Increasing Smart Card Dependability

Ludovic CASSET, Jean-Louis LANET

Gemplus Research Laboratory, Av du Pic de Bertagne,
13881 Gémenos cedex BP 100.
Ludovic.casset@gemplus.com; jean-louis.lanet@gemplus.com

1. Introduction

Open smart cards like Java Card provide application developers an opportunity to develop rapidly applications by offering the possibility to download during post issuance application into the card. The main drawback with this kind of smart cards is the risk to download a hostile application that may exploit a faulty implementation module of the platform. Security is always a big concern for smart cards, but the issue is getting more intense with multi-applicative platforms, post issuance code downloading and the constant growth of the complexity. Allowing post issuance need to verify that the incoming applet respects the semantics of Java byte code. For this purpose a byte code verifier checks the code during load phase. Unfortunately due to the lack of smart card resources this piece of code has not yet been implemented on a smart card.

As the correct design and implementation of the system is the key to shun an attack, we think that the byte code verifier as a key point of the Java Card security, need to be developed with formal techniques. Smart cards can gain benefits in using formal methods by improving its dependability despite the increasing complexity of this kind of system [Lan-00]. However to use formal methods in industry we need to provide a methodology to integrate it into the software development cycle. With this methodology introduced in section three, we will be able to provide metrics and figures on formal developments that can help industrials to get confidence in formal methods.

We have achieved two challenges: to embed a Java Card byte code verifier into a smart card and to develop this verifier using formal methods. Leroy has already [Ler-01], [Ler-02] developed a prototype of an embedded byte code verifier. He has also provided some formal specification of his verifier. But we are the first to have developed a byte code verifier for Java Card with formal methods. We will see in section two that implementing a byte code verifier for a smart card is not obvious and requires some improvements.

This work is a part of the European project MATISSE[1]. The approach of the MATISSE project is to exploit and enhance existing generic methodologies and associated technologies that support the correct construction of software-based systems. In particular, a strong emphasis is placed on the use of the B Method [Abr-96]. Within this project we evaluate the advantages and the drawbacks of using formal methods in our specific domain.

2. Enhancing the Java Card security

For the Java Card security, it is important that an applet can not have access to the data of other applets by using the sharing mechanism, or access to the code of the operating system. The verifier examines incoming code in order to ensure that it respects the syntax of the byte code language and the language typing rules. The verifier checks statically that the control flow and the data flow do not generate run time error. Other components are responsible for protecting system resources from abuse but they depend on the verifier as they rely on language features such as access restrictions (private,

[1] European IST Project MATISSE *IST-1999-11435*

protected, final, *etc*). It is obvious to say that a vulnerability in this component will be catastrophic for the card. We have specified and implemented such a verifier with all the Java Card byte code features except the subroutine treatment.

We use the proof carrying code (PCC) technique to perform the on-card verification [nec-97, Ros-98, Cas-02]. This verifier scheme is similar to the KVM verifier or to the Java lightweight verification [Sun-00]. The idea is to separate the verification process in two parts as shown below. An off-card part, that computes a certificate, or "proof" indicating that the code is correct with respect to the security policy and an on-card part, that uses the certificate to verify the correctness of downloaded code.

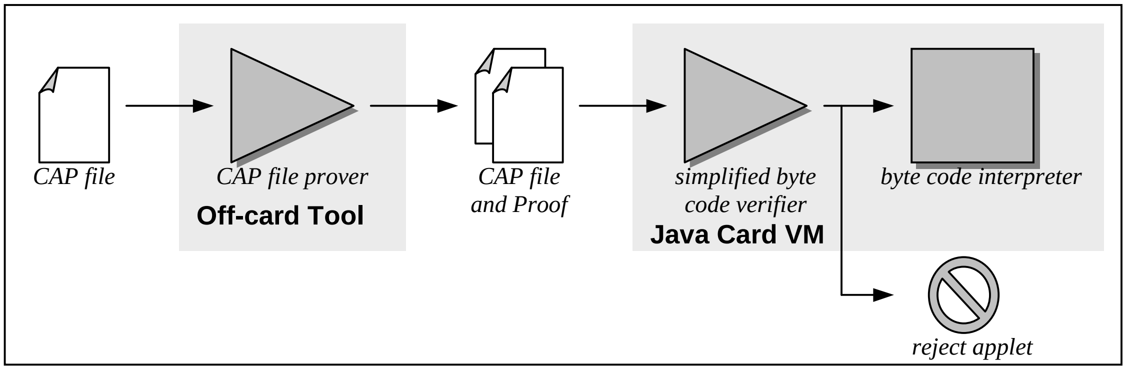

Figure 1 *The PCC verifier*

The "proof" generated is similar to the `StackMap` attribute used by the KVM, and contains the same kind of information. In our development, we add it in an additional component to the CAP file. The proof is built from the CAP file and the export files that have been used to build the CAP file and then added to the CAP. If the CAP file is not a valid CAP file, it is rejected and no proof is added. As shown previously, the proof added consists in type information for specific parts of the code (jump target). This proof is used later by the card to perform verification of Java Card applets when the applet is uploaded to the card. It is not needed afterwards. As this is an off-card treatment, the generation can be a memory and computation intensive process. A first part of the proof generation consists in classical byte code verification. Then, the proof is extracted from the information computed during the verification. Then the on-card verifier uses this additional information to speed up the verification algorithm which becomes linear.

At the end, the software has been embedded into smart card. The chip target is an ATMEL platform, the AT90 SC 6464C. This chip contains 64 kb for the program and 64 kb for the data and 3 kb of scratch memory. The code is stored in the program area while downloaded applet to be verified are stored in the data area.

3. The formal development from high level specification to the implementation

The formal development is included into the general methodology depicted in the following figure. This general methodology helps us identify the different step in an industrial software development, from the earlier requirements and design to the final implementation produced.

The formal development is split in three different phases:

- the translation from the informal specifications into a B model, called *formalization* on the previous figure,
- the formal development to obtain a formal implementation in B0, a subset of the B language, called the *refinement* in the next figure,
- the translation of this formal implementation into a classical programming language such as C, the *translation* phase in the previous figure.

The validation of the development is performed in several ways. As a formal development, it mainly relies on the proof activity that ensures the consistency between the formal implementation and the formal specification. Mathematical lemmas are generated at each step of the refinement process. If it is impossible to prove them, then the model is not consistent and needs to be corrected. The origin can be an error within the model, a lack of properties or of invariant. The proof process aims to prove all these lemmas and to make the correction if needed. In our example, 29 errors have been discovered and corrected during the proof. It appears that these 29 errors have generated hundreds of non provable lemmas. Therefore, the proof phase is a powerful debugger to find and to identify errors.

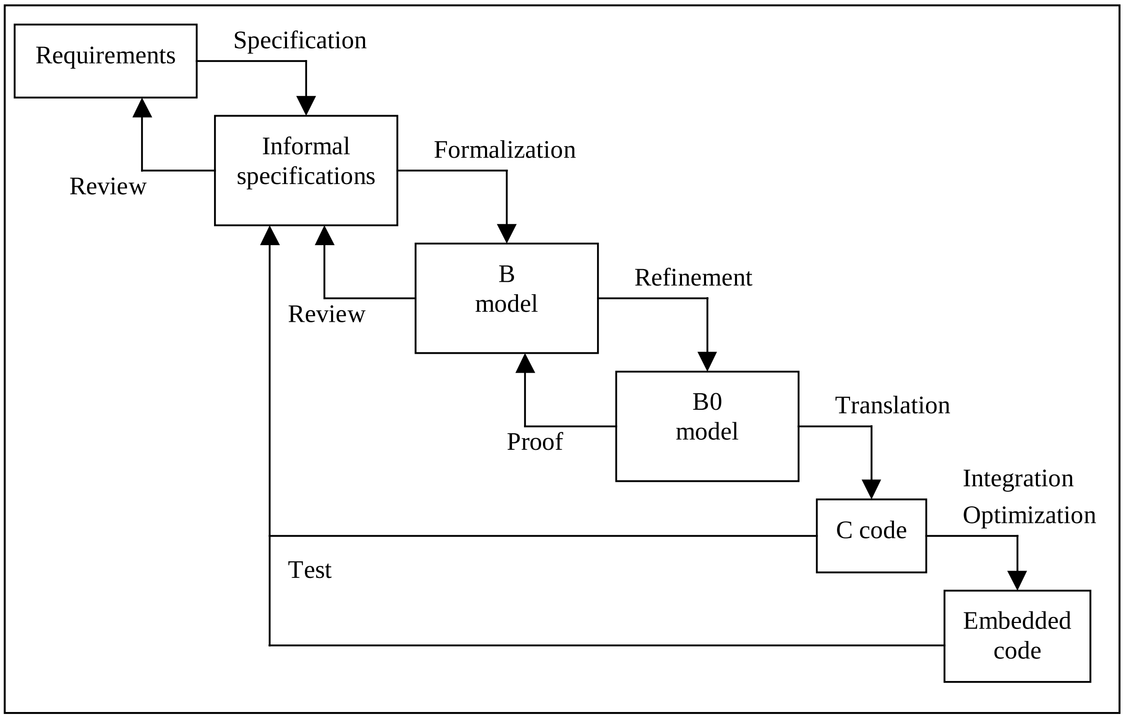

Figure 2 *The software development methodology integrating formal development*

But, as the translation phases, *i.e.* the translation of the informal requirements into formal models and the translation of the formal implementation into C code, can introduce errors due to their human and non formal activities, there is a need to perform some checks. Those checks are done in two different ways. The first one is to do a specification review, identified by the *review* item on the previous figure. The review may have two advantages: to be confident with the model before starting the proof phase and to reduce the possible errors introduced during the translation from the informal requirements to the formal models. In our example, the review process helped us to identify a particular error that has affected nearly 12 different instructions. This error finds its origin in the informal requirements itself, showing that without a review we could have never found this error. The review should be performed by a team different from the development in order to increase its efficiency.

The second way of completing the proof is to test the code produced. It is identified by the *test* item in the previous figure. Formal methods aim to reduce the use of test and mainly of unitary tests. However there is still a need for functional tests. That the kind of tests we aim to perform in order to ensure that the code obtained and installed on our smart card is compliant with the informal requirements. This test plan was generated by a team different from the formal development team. It was developed according to the informal requirements. It then allows us to tests the translation from the informal requirements to formal models, the translation into C code and eventual optimizations that can be performed. All this parts are identified on Figure 2. In our example, we have discovered 23 errors during testing that can be classified in two different categories. The first one is linked with errors introduced during the informal to formal translation: 14 errors have been discovered. This kind of errors are generally not detectable by the proof process. The second one is linked with tools that perform the translation from the formal implementation into a programming language. These tools are proprietary prototypes and have not been qualified according to the standard process. The number of errors discovered in this category is 9. In [Cas-02], the author compares the formal development of a byte code verifier with a conventional one, aiming to find the difference and to emphasize the benefits of using formal methods in industrial developments.

4. Conclusions

This work provides the evidence that the use of formal method for the development of an operating system in a strongly constrained device can improve the quality at an affordable cost. This technique is the most promising one for such dependable devices. Increasing the dependability of smart card is the key to allow applet post issuance downloading Dependability can be obtained through different techniques:

- fault tolerance, which is not compatible with the smart card constraints,
- fault removal, that has shown its limits with the complexity of the new operating system,
- fault avoidance, which can drastically improve the security of this device but unfortunately it is difficult to convince managers to use it.

The results obtained with a dual development, allow us to collect metrics in accordance to an evaluation plan and to develop a methodology. We demonstrate that generating code for a smart card is possible without a too important overhead and that some choices must be carefully done during the development by identifying which part must be formalized and which one can be developed traditionally. For example, some low-level modules of the structural verifier are entirely developed with B, requiring for the proof process significant efforts. Those modules could have been developed with the standard development procedure without reducing the confidence in the code. But integrating legacy code with formally developed one is easy. With such a method it is possible to replace unitary testing by proofs that provide us a higher confidence in our code.

We also learn that the formalization of the informal specification is a key step where we have to pay a special attention. It is also a powerful means to find ambiguities in the informal specification. Most of those conclusions are well known by the community and we just point them out in our specific domain, the smart card. We provide a non trivial example of the use of formal methods. Moreover we reach our challenge, to formally implement a complex piece of code into a smart card.

References

[Abr-96] J.R. Abrial, *The B Book, Assigning Programs to Meanings*, Cambridge University Press, 1996.

[Cas-02] L. Casset, *Development of an Embedded Verifier for Java Card Byte Code using Formal Methods*, In Proceedings of FME 2002, Copenhaguen, Denmark, July 2002.

[Lan-00]] J. -L. Lanet, *Are Smart Cards the Ideal Domain for Applying Formal Methods ?*, In Proceedings of the ZB 2000 Conference, York, United Kingdom, September 2000.

[Ler-01] X. Leroy, *On-Card Byte Code Verification for Java Card*, Proceedings of e-Smart, Cannes, France, September 2001.

[Ler-02] X. Leroy, *Bytecode Verification on Java smart Cards*, to appear in Software Practice and Experience, 2002.

[Nec-97] G. Necula, P. Lee, Proof-Carrying Code, in 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 106-119, Paris, France, 1997.

[Ros-98] E. Rose, K. H. Rose, Lightweight Bytecode Verification, in Formal Underpinnings of Java, OOPSLA'98 Workshop, Vancouver, Canada, October. 1998.

[Sun-00] *Connected, Limited Device Configuration*, Specification 1.0a, Java 2 Platform Micro Edition, Sun Microsystems, 2000.