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

Iman Narasamdya, Michaël Périn

Verimag Research Report n^o TR-2008-14

September 2008

Certification of Smart-Card Applications in Common Criteria

Iman Narasamdya, Michaël Périn

September 2008

Abstract

This report describes a certification method of smart-card applications in the framework of Common Criteria. In this framework, a smart-card application is represented consecutively by a model of its specification, a functional specification describing an input-output relationship, a low-level design, and implementation code. The certification process consists of the following tasks: (1) prove that the model, the functional specification, the low-level design, and the code satisfy security properties in the smart-card application's specification, and (2) prove that there is a representation correspondence between each two consecutive representations. For each task, a certificate or a collection of certificates are needed to certify the accomplishment of the task. We describe in this report the application of a theory of program properties to the certification process. The theory provides foundations for describing and proving properties of a single program and properties relating two programs. The theory provides a notion of verification condition as a notion of certificate. The theory is applicable to the certification process because all representations of a smart-card application are essentially programs and the representation correspondences are properties relating two programs.

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},
authors = { Iman Narasamdya, Michaël Périn},
institution = { Verimag Research Report },
number = {TR-2008-14},
year = { },
note = { }
}

1 Introduction

The use of smart cards has been pervasive in our everyday lives. For example, smart cards in the form of debit or credit cards have been used in electronic banking transactions. Smart cards have also been used for mobile telephony. With the widespread use of mobile phones, the use of smart cards will play an important role in our lives. Smart-card applications are programs embedded in the chip on smart cards. These programs control the use of smart cards. Smart card and smart-card applications have mostly been used to provide security, mainly for authentication and authorization. The security functions provided by a smart-card application are described in the specification as security properties. Since security properties are paramount for a smart-card application, one has to prove formally that an implementation of the application satisfies the security properties. Moreover, to give high confidence to the user of the smart-card application, one needs to provide a certificate showing that the implementation indeed satisfies the properties.

We describe in this report our work on certifying smart-card applications. Our work is part of an industrial project called EDEN2 that has been conducted at Verimag laboratory. The aim of the project are twofold: (1) to develop a method for formal software certification in the framework of Common Criteria certification [Com, 2007], 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), *TOE design* (TDS), and *implementation* (IMP). Given a system and its specification, an SPM is a model of the specification. An FSP describes an input-output relationship of the system. TOE stands for target of evaluation, which is the system itself. A TDS is a low-level design of the system. We often describe a TDS as a reference implementation. An IMP is the code implementing the system. Each requirement in CC has a representation. For example, in EDEN2 the SPM is written in a declarative language specifying the behavior of the smart-card application, while the FSP and the TDS are written in subsets of Java. Between every two consecutive requirements there is a so-called representation correspondence (RCR) relating the two requirement representations.

In the CC certification process one first has to demonstrate that each requirement representation satisfies the security properties, and also produce certificates that certify that the representation satisfies the properties. Second, one proves that there is an RCR between each two consecutive representations and produces a certificate about the RCR. In this report we consider only the requirements SPM, FSP, and TDS.

We apply the theory of program properties described in [Narasamdya, 2007, Voronkov and Narasamdya, 2008] to the CC certification process. The theory provides foundations for proving properties of a single program and properties that relate two programs. The formalization of the theory is based on a suitably adapted notion of program invariant for a single program. The theory is based on the notion of *assertion function*: a function that assigns assertions to program points. The theory introduces the notion of extendible assertion function as a constructive notion for describing and for proving program invariants. This notion is developed further in the theory so that it can be used to prove properties relating two programs, or inter-program properties. The theory also develops a notion of verification condition. A verification condition associated with an assertion function of a program forms a *certificate* that certifies that the program satisfies the properties described by the assertion function. A verification condition itself is a finite set of assertions constructed from the assertion function and the program. A certificate can be turned into a *proof* by proving that all assertions in the certificate are valid.

The representations of the SPM, the FSP, and the TDS are essentially programs. Although standard Floyd-style verification technique like [Floyd, 1967, Hoare, 1969] can be applied to proving their properties, the theory described above can also be used to prove the properties and, additionally, to provide certificates about those properties. The RCR between two consecutive requirements are essentially properties relating two programs. Thus, we can apply the theory to prove the RCR and to provide a certificate about the RCR.

In this reportwe discuss the application of the theory to proving properties of SPMs. Properties of FSPs and TDSs can be proved in the same way as proving properties of SPMs. In the CC certification process, one has to demonstrate that if the SPM satisfies some property, then the FSP and the TDS also satisfy the same property. Instead of proving the same property for the FSP and the TDS, we describe in this report

how properties of the SPM are preserved by the RCR between the SPM and the FSP. That is, once one proves that the SPM satisfies some property and there is an RCR between the SPM and the FSP, then the FSP satisfies the same property. Property preservation from the FSP to the TDS can be described similarly to describing property preservation from the SPM to the FSP.

The contribution of this paper is the application of the above theory to the certification of smart-card applications in CC. The application itself is not straightforward since smart-card programs have different characteristics from typical imperative programs. First, a run of a smart-card program can terminate abruptly in the middle of the program due to power loss. Thus, one has to model such an abrupt termination. Second, the low-level design of the application includes transaction mechanism. One then has to model transaction mechanism so that the theory can be applied. The definitions of RCRs are more complex than standard refinement relations. For example, when a transaction is not in progress, the order of updating some variables of the TDS must be the same as the order of updating their corresponding counterparts in the FSP. But, when a transaction is in progress, such an order is irrelevant. Mapping between variables in RCRs can be nontrivial. For example, a scalar variable in the SPM corresponds to an array variable in the FSP.

The outline of this report is as follows. We first describe the theory of program properties. We only provide the essence of the theory. A detailed description of the theory can be found in [Narasamdya, 2007, Voronkov and Narasamdya, 2008]. We then apply the theory to proving properties of SPMs. Afterward, we apply the theory to proving RCRs between SPMs and FSPs, and then RCRs between FSPs and TDSs. Having described the application of the theory to proving RCRs, we discuss property preservation from SPMs to FSPs by RCRs between SPMs and FSPs. Finally, we briefly discuss some related works and then conclude this report.

2 A Theory of Program Properties

2.1 Assumptions

The theory is based on standard assumptions about programs and their semantics. 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*. 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 $p = exit(P)$.

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 $\mathbf{G}_P$.

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 $p_0 = entry(P)$.

□

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 $\gamma_1 \mapsto_P \gamma_2$. 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. 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 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 $\sigma \not\models \alpha$.

2.2 Extendible Assertion Functions

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

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

mapping program points of P to assertions such that $I(entry(P))$ and $I(exit(P))$ are defined. 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 $I(p)$.

For proving that a program satisfies some properties, we introduce the notion of extendible assertion function. This notion provides a constructive characterization of relations between an assertion function and a program.

DEFINITION 2.2 Let I be an assertion function of a program P. I is *strongly extendible* if for every run

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

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

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

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. □

Later, to provide verification conditions associated with assertion functions, we need a notion of covering set. We say that a set C of program points in P *covers* P if $entry(P) \in C$ and every infinite path in $\mathbf{G}_P$ contains a program point in C. Verification conditions associated with assertion functions consist of assertions formed from paths in program-point flow graphs. To form such assertions, we need the notions of precondition and liberal precondition.

DEFINITION 2.3 (Weakest Precondition) Let $\pi = (p_0, \ldots, p_n)$ be a path 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. □

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. In the sequel we assume that our programming language has the *weakest precondition property*, that is, 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. Since $wlp_\pi(\psi)$ is equivalent to $wp_\pi(\psi) \vee \neg wp_\pi(\top)$, 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 2.4 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 2.5 *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.* □

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 2.6 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. □

THEOREM 2.7 *Let I and Π be as in Definition 2.6 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.* □

2.3 Inter-Program Properties

To prove properties relating two programs P and P', we consider the programs as a pair (P, P') of programs with 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. 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 $p = entry(P)$ and $p' = entry(P')$, and an *exit configuration for* (P, P') if $p = exit(P)$ and $p' = exit(P')$.

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.

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

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

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 $I(p, p')$.

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.

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 2.8 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. $\gamma_{i+n} \models I$.

□

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.

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

DEFINITION 2.9 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 2.10 *Let I and Π be as in Definition 2.9 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.* □

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

3 Proving Properties of Policy Models

3.1 Smart-Card Application Life Cycle

In this section we briefly overview the operations of smart-card application. A *card reader* (or a *terminal*) communicates with a smart-card application by first selecting the application and then sending a sequence of commands to the application. Each smart-card application is identified by its *application identifier* (AID). Commands sent by the reader are in the form of *application protocol data units* (APDUs), a standard

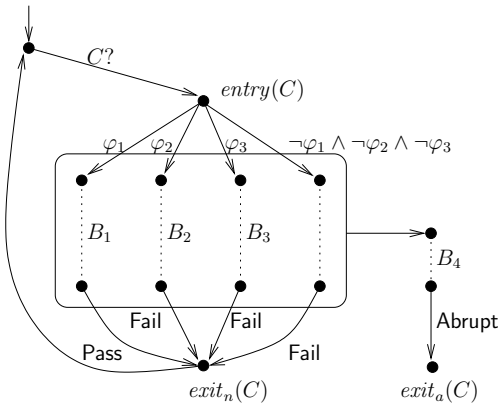

Figure 1: Semantics of SPM.

format for exchanging data defined in ISO 7816-4. The application replies to each APDU command with a status word indicating the result of the operation, and optionally with data. The reader terminates the communication with the application by deselecting the application.[1]

An application is *inactive* when it is first installed into the smart card. The application then becomes *active* when it gets selected by a card reader. From being active, the application becomes inactive if the reader deselects the application or a card tear (loss of power) occurs. Later in defining the runtime behavior of smart-card applications, we only concern with the behavior of active applications.

3.2 Command Description Language

We now discuss the language used to write SPMs in EDEN2. An SPM models the policy that a smart-card application has to respect. Such a policy includes security and safety properties. The SPM might not be able to model all security properties, but at least it models the *access control* and *information flow* of the application. The SPM models such properties by describing the behavior of each command, and therefore the language used to write SPMs is called *command description language*.

We only provide an informal description of the syntax and semantics of the language. An SPM consists of commands that will be implemented in the smart-card application. Each command in the SPM has the following form:

command $C(p_1, \ldots, p_n)$ {
  pass (φ_1) { B_1 }
  fail (φ_2) { B_2 }
  fail (φ_3) { B_3 }
  abrupt { B_4 }
}

The command C has a list $(p_1, \ldots, p_n)$ of input parameters. Each input parameter specifies the type and the name of the parameter. The list of input parameter can be empty. The behavior of a command is specified by the **pass** , **fail** , and **abrupt** clauses. The conditions $\varphi_1, \varphi_2, \varphi_3$ of the clauses are boolean expressions. The bodies B_1, B_2, B_3, B_4 of the clauses are statements written in a simple imperative language.

The semantics of the command C is described by the flow graph depicted in Figure 1. First, for every command C, there is a unique entry denoted by $entry(C)$, but there are two exit points, one exit point, denoted by $exit_n(C)$, is for normal exit and the other, denoted by $exit_a(C)$, is for abrupt exit. Second, each clause in the command description is associated with a special event. For **pass** clause, we associate a

[1] Specifically for Java Card platform, selecting and deselecting an application are performed by sending special commands to the Java-Card Runtime Environment, which in turn calls, respectively, selection and deselection methods implemented in the application.

statement emitting Pass event at the end of the body of the clause. Similarly for **fail** and **abrupt** clauses, we associate Fail and Abrupt events.

A run or an execution of a command C from a state σ is a computation sequence starting from the configuration $(entry(C), \sigma)$. If the state σ satisfies φ_1, φ_2, or φ_3, then the run will go through the bodies, respectively, B_1, B_2, or B_3. If the state σ satisfies none of φ_1, φ_2, or φ_3, then the run will go through the empty body. A run of a command C terminates normally if it reaches $exit_n(C)$, and for such a termination, the run emits either a Pass or Fail event. For simplicity, particularly to eliminate nondeterminism and blocking run, we interpret the **pass** and **fail** clauses of the command as the following **if** -**then** -**else** construct:

if φ_1 **then** B_1
else if φ_2 **then** B_2
else if φ_3 **then** B_3
else B_5

where B_5 is an empty body.

A card tear can occurs at any time and at any configuration of a command run. In Figure 1, for every point in the round box, we have an edge going to the entry of B_4. A run of a command C terminates abruptly if it reaches the point $exit_a(C)$, and in reaching this point the run emits an Abrupt event.

As also shown in Figure 1, an SPM itself can be considered as a program that takes as an input a sequence of commands. Each input command is of the form $C(a_1, \ldots, a_n)$, where C is the command's name and $a_1, \ldots, a_n$ are the input arguments. A run of an SPM can be considered as a sequence of runs of commands in the SPM. For each run of a command in the SPM, if the run terminates normally, then the run of the SPM fetches the next input $C(a_1, \ldots, a_n)$ in the input sequence. The notation $C?$ in Figure 1 means that the fetched input is a command C. When a card tear occurs, then the run of the command terminates abruptly and, in turn, the run of the SPM simply terminates. In the life cycle of a smart-card application, such an abrupt termination makes the application inactive.

A run of an SPM 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 $j \geq 1$, ε_j is an event associated with transition $\gamma_{j-1} \mapsto \gamma_j$.

Events are not restricted to Pass, Fail, and Abrupt events; we allow unobservable internal events. We assume that each SPM has an input variable and the state of configuration γ_0 maps this variable to the input value, which is a sequence of commands.

3.3 Proof Technique

We prove properties of an SPM by proving properties of each command in the SPM. Each command in the SPM is represented by two flow graphs: one for the **pass** and **fail** clauses, and the other for the **abrupt** clause. We illustrate our proof technique by the following examples.

EXAMPLE 3.1 We consider a command used to authenticate users by verifying the input PIN given by the users. We call the command checkPIN. The SPM of the command is depicted in Figure 2. For simplicity, we omit the types of variables in the SPM. We assume that pin, p, MAX, and trial are of integral type, denoted by `int`, while val is of boolean type. The variable pin holds the PIN stored in the card and the variable p holds the input PIN. The variable MAX holds the maximum number of failed trials, while the variable trial holds the remaining failed trials. The variable val is a flag denoting the validation status of the PIN.

In Figure 2 we also depict the flow graphs for the **pass** and **fail** clauses in the middle and for the **abrupt** clause on the right. Let us call the former graph P_1 and the latter P_2.

The property that we want to prove is as follows:

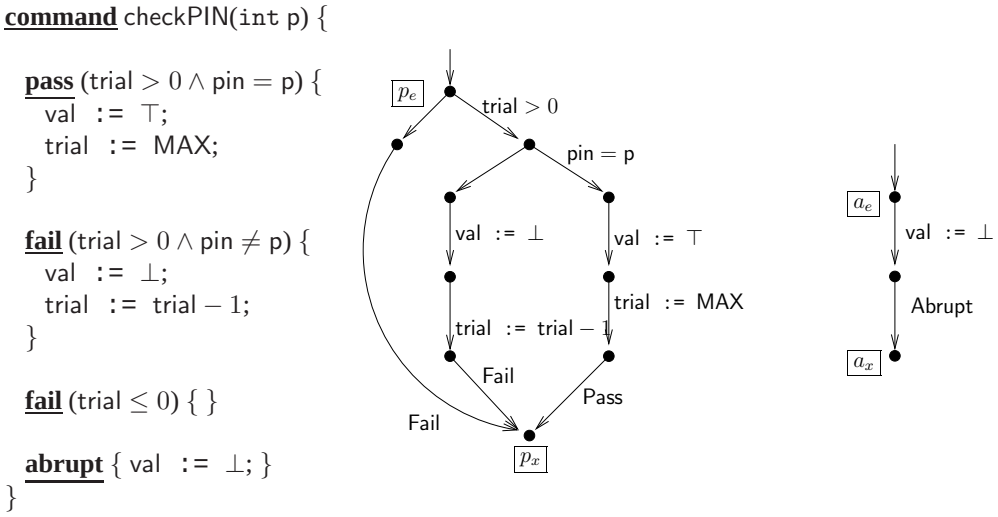

Figure 2: An SPM of checkPIN.

> In 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 Pass event.

This property describes an information flow in the command, and thus is a security property. To prove this property, we need to prove that, for any run of each command, the following sub-properties hold at the entry and normal exit configurations of the run:

1. The value of variable MAX is greater than 0.
2. The value of variable trial is between 0 and the value of MAX inclusive.
3. If the PIN is blocked, that is the value of trial is equal to 0, then the validation status, or the value of val is false.

These properties are often called *invariants of the SPM*. Note that properties (2) and (3) are safety properties.

Let the following assertions describe the above properties:

$$\begin{array}{ll} \mathsf{MAX} > 0 & (1) \\ 0 \leq \mathsf{trial} \leq \mathsf{MAX} & (2) \\ \mathsf{trial} = 0 \Rightarrow \mathsf{val} = \bot & (3). \end{array}$$

We generalize property (3) to

$$\mathsf{trial} < \mathsf{MAX} \Rightarrow \mathsf{val} = \bot \quad (3'),$$

that is, instead of proving $(1) \wedge (2) \wedge (3)$, we prove a stronger assertion $(1) \wedge (2) \wedge (3')$. Denote the latter assertion by φ.

First assume that emitting an observable event is performed by assigning the event to a special variable called ε. We assume further that $\mathsf{Pass} \neq \mathsf{Fail} \neq \mathsf{Abrupt}$. We now define assertion functions I_1 of P_1 and I_2 of P_2 as follows:

$$\begin{array}{lcl} I_1(p_e) & = & \varphi \\ I_1(p_x) & = & \varphi \wedge \mathsf{val} = \top \Leftrightarrow \varepsilon = \mathsf{Pass} \end{array}$$

$$\begin{array}{lcl} I_2(a_e) & = & \top \\ I_2(a_x) & = & \mathsf{val} = \top \Leftrightarrow \varepsilon = \mathsf{Pass} \end{array}$$

Second, since a card tear can happen at any time and at any point in the flow graph P_1. We need to prove that for any run of P_1 from a state satisfying $I_1(p_e)$, the assertion $I_2(a_e)$ holds at every configuration at any point in P_1. Since $I_2(a_e)$ is a valid assertion, then $I_2(a_e)$ holds at every configuration.

We argue that, for the command checkPIN, if the functions I_1 and I_2 are weakly extendible, then the properties that we want to prove hold. Consider a run R of checkPIN from a state σ satisfying $I_1(p_e)$. Assume first that card tears are not present. If σ satisfies the condition of the **pass** clause, then the run will reach the normal exit of checkPIN with some state σ'. Since σ' satisfies $\mathsf{trial} = \mathsf{MAX}$ and MAX are not modified in the run, then σ' satisfies $0 \leq \mathsf{trial} \leq \mathsf{MAX}$ and $\mathsf{MAX} > 0$. Because the value of MAX was assigned to trial in the run, it follows that σ' satisfies $\mathsf{trial} < \mathsf{MAX} \Rightarrow \mathsf{val} = \bot$. Finally, since val and ε are assigned with, respectively, $\top$ and Pass, the state σ' satisfies $\mathsf{val} = \top \Leftrightarrow \varepsilon = \mathsf{Pass}$. Thus, the state σ' satisfies $I_1(p_x)$.

With a similar kind of reasoning, if the state σ of the run R satisfies $I_1(p_e)$ and the condition $\mathsf{trial} > 0 \wedge \mathsf{pin} \neq \mathsf{p}$, then when card tears are not present, the run will reach the normal exit with a state σ'' that satisfies $I_1(p_x)$. The assertion $0 \leq \mathsf{trial} \leq \mathsf{MAX}$ holds at σ'' since the assertion

$$0 \leq \mathsf{trial} \leq \mathsf{MAX} \wedge \mathsf{MAX} > 0 \wedge \mathsf{trial} > 0 \Rightarrow 0 \leq (\mathsf{trial} - 1) \leq \mathsf{MAX}$$

is valid. Similarly for the assertion $\mathsf{trial} < \mathsf{MAX} \Rightarrow \mathsf{val} = \bot$. The assertion $\mathsf{val} = \top \Leftrightarrow \varepsilon = \mathsf{Pass}$ is trivially satisfied by σ'' because val and ev were assigned with $\bot$ and Fail, respectively.

Now, suppose that the state σ of the run R satisfies $I_1(p_e)$ and $\mathsf{trial} \leq 0$. When no card tears occur, the run reaches the normal exit with a state σ'''. Since there is no variable modified in the run, it follows that σ''' satisfies φ. To prove that σ''' satisfies $\mathsf{val} = \top \Leftrightarrow \varepsilon = \mathsf{Pass}$, we need to show that σ''' maps val to $\bot$. Since the state σ satisfies $\mathsf{trial} \leq 0$ and $\mathsf{MAX} > 0$, then σ satisfies $\mathsf{trial} < \mathsf{MAX}$. Because σ also satisfies $\mathsf{trial} < \mathsf{MAX} \Rightarrow \mathsf{val} = \bot$, it follows that σ satisfies $\mathsf{val} = \bot$. Finally, the state σ''' will map val to $\bot$ because val was not modified in the run. This reasoning has shown the importance of the properties described by the assertion $\varphi = I_1(p_e)$.

For the assertion function I_2, we define $\top$ on a_e because we do not make any assumption when a card tear occurs. Moreover, since val is set to $\bot$ and ε gets the value Abrupt, the assertion $\mathsf{val} = \top \Leftrightarrow ev = \mathsf{Pass}$ holds trivially at the exit configuration of a run that terminates abruptly.

We prove that I_1 and I_2 are strongly extendible. First, both I_1 and I_2 cover P_1 and P_2, respectively. Let $\pi^1_{p_e,p_x}$, $\pi^2_{p_e,p_x}$, and $\pi^3_{p_e,p_x}$ be I_1-simple paths from p_e to p_x. The path $\pi^1_{p_e,p_x}$ traverses the condition $\mathsf{pin} = \mathsf{p}$, the path $\pi^2_{p_e,p_x}$ traverses the condition $\mathsf{pin} \neq \mathsf{p}$, and the path $\pi^3_{p_e,p_x}$ traverses $\mathsf{trial} \leq 0$. Let π_{a_e,a_x} be the only path in P_2. The strong verification conditions for I_1 and I_2 consist of the following assertions:

$$\begin{aligned}
&I_1(p_e) \Rightarrow wlp_{\pi^1_{p_e,p_x}}(I(p_x))\\
&I_1(p_e) \Rightarrow wlp_{\pi^2_{p_e,p_x}}(I(p_x))\\
&I_1(p_e) \Rightarrow wlp_{\pi^3_{p_e,p_x}}(I(p_x))\\
&I_2(a_e) \Rightarrow wlp_{\pi_{a_e,a_x}}(I(a_x)).
\end{aligned}$$

One can prove that these assertions are all valid, and thus I_1 and I_2 are strongly extendible, which in turn are also weakly extendible.

To prove that the above properties hold for the whole SPM, we need to prove that the assertion φ holds at the entry and normal exit configurations of any run of *other* commands. To this end, we follow the following steps:

1. Prove that the assertion φ holds after the initialization of the SPM.
2. For each command, define an assertion function for the flow graph representing the **pass** and **fail** clauses such that the assertions defined on the entry and normal exit points of the function imply φ, and then prove that the function is weakly extendible.

These steps (1) and (2) can be carried out in the same way as proving the properties for the command checkPIN. □

In the above example we do not assume anything when a card tear occurs. That is, the assertion function I_2 is defined as $\top$ on the entry point a_e of the program P_2 that represents abrupt termination. For simplicity,

```
command updatePIN(int p) {

  pass (val) {
    if (c) {
      pin := p;
      trial := MAX;
    } else {
      pb := pin;
      tb := trial;
      c := ⊤;
      pin := p;
      trial := MAX;
      c := ⊥;
    }
  }

  fail (¬val) { }

  abrupt {
    if (c) {
      pin := pb;
      trial := pt;
    }
    val := ⊥;
  }
}
```

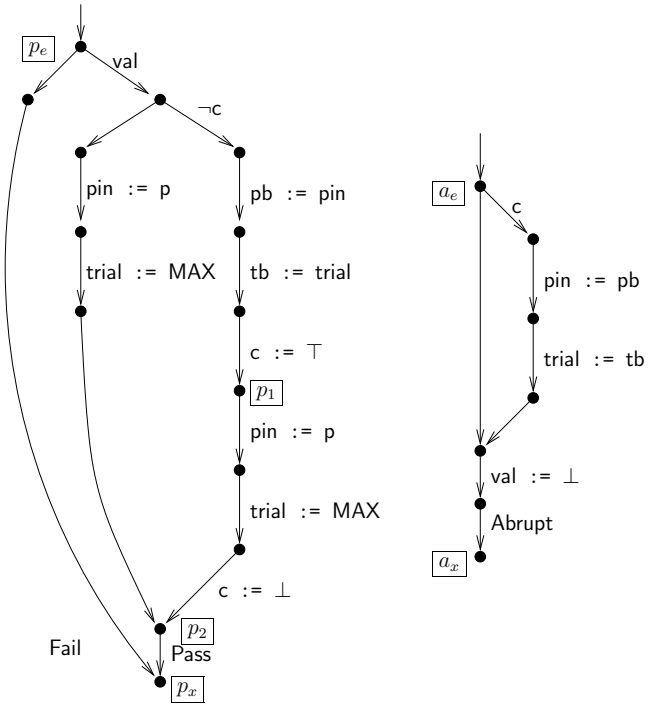

Figure 3: An SPM of updatePIN.

let us call such a P_2 an *abrupt program* or an *abrupt flow graph*. The assertion $\top$ is valid at every state, and so when a card tear occurs, the state upon the entry of P_2 satisfies $\top$. One can unnecessarily assert $\mathsf{MAX} > 0$ at the point a_e since, provided that $\mathsf{MAX} > 0$ holds at the entry point of the command and MAX is not modified in the command, the assertion always holds at every configuration of a run when the run is in the **pass** or **fail** clause. Thus, the entry configuration of any run of P_2 always satisfies $\mathsf{MAX} > 0$. However, one often needs to define a precondition at the entry point of an abrupt program such that some variables in the precondition are modified in the **pass** or **fail** clause of the command description. In such a case, one has to prove that the precondition always holds at every configuration of a run of the command when the run is in the **pass** or **fail** clause. We illustrate such a case in the following example.

EXAMPLE 3.2 We consider a command for updating PIN shown in Figure 3. The updates of pin and trial are conditional depending on the value of variable c. Such an update mechanism resembles the transaction facility in the implementation level. Since the command description language does not have any transaction facility, the conditional updates are performed manually with the help of the variable c. The variables pb and tb can be considered as variables that back up the values of pin and trial, respectively. The variables pb and tb are declared globally.

In Figure 3 we depict the flow graphs for normal termination in the middle and for the abrupt termination on the right. Denote the former flow graph by P_1 and the latter by P_2.

The property that we want to prove in this example is the following:

> In any run of updatePIN, at the exit configuration of the run, either the values of both pin and trial are updated or the values of these variables coincide with the values at the entry configuration.

To prove the property, we need to assert that at the entry point of any run of updatePIN, if the condition c is true, then the variables pb and tb hold the latest values of, respectively, pin and trial before c becomes true. Assume that the variable c is initially false. For simplicity, we assume that we have proved that, for every command besides updatePIN, we prove that whenever c is true, the variables pb and tb holds the latest values of, respectively, pin and tb before c becomes true.

We now define two assertion functions, I_1 of P_1 and I_2 of P_2 as follows:

$$\begin{array}{lcl} I_1(p_e) & = & (\mathsf{c} \Rightarrow (\mathsf{pb} = k \wedge \mathsf{tb} = l)) \wedge \mathsf{pin} = n \wedge \mathsf{trial} = m \\ I_1(p_x) & = & \psi \\ \\ I_2(a_e) & = & (\mathsf{c} \Rightarrow (\mathsf{pb} = n \wedge \mathsf{tb} = m) \vee (\mathsf{pb} = k \wedge \mathsf{tb} = l)) \\ & & \wedge(\neg \mathsf{c} \Rightarrow \psi) \\ I_2(a_x) & = & \psi \vee (\mathsf{pin} = k \wedge \mathsf{trial} = l), \end{array}$$

where k, l, m, n are logical variables, and ψ denotes the assertion

$$(\mathsf{pin} = \mathsf{p} \wedge \mathsf{trial} = \mathsf{MAX}) \vee (\mathsf{pin} = n \wedge \mathsf{trial} = m).$$

The function I_1 says that at the entry configuration of any run of updatePIN, the variables pin and trial hold some values denoted by the logical variables n and m, respectively. Moreover, if the condition c true, then the variables pb and tb also hold some values denoted by the logical variables k and l, respectively.

One can prove that I_1 and I_2 are weakly extendible. The weak extendibility of I_1 ensures that when the run reaches the exit of P_1, then either both pin and trial are updated or none of them are updated. Similarly, by being weakly extendible, the function I_2 guarantees that if the run terminates abruptly, then either the variables pin and trial are updated, or both variables retain the same values as the values upon entry, or the values of the variables must be rolled back to the latest values before the variable c becomes true.

Note that the assertion $I_2(a_e)$ "links" the flow graphs P_1 and P_2. That is, we have to prove that $I_2(a_e)$ holds at every state of any run of the command when the run is in the **pass** or **fail** clause, provided that the entry configuration of the run satisfies $I_1(p_e)$. First, since the assertion $I_1(p_e)$ implies $I_2(a_e)$, then it follows that for any run of the command such that the entry configuration satisfies $I_1(p_e)$, the configuration also satisfies $I_2(a_e)$. Next, for every statement that updates c, we prove that $I_2(a_e)$ holds immediately after the statement. We define another assertion function I_3 as follows:

$$\begin{array}{lcl} I_3(p_e) & = & I_1(p_e) \\ I_3(p_1) & = & I_2(a_e) \\ I_3(p_2) & = & I_2(a_e) \\ I_3(p_x) & = & I_2(a_e) \end{array}$$

We only consider the statements that update c because only those statements that can affect the truth value of $I_2(a_e)$. One can prove easily that the function I_3 is strongly extendible. Thus, for any run of the command such that the entry configuration of the run satisfies $I_1(p_e)$, the assertion $I_2(a_e)$ holds at every configuration of the run. □

In the implementation of the SPM, the conditions of the clauses in a command description are boolean expressions written in a subset of Java language. The bodies of the clauses are statements in a subset of Java language. Program points in the flow graphs are denoted by labels in the SPM. Labels are placed in a special comments following the JML notations [Leavens and Cheon, 2003]. Assertion functions are written in a separate file. Assertions themselves are JML expressions.

4 Proving RCRs between SPMs and FSPs

In EDEN2 an FSP is essentially a Java program written in a subset of Java. Each command in the FSP is a Java method. The return value of the method is a response status indicating whether the execution of the method is successful or not. If the method needs to return some data, then such data is assigned to a special designated variable. One can consider returning a successful response status as emitting a Pass

```
int checkPIN(int[] p, int l) {
  try {
    if (trial > 0) {
      trial = trial − 1;
      if (length = l) {
        int i = 0
        while (i < length) {
          if (pin[i] == p[i])
            i = i + 1;
          else
            return SW_PIN_VERIFICATION_FAILED;
        }
        return SW_NO_ERROR;
      }
    }
    return SW_PIN_VERIFICATION_FAILED;
  } catch (CardTearException e) {
     val = false;
     return SW_UNKNOWN;
  }
}
```

Listing 1: An FSP of checkPIN

event, while returning a response status that indicates an error or a failure as emitting a Fail event. Any exception that can occur in the method shall be encoded as returning a response status indicating failure.

An FSP describes card tears using a **try**-**catch** construct, where the **catch** part catches a special exception called CardTearException. The **try** part describes an input-output relationship when card tears are not present. The **catch** part tells what the application has to do when a card tear occurs.

EXAMPLE 4.1 In this example we show an FSP of the command checkPIN discussed in the previous section. The FSP is shown in Listing 1. The PIN in the FSP is represented by the array variable pin, while the input PIN is represented by the array variable p. The length of the PIN is stored in the variable length.[2] The FSP also takes the length of the input PIN as an input. The return statement returning SW_NO_ERROR denotes a successful completion of the command, while other return statements in the **try** part denotes terminations with failures. In the **catch** part the validation status is set to false. Since the command has to return a response status, the **catch** part returns SW_UNKNOWN. One can consider returning SW_UNKNOWN in the **catch** part as emitting an Abrupt event.

□

Similar to SPMs, an FSP is 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 FSP can be described as a sequence of runs of commands in the FSP. For each run of the command, if the run exits from the **try** part, then the run of the FSP fetches the next input $C(a_1, \ldots, a_n)$ from the input sequence. If a card tear occurs, then the run of the command exits from the **catch** part or terminates abruptly, and in turn the run of the FSP simply terminates.

A run of 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;

[2] In Java the length of an array a is stored in the field length associated with the array, and this field can be accessed by the selector a.length. We store the length of an array PIN in a separate variable not associated with the array because the length of the PIN can be different from the length of the array.

- 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 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.

We prove properties of an FSP in the same way as proving properties of an SPM. First, we prove properties of each command in the FSP separately. To this end, we represent the command by two program-point flow graphs, one for the **try** part and the other for the **catch** part. We then define assertion functions of the two flow graphs and prove that the functions are weakly or strongly extendible.

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** clause of the SPM.

DEFINITION 4.2 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 $E_O = \{\mathsf{Pass}, \mathsf{Fail}, \mathsf{Abrupt}\}$ 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 $\varepsilon_{i_k} \neq \mathsf{Abrupt}$, then $\sigma_{i_k}(x) = \sigma'_{j_k}(Obs(x))$ for all $x \in O_{SPM}$,
- if $\varepsilon_{i_k} = \mathsf{Abrupt}$, then $\sigma_{i_l}(y) = \sigma'_{j_l}(Obs(y))$ for all $y \in Ab(O_{SPM})$.

□

The observable events in the above definition of RCR consist of events that occur at the exit points of both the SPM and the FSP. In the above definition, if there is an RCR between the SPM and the FSP, then if there is a run of the FSP from a state σ' such that the run terminates normally, then there is a run of the SPM from a state σ with (σ', σ) satisfies $\bigwedge_{x \in O_{SPM}} x = Obs(x)$ such that the run of the SPM also terminates normally and, upon termination of both runs, the values of corresponding observable variables coincide at the exit configurations of the runs. If the run of the FSP terminates abruptly, then there is also a run of the SPM with the same condition on the entry points such that the run terminates abruptly; but in this case, only observable variables modified by the SPM in the **abrupt** clause must have equal values at the exit configurations to their corresponding counterparts in the FSP.

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, 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 $x \in Ab(O_{SPM})$.

Let α be an assertion such that the assertion $\alpha \Rightarrow \bigwedge_{x \in O_{SPM}} x = Obs(x)$ is valid. Let P_1 be the flow graph of the **pass** and **fail** clauses of the command C in the SPM, and let P_2 be the flow graph of the **abrupt** clause of C. Let P'_1 and P'_2 be the flow graphs of, respectively, the **try** and the **catch** parts of the same command in the FSP. In the same way as denoting the exit points of commands in SPM, we denote the exit point of P'_1 by $exit_n(P'_1)$ and the exit point of P'_2 by $exit_a(P'_2)$. We define an assertion function $\hat{I}_1$ of (P_1, P'_1) such that

$$\hat{I}_1(entry(P_1), entry(P'_1)) = \hat{I}_1(exit_n(P_1), exit_n(P'_1)) = \alpha.$$

$\hat{I}_1$ can be defined elsewhere but for all points $p \neq exit_n(P_1)$ and $p' \neq exit_n(P'_1)$, we have $\hat{I}_1(p, exit_n(P'_1))$ and $\hat{I}_1(exit_n(P_1), p')$ undefined. Furthermore, let $S_1 = \{p' \mid \exists p, \phi. \hat{I}_1(p, p') = \phi\}$ be the set of points in P'_1 such that, for any point p' in S_1, there is a point p in P_1 and $\hat{I}_1(p, p')$ is defined. We say that a path $p_0, \ldots, p_n$ is S_1-simple if $n > 0$, and p_0 and p_n are in S_1 but none of $p_1, \ldots, p_{n-1}$ are in S. We require that the set S_1 covers P'_1. Next, we define a set $\hat{\Pi}_1$ of paths of (P_1, P'_1) such that the set $\{\pi' \mid \exists(\pi, \pi') \in \hat{\Pi}_1\}$ consists of all S_1-simple paths.

We define an assertion function $\hat{I}_2$ of (P_2, P'_2) as follows. On the pair $(entry(P_2), entry(P'_2))$ of entry points the function $\hat{I}_2$ is defined as ψ with the following requirements:

- the assertion $\alpha \Rightarrow \psi$ is valid, and
- for every finite run $(p'_0, \sigma'_0), \ldots, (p'_n, \sigma'_n)$ of P'_1, there is a finite run $(p_0, \sigma_0), \ldots, (p_m, \sigma_m)$ of P_1 such that (σ_0, σ'_0) satisfies α and (σ_m, σ'_n) satisfies ψ.

On the pair $(exit_a(P_2), exit_a(P'_2))$, the function $\hat{I}_2$ is defined as ψ' such that, for all $x \in Ab(O_{SPM})$, the assertion $\psi' \Rightarrow x = Obs(x)$ is valid. Furthermore, for all points $p \neq exit_a(P_2)$ and $p' \neq exit_a(P'_2)$, we have $\hat{I}_2(exit_n(P_2), p')$ and $\hat{I}_2(p, exit_n(P'_2))$ undefined. From the function $\hat{I}_2$, we can define a set S_2 from $\hat{I}_2$ similarly to defining the set S_1 from $\hat{I}_1$. The set S_2 must cover P'_2. We also define a set $\hat{\Pi}_2$ of paths of (P_2, P'_2) similarly to defining the set $\hat{\Pi}_1$.

THEOREM 4.3 *Let $\hat{I}_1$ and $\hat{I}_2$ be assertion functions as defined above, and $\hat{\Pi}_1$ and $\hat{\Pi}_2$ be sets of paths as defined above. Let $\mathbb{W}_1$ and $\mathbb{W}_2$ be the weak verification conditions associated, respectively, with $\hat{I}_1$ and $\hat{\Pi}_1$, and with $\hat{I}_2$ and $\hat{\Pi}_2$. If all assertions of $\mathbb{W}_1$ and $\mathbb{W}_2$ are valid, then there is an RCR between the SPM and the FSP of the command C.*

PROOF. First, since all assertions in $\mathbb{W}_1$ and $\mathbb{W}_2$ are valid, $\hat{I}_1$ and $\hat{I}_2$ are weakly extendible.

Suppose that card tears do not occur. Let $R = \gamma_0, \ldots, \gamma_i$ be a run of (P_1, P'_1) such that $\gamma_0 \models \hat{I}_1(entry(P_1), entry(P'_1))$ and $\gamma_i \models \hat{I}_1(p_i, p'_i)$ where $\gamma_i = (p_i, p'_i, \sigma_i, \sigma'_i)$. Let us suppose that $p'_i \neq exit_n(P'_1)$. Since the set $S_1 = \{p' \mid \exists p, \phi. \hat{I}_1(p, p') = \phi\}$ covers P'_1, from the configuration (p'_i, σ'_i), there is a computation sequence $(p'_i, \sigma'_i), \ldots, (p'_{i+n}, \sigma'_{i+n})$ of P'_1 such that the computation sequence passes through an S_1-simple path $\pi' = p'_i, \ldots, p'_{i+n}$ in P'_1. By the construction of the set $\hat{\Pi}_1$ and the validity of all assertions in the verification condition $\mathbb{W}_1$, there is a path $\pi = p_i, \ldots, p_{i+m}$ in P_1 such that (1) there is a computation sequence $(p_i, \sigma_i), \ldots, (p_{i+m}, \sigma_{i+m})$ of P_1 such that the computation sequence follows π, (2) $\hat{I}_1(p_{i+m}, p'_{i+n})$ is defined, and (3) $(\sigma_{i+m}, \sigma'_{i+n})$ models $\hat{I}_1(p_{i+m}, p'_{i+n})$.

Now, if $p'_{i+n} = exit_n(P'_1)$, then since for all points $p \neq exit_n(P_1)$ we have $\hat{I}_1(p, exit_n(P'_1))$ undefined, we have $p_{i+m} = exit_n(P_1)$. Thus, if the run of P'_1 is terminating, then the run of P_1 is terminating. Moreover, since $(\sigma_{i+m}, \sigma'_{i+n})$ satisfies $\hat{I}_1(p_{i+m}, p'_{i+n})$, $\hat{I}_1(exit_n(P_1), exit_n(P'_1)) = \alpha$, and the assertion $\alpha \Rightarrow \bigwedge_{x \in O_{SPM}} x = Obs(x)$ is valid, we have that $(\sigma_{i+m}, \sigma'_{i+n})$ satisfies $\bigwedge_{x \in O_{SPM}} x = Obs(x)$.

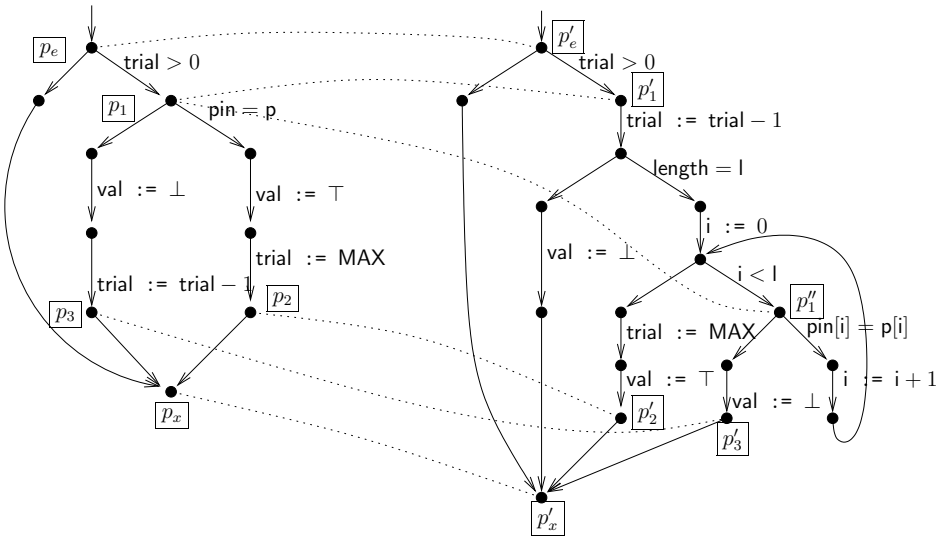

Figure 4: P_1 is on the left and P'_1 is on the right.

When card tears occur, then by the requirements of $\hat{I}_2(entry(P_2), entry(P'_2))$ we have the states on entering $entry(P_2)$ and $entry(P'_2)$ satisfy $\hat{I}_2(entry(P_2), entry(P'_2))$. With the same kind of reasoning as before, if the run of P'_2 reaches $exit_a(P'_2)$ with a state σ', then there is a run of P_2 reaching $exit_a(P_2)$ with a state σ. By the weak extendibility of $\hat{I}_2$, it follows that $(\sigma, \sigma') \models \hat{I}_2(exit_a(P_2), exit_a(P'_2))$. □

To prove that there is an RCR between the SPM and the FSP, first we require that for every command C and for every assertion function $\hat{I}_1$ of the flow graphs representing the **pass** and **fail** clauses of the command C in the SPM and the **try** part of the same command in the FSP,

$$\hat{I}_1(entry(P_1), entry(P'_1)) = \hat{I}_1(exit_n(P_1), exit_n(P'_1)) = \alpha,$$

where α is the assertion expressing the correspondence between the SPM and the FSP. Second, we have to prove that α holds when the SPM and the FSP are initialized. When a command C_1 calls another command C_2 both in the SPM and in the FSP, then since a command in a smart-card application is usually *not* recursive, we can inline the command C_2.

EXAMPLE 4.4 In this example we will show that there is an RCR between the SPM and the FSP of the command checkPIN. The SPM of the command is described in Figure 2 and the FSP is described in Listing 1. Let us first consider the flow graph representing the **pass** and **fail** clauses of the SPM and the flow graph representing the **try** part. Call the former flow graph P_1 and the latter P'_1. These flow graphs are depicted in Figure 4.

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

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

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 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

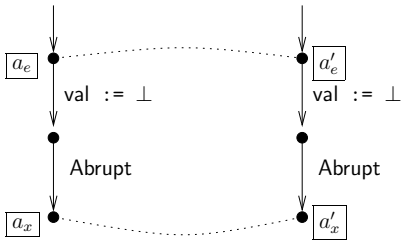

Figure 5: P_2 is on the left and P'_2 is on the right.

$p[i] = p'[i]$. 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. Note that the set $S_1 = \{p'_e, p'_1, p''_1, p'_2, p'_3, p'_x\}$ of points in P'_1 covers P'_1.

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:

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

Note that the set $\{\pi' \mid \exists \pi.(\pi, \pi') \in \hat{\Pi}_1\}$ consists of all S_1-simple paths. 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** clause and the **catch** part. Call the former one P_2 and the latter P'_2. These flow graphs are depicted in Figure 5. 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. Note that the set $S_2 = \{a'_e, a'_x\}$ covers P'_2. Note also that since the assertion $\top$ is satisfied by any state, the assertion $\hat{I}_2(a_e, a'_e)$ satisfies the requirements of the assertion ψ described before. We define a set $\hat{\Pi}_2$ of paths of (P_2, P'_2) such that the set consists only of the path

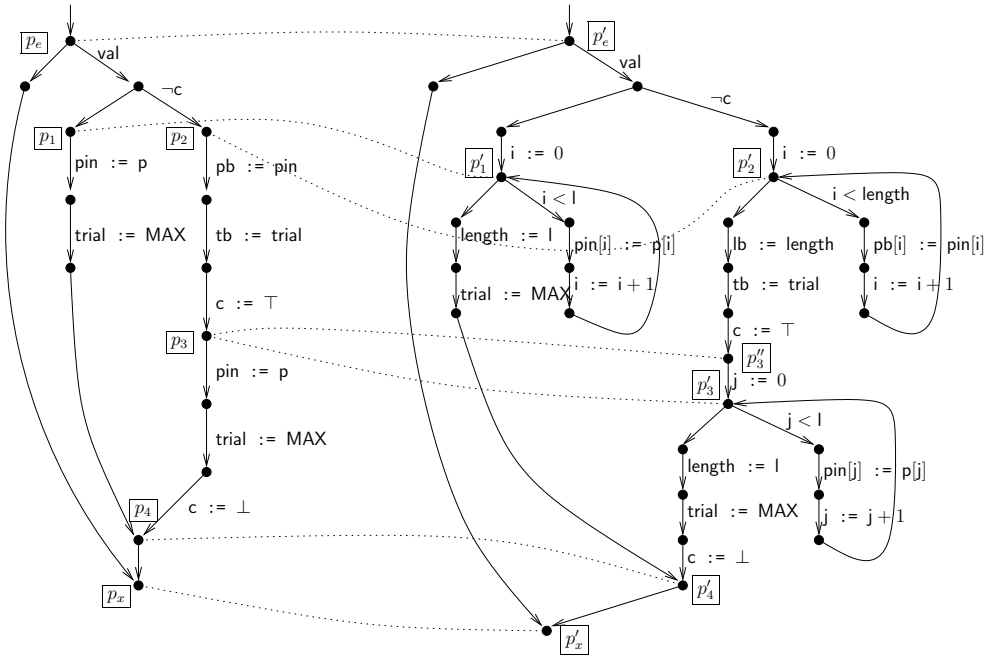

Figure 6: P_1 is on the left and P'_1 is on the right.

$(\pi_{a_e,a_x}, \pi_{a'_e,a'_x})$. The path $\pi_{a'_e,a'_x}$ is the only S_2-simple path. One can prove easily that all assertions in the weak verification condition associated with $\hat{I}_2$ and $\hat{\Pi}_2$ are valid. Thus, by Theorem 4.3 there is an RCR between the SPM and the FSP of the command checkPIN. □

EXAMPLE 4.5 We consider in this example the command updatePIN whose SPM is depicted in Figure 3. The flow graphs that represent the **pass** and **fail** clauses of the SPM and the **try** part of the FSP are depicted in Figure 6. Let us call the former flow graph P_1 and the latter one P'_1.

Let the set $O_{SPM} = \{\mathsf{c}, \mathsf{trial}, \mathsf{pin}, \mathsf{p}, \mathsf{val}, \mathsf{MAX}, \varepsilon\}$ be the set of observable variables of the SPM, and the set O_{FSP} be the set of observable variables of the FSP such that O_{FSP} consists of the primed counterparts of all variables in O_{SPM}. The one-to-one correspondence Obs between O_{SPM} and O_{FSP} simply maps each variable in O_{SPM} to its primed counterpart in O_{FSP}. The following assertions express the correspondence between the SPM and the FSP:

$$\begin{array}{lcllcl}
\phi_1 & \Leftrightarrow & \mathsf{trial} = \mathsf{trial}' & \phi_5 & \Leftrightarrow & \mathsf{c} = \mathsf{c}' \\
\phi_2 & \Leftrightarrow & \mathsf{val} = \mathsf{val}' & \phi_6 & \Leftrightarrow & \mathsf{MAX} = \mathsf{MAX}' \\
\phi_3 & \Leftrightarrow & \mathsf{pin} \sim (\mathsf{pin}', \mathsf{length}') & \phi_7 & \Leftrightarrow & \varepsilon = \varepsilon' \\
\phi_4 & \Leftrightarrow & \mathsf{p} \sim (\mathsf{p}', \mathsf{l}') & \phi_8 & \Leftrightarrow & \mathsf{c} \wedge \mathsf{c}' \Rightarrow \mathsf{pb} \sim (\mathsf{pb}', \mathsf{lb}') \wedge \mathsf{tb} = \mathsf{tb}'
\end{array}$$

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

$$\begin{array}{lcl}
\hat{I}_1(p_e, p'_e) & = & \bigwedge_{i=1}^{8} \phi_i \\
\hat{I}_1(p_1, p'_1) & = & \bigwedge_{i=1}^{2} \phi_i \wedge \bigwedge_{i=4}^{8} \phi_i \wedge \mathsf{c} \wedge (\forall l.0 \leq l < \mathsf{i} \Rightarrow \mathsf{pin}'[l] = \mathsf{p}'[l]) \\
\hat{I}_1(p_2, p'_2) & = & \bigwedge_{i=1}^{8} \phi_i \wedge \neg\mathsf{c} \wedge (\forall l.0 \leq l < \mathsf{i} \Rightarrow \mathsf{pb}'[l] = \mathsf{pin}'[l]) \\
\hat{I}_1(p_3, p'_3) & = & \bigwedge_{i=1}^{2} \phi_i \wedge \bigwedge_{i=4}^{8} \phi_i \wedge \mathsf{c} \wedge (\forall l.0 \leq l < \mathsf{j} \Rightarrow \mathsf{pin}'[l] = \mathsf{p}'[l]) \\
\hat{I}_1(p_3, p''_3) & = & \bigwedge_{i=1}^{8} \phi_i \wedge \mathsf{c} \wedge (\neg\mathsf{c} \wedge \neg\mathsf{c}' \Rightarrow \phi_1 \wedge \phi_3) \\
\hat{I}_1(p_4, p'_4) & = & \bigwedge_{i=1}^{8} \phi_i \wedge (\neg\mathsf{c} \wedge \neg\mathsf{c}' \Rightarrow \phi_1 \wedge \phi_3) \\
\hat{I}_1(p_x, p'_x) & = & \bigwedge_{i=1}^{8} \phi_i.
\end{array}$$

The function $\hat{I}_1$ is undefined elsewhere. Note that the set $S_1 = \{p'_e, p'_1, p'_2, p'_3, p''_3, p'_4, p'_x\}$ of points in P'_1

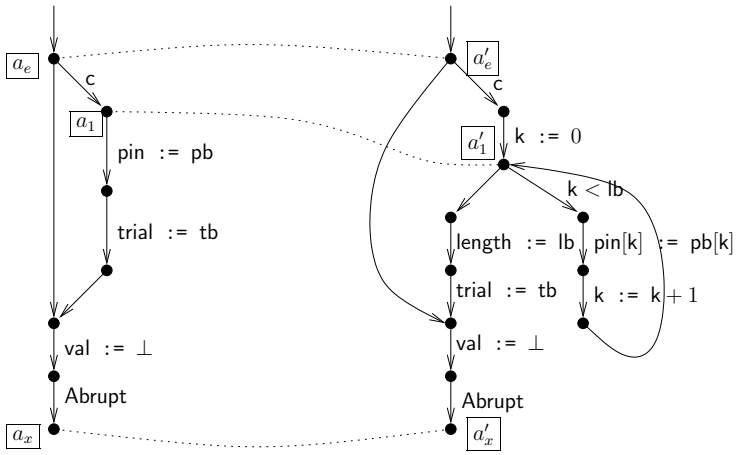

Figure 7: P_2 is on the left and P'_2 is on the right.

covers P'_1. We also define a set $\hat{\Pi_1}$ of paths of (P_1, P'_1) such that the set consists of the following paths:

$$(\pi_{p_e,p_1}, \pi_{p'_e,p'_1}), (\pi_{p_e,p_2}, \pi_{p'_e,p'_2}), (\pi_{p_e,p_x}, \pi_{p'_e,p'_x}),$$
$$(\pi_{p_1}, \pi_{p'_1,p'_1}), (\pi_{p_1,p_4}, \pi_{p'_1,p'_4}), (\pi_{p_2}, \pi_{p'_2,p'_2}), (\pi_{p_2,p_3}, \pi_{p'_2,p''_3}),$$
$$(\pi_{p_3}, \pi_{p''_3,p'_3}), (\pi_{p_3}, \pi_{p'_3,p'_3}), (\pi_{p_3,p_4}, \pi_{p'_3,p'_4}), (\pi_{p_4,p_x}, \pi_{p'_4,p'_x}).$$

The path $\pi^{\neg c'}_{p'_e,p'_4}$ is the path from p'_e to p'_4 through the $\neg c'$ branch. Similarly for $\pi^{\neg c}_{p_e,p_4}$. Note that the set $\{\pi' \mid \exists\pi.(\pi,\pi') \in \hat{\Pi_1}\}$ consists of all S_1-simple paths. 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** clause and the **catch** part. Call the former one P_2 and the latter P'_2. These flow graphs are depicted in Figure 7. We define an assertion function $\hat{I_2}$ of (P_2, P'_2) as follows:

$$\begin{array}{lcl}
\hat{I_2}(a_e, a'_e) & = & \phi_8 \wedge (\neg \mathsf{c} \wedge \neg \mathsf{c}' \Rightarrow \phi_1 \wedge \phi_3) \\
\hat{I_2}(a_1, a'_1) & = & \mathsf{pb} \sim (\mathsf{pb}', \mathsf{lb}') \wedge \mathsf{tb} = \mathsf{tb}' \wedge (\forall l.0 \leq l < \mathsf{k} \Rightarrow \mathsf{pin}'[l] = \mathsf{pb}'[l]) \\
\hat{I_2}(a_x, a'_x) & = & \phi_1 \wedge \phi_2 \wedge \phi_3
\end{array}$$

The function $\hat{I_2}$ is undefined elsewhere. Note that the set $S_2 = \{a'_e, a'_1, a'_x\}$ covers P'_1.

We also define a set $\hat{\Pi_2}$ of paths of (P_2, P'_2) such that the set consists of the following paths:

$$(\pi_{a_e,a_1}, \pi_{a'_e,a'_1}), (\pi_{a_1}, \pi_{a'_1,a'_1}), (\pi_{a_e,a_x}, \pi_{a'_e,a'_x}), (\pi_{a_1,a_x}, \pi_{a'_1,a'_x}).$$

Note that the set $\{\pi' \mid \exists\pi.(\pi,\pi') \in \hat{\Pi_2}\}$ consists of all S_2-simple paths. One can prove that all assertions in the weak verification condition associated with $\hat{I_2}$ and $\hat{\Pi_2}$ are valid.

Note also that the requirements for the assertion $\hat{I_2}(a_e, a'_e)$ is satisfied by the assertions $\hat{I_1}(p_3, p''_3)$ and $\hat{I_1}(p_4, p'_4)$ and the weak extendibility of $\hat{I_1}$. Therefore, there is an RCR between the command updatePIN of the SPM and of the FSP. □

5 Proving RCRs between FSPs and TDSs

In this section we discuss RCRs between FSPs and TDSs. Before discussing RCRs, we first describe TDSs. In EDEN2, a TDS of a smart-card application is a program describing a low-level design of the application. A TDS is also called a *reference* implementation. The language used to write a TDS in EDEN2 is a subset of Java. This subset includes memory characteristics and transaction mechanism of Java Card [Sun, 2008, Chen, 2000]. First, in the language of TDSs 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*.

The language of TDSs offers a transaction mechanism that resembles the transaction mechanism of Java Card API. The depth of a transaction is at most 1, that is, there is no nested transaction. The methods for managing transactions are beginTransaction, commitTransaction, and abortTransaction. A transaction is started by calling beginTransaction. When another transaction is in progress, calling beginTransaction throws an exception. Throwing such an exception can be represented by a return statement that returns a value indicating an error. The transaction is ended by calling either commitTransaction or abortTransaction; but when no transaction is in progress, calling commitTransaction or abortTransaction throws an exception. Similarly, throwing such an exception can be represented by a return statement. When a transaction is in progress, any updates to persistent variables are conditional. That is, if the transaction is ended by commitTransaction, then all updates to the persistent variables are committed; otherwise, all updates are discarded. The updates of transient variables are unconditional, regardless a transaction is in progress or not. Later, we will introduce a boolean variable inTransaction to keep track if a transaction is in progress or not. When a transaction begins, the value of inTransaction is set to true, and when it ends, the value of inTransaction is set to false. One can set the value of inTransaction to false to escape from a transaction. This feature is useful for variables whose updates must be unconditional. In Java Card such a feature is provided by non-atomic API methods [Sun, 2008]. Discussion on Java Card non-atomic API methods and their effects on transactions can be found in [Hubbers and Poll, 2004a].

Similar to FSPs, each command in a TDS is a Java method. Card tears are described using a **try-catch** construct in the method. For an FSP, the writer of the FSP has a freedom to write, in the **catch** part of a command, what the command has to do when a card tear occurs. For a TDS, the **catch** part of the command models *only* the clearing of transient memory and the effects of transactions. Clearing transient memory means setting all transient variables to their default values.

To cope with transactions, we modify each command in the TDS as follows:

- For each persistent variable, we introduce a fresh variable for the bookkeeping of the old value of the variable during a transaction.
- We introduce a special global variable inTransaction to keep track whether a transaction is in progress or not.
- In the **catch** part of the command, we add statements that undo the effects of a transaction if the transaction is in progress.
- We replace any call to beginTransaction by the statements

  ```
  if (inTransaction) return TRANSACTION_IN_PROGRESS;
  inTransaction = true;
  ```

  Between the above two statements we add statements that bookkeep the current values of persistent variables.
- We replace any call to commitTransaction and to abortTransaction by the statements

  ```
  if (!inTransaction) return TRANSACTION_NOT_IN_PROGRESS;
  inTransaction = false;
  ```

  Particularly for abortTransaction, between the two statements above we add statements that undo the effects of the in-progress transaction.

Our modelling of transactions follows the modelling of Java Card transactions in [Hubbers and Poll, 2004b].

EXAMPLE 5.1 We illustrate the modelling of transactions and card tears in TDSs in this example. The programs in this example is adapted from [Hubbers and Poll, 2004b]. The original command, or the command

before the modification, is shown below on the lefthand side, and the modified version is on the righthand side. We assume here that p is a persistent variable, while t is a transient variable. The default value of t is 0.

```
int command() {
  beginTransaction();
  p = p + 1; t = t + 1;
  p = p + 1;
  if (p < 10)
    commitTransaction();
  else
    abortTransaction();
  t = t + 1;
  return SW_NO_ERROR;
}
```

```
int command() {
  try {
    if (inTransaction)
      return TRANSACTION_IN_PROGRESS;
    pb = p;
    inTransaction = true;
    p = p + 1; t = t + 1; p = p + 1;
    if (p < 10)
      if (!inTransaction)
        return TRANSACTION_NOT_IN_PROGRESS;
      inTransaction = false;
    else {
      if (!inTransaction)
        return TRANSACTION_NOT_IN_PROGRESS;
      p = pb;
      inTransaction = false;
    }
    t = t + 1;
    return SW_NO_ERROR;
  } catch (CardTearException e) {
    if (inTransaction) p = pb;
    t = 0;
    return SW_UNKNOWN;
  }
}
```

□

Similar to FSPs, a TDS 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 TDSs is the same as the notion of run of FSPs. Proving properties of TDSs can be done in the same way as proving properties of SPMs or FSPs.

Having described TDSs, we now define RCRs between FSPs and TDSs. Let us first denote by $Pr(X)$ the set of persistent variables in the set X of variables of a TDS. Later in the definition of RCRs between an FSP and a TDS we require that observable persistent variables of the TDS 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 TDS and of the FSP, if no transaction is in progress and the observable persistent variables of the TDS 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 TDS 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 $\mathsf{Write}\_x(v)$. The following assertion axiomatizes the event function:

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

where the equality $\mathsf{Write}\_x = \mathsf{Write}\_y$ 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 TDS 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 TDS, emits $\mathsf{Write}\_x(v)$, where v is the updated value of y.

- In the **try** part of the TDS,
  - if no transaction is in progress, that is the variable inTransaction is false, then the update of an observable persistent variable x emits $\{\mathsf{Write\_}x(v)\}$, 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 inTransaction is not caused by a call to abortTransaction, then the resetting emits $\{\mathsf{Write\_}x_0(v_0), \ldots, \mathsf{Write\_}x_n(v_n)\}$. However, when the resetting of inTransaction is caused by a call to abortTransaction or no observable variables are updated, then no set of events is emitted.

Note that the set of events emitted by the TDS is always nonempty.

For comparing events of the TDS and events of the FSP, we say that a nonempty set $\{\varepsilon_0, \ldots, \varepsilon_m\}$ of TDS'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 TDS'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 TDS to variables of the FSP. We assume that the FSP and the TDS 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 TDS and any variable x' of the FSP, $x' = Obs(x)$ if and only if $x = Obs(x')$.

DEFINITION 5.2 Let O_{FSP} and O_{TDS} be the sets of observable variables of, respectively, an FSP and a TDS, and Obs be a one-to-one correspondence between these sets. Let the sets

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

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

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

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

□

EXAMPLE 5.3 We illustrate the above definition in this example. For simplicity, consider the following two programs, where the TDS is on the left and the FSP is on the right.

```
/* x,y,z are persistent */
/* t is transient */
z = 1;                          z' = 1;
beginTransaction();             x' = z' + 1;
x = z + 1;                      t' = x';
z = 3;                          y' = 4;
t = x;                          z' = 3;
y = 4;                          y' = 5;
commitTransaction();
y = 5;
```

The variables $\mathsf{x}, \mathsf{y}, \mathsf{z}$ are persistent, while the variable t is transient. All of these variables are assumed to be observable. The one-to-one correspondence Obs between the observables of TDS and the observables of FSP maps every variable of TDS to its primed counterpart.

There is an RCR between the TDS and the FSP with the following reasoning. First, the sequence of events in the TDS is

$$\{\mathsf{Write\_z}(1)\}, \{\mathsf{Write\_x}(2), \mathsf{Write\_z}(3), \mathsf{Write\_y}(4)\}, \{\mathsf{Write\_y}(5)\}.$$

In the FSP we have a matching sequence:

$$\mathsf{Write\_z}(1), \mid \mathsf{Write\_x}(2), \mathsf{Write\_y}(4), \mathsf{Write\_z}(3) \mid, \mathsf{Write\_y}(5)$$

We put a separator "|" as an aid for the readers to see how the sequences match. That is, for example, the set $\{\mathsf{Write\_x}(2), \mathsf{Write\_z}(3), \mathsf{Write\_y}(4)\}$ matches the sequence $\mathsf{Write\_x}(2), \mathsf{Write\_y}(4), \mathsf{Write\_z}(3)$. Second, the values of corresponding persistent variables coincide at every pair of matching events. Third, the value of transient variable t coincide with the value of t' on termination. □

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 TDS. For the RCR between an FSP and a TDS, emitting an event means concatenating the event to the current value of the special variable ε. That is, for an event or a set of events E, emitting E is equal to the assignment ε := $\varepsilon; \mathsf{E}$. We say that the value v of ε of the TDS is equal to the value v' of ε of the FSP if and only if v matches v'. The variable ε of the TDS is considered transient.

Particularly for the TDS, we use another special variable ε_t to keep track the updated observable persistent variables when a transaction is in progress. When the variable inTransaction 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 $\varepsilon_t \cup \{\mathsf{Write\_}x(v)\}$. When the variable inTransaction is set to false, the variable ε is set to $\varepsilon; \varepsilon_t$ only if the reseting of inTransaction is not caused by abortTransaction. Moreover, when the TDS emits Pass or Fail, 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 TDS emits Abrupt, then the content of ε_t is discarded and ε is updated with $\varepsilon;$ Abrupt.

Regarding the updates of variables during a transaction, one might need to modify programs further to apply our technique of recording updates with the special variables ε and ε_t. For example, suppose that in the TDS a transaction is in progress and a persistent variable x is updated by the following statements:

```
x = x + 1;
x = x + 1;
```

But its corresponding counterpart x' in the FSP is updated by a single statement x' = x' + 2. For simplicity, assume that the domains of x and x' are the same. The variable ε of the TDS will be set to $\{\mathsf{Write\_x}(v_1), \mathsf{Write\_x}(v_2)\}$, for some values v_1, v_2, but the variable ε of the FSP will be set to $\mathsf{Write\_x}(v_2)$. To handle this problem, one can always translate both programs into SSA form [Alpern et al., 1988] such that in the program texts there is only one assignment to each variable. That is, the translation of the TDS into SSA form results in

```
x1 = x0 + 1;
x2 = x1 + 1;
```

and the translation of the FSP into SSA form results in x1' = x0' + 2. For the correspondence between variables, the variable x0 corresponds to the variable $x0'$ and the variable x2 corresponds to the variable $x1'$. Thus, the variable ε of the TDS will be set to $\{\mathsf{Write\_x2}(v_2)\}$ that matches the event $\mathsf{Write\_x2}(v_2)$ set to the variable ε of the FSP.

We apply the theory of inter-program properties to proving an RCR between an FSP and a TDS by proving the RCR between each corresponding commands separately. Let α be an assertion describing the correspondence between variables induced by the one-to-one correspondence Obs between the sets O_{FSP} and O_{TDS} of observable variables of the FSP and the TDS. That is, the assertion $\alpha \Rightarrow \bigwedge_{x \in O_{TDS}} x = Obs(x)$ is valid. Let P_1 and P_1' be the flow graphs of, respectively, the FSP and the TDS of the **try** part of a command C. Let also P_2 and P_2' be the flow graphs of, respectively, the FSP and the TDS of the **catch** part of the same command.

We define an assertion function $\hat{I}_1$ of (P_1, P_1') such that

$$\hat{I}_1(entry(P_1), entry(P_1')) = \hat{I}_1(exit_n(P_1), exit_n(P_1')) = \alpha.$$

$\hat{I}_1$ can be defined elsewhere but for all points $p \neq exit_n(P_1)$ and $p' \neq exit_n(P_1')$, we have $\hat{I}_1(exit_n(P_1), p')$ and $\hat{I}_1(p, exit_n(P_1'))$ undefined. Furthermore, let $S_1 = \{p' \mid \exists p, \phi. \hat{I}_1(p, p') = \phi\}$ be the set of points in P_1' such that, for any point p' in S_1, there is a point p in P_1 and $\hat{I}_1(p, p')$ is defined. We require that the set S_1 covers P_1'. Next, we define a set $\hat{\Pi}_1$ of paths of (P_1, P_1') such that the set $\{\pi' \mid \exists(\pi, \pi') \in \hat{\Pi}_1\}$ consists of all S_1-simple paths.

Similarly for the pair (P_2, P_2'), we define an assertion function $\hat{I}_2$ of (P_2, P_2') as follows. On the pair $(entry(P_2), entry(P_2'))$ of entry points the function $\hat{I}_2$ is defined as ψ with the following requirements:

- the assertion $\alpha \Rightarrow \psi$ is valid,
- the assertion $\psi \Rightarrow \varepsilon = Obs(\varepsilon) \wedge \bigwedge_{x \in Pr(O_{TDS})} x = Obs(x))$ is valid, and
- for every finite run $(p_0', \sigma_0'), \ldots, (p_n', \sigma_n')$ of P_1', there is a finite run $(p_0, \sigma_0), \ldots, (p_m, \sigma_m)$ of P_1 such that (σ_0, σ_0') satisfies α and (σ_m, σ_n') satisfies ψ.

On the pair $(exit_a(P_2), exit_a(P_2'))$ of exit points, the function $\hat{I}_2$ is defined as ψ' such that the assertion $\psi' \Rightarrow \bigwedge_{x \in O_{TDS}} x = Obs(x)$ is valid. Furthermore, for all points $p \neq exit_a(P_2)$ and $p' \neq exit_a(P_2')$, we have $\hat{I}_2(exit_n(P_2), p')$ and $\hat{I}_2(p, exit_n(P_2'))$ undefined. From the function $\hat{I}_2$, we can define a set S_2 similarly to defining the set S_1 from $\hat{I}_1$. The set S_2 must cover P_2'. We also define a set $\hat{\Pi}_2$ of paths of (P_2, P_2') similarly to defining the set $\hat{\Pi}_1$.

THEOREM 5.4 *Let $\hat{I}_1$ and $\hat{I}_2$ be assertion functions as defined above, and $\hat{\Pi}_1$ and $\hat{\Pi}_2$ be sets of paths as defined above. Let $\mathbb{W}_1$ and $\mathbb{W}_2$ be the weak verification conditions associated, respectively, with $\hat{I}_1$ and $\hat{\Pi}_1$, and with $\hat{I}_2$ and $\hat{\Pi}_2$. If all assertions of $\mathbb{W}_1$ and $\mathbb{W}_2$ are valid, then there is an RCR between the FSP and the TDS of the command C.* □

The proof of the above theorem is similar to that of Theorem 4.3. First, consider runs that terminate normally. Since the assertions

$$\hat{I}_1(entry(P_1), entry(P_1')) \Rightarrow \varepsilon = Obs(\varepsilon)$$
$$\hat{I}_1(exit_n(P_1), exit_n(P_1')) \Rightarrow \varepsilon = Obs(\varepsilon)$$

are valid and the function I_1 is weakly extendible, it is guaranteed that the sequence of sets of events emitted by the run of the TDS matches the sequence of events emitted by the run of the FSP. Moreover, since the assertion

$$\hat{I}_1(entry(P_1), entry(P_1')) \Rightarrow \bigwedge_{x \in Pr(O_{TDS})} x = Obs(x)$$

is valid, it follows that after each corresponding updates of ε and $Obs(\varepsilon)$ the value of each variable $y \in Pr(O_{TDS})$ coincides with the value of its counterpart in the FSP. It then follows from the assertion $\hat{I}_1(exit_n(P_1), exit_n(P_1')$ that when the runs terminate normally, or emitting Pass or Fail, the values of each corresponding observable variables coincide.

Let us now consider runs that terminate abruptly. Recall that in the TDS, when a transaction is in progress, updating an observable persistent variable does not emit any event; instead, the update is remembered by the special variable ε_t. Since the assertion $\hat{I}_1(exit_n(P_1), exit_n(P_1')) \Rightarrow \varepsilon = Obs(\varepsilon)$ is valid, for any run of the TDS that emits event ε_1 and then terminates abruptly, there is a run of the FSP, on the same input for observable variables, such that the run of the FSP emits event ε_1' and terminates abruptly, and ε_1 matches ε_1'. The match between ε_1 and ε_1' fulfills the requirement that the assertion $\hat{I}_2(entry(P_2), entry(P_2')) \Rightarrow \varepsilon = Obs(\varepsilon) \wedge \bigwedge_{x \in Pr(O_{TDS})} x = Obs(x)$ is valid. Moreover, since undoing the effects of an in-progress transaction only involves values of persistent variables before the transaction begins, it is safe to assert $\bigwedge_{x \in Pr(O_{TDS})} x = Obs(x)$ at the entry of (P_2, P_2'). Moreover, by the assertion $\hat{I}_2(exit_a(P_2), exit_a(P_2'))$ and the weak extendibility of $\hat{I}_2$, it follows that, at the exit of (P_2, P_2') (or when the runs emit Abrupt), the assertion $\bigwedge_{x \in Pr(O_{TDS})} x = Obs(x)$ is preserved by the undoing of the effects of the transaction, and the value of each observable transient variable and the value of its counterpart coincide.

EXAMPLE 5.5 We consider again the command checkPIN in this example. Consider the FSP of the command shown on the righthand side of Figure 4. The figure shows the flow graph of the **try** part of the command. The value of variable trial is decremented before the PIN is checked against the input PIN. This is a desirable security property. If the value of trial is decremented after the PIN is checked against the input PIN, then if one can observe the run of the command, then he can produce a card tear once he knows that the PIN is not equal to the input PIN, and thus he can input wrong PIN infinitely often. This is a kind of security attack which is not captured by the SPM of checkPIN.

In the presence of transactions, having the value of trial decremented before the PIN check is not sufficient to handle the above security attack. First, the variable trial must be persistent since it must keep its value when power is switched off. Second, the decrement of the value of trial must not participate in a transaction; otherwise, if a card tear occurs, the content of the variable trial will be restored with its latest value before the transaction begins, and thus one can possibly try wrong PIN infinitely often.

In the TDS of the command checkPIN, for simplicity, we require that any update of the value of trial shall not participate in any transaction, or in other words, the update shall be unconditional. Figure 8 depicts the FSP and the TDS 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 TDS and it is called P_1'. In P_1', the variable inTrans denotes the variable inTransaction used in desugaring the transaction mechanism of TDSs. 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_{TDS} be the set of observable variables of the TDS such that O_{TDS} consists of the primed counterparts of all variables in O_{FSP}. The one-to-one correspondence Obs between O_{FSP} and O_{TDS} maps each variable in O_{FSP} to its primed counterpart in O_{TDS}. 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 $S_1' = \{p' \mid \exists p, \varphi . \hat{I}_1(p, p') = \varphi\}$ be the set of points in P_1' such that for each point p' in S_1', there is a point p in P_1 and $\hat{I}_1(p, p')$ is defined. Note that S_1' covers P_1'. Similarly, let $S_1 = \{p \mid \exists p', \varphi . \hat{I}_1(p, p') = \varphi\}$ 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

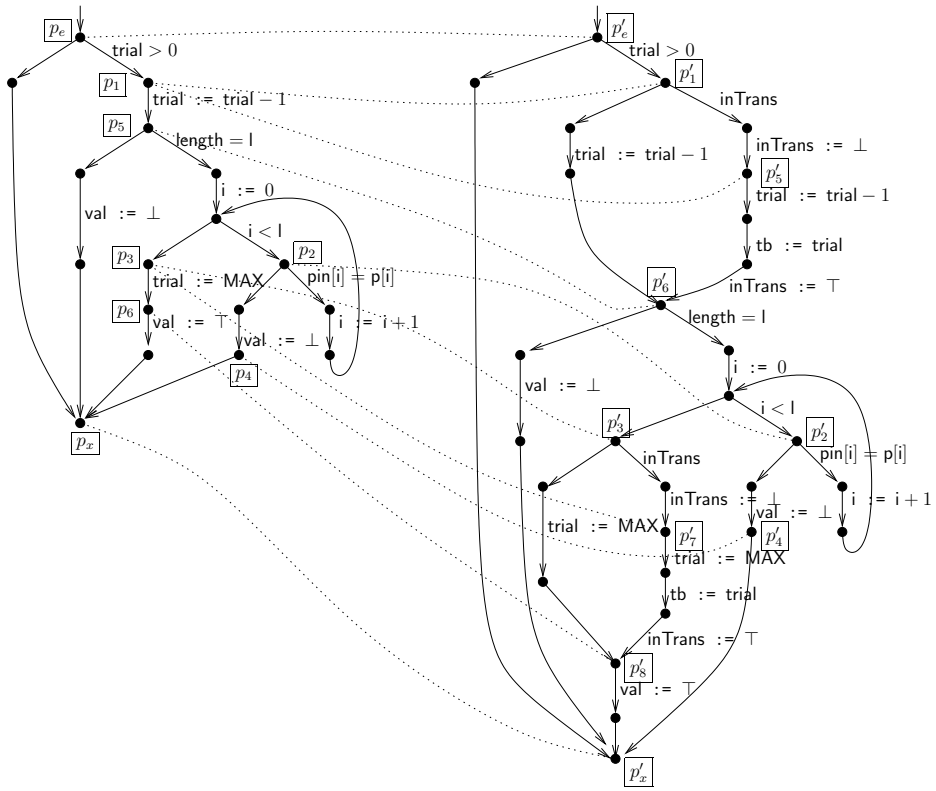

Figure 8: P_1 is on the left and P'_1 is on the right.

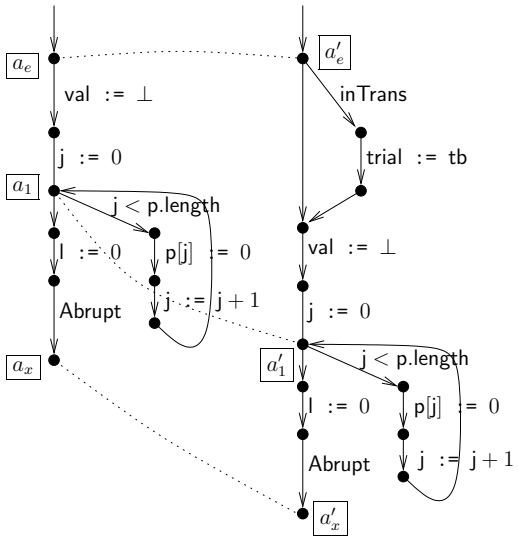

Figure 9: P_2 is on the left and P'_2 is on the right.

- 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** part of the command updatePIN. The flow graphs P_2 and P'_2 in Figure 9 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 5. The flow graph P_2 in Figure 9 updates the variables p and l. The counterparts of these variables in the TDS 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 9 and the flow graph P_2 of the SPM in Figure 5 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 $\hat{I}_2(a_e, a'_e) \Rightarrow \bigwedge_{x \in Pr(O_{TDS})} x = Obs(x) \wedge \varepsilon = \varepsilon'$ 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. Note that S'_2 covers P'_2. 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. Therefore, there is an RCR between the FSP and the TDS of the command checkPIN. □

[3] Stack variables are transient variables.

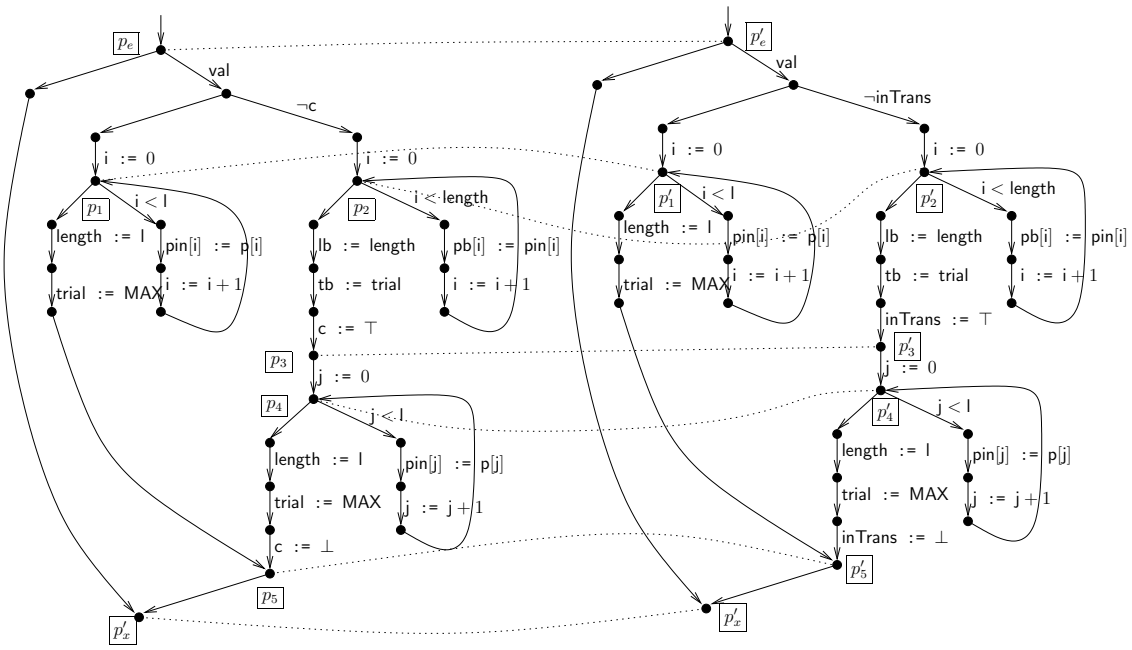

Figure 10: P_1 is on the left and P_1' is on the right.

EXAMPLE 5.6 In this example we will show that there is an RCR between the FSP and the TDS of the command updatePIN. Figure 10 shows the **try** parts of the command. The flow graphs P_1 and P_1' are the FSP and the TDS, respectively. As usual, to distinguish variables of the FSP from variable of the TDS, we use primed notation for variables of the TDS. Let the sets

$$\begin{array}{lcl} O_{FSP} & = & \{\mathsf{trial}, \mathsf{pin}, \mathsf{length}, \mathsf{p}, \mathsf{l}, \mathsf{val}, \mathsf{MAX}, \mathsf{c}, \varepsilon\} \\ O_{TDS} & = & \{\mathsf{trial}', \mathsf{pin}', \mathsf{length}', \mathsf{p}', \mathsf{l}', \mathsf{val}', \mathsf{MAX}', \mathsf{inTrans}', \varepsilon\} \end{array}$$

be the sets of observable variables of, respectively, the FSP and the TDS. The one-to-one correspondence Obs maps each variable in O_{FSP}, except c, to its primed counterpart in O_{TDS}; the variable c itself is mapped to inTrans′. The following assertions express the correspondence between the observable variables:

$$\begin{array}{lcl} \phi_1 & \Leftrightarrow & \mathsf{trial} = \mathsf{trial}' \wedge \mathsf{MAX} = \mathsf{MAX}' \wedge \mathsf{length} = \mathsf{length}' \wedge \mathsf{pin} = \mathsf{pin}' \\ \phi_2 & \Leftrightarrow & \mathsf{val} = \mathsf{val}' \wedge \mathsf{c} = \mathsf{inTrans}' \wedge \mathsf{p} = \mathsf{p}' \wedge \mathsf{l} = \mathsf{l}' \\ \phi_3 & \Leftrightarrow & \varepsilon = \varepsilon' \\ \phi_4 & \Leftrightarrow & \mathsf{c} \wedge \mathsf{inTrans}' \Rightarrow \mathsf{pb} = \mathsf{pb}' \wedge \mathsf{lb} = \mathsf{lb}' \wedge \mathsf{tb} = \mathsf{tb}' \\ \phi & \Leftrightarrow & \phi_1 \wedge \phi_2 \wedge \phi_3 \wedge \phi_4 \end{array}$$

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

$$\begin{array}{lcl} \hat{I}_1(p_e, p_e') & = & \phi \\ \hat{I}_1(p_1, p_1') & = & \phi_1 \wedge \phi_2 \wedge \phi_4 \wedge \mathsf{c} \wedge \varepsilon = \varepsilon'; \varepsilon_t' \wedge \mathsf{i} = \mathsf{i}' \\ \hat{I}_1(p_2, p_2') & = & \phi \wedge \neg\mathsf{c} \wedge \mathsf{i} = \mathsf{i}' \\ \hat{I}_1(p_3, p_3') & = & \phi \wedge \mathsf{c} \\ \hat{I}_1(p_4, p_4') & = & \phi_1 \wedge \phi_2 \wedge \phi_4 \wedge \mathsf{c} \wedge \varepsilon = \varepsilon'; \varepsilon_t' \wedge \mathsf{j} = \mathsf{j}' \\ \hat{I}_5(p_5, p_5') & = & \phi \\ \hat{I}_5(p_x, p_x') & = & \phi \end{array}$$

The function $\hat{I}_1$ is undefined elsewhere. Note that the set $S_1 = \{p_e', p_1', p_2', p_3', p_4', p_5', p_x'\}$ of points in P_1'

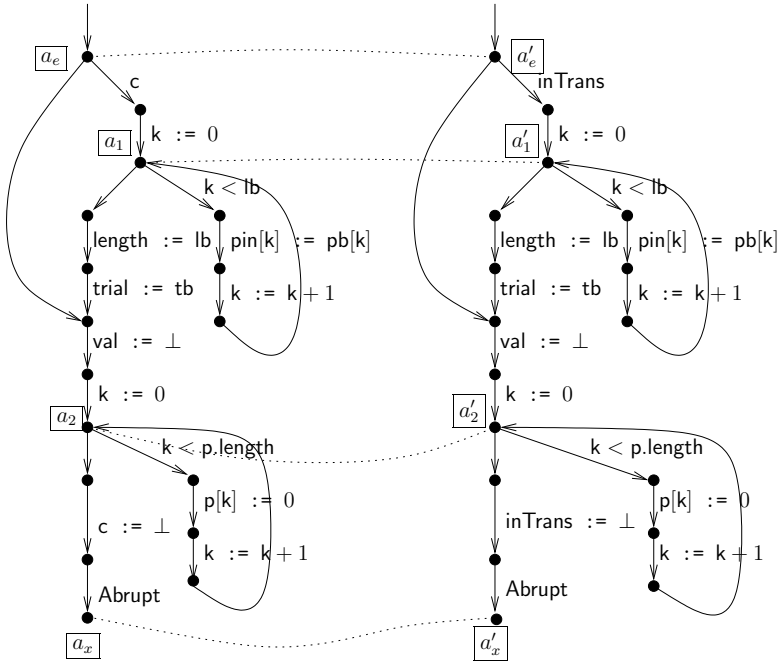

Figure 11: P_2 is on the left and P'_2 is on the right.

covers P'_1. We also define a set $\hat{\Pi_1}$ of paths of (P_1, P'_1) such that the set consists of the following paths:

$$(\pi_{p_e,p_1}, \pi_{p'_e,p'_1}), (\pi_{p_e,p_2}, \pi_{p'_e,p'_2}), (\pi_{p_e,p_x}, \pi_{p'_e,p'_x}),$$
$$(\pi_{p_1,p_1}, \pi_{p'_1,p'_1}), (\pi_{p_1,p_4}, \pi_{p'_1,p'_4}), (\pi_{p_2,p_2}, \pi_{p'_2,p'_2}), (\pi_{p_2,p_3}, \pi_{p'_2,p'_3}),$$
$$(\pi_{p_3,p_3}, \pi_{p'_3,p'_3}), (\pi_{p_3,p_4}, \pi_{p'_3,p'_4}), (\pi_{p_4,p_5}, \pi_{p'_4,p'_5}).(\pi_{p_5,p_x}, \pi_{p'_5,p'_x}).$$

Note that the set $\{\pi' \mid \exists\pi.(\pi, \pi') \in \hat{\Pi_1}\}$ consists of all S_1-simple paths. 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 **catch** parts of the command. Call the flow graph of the FSP P_2 and the flow graph of the TDS P'_2. These flow graphs are depicted in Figure 11. We define an assertion function $\hat{I_2}$ of (P_2, P'_2) as follows:

$$\begin{array}{lcl}
\hat{I_2}(a_e, a'_e) & = & \phi_1 \wedge \phi_3 \wedge \phi_4 \\
\hat{I_2}(a_1, a'_1) & = & \phi_1 \wedge \phi_3 \wedge \phi_4 \wedge \mathsf{c} \wedge \mathsf{k} = \mathsf{k}' \\
\hat{I_2}(a_2, a'_2) & = & \phi_1 \wedge \phi_3 \wedge \mathsf{val} = \mathsf{val}' \\
\hat{I_2}(a_x, a'_x) & = & \phi_1 \wedge \phi_2 \wedge \phi_3
\end{array}$$

The function $\hat{I_2}$ is undefined elsewhere. Note that the set $S_2 = \{a'_e, a'_1, a'_2, a'_x\}$ covers P'_1.

We also define a set $\hat{\Pi_2}$ of paths of (P_2, P'_2) such that the set consists of the following paths:

$$(\pi_{a_e,a_1}, \pi_{a'_e,a'_1}), (\pi_{a_1,a_1}, \pi_{a'_1,a'_1}), (\pi_{a_1,a_2}, \pi_{a'_1,a'_2}), (\pi_{a_e,a_2}, \pi_{a'_e,a'_2}), (\pi_{a_1,a_x}, \pi_{a'_1,a'_x}).$$

Note that the set $\{\pi' \mid \exists\pi.(\pi, \pi') \in \hat{\Pi_2}\}$ consists of all S_2-simple paths. One can prove that all assertions in the weak verification condition associated with $\hat{I_2}$ and $\hat{\Pi_2}$ are valid.

Note also that the requirements for the assertion $\hat{I_2}(a_e, a'_e)$ is satisfied by the assertions $\hat{I_1}(p_3, p'_3)$ and $\hat{I_1}(p_5, p'_5)$ and the weak extendibility of $\hat{I_1}$. Therefore, there is an RCR between the command updatePIN of the SPM and of the FSP. □

6 Property Preservation

We have discussed in Section 3 a technique for proving properties of SPMs. The same proof technique is also applicable to proving properties of FSPs and TDSs. Proving properties of SPMs is easier than proving properties of FSPs or TDSs because the language of SPMs is simpler than those of FSPs and TDSs. Let us be given an SPM, an FSP, and a TDS of a smart-card application. Suppose that we have proven that the SPM satisfies a property φ. Suppose further that there are an RCR between the SPM and the FSP and an RCR between the FSP and the TDS. Instead of proving that the FSP and the TDS satisfy φ in the same way as proving that the SPM satisfies φ, we prove that the FSP and the TDS satisfy φ by showing that φ is *preserved* by the RCRs. We discuss in this section how properties are preserved by RCRs, particularly RCRs between SPMs and FSPs; property preservation between FSPs and TDSs can be described similarly.

We are interested in the partial correctness property. 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. We now introduce a notion of partial correctness that also respects abrupt terminations. A program P is partially correct with respect to a precondition φ, a normal postcondition ψ_1 and an abrupt postcondition ψ_2, denoted by $\{\varphi\}P\{\psi_1\}\{\psi_2\}$, if for every run of P from a configuration satisfying φ, if the run reaches a normal exit configuration, then this exit configuration satisfies ψ_1, and if the run reaches an abrupt exit configuration, then this exit configuration satisfies ψ_2. The program P itself can be a command in an SPM, an FSP, or a TDS. In the sequel the former notion of partial correctness is called the standard notion, while the latter notion is called the non-standard notion.

Weakly-extendible assertion functions are sufficient for proving standard partial correctness.

THEOREM 6.1 *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.* □

A proof of the above theorem can be found in [Narasamdya, 2007].

To prove non-standard partial correctness $\{\varphi_1\}P\{\psi_1\}\{\psi_2\}$, we first represent the program P by two programs P_1 and P_2. The program P_1 describes the normal behavior of P, that is in the case of smart-card applications, the behavior of P when no card tears are present. The program P_2 describes the behavior of P when a card tear occurs. Recall that if P is a command in an SPM, then P_1 is the program representing the **pass** and **fail** clauses and P_2 is the program representing the **abrupt** clause. If P is a command in an FSP or a TDS, then P_1 is the program representing the **try** part and P_2 is the program representing the **catch** part.

Second, we introduce an assertion φ_2 as a "linking" assertion between P_1 and P_2, such that the assertion $\varphi_1 \Rightarrow \varphi_2$ is valid. We then prove the following properties:

1. $\{\varphi_1\}P_1\{\psi_1\}$,
2. $\{\varphi_2\}P_2\{\psi_2\}$, and
3. for every run of P from an entry configuration satisfying φ_1, every configuration of the run satisfies φ_2.

By Theorem 6.1, weakly-extendible assertion functions are sufficient for proving properties 1 and 2. As shown in Example 3.1 and Example 3.2, we can use the notions of (weakly or strongly) extendible assertion function and verification condition associated with the function to prove property 3.

THEOREM 6.2 *Let P and P' be programs representing, respectively, the SPM and the FSP of a command C. Let Obs be a one-to-one correspondence between the set O_{SPM} of observable variables of the SPM and the set O_{FSP} of observable variables of the FSP. Let φ, ψ_1, ψ_2 be assertions consisting only of variables of the SPM and $\varphi', \psi'_1, \psi'_2$ be assertions consisting only of variables of the FSP such that the following*

assertions are valid:

$$\bigwedge_{x \in O_{SPM}} x = Obs(x) \Rightarrow (\varphi \Leftrightarrow \varphi')$$
$$\bigwedge_{x \in O_{SPM}} x = Obs(x) \Rightarrow (\psi_1 \Leftrightarrow \psi_1')$$

$$\bigwedge_{x \in Ab(O_{SPM})} x = Obs(x) \Rightarrow (\psi_2 \Leftrightarrow \psi_2').$$

If P is partially correct with respect to the precondition φ, normal postcondition ψ_1, and abrupt postcondition ψ_2, or $\{\varphi\}P\{\psi_1\}\{\psi_2\}$ and there is an RCR between the SPM and the FSP of the command C, then $\{\varphi'\}P'\{\psi_1'\}\{\psi_2'\}$.

PROOF. Consider a run of P' from a state σ_1' satisfying φ' such that the run reaches a normal exit configuration with state σ_2'. Since there is an RCR between P and P', there is a run of P from a state σ_1 such that (σ_1, σ_1') satisfies $\bigwedge_{x \in O_{SPM}} x = Obs(x)$, and thus σ_1 satisfies φ. By the RCR, the run of P' also reaches a normal exit configuration with state σ_2. Since $\{\varphi\}P\{\psi_1\}\{\psi_2\}$, the state σ_2 satisfies ψ_1. It follows from the validity of assertion $\bigwedge_{x \in O_{SPM}} x = Obs(x) \Rightarrow (\psi_1 \Leftrightarrow \psi_1')$ that σ_2' satisfies ψ_1'.

Suppose that the run of P' reaches an abrupt exit configuration with a state σ_3'. Then by the RCR there is a run of P from a state σ_1 such that (σ_1, σ_1') satisfies $\bigwedge_{x \in O_{SPM}} x = Obs(x)$, and thus σ_1 satisfies φ, and the run reaches an abrupt exit configuration with state σ_3. Since $\{\varphi\}P\{\psi_1\}\{\psi_2\}$, the state σ_3 satisfies ψ_2. It follows from the validity of assertion $\bigwedge_{x \in Ab(O_{SPM})} x = Obs(x) \Rightarrow (\psi_2 \Leftrightarrow \psi_2')$ that σ_3' satisfies ψ_2'. □

Properties of SPMs or FSPs and RCRs between SPMs and FSPs are often described by assertion functions. Let P_1 and P_2 be programs representing, respectively, the **pass** and **fail** clauses and the **abrupt** clause of a command C in an SPM. Let P_1' and P_2' be programs representing, respectively,the **try** and **catch** parts of the command C in an FSP. Let I_1 and I_2 be assertion functions of, respectively, P_1 and P_2, such that

$$\begin{array}{lcl} I_1(entry(P_1)) & = & \varphi_1 \\ I_1(exit(P_1)) & = & \psi_1 \\ I_2(entry(P_2)) & = & \varphi_2 \\ I_2(exit(P_2)) & = & \psi_2 \end{array}$$

for some assertions $\varphi_1, \varphi_2, \psi_1, \psi_2$, and $\{\varphi_1\}P\{\psi_1\}\{\psi_2\}$. The same property for the FSP is represented by assertion functions I_1' and I_2' of, respectively, P_1' and P_2', such that

$$\begin{array}{lcl} I_1'(entry(P_1')) & = & \varphi_1' \\ I_1'(exit(P_1')) & = & \psi_1' \\ I_2'(entry(P_2')) & = & \varphi_2' \\ I_2'(exit(P_2')) & = & \psi_2' \end{array}$$

for some assertions $\varphi_1, \varphi_2, \psi_1, \psi_2$.

Let the RCR between the SPM and the FSP of the command C is represented by assertion functions $\hat{I}_1$ and $\hat{I}_2$ of, respectively, (P_1, P_1') and (P_2, P_2'), such that

$$\begin{array}{lcl} \hat{I}_1(entry(P_1), entry(P_1')) & = & \alpha \\ \hat{I}_1(exit(P_1), exit(P_1')) & = & \alpha' \\ \hat{I}_2(entry(P_2), entry(P_2')) & = & \beta \\ \hat{I}_1(exit(P_2), exit(P_2')) & = & \beta' \end{array}$$

for some assertions $\alpha, \alpha', \beta, \beta'$. To prove that $\{\varphi_1'\}P'\{\psi_1'\}\{\psi_2'\}$, as the above theorem has shown, we have to prove that the following assertions are valid:

$$\alpha \Rightarrow (\varphi_1 \Leftrightarrow \varphi_1')$$
$$\alpha' \Rightarrow (\psi_1 \Leftrightarrow \psi_1')$$
$$\beta' \Rightarrow (\psi_2 \Leftrightarrow \psi_2').$$

In addition we have to show that if a card tear occurs during a run of P_1', then the final state of the run satisfies $I_2'(entry(P_2'))$. To this end, we need to prove that the assertion $\beta \Rightarrow (\varphi_2 \Leftrightarrow \varphi_2')$ is valid. Thus, if

a card tear occurs during a run of P_1' such that the final state of the run is σ', then by the RCR there is a finite run of P such that the final state of the run is σ and (σ, σ') satisfies $I_2'(entry(P_2'))$. Since $\{\varphi_1\}P\{\psi_1\}\{\psi_2\}$, we have σ satisfies φ_2. Now, since $I_2'(entry(P_2')) = \beta$ and the assertion $\beta \Rightarrow (\varphi_2 \Leftrightarrow \varphi_2')$ is valid, it follows that σ' satisfies φ_2', and thus $\{\varphi_1'\}P'\{\psi_1'\}\{\psi_2'\}$.

EXAMPLE 6.3 In this example we consider the property that we have proven for the SPM in Example 3.1. That is, in any run of checkPIN the validation status at the exit configuration of the run is true if and only if the run emits a Pass event. This property is a partial correctness property of the SPM. We want to show that this property is preserved by the RCR described in Example 4.4. That is, the FSP shown on the righthand side of Figure 4 and Figure 5 satisfies the property.

We assume that all variables in the FSP are in primed notation. To describe the property we define two assertion functions I_1' and I_2' such that

$$\begin{array}{lcl} I_1'(p_e') & = & \varphi' \\ I_1'(p_x') & = & \varphi' \wedge \mathsf{val}' = \top \Leftrightarrow \varepsilon = \mathsf{Pass} \\ \\ I_2'(a_e') & = & \top \\ I_2'(a_x') & = & \mathsf{val}' = \top \Leftrightarrow \varepsilon = \mathsf{Pass}, \end{array}$$

where φ' is the conjunction of the assertions $\mathsf{MAX}' > 0$, $0 \leq \mathsf{trial}' \leq \mathsf{MAX}'$, and $\mathsf{trial}' < \mathsf{MAX}' \Rightarrow \mathsf{val}' = \bot$. The function I_1' and I_2' can be defined elsewhere.

Consider the assertion functions I_1 and I_2 defined in Example 3.1, and the assertion functions $\hat{I}_1$ and $\hat{I}_2$ defined in Example 4.4. Since the following assertions are valid:

$$\begin{array}{l} \hat{I}_1(p_e, p_e') \Rightarrow (I_1(p_e) \Leftrightarrow I_1'(p_e')) \\ \hat{I}_1(p_x, p_x') \Rightarrow (I_1(p_x) \Leftrightarrow I_1'(p_x')) \\ \hat{I}_2(p_e, p_e') \Rightarrow (I_2(p_e) \Leftrightarrow I_1'(p_e')) \\ \hat{I}_2(p_x, p_x') \Rightarrow (I_2(p_x) \Leftrightarrow I_1'(p_x')) \end{array}$$

it follows that the FSP of the command checkPIN is partially correct with respect to the precondition $I_1'(p_e')$, the normal postcondition $I_1'(p_x')$, and the abrupt postcondition $I_2'(p_x')$. □

In this section we have shown how RCRs between SPMs and FSPs preserve properties of SPMs. Property preservation between FSPs and TDSs can be described similarly.

7 Related Work

There have been some works related to the specification and verification of smart-card applications and to CC certification. For example, the work in [Breunesse et al., 2005] describes a case study in the specification and verification of an electronic purse application. The work is not in the framework of CC and only concerned with the specification and verification of a single program, which is the implementation code. The work can complement our work in proving properties of the implementation code.

An example work on CC certification is [Heitmeyer et al., 2006]. The work is concerned with verifying that the kernel of a software-based embedded device enforces data separation. Similar to our SPMs, the specification is modelled as a finite state machine. The RCR in this work is only between the state machine and the implementation code, and also is a standard refinement relation.

8 Conclusion

We have successfully applied the theory of program properties described in [Narasamdya, 2007] to the certification of smart-card applications in the framework of Common Criteria. In the application of the theory to proving properties of SPMs, FSPs, and TDSs, we prove the properties of each command separately. Each command is represented by two flow graphs or programs, one program describes the normal behavior of

the command and the other program describes what the command has to do when a card tear occurs. Properties that we want to prove are encoded as assertion functions of these two programs. Weakly-extendible assertion functions are then sufficient to prove the properties.

In the application of the theory to proving RCRs, we prove that there is an RCR between each corresponding commands separately. Each corresponding programs representing the normal behavior and card tears are considered as two pairs of programs. The RCR itself is then encoded as assertion functions of these pairs of programs. The notions of weakly-extendible assertion function and weak verification condition are used to prove the RCR and to provide a certificate about the RCR. Particularly for RCRs between FSPs and TDSs, the application of the theory to proving such RCRs also handles memory characteristics and transaction mechanism that exist in the low-level design, or the TDSs.

We have also shown that, using the theory, the properties that are satisfied by a requirement representation of an application can be brought forward to the subsequent requirement representations, and so there is no need to prove that the same properties are satisfied by the subsequent representations.

References

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

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

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. 7

Z. Chen. *Java Card Technology for Smart Cards*. The Java Series. Addison-Wesley, 2000. 5

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

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. 7

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

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. 5

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. 5, 5.1

G. Leavens and Y. Cheon. Design by contract with jml, 2003. 3.3

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`. 1, 6, 8

Java Card 3.0 Platform Specification. Sun Micro systems, Inc, Palo Alto, California, 2008. `http://java.sun.com/javacard/3.0/`. 5

Andrei Voronkov and Iman Narasamdya. Proving inter-program properties. Technical Report TR-2008-13, Verimag, September 2008. 1