Using B Method to Formalize the Java Card Runtime Security Policy for a Common Criteria Evaluation

Stéphanie Motré - Corinne Téri

stephanie.motre@gemplus.com - corinne.teri@gemplus.com

Gemplus - Avenue du Pic de Bertagne
13881 Gemenos – France
fax: (3) 442-36-64-04

Type of submission: Paper

Abstract. A smart card is an embedded system that is generally used to supply security to an information system. Traditionally the application and the OS were developed in a secure environment by the card issuer. For a few years, open platforms (e.g., Java Card, MultOS and Smart Card for Windows) have provided new facilities for application developers. They allow dynamic storage and execution of downloaded executable code. Such architecture introduces new risks: it offers the possibility to attack the card from an applet by exploiting some implementation faults. This document provides an overview of a set of techniques required to obtain Common Criteria (CC) high Evaluation Assurance Levels (EALs) of a Java Card. It is not dedicated to smart card specialists as it presents the security stakes of such a technology. We present the motivation for a Java Card evaluation: reach the same security level for the new open smart card than for traditional embedded platforms. We introduce the UML and the B method to illustrate the semi-formal and formal models required for a high level evaluation. The B method has been already used in GEMPLUS to formally model security mechanisms of the Java Card: bytecode verifier, interpreter and firewall. These case studies reveal the interest of using the B method to formalize the Java Card Virtual Machine (JCVM). In a CC evaluation the use of semi-formal and formal techniques is required to obtain the assurance of a high security level.

Key words: B, Common Criteria (CC), Formal Method (FM), Java Card (JC), Security policy, UML, Visa Open Platform (VOP).

Using B Method to Formalize the Java Card Runtime Security Policy for a Common Criteria Evaluation

Abstract. A smart card is an embedded system that is generally used to supply security to an information system. Traditionally the application and the OS were developed in a secure environment by the card issuer. For a few years, open platforms (e.g., Java Card, MultOS and Smart Card for Windows) have provided new facilities for application developers. They allow dynamic storage and execution of downloaded executable code. Such architecture introduces new risks: it offers the possibility to attack the card from an applet by exploiting some implementation faults. This document provides an overview of a set of techniques required to obtain Common Criteria (CC) high Evaluation Assurance Levels (EALs) of a Java Card. It is not dedicated to smart card specialists as it presents the security stakes of such a technology. We present the motivation for a Java Card evaluation: reach the same security level for the new open smart card than for traditional embedded platforms. We introduce the UML and the B method to illustrate the semi-formal and formal models required for a high level evaluation. The B method we have already used to formally model security mechanisms of the Java Card: bytecode verifier, interpreter and firewall. These case studies reveal the interest of using the B method to formalize the Java Card Virtual Machine (JCVM). In a CC evaluation the use of semi-formal and formal techniques is required to obtain the assurance of a high security level.

Key words: B, Common Criteria (CC), Formal Method (FM), Java Card (JC), Security policy, UML, Visa Open Platform (VOP).

1 Introduction

A smart card is an embedded system that is generally used to supply security to an information system. Open smart cards, like the Java Card, introduce new risks: it offers the possibility to attack the card from an applet by exploiting some implementation faults. Actually, every product and more particularly Smart Cards Integrated Circuits (IC) with Open Platform should prove their robustness in order to be certified. The Common Criteria (CC) responds to strict security requirements like French banking requirements presented in the Vocable project [13]. Thanks to Java Card platform specification assurance, every cardholder can securely load and run any application on its card.

This document provides a practical overview of a CC high Evaluation Assurance Level (EAL) of a Java Card. This paper provides a set of techniques to specify a security policy in Common Criteria evaluation for an EAL4 with informal and semi-formal models as states graph (part 4.1), for an EAL5 with formal model and B formal language (part 4.2) and for an EAL7 (part 4.3).

In the first section of the paper, we make a brief summary of the Common Criteria evaluation. In the second section, we present the scope of the Java Card Runtime specifications. Finally, in the third part, we take a security policy case for EAL4, EAL5 and EAL7.

2 Product evaluation with the Common Criteria

The CC [1][3] combines the best aspects of existing European (ITSEC), US (TCSEC) and Canadian (CTCPEC) criteria for the security evaluation of **Information Technology** (IT) systems and products. The CC documentation [3] introduces requirements for the IT security of a product or system under the distinct categories of **functional requirements** and **assurance requirements**. Functional requirements define expected security behavior: they describe security functionalities that could be implemented by the product. Assurance requirements are a solution to gain confidence that the claimed security measures are effective and correctly implemented.

The CC defines a set of IT requirements of known validity, which can be used to establish security requirements for prospective products and systems. The **Target of Evaluation** (TOE) is the part of the product or system that is subject to evaluation (figure 1). The **Security Target** (ST) that is used by the evaluators as the evaluation basis, describes the threats, the security objectives and the requirements (functional and assurance measures) of a specific identified TOE. The **TOE Security Functions** (TSF) are the components that enforce security in the considered system: the **TOE Security Policy** (TSP).

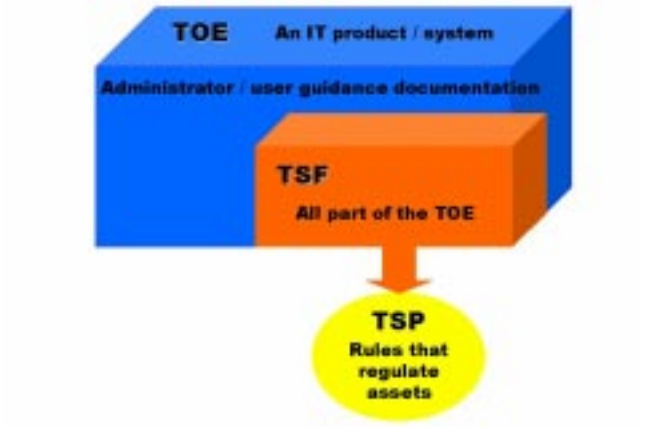


Fig.1 CC evaluation target

The expected result of the evaluation process is a confirmation that the TOE satisfies the ST. Confidence in IT security can be gained through actions that may be taken during the process of development, evaluation and operation.

The definition of the CC security requirements [1] is based on the threats identification against the IT environment. The CC defines a set of constructs, which classify these security requirements into related sets called components: **functional components** and **assurance components**. The functional components are described in the Part 2 of the Common Criteria documentation, which can be compared to a requirement

catalogue. These components are used to express a wide range of security functional requirements within **Protection Profiles** (PPs) and STs. They are ordered sets of functional elements; these sets are grouped into families with common objectives and classes with common intent. A hierarchy may exist between components. The assurance components are presented in the Part 3 of the CC.

The CC defines a set of defined **assurance levels** constructed using components from the assurance families. These levels are intended to provide backward compatibility to source criteria and to provide internally consistent general-purpose assurance packages. To meet specific objectives, an assurance level can be augmented by one or more additional components.

EALs are constructed from the assurance components. Every assurance family contributes to the assurance that a TOE meets its security claims. EALs provide a uniformly increasing scale that balances the level of assurance obtained with the cost and feasibility to acquire this assurance degree. There are seven hierarchically higher assurance components from the same assurance family, and by the addition of assurance components from other assurance families. Figure 2 illustrates the evaluation procedure based on the CC documentation.

Fig.2 Common Criteria components architecture

A TOE evaluation [3] is an assessment of an IT product or system against defined criteria. Distinct stages of evaluation are identified, corresponding to the principal layers of TOE representation: **PP**, **ST**, **TOE** and **Assurance maintenance evaluations**.

3 Java Card Runtime Environment

3.1 Security Issues

Security has always been a great concern for smart cards, but the issue is getting more important with multi-application platforms and post issuance code downloading. The security of a platform is built on the operating system security, which must provide reliable services. This is of special concern for the cryptographic primitives that must be safe regarding all the state-of-the-art attacks such as timing attacks, power analysis, etc. Memory management is also a critical issue and must be safe and robust against card power-loss or memory failure. The last security issue for such smart cards concerns the application level and information flows analysis.

3.2 Java Card Operating Systems

3.2.1 The Java Card Architecture

Java Card is defined by the standards as a smart card capable of running Java programs called card applets. But the **Java Card Virtual Machine** (JCVM) architecture is not limited to a runtime area: programs must be compiled, verified, converted, loaded, and linked before expecting to be executed. Figure 3 presents the JCVM architecture.

The bytecode *verifier* is the offensive security process of the JCVM. It performs the static code verifications required by the JVM specification. The *verifier* guarantees the validity of the code being loaded in the card. The bytecode *converter* transforms the Java class files, which has been verified and validated, into a form that is more suitable for smart cards. The JCVM *loader* is split in two parts: an *off-card* loader that sends the file to the card, and an *on-card* part that installs the classes into the card memory. The conversion and the loading are not executed consecutively (a lot of time can separate them).

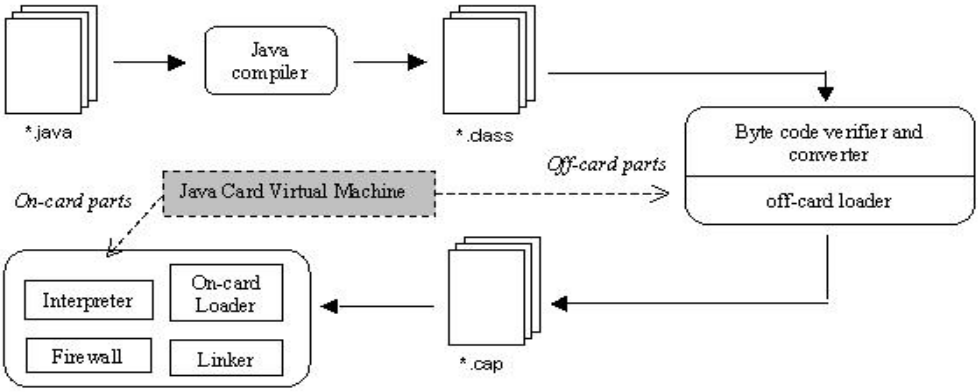


Fig.3 Java Card Virtual Machine

Actually, it is not specified that a package is immediately installed in the card after its conversion. Thus, it may be possible to corrupt it, intentionally or not, during this transition period. To avoid it, the Visa Open Platform checks the integrity and authenticates the package before its registration in the card. Actually, the package contains a signature that identifies its issuer and its contents.

The **Java Card Runtime Environment** (JCRE) considered in our study is conform to Sun Microsystems Inc. [6] specification. Its main elements are presented in figure 4.

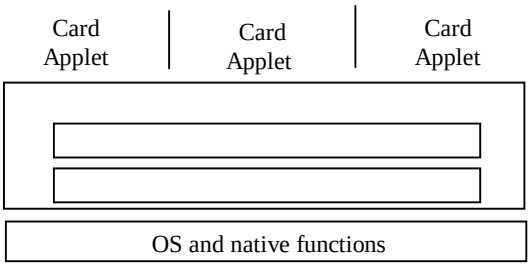


Fig.4 JCRE structure

This specification provides the basis of the JCVM implementation: the JCRE contains the **Java Card Application Programmer Interface** (JCAPI) classes [7] and our own specific extensions, the support services, and the JCVM. The JCRE also has its particular dynamic security process: the *firewall*. This feature is due to a specific JCVM requirement, the on-card object access policy. The *firewall* creates a secure environment, controlling every information access between applets. Every object (class instance or array) on the card is owned by the applet which instantiated it, that is, the applet which was active at the time the object was created. An applet has full rights to access its objects, but the *firewall* still verifies that an applet does not try to illegally access information. The rules used by the *firewall* describe the access policy that is enforced at runtime.

3.2.2 The Java Card specificity

Most of the differences existing between a JVM and the JCVM are due to on-card limited resources. The Java Card language is a subset of Java: applets written for the Java Card platform are written in the Java programming language. They are compiled like any Java applet, but they use a subset of the Java language. The number of enabled programming instructions is reduced to fit the Java Card memory capacities without penalizing too much Java programming possibilities. The subset is a compromise between on-card resources and performances. On card, the JVM is not the same as for any Java platform: there are no security manager,

no dynamic class loading (classes are burnt in memory or loaded within packages), no multiple threading, and no garbage collector[1].

4 Security Policy of Java Card Runtime

4.1 Security policy in EAL4

4.1.1 EAL4 : methodically designed, tested and reviewed

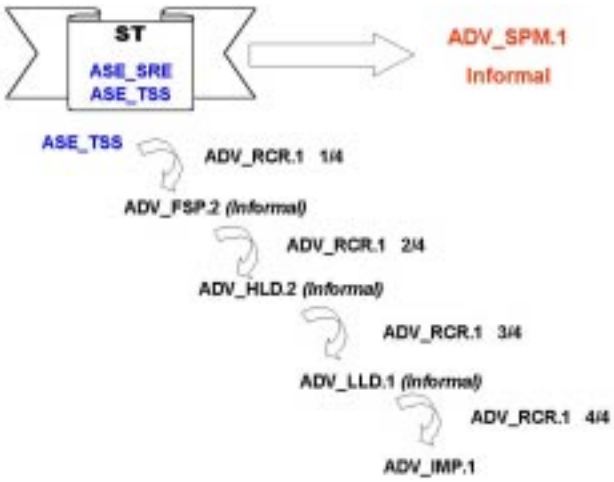


Fig.5 EAL4 description

An **EAL4 evaluation** [1] corresponds to the highest assurance level that does not require any formal nor semi-formal modeling. The TOE must be deeply analyzed in order to provide the description of the TOE modules **High Level and Low-Level Design** (HLD and LLD), and of an implementation subset. Testing is supported by an independent search for obvious vulnerabilities. Development controls are supported by a life-cycle model, tools, and automated configuration management.

EAL4 is the first level that imposes a description of the system security policy. The required model is an informal one. The figure 5 presents the EAL4 level and a part of the required documentation.

We have chosen to increase the EAL4 level by furnishing a semi-formal model of the Java Card runtime: the **Security Policy Model** (SPM) component **ADV_SPM.2**. The security policy model must correspond to the functional specification of the TOE. The following subsections present the semi-formal modeling technique that is used and the dynamic security policy model.

4.1.2 Semi-formal model in UML

UML (Unified Modeling Language) [9] is the standard set by the Object Management Group (OMG) for object-oriented analysis and design facilities. UML includes model diagrams, their semantics, and an interchange format between case tools. Within UML, the OCL (Object Constraint Language) [10] is used to specify constraints like invariants, preconditions, or post-conditions.

UML allows the use of interactions and state machines. A state machine can be used to model the behavior of an individual object. An interaction is used to model the behavior of a society of objects that work together. A state machine corresponds to a behavior that specifies the sequences of states an object goes through during its lifetime. The transitions between two states are relative to event launch. A state machine is used to model the dynamic aspects of runtime. The state of the system (JCVM at runtime) is a condition or situation that satisfies some condition, performs some activity, or waits for some event. A state machine is sufficient to model the policy enforced in a system. Actually, the security policy is not a way to express how things shall be done, but what shall be done. In this section, the construction of the state machine is presented, from the analysis of the Java Card security policy to the UML model.

The aim of the Java Card dynamic security policy is to assure that Java objects are correctly accessed. The security mechanism that performs this verification is the *firewall*. It acts at runtime when the bytecode *interpreter* requires an access check before the current instruction execution. The rules applied by the Java Card applet *firewall* are exposed in the **JCVM section 5** [5] and in the **JCRE section 6** [6]. In the Muse [8] Security Target (ST) this information is included in requirements associated to the **SF_FIREWALL** security function

[1] The garbage collecting is not a standard feature, but is implemented in some of the Gemplus Java Cards.

(TOE Security Function part) description: **FDP_ACC.2/JavaObject** component (TOE Security Functional Requirements part), and **FAU_SAA.1.2/Soft** (lines 3, 4, and 6).

This set of rules is necessary but not sufficient to guarantee the respect of the dynamic security policy. The three following rules must be also considered:

- **FDP_ACF.1.1/JavaObject**, and **FDP_ACF.1.2/JavaObject:** "any access to an object or to the property of an object must be verified",
- **FDP_ACF.1.3/JavaObject:** "no rule can be included to explicitly grant an access permission",
- **FDP_ACF.1.4/JavaObject:** "no rule can be included to explicitly refuse an access".

These rules reveal a strong relation between the *interpreter* and the *firewall*: the *interpreter* cannot decide of the legality or the validity of an access without the *firewall*.

4.1.3 State Diagram

A bytecode execution can be represented as exchanges between the bytecode *interpreter*, the Java stack and the *firewall*. The *interpreter* has to get the parameters of the current instruction from the stack (if any). The *verifier* guarantees the correct issue of this first step, and no system exception shall be thrown. If the instruction execution requires access verification, then the *interpreter* asks the *firewall* for it. An exception shall be thrown if the access is not authorized (SecurityException). If no error occurs during the execution preparation phase, the *interpreter* can execute the code. The following figure illustrates the instruction execution mechanism.

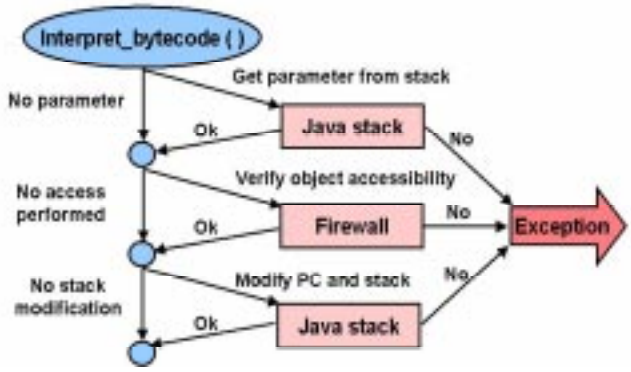


Fig.6 Bytecode execution phases

An analysis of the description of runtime reveals the different states of the JCRE. The definition of those different possible states is the following:

- **State 0**: initial state that corresponds to an applet selection expectation.
- **State 1**: beginning of an instruction execution.
- **State 2**: the instruction requires an object access verification
- **State 3**: the JCRE can execute the code.
- **State 4**: an exception has been thrown and shall be treated.
- **State 5**: final state: end of a JCRE execution phase.

These states underline the execution mechanism. The UML state diagram specifying the runtime security policy also defines the state transitions and some eventually associated constraints (conditions).

The transitions between the different states are the following:

- **Get_Parameters [(a/no)_reference_parameter, (ok/error)]**: the instruction required parameters must be on the stack. A reference may be one of the parameters.
- **Firewall_checks [(ok/exception_throw)]**: the firewall checks that the object reference parameter is accessible in the current context. If the access is illegal, then the JCRE throws a security exception.
- **Bytecode_execution [(next_bytecode/end), (ok/error)]**: execution of the current instruction. The final state is reached if the current bytecode is the last one. An exception shall be thrown if the execution fails.

- **Exception_catch [(ok /error)]**: the exception treatment is the only transition that quits the exception state. It assures that it is not possible to run an instruction that has thrown an exception. If the JCRE detects a catch zone corresponding to the thrown exception, execution continues in the secure zone found. In the other case, runtime ends.

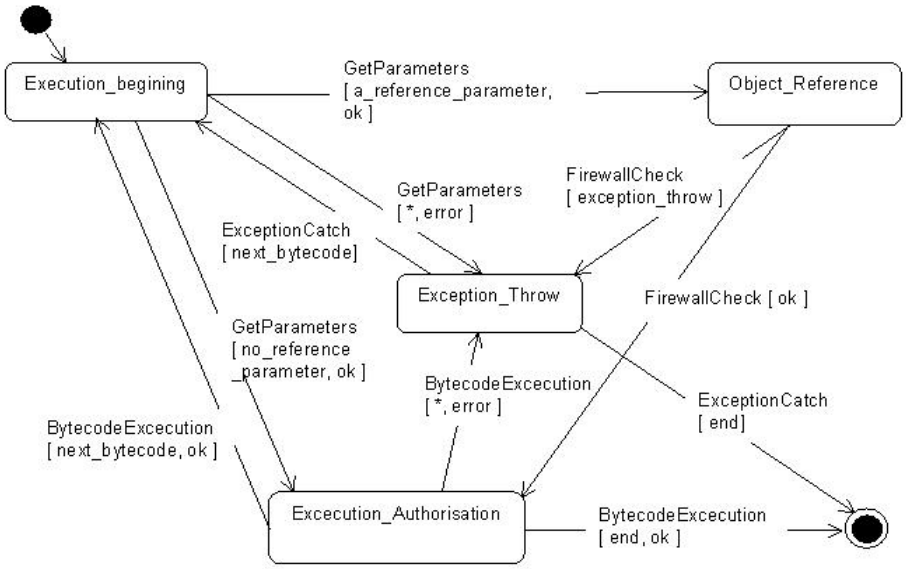


Fig.7 State diagram of the runtime policy

The state diagram of the figure 7 describes the rules and characteristics of the Java Card runtime policy. The correspondence between the informal functional specification and the semi-formal policy model shall be informal. It shall contain the explicit relations between the model and each property exposed in the specification. The use of a semi-formal model in an EAL4 evaluation is not required but recommended: it is a good reflection base for the EAL5 and higher evaluations. The next section is dedicated to the requirements of an EAL5 evaluation for the security policy.

4.2 Security policy in EAL5

4.2.1 EAL5 : semiformally designed and tested

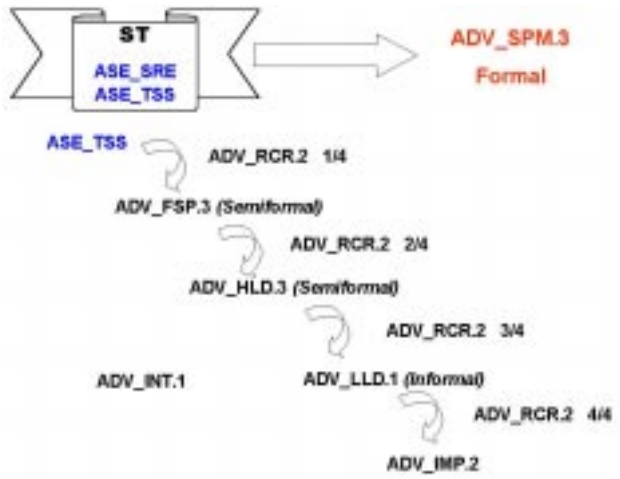


Fig.8 EAL5 description

An **EAL5 evaluation** [1] provides an analysis that includes all the implementation. Assurance is achieved by a formal model, a semiformal presentation of the functional specification and of the HLD, and a semiformal demonstration of correspondence. The search for vulnerabilities must ensure resistance to penetration attackers with a moderate attack potential. Covert channel analysis and modular design are also required.

An EAL5 level permits a developer to gain maximum assurance from security engineering based on rigorous commercial development practices. EAL5 is a high assurance, based on a rigorous development approach, but that does not incur unreasonable costs for specialized security engineering techniques.

At this CC level, a formal model of the TSP shall be provided (**ADV_SPM.3**). The developer shall also demonstrate or prove the correspondence between the TSP and the functional specification. The correspondence must establish the following items:

- the SF are consistent and complete, with respect to the model,
- the model contains the rules and characteristics of all policies of the TSP that can be modeled,
- the model is consistent and complete

4.2.2 Short presentation of the B method

The B method [11] can be used in order to specify, design and code critical software systems [4]. The method covers the entire software life cycle: from the specification to the executable code. This formal method is based on the propositional calculus, the first order predicate, the set theory, and the inference rules. A refinement process is used to obtain the implementation of a B specification. A refinement is a way to reformulate machines data and operations owing to more concrete information (machines expansion). The global correctness of the software system is involved by the verification of mathematical proofs. Each refinement level must be internally correct and must be correct toward its abstraction.

B models are independent entities that represent independent processes. A model can include several machines linked together using the INCLUDES clauses. The architecture of a model depends on the case study modularity. As an example the JCVM can be separated in five modules: the *dispatcher*, the *interpreter*, the *firewall*, the Java stack, the exception manager and the memory. B architecture is based on the object instance theory: a machine that includes another machine cannot be included by this last one. The architecture of the JCVM could be the following.

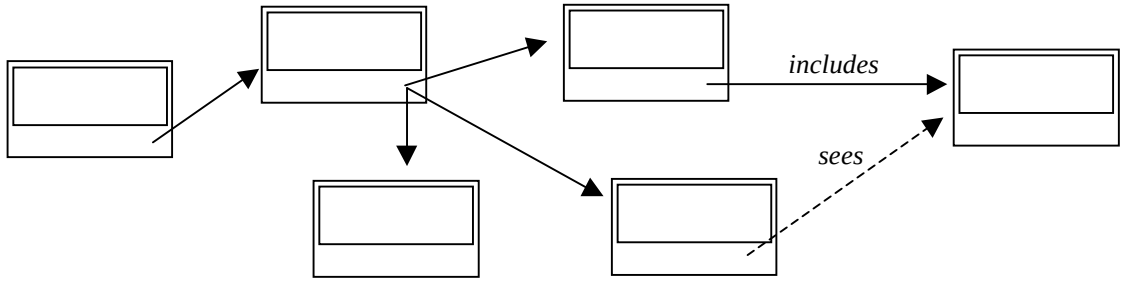


Fig.9 Architecture of the JCVM B machines

A B machine is a way to describe the interface of a module. It contains:

- the objects managed by the module: variables definition and typing,
- the entry points of the module: operations header,
- the internal evolution of objects (machine variables): operations body,
- the dependencies of the module: included, imported or sees machines,
- the properties fulfilled by the module: invariant,
- the initial state of the module: initialization

4.2.3 Security Policy Formal Model

The EAL5 level requires the formal modeling of the TOE policy [2]. The policy of a system is the set of rules that have to be always fulfilled. It is specified and exposed in the INVARIANT clause of a B machine. The properties contained in this clause must be respected after the initialization of the machine variables and after each operation call. A model cannot be proved if this condition is not true. The proof process is a mean to check the global coherence of the mathematical model associated to each B machine. The B tool [12] generates the proof obligations (POs) of the specification according to the mathematical model. A theorem prover is provided to discharge automatically the POs and an interactive theorem prover allows the user to intervene in the proof, when it becomes too complex for the automatic prover.

The B model of the dynamic security policy should not contain any operation: it is a set of predicates to be verified. The formal model required by an EAL5 evaluation should not specify how things are done but what is done.

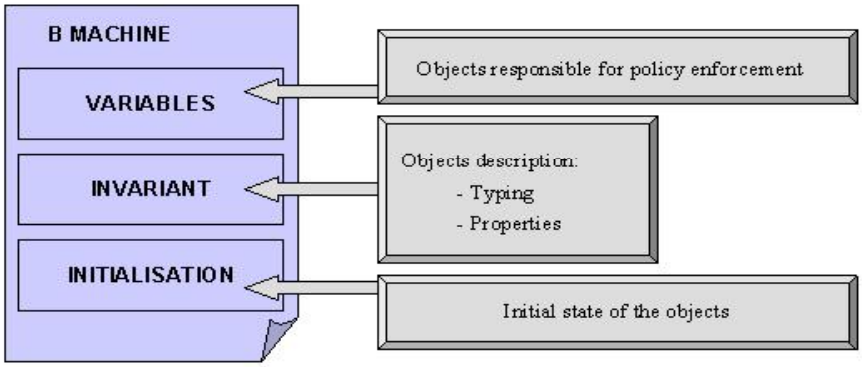


Fig.10 The different clauses of a B machine

Thus, the machines of the policy model only contain:

- the definition of a set of variables (concrete or abstract),
- the initialization of those variables,
- the invariant to be fulfilled by the variables

The dynamic security policy is the set of functional requirements exposed in a CC document (SPM). They are enlightened by analyzing the SFs and mechanisms that "act" at runtime. Then, the Java Card bytecode *interpreter*, the applet *firewall*, the exception manager and the *dispatcher* have to be considered.

The formal model of the Java Card runtime policy is hardly based on the semi-formal model. The idea is to describe exactly what shall happen. Thus, the formal model contains a single machine that mathematically describes all the possible paths in the runtime graph. In the formal model, the different states of the graph must be associated to concrete data of the system. The considered system can be separated into two entities: the bytecode *interpreter* and the applet *firewall*. The *interpreter* processes the current bytecode and eventually calls the *firewall* if the execution requires an access authorization to be delivered. An analysis of the checks performed by the *firewall*, in order to decide whether an object access is legal or not, reveals the information required for those checks. This analysis is based on the 6th section of the JCRE [6]. The *firewall* requires the following information:

- the runtime current context,
- some attributes of the accessed object (its context, its type, its shared property if it is an interface object, its public or global properties),

In the semiformal model, the *interpreter* actions are separated in two parts: parameters obtaining from the stack and the bytecode execution itself. These two actions are respectively associated to the *Java stack* and the *interpreter*. System state characteristics are identified by B machine variables. The sets which name is written in capitals are defined in a context machine. *CONTEXTS* and *MEMORY* represent correct package and reference attributes (16 bits long). *BYTECODES* is the set of supported instructions. *STATUS* is a means to describe the state of the different virtual machine components.

The *RuntimePolicy* machine defines the different JCRE status and the constraints applied to the states transitions. The operation clause may be used in order to specify the transitions' code. This is not required in an EAL5 evaluation: the relations between the security functions (SF) and the policy (policy enforcement) are not required in a formal way. It is introduced in the next sub-section dedicated to the EAL7 evaluation.

4.3 Security policy in EAL7

4.3.1 EAL7 : formally verified design and tested

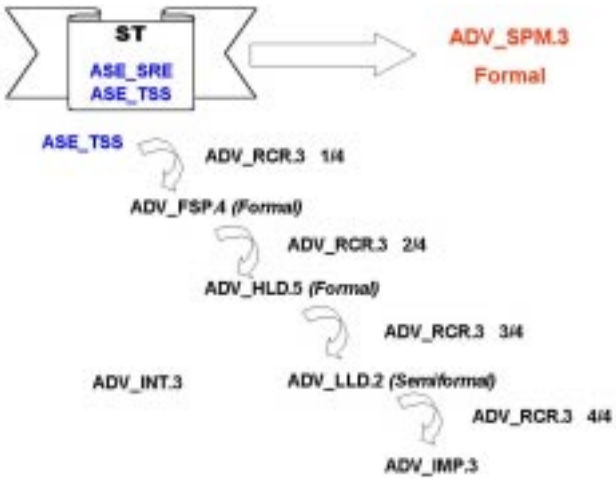


Fig.11 EAL7 description

In an **EAL7 evaluation** [1], the formal model is supplemented by a formal description of the functional specification and high level design, showing the correspondence. Evidence of developer "white box" testing and complete independent confirmations of developer test results are required. Complexity of the design must be minimized.

The EAL7 level represents a meaningful increase in assurance from EAL6 by requiring more comprehensive analysis using formal representations, formal correspondence, and comprehensive testing. In an EAL7 evaluation, an important part of the work has to be done in a formal style. The TSP design provides the formal definition of the policy to be fulfilled by the TOE model. The **Functional Security Policy** (FSP) component imposes the establishment of a model of each SF.

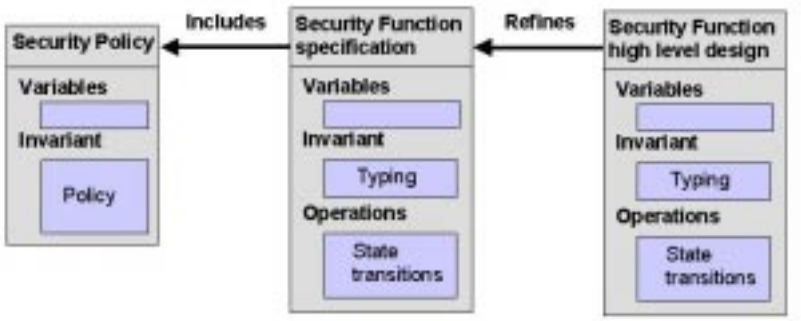
This model shall define and describe all the function interfaces (mechanisms and environment). The high-level design is a refinement of the whole functional specification in a modular way. Thus, the set of the modular TSF models can represent a convenient base for the high level formal model. The same remark can be expressed about the low-level formal model that shall also be furnished. The HLD, LLD and FSP formal model must be consistent, and justify that it is an accurate and complete instantiation of the TOE requirements. The proof or the formal demonstration of the model, according to the security formal properties justifies this point. The **Representation CoRrespondence** (RCR) between two formal descriptions shall be formal. This demonstration is automatically expressed by the proof process. The correspondence between the LLD and HLD shall be semiformal. This semiformal correspondence is based on the identification of a total bijection between the concrete data of both formal and semiformal models. The bijection reveals a similar evolution of all the data and states of the TSF.

4.3.2 Requirements for the security policy formal model

The formal model of the security policy is the same for either EAL5 or EAL7 evaluations. The EAL7 requires a demonstration that the SFs enforce the security policy. This is realized by the proof of the formal model containing both the policy and the functions. The SFs must be formally specified and designed (high level design). The HLD formal model is obtained by refinement of the SF specification, which is an abstraction of the functions description.

The refinement must retain the concrete variables defined in the abstraction. The SF formal models are not expected to expressed any other system property than variables typing (buffer size, data length…). The properties of the mechanism they specify must be inherited from their corresponding security policy model. This is realized using a machine inclusion. It means that an instance of the included machine is associated to the machine. The inclusion link between the SF and the policy they might enforce is not sufficient to demonstrate the correspondence. Actually, each variable of the security policy model must be associated with the variables of the security policy model. Further more, the operations of the SF must be exactly the transitions exposed in the policy state diagram and the evolution of the state number must be considered in each operation.

Fig.12 Security Policy enforcement verification



The proof process assures that:

- the machine inclusion link is correct,
- the machines are independently consistent,
- the operations describe correct paths in the diagram, according to the security policy,
- the high-level design corresponds to the specification

The correspondence proof required by the RCR component is based on the inclusion of the security policy model (entire model or only the concerned modules) in the machines. The figure 12 presents the inclusion mechanism.

5 Conclusions

The CC is a very useful framework for stating security requirements because they use a common assurance framework that has been universally accepted. The CC permits to construct security requirements in order to produce high quality documents. The standard documents are easily comprehensible and usable by any IT professional. The strong security assurance for the CC high levels is achieved with the SPM component that must be written in an informal (ADV_SPM.1), semi-formal (ADV_SPM.2) or formal (ADV_SPM.3) way.

Through this document we provide a methodology to evaluate a product and to obtain the assurance of a high security level owing to formal methods. Moreover, we propose an analysis of the different modeling techniques required in the SPM components. The case study that we have analyzed is the JCRE security policy model case. We show that it is important to base any formal modeling phase on a semiformal analysis, which is also based on an analysis of the informal specification. The security policy model is not a complex one. It only contains a set of predicates to be fulfilled by the TOE. Thus, the formal machines are reduced in an INVARIANT B clause. The construction of such a model requires a deep understanding of the TOE functional requirements. The formal model of the TSFs is a means to describe the security mechanisms represented by the requirements. Their integration in the security policy model requires that the two models have the same format: variable names, state numbers, or machine architecture. Thus, it may be necessary to reconstruct the entire EAL4 semiformal policy model for an EAL7 evaluation. Actually, if the security functions have not been sufficiently analyzed and if the policy representation is not close enough from reality, then the model shall be changed. The proof mechanism is also very important in the assurance procedure. It reveals the quality, the consistency and the refinement correspondence of the models. It is used to show the correspondence between the different formal representations of the security functions and policy. The proof analysis and the proving phases must be respected.

6 References

[1] Common Criteria for Information Technology Security Evaluation Criteria, Version 2.1, August 1999.

[2] Common Evaluation Methodology for Information Technology Security, Version 1.0, August 1999.

[3] Web sites http://www.niap.nist.gov and http://www.scssi.gouv.fr.

[4] "Formal model and implementation of the Java Card Dynamic Security Policy", Stéphanie Motré, Gemplus Research Lab', AFADL'2000, Grenoble, France, January 2000

[5] Java Card 2.1 Virtual Machine (JCVM) specification, Final revision 1.1 June 7, 1999. Published by Sun Microsystems.

[6] Java Card 2.1 Runtime Environment (JCRE) specification, Final revision 1.1 June 7, 1999. Published by Sun Microsystems.

[7] Java Card 2.1 Application Programmer Interface (JCAPI) specification, Final revision 1.1 June 7, 1999. Published by Sun Microsystems.*

[8] The Security Target of Muse project.

[9] Grady Booch, Jame Rumbaugh and Ivar Jacobson, "The Unified Modelling Language", Addison Wesley edition, October 1998

[10] Jos Warmer and Annek Kleppe, "The Object Constraint Language", Addison Wesley edition, October 1998

[11] The B Book, Assigning programs to meanings, J-R Abrial, Cambridge University Press, first published 1996

[12] "B tool User guidelines", Steria Méditerannée, December 1996

[13] Vocable Project, "The new millenium already has its card", Carte Bleue Visa, November 16th 1999