Available online at www.sciencedirect.com

SciVerse ScienceDirect

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 280 (2011) 3–22

www.elsevier.com/locate/entcs

On Using B in the Design of Secure Micro-controllers: An Experience Report

Marc Benveniste[1,2,3]

Secure Microcontrollers Division
STMicroelectronics
Rousset, France

Abstract

The stepwise formal development of safety critical software is now a well established engineering practice, noticeably in railway systems. However, it has not been applied as successfully to hardware development, where formal methods are mainly used for verification and gate level transformations and optimizations. In this paper, we report our recent experience in the stepwise formal development of a real macro-cell, that opens the way to the design of synchronous digital circuits with zero functional bugs. We propose a development flow suited for obtaining proven correct-by-construction circuits that further possess additional robustness properties desirable for secure chips. The reported work is prospective and is meant to show the feasibility of such a technique for high confidence trustful devices.

Keywords: formal development flow, digital circuit, robustness, secure micro-controller, correct by construction.

1 Introduction

The first contribution of this work shows the feasibility of a stepwise transformation of a formal security policy model into a synthesize-able hardware description of the security functionality that implements it.

The second contribution of this work enhances the experimental formal development flow so that robustness properties can also be handled in a proven way.

The third contribution of this work is an experimental combination of abstract interpretation and model checking to verify a given set of properties on an algorithm before any attempt to implement it.

Finally, we report on unsolved issues that have been identified and that call for further research and development work.

[1] Partially supported by Provence-Alpes-Côte d'Azur (PACA) Regional Council STM-focused Forcoment project, PS-17bis, in a close and fruitful collaboration with Clearsy System Engineering.

[2] Partially supported by Medea+ 2A303 Biop@ss project.

[3] Email: marc.benveniste@st.com

doi:10.1016/j.entcs.2011.11.014

2 Towards a formal development flow

A traditional development flow for digital circuits heavily relies on testing for the verification steps performed before launching the manufacturing process in a foundry. Figure 1 below presents a schematic view of the main artefacts used during the first steps of a typical micro-controller function development.

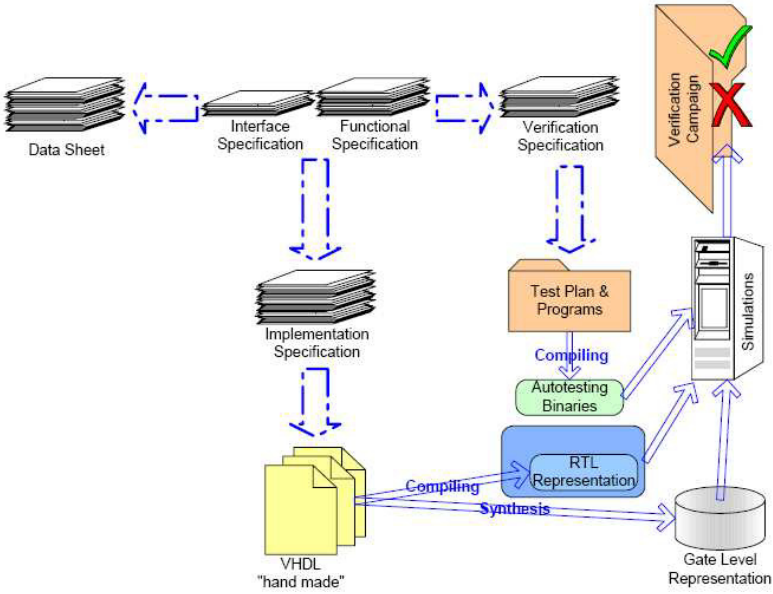

Fig. 1. First steps of the development flow of a digital function for a micro-controller circuit

Writing test programs, running and analysing simulations, represent a preponderant part of this first development effort. The verification plan is solely built upon expertise and experience of specialized designers. Coverage of a verification campaign is measured by using some specialized tools, but functional specification coverage is at best mainly a matter of peer review: the only available reference is the natural language description of the function to be designed.

A simple idea roots our work. Looking at a safety-critical software development widely recognized as a success story, i.e. the Paris Métro Line 14 automated system [25], it can be noticed that verification was performed all along the development through the use of the stepwise refinement methodology underpinning the B technology [2].

The simple idea was to adapt this B formal method to the development of our digital circuits. Indeed, microelectronics development tools already resort to many formal methods, but they are used mainly behind scenes, i.e. without designer awareness of their presence, and more importantly, their application only starts after the source code has been written. These methods are used for verification purpose and for various transformations in the long path leading from a Register Transfer Level (RTL) representation to a final placed and routed net list. Nevertheless, the more expensive functional errors are often introduced before the first line of code is even written. Usually they stem from the functional, the high-level design, or

even the detailed design specifications. As already mentioned, no verification tool is available for these natural language representations. Only a thorough peer review can possibly filter out these error seeds before they blossom into forests of erroneous behaviours, once embedded into the circuit.

We defined, and experienced on a use case [4], a new development flow that makes extensive use of formal proof and produces an exhaustively verified source code both in its functional behaviour and in its interface definition (see figure 2). This source code, actually VHDL, can directly be synthesized. It has therefore the capability to enter the rest of the standard industrial flow untouched.

This experimental development flow relies on a formal model of the required function expressed in Event-B [1], a recent evolution of the B Method targeting system development. We believe that this new flow can be used to build any digital function in a proven correct-by-construction way. Analogue functions fall out of the scope of these techniques, but their digital interfaces are used as a formal correctness contract in order to include them in the otherwise proven digital circuit.

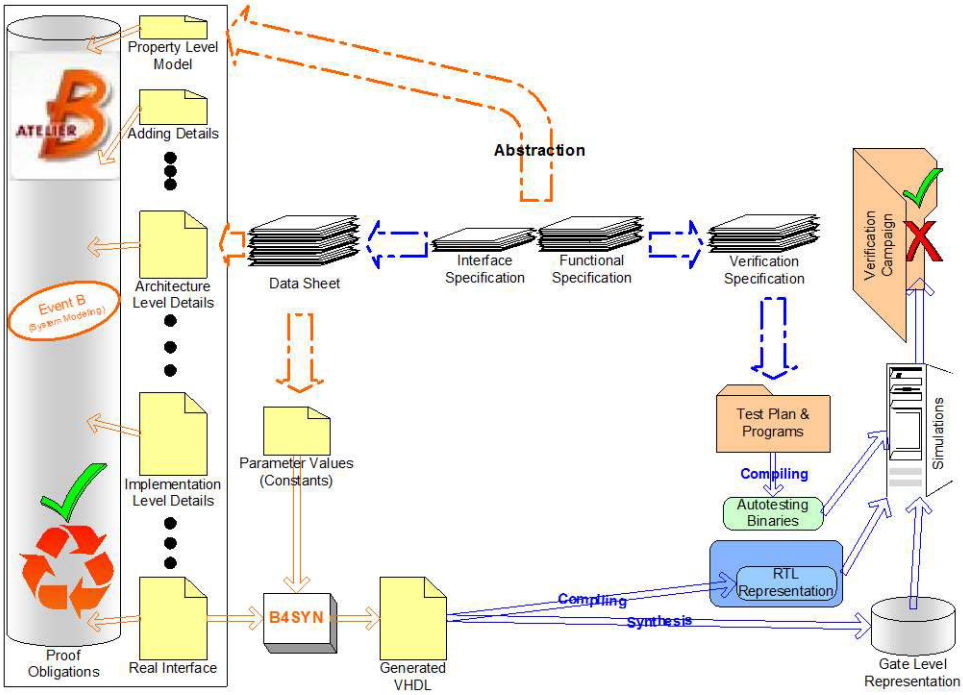

Fig. 2. The proposed development flow, as defined and experienced on a case study

3 Results

The proposed flow begins with a huge abstraction step. The designer must write an Event-B model of only a few lines that captures the *essential* property of the function to be designed. Achieving this goal is no easy task. Many trials are required even for a highly trained engineer. Just be reminded that simplicity is almost always very hard to achieve.

The case study we experienced with was no exception. We had to struggle a while before stating the following:

Definition 3.1 [Essential Property] The *essential* property of a memory protection unit is to monitor all accesses a microprocessor performs.

This function is central to the access control security policy to be implemented on the secure micro-controller as illustrated in figure 3.

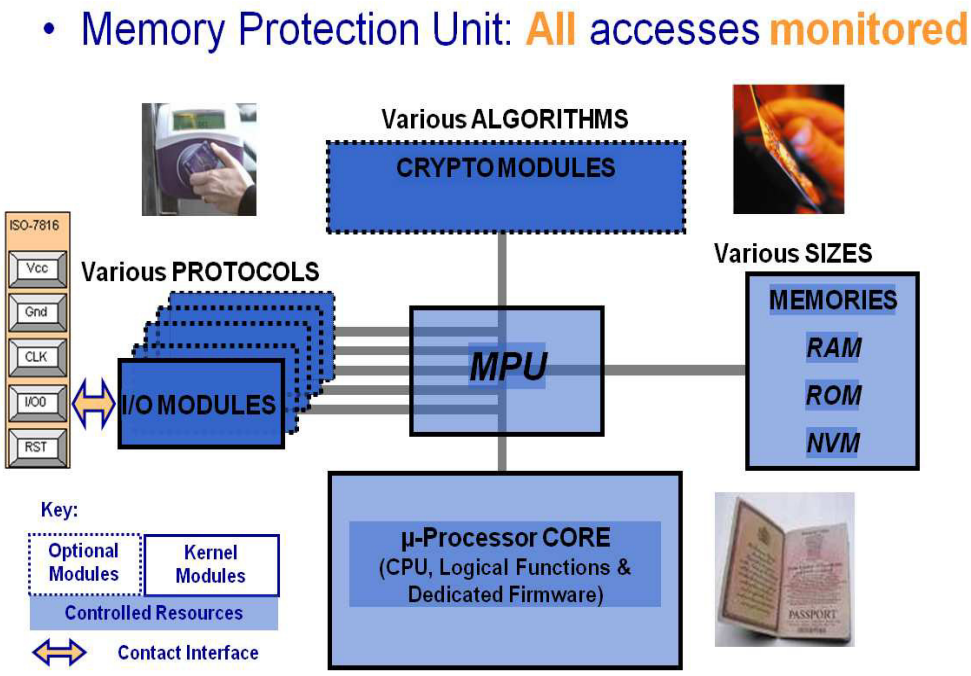

Fig. 3. Case study: The memory protection unit of a secure micro-controller

The initial property level model is presented in a simplified way in figure 4.

Implementation details of the designed function are introduced gradually as it is embedded into the host micro-controller. The main strategy of the refinement plan is to gradually define and transform the variable denoting the access rights matrix, *r0*. The twofold transformation is guided on one hand, by the introduction of different type of accesses and their associated details, and on the other, by the implementation representations for the rights matrix. It should be noted that our refinement path was constrained by the existence of a legacy module already implementing the required functionality. That is why an *Interface Specification* and a chapter of the *Data Sheet* are identified as inputs to the development flow in figure 2: the former provides the wire-level interface for the module and the bus protocol to comply to; the latter defines the register addresses, names, functions, bit meanings and programmer's guidance. An example of a timer description embedded in a general purpose micro-controller public data sheet [23] is provided in figure 5.

Various refinement paths and strategies were tried, often leading to a dead-end or to complex situations were proof work would become overwhelming. The

```
SYSTEM
    MPU
SETS
  ADs ; CLs ; BYs ; TYs
ABSTRACT_CONSTANTS
    C_0, R_0
PROPERTIES
    C_0 ∈ CLs
∧   R_0 ⊆ ADs × CLs × BYs × TYs
VARIABLES
    c0, r0, v0
INVARIANT
    c0 ∈ CLs
∧   r0 ⊆ ADs × CLs × BYs × TYs
∧   v0 ∈ BOOL
INITIALISATION
    c0, r0, v0 := C_0, R_0, FALSE
EVENTS
  allow = ANY a0,d0,t0
       WHERE
         a0 ∈ ADs ∧ d0 ∈ BYs ∧ t0 ∈ TYs
     ∧      (a0 ↦ c0 ↦ d0 ↦ t0) ∈ r0
       THEN
        v0 := FALSE
       || c0 :∈ CLs
       || r0 :∈ ADs × CLs × BYs ↔ TYs
  END ;
  deny = ANY a0,d0,t0
       WHERE
         a0 ∈ ADs ∧ d0 ∈ BYs ∧ t0 ∈ TYs
     ∧      (a0 ↦ c0 ↦ d0 ↦ t0) ∉ r0
       THEN
         v0 := TRUE
       END
END
```

Fig. 4. Case study: Initial property level model

refinement plan we finally established has two important "no" properties: no loss of coverage and no non determinism. Indeed, as this clocked circuit must have a well-defined behaviour for any combination of inputs at every cycle, the Event-B model must implement a *total* and *deterministic* function:

- The disjunction of its event's guards must be vacuously true. Furthermore, we imposed that no "holes" could be introduced when refining an event. Whenever an (abstract) event *ea* is refined (split) by (concrete) events ec_i, if the disjunction of the concrete event's guards is implied by the guard of the abstract event, we can be sure that no "case" has been left uncovered. In a dual way, whenever (abstract) events ea_i are refined (grouped) by a (concrete) event *ec*, we have to prove that the concrete event's guard is implied by the disjunction of the abstract event's guards. We call this property the *coverage* property;
- Non deterministic behaviours are proscribed for secure micro-controllers. We had to ensure that events were *pairwise exclusive*. We call this property the *exclusiveness* property.

Enforcing these properties was implemented through the generation of additional proof obligations in the Event-B system engineering tool Atelier B [3]. Note that proving these properties at each refinement stage reduces the total number of proof obligations, noticeably for the exclusiveness property, combinatorial in nature.

In order to provide a flavour of the strategy we propose for developing digital circuits, the main refinement stages are sketched in the following paragraphs.

TIMER 1& 2 (Cont'd)

4.2.5 TIMER 1 Registers

Timer Status Control Register (TSCR)

Address: 0D4h — Read/Write

7							0
TMZ	ETI	TOUT	DOUT	PSI	PS2	PS1	PS0

Bit 7 = **TMZ**: *Timer Zero bit*

A low-to-high transition indicates that the timer count register has decrement to zero. This bit must be cleared by user software before starting a new count.

Bit 6 = **ETI**: *Enable Timer Interrup*

When set, enables the timer interrupt request. If ETI=0 the timer interrupt is disabled. If ETI=1 and TMZ=1 an interrupt request is generated.

Bit 5 = **TOUT**: *Timer Output Control.*

When low, this bit select the input mode for the TIMER pin. when high the output mode is selected.

Bit 4 = **DOUT**: *Data Ouput*

Data sent to the timer output when TMZ is set high (output mode only). Input mode selection (input mode only)

Bit 3 = **PSI**: *Prescaler Initialize Bit*

Used to initialize the prescaler and inhibit its counting. When PSI="0" the prescaler is set to 7Fh and the counter is inhibited. When PSI="1" the prescaler is enabled to count downwards. As long as PSI="0" both counter and prescaler are not running.

Bit 2, 1, 0 = **PS2, PS1, PS0**: *Prescaler Mux. Select.* These bits select the division ratio of the prescaler register.

Table 17. Prescaler Division Factors

PS2	PS1	PS0	Divided by
0	0	0	1
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

Timer Counter Register (TCR)

Address: 0D3h — Read/Write

7							0
D7	D6	D5	D4	D3	D2	D1	D0

Bit 7-0 = **D7-D0**: *Counter Bits.*

Prescaler Register PSC

Address: 0D2h — Read/Write

7							0
D7	D6	D5	D4	D3	D2	D1	D0

Bit 7 = **D7**: Always read as "0".

Bit 6-0 = **D6-D0**: Prescaler Bits.

Fig. 5. Case study: Example of a data sheet level description, here a timer register

3.1 First levels

The first refinement introduced the different types of accesses, like read (TyR) and write (TyW) for data, fetch (TyX) for code, begin interrupt (TyB) and end interrupt (TyE), and no access (TyN). The latter was initially forgotten but was highlighted by the coverage proof obligations. These events are all refinements of the root "positive" event *allow*. The access matrix *r0* is more precisely defined as a collection of matrices, one for each access type. Each of these matrices is left undefined at this stage. Only their relationship to *r0* is fully defined as illustrated in figure 6.

Next refinement introduces the automaton that drives some exceptions in case of interrupts and on reset conditions of the host micro-controller. Only the state of the automaton and the transitions performed by the concerned events are kept in the next refinement. This is a quite elegant way to constraint the system behaviour for the rest of the development.

Next refinement splits the root "negative" event *deny* according to the access

$$
\begin{aligned}
r0 \quad = \quad & (rr1 \times BYs \times \{TyR\}) \cup \\
& (rw1 \times \{TyW\}) \cup \\
& (rx1 \times BYs \times \{TyX\}) \cup \\
& (rb1 \times BYs \times \{TyB\}) \cup \\
& (re1 \times BYs \times \{TyE\}) \cup \\
& (ADs \times CLs \times BYs \times \{TyN\})
\end{aligned}
$$

Fig. 6. Case study: Invariant anchoring the *definition* of *r0*

type. It could have been done together with the first refinement, but it would have added useless complexity to the automaton definition. This refinement introduces also a local stack to manage clearance levels in case of nested interrupts. Matrices *rb1* and *re1* get totally defined at this stage using the top of stack (empty or full) and the state of the automaton introduced in the previous refinement.

3.2 Architecture levels

Architecture level details are then introduced. For instance, the memory map of the host micro-controller is used to partition the set of addresses and the security policy is specialized on the one hand for register addresses, and on the other, for memory addresses. Correspondingly, matrices *rr1*, *rw1* and *rx1* are defined by expressions involving the adequate addresses, even though some of these expressions are not yet fully defined. They will get defined at a later stage. This is the essence of this refinement technique.

The next two refinements deal separately with the access policy for data and that for code. Matrices *rr4* (part of *rr1*) and *rw4* (part of *rw1*) get more precisely defined for the data access policy, introducing (new) matrices *rr5* and *rw5* that will get defined in later stages. Just as for *r0*, the relation between *rr4* (resp. *rw4*) and *rr5* (resp. *rw5*) is fully defined. The same holds for matrix *rx4* (part of *rx1*) and the relations and sets introduced to *define* it in the 7^{th} refinement.

3.3 Implementation levels

Implementation details make their appearance in the following refinements. One of the key points of our successful refinement plan is the way we managed to keep out of arithmetic although we were dealing with addresses. We deferred the use of arithmetic until the definition of the concrete constants parametrizing the whole development. That is why in figure 2, the file that provides the valuation of all constants is an input to the B4SYN, a specific translator of B for synthesis into VHDL [4]. The "trick" here is to rely on VHDL's type checking and proper use of arithmetic while preferring set theory for Event-B. Indeed, the basic data type in VHDL for synthesis is the `std_logic_vector` that represents *both* numbers in binary base *and* sets of powers of 2.

We introduced the main concept of the case study, the *segment*, i.e. an accessible address window, through the sole use of sets (given set *SEs*). Firstly introduced as a set of addresses ($sa7$), see example 3.2, they became sets of address ranges specified with a start and an end address. They were finally refined to be defined by start and end address registers.

Example 3.2 [Case study: Segments as sets of addresses]

$$sa7 \in SEs \leftrightarrow ADs$$

Let us mention our use of constructive set expressions to pave the way towards a VHDL translation that can be synthesized. For instance, letting $ms7$ be the set of mapped segments, we can *build* the set $ea7$ of segments associated to a given address $a0$ with the expression shown in example 3.3.

Example 3.3 [Case study: Constructive set expressions]

$$ea7 = \{\ xa \mid xa \in SEs \wedge xa \in ms7 \wedge (xa \mapsto a0) \in sa7\ \}$$

These expressions are logically equivalent to let-expressions in B and in functional languages like ML [14]. They smoothly translate into combinatorial logic when simple enough predicates are used.

Two more refinement stages allowed us to identify precisely the conditions under which each violation alarm bit was required to be set. Again, the alarm register is simply modelled by a set-valued variable in Event-B. Setting and clearing an alarm bit provides code for set union and set difference operations as easily as testing a bit value provides code for predicates of alarm membership. This is illustrated in figure 7 where the unmapped alarm is risen in *alm8* because the read access, not part of an exception, is made to an address that does not belong to any mapped segment.

```
:
  alm8 ⊆ ALMs
  alm8 := ∅
:
  deny_read_memory_Unmap
    ref deny_read_memory
=
  SELECT
    m0 = 2
  ∧ t0 = TyR
  ∧ a0 ∈ MAs
  ∧ (a0 ∈ IVs ⇒ i2 ≠ 1)
  ∧ ea7 = ∅
  THEN
    m0 := 0
  || v0 := TRUE
  || alm8 := alm8 ∪ {Unmap}
  END
```

Fig. 7. Case study: Alarm register modelled as a set

We were also careful not to define modifications of the variables bound to become registers and/or outputs *too early* to avoid complex proof work during refinement. Modifications are only indicated with the "becomes such that" substitution for as long as possible. They become concrete modifications in the three last refinements. To keep proof work manageable, read (producing outputs) and write (modifying

registers) operations get concrete in separate refinement stages. The effective read and write operations are introduced with constant functions that clearly define the functional interface between the host and the circuit under development, see *status1b* and *lock* in figure 8. In fact, the natural use of constants in Event-B developments promotes a total separation of concerns between functionality and host interface through "accessors" constants. This separation is not widely practised in traditional VHDL writing.

```
...
  status1b ∈ P (ALMs) ⇸ BYs
  lock ∈ BYs ⇸ BOOL
...
  read_status1
=
  SELECT
    m0 = 2 ∧ t0 = TyR ∧ c0 ∈ p4 ∧ a0 ∈ A3Ds ∧ b13 = BdS1
  THEN
    m0, v0, out13 := 0, FALSE, status1b(alm8)
  END ;
  write_lock
=
  SELECT
    m0 = 2 ∧ t0 = TyW ∧ c0 = ClSy ∧ a0 ∈ A3Ds ∧ b13 = BdL
  THEN
    m0, v0, l4 := 0, FALSE, lock(d0)
  END
```

Fig. 8. Case study: Separation of concerns using accessors to interface to the host

3.4 Real interface

A last refinement introduces the physical interface of the circuit as it is embedded into the chosen host micro-controller. This refinement strikingly illustrates the "Assigning Programs to Meanings" essence of the B technology [2]. Each combination of values of the incoming wires "codes" one of the semantic events as illustrated in figure 9. We advocate to write this correspondence in the invariant so that the concrete substitutions of the event refinements get cross checked by the proof obligations of invariance. When interfaces get trickier, this redundancy proves very useful. Besides, one can imagine using this invariant as an assertion to monitor the host system as reported in [6]. Be reminded that the coverage property will ensure that all possible combinations of inputs have an associated event, and that the exclusiveness property will ensure that each associated event is unique.

To summarize these subsections, the chosen refinement strategy clearly shows that predicate logic, set operations, relations and functions are concepts much closer to the binary logics of a circuit than it may appear at first sight.

3.5 Generating VHDL

The refined model has become close enough to an explicit implementation. The last level of refinement is translated into a VHDL module by the use of B4SYN [4], the constant valuation file, and a translator configuration file that indicates, among others, the list of inputs, outputs, clocks and synchronous events. The produced

```
⋮
      (m0 ≠ 0 ⇒
        (t1 = TyR   ⇔
          ( (PSEL ↦ PENABLE ↦ PWRITE)  = (TRUE ↦ TRUE ↦ FALSE) ) ) )
∧     (m0 ≠ 0 ⇒
        (t1 = TyW   ⇔
          ( (PSEL ↦ PENABLE ↦ PWRITE)  = (TRUE ↦ TRUE ↦ TRUE)  ) ) )
⋮
  phi_read ref phi =
  SELECT
    m0 = 0
∧     RESET = FALSE
∧     CLK = TRUE
∧     PSEL = TRUE
∧     PENABLE = TRUE
∧     PWRITE = FALSE
  THEN
    m0, a1, t1 := 1, PADDR, TyR
  END;
  phi_write ref phi =
  SELECT
    m0 = 0
∧     RESET = FALSE
∧     CLK = TRUE
∧     PSEL = TRUE
∧     PENABLE = TRUE
∧     PWRITE = TRUE
  THEN
    m0, a1, d1, t1 := 1, PADDR, PWDATA, TyW
  END
```

Fig. 9. Case study: Assigning wires to events

VHDL can directly be synthesized, and as B4SYN preserves the Event-B semantics [4], this VHDL source code is a proven correct-by-construction implementation of the original abstract function essential property 3.1.

The produced VHDL source code can now safely be integrated into the classical flow for the rest of the circuit development. All the proof work performed by the designer, and recorded by Atelier B [3], remains as deliverable evidence to any third party inquiry on the correctness of that function of the circuit. Indeed, these discharged proof obligations formalize the correctness rationale of all the design decisions that found their way into the function development. Our case study required about 18 development stages generating around 1 600 proof obligations for a final net list in the order of 5 000 elementary gates. The obtained VHDL module was submitted to the verification campaign available from the legacy development of this function and all tests reached a pass verdict both at RTL level and at gate level, thereby confirming an indistinguishable host-level behaviour over the campaign coverage. All this work has been assessed by a security evaluation facility and contributed to the world's first EAL6+ Common Criteria 3.1 certificate, awarded by the French certification body, ANSSI [13,9].

4 From correctness to robustness

It is well known that a correct design is not necessarily a secure design, the converse being true however. Our experimental flow uses a correctness refinement

[4] Work not yet done.

relation that fails, in general, to preserve security properties such as confidentiality or integrity. This is by no means a serious drawback to the proposed flow. We strongly believe that alternative specific formal methods, and their associated toolsets, should be used to tackle those properties at each development step. Although the following publications are not very recent, we can refer the reader to [5] for a practical approach to the verification of cryptographic protocols, to [10,8] for an original organization based access control model and its application to network security policies, to [22] for a survey on enforcing information-flow policies like integrity or confidentiality through static program analysis techniques, or to [20] for a non-interference formulation that can be preserved under process algebra refinement, amongst many other relevant work devoted to security properties.

However, some security properties, in practice, can indeed be handled by our proposed flow.

Take integrity for instance. The proposed development flow makes the silent assumption that the underlying execution model, i.e. the elementary gate, is a perfect device. Note that this assumption is also present in all source code descriptions of digital circuits, i.e. in all current development flows. Real life teaches us otherwise: laboratory experience shows that building block devices are not perfect and are subject to various disruptions, be them of accidental or malicious nature. For instance, a flip-flop, the basic logic (i.e. volatile) memory element, could be forced to a given value through controlled aggressions of the digital circuit environment [12]. The standard semantics of Event-B excludes this kind of misbehaviour and therefore our proposed flow is not well equipped to mention this kind of integrity losses. We call them robustness issues.

A model of a flip-flop bank, or register, written in Event-B is presented in figure 10. It is parametrized by its reset value, *Q_1*, and its logical address *A_1*. On a *write_reg* event, the presented input *d1* is stored into its state variable *q1*. Just as we did in the previous case study, we defer the explicit introduction of an output variable to a later refinement stage, so on a *read_reg* event, no observable modification occurs, but *q1*, the last stored value is available for output. The last event, *none*, models an irrelevant access, i.e. either not to this address or not a read or a write.

We could introduce these issues into our developments by explicitly modelling disruption as follows: simply replace every assignment by an non-deterministic choice between the correct assignment and a "becomes any value" command. This path, although interesting to *describe* robustness properties, happens to be a dead-end for the *construction* of any desired function since following it would allow us to build a randomly failing device!

The second simple idea we followed stems from the observation that once a failing mode causing the integrity loss is identified, a secure-circuit designer has no other choice than to build a counter-measure. That counter-measure is designed-in and, in the end, it happens to be just another function! For the flip-flop example at hand, it could be some kind of redundant information management function such that it is physically impractical to externally force a change both in the vulnerable

REFINEMENT *ff01a* **REFINES** *ff00a*
SETS
$ADs\ ;\ BYs\ ; TYs = \{TyR,\ TyW,\ TyN\}$
ABSTRACT_CONSTANTS *A_1, Q_1*
PROPERTIES $A\_1 \in ADs \wedge Q\_1 \in BYs$
VARIABLES *m0, a1, d1, t1, q1*
INVARIANT
$a1 \in ADs \wedge d1 \in BYs \wedge t1 \in TYs \wedge q1 \in BYs$
INITIALISATION
$m0 := 0 \ ||\ a1 :\in ADs \ ||\ d1 :\in BYs \ ||\ t1 :\in TYs \ ||\ q1 :\in BYs$
EVENTS
read_reg =
SELECT
$m0 = 2 \wedge a1 = A\_1 \wedge t1 = TyR$
THEN
$m0 := 0$
END;
write_reg =
SELECT
$m0 = 2 \wedge a1 = A\_1 \wedge t1 = TyW$
THEN
$m0,\ q1 := 0,\ d1$
END;
none =
SELECT
$m0 = 2 \wedge (a1 = A\_1 \Rightarrow t1 = TyN)$
THEN
$m0 := 0$
END
END

Fig. 10. Example of a flip-flop bank, or register, in Event-B

flip-flop and in the implementation of its redundant function. In this way, integrity loss is not avoided but it becomes detected and hence, the security breach attempt can be reported and dealt with through other functions either in hardware, firmware or software.

4.1 Focusing on correctness

For our register example, we start with the introduction of the anticipated alarm, *err0*, initially unset. Events *write_reg*, *read_reg* and *none* get their guards reinforced by the condition that the alarm is not set. We introduce a redundancy constant function, in fact a bijection, *RED_2*, left undefined at this stage, and the redundant state variable *q2*. The invariant ties everything together formalizing the counter-measure intent as shown in figure 11.

⋮
INVARIANT
$q2 \in BYs$
⋮
$\wedge \quad (q1 \mapsto q2) \in RED\_2$
$\wedge \quad err0 = \mathbf{bool}(\ (q1 \mapsto q2) \notin RED\_2\ )$

Fig. 11. Excerpt from the register example with anticipated integrity loss counter-measure

Now event *write_reg* has to update *q2* so that the invariant holds. This is easily accomplished by assigning it the image of the presented input *d1* through the redundancy constant function, *RED_2*.

The counter-measure works if, and only if, the alarm is constantly updated, i.e. it must be implemented in the combinatorial logic of the circuit. Therefore, the corresponding event in our execution model, *psi*, is refined to specify this alarm

updating as shown in figure 12.

The critical reader must have noticed that the exhibited invariant is slightly too strong as it only holds for undisrupted circuits. Indeed, as long as none of the state variables, i.e. neither *q1* nor *q2*, gets modified in an uncontrolled way, we can prove that the alarm is never set. This refinement stage allows to prove exactly that.

The next refinement stage, in fact the implementation, weakens slightly the invariant and introduces the real bus interface. Abstract constants *A_1*, *Q_1*, and *RED_2* get refined by concrete constants identical but with a suffix *_i* for valuation in the final implementation. Besides, the condition on the alarm is removed from the event's guards because the host ensures that under alarm a reset event is always generated before the next clock rising edge. An excerpt of this last refinement is shown in figure 12. An auxiliary *q3* variable is introduced to enable the translation of the expression that sets the alarm; this technique was already illustrated in the code presented in example 3.3. The output signal *ERR_FF* is just the alarm.

```
...
INVARIANT
...
∧ ERR_FF ∈ BOOL
∧     err0 = ERR_FF
∧   ( (m0 ≠ 1) ⇒
        ERR_FF = bool( (q1 ↦ q2) ∉ RED_2_i ) )
...
EVENTS
  psi =
  SELECT
    m0 = 1
  THEN
    m0 := 2
||    q3, ERR_FF :( q3 = RED_2_i(q1) ∧ ERR_FF = (bool( ¬ ( q3 = q2 ))) )
  END;
   read_reg=
  SELECT
    m0 = 2 ∧ a1 = A_1_i ∧ t1 = TyR
  THEN
    m0, PRDATA := 0, q1
  END;
  write_reg=
  SELECT
    m0 = 2 ∧ a1 = A_1_i ∧ t1 = TyW
  THEN
    m0, q1, q2 := 0, d1, RED_2_i(d1)
  END
...
```

Fig. 12. Excerpt of the robust register implementation example

As just stated, this model is not equipped to notice a disruption. Hence, the implemented function can indeed be proved correct. The main reason for weakening the invariant is to make it as similar as possible to the invariant of the companion model whose writing is explained in the following paragraphs.

4.2 Proving effectiveness

We write a not-to-be-implemented refinement of the model partially exhibited in figure 11. We enrich it with a fault model and its associated disruption event in order to prove the efficiency of the counter-measure in those precise circumstances.

For this endeavour we resort to a change of variables for the state *q1*, its redundancy *q2*, and the alarm *err0*. We also introduce an abstract witness to denote the occurrence of a disruption, *df3*. The fault model we envision is one were only one of the state variables is modified. We further impose that disruption happens before combinatorial logic gets stable. Any number of bits can be modified as long as they all belong to the same variable representation. This fault model is quite general because we do not get into the details of how the fault is injected, concentrating only on its *memorized* effects, i.e. situations where state or output are *observed* modified. We do not cover combinatorial disruptions, but we do not foresee any technical limitation to build a fault model covering them also.

The resulting refinement is partially shown in figure 13. The invariant tells us that the "sibling" implemented case, partially shown in figure 12, coincides with the state space of this faulty version if, and only if, no disruption occurs. This is precisely our intent. Do notice the designed similarity of the predicates formalizing the meaning of the alarm in figures 12 and 13.

```
⋮
INVARIANT
⋮
   df3 = bool( (q3 ↦ p3) ∉ RED_2 )
∧ (     (df3 = FALSE) ⇔
         (q1 = q3 ∧ q2 = p3 ∧ err0 = err3 ) )
∧ ( (m0 ≠ 1) ⇒
         err3 = bool( (q3 ↦ p3) ∉ RED_2 ) )
ASSERTIONS
  ( (m0 ≠ 1) ⇒ df3 = err3 )
EVENTS
  disrupt =
  SELECT
    m0 = 1 ∧ df3 = FALSE
  THEN
    q3, df3 :( q3 ∈ BYs ∧ df3 = bool(q3 ↦ p3 ∉ RED_2) )
  END;
  psi =
  SELECT
    m0 = 1
  THEN
    m0, err3 := 2, bool(q3 ↦ p3 ∉ RED_2)
  END;
  read_ok ref read_reg =
  SELECT
    m0 = 2 ∧ err3 = FALSE ∧ a1 = A_1 ∧ t1 = TyR
  THEN
    m0 := 0
  END;
  write_ok ref write_reg =
  SELECT
    m0 = 2 ∧ a1 = A_1 ∧ t1 = TyW ∧ err3 = FALSE
  THEN
    m0, q3, p3 := 0, d1, RED_2(d1)
  END
⋮
```

Fig. 13. Excerpt of the robust register example with disruption

Although they are not shown in figure 13, we introduce events *read_ko*, *write_ko*, and *none_ko* in the faulty model in order to satisfy the coverage property because now the alarm can indeed fire. The associated substitutions just start a new cycle (*m0* gets 0) and in a next refinement, these three events are merge into a single *faulty* event, a "miracle" when disruption cannot happen, i.e. in the sibling implementable

model partially shown in figure 12.

This simple example clearly shows our point: a security property can very well be, at least partially, translated into a functional requirement, overcoming thereby the refinement limitation regarding security preservation.

We successfully applied this technique to enrich the MPU case study development with the integrity property that makes inviolable the updated implementation of this control access policy, considering current state-of-the-art vulnerabilities of the underlying semiconductor technology. We added a functional redundancy in the early implementation stages, planting there for the rest of the development a new alarm to be triggered whenever redundant information looses consistency. For proof work only, a dead-end branch of the refinement introduced a "disrupt" event that can always be triggered and whose action is precisely the "becomes any value" command. Several fault models were tried in turn. Each considered fault model (single bit fault, double fault, etc.) was introduced by properly choosing the variables impacted by this "disrupt" event. We also reinforced the invariant predicates so that the proof obligations formalized the fact that no loss of consistency could fail to trigger the new alarm event. Discharging these proof obligations meant that whatever the fault on those variables impacted in the tried fault models, a correct implementation would never fail to trigger an alarm as specified. This side-way proof work established, once for all, the *effectiveness* of the counter-measure function; the regular proof work in the main development branch only dealt with the *correctness* of its implementation [7,11].

To sum up this section, we have been able to circumvent an apparent shortcoming of the proposed flow by decomposing a robustness issue into two essentially distinct parts:

(i) one model to state and prove the effectiveness of the functional description of a proposed counter-measure to thwart the effects of inflicted faults;

(ii) one model to build the proposed function in a correct-by-construction proven way, both models being refinements of the original vulnerable function.

5 A diverted use to verify an algorithm

Besides robustness, secure micro-controllers are required to be very *discreet*, noticeably when cryptographic operations are executed. Indeed, power analysis [26] provides very powerful techniques to extract secrets, usually cryptographic keys, from physical characteristics, usually power consumption, of the circuit's operation. Therefore, the design of cryptographic accelerators of secure micro-controllers must integrate counter-measures to mitigate the exploitation of these inescapable observations.

The design starts with the description of an algorithm, usually a standard one [16,15,21]. For some of these algorithms, desirable security properties are known to significantly thwart power analysis success. For instance, in the case of a symmetric key algorithm [16,15], property 5.1 has to be satisfied in the final layout of the circuit.

Definition 5.1 [Security property] Any value depending on both the input message and the key must be masked with a random.

Although correctness preservation does not entail confidentiality preservation in a development, if the detailed description of the algorithm violates the property, the final layout will violate it also with a probability close to certainty.

A third simple idea allowed us to contribute to the analysis of the detailed algorithm proposed by the designers. Observing that the required property 5.1 involves *dependencies* among values more than actual *data* values, we manually transcoded [5] the detailed algorithm using abstract interpretation [24], changing its domain from data to dependencies. The used abstraction focuses on masking and unmasking operations on sensitive data.

The transformation is illustrated in figure 14, where an operation *Op* is performed on the content *D1* and *D2* of two registers, masked [6] respectively by random numbers *m1* and *m2*. An interface *output* register, labelled `reg_o`, either gets its value from the left-hand data path, labelled `lmux`, unmasked with *m1*, i.e. value $D1 = (D1 \oplus m1) \oplus m1$, or it gets the constant value 0, according to the multiplexer setting, i.e. left or right position.

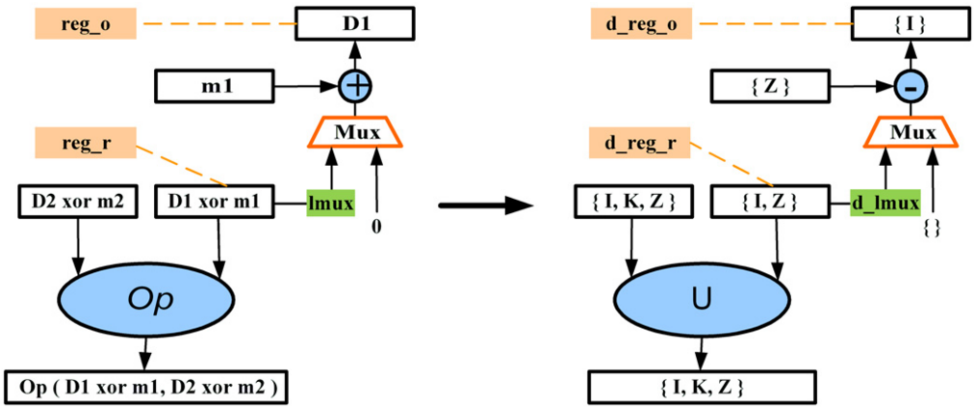

Fig. 14. From "data" to "dependencies" domains, excerpt of a detailed algorithm data path

As reported in [19], we wrote an Event-B (flat) system to represent the set of dependencies on each wire and in each register, observed after each clocked step of the detailed algorithm under analysis. We equipped this model with an invariant for each required property.

Just to provide a flavour of the resulting Event-B system, the excerpt of a detailed algorithm data path shown in figure 14 is used to write a small corresponding Event-B system. It is provided in figure 15.

Getting back to the example in figure 14, a sensitive data is a data that depends both on the input message and the key. In a first approximation, the input message and the key are abstracted by two distinct constant dependency labels, I and K [7].

[5] The rigour of this step can be improved through automation.

[6] The masking operation is a bitwise exclusive disjunction, `xor`, $\oplus$.

[7] In the Event-B system of figure 15, *D1* and *D2* are considered sensitive data on their own.

SYSTEM edp_dep
DEFINITIONS
  MASKS == {*M1*, *M2*}
; *XOR*(*A_*, *B_*) == (((*A_*) ∪ (*B_*)) - ((*A_*) ∩ (*B_*) ∩ *MASKS*))
; *SECRET* == $\mathcal{P}_1$ ({*D1*,*D2*})
; *REGDEPs* == {*dregо*, *dreg1*, *dregr*, *dreg3*}
; *SENSITIVE_ALWAYS_MASKED* == (*SECRET* ∩ *REGDEPs* = ∅)
SETS
  DEPs = {*D1*, *D2*, *M1*, *M2*}; *MUXPOS* = {*LP*, *RP*}
VARIABLES *clk*, *step*, *dregо*, *dreg1*, *dregr*, *dreg3*, *dlmux*, *muxpos*
INVARIANT
  clk ∈ **BOOL** ∧ *step* ∈ $\mathcal{Z}$ ∧ *dregо* ⊆ *DEPs* ∧ *dreg1* ⊆ *DEPs* ∧ *dregr* ⊆ *DEPs*
∧ *dreg3* ⊆ *DEPs* ∧ *dlmux* ⊆ *DEPs* ∧ *muxpos* ∈ *MUXPOS*
∧ *step* ∈ 1 . . 17
∧ *SENSITIVE_ALWAYS_MASKED*
INITIALISATION
  clk, *step* := **TRUE**, 1
|| *muxpos* :∈ *MUXPOS*
|| *dregо*, *dreg1*, *dregr*, *dreg3*, *dlmux* := ∅ , ∅ , ∅ , ∅ , ∅
EVENTS
  switch_left = **SELECT** *muxpos* = *RP* ∧ *clk* = **TRUE**
      THEN *muxpos*, *dlmux*, *clk* := *LP*, *dreg1*, **FALSE END**;
  switch_right = **SELECT** *muxpos* = *LP* ∧ *clk* = **TRUE**
      THEN *muxpos*, *dlmux*, *clk* := *RP*, ∅ , **FALSE END**;
  load_reg1 = **SELECT** *step* = 1 ∧ *clk* = **TRUE**
      THEN *dreg1*, *step*, *clk* := {*D1*,*M1*}, *step* + 1, **FALSE END**;
  load_reg2 = **SELECT** *step* = 2 ∧ *clk* = **TRUE**
      THEN *dregr*, *step*, *clk* := {*D2*,*M2*}, *step* + 1, **FALSE END** ;
  do_op_and_feed =
  SELECT
    step = 3 ∧ *clk* = **TRUE**
  THEN
    dreg3, *dregо*, *step*, *clk* := *dreg1* ∪ *dregr*, *XOR*(*dlmux*, {*M1*}), *step* + 1, **FALSE**
  END;
  update_wire_left = **SELECT** *clk* = **FALSE** ∧ *muxpos* = *LP*
      THEN *dlmux*, *clk* := *dreg1*, **TRUE END** ;
  update_wire_right = **SELECT** *clk* = **FALSE** ∧ *muxpos* = *RP*
      THEN *dlmux*, *clk* := ∅ , **TRUE END** ;
  other_steps = **SELECT** *step* > 3 ∧ *step* ≤ 16 ∧ *clk* = **TRUE**
      THEN *clk*, *step* := **FALSE**, 17 **END** ;
  do_cycle = **SELECT** *step* > 16 ∧ *clk* = **TRUE THEN** *clk*, *step* := **FALSE**, 1 **END**
END

Fig. 15. Event-B system code based on the algorithm shown in figure 14

A sensitive data is abstracted by the set:

(1) secret = $\{I, K\}$

Masks are abstracted into a unique constant dependency label, i.e. Z.

Each data operation of the algorithm is abstracted into its effect on the dependencies of the operated data. Still referring to figure 14, empirical experience shows that *Op* is such that a dependency on both its inputs can be observed in its output. It is therefore abstracted into the set union. The `xor` operation used to unmask data is abstracted into the set difference.

Each wire and each register is represented by a variable that holds its current dependencies. In figure 14, these variables are labelled with the name of the wire (resp. register) prefixed by *d_* (for dependencies).

Each step is modelled by a sequence of two events as we need to observe registers and wires. One event represents the update of registers, and the other one represents the stable update of all the outgoing wires of the registers.

The security property 5.1 becomes the following invariant (for w wires and r registers) :

(2) secret $\notin \{d\_\text{wire}_1, \ldots, d\_\text{wire}_w, d\_\text{reg}_1, \ldots, d\_\text{reg}_r\}$

We used the model checker ProB [17] to verify whether all the properties were

satisfied by the detailed algorithm. We found the model checker very efficient for this analysis because counter-examples are provided whenever a property is violated. Each step leading to the violation is clearly identified by the tool.

Referring to figure 14, ProB[8] immediately shows that when `d_reg_r` gets a dependency to a masked sensitive data and the multiplexer is set to the left, property 2 is violated because the wire input of `d_reg_o` gets the *secret* dependency.

This may seem trivial to most of us, but performing the verification on the real detailed algorithm unveiled quite tricky situations that only a very thorough analysis could have discovered. Furthermore, we were able to suggest an improvement to the detailed algorithm. We convinced the designers to implement it by showing that with the proposed modification, ProB could complete the exhaustive verification of all the properties.

Although our findings had an immediate impact on the development of the studied algorithm, this approach is not to be considered as a practical solution yet, but rather as an invitation to the research community to look at these new diverted ways of combining abstract interpretation and proof and/or model checking to tackle different facets of a complex system.

6 Concluding remarks and ways forward

Although feasibility of a correct-by-construction proven development flow for digital circuits has clearly been established through a sizeable case study, we have to temper our enthusiasm in view of the huge amount of further work required by the following issues:

- Robustness issues have been shown, through the same sizeable case study, to be amenable to a convenient functional-like treatment, but neither a systematic process nor a sound underlying theory have been defined to consider the matter settled;
- A whole part of the security of secure micro-controllers has not even been considered in the proposed flow: indeed confidentiality issues, as explained, do not behave in a conservative manner with refinement relations as the one we used. We have hinted at the use of other specialized formal methods to deal with these issues, a more precise articulation of various specialized formal methods to achieve a realistic development flow for secure elements still needs to be investigated;
- Lastly, feasibility is only a first step towards deploying a new technique. A lot of engineering, technical, economical and human problems rest ahead in order to scale up to an industrial use of the secure-by-construction development flow for digital circuits.

[8] ProB is completely integrated both in the Atelier B and in Rodin [18].

Acknowledgement

This experiment could not have succeeded without the brilliant work of Louis Mussat who developed the original Event-B models and associated proofs for the case studies. Thierry Lecomte and Antoine Requet designed and implemented the B4SYN translator and its integration into Atelier B [3]. May they read in these few lines a thankful recognition of their sizeable contribution to the project. I also wish to acknowledge the constructive remarks provided by Marie-Laure Potet on the first draft of this paper, and thank the program committee members of the workshop for their interest in this work.

References

[1] Abrial, J.-R., "Modeling in Event-B", Cambridge University Press, 2010.

[2] Abrial, J.-R., "The B Book : Assigning Programs to Meanings", Cambridge University Press, 1996.

[3] "Atelier B", http://www.atelierb.eu/index_en.html.

[4] Benveniste, M., *A Correct by Construction Realistic Digital Circuit*, RIAB Workshop, FMWeek 2009, Eindhoven, handout.pdf.

[5] Bolignano, D., *Towards a Mechanization of Cryptographic Protocol Verification*, in CAV'97, Lecture Notes in Computer Science **1254** (1997), 131–142, SpringerLink.

[6] Borrione, D., M. Liu, P. Ostier, L. Fesquet, *PSL-based online monitoring of digital systems*, in A. Vachoux (ed.) "Applications of specification and design languages for SoCs: Selected papers from FDL 2005" Chap. 2 (2006), 5–22, SpringerLink.

[7] Commission of the European Communities Directorate XIII/F SOG-IS, "Information Technology Evaluation Criteria", June 1991, ITSEC.

[8] Cuppens, F., N. Cuppens-Boulahia, T. Sans, and A. Miège, *A Formal Approach to Specify and Deploy a Network Security Policy*, in Second IFIP Workshop on Formal Aspects in Security and Trust (FAST), 203–218, 2004, [pdf].

[9] *EE Herald*, New products, 5 November 2010, [Web page].

[10] El Kamal, A.A., R. El Baida, P. Balbiani, S. Benferhat, F. Cuppens, Y. Deswarte, A. Miège, C. Sorel, and G. Trouessin, *Organization Based Access Control*, in 4th IEEE International Workshop on Policies for Distributed Systems and Networks (Policy'03), 120–134, 2003, [pdf].

[11] Jansen, W., "Directions in Security Metrics Research", National Institute of Standards and Technology Interagency Report, NISTIR 7569, Computer Security Division, Information Technology Laboratory, April 2009.

[12] Leveugle, R., A. Ammari, V. Maingot, E. Teyssou, P. Moitrel, C. Mourtel, N. Feyt, J.-B. Rigaud, and A. Tria, *Experimental Evaluation of Protections Against Laser-induced Faults and Consequences on Fault Modeling*, in DATE 07, 1587–1592, 2007, [pdf].

[13] Microcontrôleurs sécurisés SAYR48/80B et SBYR48/80B, incluant la bibliothèque cryptographique NesLib v2.0 ou v3.0 en configuration SA ou SB, ANSSI-CC-2010/02, February 2010.

[14] Milner, R., M. Tofte, R. Harper, and D. MacQueen, "The Definition of Standard ML: Revised 1997", The MIT Press, 1997, Online ML Tutorial.

[15] National Institute of Standards and Technology, U.S. Department of Commerce, "Advanced Encryption Standard (AES)", FIPS PUB 197, November 2001.

[16] National Institute of Standards and Technology, U.S. Department of Commerce, "Data Encryption Standard (DES)", FIPS PUB 46-3, October 1999.

[17] "Pro B", http://www.stups.uni-duesseldorf.de/ProB/index.php5/Main_Page.

[18] "RODIN, Rigorous Open Development Environment for Complex Systems", http://rodin.cs.ncl.ac.uk/.

[19] Romain, F., M. Benveniste, J. Mercier, *Designing a secure accelerator for symmetric cryptography*, e-Smart 2010, Day 2, Cryptographic Implementations Breakthroughs, Sophia-Antipolis, September 2010.

[20] Roscoe, A. W., J. C. P. Woodcock, and L. Wulf, *Non-interference through determinism*, in ESORICS'94, Lecture Notes in Computer Science **875** (1994), 33–53, , [pdf] of extended paper.

[21] RSA Laboratories, "PKCS #1 v2.1: RSA Cryptography Standard", PKCS #1 v2.1, June 2002.

[22] Sabelfeld, A., and A. C. Myers, *Language-Based Information-Flow Security*, in IEEE Journal on Selected Areas in Communications **21**(1) (2003), 5–19, [pdf].

[23] STMicroelectronics, "ST62T40B/E40B: 8-bit OTP/EPROM MCU with LCD driver, EEPROM and A/D converter", Data Sheet, 1999, [Web page].

[24] Wikipedia, *Abstract interpretation*, [Wiki].

[25] Wikipedia, *Météor, Paris Métro Line 14*, [Wiki].

[26] Wikipedia, *Power analysis*, [Wiki].