GemClassifier, a formally developed smart card

Jean-Louis LANET

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

1. Introduction

In a previous paper [Lan-00] we explained that smart cards could be the ideal domain for applying formal methods. We said that the need of formal methods has three origins: mastering the complexity of the new operating systems, certifying at a high level a part of the smart card and reducing the cost of the test. We believed that these reasons were enough to introduce formal methods in the software live cycle. Unfortunately the efforts for integrating data and behavior in a same framework for generating automatically test cases, have not yet been successful. For the certification, the certification obtained by Multos (ITSEC EAL 6) did not encourage the other smart card manufacturers to propose high level certification. If certification helps to introduce formal methods it will be just as a side effect. Finally it was the complexity of the operating systems and the need to avoid vulnerabilities that initiated the GemClassifier smart card development.

We have demonstrated that it was technically feasible to use a given formal method but it was not sure if it was economically acceptable. At that time we have identified some bottlenecks that must be solved in order to allow a real acceptance by the managers:

- development overhead,
- lack of methodology,
- human resistance and
- tools improvements.

We believe that a clean methodology with related metrics and tools improvements will consequently help the integration of formal methods and in particular the B method in the software process. It is important to have guidelines for the specifications and proofs that help the designers. For this purpose we joined a European project, named 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 was placed on the use of the B Method. Within this project we evaluate the advantages and the drawbacks of using formal methods in our specific domain. For this purpose the first task was to set up an evaluation plan [Mat-01].

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

2. MATISSE evaluation plan:

With this project we wanted to determine if **formal methods can be used in developing smart cards in such a way that gains in quality come at predictable and acceptable cost**. By stating this goal we defined the expected effects of using formal methods. The hypotheses to be tested are detailed enough to make clear what measurements are needed to demonstrate the effects. With the following aims, we define five hypotheses.

- H1: The use of formal methods and in particular the B method improve the quality of resulting software. It is of prime importance to demonstrate to managers that a consequent formal development increases the quality.

- H2: The cost overhead of a formal development is acceptable. It is expected that the introduction of formal methods will not raise too much the costs of development. This hypothesis asserts that different treatments have different effects on a project. To draw some conclusions from the case study result, it is necessary to have two developments to determine if there are some differences. The formal development in the case study must have a sister project using the Gemplus procedure for development. The measurements will take into account developments, proofs, reviews, tests and documentation for traditional and formal developments.

- H3: Non specialist engineer can use the formal method effectively when guided by an expert. This hypothesis will point out the importance of know-how and difficulties in using formal methods and will help in the definition of the team. It will also help the project manager to evaluate the part of subcontracting. From the Gemplus experience, it seems very useful to subcontract at the early phase of a project and possibly at the end for solving complex proofs. The know-how of an external expert will be necessary until it can be supplied by experienced practitioners within Gemplus. The second part concerns the training of the developer for formal modeling. It is claimed that B models can be developed by beginners. To verify that point a part of the development will be given to trainees and a special attention will be paid to fluctuations in measurements.

- H4: The use of formal methods and B facilitates fulfilling regulatory requirements. The Common Criteria (CC) requires the use of formal methods from the so-called EAL5 level and upwards. At the EAL4 level, only semi-formal documents are required, but to reach the higher levels formal (mathematical) proofs are required and formal methods and associated tools have to be used (*e.g.* for the formal description of security policies, formal proofs of the consistency of the security policies, *etc*.). There are already very professional development models that fulfil the regulatory requirements at Gemplus. However, formal specification and in some cases even direct software code generation from these specifications would help to improve the way of working and the productivity as well as to prepare for future regulations. A special emphasis is put on using the B method to meet these regulatory requirements. Even if formal methods are not required for meeting these regulations, they would be favorable by making it easier to trace the requirements all the way to the final code. This traceability is a very important feature, since it facilitates the reuse of code in future developments.

- H5: Code generated by the use of the B method does not have significantly increased memory requirements or the execution time. We have verified in previous studies that the produced code from the Atelier B had too large memory footprint to be usable. Moreover we do not know whether using the B method increases the size of the code independently of the code generator. Engineers have a great experience in generating very efficient C code for a smart card. It will be of a great interest to compare the codes because it is often said that constructs used in formal specification may not translate well into the target language leading to either an inefficient implementation, or a substantial amount of re-work to optimize the code design.

The hypotheses H1 and H2 need to have two developments of the same or similar software. We did not have enough time to validate the hypothesis H4. The last hypothesis needed to generate the code that fits the smart card constraints. Between the formal methods only the B method has the possibility to generate acceptable code for smart cards. But the current code translator was not efficient enough for the card, so we developed our own translator. It was a prototype, and has not been validated.

3. The B method

The B Method is a formal method for software engineering developed by J.R. Abrial [Abr-96]. It is a *model oriented* approach to software construction. This method is based on set theory and first order logic and the generalized substitutions. It covers the complete development cycle from specifications to code generation. The basic concept is the *abstract machine* which is used to encapsulate data describing the state of the system. A B project can contain several abstract machines. Invariant can be expressed on the state of the machine which can only be accessed by the specified operations. These operations are the means to access and modify the variables of the machine and they contain a precondition which is a predicate that must hold when the operation is called.

The abstract machine is refined by adding specification details. Several steps can be used before reaching the implementation level where the specification is detailed enough to generate code. The refinements and the implementation have other invariant (gluing invariant) which express relations between the states of the different refinements and preserves the correctness of the abstraction. The implementation corresponds to the last refinement and is written with a subset of the B language: the B0.

The proof process allows to check the consistency among the mathematical model, the abstract machine and the refinements. This way, it is possible to prove that the implementation is correct according to its specification. The tool *AtelierB* generates the proof obligations of the specification according to the mathematical model. A theorem prover is provided to automatically discharge some of the proof obligations and an interactive theorem prover allows the user to interact in the proof process.

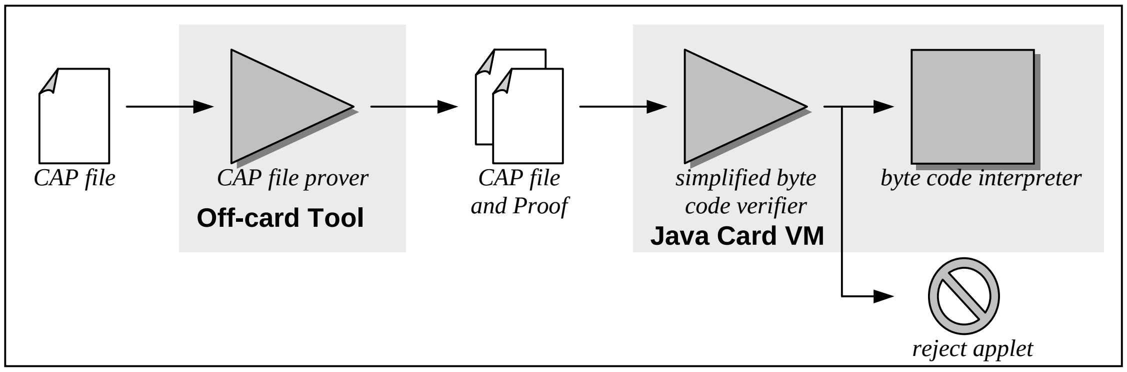
4. The case study

We have chosen to specify and implement a Java byte code verifier on a smart card. For the Java card it is important that an applet can not has access to the data of other applets expect by using the sharing mechanism, or accessing the code of the operating system. The verifier is a key component of the Java security architecture. It examines incoming code in order to ensure that it is valid. It checks that the code respects the syntax of the byte code language and that it respects 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, 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 byte code features except the subroutine treatment.

The two teams have not developed exactly the same software. We developed formally a PCC based verifier [Nec-97] and the algorithm specified by SUN [Lin-96] (we call it stand alone byte code verifier) with the standard development procedure. Both software share a common part: type inference and byte code verification. The evaluation has been done on this part only.

The PCC verifier

The PCC verifier scheme similar to the KVM verifier or to the Java lightweight verification that could be used for Java Card to perform similar verification on card. 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.



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 (for example `COMPONENT_Proof`, or `COMPONENT_StackMap` to follow the KVM naming convention).

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 it. If the CAP file is not a valid one, 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 will be used later by to card to perform verification of Java Card applets when the applet is uploaded to the card, and 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.

Stand alone verifier

This verification algorithm was said to be a too complex operation to be performed by a smart-card, in particular because of the time complexity, and also memory requirement. By taking into account particularity of smart-card hardware and of the verification process, we have found solutions.

Type unification can be optimized by stating that during verification of a Java card applet, we perform a lot of unification between primitive types, and the results for these unification can be statically known. We have taken this property into account by using a particular encoding of the type lattice for Java Card. One of the major problem in smart card programming is the fact that the E^2 sensitive to stress. Our encoding of typing information combined to some interesting properties of the type inference algorithm allows us to heavily decrease the E^2 stress. We have reduced the problem linked to memory consumption by using software cache between RAM and E^2. We use the control graph flow of the currently verified program combined with some interesting properties of the verification algorithm, so that we can avoid some data transfer between RAM and E^2 when it is not necessary. Thus we have less E^2 stress, and verification is faster.

At the end, both software have been embedded into smart card. The chip target is an ATMEL platform, the AT90 SC 3232. This chip contains 32 kb for the program and 32 kb for the data and 1.5 kb of scratch memory. The code is stored in the program area while downloaded applet to be verified are stored in the data area.

5. Main results

We will not comment here the formal development process, details can be found in [Cas-02], but we will compare the two approaches. We are currently in the process of analyzing the collected metrics and but we can provide here some information. The two developments started at the same time and the teams were not separated. Both of them shared the same specification that has been written prior to the development and reviewed by another team. The development is split in three phases:

- the translation from informal specification to an abstract formal model. Its verification is done by review,
- the formal development, its validation is done by refinement and proofs,

- the translation from a formal low level specification into an executable code, verification is done by code review and test.

During the first phase, errors have been discovered by review done by an external reviewer. In the formal development, reviewing may have two main advantages. The first one is to be confident on the model before starting the proof step. As the proof step is a costly step, if some errors can be eliminated by review, it could save time. The second advantages is to reduce possible errors introduced during the informal to formal translation. Some errors have been found during this step. In particular, an error has affected 12 instructions. It concerns a bad transformation of the stack by those instructions. This error is not due to the translation informal to formal, but finds its origin in the informal specification that has been written for the type verifier.

The second source of errors is linked with the B development process. At each step, the proof obligation generator generates lemmas to be proved. If it is impossible to prove them, there is some inconsistencies, errors in the model or lack of properties. The problem is that the automatic prover cannot prove all of them. The ratio is often around 75% of success. The regular process is to look at the lemmas and if they seem to be true to not prove them with the interactive prover and to postpone this phase unless the model development is complete. The interactive proof process is done at the end. Unfortunately 29 errors have generated hundred of unprovable lemmas that have not been immediately identified as false but have been corrected during the development phase. We use the proof process as a powerful debugger.

Another team generated the test plan in accordance to the informal specification and developed the test cases. The error discovered during this phase (23) had two origins. The first one is linked with the specification process (14 errors). If an error occurs during the translation from informal specification to formal specification the proof process is unable to detect them. The second type of errors (9 errors) were due to bugs into the translator. At the end of the refinement process, the final component is an implementation in a subset of the B language the B0 which is very close to C. The translator we used was a proprietary prototype with faults. This prototype has not be qualified according to the standard process.

	Formal development	Standard development procedure
Development workload	12 weeks	12 weeks
Proof workload	6 weeks	NA
Test workload	1 week	3 weeks
Integration	1 week	2 weeks
Total	**20 weeks**	**17 weeks**
Bugs discovered by review	13	24
Bugs discovered by proof	29	-
Bugs discovered with tests	32	71
Total	**74**	**95**

All the bugs discovered with test where at the boundaries of the formal method: at the beginning, the specification process and at the end, the code translation. The other bugs have been discovered during the formal development more or less quickly. Reviewing specification is a task that can be performed in both development. It is an important task as many errors can be rapidly discovered. Discovering errors as soon as possible is the aim of formal methods. Reviewing specification and source code is mandatory for good development.

Regarding the metrics it is clear that the hypothesis H1 is true. At the end, with a non-prototype translator the formal development has less bugs: 27 (13 discovered by review and 14 by test) versus 71 (review and test). The hypothesis H2 considers that the overhead is limited. We needed around 20% more time to develop the type verifier. It is now to product manager to decide if 20% is acceptable or not for increasing the quality of the resulting code. We have not yet results for assessing the hypothesis H3. We have not have trainee on this project, but part of the proof process has been subcontracted. The hypothesis H4 cannot be assess, we have had not enough time to define a target and a security policy and to check which help the developed model would provide to the design of the HLD. This design is a refinement of the whole functional specification in a modular way. The hypothesis H5 is true, the code has been embedded into a card. However, more improvements can be done on the code generator.

6. Conclusions

At the beginning of the project we wanted to assess if formal methods can be used in developing smart cards in such a way that gains in quality come at predictable and acceptable cost. For verifying it we proposed five hypotheses and none of them have been invalidated. It is clear yet that some improvements in the methodology and tool are needed.

Other results of formal development show that not all the parts of the program need to use formal methods. For example, some low-level modules of the structural verifier were 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. We also learnt that the formalization of the informal specification is a key step where we have to pay a special attention. It was also a powerful means to find ambiguities in the informal specification. Most of those conclusions were well known by the community and we just point them out in our specific domain the smart card. We provided a non trivial example of the formal method use in our domain. Moreover we implemented it into a smart card which was a real challenge.

References

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

[Cas-02] L. Casset, L. Burdy, A.Requet, *Formal Development of an Embedded Verifier for Java Card Byte Code*, submitted at DSN 2002.

[Lan-00] J.-L. Lanet, *Are Smart Card the Ideal Domain for Applying Formal Methods*, ZB 2000, August 2000, York.

[Lin-96] T. Lindholm, F. Yellin, *The Java Virtual Machine Specification*, Addison Wessley 1996.

[Mat-01] Matisse Project IST-1999-11435, *E0, the Common Evaluation Plan*, 2001, http://www.matisse.qinetiq.com.

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