A Few Remarks About Formal Development of Secure Systems

Éric Jaeger
Direction centrale de la sécurité des systèmes d'information
51 boulevard de la Tour-Maubourg
75700 Paris 07 SP, France

Thérèse Hardin
LIP6, Université Pierre et Marie Curie (Paris 6)
4 place Jussieu
75252 Paris Cedex 05, France

***Abstract*—Formal methods provide remarkable tools allowing for high levels of confidence in the correctness of developments. Their use is therefore encouraged, when not required, for the development of systems in which safety or security is mandatory. But effectively specifying a secure system or deriving a secure implementation can be tricky. We propose a review of some classical 'gotchas' and other possible sources of concerns with the objective to improve the confidence in formal developments, or at least to better assess the actual confidence level.**

I. Introduction

Formal methods applied to the development of systems or software are very efficient tools that allow for high levels of assurance in the validity of the results. By defining languages with clear semantics and by making explicit how to reason on these languages, they provide a mathematical framework in which it is possible to ensure the correctness of implementations. Formal guarantees are often unreachable by more classical approaches; for example they are exhaustive whereas tests cover only a part of the possible executions.

For these reasons, the use of formal methods is encouraged, when not required, by standards for the development of systems in which safety is mandatory, e.g. *IEC 61508* [1]. The situation is similar for the development of secure systems: for the highest levels of assurance the *Common Criteria* (*CC*, [2]) require the use of formal methods to improve confidence in the development, as well as to ease the independent evaluation process. Indeed, the verification that the delivered product complies with its specification is expected to rely, at least to some extent, on a mechanically checked proof of correctness.

One should not however confuse safety with security. They are overlapping but none includes the other. Safety mostly aims at limiting consequences of random events (dealing with probabilities) and security at managing malicious actions (dealing with the difficulty of an attack). In this paper, we discuss a few concerns more specifically related to the formal development of secure systems. These concerns are illustrated through simple examples (sometimes involving a *malicious* developer) in *Coq* [3] or in *B* [4] but most of them are relevant for other deductive formal methods such as *FoCal* [5], *PVS* [6], Isabelle/HOL [7], etc.

This work is supported in part by the *Agence Nationale de la Recherche* under grant ANR-06-SETI-016 for the *SSURF* Project.

II. Formal methods

Standard development processes identify several phases such as specification, design, implementation and verification operations. Different languages can be used for different phases; beyond programming languages it is frequent to use natural language, automata, graphical languages, *UML*, etc. The problem of the correctness of a development can then be seen as a problem of *traceability* between the various descriptions of the system produced at different phases.

Formal methods considered in this paper also allow for multiple descriptions of a system; they differ from standard approaches by enforcing the use of languages with explicit and clear semantics, and by providing a logical framework to reason on them. Ensuring the correctness then becomes a mathematical analysis of the traceability (or consistency).

A. About formal specification

At least two descriptions of a system are generally considered in formal methods, a *formal specification* and an implementation. The specification is often written in a logical language (e.g. based on predicates) and is ideally declarative, abstract, high-level and possibly non-deterministic, describing the *what*. On the other hand, the implementation is imperative, concrete, low-level, and deterministic, describing the *how*. To emphasise the difference between declarative and imperative approaches, consider the specification of the integer square root function, $\sqrt{n}^2 \leq n < (\sqrt{n}+1)^2$, which is deterministic (for any n there is at most one acceptable value for $\sqrt{n}$) but is not a program: how it is computed is left to the developer.

Simply writing a formal specification is already an improvement compared to standard approaches. Indeed, by using a formal language, ambiguities are resolved. Furthermore, formal methods provide ways to check, at least partially, the consistency of the specification.

B. About refinement

The process of going from a specification to an implementation, while checking the compliance, is called *refinement*. This concept captures the activity of designing a system; it encompasses a lot of subtle activities, including choosing concrete representations for abstract data or producing operational algorithms matching declarative descriptions. From a logical point of view, a formal specification describes a family of *models* (that is, intuitively, implementations) and the refinement process consists of choosing one of those models.

Any formal method defines, implicitly or explicitly, a form of refinement. The definition generally ensures that the refinement is transitive (allowing for an arbitrary number of refinement steps in the development process) and monotone (allowing for the decomposition of a problem into several sub-problems that will be refined independently).

Formal methods do not automatically produce refinements but explain how to check that a refinement is valid, that is they ensure that very different objects (a logical description and an operational implementation) are sufficiently 'similar'. To allow for this comparison the refinement is by nature *extensional*: the objects are seen from a functional point of view, as black boxes whose only inputs and outputs are relevant. If the specification is to sort values, any sorting algorithm is valid and any two algorithms are considered equal (undistinguishable). Note that the word *intensional* is often used to refer to properties that are not extensional; for example, execution time or memory use for a sorting algorithm can be considered as intensional.

C. About logic

Behind any formal method, there is logic – or more accurately, a logic. It is not the intent of this paper to discuss at length the various types of logic, once pointed out the common fact that a specification can be inconsistent.

A specification is *inconsistent* if it is self-contradictory – a trivial example is to specify v as a natural value equal to both 0 and 1. Such a specification is also said to be *unsatisfiable*, that is it does not admit a model in the logical sense. There are three important points about inconsistent specifications:

- the detection of inconsistency cannot be automated in the general case (the problem of satisfiability is *undecidable*);
- an inconsistent specification cannot be implemented;
- an inconsistent specification can prove any property.

Tools implementing formal methods considered in this paper do not even try to detect inconsistencies (even the trivial example of $v = 0 = 1$), due to the undecidability as well as because the aim of a formal development being an implementation, any inconsistency is detected sooner or later. The last point results from the fact that for any proposition P we have $\mathbf{False} \Rightarrow P$ (using false assumptions one can prove anything). The consequences of these points are discussed in this paper.

III. A SHORT PRESENTATION OF *B* AND *Coq*

A. About B

The *B Method* [4] is a formal method widely used by both the academic world and the industry. Beyond the well-known examples of developments of safety systems (e.g. [8]), it is also recognised for security developments.

B defines a first-order predicate logic completed with elements of set theory, the *Generalised Substitution Language* (*GSL*) and a methodology of development in which the notion of refinement is explicit and central. The logic is used to express preconditions, invariants and to conduct proofs. The *GSL* allows for definitions of substitutions that can be abstract, declarative and non-deterministic (that is, specifications) as well as concrete, imperative and deterministic (that is, programs). The following example uses the non-deterministic substitution **ANY** (a 'magic' operator finding a value which satisfies a property) to specify the square root of a natural number n:

$$\mathbf{ANY}\ x\ \mathbf{WHERE}\ x^2 \leq n < (x+1)^2\ \mathbf{THEN}\ \sqrt{(n)} := x$$

The notion of refinement is expressed between *machines* (modules combining a *state* defined by variables, *properties* such as an invariant on the state and *operations* encoded as substitutions to read or alter the state) and captures the essence of program correctness w.r.t. their specification as follows: an implementation refines a specification if the user cannot exhibit a behaviour of the implementation that is not compliant with what is required by the specification. This concept is incorporated into the methodology by the automated generation of proof obligations at each refinement step, and is sustained by mathematical justifications not detailed here.

One of the characteristics of the refinement in *B* is that it is independent of the internal representation used by the machines, as illustrated by the following example of a system returning the maximum of a set of stored natural values:

$$\begin{array}{l}
\mathbf{MACHINE}\ M_A \\
\mathbf{VARIABLES}\ S \\
\mathbf{INVARIANT}\ S \subseteq \mathbb{N} \\
\mathbf{INITIALISATION}\ S := \emptyset \\
\mathbf{OPERATIONS} \\
\quad \mathit{store}(n) \triangleq \mathbf{PRE}\ n \in \mathbb{N}\ \mathbf{THEN}\ S := S \cup \{n\} \\
\quad m \leftarrow \mathit{get} \triangleq \mathbf{PRE}\ S \neq \emptyset\ \mathbf{THEN}\ m := \mathbf{max}(S) \\
\mathbf{END}
\end{array}$$

$$\begin{array}{l}
\mathbf{MACHINE}\ M_C\ \mathbf{REFINES}\ M_A \\
\mathbf{VARIABLES}\ s \\
\mathbf{INVARIANT}\ s = \mathbf{max}(S \cup \{0\}) \\
\mathbf{INITIALISATION}\ s := 0 \\
\mathbf{OPERATIONS} \\
\quad \mathit{store}(n) \triangleq \mathbf{IF}\ s < n\ \mathbf{THEN}\ s := n \\
\quad m \leftarrow \mathit{get} \triangleq m := s \\
\mathbf{END}
\end{array}$$

The state of the machines is described in the **VARIABLES** clause; for the specification M_A it is a set of natural numbers and for the implementation M_C a natural number. The **INVARIANT** clause defines a constraint over the state; for M_A it indicates that S is a subset of $\mathbb{N}$, whereas for M_C it describes the *glue* between the states of M_A and M_C (intuitively claiming that if both machines are used in parallel then s is always equal to $\mathbf{max}(S)$). The **INITIALISATION** clause sets the initial state, while the **OPERATIONS** clause details the operations used to read or alter the state. The two machines differ yet M_C refines M_A: roughly speaking one cannot exhibit a property of M_C which contradicts one of M_A.

Note the use of the **PRE** substitution defining a precondition, that is a condition that the user has to check before calling an operation. This is an *offensive* approach; an operation (should not but) can be used when this condition is not satisfied, yet in such a case there is no guarantee about the result (it may even cause a crash). By opposition the *defensive* approach is represented in *B* by using *guards* (that is an **IF**) that prevent unauthorised uses. These notions are standard in formal methods and will be discussed further later in this paper.

B. *About* Coq

Coq is a proof assistant based on a type theory. It offers a higher-order logical framework that allows for the construction and verification of proofs, as well as the development and analysis of functional programs in an *ML*-like language with pattern-matching.

Coq implements the *Calculus of Inductive Constructions* [9] and it is frequent in developments to use inductive definitions. For example, $\mathbb{N}$ is defined in the *Peano* style as follows:

$$\textbf{Inductive}\, \mathbb{N} := 0 : \mathbb{N} \mid S : \mathbb{N} \rightarrow \mathbb{N}$$

This definition means that $\mathbb{N}$ is the smallest set of terms closed under (finite number of) applications of the *constructors* 0 and S. $\mathbb{N}$ is thus made of the terms 0 and $S^n(0)$ for any finite n; being well-founded, structural induction on $\mathbb{N}$ is possible (the induction principle is automatically derived by *Coq* after the definition of $\mathbb{N}$). The definition also means that $\mathbb{N}$ contains no other values (*surjectivity*) and that $\forall\,(n:\mathbb{N}),\ 0 \neq S(n)$ and $\forall\,(m\,n:\mathbb{N}),\ S(m)=S(n) \Rightarrow m=n$ (*injectivity*).

Contrary to *B*, there is no enforced development methodology in *Coq*, nor any explicit refinement process. The user can choose between several styles of specification and implementation, and has to decide on its own about the properties to be checked. For example the *weak specification* style consists of defining functions as programs in the internal *ML*-like language and later checking properties of these functions, as illustrated here by the division by 2:

$$\begin{array}{l} \textbf{Fixpoint}\ \textit{div2}(x:\mathbb{N}):\mathbb{N} := \\ \quad \textbf{match}\ x\ \textit{with}\ S(S(x')) \rightarrow S(\textit{div2}(x')) \mid \_ \rightarrow 0\ \textbf{end}. \end{array}$$

$$\begin{array}{l} \textbf{Theorem}\ \textit{div2\_def}: \\ \quad \textit{forall}\ (x:\mathbb{N}),\ n=2*\textit{div2}(n) \vee n=2*\textit{div2}(n)+1. \\ \textbf{Proof}. \\ \quad \ldots \\ \textbf{Qed}. \end{array}$$

div2 is a recursive program (using $\textit{div2}(x+2)=\textit{div2}(x)+1$) and *div2_def* a property claimed about it; the proof, not detailed here, ensures that *div2* indeed satisfies *div2_def*.

IV. Specifying secure systems

We now begin our discussion about developing secure systems using formal methods by considering more specifically formal specifications of secure systems.

To start with trivial considerations, we first have to note that formal methods offer tools to express specifications but that there is no way to force a developer to describe the properties required of the system under development. Clearly, using even the most efficient formal method without adopting the 'formal spirit' is meaningless, as there is no benefit compared to standard approaches if the formal specification is empty. Note also that a formal development is a development, and so can also benefit from standard practices such as naming conventions, modularity, documentation, etc. In the case of formal methods, in fact, the very process of deriving a formal specification from the book of specifications should be documented, justifying the formalisation choices and identifying, if any, aspects of the system left out (as it is generally not reasonable or even feasible to aim at a full formalisation of a complete system).

Assuming a developer that has adopted the formal spirit, there are further points to care about in order to develop an 'adequate' formal specification for a secure system, that is a specification not only expressing the required properties, but also ensuring that those properties are enforced at all stages of the development as well as in any (reasonable) scenario of usage of the implementation.

Some of the concerns that will be discussed below are applicable for safety or any high assurance system; for others a malicious developer will be assumed (a threat generally irrelevant for safety but applicable in security). The ultimate objective of such a malicious developer is to exploit any weakness of a specification, in order to trap a system while delivering a mechanically checked proof of compliance. One could consider that such traps would be detected through code review or testing. Yet, beyond the fact that formal methods are expected to reduce the need for such activities, we warn the reader that our illustrations are voluntarily simplistic, and that real life examples of *Trojan Horse* are difficult to detect.

A. *About invalid specifications*

As pointed out in Par. II-C, inconsistent specifications are disastrous. Indeed, whereas inconsistency cannot be automatically detected, it also permits to discharge any proof obligation expressed – that is an inconsistent specification can in practice make the developer life more comfortable. An inconsistent specification is therefore dangerous for safety developments if a distracted developer fails to notice that its proofs are a little too easy to produce, and more so for security developments as a malicious developer identifying such a flaw would be able to prove whatever he wants.

Of course, an inconsistent specification is not implementable. It is therefore possible to check the consistency by providing an implementation – any one will do the trick, so even a dummy implementation is sufficient. Yet there are in security situations in which a formal specification is mandatory while a formal implementation is not. This is the case for the *CC*, at some assurance levels, that just require a formal specification of the *Security Policy*. An undetected inconsistent specification is therefore a possibility.

In *B* the consistency of a specification is partially checked through proof obligations to be discharged by the developer. Yet the obligations related to the existence of values satisfying the expressed constraints for parameters, variables and constants are deferred. Both following specifications are inconsistent, yet all *explicit* proofs obligations can be discharged (that is, most *B* tools will report a '100% proven' status):

$$\begin{array}{ll}
\textbf{MACHINE}\ \textit{absurd_var} & \textbf{MACHINE}\ \textit{absurd_cst} \\
\textbf{VARIABLES}\ v & \textbf{CONSTANTS}\ f \\
\textbf{INVARIANT} & \textbf{PROPERTIES} \\
\quad v \in \mathbb{N} \wedge & \quad f \in \mathbb{N} \to \mathbb{N} \wedge \\
\quad v = 0 \wedge v = 1 & \quad \forall\, x, y,\ x < y \Rightarrow f(x) > f(y) \\
\textbf{ASSERTION}\ 0 = 1 & \textbf{ASSERTION}\ 0 = 1
\end{array}$$

Of course, delaying such proof obligations is justified, as implementing the specification will force the developer to exhibit a *witness* for v that meets the specification (a constructive proof that the specification is satisfiable). Therefore, *B* ensures that any inconsistency is detected, at the latest, at the implementation stage. But we would like to remind the reader that a formally derived implementation is not always required. In such a case, one should consider additional manual verifications to check the existence of valid values for parameters, constants and variables.

Inconsistencies can be rather easy to introduce, accidentally or not, by contradicting *implicit* hypotheses associated to the used formal method. In *B* for example there is a clause **SETS** that allows for the declaration of abstract sets used in a machine; one can easily forget that such a set is always in *B* finite and non-empty. If the developer contradicts one of these implicit hypotheses the specification becomes inconsistent without any warning by the tool; in fact the automated prover will very efficiently detect the contradiction as a lemma usable to discharge any proof obligation. Contradiction of implicit principles of the underlying logic can also be illustrated in *Coq* with two very simple examples. The first one is a naive tentative of specifying $\mathbb{Z}$ using $\mathbb{N}$:

$$\begin{array}{l}
\textbf{Inductive}\ \mathbb{Z} : \textbf{Set} := \textit{plus} : \mathbb{N} \to \mathbb{Z} \mid \textit{minus} : \mathbb{N} \to \mathbb{Z}. \\
\textbf{Hypothesis}\ \textit{zero_unsigned} : \textit{plus}(0) = \textit{minus}(0).
\end{array}$$

Unfortunately, as pointed out in Par. III-B, the definition of $\mathbb{Z}$ is not a specification but an implementation ($\mathbb{Z}$ is the set of all terms of the form $\textit{plus}(n)$ or $\textit{minus}(n)$). *zero_unsigned* introduces an inconsistency because it contradicts the injectivity principle for the constructors: for any natural values n and m it is possible to prove in *Coq* that $\textit{plus}(n) \neq \textit{minus}(m)$.

The second example is related to the unexpected consequences of using possibly empty types. This is illustrated by the following (missed) attempt to define bi-colored lists of natural values, that is lists with each element marked red or blue:

$$\begin{array}{rl}
\textbf{Inductive}\ \textit{blst} : \textbf{Set} := & \textit{red} : \textit{blst} \to \mathbb{N} \to \textit{blst} \\
\mid & \textit{blue} : \textit{blst} \to \mathbb{N} \to \textit{blst}.
\end{array}$$

In the absence of an atomic constructor for the empty list, *blst* which is the smallest set of terms stable by application of the constructors is indeed empty. Therefore, assuming the existence of such a list is inconsistent, and any theorem of the form $\forall\,(b : \textit{blst}),\ P$ is provable – hardly a problem from the developer's point of view, as he generally tries to prove only those properties he expects. It would be prudent for any type T introduced in *Coq*, to ensure that it is not empty e.g. by proving a theorem of the form $\exists\,(t : T),\ \textbf{True}$.

One could also investigate the satisfiability of the preconditions or guards, as defined in Par. III-A, associated to functions or operations. Indeed, while unsatisfiable preconditions are not inconsistent, they often represent a form of deadlock, as they mean that it is never possible to use an operation. They may however be difficult to detect – there is a famous example of the database of individuals developed in [4], in which it is impossible to insert new entries, as pointed out in [10], due to the fact that any new individual introduced in the database should have a father and a mother, while the initial state is an empty database. To avoid such difficulties the use of adequate tools (animation of models, model-checking, automatic tests generator, cf. [11]–[15]) can be of considerable help.

We would also like to draw the attention of the readers to other types of problematic specifications. For example in some cases it may happen that a specification mixes predicates of the form $P \Rightarrow Q$ and $P \Rightarrow \neg Q$. Such a specification is consistent but only as long as P is false; to the least this type of specification should be considered inappropriate. This is one of the cases for which specification engineering tools would be considered useful. Such tools associate for example to a specification $\forall\, x,\ P \Rightarrow Q$ an additional proof obligation $\exists\, x,\ P$; indeed the specification can be *vacuously* true if P is always false, but it is unlikely that such a specification convey the intended meaning [16].

B. About (mis)understandings

Consequences of invalid specifications have been identified and justify establishing procedures to check consistency. We now discuss the problem of insufficient specifications, which is more tricky to detect as it generally refers to a difference between a specification and its intended meaning.

Our very first concern is related to the understanding of the chosen formal method. It is not reasonable to expect all 'users' of formal methods to be expert. One may consider for example a situation in which a customer convinced by the interest of formal methods may however not have any in-depth knowledge about any of them. In fact, we would also argue that should formal methods be more widely used – definitely something we expect for the future – they should be accessible to people having received a dedicated training but which are not expert (this is one of the main objectives of the *FoCal* project [5], [17]–[19]). The minimum, however, is to ensure that *any* user has a basic understanding of some of the underlying principles to avoid misinterpretation.

For example, consider the concept of refinement as introduced in Par. II-B. The essence of this concept is to allow to check that specifications and implementations are 'similar'. This similarity should not be too strong, as a refinement relation reduced to intensional equality of programs (that is, the same code) would be useless. It is for example standard to consider that computations and transient states are irrelevant. In *Coq* this is translated by the fact that the equality is modulo β-reduction (in other words, $\textit{square}(3) = 9$ because computing $\textit{square}(3)$ yields 9). Our concern is illustrated in *B* by the following specification of an airlock system:

$$\begin{array}{l}
\textbf{MACHINE}\ \mathit{Sas}\\
\textbf{VARIABLES}\ \mathit{door}_1, \mathit{door}_2\\
\textbf{INVARIANT}\ \mathit{door}_1, \mathit{door}_2 \in \{\mathit{open}, \mathit{locked}\} \wedge\\
\qquad\qquad\qquad \neg(\mathit{door}_1 = \mathit{open} \wedge \mathit{door}_2 = \mathit{open})\\
\textbf{OPERATIONS}\\
\quad \mathit{open}_1 \triangleq \textbf{IF}\ \mathit{door}_2 = \mathit{locked}\ \textbf{THEN}\ \mathit{door}_1 := \mathit{open}\\
\quad \mathit{close}_1 \triangleq \mathit{door}_1 := \mathit{locked}\\
\quad \mathit{open}_2 \triangleq \textbf{IF}\ \mathit{door}_1 = \mathit{locked}\ \textbf{THEN}\ \mathit{door}_2 := \mathit{open}\\
\quad \mathit{close}_2 \triangleq \mathit{door}_2 := \mathit{locked}
\end{array}$$

If the underlying principles of the *B* are not understood, one can easily consider that the **INVARIANT** clause in a proven *B* machine is 'always true'. Therefore, any compliant implementation of this specification would be considered safe. Of course, this is not the case, as we may for example refine the operation open_1 as follows:

$$\begin{array}{l}
\mathit{open}_1 \triangleq \textbf{IF}\ \mathit{door}_2 = \mathit{locked}\ \textbf{THEN}\\
\qquad\qquad \mathit{door}_1 := \mathit{open};\\
\qquad\qquad \textbf{IF}\ \mathit{attack}\ \textbf{THEN}\ \mathit{door}_2 := \mathit{open}; \mathit{wait}; \mathit{door}_2 := \mathit{locked}
\end{array}$$

where *wait* is a passive but slow operation and *attack* any condition the malicious developer can imagine to obfuscate the dangerous behaviour during tests.

If stronger forms of invariant are required, e.g. to take into account interruptions, specific modelisation choices or dedicated techniques are to be used (cf. [20]).

C. About partial specifications

Another aspect of a formal specification of a secure system to check is *totality*: is the behaviour of the system specified in any possible circumstance? It is frequent in formal methods to define partial specifications – either to represent a form of contract (a condition to be realised before having the right to use the system) or a form of freedom left to the developer (because the systems is not planned to be used in such conditions or because the result is irrelevant). If the first interpretation can be considered during formal developments, the second one becomes the only relevant one once leaving the abstract world of formal methods to tackle with implemented systems. And the extent of the freedom given to the developer is easily underestimated, as illustrated in the following examples.

We start by two specifications of the *head* function (returning the first element of a list of natural values) in *Coq*, in the strong specification style[1]:

$$\begin{array}{l}
\mathit{head}_1(l : \mathit{list}\ \mathbb{N})(p : l \neq [\,]) : \{x : \mathbb{N} \mid \exists\, l' : \mathit{list}\ \mathbb{N},\ l = x :: l'\}.\\
\mathit{head}_2(l : \mathit{list}\ \mathbb{N}) : \{x : \mathbb{N} \mid l \neq [\,] \rightarrow \exists\, l' : \mathit{list}\ \mathbb{N},\ l = x :: l'\}.
\end{array}$$

Both specifications ensure that the function, called upon a non empty list, will return the head element. Yet the first specification is associated to a precondition, the parameter p being a proof that the list parameter l is not empty – making it *impossible* to call head_1 over an empty list as it would not be possible to build such a proof. The second specification is on the contrary partial, allowing to use head_2 with an empty list but not constraining the result in such a case (except for being a natural value).

[1] In which the return value of a function is described as satisfying a property.

The point is that these two specifications are not so different: all the logical parts of a *Coq* development are eliminated at *extraction* (the process that extract proved programs). This is not specific to *Coq*: by nature, logical contents in a formal development are not computable and have therefore to be discarded in some way before being able to produce a program. And it is easy to implement both specifications in a way that produces the same following *OCaml* code, where *secret* is any value the malicious developer would care to export:

$$\textbf{let}\ \mathit{head} = \textbf{function}\ [\,] \rightarrow \mathit{secret} \mid h :: \_ \rightarrow h$$

We illustrate the same concern in *B* by the specification of a file system manager. We define the sets *USR* (users), $\mathit{Fil} \subseteq \mathit{FIL}$ (files), *CNT* (contents) and *RGT* (access rights). *Cnt* associates for any file a content, *Rgt* associates for a user and a file the rights, and *cpt* gives the number of existing files. Various operations to create, delete or access the files are assumed to be specified but are not detailed here, except for *read*:

$$\begin{array}{l}
\textbf{MACHINE}\ \mathit{filesystem}\\
\textbf{SETS}\ \mathit{USR}; \mathit{FIL}; \mathit{CNT}; \mathit{RGT} = \{r, w\}\\
\textbf{CONSTANTS}\ \mathit{cnul}\\
\textbf{PROPERTIES}\ \mathit{cnul} \in \mathit{CNT}\\
\textbf{VARIABLES}\ \mathit{Fil}, \mathit{Cnt}, \mathit{Rgt}, \mathit{cpt}\\
\textbf{INVARIANT}\ \mathit{Fil} \subseteq \mathit{FIL} \wedge\\
\qquad\qquad\qquad \mathit{Cnt} \in \mathit{Fil} \rightarrow \mathit{CNT} \wedge\\
\qquad\qquad\qquad \mathit{Rgt} \subseteq (\mathit{USR} \times \mathit{Fil}) \times \mathit{RGT} \wedge\\
\qquad\qquad\qquad \mathit{cpt} = \mathbf{card}(\mathit{Fil})\\
\textbf{INITIALISATION}\ \mathit{Fil} := \emptyset \parallel \mathit{Cnt} := \emptyset \parallel \mathit{Rgt} := \emptyset \parallel \mathit{cpt} := 0\\
\textbf{OPERATIONS}\\
\quad \ldots\\
\quad \mathit{out} \leftarrow \mathit{read}(f, u) \triangleq\\
\qquad \textbf{PRE}\ f \in \mathit{Fil} \wedge u \in \mathit{USR}\ \textbf{THEN}\\
\qquad\quad \textbf{IF}\ ((u \mapsto f) \mapsto r) \in \mathit{Rgt}\ \textbf{THEN}\ \mathit{out} := \mathit{Cnt}(f)\\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{ELSE}\ \mathit{out} := \mathit{cnul}\\
\quad \ldots
\end{array}$$

read is specified as returning the content of a file f, provided that the user u has the right to read it. Yet it is only partially specified, as we do not describe what happens when the file does not exist. Any call of *read* implemented in *B* would be associated to a proof obligation to ensure that the precondition is met, but this constraint goes as far as goes the use of the *B*. So let's assume the following malicious refinement of *read* is called over a non existing file:

$$\begin{array}{l}
\mathit{out} \leftarrow \mathit{read}(f, u) \triangleq\\
\quad \textbf{IF}\ f \in \mathit{Fil}\ \textbf{THEN}\\
\qquad \textbf{IF}\ ((u \mapsto f) \mapsto r) \in \mathit{Rgt}\ \textbf{THEN}\ \mathit{out} := \mathit{Cnt}(f)\\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{ELSE}\ \mathit{out} := \mathit{cnul}\\
\quad \textbf{ELSE}\ \mathit{Fil} := \mathit{Fil} \cup \{f_S\} \parallel\\
\qquad\qquad \mathit{Cnt} := \mathit{Cnt} \cup \{f_S \mapsto S\} \parallel\\
\qquad\qquad \mathit{Rgt} := \mathit{Rgt} \cup \{(\mathit{eni} \mapsto f_S) \mapsto r\}
\end{array}$$

Whereas the specification of *read* was *apparently* passive (not modifying the state), this refinement creates a file f_S storing a (confidential) value S, file only accessible by an arbitrary user *eni* invented by the developer. Furthermore the invariant is broken as f_S is created yet not accounted for in *cpt*, that is f_S is virtually invisible for the system. Note also that defining the returned value when the file does not exist is not even required by *B*; a malicious developer may however prefer to return *cnul* for a better obfuscation of its code.

Clearly, a partial specification cannot enforce security, and one should favor a total (and *defensive*) specification. In *B* this would translate into using a **IF** instead of a **PRE**. When the condition associated to an **IF** substitution is not satisfied, the **ELSE** branch is executed – if it is absent it is equivalent to a **skip** substitution, that is it *enforces* to do nothing. On the contrary when the condition associated to a **PRE** substitution is not satisfied, there is *absolutely no guarantee* about the result. Note that the defensive approach (with redundant checks) is an implementation of the *defence in depth* concept.

D. About elusive properties

For our next point, we would like to emphasise that some concepts often encountered in security can be difficult to express in a formal specification. Confidentiality is a good example: while a formal specification may appear to *implicitly* provide confidentiality, one should be extremely careful about its exact meaning, as illustrated by the following example of a login manager in *B*.

The system state is defined by $Acc \subseteq USR$ the accounts, *log* to identify the currently logged account (*nouser* encoding no opened session), and *Pwd* to associate to any account a password. This last piece of information is confidential and should not be disclosed. Operations (not detailed in this paper) allows to log, exit, create or destroy an account, with only the log operation specified as depending upon *Pwd* to represent the confidentiality of this data. The operation *accounts*, detailed here, returns the existing accounts:

$$
\begin{array}{l}
\mathbf{MACHINE}\ login\\
\mathbf{SETS}\ USR; PWD\\
\mathbf{CONSTANTS}\ root, nouser\\
\mathbf{PROPERTIES}\ root \in USR \wedge nouser \in USR \setminus \{root\}\\
\mathbf{VARIABLES}\ Acc, log, Pwd\\
\mathbf{INVARIANT}\\
\quad Acc \subseteq USR \wedge root \in Acc \wedge nouser \notin Acc \wedge\\
\quad log \in Acc \cup \{nouser\} \wedge Pwd \in Acc \rightarrow PWD\\
\mathbf{INITIALISATION}\\
\quad Acc := \{root\} \parallel log := nouser \parallel Pwd :\in \{root\} \rightarrow PWD\\
\mathbf{OPERATIONS}\\
\quad \ldots\\
\quad out \leftarrow accounts \triangleq\\
\quad\quad \mathbf{IF}\ log \in Acc\ \mathbf{THEN}\\
\quad\quad\quad \mathbf{ANY}\ s\ \mathbf{WHERE}\ s \in \mathbf{seq}(USR) \wedge\\
\quad\quad\quad\quad \mathbf{ran}(s) = Acc \wedge\\
\quad\quad\quad\quad \mathbf{size}(s) = \mathbf{card}(Acc)\\
\quad\quad\quad \mathbf{THEN}\ out := s\\
\quad\quad \mathbf{ELSE}\ out := \emptyset\\
\quad \ldots
\end{array}
$$

Input and output values being not refinable in *B* (cf. Par. III-A), the type of the return value of *accounts* has to be finalised in the specification. In our example, we have chosen to implement the set *Acc* returned by *accounts* as a list (or sequence in the *B* terminology) s of values of *USR*; $\mathbf{ran}(s) = Acc$ ensures that the same values appear in *Acc* and s, $\mathbf{size}(s) = \mathbf{card}(Acc)$ that the length of the list s is equal to the cardinal of *Acc*. The proposed malicious refinement of *accounts* is the following one:

$$
\begin{array}{l}
out \leftarrow accounts \triangleq\\
\quad \mathbf{IF}\ log \in Acc\ \mathbf{THEN}\\
\quad\quad \mathbf{ANY}\ s\ \mathbf{WHERE}\ s \in \mathbf{seq}(USR) \wedge\\
\quad\quad\quad \mathbf{ran}(s) = Acc \wedge\\
\quad\quad\quad \mathbf{size}(s) = \mathbf{card}(Acc)\\
\quad\quad \mathbf{THEN}\ \mathbf{IF}\ Pwd(root) < guess\\
\quad\quad\quad \mathbf{THEN}\ out := \mathbf{sort}(s)\\
\quad\quad\quad \mathbf{ELSE}\ out := \mathbf{rev}(\mathbf{sort}(s))\\
\quad \mathbf{ELSE}\ out := \emptyset
\end{array}
$$

where *guess* is a new variable controlled by the malicious developer. Combining calls to *accounts* and changes of *guess*, one can quickly derive $Pwd(root)$ through the artificial dependency introduced in the returned value.

This example illustrates a *covert channel* exploit [21], as discussed in [22]. Even if the implementation stores *Pwd* in a private memory location protected by a trusted operating system – a rather optimistic assumption – its confidentiality cannot be guaranteed without a form of control over dependencies (e.g. considering *data-flow*).

It is of course possible to impose a *complete* (or *monomorphic*) specification [23] – a deterministic specification, enforcing the extensional behaviour of any implementation. A complete specification would not let any freedom to the developer and thus would ensure that there is no *covert channel* to be exploited. In our example, a complete specification would for example require s to be sorted in ascending order. This is however an impractical technical solution, an indirect mean to ensure confidentiality. Furthermore completeness is not expressible in the *B* specification language (or in most languages considered in this paper), is generally undecidable and is *not* stable by refinement of the representation of the data – e.g. refining a set by an ordered structure.

It is also possible to better control dependencies in *B* by specifying operations using constant functions. The following modified specification claims that the operation *accounts* behaves like a function depending only upon the set *Acc* and returning a list of values of *USR*:

$$
\begin{array}{l}
\mathbf{CONSTANTS}\ \ldots, fct\\
\mathbf{PROPERTIES}\ \ldots \wedge fct \in \mathbb{P}(USR) \rightarrow \mathbf{seq}(USR)\\
\mathbf{OPERATIONS}\\
\quad out \leftarrow accounts \triangleq\\
\quad\quad \mathbf{IF}\ log \in Acc\ \mathbf{THEN}\ out := fct(Acc)\ \mathbf{ELSE}\ out := \emptyset
\end{array}
$$

This approach is not yet fully satisfactory as only the dependencies for the *result* are described (the extensional point of view). It is therefore still possible to affect the *behaviour* of *accounts*, as in this valid refinement:

$$
\begin{array}{l}
out \leftarrow accounts \triangleq\\
\quad out := encode(Pwd(root));\\
\quad \mathbf{IF}\ Pwd(root) < guess\ \mathbf{THEN}\ wait(10)\ \mathbf{ELSE}\ wait(20);\\
\quad \mathbf{IF}\ log \in Acc\ \mathbf{THEN}\ out := fct(Acc)\ \mathbf{ELSE}\ out := \emptyset
\end{array}
$$

In this refinement the malicious developer implements both a *timed channel* as well as a possibly observable transient state of the output.

This illustration is just intended to show why, in some cases, expressing confidentiality can be difficult. For such properties, complementary approaches should be considered, based e.g. on dependency calculus or non-interference [24], [25], and associated to standard code analysis. Note that confidentiality is often formally addressed through access control enforced by a form of *monitor*, that is according to the *Orange Book* a tamperproof, unavoidable, and 'simple enough to be trusted' mechanism filtering accesses (cf. recent discussions in [26]–[28]). Such a monitor can itself implement this type of *covert channel* attacks if it is poorly specified. Note also that the confidence in a system implementing a monitor relies on the confidence in the information used by this monitor, such as the source of an access request (that would require a form of *authentication*) as well as the level of protection required by the accessed object (a meta-information whose origin is generally unclear, but for which effective implementations such as *security labels* protected in integrity have been proposed).

We mention authentication and integrity to point out another source of rather elusive properties, that is the characterisation of cryptographic functions. For example, a (cryptographic) hash function H is such that:

- given h it is not possible to find x s.t. $H(x)=h$;
- given x it is not possible to find $y \neq x$ s.t. $H(x)=H(y)$;
- it is not possible to find $x \neq y$ s.t. $H(x)=H(y)$.

The first property, for example, guarantees the security of the *Unix* login scheme; being able to specify a hash function (without giving any details on its implementation) by formally describing these properties has therefore some interest to certify such a scheme. Yet these properties appear to be rather *difficult* to express formally. A naive translation of the last property would just say that H is injective, which is false (as H projects an infinite set in a finite set of binary words of fixed length) and would lead to an inconsistent specification. Formally expressing such properties is possible, but generally less straightforward than one may expect.

E. About the refinement paradox

Most of the examples detailed in Pars. IV-C and IV-D are illustrations of what is often referred to as the *refinement paradox*: some properties are preserved by refinement (safety ones generally are), other are not (security ones).

Back to the discussion of Par. IV-D, the most simple example of 'devious' refinement that we can exhibit in *B* is the following one:

$\mathbf{MACHINE}\ \mathit{Boolean}$
$\mathbf{OPERATIONS}\ \mathit{out} \leftarrow \mathit{go} \triangleq \mathit{out} := \mathit{true}\ [\!]\ \mathit{out} := \mathit{false}$

This machine is a very simple one, having no state and defining a single operation *go* returning a boolean value. There are of course two straightforward refinements:

$\mathbf{MACHINE}\ \mathit{Boolean\_True}$
$\mathbf{REFINES}\ \mathit{Boolean}$
$\mathbf{OPERATIONS}\ \mathit{out} \leftarrow \mathit{go} \triangleq \mathit{out} := \mathit{true}$

$\mathbf{MACHINE}\ \mathit{Boolean\_False}$
$\mathbf{REFINES}\ \mathit{Boolean}$
$\mathbf{OPERATIONS}\ \mathit{out} \leftarrow \mathit{go} \triangleq \mathit{out} := \mathit{false}$

Yet it also accepts other refinements, such as the following one:

$\mathbf{MACHINE}\ \mathit{Boolean\_Covert\_Channel}$
$\mathbf{REFINES}\ \mathit{Boolean}$
$\mathbf{VARIABLE}\ \mathit{dump}$
$\mathbf{INVARIANT}\ \mathit{dump} \in \mathbb{N}$
$\mathbf{INITIALISATION}\ \mathit{dump} := \mathit{private\_key}$
$\mathbf{OPERATIONS}\ \mathit{out} \leftarrow \mathit{go} \triangleq \mathbf{IF}\ \mathit{dump}\ \mathbf{mod}\ 2=0$
$\quad \mathbf{THEN}\ \mathit{out} := \mathit{true}$
$\quad \mathbf{ELSE}\ \mathit{out} := \mathit{false};$
$\quad \mathit{dump} := \mathit{dump}/2$

One should not believe that the refinement paradox is specific to those methods which are providing an explicit form of refinement, such as *B* or *Z* for example. Our devious refinements include implicitly a non functional refinement of the representation of data: we accept several implementations as representing a single abstract value of the specification. This intuitively describes why some variables are *hidden* at the specification level. From this intuition, we suggest the following counterpart in *Coq* of the refinement paradox. Let's consider the example of the specification of booleans as an *Abstract Data Type*, with the equality and a boolean function:

$\mathbf{Module\ Type}\ \mathit{Boolean\_Function}.$
$\quad \mathbf{Parameter}\ B : \mathbf{Set}.$
$\quad \mathbf{Parameters}\ \top\ \bot : B.$
$\quad \mathbf{Parameter}\ \equiv : B \to B \to \mathbf{Prop}.$
$\quad \mathbf{Hypothesis}\ \mathit{refl} : \forall\ (b : B),\ b \equiv b.$
$\quad \mathbf{Hypothesis}\ \mathit{sym} : \forall\ (b_1\ b_2 : B),\ b_1 \equiv b_2 \to b_2 \equiv b_1.$
$\quad \mathbf{Hypothesis}\ \mathit{tran} : \forall\ (b_1\ b_2\ b_3 : B),\ b_1 \equiv b_2 \to b_2 \equiv b_3 \to b_1 \equiv b_3.$
$\quad \mathbf{Hypothesis}\ \mathit{inj} : \neg \top \equiv \bot.$
$\quad \mathbf{Hypothesis}\ \mathit{surj} : \forall\ (b : B),\ b \equiv \top \vee b \equiv \bot.$
$\quad \mathbf{Parameter}\ \mathit{fnc} : B \to B.$
$\mathbf{End}\ \mathit{Boolean\_Function}.$

The straightforward refinement of this specification is of course to implement B as $\mathbb{B}$, the *Coq* type of booleans, and to implement *fnc* as one of the four possible boolean functions (*true*, *false*, *identity* or *not*). But a devious implementation gives much more freedom; we can for example choose to implement B as $\mathbb{N}$, even values representing $\bot$ and odd values representing $\top$:

$\mathbf{Module}\ \mathit{Covert\_Channel} : \mathit{Boolean\_Function}.$
$\quad \mathbf{Definition}\ B := \mathbb{N}.$
$\quad \mathbf{Definition}\ \bot := 0.\ \mathbf{Definition}\ \top := 1.$
$\quad \mathbf{Definition}\ \equiv (b_1\ b_2 : B) := (b_1 + b_2\ \mathbf{mod}\ 2 = 0).$
$\quad \ldots$
$\quad \mathbf{Definition}\ \mathit{fnc}(b : B) : B := \mathbf{match}\ ((b/2)\ \mathbf{mod}\ 4)\ \mathbf{with}$
$\quad\quad |\ 0\ \Rightarrow\ \bot$
$\quad\quad |\ 1\ \Rightarrow\ \top$
$\quad\quad |\ 2\ \Rightarrow\ b$
$\quad\quad |\ \_\ \Rightarrow\ b+1$
$\quad\quad \mathbf{end}.$
$\mathbf{End}\ \mathit{Covert\_Channel}.$

This implementation introduces a new dimension in the representation of the data, which is hidden at specification level and can be used by a malicious developer to store information and modify results: *fnc* now emulates any of the boolean functions.

Note that the term of refinement paradox may be considered an overstatement, provided the presentation of refinement in Par. II-B. Clearly the very concept of refinement is extensional, whereas on the contrary confidentiality can be considered as intensional: rather than describing *what* a result should be, it aims at constraining *how* a result is produced (in this case, without depending upon the confidential value). Similarly, if refinement is intended to preserve properties described in a specification, it does not aim at preserving properties of the specification itself, or any other form of *meta-properties*; so the fact that for example completeness is not preserved should not be a surprise.

V. Building on Sand?

In Par. IV-A, we have shown possible consequences of inconsistent specifications. Obviously similar or worse consequences can result from other sources of inconsistencies, such as a bug in the tool implementing the formal method, or a mistake in the theory of the formal method itself. For a malicious developer, a paradox (a flaw in the logic that can be used to prove at the same time both P and $\neg P$) discovered in a theory or in a tool can be used to prove any property about any development, that is to implement any unpleasant behaviour while getting a certification.

When trying to assess the level of confidence one may have in the result of a formal development, the question of the validity of the tool and of the theory should therefore be addressed.

A. About the logic

In [29] a deep embedding (cf. [30], [31]) of the *B* logic in *Coq* is described, that is intuitively a form of *B virtual machine* developed in *Coq* with the objective to check the validity of the *B* logic. While this deep embedding has not identified any paradox[2], it has shown that the following 'theorems' from [4] are in fact not provable using the defined logic:

$$\begin{array}{l} E_1{\mapsto}F_1 = E_2{\mapsto}F_2 \Rightarrow E_1 = E_2 \\ E_1{\mapsto}F_1 = E_2{\mapsto}F_2 \Rightarrow F_1 = F_2 \\ S_1 \subseteq S_2 \wedge T_1 \subseteq T_2 \Rightarrow S_1 \times T_1 \subseteq S_2 \times T_2 \end{array}$$

These results are not provable because of the definition of the *B* inference rules, which are not sufficiently precise regarding the formal definition of what is a cartesian product. To our knowledge, the fact that these results were not valid in *B* was not known by the *B* community. Being apparently trivial, they were never checked and have been integrated for example in provers for the *B* logic. That means, at a fundamental level, that these results were in fact taken as additional axioms, without people knowing it – an approach that could have created a paradox in the logic.

Further investigations have emphasised another form of subtile glitch that may appear in the theory of a formal method. As pointed out in Par. II-A, formal methods allow for multiple descriptions of a system as well as the verification of the similarity of these descriptions. This is sometimes obtained by defining several semantics for a single construct.

In *B*, substitutions of the *GSL* (used to write operations) are defined as *predicate transformers*, that is a logical semantic. On the other hand the substitutions of the *B0* sub-language are used for implementation and also have an operational semantic. This is the case of the $\mathbf{WHILE}\,P\,\mathbf{DO}\,S\,\mathbf{INVARIANT}\,I\,\mathbf{VARIANT}\,V$ substitution, illustrated in [4] by the extraction of the minimum of a non-empty set of natural values:

$$x := 0;\quad \begin{array}{l} \mathbf{WHILE}\; x \notin S \\ \quad \mathbf{DO}\; x := x+1 \\ \quad \mathbf{INVARIANT}\; x \in [0, \mathbf{min}(S)] \\ \quad \mathbf{VARIANT}\; \mathbf{min}(S) - x \\ \mathbf{END} \end{array}$$

Using the definition of the $\mathbf{WHILE}$ substitution as a predicate transformer, one can indeed show that this substitution realises (that is, transforms into a tautology) the predicate $x = \mathbf{min}(S)$. In other words the substitution is proven to extract the minimum in any case of use (provided $S \neq \emptyset$).

By denoting $[\![]\!]$ the translation producing a *C* program from a *B0* substitution, the operational semantic is defined by:

$$\left[\!\!\left[\begin{array}{l} \mathbf{WHILE}\; P \\ \mathbf{DO}\; S \\ \mathbf{INVARIANT}\; I \\ \mathbf{VARIANT}\; V \end{array} \right]\!\!\right] = \mathbf{while}\; [\![P]\!]\{[\![S]\!]\}$$

The interesting point is that this semantic forgets I (the loop invariant) and V (the loop variant) that are pure logical contents, important for the proofs (e.g. of termination) but irrelevant for the execution.

Modifying the invariant does not change the program (the operational semantic) and should therefore only have limited impact on the logical semantic. The surprise is that by replacing in the previous example the invariant $m \in [0, \mathbf{min}(S)]$ by $m \in \mathbb{N}$, less precise but still correct, the logical semantic is *radically* modified. This modified logical semantic leads to a refutation of the previous proposition, that is it indicates that the substitution is not always extracting the minimum. A rather strange conclusion, as both versions of the logical semantic describe the same program.

We have also identified a similar concern with *Coq*. In this case there is a single language, mixing logical and computational constructs, an extraction mechanism allowing for the elimination of the former to derive from the latter a program in a functional language, e.g. in *OCaml*.

As already pointed out in Par. IV-A, an inductive definition such as $\mathbf{Inductive}\; E : \mathbf{Set} := \mathit{nxt} : E \rightarrow E$ lacks an atomic constructor and is therefore empty. Emptyness is not, by itself, inconsistent but makes possible to prove any result of the form $\forall\,(e\!:\!E),\ P$. Its extraction in *OCaml* is a straightforward translation to $\mathtt{type}\; E = \mathit{Nxt}\; \mathbf{of}\; E$. The interesting point is that this *OCaml* type is *not* empty, as it contains the value $\mathbf{let\,rec}\; e = \mathit{Nxt}(e)$, not valid in *Coq* but making possible to use a program extracted from a fully certified *Coq* library with unexpected (and therefore unwanted) behaviours.

It is beyond the scope of this paper to further discuss these questions, once noted that any such bias is a potential weakness usable by a malicious developer (or a trap for an honest but inattentive developer). These remarks are not intended to criticize the tremendous work represented by the

[2]The consistency of the *B* logic has not been proved either.

full development of the theories supporting formal methods. They however justify the interest in mechanically checking such theories, pursuing works described e.g. in [32]–[34].

B. About the tools

Beyond the concerns about the theory, one may also question the validity of the tool implementing a formal method. For example a prover can be incomplete (unable to prove results valid in the theory) or incorrect (able to prove results unprovable in the theory), the latter being more worrying, at least from the evaluation and certification perspective, as it may lead to an artificial paradox. And indeed such paradoxes have been discovered in well established tools.

Clearly, implementing a formal method is a difficult task, dealing not only with completeness, correctness, but also with performance, automation, and ergonomy. In our view, the (potential) existence of bugs in a tool does not mean that it should not be used, but that the provided results should be considered with some care, and possibly verified by other mechanisms. This is addressed for example by [29], [35].

VI. Stepping Out of the Model

We have discussed at length some concerns regarding the formal development of secure systems, through questionning paradoxes in the theory, bugs in the tools or more simply by identifying gotchas in the specifications. Let's now assume that we have been able to produce a consistent specification with security properties correctly expressed, and a compliant implementation whose all proof obligations have been discharged, using a well-established formal method and a trusted tool – that is, we finally have a *proven security* system. That does *not* mean however that the system is secure, but that any attack has to contradict at least one of the hypotheses (a good heuristic for those willing to attack formally validated systems).

Preconditions, for example, are hypotheses whose violation can be devastating, as illustrated in Par. IV-C. But one should take care also to identify all the *implicit* hypotheses when developing a system or evaluating its security. Such implicit hypotheses are not only those that are introduced by the formal method (cf. Par. IV-A), but also those that are related to the modelisation choices themselves.

A. About Closure

A frequent implicit hypothesis is related to the use of closure proofs. For example, proving a *B* machine requires proving the preservation of its invariant by any of its operations. This is justified if there is no other way to influence the system state than the provided operations. The extent to which this is enforced in the real system has to be carefully analysed. Threats considered during security analysis may reflect actions that are not in the model (data stored in files by proven applications can be modified by other applications, signals in electronic circuits can be jammed by *fault injection*, etc). There is no silver bullet to address this problem; current approaches include defensive style programming, redundancy, and dysfunctional considerations (e.g. by modelling errors such as unexpected values or inconsistent states).

B. About Typing

A second example of implicit hypothesis, much less obvious, is related to types. An adequate use of types in a specification (for example modelling *IP* addresses and ports as values of abstract sets rather than natural values) ensures that some forms of error will be automatically detected (such as using a port where an address is expected). But it is also important to understand how strong an hypothesis it is, and how easily it can be violated. Indeed, types are again logical information that have generally no concrete implementation; in most programming languages, they just disappear at compilation. So, while ill typed operation calls *cannot* be considered during formal analysis, they are in some cases *executable*.

A typical example is provided in [36], describing a flaw in the *PKCS#11* API for cryptographic resources, summarised here. A central authority (e.g. a bank) distributes cryptographic resources to customers. Such a resource can perform cryptographic operations, $C \leftarrow \mathit{cipher}(M, K)$ to cipher the message M with the key numbered K, or $M \leftarrow \mathit{uncipher}(C, K)$ for the inverse operation. The resource never discloses keys to the customer, but permits exchange of keys with other resources through export of *wrapped* (cyphered) keys using $D \leftarrow \mathit{export}(K, W)$ where K is the number of the exported key and W the number of the wrapping key, and $\mathit{import}(D, W, K)$ for the inverse operation (that stores internally the unwrapped key under number K without disclosing it). In a model where cyphertexts and wrapped keys are of different types, one can prove that no sequence of calls will disclose a sensitive key. Unfortunately the implementations of cyphertexts and wrapped keys are indistinguishable, and stored keys are not tagged with their role. It is so possible to disclose a key K with the (ill-typed) sequence $D \leftarrow \mathit{export}(K, W); M \leftarrow \mathit{uncipher}(D, W)$.

This demonstrates that it is important to identify implicit hypotheses associated to the use of types to detect possible consequences of type violations, or to maintain type information in the implementation to prevent such attacks.

VII. Conclusion

We summarise and discuss difficulties related to the development of secure systems using formal methods, identifying – where possible – proposals for improvement. The concerns described in this paper were identified during a systematic review of the process of formal development, investigating possible difficulties.

A quick read of this paper could seem to imply that the reputation of formal methods to develop correct systems is overestimated. *This is not our message*. We consider that formal methods are very efficient tools to obtain high level of assurance and confidence for the development of systems in general, and of secure systems in particular.

Yet to fully benefit from such tools, one has to understand their strengths but also their limitations. Pretending that proven secure systems are perfectly secure is nothing more than a renewed version of the first myth about formal methods pointed out in [37], and is to the least inadequate; in fact, we consider that such a claim is detrimental to formal methods. Taking this into account, we expect our proposals to help, where possible, for improving the quality of formal specifications and the adequacy of formal developments of secure systems (in some cases relying on other methodologies or technologies); our second expectation is to shed some light on the difficulties to at least allow for a better evaluation of the genuine level of confidence obtained through the use of formal methods.

Nota: An extended version of this paper is available in French language at [38].

REFERENCES

[1] IEC 61508, "Functional safety of electrical, electronic, programmable electronic safety-related systems," (www.iec.ch/zone/fsafety).

[2] ISO/IEC 15408, "Common criteria for information technology security evaluation," (www.commoncriteriaportal.org).

[3] The Coq development team, *The Coq proof assistant reference manual*, LogiCal Project, 2004. [Online]. Available: http://coq.inria.fr

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

[5] The FoCal development team, "The Focal project," (focal.inria.fr).

[6] S. Owre, J. M. Rushby, , and N. Shankar, "PVS: A prototype verification system," in *11th International Conference on Automated Deduction (CADE)*, ser. Lecture Notes in Artificial Intelligence, D. Kapur, Ed., vol. 607. Saratoga, NY: Springer-Verlag, jun 1992, pp. 748–752. [Online]. Available: http://www.csl.sri.com/papers/cade92-pvs/

[7] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, ser. Lecture Notes in Computer Science. Springer, 2002, vol. 2283.

[8] P. Behm, P. Desforges, and J. M. Meynadier, "MÉTÉOR : An industrial success in formal development." in *B '98: Proceedings of the Second International B Conference on Recent Advances in the Development and Use of the B Method*, ser. Lecture Notes in Computer Science, D. Bert, Ed., vol. 1393. Springer, 1998, p. 26.

[9] T. Coquand and C. Paulin, "Inductively defined types," in *Conference on Computer Logic*, ser. Lecture Notes in Computer Science, P. Martin-Löf and G. Mints, Eds., vol. 417. Springer, 1988, pp. 50–66.

[10] L. Mussat, 2005, private Communication.

[11] M. Carlier and C. Dubois, "Functional testing in the focal environment," in *Test And Proof (TAP'2008)*, B. Berckert and R. Hahnle, Eds., vol. 4966. LNCS, 2008, pp. 84–98.

[12] E. Jaffuel and B. Legeard, "LEIRIOS test generator: Automated test generation from B models." in *The 7th International B Conference*, ser. Lecture Notes in Computer Science, J. Julliand and O. Kouchnarenko, Eds., vol. 4355. Springer, 2007, pp. 277–280.

[13] E. M. Clarke, O. Grumberg, and D. E. Long, "Model checking," in *NATO ASI DPD*, M. Broy, Ed., 1996, pp. 305–349.

[14] D. Plagge and M. Leuschel, "Validating Z specifications using the ProBAnimator and model checker," in *IFM*, ser. Lecture Notes in Computer Science, J. Davies and J. Gibbons, Eds., vol. 4591. Springer, 2007, pp. 480–500.

[15] Y. Yu, P. Manolios, and L. Lamport, "Model checking TLA$^+$ specifications," in *CHARME*, ser. Lecture Notes in Computer Science, L. Pierre and T. Kropf, Eds., vol. 1703. Springer, 1999, pp. 54–66.

[16] M. Samer and H. Veith, "On the notion of vacuous truth," in *LPAR*, ser. Lecture Notes in Computer Science, N. Dershowitz and A. Voronkov, Eds., vol. 4790. Springer, 2007, pp. 2–14.

[17] C. Dubois, T. Hardin, and V. V. Donzeau Gouge, "Building certified components within FOCAL," in *Trends in Functional Programming*, H.-W. Loidl, Ed., vol. 5. Bristol, UK: Intellect, 2004, pp. 33–48.

[18] D. Doligez, T. Hardin, and V. Prévosto, "Algebraic structures and dependent records," in *Proceedings of TPHOL'03*, B. Rick, Ed., NASA, Hampton, USA, 2003.

[19] V. Prévosto and M. Jaume, "Making proofs in a hierarchy of mathematical structures," in *11th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning, Calculemus 2003*. Aracne, 2003, pp. 89–100.

[20] J. Andronick, B. Chetali, and C. Paulin-Mohring, "Formal verification of security properties of smart card embedded source code," in *FM*, ser. Lecture Notes in Computer Science, J. Fitzgerald, I. J. Hayes, and A. Tarlecki, Eds., vol. 3582. Springer, 2005, pp. 302–317.

[21] B. W. Lampson, "A note on the confinement problem," *Commun. ACM*, vol. 16, no. 10, pp. 613–615, 1973.

[22] J. A. Clark, S. Stepney, and H. Chivers, "Breaking the model: Finalisation and a taxonomy of security attacks." [Online]. Available: citeseer.ist.psu.edu/clark04breaking.html

[23] A. Schönegge, "Proof obligations for monomorphicity." [Online]. Available: citeseer.ist.psu.edu/27234.html

[24] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke, "A core calculus of dependency," in *POPL '99. Proceedings of the 26th ACM SIGPLAN-SIGACT on Principles of programming languages, January 20–22, 1999, San Antonio, TX*, ACM, Ed. New York, NY, USA: ACM Press, 1999, pp. 147–160. [Online]. Available: citeseer.ist.psu.edu/abadi99core.html

[25] J. Goguen and J. Meseguer, "Security policies and security models," in *IEEE Symposium on Security and Privacy*. IEEE Press, 1992, pp. 11–20.

[26] N. Benaïssa, D. Cansell, and D. Méry, "Integration of security policy into system modeling," in *B*, ser. Lecture Notes in Computer Science, J. Julliand and O. Kouchnarenko, Eds., vol. 4355. Springer, 2007, pp. 232–247.

[27] A. Haddad, "Meca: A tool for access control models," in *B*, ser. Lecture Notes in Computer Science, J. Julliand and O. Kouchnarenko, Eds., vol. 4355. Springer, 2007, pp. 281–284.

[28] S. Hoffmann, G. Haugou, S. Gabriele, and L. Burdy, "The B-Method for the construction of microkernel-based systems," in *B*, ser. Lecture Notes in Computer Science, J. Julliand and O. Kouchnarenko, Eds., vol. 4355. Springer, 2007, pp. 257–259.

[29] É. Jaeger and C. Dubois, "Why would you trust B ?" in *LPAR*, ser. Lecture Notes in Computer Science, N. Dershowitz and A. Voronkov, Eds., vol. 4790. Springer, 2007, pp. 288–302.

[30] M.J.C. Gordon, "Mechanizing programming logics in higher-order logic," in *Current Trends in Hardware Verification and Automatic Theorem Proving (Proceedings of the Workshop on Hardware Verification)*, G.M. Birtwistle and P.A. Subrahmanyam, Eds. Banff, Canada: Springer-Verlag, Berlin, 1988, pp. 387–439. [Online]. Available: citeseer.ist.psu.edu/gordon88mechanizing.html

[31] A. Azurat and I. Prasetya, "A survey on embedding programming logics in a theorem prover," Institute of Information and Computing Sciences, Utrecht University, Tech. Rep. UU-CS-2002-007, 2002.

[32] P. Chartier, "Formalisation of B in Isabelle/HOL," in *B '98: Proceedings of the Second International B Conference on Recent Advances in the Development and Use of the B Method*, ser. Lecture Notes in Computer Science, D. Bert, Ed., vol. 1393. London, UK: Springer-Verlag, 1998, pp. 66–82.

[33] J.-P. Bodeveix, M. Filali, and C. Muñoz, "A formalization of the B-Method in Coq and PVS," in *Electronic Proceedings of the B-User Group Meeting at the World Congress on Formal Methods FM 99*, 1999, pp. 33–49.

[34] B. Barras, "Auto-validation d'un système de preuves avec familles inductives," Thèse de Doctorat, Université Paris 7, Nov. 1999.

[35] T. Ridge and J. Margetson, "A mechanically verified, sound and complete theorem prover for first order logic." in *TPHOLs*, ser. Lecture Notes in Computer Science, J. Hurd and T. F. Melham, Eds., vol. 3603. Springer, 2005, pp. 294–309.

[36] J. Clulow, "On the security of PKCS#11," in *CHES*, ser. Lecture Notes in Computer Science, C. D. Walter, Çetin Kaya Koç, and C. Paar, Eds., vol. 2779. Springer, 2003, pp. 411–425.

[37] A. Hall, "Seven myths of formal methods," *IEEE Software*, vol. 07, no. 5, pp. 11–19, 1990.

[38] DCSSI, "Central directorate for information systems security, publications page," (http://www.ssi.gouv.fr/fr/sciences/publications.html).

[39] D. Bert, Ed., *B'98: Recent Advances in the Development and Use of the B Method, Second International B Conference, Montpellier, France, April 22-24, 1998, Proceedings*, ser. Lecture Notes in Computer Science, vol. 1393. Springer, 1998.

[40] N. Dershowitz and A. Voronkov, Eds., *Logic for Programming, Artificial Intelligence, and Reasoning, 14th International Conference, LPAR 2007, Yerevan, Armenia, October 15-19, 2007, Proceedings*, ser. Lecture Notes in Computer Science, vol. 4790. Springer, 2007.

[41] J. Julliand and O. Kouchnarenko, Eds., *B 2007: Formal Specification and Development in B, 7th International Conference of B Users, Besançon, France, January 17-19, 2007, Proceedings*, ser. Lecture Notes in Computer Science, vol. 4355. Springer, 2006.