Certification of Smart-Card Applications in Common Criteria *

Iman Narasamdya
Verimag - Université de Grenoble I
2, avenue de Vignate
Gieres, France
Iman.Narasamdya@imag.fr

Michaël Périn
Verimag - Université de Grenoble I
2, avenue de Vignate
Gieres, France
Michael.Perin@imag.fr

ABSTRACT

This paper describes the certification of smart-card applications in the framework of Common Criteria. In this framework, a smart-card application is represented 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. All representations of a smart-card application are essentially programs and the representation correspondences are properties relating two programs. We show that a theory of program properties can be applied 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 certificate that is essential to the certification process.

Categories and Subject Descriptors

F.3.1 [**Logics and Meaning of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms

Assertions, Invariants, Specification techniques

Keywords

Assertion Functions, Invariants, Common Criteria Certification, Smart-Card Applications

*This work is supported by EDEN2 ANR project

SAC'09 March 8-12, 2009, Honolulu, Hawaii, U.S.A.

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-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 paper 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 [1], and (2) to provide a certificate or a collection of certificates showing that a smart-card application follows its specification or a model of its specification.

Common Criteria (CC) is an international standard for the evaluation of security related systems. CC defines requirements for certification: *security policy model* (SPM), *functional specification* (FSP), *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. 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 paper we consider only the requirements SPM, FSP, and TDS.

We apply the theory of intra-program and inter-program properties described in [10] to the 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 [3, 5] can be applied to prove 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.

The contribution of this paper is the application of the above theory in 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, mapping between variables in RCRs can be nontrivial. For example, a scalar variable in the SPM can correspond to an array variable in the FSP. The low-level design of the application can include a transaction mechanism and memory characteristics that are specific to smart-card applications. One then has to model these features to apply the theory to the certification process.

Due to space limitation, in this paper we are only concerned with proving properties of the SPM and proving RCRs between the SPM and the FSP. In our technical report [8] we extend our discussion in this paper further to proving RCRs between the FSP and the TDS, and proving property preservation from the SPM to the FSP and the TDS.

The outline of this paper 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 [10]. We then apply the theory to proving properties of SPMs. Afterward, we apply the theory to proving RCRs between the SPM and the FSPs. We finally conclude this paper and briefly discuss some related work.

2. PROVING 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*. 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 describe the run-time behavior of a program as sequences of configurations. A configuration is a pair (p, σ), where p is a program point and σ is a state mapping variables to values. 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, σ_1) and (p_2, σ_2) are configurations and (p_1, p_2) is an edge in $\mathbf{G}_P$. A *computation sequence of a program* P is either a finite or an infinite sequence of configurations $(p_0, \sigma_0), (p_1, \sigma_1), \ldots$, 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 $(p_0, \sigma_0), (p_1, \sigma_1), \ldots$, such that $p_0 = entry(P)$.

We introduce two restrictions on the semantics of programs. First, we assume that programs are deterministic. 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. We also 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 $\gamma_1 \mapsto_P \gamma_2$.

Further, we assume some *assertion language* in which one can write *assertions* involving variables and express properties of states. We write $\sigma \models \alpha$ to mean an assertion α is true in a state σ, or σ *satisfies* α, or 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 $\sigma \models \alpha$. The assertion language is closed under the standard propositional connectives and respects their semantics

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 is defined on $entry(P)$ and $exit(P)$. 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 2.1 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 2.2 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 2.3 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 2.4 *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 2.5 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 2.6 *Let I and Π be as in Definition 2.5 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', σ, σ'), where $p \in \mathbf{Point}_P$, $p' \in \mathbf{Point}_{P'}$, and σ and σ' are states for P and P', respectively.

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 $(exit(P), exit(P'))$.

Unlike in the case of a single program, for a pair of programs, there are no notions of invariant and strong extendibility. The notion of weakly-extendible assertion function is better suited for describing inter-program properties. For a pair of programs (P, P'), the definition of weakly-extendible assertion function of (P, P') is similar to Definition 2.1. The only difference is, for a pair (P, P') of programs, the assertion function I in Definition 2.1 is an assertion function of (P, P').

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 π' 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.

The definition of weak verification condition for the case of a pair of programs is similar to Definition 2.5. The only differences are, for a pair (P, P') of programs, the assertion function I in Definition 2.5 is an assertion function of (P, P') and the set Π in Definition 2.5 is a set of non-trivial paths of (P, P'). Moreover, Theorem 2.6 about weak verification conditions still holds for the case of a pair of programs. The notion of weak verification condition forms a suitable notion of certificate about properties involving two programs.

3. PROVING PROPERTIES OF SECURITY POLICY MODELS

3.1 Life Cycle of Smart-Card Applications

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

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

In EDEN2 an SPM is written in a so-called *command description language*. An SPM consists of commands that will be implemented in the smart-card application. Each command in the SPM has the following form:

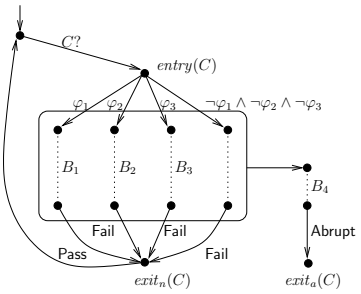

Figure 1: Semantics of SPM.

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. 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 clauses are statements written in a simple imperative language. The semantics of the command is shown in Figure 1. 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. A run of a command C from a state σ is a computation sequence starting from the configuration $(entry(C), \sigma)$. A run of a command C terminates normally if it reaches $exit_n(C)$. For a such termination, the run emits either a Pass or a Fail event. When, a card tear occurs, the run terminates abruptly and emits an Abrupt event.

As shown in Figure 1, an SPM itself can be considered as a program that takes as an input a sequence of commands. A run of an SPM is a finite or infinite alternating sequence $\gamma_0, \varepsilon_1, \gamma_2, \varepsilon_2, \ldots$, where (1) γ_0 is an entry configuration, (2) $\gamma_i \mapsto \gamma_{i+1}$ for all $i \geq 0$, and (3) 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.

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 example.

EXAMPLE 3.1 We consider a command checkPIN used to authenticate users by verifying the input PIN. The flow graphs representing the command are depicted in Figure 2. The lefthand flow graph P_1 is for the **pass** and **fail** clauses, while the righthand flow graph P_2 is for the **abrupt** clause. The variables pin, p, MAX, and trial are of integral type, while the variable val is of boolean type. The variable pin is the PIN stored in the card and the variable p is 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.

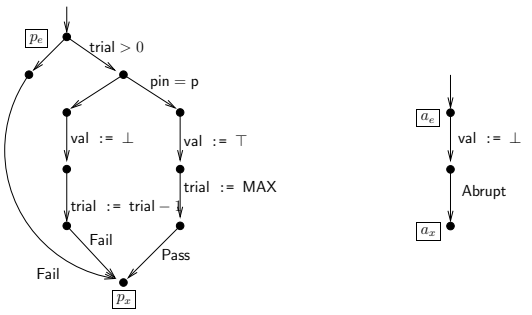

Figure 2: An SPM of checkPIN.

The property that we want to prove is as follows: "In any run of checkPIN, the value of variable val at the exit configuration of the run is true *if and only if* the execution emits a Pass event."

To prove this property, we need to prove that, for any run of the command, the following sub-property holds at the entry and normal exit configurations of the run: If the PIN is blocked, that is the value of trial is equal to 0, then the value of val is false.

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

$$I_1(p_e) = \varphi, \qquad I_1(p_x) = \varphi \wedge \mathsf{val} = \top \Leftrightarrow \varepsilon = \mathsf{Pass}$$
$$I_2(a_e) = \top, \qquad I_2(a_x) = \mathsf{val} = \top \Leftrightarrow \varepsilon = \mathsf{Pass},$$

where the assertion φ is defined as $\varphi \Leftrightarrow (\mathsf{MAX} > 0 \wedge 0 \leq \mathsf{trial} \leq \mathsf{MAX} \wedge (\mathsf{trial} < \mathsf{MAX} \Rightarrow \mathsf{val} = \bot))$. The last conjunct above generalize the sub-property that we want to prove. We also use a special variable ε to store emitted events.

Next, 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 if I_1 and I_2 are weakly extendible, then the properties that we want to prove hold. Consider a run R of the command from a state σ. We concern only with the run R emitting Fail. If σ satisfies $I_1(p_e)$, $\mathsf{trial} > 0$, and $\mathsf{pin} \neq \mathsf{p}$, then when R terminates normally with states σ', then σ' satisfies $I_1(p_x)$. Particularly, the state σ' satisfies $0 \leq \mathsf{trial} \leq \mathsf{MAX}$ because 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. If the state σ satisfies $I_1(p_e) \wedge \mathsf{trial} < 0$, then by the assertion φ, we have σ satisfies $\mathsf{val} = \bot$. Moreover, R will not modify any variables, and thus $I_1(p_x)$ holds at the state σ'. One can prove that I_1 and I_2 are strongly extendible easily, and thus I_1 and I_2 are weakly extendible.

To prove the above properties 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. □

4. PROVING REPRESENTATION CORRESPONDENCES

In EDEN2 an FSP is essentially a Java program written in a subset of Java. Each command in an 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 event, while returning a response status that indicates error or 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.

Like an SPM, an FSP 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. For each execution of the command, if the execution exits from the **try** part, then the execution fetches the next input $C(a_1, \ldots, a_n)$ from the input sequence. If a card tear occurs, then the execution exits from the **catch** part or terminates abruptly, and in turn the execution of the FSP simply terminates. A run of an FSP is defined in the same way as a run of an SPM. We also prove properties of an FSP similarly to proving properties of an SPM. First, each command in an FSP is represented by two 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. Let E be a set of observable events. Denote by $R|_E$ the subsequence of R consisting only of events in E:

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

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 4.1 Let $E_O = \{\mathsf{Pass}, \mathsf{Fail}, \mathsf{Abrupt}\}$ be the set of observable events. Let O_{SPM} and O_{FSP} be the sets of observable variables of, respectively, the SPM and the FSP such that there is a one-to-one correspondence Obs between O_{SPM} and O_{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,

1. $\varepsilon_{i_k} = \varepsilon'_{j_k}$,
2. if $\varepsilon_{i_k} \neq \mathsf{Abrupt}$, then $\sigma_{i_k}(x) = \sigma'_{j_k}(Obs(x))$ for all $x \in O_{SPM}$,

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

□

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

1. R is terminating if and only if so is R',
2. 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 $\varepsilon \neq \mathsf{Abrupt}$, then the states σ_2 and σ'_2 satisfy $\bigwedge_{x \in O_{SPM}} x = Obs(x)$;
   - otherwise the states σ_2 and σ'_2 satisfy $x = Obs(x)$ for all $x \in Ab(O_{SPM})$.

Let α be an assertion such that α implies $\bigwedge_{x \in O_{SPM}} x = Obs(x)$. 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

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

The function $\hat{I}_1$ 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 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:

1. the assertion $\alpha \Rightarrow \psi$ is valid, and
2. 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 $\hat{\Pi}_1$.

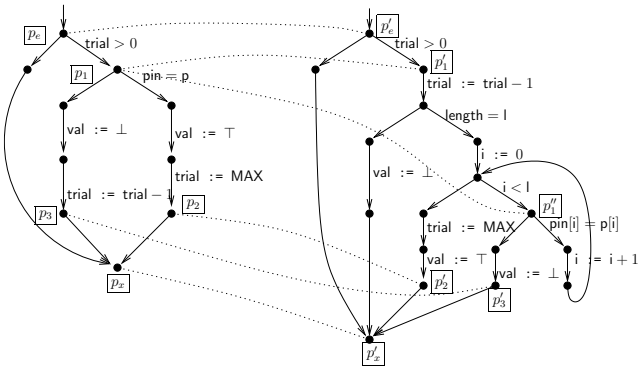

Figure 3: P_1 is on the left and P'_1 is on the right.

THEOREM 4.2 *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.*

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 4.3 In this example we will show that there is an RCR between the SPM and the FSP of the command checkPIN. The flow graph P_1 representing the **pass** and **fail** clauses and the flow graph P'_1 representing the **try** part are depicted in Figure 3. 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, and so we have to define the equality 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 a pair 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 $p[i] = p'[i]$. 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, (1) $x \sim y \Rightarrow (y \equiv z \Leftrightarrow x \sim z)$ and (2) $x \sim y \Rightarrow (w = x \Leftrightarrow w \sim y)$. 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:

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

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 construct a set $\hat{\Pi_1}$ of paths of (P_1, P_1') as follows: for each S_1-simple path $\pi_{p',q'}$, we only pair $\pi_{p',q'}$ with a path π in $\hat{\Pi_1}$, that is $(\pi, \pi_{p',q'}) \in \hat{\Pi_1}$, such that if π is nontrivial, that is $\pi = \pi_{p,q}$, then $\hat{I}_1(p, p')$ and $\hat{I}_1(q, q')$ are defined, or if π is trivial, that is $\pi = \pi_p$, then $\hat{I}_1(p, p')$ and $\hat{I}_1(p, q')$ are defined. 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 graph P_2 of the **abrupt** clause and the flow graph P_2' of the **catch** part. For simplicity in this example, the flow graph P_2' is identical to the flow graph P_2 depicted on the righthand side of Figure 2.

We define an assertion function $\hat{I}_2$ of (P_2, P_2') such that $\hat{I}_2(entry(P_2), entry(P_2')) = \top$ and $\hat{I}_2(exit_a(P_2), exit_a(P_2')) = \mathsf{val} = \mathsf{val}'$. The set $S_2 = \{entry(P_2'), exit_a(P_2')\}$ 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. The set $\hat{\Pi_2}$ consists only of a pair of paths from the entry points to the exit points. 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 4.2 there is an RCR between the command checkPIN of the SPM and of the FSP.

□

The proof technique for proving RCRs between an SPM and an FSP is also applicable to proving RCRs between an FSP and a TDS. The latter is challenging due to features introduce by the language of TDSs. A TDS is written in a subset of Java Card [9]. This subset includes the memory characteristics and transaction mechanism of Java Card. 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 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 also features the transaction mechanism of Java Card. Transactions are managed by methods beginTransaction, commitTransaction, and abortTransaction with standard functionalities. The updates of persistent variables are conditional when a transaction is in progress. That is, the updates are materialized if commitTransaction is called. The updates of transient variables are always unconditional regardless a transaction is in progress or not.

We use the desugaring method described in [6] to model card tears and transactions. Similar to the FSP, each command in the TDS is a Java method. Desugaring the command translates the method into the same form as that of the FSP, that is, the method has a big **try-catch** construct. The **catch** construct sets all transient variables to their default values, and cancel the updates of persistent variables if a card tear occurs when a transaction is in progress. The desugaring method introduces fresh global variables that are used to back up persistent variables before a transaction begins. In the case that the transaction is aborted or a card tear occurs, then using the back-up variables, the values of the persistent variables are rolled back to the values before the transaction begins. The desugaring method also introduces a fresh boolean variable inTransaction that keeps track whether a transaction is in progress or not. Let us consider the following TDS of a toy command:

```
t = 4;
beginTransaction();
x = 5;
t = 6;
y = 7;
endTransaction();
```

```
try {
  t = 4;
  if (inTransaction)
    return IN_PROGRESS
  xb = x; yb = y;
  inTransaction = true;
  x = 5; t = 6;
  y = 7;
  inTransaction = false;
} catch (CardTearException e) {
  if (inTransaction) {
    x = xb; y = yb; }
  t = 0;
}
endTransaction();
```

The lower program is obtained by desugaring the upper program. The variables xb and yb are back-up variables for the persistent variables x and y, and t is a transient variable. Similar to the FSP, we then have two flow graphs, one for the **try** part and the other for the **catch** part. 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 [9]. Discussion on Java Card non-atomic API methods and their effects on transactions can be found in [7].

To prove RCRs between the FSP and the TDS, we consider the pair of flow graphs of the **try** parts of the FSP and of the TDS, and the pair of flow graphs of the **catch** parts of the FSP and of the TDS. Suppose that the FSP of the

above command is as follows:

```
try {
  yp = 7;
  xp = 5;
  tp = 6;
} catch (CardTearException e) {
  tp = 0;
}
```

Suppose that the variables $\mathsf{x}, \mathsf{y}, \mathsf{t}$ in the TDS are observable variables that correspond to the variables $\mathsf{xp}, \mathsf{yptp}$, respectively. We want to prove that this correspondence holds even when a card tear occurs. To this end, we have to assert at the entries of the flow graphs of the **catch** parts the following assertion:

$$(\neg \mathsf{inTransaction} \Rightarrow \mathsf{x} = \mathsf{xp} \wedge \mathsf{y} = \mathsf{yp}) \\ \wedge (\mathsf{inTransaction} \Rightarrow \mathsf{xp} = \mathsf{xb} \wedge \mathsf{yp} = \mathsf{yb})$$

To prove that every finite run of the TDS, there is finite run of the FSP such that the above assertion holds, we need to associate events with the updates of observable persistent variables and use event variables that keep track the occurrences of these events. In particular, during a transaction the order of independent updates, such as the updates of x and y is irrelevant. So, an event variable that keeps track events during the transaction has to collect a set of events instead of a sequence of events.

Due to lack of space, detailed discussion on proving RCRs between the FSP and the TDS can be found in our technical report [8].

5. CONCLUSIONS

We have successfully applied the theory of program properties described in [10] to the certification of smart-card applications in the framework of Common Criteria. The application of the theory also handles memory characteristics and transaction mechanism that exist in the low-level design.

There have been some works related to the specification and verification of smart-card applications and to CC certification. For example, the work in [2] 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 [4]. 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.

6. REFERENCES

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

[2] 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.

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

[4] 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.

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

[6] E.-M.G.M. Hubbers and E. Poll. Reasoning about card tears and transactions in Java Card. In M. Wermelinger and T. Margaria-Steffen, editors, *Fundamental Approaches to Software Engineering, 7th International Conference, FASE 2004*, volume 2984 of *LNCS*, pages 114–128. Springer-Verlag, 2004.

[7] 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 2004.

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

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

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