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

Certification of Smart-Card Applications in Common Criteria: *Proving Representation Correspondences*

Iman Narasamdya, Michaël Périn

Verimag Research Report n^o TR-2008-18

November 2008

Certification of Smart-Card Applications in Common Criteria: *Proving Representation Correspondences*

Iman Narasamdya, Michaël Périn

November 2008

Abstract

We present a method for proving representation correspondences in the Common Criteria (CC) certification of smart-card applications. For security policy enforcement, the CC defines a chain of requirements: a security policy model (SPM), a functional specification (FSP), and a target-of-evaluation design (TDS). In our approach to the CC certification, these requirements are models of applications that can have different representations. A representation correspondence (RCR) describes a correlation between the representations of two adjacent requirements. One task in the CC certification is to demonstrate formal proofs of RCRs. We first develop a modelling framework by which the representations of SPM, FSP and TDS can be described uniformly as models of an application. We then define RCRs as mutual simulations between two application models over sets of observable events and variables. We describe a proof technique for proving RCRs and providing certificates about them based on assertions relating two models at specific locations. We show how RCRs can help us prove property preservation from the SPM to the FSP and the TDS.

Keywords: Software Certification, Common Criteria, Program Invariants, Inter-Program Properties

Reviewers: Laurent Mounier

Notes:

How to cite this report:

@techreport { ,
title = { Certification of Smart-Card Applications in Common Criteria: *Proving Representation Correspondences*},
authors = { Iman Narasamdya, Michaël Périn},
institution = { Verimag Research Report },
number = {TR-2008-18},
year = { },
note = { }
}

1 Introduction

The use of smart card and smart-card applications has been pervasive in our everyday lives. Smart-card applications are programs embedded in the chip on the smart cards. These application have mainly been used to provide security functions, in particular user authentication and authorization. These functions are specified by a security policy. Since the fulfillment of this policy is paramount for a smart-card application, to give high confidence to the users, smart-card application vendors need to provide an assurance that an implementation of the application satisfies the policy. The process of providing such an assurance is often referred to as certification process.

We describe in this report our work on developing a method for formal certification of smart-card applications in the framework of Common Criteria (CC) Com [2007]. This work is part of an industrial project called EDEN2.[1] The CC is an international standard for the evaluation of security related systems. It guarantees that a target of evaluation (TOE), or a system, enforces security policies by means of an assurance architecture. For assurances in the development process, this architecture consists of a chain of requirements starting from the model of the policies at the start of the chain, to the low-level design and the implementation of the system at the end of the chain.

At the highest level of the CC certification, which is called evaluation assurance level 7 (or EAL7), the following chain of requirements are needed in the assurance architecture: (1) a formal security model (SPM), (2) a formal functional specification of security functions (FSP), and (3) a TOE design (TDS). The SPM models the policy independently of the implementation, the FSP describes input-output relationships of security functions, and the TDS is a low-level design that is close to the implementation. A representation correspondence (RCR) demonstrates the correlation between each two adjacent requirements in the chain. The CC EAL7 certification consists of proving that the SPM, the TDS, and the FSP satisfy the security policies, and providing certificates about this satisfaction. In addition, the CC EAL7 also requires formal proofs of RCRs between the SPM and the FSP, and between the FSP and the TDS.

In this report we are concerned with proving RCRs and providing certificates about them. We present a method for proving RCRs in the context of smart-card applications. First, we develop a framework for modelling smart-card applications such that the formal models capture the operations of the applications, in particular our model allows one to reason about card tears (or power loss) and transaction mechanism that are present in smart-card applications. In this framework, a model of an application consists of a set of command procedures (or simply command). Each command is presented by two transition graphs (or control-flow graphs), one describes the normal behavior of the command and the other describes what the command has to perform when a card tear occurs. The FSP and the TDS are essentially models of an application. In EDEN2, the SPM consists of two entities: one entity is a model of the application and the other is a set of assertions (or formulas) in some logic such that the assertions describe security properties. In the sequel, we refer to the former entity when we speak about SPM. Card readers communicate with a smart-card application by sending a sequence of commands. We model this interaction with a main procedure that takes as the only input a sequence of commands, and for each command, the procedure calls the corresponding command procedure in the application. The semantics of an application is then characterized by the set of the main procedure's runs.

We define RCRs between two application models as bisimulation equivalence consisting of mutual simulations between the models over observable events and variables. To this end, given two models S and T of an application, we associate with S and T the same set of observable events and for each event we associate a mapping between observable variables. Intuitively, we say that there is an RCR between S and T if for every run of S, there is a run of T on the same input, and vice versa, such that (1) both runs exhibit the same sequence of observable events, and (2) for each two equal events, the values of corresponding observable variables coincide. Having a unified model for smart-card applications allows us to have only a single definition of RCRs such that the definition is applicable for RCRs between the SPM and the FSP, and between the FSP and the TDS. Furthermore, we will show that our definition of RCR helps us prove *property preservation* from one model to the other. That is, as required by the CC EAL7 certification, the RCRs must guarantee that all security properties satisfied by the SPM are satisfied by the FSP and the TDS.

We develop a proof technique for proving RCRs. We prove RCRs between S and T by proving the

[1] Research and industrial partners include Verimag, CEA, Gemalto, and Trusted-Logic; see http://www.eden-rntl.org.

RCR between each corresponding commands in S and T. We apply a theory of inter-program properties described in Voronkov and Narasamdya [2008] to proving RCRs. Inter-program properties are properties relating two programs. RCRs are essentially inter-program properties. We prove RCRs by using assertions that describe data abstraction and control mapping between the transition graphs of the corresponding commands. The theory also provides a notion of certificate about inter-program properties. Such a certificate is essential to the CC EAL7 certification.

Proving RCRs are challenging due to nontrivial data abstractions between application models and due to language features in which the models are written. Consider a command checkPIN used to authenticate users by checking an input PIN against the PIN stored on the card. The security policy does not require the PIN to be in some specific format. Thus, in the SPM the PIN can simply be a natural number. For security, the command uses variables trial as a trial-remaining counter. If the input PIN does not match the stored PIN, then trial is decremented, and if it gets 0, then the PIN is blocked. In the FSP, developers usually take defensive measures. The PIN in the FSP is now an array of natural numbers, and prior to checking the input PIN, the variable trial must be decremented. We then have the following excerpts of over-simplified checkPIN, the SPM and the FSP are on the lefthand and righthand, respectively:

```
if (pin ≠ input) {                trial := trial − 1;
  trial := trial − 1;             while (i < length) {
  return fail;                      if (pin[i] ≠ input[i])
}                                     return fail;
                                  }
```

If we associate an event with every update of trial, then in the SPM this event occurs at the end of command execution, but in the FSP it occurs at the beginning. Thus, we may end up with different sequences of observable events. This poses some difficulties in determining observable events in RCRs. Note that in the SPM and the FSP above, the data abstraction introduces a loop in the FSP. To prove that for every run of the SPM there is a "corresponding" run of the FSP, one has to prove that the loop will not yield non-terminating run. We will show later that in the presence of transaction mechanism, we sometimes have to relax the definition of RCR. That is, we only require that for every run of the TDS, there is a corresponding run of the FSP.

In summary the contributions of this report is a method for proving representation correspondences as a part of the CC EAL7 certification of smart-card applications.

The outline of this report is the following. We first discuss our framework for formally modelling smart-card applications. We then develop a notion of representation correspondence based on this framework. Afterward we describe briefly the theory of inter-program properties. Then, we discuss our proof technique for proving RCRs based on the theory. We then show how RCRs allow us to preserve property in the chain of the CC requirements. Finally, we discuss some related work and conclude this report.

2 Formal Models and Representation Correspondences

2.1 Transition Graphs and Computation Sequences

A smart-card application is a program consisting of $m+1$ procedures: $main$, $c_1, \ldots, c_m$, where $main$ is the main procedure and $c_1, \ldots, c_m$ are command procedures. In the sequel command procedures are often called *commands*. Each procedure P consists of a finite set of *program points* and is presented as two disjoint *transition graphs* (or *program-point flow graphs*) $\mathbf{G}_P^n$ and $\mathbf{G}_P^a$. A transition graph is a finite directed graph whose nodes are program points. Each edge of a transition graph is labelled with a guard, an assignment instruction, a goto instruction (or a skip instruction), or a procedure call. The transition graph $\mathbf{G}_P^n$ describes the normal behavior of P, while the transition graph $\mathbf{G}_P^a$ describes what the application has to do when a card tear occurs during the execution of P.

We assume that every transition graph $\mathbf{G}_P$ has a unique entry point, denoted by $entry(\mathbf{G}_P)$ and a unique exit point, denoted by $exit(\mathbf{G}_P)$. As such, every procedure P has a unique entry point $entry(P) = entry(\mathbf{G}_P^n)$, and two exit points, *normal exit point* $exit_n(P) = exit(\mathbf{G}_P^n)$ and *abrupt exit point* $exit_a(P) = exit(\mathbf{G}_P^a)$.

The main procedure takes as input a sequence of input commands. In turn, the procedure reads each

input command of the form $(C, \bar{v})$, where C is the command name and $\bar{v}$ are the input values for C. For each input command $(C, \bar{v})$, the main procedure calls the corresponding command C on input $\bar{v}$, or call $C(\bar{v})$.

We introduce a restriction on command procedures, that is, for every command procedure P, the graphs $\mathbf{G}_P^n$ and $\mathbf{G}_P^a$ do not contain edges labelled with procedure calls. Similarly, the graph $\mathbf{G}_{main}^a$ does not contain such edges. This restriction does not limit the applications that can be modelled in our framework. Procedures called by command procedures in smart-card applications are usually not recursive and thus can be inlined. For technical reason, we assume that, for every command procedure, $\mathbf{G}_{main}^n$ contains an edge labelled with a call to the procedure.

We describe the run-time behavior of an application as sequences of configurations. A *configuration* of a run is a pair (p, σ) where p is a program point and σ is a *state* mapping variables to values. Given a procedure P, a configuration (p, σ) is a called an *entry configuration for* P if p is an entry point of P, a *normal exit configuration for* P if p is a normal exit point of P, and an *abrupt exit configuration for* P if p is an abrupt exit point of P

The semantics of an application is defined as a transition relation with transitions of the form $(p_1, \sigma_1) \overset{l}{\mapsto} (p_2, \sigma_2)$, where (p_1, σ_1) and (p_2, σ_2) are configurations and l is a transition label. Transitions are of the following kinds:

- Intra-graph transition, where the pair (p_1, p_2) is an edge of a transition graph, l is the label of the edge such that l is not a procedure call.
- Call transition is a transition where $p_2 = entry(P)$ of a procedure P such that there is a call edge (p_1, p) labelled with $P(y_1, \ldots, y_n)$ in the transition graph containing point p_1, and there is a procedure P whose input parameters are $x_1, \ldots, x_n$. The state σ_2 maintains all global variables in σ_1 and, for all $i = 1, \ldots, n$, the state σ_2 maps x_i to the value of y_i. The label l of the transition is $P(y_1, \ldots, y_n)$.
- Return transition is a transition where $p_1 = exit_n(P)$ for a procedure P such that there is a call transition $(p_3, \sigma_3) \overset{l'}{\mapsto} (p_4, \sigma_4)$ where $p_4 = entry(P)$ and (p_3, p_2) is a call edge labelled with $P(\bar{y})$. The state σ_2 maintains all global variables in σ_1, and maps designated variables to the values of the return variables in σ_1. The label l is a special label ret.
- Abrupt transition is a transition where p_1 belongs to $\mathbf{G}_P^n$ for a procedure P, $p_2 = entry(\mathbf{G}_P^a)$, and $\sigma_1 = \sigma_2$.
- Abrupt transition, where p_1 is in $\mathbf{G}_P^n$, p_2 is $entry(\mathbf{G}_P^a)$, l is a special label ab, and $\sigma_1 = \sigma_2$.

We allow labels of transitions (or edges of transition graphs) to be associated with events, which means that the transitions emit the events. We will use a special event variable ε to store emitted events. That is, if a transition emits an event E, then it is the same as an assignment of E to ε.

We use the following assumptions for transition relations. First, for every procedure P, every point p in $\mathbf{G}_P^n$, and every state σ, there is a transition $(p, \sigma) \overset{l}{\mapsto} (entry(\mathbf{G}_P^a), \sigma)$. That is, a card tear can occur non-deterministically. Second, there is no transition from an exit configuration (p, σ), where $p = exit_a(P)$ for every procedure P, or $p = exit_n(main)$. Third, intra-graph transitions are deterministic. Forth, transitions are atomic.

A *computation sequence* of an application A is either a finite or an infinite sequence of

$$(p_0, \sigma_0) \overset{l_1}{\mapsto} (p_1, \sigma_1) \overset{l_2}{\mapsto} (p_2, \sigma_2) \ldots$$

where, for all i, the transition $(p_i, \sigma_i) \overset{l_{i+1}}{\mapsto} (p_{i+1}, \sigma_{i+1})$ is justified by a transition in the transition relation of A. When a computation sequence is finite, then it ends with a configuration. A *run of a procedure P in A from a state σ_0* is a computation sequence of A such that $p_0 = entry(P)$. For every run of a command procedure P, the run terminates when it reaches an exit configuration for P, and can only terminate in such a configuration. We say that the run *terminates normally* (*terminates abruptly*) if the final configuration is a normal (abrupt) exit configuration for P. A *run of an application A from a state σ* is a run of the

procedure $main$ from σ. Especially for $main$, a run of $main$ *terminates normally* if the final configuration is a normal exit configuration for $main$, and *terminates abruptly* if the final configuration is an abrupt exit configuration for any procedure. A *run of a transition graph G in an application A* is a computation sequence of A such that $p_0 = entry(G)$ and for all i, the pairs (p_i, p_{i+1}) is an edge of the graph.

2.2 Representation Correspondences

For our discussion on representation correspondences (RCRs), we assume that we are given two models S and T of an application, where T is an implementation of S. That is, S and T can be, respectively, an SPM and an FSP, or they can be, respectively an FSP and a TDS. For simplicity, we assume that each command in S has a corresponding command, with the same name, in T, and vice versa. We assume further that S and T have disjoint sets of transition graphs and disjoint sets of variables.

To define RCRs, we associate with both S and T the same set of observable events, and for each observable event we associate a one-to-one correspondence between observable variables of S and T at the start or final configurations of the transitions that emit the event. Intuitively, there is an RCR between S and T if for every run of T, there is a run of S on the same input, such that (1) both runs terminate or generate infinite computation sequences, (2) these runs exhibit the same sequence of observable events, (3) the values of corresponding observable variables in the configurations of each corresponding events coincide, and (4) vice versa for every run of S.

We first discuss the set of observable events. For every procedure P, we associate every incoming edge into $exit_n(P)$ with either a Pass_P or a Fail_P events. The first event denotes a successful completion of a run of P, while the latter denotes a logic failure. We associate every incoming edge into $exit_a(P)$ with an Abrupt_P event and every call transition to a procedure P with a Call_P event.

Next, we associate one-to-one correspondences between observable variables for events. For each command procedure P and for every configuration γ such that there is a configuration γ' and $\gamma' \overset{l}{\mapsto} \gamma$ where l is associated with Pass_P, we associate with γ a set O_S of observable variables if γ belongs to an S's run, and a set O_T if γ belongs to a T's run, such that there is a one-to-one correspondence Obs between O_S and O_T. Similarly for l associated with Fail_P and Abrupt_P. When l is associated with Call_P, then, instead of γ, we associate O_S and O_T with γ' such that if the parameters of P in S and in T are, respectively, $\bar{x} = x_1, \ldots, x_m$ and $\bar{y} = y_1, \ldots, y_n$, then $m = n$, $\{x_1, \ldots, x_m\} \subseteq O_S$ and $\{y_1, \ldots, y_n\} \subseteq O_T$, and Obs maps x_i to y_i for all $i = 1, \ldots, m$. We also associate entry configurations of $main$ with the sets O_S and O_T such that the input variables of S and T are mapped to each other.

We associate *observation function $\mathcal{O}$* with each S and T to identify observable configurations and transition labels. That is, for a configuration γ, the function $\mathcal{O}(\gamma) = \gamma$ if γ is associated with a set of observable variables, otherwise $\mathcal{O}(\gamma) = \perp$. Similarly, for a label l of a transition, $\mathcal{O}(l) = e$ if l emits an observable event e, otherwise $\mathcal{O}(l) = \perp$. An *observation sequence* of a computation sequence R, denoted by $o(R)$, is obtained by turning R into an alternating sequence of configurations and transition labels, and applying the observation function $\mathcal{O}$ to each configuration and transition label of R. That is, for a computation sequence $R = \gamma_0 \overset{l_1}{\mapsto} \gamma_1 \overset{l_2}{\mapsto} \gamma_2 \overset{l_3}{\mapsto} \ldots$, we have $o(R) = \mathcal{O}(\gamma_0), \mathcal{O}(l_1), \mathcal{O}(\gamma_1), \mathcal{O}(l_2), \mathcal{O}(\gamma_2), \mathcal{O}(l_3), \ldots$. A *$\perp$-free observation sequence* of a computation sequence R, denoted by $o_\perp(R)$ is obtained from $o(R)$ by suppressing $\perp$ in $o(R)$.

We say that two states σ_1 and σ_2 are *compatible* with respect to a one-to-one correspondence Obs between the sets O_1 and O_2 of observable variables in the domain of, respectively, σ_1 and σ_2 if for every $x \in O_1$, we have $\sigma_1(x) = \sigma_2(Obs(x))$. Two configurations $\gamma_1 = (p_1, \sigma_1)$ and $\gamma_2 = (p_2, \sigma_2)$ are *compatible* if there are sets O_1 and O_2 of observable variables associated with γ_1 and γ_2 such that (1) there is a one-to-one correspondence Obs between O_1 and O_2, and (2) σ_1 and σ_2 are compatible with respect to Obs.

DEFINITION 2.1 We say that two computation sequences R_1 and R_2 are *observationally equivalent* (or *stuttering equivalent*) if, let

$$o_\perp(R_1) = \theta_1, \theta_2, \ldots \qquad o_\perp(R_2) = \theta_1, \theta'_2, \ldots,$$

$o_\perp(R_1)$ and $o_\perp(R_2)$ are of the same length, and for all i, we have either (1) $\theta_i = \gamma$ and $\theta'_i = \gamma'$, for configurations γ and γ', such that γ and γ' are compatible, or (2) $\theta_i = \theta'_i$. □

DEFINITION 2.2 There is a *representation correspondence* between a procedure P of S and a procedure P' of T if for every run R of P from a configuration γ, there is a run R' of P' from a configuration γ', where γ and γ' are compatible, and vice versa, such that R and R' are observationally equivalent.

There is a *representation correspondence* between S and T if there is a representation correspondence between $main$ of S and $main$ of T. □

In the above definition, due to call transitions and our assumption that $\mathbf{G}^n_{main}$ contains at least a call edge for every command procedure, the configurations γ and γ' have sets of observable variables associated with them. Note that to have γ and γ' compatible, then the procedures P and P' must refer to the same command. The notion of RCR for procedures is useful for proving RCR between S and T. Since $main$ can be thought of as a loop that read input command and call the command, then proving RCR between S and T can be reduced to proving RCR between each corresponding commands.

3 Theory of Inter-Program Properties

In this section we describe an abstract theory for describing and proving properties that relate two programs. Such properties are called *inter-program properties*. A detailed description of the theory can be found in Voronkov and Narasamdya [2008]. The theory deals with programs that are represented as transition graphs described in the previous section.

For describing and proving inter-program properties, the theory considers two programs P_1 and P_2 as a pair (P_1, P_2), such that they have disjoint flow graphs and disjoint sets of variables. A state σ for the pair (P_1, P_2) can be considered as a pair $(\sigma_1, \sigma_2) = \sigma$, such that σ_1 is for P_1 and σ_2 is for P_2. A configuration is a tuple $(p_1, p_2, \sigma_1, \sigma_2)$ such that (p_1, σ_1) is a configuration for P_1 and (p_2, σ_2) is a configuration for P_2. The semantics of (P_1, P_2) is a transition relations containing two kinds of transitions:

1. $(p_1, p_2, \sigma_1, \sigma_2) \mapsto (p'_1, p_2, \sigma'_1, \sigma_2)$, such that $(p_1, \sigma_1) \mapsto (p'_1, \sigma'_1)$ is in P_1;
2. $(p_1, p_2, \sigma_1, \sigma_2) \mapsto (p_1, p'_2, \sigma_1, \sigma'_2)$, such that $(p_2, \sigma_2) \mapsto (p'_2, \sigma'_2)$ is in P_2.

In the description of the theory in this section, we omit the transition labels for simplicity. Thus, a computation sequence is simply a sequence of configurations.

The theory assumes an *assertion language* and uses relation $\sigma \models \alpha$ to mean that the state σ satisfies the assertion α. For a configuration $\gamma = (p, \sigma)$, we write $\gamma \models \alpha$ for $\sigma \models \alpha$. An assertion is *valid* if it is satisfied by any state.

The formalization of the theory is based on the notion of assertion function. An *assertion function* of (P_1, P_2) is a partial function

$$I : \mathbf{Point}_{P_1} \times \mathbf{Point}_{P_2} \to \mathbf{Assertion}$$

mapping pairs of program points of (P_1, P_2) to assertions, such that I is defined on $(entry(P_1), entry(P_2))$ and $(exit(P_1), exit(P_2))$. This requirement is technical as one can always define I on these pairs as $\top$. Assertions defined on such an I are called *inter-program assertions*. Given a pair of points $\hat{p}$ and a pair of states $\hat{\sigma}$ of (P_1, P_2), we say that $\hat{p}$ is I-observable if $I(\hat{p})$ is defined. For a configuration $\gamma = (\hat{p}, \hat{\sigma})$, we write $\gamma \models I$ if $I(\hat{p})$ is defined and $\hat{\sigma} \models I(\hat{p})$.

The theory introduces the notion of weakly-extendible assertion function as a well-suited notion for describing inter-program properties.

DEFINITION 3.1 Let I be an assertion function of a pair (P_1, P_2) of programs. The function I is *weakly extendible* if every run

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

of (P_1, P_2), such that $i \geq 0$, $\gamma_0 \models I$, $\gamma_i \models I$, and γ_i is not an exit configuration, can be extended to a run

$$\gamma_0, \ldots, \gamma_i, \ldots, \gamma_{i+n}$$

such that (1) $n > 0$, and (2) $\gamma_{i+n} \models I$. □

EXAMPLE 3.2 We illustrate in this example the notion of weak extendibility. Consider the following two programs P_1, on the right, and P_2, on the left:

```
i := 0; j := 0;                    i' := 0; j' := 0
while (j ≤ i ∨ j ≠ i) {            while (j' ≤ i' ∨ j' ≠ i') {
  if (i > j) j := j + 1;             i' := i' + 1;
  else i := i + 1;                   j' := j' + 1;
[q :] }                            [q' :] }
```

We define an assertion function I such that $I(entry(P_1), entry(P_2)) = \top$, $I(q, q') = (\mathsf{i} = \mathsf{i}' \wedge \mathsf{j} = \mathsf{j}' \wedge \mathsf{i} = \mathsf{j})$, and $I(exit(P_1), exit(P_2)) = \bot$. The function I is weakly extendible with the following reasoning: (1) For every run of (P_1, P_2) from any entry configuration, by taking two iteration of the loop in P_1 and one iteration of the loop in P_2, the run reaches (q, q') such that the last configuration satisfies $I(q, q')$; (2) From every configuration that satisfies $I(q, q')$, by taking the same path as before, the run reaches (q, q') again with a configuration that satisfies $I(q, q')$; (3) no exit configuration can be reached by any run of (P_1, P_2). □

In Voronkov and Narasamdya [2008] we show that, without appealing to the standard proof technique that uses well-founded set, and using only inter-program assertions and the notion of weak extendibility, we can prove program equivalence and mutual simulations of two programs where one program has a loop that does not correspond to any loop in the other program, or even the loop is eliminated in the other program. For proving RCRs, we often encounter such a situation. For example, PIN is a scalar variable in the SPM, but is an array variable in the FSP. So, for checking and updating the PIN, the FSP contains loops that do not exist in the SPM.

We now develop verification conditions that guarantee weak extendibility. To this end, we need a notion of path of pairs of programs. A path π of (P_1, P_2) can be viewed as a *trajectory* in a two dimensional space: $\pi = (\pi_1, \pi_2)$, where π_1 is a path in the flow graph of P_1 and π_2 is a path in the flow graph of P_2. A path is *trivial* if it consists of a single pair of points. Given a *path* π and an assertion ψ, we denote by $wp_\pi(\psi)$ and $wlp_\pi(\psi)$, respectively, the weakest and the weakest liberal preconditions of π and ψ. Since we have to compute these preconditions, we assume that the programming language that we consider has the *weak precondition property*: for every path π and every assertion ψ, $wp_\pi(\psi)$ exists and can effectively be computed. One can also compute $wlp_\pi(\psi)$ since it is equivalent to $wp_\pi(\psi) \vee \neg wp_\pi(\top)$. The precondition for paths of pairs of programs can also be derived from the precondition of paths of single programs.

DEFINITION 3.3 Let I be an assertion function and Π be a set of nontrivial path such that, for every $\pi \in \Pi$, we have $start(\pi)$ and $end(\pi)$ to be I-observable. Denote by $\Pi|_(p, p')$ the set of paths in Π whose first pair of points is (p, p').

The *weak verification condition* $\mathbb{W}$ associated with I and Π consists of assertions of the form

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

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

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

where (p, p') is I-observable. □

The first kind of assertion is a standard assertion for proving partial correctness of path. The second kind of assertion expresses that, whenever a configuration at p satisfies $I(p)$, the computation from this configuration will *inevitably* follows at least one path in Π.

THEOREM 3.4 *Let $\mathbb{W}$, I and Π be as in Definition 3.3. If every assertion in $\mathbb{W}$ is valid, then I is weakly extendible.* □

The notion of weak verification condition is our notion for certificates that certify inter-program properties. In the next section we will use inter-program assertions to describe correspondences between observable variables. Later, to prove an RCR between two commands, one has to prove other inter-program properties between transition graphs of the commands. These program properties altogether describe the RCR. To prove such properties, we define an assertion function and prove that the function is weakly extendible. The certificates certifying these properties form a certificate for the RCR.

4 Proving Representation Correspondences

For our discussion on proving RCRs, we consider the application models S and T described in Section 2. To prove an RCR between S and T, we are only concerned with command procedures, that is, for each corresponding command procedures, we prove an RCR between the procedures.

For two models S and T, there is usually a one-to-one correspondence Obs between global observable variables of S and T such that the values of each corresponding variables coincide at the entry and normal exit configurations of every command run. To this end, let us consider some command procedure P. Let Obs_p, Obs_f, Obs_a be one-to-one correspondences specified for the end configurations of transitions emitting, respectively, a a Pass_P, a Fail_P, an Abrupt_P event. For simplicity of presentation, in the sequel let $Obs_p = Obs_f$. Let Obs_c be a one-to-one correspondence specified for the start configurations of transitions emitting Call_P. We require that Obs is included in Obs_p and Obs_c. We say that a correspondence f is included in a correspondence g if for every mapping $x \mapsto y$ in f is a mapping in g.

Denote by P^S and P^T, respectively, the command P in S and in T. Given a function f, we denote by $dom(f)$ the domain of f. For simplicity of notation, given a one-to-one correspondence g, we abbreviate the assertion $\bigwedge_{x \in dom(g)} x = g(x)$ to simply g. To prove an RCR between P^S and P^T, we do the following steps:

1. Let α be an assertion, such that the assertion $\alpha \Rightarrow Obs_c$ is valid. That is, α describes the correspondence Obs_c. The assertion α can also describe invariants specific to S or T. We prove that α is satisfied by the initializations of global variables.

2. We assert α at $(entry(\mathbf{G}^n_{P^S}), entry(\mathbf{G}^n_{P^T}))$ and α' at $(exit(\mathbf{G}^n_{P^S}), exit(\mathbf{G}^n_{P^T}))$ such that the assertions $\alpha' \Rightarrow Obs_p$ and $\alpha' \Rightarrow \alpha$ are valid. That is, we assume that the correspondence expressed by α holds in the entry configurations of the procedures, and is preserved in the exit configurations.

3. Let ψ, ψ' be assertions asserted at, respectively, the pairs of points $(entry(\mathbf{G}^a_{P^S}), entry(\mathbf{G}^a_{P^T}))$ and $(exit(\mathbf{G}^a_{P^S}), exit(\mathbf{G}^a_{P^T}))$ such that the assertion $\psi' \Rightarrow Obs_a$ is valid. That is, the correspondence Obs_a holds when procedure runs terminate abruptly.

4. We prove that for every finite run of $\mathbf{G}^n_{P^T}$, there is a finite run of $\mathbf{G}^n_{P^S}$ from configurations satisfying α, and vice versa, such that the final configurations of the runs satisfy the assertion ψ.

One can demonstrate (1) easily since it amounts to proving that the initializations of global variables satisfy α. In the sequel we focus on the steps (2), (3), and (4).

We present our proof technique for proving RCRs of commands by means of a *real* example of a command called checkPIN that is used for authenticating users. In this report we only consider proving RCRs between the SPM and the FSP of the command. Proving RCRs between the FSP and the TDS follows the same steps above. The SPM is written in a domain-specific language, called command description language, that resembles a subset of Java. Each command can be thought of as a method that has clauses: one **pass** clause describing conditions and state updates of successful completion of a run of the command; one or more **fail** clauses describing logic failures and the corresponding state updates; and one **abrupt** clause describing abrupt behavior of the command. For each command procedure P, the **pass** and **fail** clauses of the command constitute the transition graph $\mathbf{G}^n_P$, while the **abrupt** clause constitutes the transition graph $\mathbf{G}^a_P$.

The FSP is written in a subset of Java. Each command procedure P is a method of the form:

```
P(...) { try { ... } catch(CardTearException) { ... } }
```

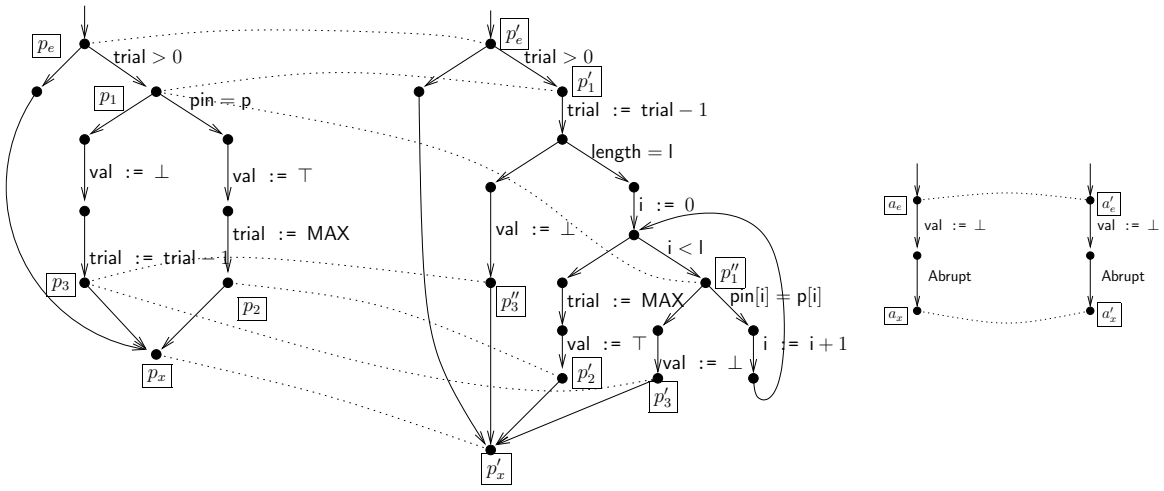

Figure 1: SPM and FSP of checkPIN.

The **try** part constitutes $\mathbf{G}^n_P$, while the **catch** part constitutes $\mathbf{G}^a_P$. Details of SPM and FSP can be found in another technical report Narasamdya and Périn [2008].

EXAMPLE 4.1 We prove that there is an RCR between two corresponding commands procedure P_c by considering their transition graphs $\mathbf{G}^n_{P_c}$ and $\mathbf{G}^a_{P_c}$ separately. The lefthand pair of transition graphs in Figure 1 is $\mathbf{G}^n_{\mathsf{checkPIN}}$ of the SPM, on the left of the pair, and $\mathbf{G}^n_{\mathsf{checkPIN}}$ of the FSP, on the right of the pair. As a shorthand, we call the former P_1 and the latter P'_1. The FSP ensures that the variable trial is decremented prior checking the PIN value. For disjointness, we assume that all variables in P_1 are primed.

First the global variables of the SPM that we want to observe are trial, pin, val, and MAX. They correspond to their primed counterparts in the FSP. Additionally, at the entries of P_1 and P'_1, the input pin p corresponds to p′, and at the exits of P_1 and P'_1, the event variable ε corresponds to ε'. Next, we have to define the equality between scalar PIN and array PIN. Every array PIN p is associated with a length l; we write this association as (p, l). We introduce predicate $\equiv$ between such pairs such that, given array PINs (p, l) and (p', l'), we say that $(p, l) \equiv (p', l')$ if $l = l'$, $l \geq 0$, and for all $i = 0, \ldots, l-1$, we have $p[i] = p'[i]$. We introduce a predicate $\sim$ which is axiomatized as follows: for every scalar PINs w, x and for every array PINs y, z,

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

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

The following assertions express the correspondence between observable variables:

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

Next, we define an assertion function I_1 of (P_1, P'_1) as follows:

$$\begin{array}{lcl} I_1(p_e, p'_e) & = & \bigwedge_{i=1}^{5} \phi_i \qquad I_1(p_x, p'_x) = \bigwedge_{i=1}^{6} \phi_i \\ I_1(p_1, p'_1) & = & \bigwedge_{i=1}^{5} \phi_i \wedge \mathsf{trial} > 0 \\ I_1(p_1, p''_1) & = & \bigwedge_{i=2}^{5} \phi_i \wedge \mathsf{trial} > 0 \wedge \mathsf{trial} = \mathsf{trial}' + 1 \\ & & \wedge \mathsf{length}' = \mathsf{l}' \wedge \mathsf{i}' < \mathsf{l}' \wedge (\forall j. 0 \leq j < \mathsf{i}' \Rightarrow \mathsf{pin}'[j] = \mathsf{p}'[j]) \\ I_1(p_2, p'_2) & = & \bigwedge_{i=1}^{5} \phi_i \wedge \mathsf{pin} = \mathsf{p} \wedge (\mathsf{pin}, \mathsf{length}) \equiv (\mathsf{p}, \mathsf{l}) \\ I_1(p_3, p'_3) & = & I_1(p_3, p''_3) = \bigwedge_{i=1}^{5} \phi_i \wedge \mathsf{pin} \neq \mathsf{p} \wedge (\mathsf{pin}, \mathsf{length}) \not\equiv (\mathsf{p}, \mathsf{l}) \end{array}$$

In this example, we prove an interesting part of RCR, that is, without any presence of card tears, for every run R of P_1, there is a run R' of P'_1 from compatible states, such that R and R' are observationally

equivalent. We denote by $\pi_{p,p'}$ a path from p to p', and by π_p a trivial path consisting only of point p. We prove that I_1 is weakly extendible by the following reasoning. First, for every run of (P_1, P_1') from an entry configuration that satisfies $I_1(p_e, p_e')$, the run can reach (p_1, p_1') by following the path $(\pi_{p_e,p_1}, \pi_{p_e',p_1'})$ such that the end configuration satisfies $I_1(p_1, p_1')$. From this configuration, the run can be extended either by following the path $(\pi_{p_1,p_3}, \pi_{p_1',p_3''})$ or by following the path $(\pi_{p_1}, \pi_{p_1',p_1''})$ such that the end configuration satisfies I_1. From the configuration that satisfies $I(p_1, p_1'')$, the run can be extended either by following $(\pi_{p_1}, \pi_{p_1'',p_1''})$, or by following $(\pi_{p_1,p_2}, \pi_{p_1'',p_2'})$, or by following $(\pi_{p_1,p_3}, \pi_{p_1'',p_3'})$. Without any of these paths, I_1 would not be weakly extendible. Thus, we have shown that, using the notion of weak extendibility, these paths show that the loop in P_1' terminates.

Note that every possible transition of P_1 is described by the nontrivial paths that constitute the first elements of all pairs of paths above. Therefore, we have proved that for every run R of P_1, there is a run R' of P_1' from compatible states, such that R and R' are observationally equivalent. We can use the same reasoning for proving the other direction. Indeed, by taking the set of all the above pairs of paths, one can prove that all assertions of the weak verification condition associated with I_1 and the set are valid.

Consider now the righthand pair of transition graphs in Figure 1 is $\mathbf{G}^a_{\mathsf{checkPIN}}$ of the SPM, on the left of the pair, and $\mathbf{G}^a_{\mathsf{checkPIN}}$ of the FSP on the right of the pair. As a shorthand, we call the former P_2 and the latter P_2'. The SPM and FSP only have to guarantee that the validation status is set to false in case of power loss. That is, the only observable variables are val and its primed counterpart.

We define an assertion function I_2 of (P_2, P_2') such that we have $I_2(a_e, a_e') = \top$ and $I_2(a_x, a_x') = (\mathsf{val} = \mathsf{val})$. It is easy to that I_2 is weakly extendible, which means that if a card tear occurs and the configurations of the runs at (a_e, a_e') satisfies I_2, then both runs will emit the same event, which is $\mathsf{Abrupt}_{\mathsf{checkPIN}}$ and they both terminate in compatible states.

Finally we have to prove that for every finite run of P_1 with end state σ, there is a finite run of P_1' with end state σ', and vice versa, such that (σ, σ') satisfies $I_2(a_e, a_e')$. Since $I_2(a_e, a_e')$ is satisfied by every state, then we have finished our proof. □

Proving RCRs between an FSP and a TDS is challenging due to the features of the language of the TDS. A TDS is written in a subset of Java Card Sun [2008], which includes transient and persistent memory as well as transaction mechanism. When a card tear occurs, data stored in persistent memory will be kept in the memory, while those stored in transient memory will be lost. Variables whose values are stored in persistent memory are called *persistent variables*, while those whose values are stored in transient memory are called *transient variables*.

Transactions are managed by methods beginTransaction, commitTransaction, and abortTransaction with standard functionalities. The depth of a transaction is at most 1. When a transaction is in progress, the updates of persistent variables are conditional, in the sense that the updates will be materialized if commitTransaction is called. Regardless a transaction is in progress or not, the updates of transient variables are unconditional. To model card tears and transactions, we use the desugaring method in Hubbers and Poll [2004b]. Each command in the TDS is a Java method, and desugaring the command means translating the method into the same form as that of the FSP, that is, the method has a big **try-catch** construct. The **catch** construct sets all transient variables to their default values, and cancel the updates of persistent variables if the card tear occurs when a transaction is in progress.

EXAMPLE 4.2 In this example we only consider $\mathbf{G}^n_{\mathsf{checkPIN}}$ of the SPM and $\mathbf{G}^n_{\mathsf{checkPIN}}$ of the FSP, depicted on the lefthand and righthand of figures 2, respectively. As a shorthand, we call the former P and the latter P'.

For disjointness, we assume that all variables in the TDS are primed. The input variables and the variable val' are the only transient variables; others are persistent. The boolean variable inTrans and the variable tb come from the desugaring method. The value of inTrans is true if a transaction is in progress; otherwise false. The variable tb backs up the value of trial.

In the presence of transactions, we require that the updates of trial and val must be unconditional. For this reason, Java Card has so-called non-atomic methods for updating persistent variables. Discussion on these methods and their effects on transactions can be found in Hubbers and Poll [2004a]. Since the TDS is only a reference implementation, in this example, we use inTrans to model the non-atomic methods.

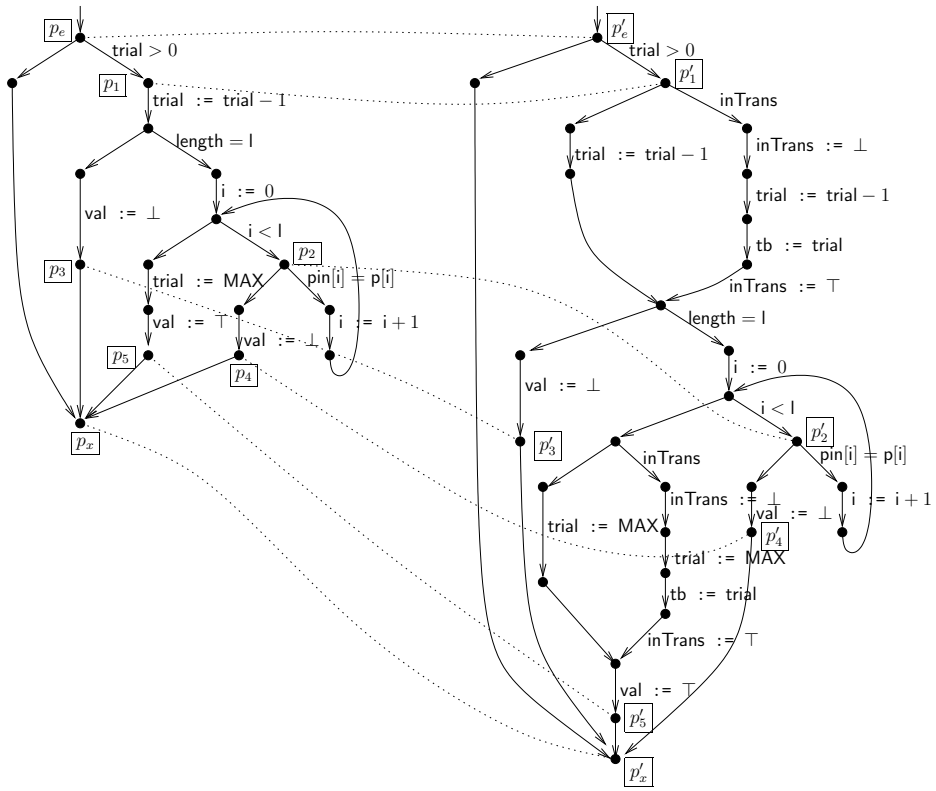

Figure 2: FSP and TDS of checkPIN.

The following assertions express the one-to-one correspondence between observable variables:

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

We then define an assertion function I of (P, P') such that

$$\begin{array}{l} I(p_e, p'_e) = I(p_1, p'_1) = I(p_2, p'_2) = \phi_1 \wedge \phi_2 \\ I(p_3, p'_3) = I(p_4, p'_4) = I(p_5, p'_5) = I(p_x, p'_x) = \phi_1 \end{array}$$

Given a set S of program points, we say that a path $\pi = p_0, \ldots, p_n$ 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. Let D be the set of pairs of program points, we denote by $fst(D)$ the set of first point of the pairs in D, and similarly by $snd(D)$ the set of second point. We denote by $Dom(I)$ the defined domain of the assertion function I.

By taking the set of pairs (π, π') of paths of (P, P'), such that π is $fst(Dom(I))$-simple and π' is $snd(Dom(I))$-simple, and I is defined on $(start(\pi), start(\pi'))$ and $(end(\pi), end(\pi'))$, one can easily prove that I is weakly extendible in a similar way to the previous example. These pairs of paths also show that, without any card tears, for every run R of P, there is a run R' of P' from compatible states, such that R and R' are observationally equivalent, and vice versa.

The remaining steps of proving the RCR follows those of the previous example. □

One might have to relax Definition 2.2 of RCRs to prove RCRs between an FSP and a TDS. Let us consider the following toy commands:

P_1:	P_2:	P'_1:	P'_2 :
x := 5;	ϵ	**if** (inTrans) **return** ERROR;	**if** (inTrans) {
y := 6;		xb := x′;	x′ := xb;
		yb := y′;	y′ := yb;
		inTrans := ⊤;	}
		x′ := 6;	
		y′ := 5;	
		inTrans := ⊥;	

The programs (or transition graphs) P_1 and P_2 constitute the **try** and **catch** parts of the FSP, respectively. (P_2 has no instruction.) Similarly for P'_1 and P'_2 of the desugared form of the TDS. Suppose that x′ and y′ are observable persistent variables that correspond to x and y, respectively. The variables xb and yb are back-up variables for x′ and y′. The boolean variable inTrans indicates whether a transaction is in progress or not; assume that it is false at the entry of P'_1. In case of abrupt terminations, we want to ensure that the above correspondence holds. To this end, we have to assert at the entries of P_2 and P'_2 the assertion ϕ below:

$$(\neg \mathsf{inTrans} \Rightarrow \mathsf{x} = \mathsf{x}' \wedge \mathsf{y} = \mathsf{y}') \wedge (\mathsf{inTrans} \Rightarrow \mathsf{x} = \mathsf{xb} \wedge \mathsf{y} = \mathsf{yb})$$

For every finite run R' of P'_1 from a state satisfying $\mathsf{inTrans} = \bot$, there is a finite run R of P_1 such that the final configurations of the runs satisfy ϕ. For example, if R' reaches the middle of transaction, e.g., the entry of y′ := 6, then R simply stays at the entry of P_1. However, showing the other way around is not possible. When a run R reaches the entry of y := 6, then there is no finite run R' of P'_1 such that the final configurations satisfy ϕ. Thus, according to Definition 2.2 there is no RCR between the commands.

To handle such an above case, one can relax Definition 2.2. That is, we only require that for every run R of P^T from a configuration γ, there is a run R' of P^S from a configuration γ', where γ and γ' are compatible, such that R and R' are observationally equivalent. The drawback of this relaxed definition is that if P^T does not terminate and the assertion at the entries of abrupt graphs is valid, then there is always an RCR between P^T and P^S. Nevertheless, with this relaxed definition, we can still preserve security properties for S in T, as shown in the following section.

5 Property Preservation

In this section we show how security properties of the SPM can be preserved in the FSP using RCRs. Property preservation between the FSP and the TDS can be explained in the same way. We are only

concerned with security properties that can be characterized as partial correctness properties: a procedure P is *partially correct* with respect to a precondition α and a postcondition β, denoted by $\{\alpha\}P\{\beta\}$, if for every run of P from a state satisfying α and reaching an exit configuration, this configuration satisfies β.

Consider again the application models S and T and the one-to-one correspondences Obs_p, Obs_f, Obs_a, Obs_c described at the beginning of Section 4. We show property preservation by the following theorem:

THEOREM 5.1 *Let α and β be, respectively, a precondition and a postcondition for a procedure P^S such that $\{\alpha\}P^S\{\beta\}$. Let α' and β' be, respectively, a precondition and a postcondition for a procedure P^T such that the assertions*

$$Obs_c \Rightarrow (\alpha \Leftrightarrow \alpha')$$
$$(Obs_p \wedge \varepsilon = \mathsf{Pass}_P) \vee (Obs_f \wedge \varepsilon = \mathsf{Fail}_P) \vee (Obs_a \wedge \varepsilon = \mathsf{Abrupt}_P) \Rightarrow (\beta \Leftrightarrow \beta')$$

are valid. If there is an RCR between P^S and P^T, then $\{\alpha'\}P^T\{\beta'\}$. □

As an example, consider again the command and assertions in Example 4.1. Suppose that the property that we want to preserve is as follows: for any run of checkPIN, the value of variable val at the exit configuration of the run is true *if and only if* the run emits a $\mathsf{Pass}_{\mathsf{checkPIN}}$ event.

Let ψ be the assertion $(\mathsf{val} = \top \Leftrightarrow \varepsilon = \mathsf{Pass}_{\mathsf{checkPIN}})$ and φ be the conjunction of the following assertions: (1) $\mathsf{MAX} > 0$, (2) $0 \leq \mathsf{trial} \leq \mathsf{MAX}$, and (3) $\mathsf{trial} < \mathsf{MAX} \Rightarrow \mathsf{val} = \bot$. The above property can be expressed as a partial correctness property $\{\varphi\}\mathsf{checkPIN}\{\psi\}$. One can use standard Floyd-Hoard proof technique Floyd [1967], Hoare [1969] to prove the property for both the SPM and the FSP.

Suppose that we have proved that the property holds for the SPM. We have shown in Example 4.1 that there is an RCR between the command checkPIN of the SPM and of the FSP. Let P be the command checkPIN in the FSP. Recall again the assertions $\phi_1, \ldots, \phi_6$ in the example. Given an assertion α, let us denote by $p(\alpha)$ the assertion obtained from α by replacing each variable in α with its primed notation. Now, since the following assertions

$$\textstyle\bigwedge_{i=1}^{5} \phi_i \Rightarrow (\varphi \Leftrightarrow p(\varphi))$$
$$(\textstyle\bigwedge_{i=1}^{6} \phi_i \wedge (\varepsilon = \mathsf{Pass}_P \vee \varepsilon = \mathsf{Fail}_P)) \vee (\phi_2 \wedge \varepsilon = \mathsf{Abrupt}_P) \Rightarrow (\psi \Leftrightarrow p(\psi))$$

are valid, then by Theorem 5.1 we have $\{p(\varphi)\}P\{p(\psi)\}$.

6 Related Work and Conclusion

We developed a method for proving RCRs in the CC EAL7 certification of smart-card applications. We presented a modelling framework by which the representations of the SPM, the FSP, and the TDS can be modelled uniformly. Our framework is an extension of the modelling framework of procedural programs in Zaks and Pnueli [2008], in the sense that we model abrupt behavior of procedures. Our definition of RCRs is mutual simulations between two application models. We apply the theory of inter-program properties in Voronkov and Narasamdya [2008] for proving RCRs and providing certificates about them. The theory has been used for proving properties in translation validation approach to compiler verification Rinard and Marinov [1999], Zuck et al. [2003], Voronkov and Narasamdya [2008]. In this report we have shown another venue for the application of the theory. The application is beneficial since the theory provides a notion of certificate, which is essential in the CC EAL7 certification.

There have been a few works on formal specification and verification in the CC framework; closely related to ours is Dadeau et al. [2008]. Their work is based on B method. Their definition of RCRs is similar to ours, in the sense that, for each command, they have a mapping between input-output relationships of two application models. Their work does not address complex data abstractions like our PIN, and their commands do not contain loops. However, their work has gone beyond ours in the sense that they included a model of Java Card API for APDU commands Sun [2008].

Another related work is by Heitmeyer et. al. on verifying enforcement of data separation in the kernel of a software-based embedded device Heitmeyer et al. [2006]. Similar to ours, their work uses a state machine model consisting of events as a specification. Concrete code is partition into event code, trusted

code, and other code. Event code corresponds to an event in the state-machine specification and such code is annotated with preconditions and postconditions. Their work construct two mappings: one is between events of the state machine and of the code, and the other is between assertions describing preconditions and postconditions of corresponding events. RCRs are proved for each corresponding events, that is, the precondition and the postcondition of an event in the code imply, respectively, the precondition and postcondition of the corresponding event in the specification. In their work, event code contains no loops, and they do not prove the relation between the code and its precondition and postcondition. Moreover, the mapping between assertions is based only on syntactic matching. Unlike ours, their work deals with real C code.

Other works on the CC certification have not addressed RCRs, or have only given little efforts on RCRs Chetali and Nguyen [2008], Wilding et al. [2001]. One distinguish feature in our work that has not been addressed by others is proving property preservation using RCRs.

There has been some work related to the specification and verification of smart-card applications, but not in the CC certification. Paper Schellhorn et al. [2006] describes a verification of Mondex electronic purse based on abstract state machine (ASM). The work is not in the CC, but it uses a notion of refinement simulation between ASMs to show correctness of a concrete implementation. The operations (similar to commands) in Mondex are simple and contains no loops and no complex data abstractions. The work in Breunesse et al. [2005] describes a case study in the specification and verification of an electronic purse application. The work is concerned only with the specification and verification of commands in the implementation code. The work can complement our work in proving properties of the implementation code.

In this report we do not address RCRs between the TDS and the implementation code. We assume that existing work on certified and certifying compilers Leroy [2006], Rinard and Marinov [1999] can be used to provide RCRs between the TDS and the implementation code. We are currently developing certification tools based on the method described in this report. We take JML approach Leavens and Cheon [2003] to specifying assertion function. That is, we use special comments to put labels denoting program points in the programs, and write the assertion function in a separate file. We use off-the-shelf data-flow analyses, such as global value numbering, to assist users in defining assertion functions, so that users only concentrate on one-to-one correspondences between observable variables. We are developing heuristics based on observable events to alleviate the burdens of specifying paths in weak verification conditions; this is the topic of our future work. Assertions in the verification conditions can then be proved using SMT solvers, such as Yices Dutertre and de Moura [2006].

References

Common Criteria for Information Technology Security Evaluation, 2007. Version 3.1, CCMB-2007-09-003. 1

C.-B. Breunesse, N. Cataño, M. Huisman, and B. Jacobs. Formal methods for smart cards: an experience report. *Sci. Comput. Program.*, 55(1-3):53–80, 2005. 6

Boutheina Chetali and Quang-Huy Nguyen. Industrial use of formal methods for a high-level security evaluation. In *Formal Methods*, pages 198–213, 2008. 6

Frédéric Dadeau, Marie-Laure Potet, and Régis Tissot. A B formal framework for security developments in the domain of smart card applications. In *Security Conference*, pages 141–155, 2008. 6

Bruno Dutertre and Leonardo Mendonça de Moura. A fast linear-arithmetic solver for DPLL(T). In *Computer Aided Verification*, pages 81–94, 2006. 6

Robert W. Floyd. Assigning meaning to programs. In J. T. Schwartz, editor, *Proceedings of Symposium in Applied Mathematics*, pages 19–32, 1967. 5

Constance L. Heitmeyer, Myla Archer, Elizabeth I. Leonard, and John McLean. Formal specification and verification of data separation in a separation kernel for an embedded system. In *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, pages 346–355, New York, NY, USA, 2006. ACM. 6

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

E.-M.G.M. Hubbers and E. Poll. Transactions and non-atomic API methods in Java Card: specification ambiguity and strange implementation behaviors. Technical Report NIII R0438, University of Nijmegen, Toernooiveld, 6525 ED Nijmegen, The Netherlands, October 2004a. 4.2

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, 2004b. ISBN 3-540-21305-8. 4

G. Leavens and Y. Cheon. Design by contract with JML, 2003. 6

Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. *SIGPLAN Not.*, 41(1):42–54, 2006. ISSN 0362-1340. 6

Iman Narasamdya and Michaël Périn. Certification of smart-card applications in common criteria. Technical Report TR-2008-14, Verimag, September 2008. 4

M. Rinard and D. Marinov. Credible compilation with pointers. In *Proceedings of the FLoC Workshop on Run-Time Result Verification*, Trento, Italy, July 1999. 6

Gerhard Schellhorn, Holger Grandy, Dominik Haneberg, and Wolfgang Reif. The mondex challenge: Machine checked proofs for an electronic purse. In *FM*, pages 16–31, 2006. 6

Java Card 3.0 Platform Specification. Sun Micro systems, Inc, Palo Alto, California, 2008. `http://java.sun.com/javacard/3.0/`. 4, 6

A. Voronkov and I. Narasamdya. Proving inter-program properties. Technical Report TR-2008-13, Verimag, 2008. 1, 3, 3, 6

M. Wilding, D. A. Greve, and D. Hardin. Efficient simulation of formal processor models. *Formal Methods in System Design*, 18(3):233–248, 2001. 6

Anna Zaks and Amir Pnueli. CoVaC: Compiler validation by program analysis of the cross-product. In *FM*, pages 35–51, 2008. 6

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. 6