat{\Pi} = \{(\pi, \pi') \mid \exists \Pi_{p_1',p_2'}, \Pi_{\kappa(p_1'),\kappa(p_2')}.\pi' \in \Pi_{p_1',p_2'} \wedge \pi \in \Pi_{\kappa(p_1'),\kappa(p_2')}\}.$$

Note that the definition of $\hat{I}$ is different from $\hat{I}$ discussed in the previous section on basic-block and variable correspondences. The function $\hat{I}$ in this section is only defined on pairs of control points.

Having the function $\hat{I}$ and the set $\hat{\Pi}$, one can prove a property described by rule VALIDATE by proving the weak verification condition associated with $\hat{I}$ and $\hat{\Pi}$.

THEOREM 5.8 *Let $\hat{I}$ and $\hat{\Pi}$ be as defined above, and $\mathbb{W}$ be the weak verification condition associated with $\hat{I}$ and $\hat{\Pi}$. Then, if all assertions in $\mathbb{W}$ are valid, then so are all assertions of the forms (4) and (5).*

PROOF. We first prove that all assertions of the form (5) are valid. Since the assertion $\hat{I}(\kappa(p_1'), p_1')$ is equivalent to $\alpha_{p_1'} \wedge \delta$, the assertion $\pi_{p_1',p_2'} \wedge \pi_{\kappa(p_1'),\kappa(p_2')} \Rightarrow \delta^* \wedge \alpha_{p_2'}^*$ is equivalent to $wlp_{(\pi_{p_1',p_2'},\pi_{\kappa(p_1'),\kappa(p_2')})}(\delta \wedge \alpha_{p_2'})$, and the assertion

$$\hat{I}(\kappa(p_1'), p_1') \Rightarrow wlp_{(\pi_{p_1',p_2'},\pi_{\kappa(p_1'),\kappa(p_2')})}(\delta)$$

is valid, it follows that the assertion

$$\alpha_{p_1'} \wedge \delta \wedge \pi_{p_1',p_2'} \wedge \pi_{\kappa(p_1'),\kappa(p_2')} \Rightarrow \delta^* \wedge \alpha_{p_2'}^*$$

is also valid.

Now, assume that, for some points p_1' and p_2', the assertion

$$\alpha_{p_1'} \wedge \delta \wedge \pi_{p_1',p_2'} \Rightarrow \exists V_P^*.(\bigvee_{\pi_{\kappa(p_1'),\kappa(p_2')} \in \Pi_{\kappa(p_1'),\kappa(p_2')}})\pi_{\kappa(p_1'),\kappa(p_2')}) \wedge \delta^* \wedge \alpha_{p_2'}^*,$$

is not valid. It means that there is a pair (σ, σ') of states satisfying $\alpha_{p_1'} \wedge \delta \wedge \pi_{p_1',p_2'}$, there is a computation sequence from p_1' on σ' traversing the path $\pi_{p_1',p_2'}$, but there is no computation sequence from p_1 on σ traversing any path in $\Pi_{\kappa(p_1'),\kappa(p_2')}$. Since CP_P is a cut-point sets, all paths from $\kappa(p_1')$ to $\kappa(p_2')$ are all in $\Pi_{\kappa(p_1'),\kappa(p_2')}$. By the definition of $\hat{\Pi}$, the path $\pi_{p_1',p_2'}$ is only paired with all paths in $\Pi_{\kappa(p_1'),\kappa(p_2')}$. Thus, there

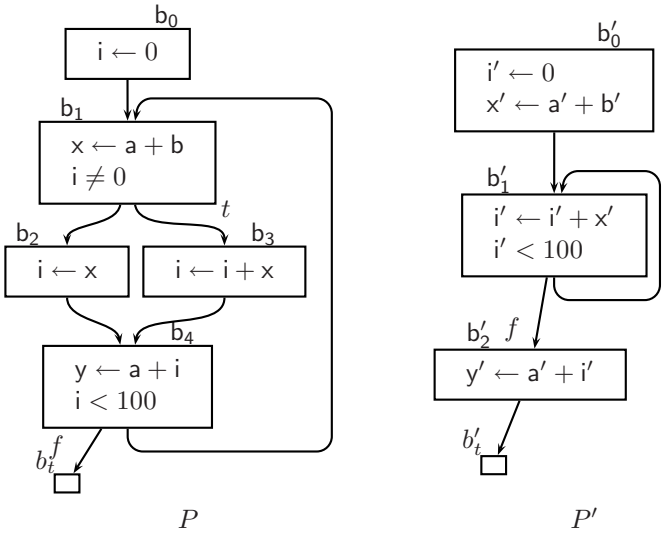

Figure 3: P' is an optimized version of P.

are states (σ, σ') satisfying $\hat{I}(\kappa(p'_1), p'_1)$ but the computation sequence from $(\kappa(p'_1), p'_1)$ on (σ, σ') does not follow any path in $\hat{\Pi}$. However, since all assertions in $\mathbb{W}$ are valid, our assumption is then contradictory. □

EXAMPLE 5.9 In this example we consider programs used as an example in [15]. We depict the programs in Figure 3.

Let us denote the entry point of a basic block by the name of the basic block itself. The control abstraction κ maps b_0 to b'_0, b_1 to b'_1, and b_t to b'_t. The blocks b_t and b'_t are the exit blocks of P and P', respectively. The data abstraction δ is defined as the following assertion:

$$\rho = \kappa(\rho') \wedge \mathsf{i} = \mathsf{i}' \wedge \mathsf{a} = \mathsf{a}' \wedge \mathsf{b} = \mathsf{b}' \wedge (\rho' \neq \mathsf{b}'_1 \Rightarrow \mathsf{x} = \mathsf{x}' \wedge \mathsf{y} = \mathsf{y}'),$$

where ρ is a program counter, and the data abstraction always implies the equality $\rho = \kappa(\rho)$. Furthermore, at the entry of b'_1 we have the assertion $\alpha_{\mathsf{b}'_1}$ equivalent to $\mathsf{x}' = \mathsf{a}' + \mathsf{b}'$.

We define the assertion function $\hat{I}$ as follows:

$$\begin{array}{lcl} \hat{I}(\mathsf{b}_0, \mathsf{b}'_0) & = & \delta \\ \hat{I}(b_t, b'_t) & = & \delta \\ \hat{I}(\mathsf{b}_1, \mathsf{b}'_1) & = & \delta \wedge \alpha_{\mathsf{b}'_1}, \end{array}$$

and $\hat{I}$ is undefined on other pairs of points. The set $\hat{\Pi}$ of paths of (P, P') consists of the following paths:

$$(\pi_{\mathsf{b}_0,\mathsf{b}_1}, \pi_{\mathsf{b}'_0,\mathsf{b}'_1}), (\pi^{\mathsf{b}_2}_{\mathsf{b}_1,\mathsf{b}_1}, \pi_{\mathsf{b}'_1,\mathsf{b}'_1}), (\pi^{\mathsf{b}_3}_{\mathsf{b}_1,\mathsf{b}_1}, \pi_{\mathsf{b}'_1,\mathsf{b}'_1}), (\pi^{\mathsf{b}_2}_{\mathsf{b}_1,b_t}, \pi_{\mathsf{b}'_1,b'_t}), (\pi^{\mathsf{b}_3}_{\mathsf{b}_1,b_t}, \pi_{\mathsf{b}'_1,b'_t}).$$

These pairs of paths are all pairs of simple paths considered in the proof rule VALIDATE. Thus, by Theorem 5.8 if all assertions in the verification condition associated with $\hat{I}$ and $\hat{\Pi}$ are valid, then all assertions generated by VALIDATE are also valid. □

The proof rule VALIDATE cannot prove the inter-program property in Example 5.7. For each pair $\pi_{b'_1,b'_2}$ and $\pi_{\kappa(b'_1),\kappa(b'_2)}$ of simple paths used in generating assertions of the forms (4) and (5), both paths are nontrivial. However, in Example 5.7 the set $\hat{\Pi}$ contains the path $(\pi_{\mathsf{b}_4,\mathsf{b}_4}, \pi_{b_s})$, where π_{b_s} is a trivial path. In this sense, our notions of extendible assertion function and weak verification condition are more powerful than the proof rule VALIDATE.

5.3 Simulation Invariants

We show in this section that the notion of simulation invariant introduced in the work on credible compilation [12] can be captured by our notions of extendible assertion function and weak verification condition.

There are two kinds of invariant introduced in [12], they are standard invariants and simulation invariants. A standard invariant of a program P is written as $\langle\alpha\rangle p$, where α is an assertion and p is a program point of P. The invariant is true if, for all executions of P, the assertion α holds on the state at point p.

Simulation invariants express a simulation relationship between the partial executions of programs. Partial executions of a program are computation sequences starting from the entry of the program. A simulation invariant between two programs P and P' is written as $\langle\alpha, \bar{e}\rangle p \lhd \langle\alpha', \bar{e}'\rangle p'$, where α, α' are assertions, $\bar{e}, \bar{e}'$ are equally long sequences of expressions, and p, p' are program points of P, P', respectively. For sequences $\bar{e} = e_1, \ldots, e_n$ and $\bar{e}' = e'_1, \ldots, e'_n$ of expressions, we write $\bar{e} = \bar{e}'$ for $\bigwedge_{i=1}^{n} e_i = e'_i$. The invariant is true if for all partial executions of P' reaching p' with α' true, there exists a partial execution of P reaching p with α true such that the execution of P is on the same input as that of P' and $\bar{e} = \bar{e}'$.

The notion of standard invariant can be captured by the notion of extendible assertion function. Instead of proving a single standard invariant, in credible compilation one usually proves a set of standard invariants. Given a set S of standard invariants, we define an assertion function $\hat{I}$ as follows: for $\langle\alpha\rangle p \in S$, $\hat{I}(p) = \alpha$. Note that $\hat{I}$ can be defined on other points. We then prove that $\hat{I}$ is a strongly-extendible assertion function.

THEOREM 5.10 *Let S be a set of standard invariants and $\hat{I}$ be an assertion function such that, for all $\langle\alpha\rangle p \in S$, $\hat{I}(p) = \alpha$. If $\hat{I}$ is strongly extendible, then all standard invariants in S are true.* □

The proof of the above theorem is straightforward from the definition of strongly-extendible assertion functions.

The notion of simulation invariant can be captured by the notions of weakly-extendible assertion function and weak verification condition. Similar to proving standard invariants, instead of proving a single simulation invariant, one usually proves a set of simulation invariants. Similar to proving basic-block and variable correspondences and properties described by rule VALIDATE, proving a set of simulation invariants requires some standard invariants that are assumed to be true. Let S be a set of simulation and standard invariants of programs P and P'. Denote by

$$S|p = \{\alpha \mid \exists\langle\alpha\rangle p \in S\}$$

the set of assertions of all standard invariants in S such that the points of the invariants are p. Denote by

$$S|(p, p') = \{\alpha \wedge \alpha' \wedge \bar{e} = \bar{e}' \mid \exists\langle\alpha, \bar{e}\rangle p \lhd \langle\alpha', \bar{e}'\rangle p' \in S\}$$

the set of assertions of all simulation invariants in S such that the pairs of points are (p, p'). We define an assertion function $\hat{I}$ of (P, P') as follows: for every pair (p, p') of points such that $S|(p, p')$ is not empty,

$$\hat{I}(p, p') = \bigwedge S|(p, p') \wedge \bigwedge S|p \wedge \bigwedge S|p'.$$

Let $T_S = \{p' \mid \exists\langle\alpha, \bar{e}\rangle p \lhd \langle\alpha', \bar{e}'\rangle p' \in S\}$ be the set of all program points of P' such that there is a simulation invariant in S involving these points. We assume that the set of all T_S-simple paths is finite. This assumption is also used in the proof rules described in [12] to prove a set of simulation invariants. We define a set $\hat{\Pi}$ of paths of (P, P') with the following requirement: for every T_S-simple path $\pi_{p'_1,p'_2}$, there is a path a path π_{p_1,p_2} in the flow graph of P such that (1) there are simulation invariants $\langle\alpha_1, \bar{e}_1\rangle p_1 \lhd \langle\alpha'_1, \bar{e}'_1\rangle p'_1$ and $\langle\alpha_2, \bar{e}_2\rangle p_2 \lhd \langle\alpha'_2, \bar{e}'_2\rangle p'_2$ in S, and (2) $(\pi_{p_1,p_2}, \pi_{p'_1,p'_2})$ is in $\hat{\Pi}$. Note that the path π_{p_1,p_2} can be trivial.

Having the function $\hat{I}$ and the set $\hat{\Pi}$, one can prove a set of simulation invariants by proving the weak verification condition associated with $\hat{I}$ and $\hat{\Pi}$.

THEOREM 5.11 *Let S, $\hat{I}$, and $\hat{\Pi}$ be as defined above. Let $\mathbb{W}$ be the weak verification condition associated with $\hat{I}$ and $\hat{\Pi}$. If all assertions in $\mathbb{W}$ are valid, then all simulation invariants in S are true.* □

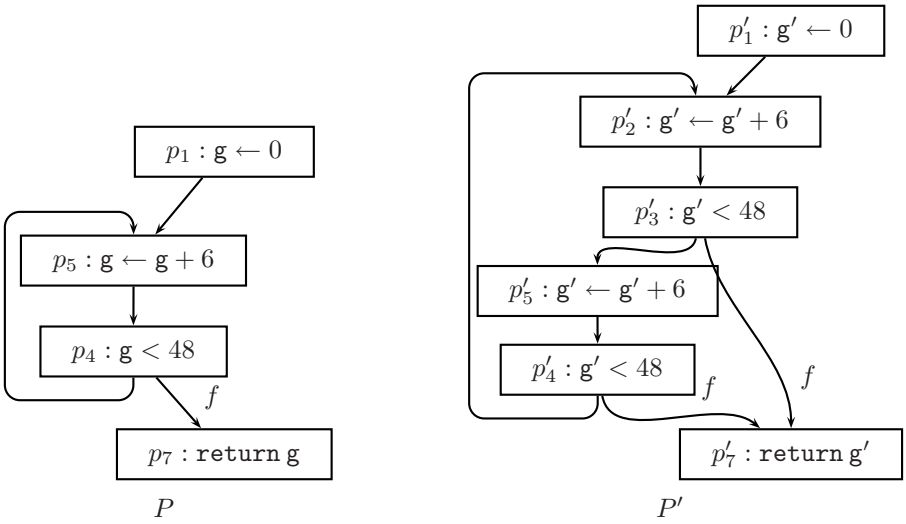

Figure 4: P' is an optimized version of P.

To prove the above theorem, we need to describe the proof rules in [12]. In this section we will only provide an informal proof of the theorem. First, the set T_S contains all program points of P' such that there is a simulation relation there. Since we consider all T_S-simple path, by the requirement of $\hat{\Pi}$, if there is a T_S-simple path $\pi_{p'_1,p'_2}$, then there is a path π_{p_1,p_2} in the flow graph of P such that there are simulation invariants at (p_1, p'_1) and at (p_2, p'_2). Thus, the paths in $\hat{\Pi}$ represents the directions that the rules in [12] follow in the proof.

EXAMPLE 5.12 We consider an example taken from the work on credible compilation in [12]. The programs in the example are depicted in Figure 4. Each block in the programs has only one instruction, and the instruction is labelled with a point denoting the entry of the block. The set S of simulation and standard invariants to be proved consists of the following invariants:

$$
\begin{array}{l}
\langle \mathrm{g}\%12 = 0 \vee \mathrm{g}\%12 = 6\rangle p_4, \langle \mathrm{g}'\%12 = 0\rangle p'_4, \langle \mathrm{g}'\%12 = 6\rangle p'_3 \\
\langle \mathrm{g}\%12 = 0, \mathrm{g}\rangle p_5 \lhd \langle \top, \mathrm{g}'\rangle p'_2, \\
\langle \mathrm{g}\%12 = 6, \mathrm{g}\rangle p_4 \lhd \langle \top, \mathrm{g}'\rangle p'_3, \\
\langle \mathrm{g}\%12 = 6, \mathrm{g}\rangle p_5 \lhd \langle \top, \mathrm{g}'\rangle p'_5, \\
\langle \mathrm{g}\%12 = 0, \mathrm{g}\rangle p_4 \lhd \langle \top, \mathrm{g}'\rangle p'_4, \\
\langle \top, \mathrm{g}\rangle p_7 \lhd \langle \top, \mathrm{g}'\rangle p'_7.
\end{array}
$$

The assertion function $\hat{I}$ is defined as follows:

$$
\begin{array}{lcl}
\hat{I}(p_5, p'_2) & = & \mathrm{g}\%12 = 0 \wedge \mathrm{g} = \mathrm{g}' \\
\hat{I}(p_4, p'_3) & = & \mathrm{g}\%12 = 6 \wedge \mathrm{g} = \mathrm{g}' \wedge \mathrm{g}'\%12 = 6 \wedge (\mathrm{g}'\%12 = 0 \vee \mathrm{g}\%12 = 6) \\
\hat{I}(p_5, p'_5) & = & \mathrm{g}\%12 = 6 \wedge \mathrm{g} = \mathrm{g}' \\
\hat{I}(p_4, p'_4) & = & \mathrm{g}\%12 = 6 \wedge \mathrm{g} = \mathrm{g}' \wedge \mathrm{g}'\%12 = 0 \wedge (\mathrm{g}'\%12 = 0 \vee \mathrm{g}\%12 = 6) \\
\hat{I}(p_7, p'_7) & = & \mathrm{g} = \mathrm{g}'
\end{array}
$$

Both at the pair of exit points and at the pair of entry points, $\hat{I}$ is defined as $\top$. The set $\hat{\Pi}$ of paths of (P, P') consists of the following paths:

$$
\begin{array}{l}
(\pi_{p_1,p_5}, \pi_{p'_1,p'_2}), (\pi_{p_5,p_4}, \pi_{p'_2,p'_3}), (\pi_{p_4,p_5}, \pi_{p'_3,p'_5}), \\
(\pi_{p_5,p_4}, \pi_{p'_5,p'_4}), (\pi_{p_4,p_7}, \pi_{p'_4,p'_7}), (\pi_{p_4,p_7}, \pi_{p'_3,p'_7}).
\end{array}
$$

It is easy to see that if all assertions in the weak verification condition $\mathbb{W}$ associated with $\hat{I}$ and $\hat{\Pi}$ are valid, then all simulation assertions in S are true. Assume that all assertions in $\mathbb{W}$ are valid. The invariant

$\langle \top, \mathrm{g} \rangle p_7 \lhd \langle \top, \mathrm{g}' \rangle p'_7$ is true by the following reasoning. For every path π' going to p'_7, there is a path π going to p_7 such that (π, π') is in $\hat{\Pi}$ and $\hat{I}$ is defined on the starts of π and π'. For the path $\pi_{p'_4,p'_7}$, we have the path π_{p_4,p_7} such that $(\pi_{p_4,p_7}, \pi_{p'_4,p'_7})$ in $\hat{\Pi}$ and $\hat{I}$ is defined on (p_4, p'_4). Since all assertions in $\mathbb{W}$ are valid, we can ignore the assertions defined on (p_4, p'_4) and on (p_7, p'_7). For the path $\pi_{p'_3,p'_7}$, we have the path π_{p_4,p_7} such that $(\pi_{p_4,p_7}, \pi_{p'_3,p'_7})$ is in $\hat{\Pi}$ and $\hat{I}$ is defined on (p_4, p'_3). Assuming that the invariants $\langle \mathrm{g}\%12 = 0, \mathrm{g} \rangle p_4 \lhd \langle \top, \mathrm{g}' \rangle p'_4$ and $\langle \mathrm{g}\%12 = 6, \mathrm{g} \rangle p_4 \lhd \langle \top, \mathrm{g}' \rangle p'_3$ are true, we can prove inductively that, for all partial executions of P' and the executions reach p'_7, there is a partial execution of P' such that the execution reaches p_7 and the values of g and g′ on reaching p_7 and p'_7 coincide. □

Recall that we require that, for every path (π, π') in the set $\hat{\Pi}$, the path π' is not trivial. Due to this requirement, the proof rules described [12] cannot prove the inter-program property in Example 5.7. Thus, with our notions of extendible assertion function and weak verification condition, we can prove more inter-program properties than that of the proof rules in [12].

The notion of simulation invariant used in credible compilation is similar to the notion of simulation triple in Necula's work on translation validation [10]. Thus, our notions of extendible assertion function and weak verification condition can capture the notion of simulation triple as well.

6 Common Criteria Certification

We discuss in this section an application of our theory of inter-program properties in the certification of smart-card applications. The work described in this section is part of an industrial project, called EDEN2, that has been conducted at Verimag laboratory.[1] The aim of the project are twofold: (1) to develop a method for software certification in the framework of Common Criteria certification [1], and (2) to provide a certificate or a collection of certificates showing that a smart-card application follows its specification or a model of its specification.

Common Criteria (CC) is an international standard for the evaluation of security related systems. CC defines requirements for certification: security policy model (SPM), functional specification (FSP), high-level design (HLD), low-level design (LLD), and implementation (IMP). Given a specification of a system or a program, an SPM is a model of the specification. an FSP describes an input-output relationship of the system. HLDs are often fused into FSPs or into LLDs. An LLD itself is described as a reference implementation.[2] The IMP is the code implementing the system.

Each requirement in CC has a representation. For example, in EDEN2 project the SPM is written in a declarative language that specifies, for each smart-card command, the normal behavior of the command and the actions that the command has to perform when a card tear (or power loss) occurs. The FSP and the LLD in EDEN2 project are programs written in subsets of Java, while the IMP are Java Card programs [3, 14]. The HLD in EDEN2 is fused into the LLD. Essentially, the SPM, the FSP, the LLD, and the IMP are programs that can be represented as program-point flow graphs. Between every two consecutive requirement representations there is a so-called representation correspondence (RCR). An RCR is essentially a property relating two programs, or an inter-program property, and thus we can apply our theory of inter-program properties to proving RCRs and providing certificates about the RCRs. Our theory is also applicable to proving properties of SPMs, FSPs, LLDs, and IMPs. In this report we focus on the application of the theory to proving RCRs.

The definitions of RCRs between two consecutive requirements are different. We first discuss the definition of RCRs between SPMs and FSPs. To this end, we discuss the SPM and the FSP. An SPM and an FSP consist of a set of commands. A command can be thought of as a method in a Java program or a function in a C program. For each command in the SPM and the FSP, the command can be represented by two programs, one program specifies the normal behavior of the command and the other specifies what the command has to do when a card tear occurs. For simplicity, we call the former program the *normal fragment* of the command and the latter one the *abrupt fragment* of the command. In the FSP the normal

[1] Industrial partners involved in this project include companies that work on security for embedded systems, e.g., Gemalto and Trusted Logic.

[2] In the latest version of Common Criteria report [1], HLD and LLD are replaced by TOE design description (TDS). In this report we regard LLDs as TDSs.

and abrupt fragments are represented by a try-catch construct. The try part represents the normal fragment, the catch part catches a special exception and represents the abrupt fragment.

The operation of SPMs and FSPs resembles the operation of smart-card applications, that is, by sending a sequence of commands to the SPMs and the FSPs. One can think of an SPM or an FSP as a program that takes as an input a sequence of commands of the form $C(a_1, \ldots, a_n)$, where C is the command's name and $a_1, \ldots, a_n$ are input arguments. A run of an SPM or an FSP can be described as a sequence of runs of commands. For each run of a command, if no card tears occur and the run of the command terminates normally, then the run of the SPM or the FSP fetches the next input $C(a_1, \ldots, a_n)$ from the input sequence. If a card tear occurs, then the run goes to the abrupt fragment of the command. If the run of the abrupt fragment then terminates, the run of the command is said to terminate abruptly, and in turn the run of the SPM or the FSP simply terminates.

A run of an SPM or an FSP is a finite or infinite alternating sequence

$$\gamma_0, \varepsilon_1, \gamma_2, \varepsilon_2, \ldots,$$

where

- γ_0 is an entry configuration;
- for all $i \geq 0$, we have $\gamma_i \mapsto \gamma_{i+1}$; and
- for all $j \geq 1$, the event ε_j is an event associated with transition $\gamma_{j-1} \mapsto \gamma_j$.

We assume that each of the SPM and the FSP has an input variable, and the state of configuration γ_0 maps this variable to the input value, which is a sequence of commands. Later in the definition of RCRs between SPMs and FSPs we introduce a one-to-one correspondence Obs between the set of observable variables of an SPM and the set of observable variables of an FSP. We assume that Obs maps the input variable of the SPM to the input variable of the FSP.

For every run of a command, upon reaching the exit of normal fragment, the run of an SPM or an FSP emits either a $Pass$ event or a $Fail$ event, and upon reaching the exit of abrupt fragment, the run emits an $Abrupt$ event. We assume that emitting an event is the same as assigning the event to a special variable ε. Events are not restricted to $Pass$, $Fail$, and $Abrupt$ events; we allow internal or unobservable events.

We now define the notion of RCR between SPMs and FSPs that we use in EDEN2. Let E be a set of observable events. Denote by $R|_E$ the subsequence of R consisting only of events in E:

$$\begin{array}{lcl} R & = & (p_0, \sigma_0), \varepsilon_1, (p_1, \sigma_1), \varepsilon_2, \ldots \\ R|_E & = & (p_0, \sigma_0), \varepsilon_{i_1}, (p_{i_1}, \sigma_{i_1}), \varepsilon_{i_2}, (p_{i_2}, \sigma_{i_2}), \end{array}$$

where $\varepsilon_{i_j} \in E$ for all j. Let X be a set of variables of an SPM, we denote by $Ab(X)$ the set of variables in X such that the variables are modified in the abrupt fragment of the SPM.

DEFINITION 6.1 Let O_{SPM} and O_{FSP} be the sets of observable variables of, respectively, an SPM and an FSP such that there is a one-to-one correspondence Obs between O_{SPM} and O_{FSP}. Let $E_O = \{Pass, Fail, Abrupt\}$ be the set of observable events of the SPM and the FSP. There is an *RCR between the SPM and the FSP* if, for every run

$$R|_{E_O} = (p_0, \sigma_0), \varepsilon_{i_1}, (p_{i_1}, \sigma_{i_1}), \ldots$$

of the FSP, there is a run

$$R'|_{E_O} = (p'_0, \sigma'_0), \varepsilon'_{j_1}, (p'_{j_1}, \sigma'_{j_1}), \ldots$$

of the SPM, where for all $x \in O_{SPM}$, we have $\sigma_0(x) = \sigma'_0(Obs(x))$, such that, for all k

- $\varepsilon_{i_k} = \varepsilon'_{j_k}$,
- if $\varepsilon_{i_k} \neq Abrupt$, then $\sigma_{i_k}(x) = \sigma'_{j_k}(Obs(x))$ for all $x \in O_{SPM}$,
- if $\varepsilon_{i_k} = Abrupt$, then $\sigma_{i_l}(y) = \sigma'_{j_l}(Obs(y))$ for all $y \in Ab(O_{SPM})$.

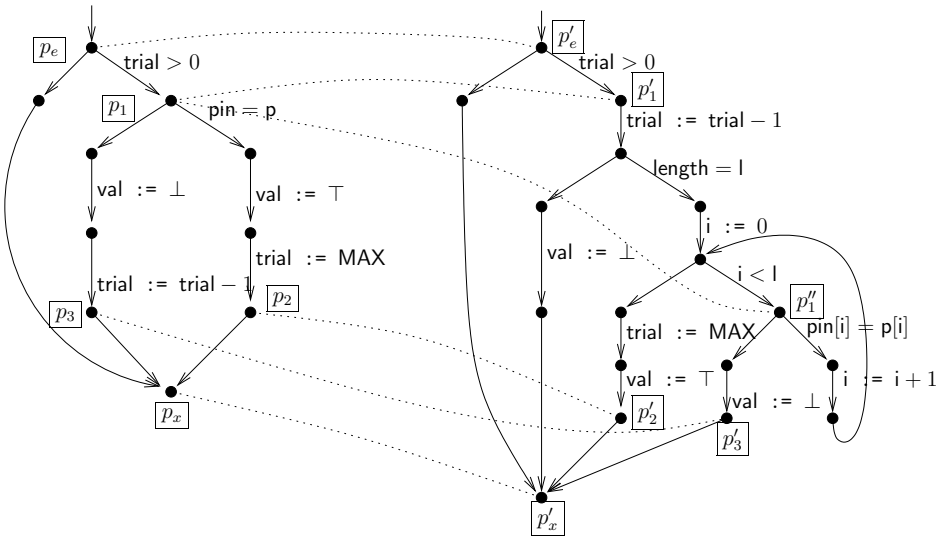

Figure 5: P_1 is on the left and P'_1 is on the right.

□

To apply the theory of inter-program properties to proving an RCR between an SPM and an FSP, we prove the RCR between each corresponding commands separately. Let Obs be a one-to-one correspondence between observable variables of the SPM and of the FSP. There is an RCR between the SPM and the FSP of a command C if the following conditions hold. For any run R of the command C in the FSP from a state σ_1, there is a run R' of the same command in the SPM from a state σ'_1 such that σ_1 and σ'_1 satisfy $\bigwedge_{x \in O_{SPM}} x = Obs(x)$, and

- if R is terminating (or the run reaches the exit of normal or abrupt fragment), then so is R',
- when R and R' are terminating with, respectively, states σ_2 and σ'_2, R and R' emit the same event ε such that
  - if $\varepsilon \neq Abrupt$, then σ_2 and σ'_2 satisfy $\bigwedge_{x \in O_{SPM}} x = Obs(x)$;
  - otherwise σ_2 and σ'_2 satisfy $x = Obs(x)$ for all $x \in Ab(O_{SPM})$.

EXAMPLE 6.2 In this example we will show that there is an RCR between the SPM and the FSP of the command checkPIN used for PIN verification. Let us first consider the flow graphs representing the normal fragments of the SPM and of the FSP. Call the former flow graph P_1 and the latter P'_1. These flow graphs are depicted in Figure 5. The edge (p_2, p_x) emits a $Pass$ event, while other edges coming to p_x emits a $Fail$ event. Similarly, the edge (p'_2, p'_x) emits a $Pass$ event, while other edges coming to p'_x emit a $Fail$ event.

For clarity, we assume that the SPM and the FSP have disjoint sets of variables. To this end, we consider that all variables in the FSP are in primed notation. Let the sets

$$\begin{array}{lcl} O_{SPM} & = & \{\mathsf{trial}, \mathsf{pin}, \mathsf{p}, \mathsf{val}, \mathsf{MAX}, \varepsilon\} \\ O_{FSP} & = & \{\mathsf{trial}', \mathsf{pin}', \mathsf{p}', \mathsf{val}', \mathsf{MAX}', \varepsilon'\} \end{array}$$

be the sets of observable variables of, respectively, the SPM and the FSP such that a one-to-one correspondence Obs between O_{SPM} and O_{FSP} maps each variable in O_{SPM} to its primed counterpart in O_{FSP}

Note that pin in the SPM has a scalar type but pin′ in the FSP has an array type. So, we have to define the equivalence between pin and pin′. First, every array PIN p has a length l associated with the array; we write the association as a pair (p, l). We introduce a predicate $\equiv$ between such pairs such that, given an

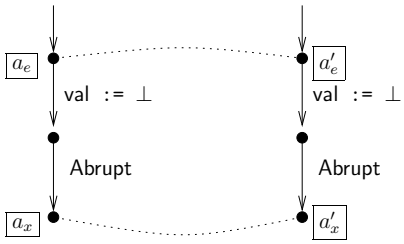

Figure 6: P_2 is on the left and P'_2 is on the right.

array PINs p, p' and lengths l, l', we say that $(p, l) \equiv (p', l')$ if $l = l'$ and for all $i = 0, \ldots, l-1$, we have $p[i] = p'[i]$. Next we introduce a predicate $\sim$ between scalar PINs and array PINs. The predicate $\sim$ is axiomatized as follows: for every scalar PINs w, x and for every array PINs y, z,

$$x \sim y \Rightarrow (y \equiv z \Leftrightarrow x \sim z)$$
$$x \sim y \Rightarrow (w = x \Leftrightarrow w \sim y).$$

The predicate $\sim$ defines the equality between a scalar PIN and an array PIN.

The following assertions express the correspondence between observable variables of the SPM and of the FSP:

$$\begin{array}{lcllcl} \phi_1 & \Leftrightarrow & \mathsf{trial} = \mathsf{trial}' & \phi_4 & \Leftrightarrow & \mathsf{p} \sim (\mathsf{p}', \mathsf{l}') \\ \phi_2 & \Leftrightarrow & \mathsf{val} = \mathsf{val}' & \phi_5 & \Leftrightarrow & \mathsf{MAX} = \mathsf{MAX}' \\ \phi_3 & \Leftrightarrow & \mathsf{pin} \sim (\mathsf{pin}', \mathsf{length}') & \phi_6 & \Leftrightarrow & \varepsilon = \varepsilon' \end{array}$$

Next, we define an assertion function $\hat{I}_1$ of (P_1, P'_1) as follows:

$$\begin{array}{lcl} \hat{I}_1(p_e, p'_e) & = & \bigwedge_{i=1}^{6} \phi_i \\ \hat{I}_1(p_1, p'_1) & = & \bigwedge_{i=1}^{6} \phi_i \wedge \mathsf{trial} > 0 \\ \hat{I}_1(p_1, p''_1) & = & \bigwedge_{i=2}^{6} \phi_i \wedge \mathsf{trial} > 0 \wedge \mathsf{trial} = \mathsf{trial}' + 1 \\ & & \wedge \mathsf{length}' = \mathsf{l}' \wedge \mathsf{i}' < \mathsf{l}' \wedge (\forall j . 0 \leq j < \mathsf{i}' \Rightarrow \mathsf{pin}'[j] = \mathsf{p}'[j]) \\ \hat{I}_1(p_2, p'_2) & = & \bigwedge_{i=1}^{6} \phi_i \wedge \mathsf{pin} = \mathsf{p} \wedge (\mathsf{pin}, \mathsf{length}) \equiv (\mathsf{p}, \mathsf{l}) \\ \hat{I}_1(p_3, p'_3) & = & \bigwedge_{i=1}^{6} \phi_i \wedge \mathsf{pin} \neq \mathsf{p} \wedge (\mathsf{pin}, \mathsf{length}) \not\equiv (\mathsf{p}, \mathsf{l}) \\ \hat{I}_1(p_x, p'_x) & = & \bigwedge_{i=1}^{6} \phi_i \end{array}$$

The function $\hat{I}_1$ is undefined elsewhere.

Denote a path from point p to q in a program-point flow graph by $\pi_{p,q}$. We define a set $\hat{\Pi}_1$ of paths of (P_1, P'_1) such that the set consists of the following paths:

$$\begin{array}{l} (\pi_{p_e,p_1}, \pi_{p'_e,p'_1}), (\pi_{p_e,p_x}, \pi_{p'_e,p'_x}), (\pi_{p_1}, \pi_{p'_1,p''_1}), (\pi_{p_1,p_x}, \pi_{p'_1,p'_x}), \\ (\pi_{p_1,p_2}, \pi_{p'_1,p'_2}), (\pi_{p_1}, \pi_{p''_1,p''_1}), (\pi_{p_1,p_2}, \pi_{p''_1,p'_2}), (\pi_{p_1,p_3}, \pi_{p''_1,p'_3}), \\ (\pi_{p_2,p_x}, \pi_{p'_2,p'_x}), (\pi_{p_3,p_x}, \pi_{p'_3,p'_x}). \end{array}$$

One can prove that all assertions in the weak verification condition associated with $\hat{I}_1$ and $\hat{\Pi}_1$ are valid.

We now consider the flow graphs of the abrupt fragments of the SPM and of the FSP. Call the former one P_2 and the latter P'_2. These flow graphs are depicted in Figure 6. We define an assertion function $\hat{I}_2$ of (P_2, P'_2) as follows:

$$\begin{array}{lcl} \hat{I}_2(a_e, a'_e) & = & \top \\ \hat{I}_2(a_x, a'_x) & = & \mathsf{val} = \mathsf{val}'. \end{array}$$

The function $\hat{I}_2$ is undefined elsewhere. One can prove easily that all assertions in the weak verification condition associated with $\hat{I}_2$ and $\hat{\Pi}_2$ are valid.

From the assertion functions $\hat{I}_1$, $\hat{I}_2$ and from the sets $\hat{\Pi}_1$, $\hat{\Pi}_2$, one can easily see that there is an RCR between the SPM and the FSP of the command checkPIN. First, since for all $p \neq p_x$, $\hat{I}_1(p, p'_x)$

is undefined and the set $\{\pi' \mid \exists\pi.(\pi,\pi') \in \hat{\Pi}_1\}$ is the set of all simple paths induced by the points in P_1', then if for a pair of runs R' of P_1' and R of P_1 such that the entry configurations of the runs satisfy $\hat{I}_1(p_e, p_e')$, then if the run R' is terminating, then so is the run of R. Since $\hat{I}_1$ is weakly extendible and the assertion $\hat{I}_1(p_x, p_x') \Rightarrow \bigwedge_{x\in O_{SPM}} x = Obs(x)$ is valid, the exit configurations of both runs satisfy $\bigwedge_{x\in O_{SPM}} x = Obs(x)$.

Now, if a card tear occurs then the run R' will go to the entry of P_2'. Since the $\hat{I}_2(a_e, a_e')$ is valid, the run R can also go to the entry P_2 such that the entry configurations of the runs R and R' on reaching the entries of P_2 and P_2' satisfy $\hat{I}_2(a_e, a_e')$. With the same reasoning as above, if R' reaches a_e', then R reaches a_e too. Since $\hat{I}_2$ is weakly extendible and the assertion $\hat{I}_2(a_x, a_x') \Rightarrow \bigwedge_{x\in Ab(O_{SPM})} x = Obs(x)$ is valid, then the exit configurations of both runs satisfy $\bigwedge_{x\in Ab(O_{SPM})} x = Obs(x)$. Therefore, there is an RCR between the SPM and the FSP of the command.

□

We now focus on RCRs between FSPs and LLDs. Before discussing RCRs, we first describe LLDs. In EDEN2 the language used to write an LLD is a subset of Java. This subset includes memory characteristics and transaction mechanism of Java Card [14, 3]. First, in the language of LLDs there are two kinds of memory, persistent memory and transient memory. The difference between these kinds of memory is the following: when power is lost (or a card tear occurs), data stored in the persistent memory will be kept in the memory, while data stored in the transient memory will be lost. In the sequel, variables whose values are stored in the persistent memory are called *persistent variables*, and variables whose values are stored in the transient memory are called *transient variables*.

Similar to the FSP, an LLD consists of a set of commands where each command is a Java method. Card tears are capture using a try-catch construct where the try part represents the normal fragment of the LLD and the catch part catches a special exception and represents the abrupt fragment of the LLD. The language of LLDs offers a transaction mechanism that resembles the transaction mechanism of Java Card API. Our modelling of transactions follows the modelling of Java Card transactions in [6]. We introduce a boolean variable inTrans to keep track if a transaction is in progress or not. When a transaction begins, the value of inTrans is set to true, and when it ends, the value of inTrans is set to false. One can set the value of inTrans to false to escape from a transaction. This feature is useful for variables whose updates must be unconditional.

Similar to FSPs, an LLD is a program that takes as an input a sequence of command calls of the form $C(a_1, \ldots, a_n)$, where C is the command's name and $a_1, \ldots, a_n$ are input arguments. The notion of run of LLDs is the same as the notion of run of FSPs.

Having described LLDs, we now define RCRs between FSPs and LLDs. Let us first denote by $Pr(X)$ the set of persistent variables in the set X of variables of an LLD. Later in the definition of RCRs between an FSP and an LLD we require that observable persistent variables of the LLD are updated in the same order as their counterparts of the FSP. But, when a transaction is in progress, then such an order becomes irrelevant. For example, given a one-to-one correspondence Obs between observable variables of the LLD and of the FSP, if no transaction is in progress and the observable persistent variables of the LLD are updated in the order x_1, x_2, x_3, then their counterparts are updated in the order $Obs(x_1), Obs(x_2), Obs(x_3)$. However, when a transaction is in progress, then the order of updating $Obs(x_1), Obs(x_2), Obs(x_3)$ is irrelevant. Moreover, whether a transaction is in progress or not, each variable is updated with the same value as its counterpart. To this end, first, for each persistent variable x of the LLD and its counterpart $Obs(x)$ of the FSP, we associate with both variables an event function $Write\_x$. This function takes as an input the value v of x or $Obs(x)$ and returns an event $Write\_x(v)$. The following assertion axiomatizes the event function:

$$\forall x, y, v, w.(Write\_x(v) = Write\_y(w) \Leftrightarrow Write\_x = Write\_y \wedge v = w),$$

where the equality $Write\_x = Write\_y$ denotes a syntactic equality. In the sequel we denote by τ_x the domain of variable x.

Second, the set of events emitted by the LLD is a power set of the set of events emitted by the FSP. Next, assignments to observable persistent variables and committing transactions emit events in the following way:

- In the try part of the FSP, the update of a variable y, where $y = Obs(x)$ for an observable persistent variable x in the LLD, emits $Write\_x(v)$, where v is the updated value of y.
- In the try part of the LLD,
  - if no transaction is in progress, that is the variable inTrans is false, then the update of an observable persistent variable x emits $\{Write\_x(v)\}$, where v is the updated value of x;
  - if a transaction is in progress, that is the variable inTransaction is true, then when inTransaction is set to false and beforehand the observable persistent variables $x_0, \ldots, x_n$ are updated such that the *latest* updated values of these variables are, respectively, $v_0, \ldots, v_n$, then if the resetting of inTrans is not caused by aborting the in-progress transaction, then the resetting emits $\{Write\_x_0(v_0), \ldots, Write\_x_n(v_n)\}$. However, when the resetting of inTrans is caused by aborting the in-progress transaction or no observable variables are updated, then no set of events is emitted.

For comparing events of the LLD and events of the FSP, we say that a nonempty set $\{\varepsilon_0, \ldots, \varepsilon_m\}$ of LLD's events *matches* a sequence $\varepsilon'_0, \ldots, \varepsilon'_n$ of FSP's events if (1) $m = n$, and (2) for all $i = 0, \ldots, m$, there exists j such that $0 \leq j \leq n$ and $\varepsilon'_j = \varepsilon_i$. Now, we say that a sequence $\hat{\varepsilon_1}, \hat{\varepsilon_2}, \ldots$ of sets of LLD's events *matches* a sequence $\varepsilon'_1, \varepsilon'_2, \ldots$ of FSP events if either both sequences are of length 0, or there is an increasing sequence $n_1 < n_2 < \ldots$ of positive integers such that

1. $\hat{\varepsilon_1}$ matches $\varepsilon'_1, \ldots, \varepsilon'_{n_1}$, and
2. for all $i \geq 2$, $\hat{\varepsilon_i}$ matches $\varepsilon'_{n_{i-1}+1}, \ldots, \varepsilon'_{n_i}$.

Note that the one-to-one correspondence Obs maps variables of the LLD to variables of the FSP. We assume that the FSP and the LLD have disjoint sets of variables. In the sequel, for simplicity, the inverse of Obs is called Obs as well. That is, for any variable x of the LLD and any variable x' of the FSP, $x' = Obs(x)$ if and only if $x = Obs(x')$.

DEFINITION 6.3 Let O_{FSP} and O_{LLD} be the sets of observable variables of, respectively, an FSP and a LLD, and Obs be a one-to-one correspondence between these sets. Let the sets

$$\begin{array}{rcl} E_{FSP} & = & \{Pass, Fail, Abrupt\} \\ & & \cup\{Write\_x(v) \mid x \in Pr(O_{LLD}) \wedge v \in \tau_{Obs(x)}\} \\ E_{LLD} & = & \{\{Pass\}, \{Fail\}, \{Abrupt\}\} \\ & & \cup(\mathcal{P}(\{Write\_x(v) \mid x \in Pr(O_{LLD}) \wedge v \in \tau_x\}) - \{\emptyset\}) \end{array}$$

be the sets of observable events of the FSP and of the LLD, respectively. There is an *RCR between the FSP and the LLD* if, for every run

$$R|_{E_{LLD}} = (p_0, \sigma_0), \varepsilon_{i_1}, (p_{i_1}, \sigma_{i_1}), \ldots$$

of the LLD, there is a run

$$R'|_{E_{FSP}} = (p'_0, \sigma'_0), \varepsilon'_{j_1}, (p'_{j_1}, \sigma'_{j_1}), \ldots$$

of the FSP, where for all $x \in O_{LLD}$, we have $\sigma_0(x) = \sigma'_0(Obs(x))$, such that there is an increasing sequence $n_1 < n_2 < \ldots$ of positive integers such that

1. ε_{i_1} matches $\varepsilon'_{j_1}, \ldots, \varepsilon'_{j_{n_1}}$, and
2. for all $k > 1$, ε_{i_k} matches $\varepsilon'_{j_{n_{k-1}+1}}, \ldots, \varepsilon'_{j_{n_k}}$,

and

- for all l, if $\varepsilon_{i_l} \neq \{Pass\} \neq \{Fail\} \neq \{Abrupt\}$, then $\sigma_{i_l}(y) = \sigma'_{j_{n_l}}(Obs(y))$ for all $y \in Pr(O_{LLD})$; otherwise
- $\sigma_{i_l}(x) = \sigma'_{j_{n_l}}(Obs(x))$ for all $x \in O_{LLD}$.

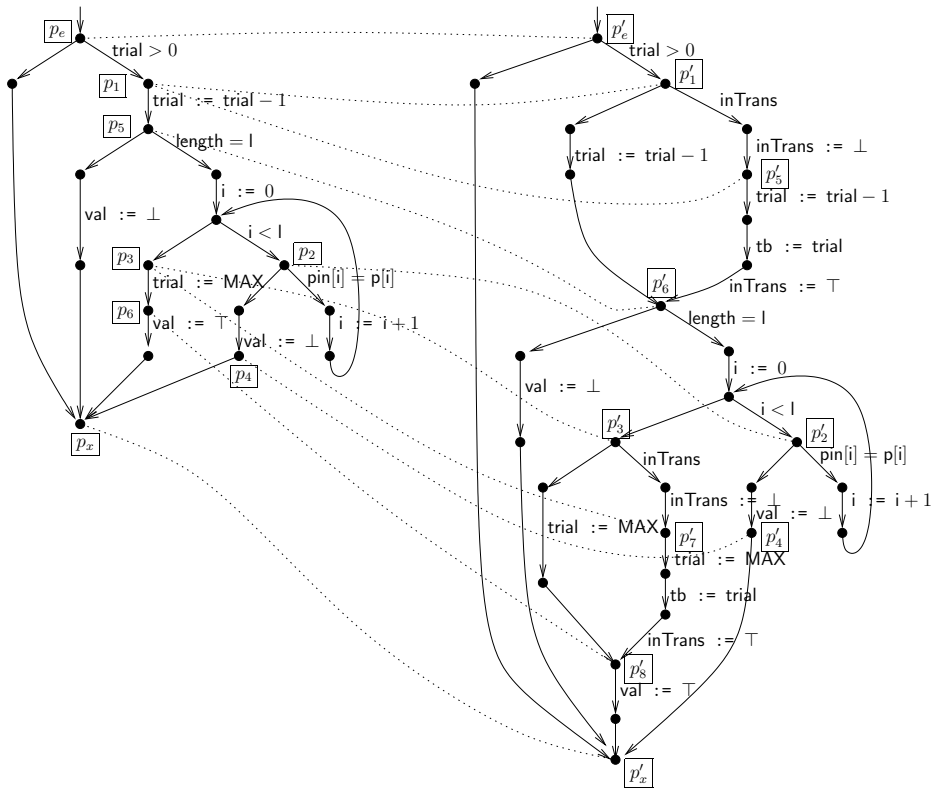

Figure 7: P_1 is on the left and P'_1 is on the right.

□

Similar to the RCR between an SPM and an FSP, we use the special variable ε to store the events emitted by the FSP and the LLD. For the RCR between an FSP and a LLD, emitting an event means concatenating the event to the current value of the special variable ε. Particularly for the LLD, we use another special variable ε_t to keep track the updated observable persistent variables when a transaction is in progress. When the variable inTrans is set to true, the variable ε_t is set to the empty set. During the transaction, any update to an observable persistent variable x with value v is recorded by updating ε_t with $\varepsilon_t \cup \{Write\_x(v)\}$. When the variable inTrans is set to false, the variable ε is set to $\varepsilon; \varepsilon_t$ only if the reseting of inTrans is not caused by aborting the in-progress transaction. Moreover, when the LLD emits a $Pass$ or $Fail$ event, and a transaction is in progress, then ε is updated with $\varepsilon; \varepsilon_t; Pass$ or $\varepsilon; \varepsilon_t; Fail$, respectively. When a card tear occurs and the LLD emits $Abrupt$, then the content of ε_t is discarded and ε is updated with $\varepsilon; Abrupt$. When an observable persistent variable is updated more than once in a transaction, then one can always translate the LLD into SSA form [2] such that in the program texts there is only one assignment to each variable.

Similar to RCRs between SPMs and FSPs, we apply the theory of inter-program properties to proving an RCR between an FSP and an LLD by proving the RCR between each corresponding commands separately.

EXAMPLE 6.4 We consider again the command checkPIN in this example. Figure 7 depicts the FSP and the LLD of the try parts of the command checkPIN. The flow graph of the FSP is called P_1 and is on the lefthand side of the figure, while the other flow graph is the flow graph of the LLD and it is called P'_1. Persistent variables in P'_1 are trial, pin, length, MAX. Other variables are transient. The variable tb is a backup variable for the variable trial.

Let the set

$$O_{FSP} = \{\mathsf{trial}, \mathsf{pin}, \mathsf{length}, \mathsf{p}, \mathsf{l}, \mathsf{val}, \mathsf{MAX}, \varepsilon\}$$

be the set of observable variables of the FSP and the set O_{LLD} be the set of observable variables of the LLD such that O_{LLD} consists of the primed counterparts of all variables in O_{FSP}. The one-to-one correspondence Obs between O_{FSP} and O_{LLD} maps each variable in O_{FSP} to its primed counterpart in O_{LLD}. We express the relationship of observable variables by the following assertions:

$$\begin{array}{lcl} \phi_1 & \Leftrightarrow & \mathsf{pin} = \mathsf{pin}' \wedge \mathsf{length} = \mathsf{length}' \wedge \mathsf{MAX} = \mathsf{MAX}' \wedge \mathsf{trial} = \mathsf{trial}' \\ \phi_2 & \Leftrightarrow & \mathsf{p} = \mathsf{p}' \wedge \mathsf{l} = \mathsf{l}' \wedge \mathsf{val} = \mathsf{val}' \wedge \varepsilon = \varepsilon' \\ \phi & \Leftrightarrow & \phi_1 \wedge \phi_2 \wedge (\mathsf{inTrans}' \Rightarrow \mathsf{trial} = \mathsf{tb}') \end{array}$$

The assertions ϕ_1 and ϕ_2 describe the correspondence of, respectively, persistent and transient variables.

We define an assertion function of (P_1, P_1') as follows:

$$\begin{array}{l} \hat{I}_1(p_e, p_e') = \hat{I}_1(p_1, p_1') = \hat{I}_1(p_1, p_5') = \hat{I}_1(p_5, p_6') = \hat{I}_1(p_2, p_2') \\ = \hat{I}_1(p_3, p_3') = \hat{I}_1(p_3, p_7') = \hat{I}_1(p_4, p_4') = \hat{I}_1(p_6, p_8') = \hat{I}_1(p_x, p_x') = \phi \end{array}$$

Let $S_1 = \{p \mid \exists p', \varphi. \hat{I}_1(p, p') = \varphi\}$ and $S_1' = \{p' \mid \exists p, \varphi. \hat{I}_1(p, p') = \varphi\}$. be the sets of program points on which $\hat{I}_1$ is defined. Given a set S of program points in a flow graph, we say that a path $\pi = p_0, \ldots, p_n$ in the flow graph is S-simple if $n > 0$, p_0 and p_n are in S, and none of $p_1, \ldots, p_{n-1}$ are in S.

We define a set $\hat{\Pi}_1$ of paths of (P_1, P_1') as follows: for every S_1'-simple path $\pi_{p',q'}$,

- there is an S_1-simple path $\pi_{p,q}$ such that $\hat{I}_1(p, p')$ and $\hat{I}_1(q, q')$ are defined, or
- there is a trivial path π_p, where $p \in S_1$, such that $\hat{I}_1(p, p')$ and $\hat{I}_1(p, q')$ are defined.

One can easily prove that the assertions in the verification condition associated with $\hat{I}_1$ and $\hat{\Pi}_1$ are valid, and thus $\hat{I}_1$ is weakly extendible.

We next consider the catch parts of the command updatePIN. The flow graphs P_2 and P_2' in Figure 8 are the catch parts of the command. Note that the catch part P_2 of the FSP is different from the one shown on the righthand side of Figure 6. The flow graph P_2 in Figure 8 updates the variables p and l. The counterparts of these variables in the LLD are transient variables,[3] and so on abrupt they are set to their default values. Nevertheless, one can easily define an assertion function of the flow graph P_2 in Figure 8 and the flow graph P_2 of the SPM in Figure 6 such that there is still an RCR between the SPM and the FSP of the command checkPIN.

We define an assertion function $\hat{I}_2$ of (P_2, P_2') as follows:

$$\begin{array}{lcl} \hat{I}_2(a_e, a_e') & = & \phi_1 \wedge \mathsf{p} = \mathsf{p}' \wedge \varepsilon = \varepsilon' \wedge (\mathsf{inTrans}' \Rightarrow \mathsf{trial} = \mathsf{tb}') \\ \hat{I}_2(a_1, a_1') & = & \phi_1 \wedge \mathsf{p} = \mathsf{p}' \wedge \mathsf{val} = \mathsf{val}' \wedge \varepsilon = \varepsilon' \\ \hat{I}_2(a_x, a_x') & = & \phi. \end{array}$$

Note that the assertions $\phi \Rightarrow \hat{I}_2(a_e, a_e')$ and $\hat{I}_2(a_e, a_e') \Rightarrow \bigwedge_{x \in Pr(O_{LLD})} x = Obs(x) \wedge \varepsilon = \varepsilon'$ are valid. Moreover, since the set S_1' above covers P_1', by the weak-extendibility of $\hat{I}_1$, it follows that for every finite run of P_1', there is a finite run of P_1 such that the initial configurations of the runs satisfy ϕ and the last configurations of the runs satisfy $\hat{I}_2(a_e, a_e')$.

Let $S_2' = \{p' \mid \exists p, \varphi. \hat{I}_2(p, p') = \varphi\}$ be the set of points in P_2' such that for each point p' in S_2', there is a point p in P_2 and $\hat{I}_2(p, p')$ is defined. Similarly, let $S_2 = \{p \mid \exists p', \varphi. \hat{I}_2(p, p') = \varphi\}$. Let $\Pi_{S_2'}$ be the set of all S_2'-simple paths and Π_{S_2} be the set of all S_2-simple paths. We define a set $\hat{\Pi}_2$ of paths of (P_2, P_2') as follows:

$$\hat{\Pi}_2 = \{(\pi_{p,q}, \pi_{p',q'}) \mid \exists \varphi_1, \varphi_2. (\pi_{p,q}, \pi_{p',q'}) \in \Pi_{S_2} \times \Pi_{S_2'} \text{ and } \hat{I}_1(p, p') = \varphi_1 \text{ and } \hat{I}_1(q, q') = \varphi_2\}.$$

One can prove that the assertions in the weak verification condition associated with $\hat{I}_2$ and $\hat{\Pi}_2$ are valid.

From the assertion functions $\hat{I}_1, \hat{I}_2$ and the sets $\hat{\Pi}_1, \hat{\Pi}_2$, and the weak extendibility of $\hat{I}_1$ and $\hat{I}_2$, one can easily see that there is an RCR between the FSP and the LLD of the command checkPIN. □

[3] Stack variables are transient variables.

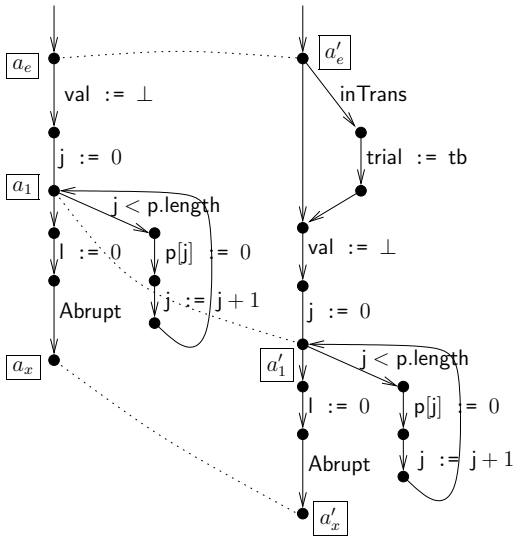

Figure 8: P_2 is on the left and P'_2 is on the right.

7 Conclusion

We have developed a theory of inter-program properties. The theory forms a basis for describing and proving properties between two programs. The cornerstone of the theory is the notion of weak verification condition, by which one can provide certificates certifying the inter-program properties. The theory itself is abstract and general, in the sense that it can be applied to programs written in any programming languages as long as these languages have the weakest precondition property.

We have applied the theory in the translation validation for optimizing compilers and in Common Criteria certification. In translation validation, we have shown that, using the notions of extendible assertion function and weak verification condition, we can capture different notions of correspondence used in different translation validation work. We have also shown that we can prove the equivalence of two programs in the presence of optimizations that introduce or eliminate loops. In Common Criteria certification, we have shown that the theory can be applied to two programs written in different languages, and the theory can also provide certificates certifying representation correspondences between requirements in Common Criteria.

References

[1] *Common Criteria for Information Technology Security Evaluation*. Version 3.1, CCMB-2007-09-003. 1, 6, 2

[2] B. Alpern, M.N. Wegman, and F.K. Zadeck. Detecting equality of variables in programs. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages (POPL 1988)*, pages 1–11, 1988. 5.1, 6

[3] Z. Chen. *Java Card Technology for Smart Cards*. The Java Series. Addison-Wesley, 2000. 6, 6

[4] Robert W. Floyd. Assigning meaning to programs. In J. T. Schwartz, editor, *Proceedings of Symposium in Applied Mathematics*, pages 19–32, 1967. 1

[5] C. A. R. Hoare. An axiomatic basis for computer programming. *CACM*, 12(10):576–580, 1969. 1

[6] E.-M.G.M. Hubbers and E. Poll. Reasoning about card tears and transactions in Java Card. In M. Wermelinger and T. Margaria-Steffen, editors, *Fundamental Approaches to Software Engineering*,

7th International Conference, FASE 2004, volume 2984 of *LNCS*, pages 114–128. Springer-Verlag, 2004. 6

[7] Jacques Leockx, Kurt Sieber, and Ryan D. Stansifer. *The Foundations of Program Verification (2nd ed.)*. John Wiley & Sons, Inc., New York, NY, USA, 1987. 3.2

[8] I. Narasamdya and A. Voronkov. Finding basic block and variable correspondence. In *Proceedings of the 12th International Static Analysis Symposium (SAS)*, 2005. 1, 5, 5.1

[9] Iman Narasamdya. *Establishing Program Equivalence in Translation Validation for Optimizing Compilers*. PhD thesis, The University of Manchester, 2007. Downloadable at `http://www-verimag.imag.fr/~narasamd/NarasamdyaThesis.ps`. 5.1, 5.1

[10] George C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN Conference on Principles of Programming Languages Design and Implementation (PLDI)*, pages 83–95, June 2000. 5, 5.3

[11] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. *LNCS*, 1384, 1998. 1, 5

[12] M. Rinard and D. Marinov. Credible compilation with pointers. In *Proceedings of the FLoC Workshop on Run-Time Result Verification*, Trento, Italy, July 1999. 5, 5.3, 5.3, 5.3, 5.12, 5.3

[13] Xavier Rival. Symbolic transfer function-based approaches to certified compilation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–13. ACM Press, 2004. 5

[14] Sun Micro systems, Inc, Palo Alto, California. *Java Card 3.0 Platform Specification*, 2008. `http://java.sun.com/javacard/3.0/`. 6, 6

[15] Lenore D. Zuck, Amir Pnueli, and Benjamin Goldberg. VOC: A methodology for the translation validation of optimizing compilers. *J. UCS*, 9(3):223–247, 2003. 5, 5.2, 5.2, 5.9