Formal verification of tamper-evident storage for e-voting

Dominique Cansell, Paul Gibson, Dominique Méry

Formal verification of *tamper-evident* storage for e-voting

Dominique Cansell
LORIA & Université de Metz
cansell@loria.fr

J. Paul Gibson
INT Evry, Paris
paul.gibson@int-evry.fr

Dominique Méry *
Nancy-Université, Université Henri Poincaré Nancy1 & LORIA
mery@loria.fr

Abstract

The storage of votes is a critical component of any voting system. In traditional systems there is a high level of transparency in the mechanisms used to store votes, and thus a reasonable degree of trustworthiness in the security of the votes in storage. This degree of transparency is much more difficult to attain in electronic voting systems, and so the specific mechanisms put in place to ensure the security of stored votes require much stronger verification in order for them to be trusted by the public. There are many desirable properties that one could reasonably expect a vote store to exhibit. From the point of view of security, we argue that tamper-evident storage is one of the most important requirements: the changing, or deletion of already validated and stored votes should be detectable; as should the addition of unauthorised votes after the election is concluded. We propose the application of formal methods (in this paper, event-B) for guaranteeing, through construction, the correctness of a vote store with respect to the requirement for tamper-evident storage. We illustrate the utility of our refinement-based approach by verifying — through the application of a reusable formal design pattern — a store design that uses a specific PROM technology and applies a specific encoding mechanism.

1. Introduction

1.1. Motivation: the e-voting problem

Computer technology has the potential to modernise the voting process and to improve upon existing systems; but it also introduces new concerns with respect to secrecy, accuracy, trust and security[8]. The debate over e-voting is not a new one — recent use of such systems in actual elections has led to their analysis from a number of different viewpoints: usability[9], trustworthiness and safety criticality[14], transparency and openness[15], and risks and threats[19].

The potential advantages are generally accepted, for example: faster result tabulation, elimination of human error which occurs in manual vote tabulation, assistance to voters with "special" needs, defence against fraudulent practices (e.g. with postal votes[3]), and improving the "fairness" of count systems that incorporate "unfair" non-deterministic procedures[23].

Despite ever-increasing uncertainty over the trustworthiness of these systems — which is one of the major disadvantage associated with them — many countries (particularly in Europe[22]) have recently chosen to adopt e-voting. The main risks that have been clearly identified seem not to concern those responsible for procuring the systems. In fact, it appears that e-voting is just one, well-publicised, example of governments wishing to adopt new technologies[20] before the risks and benefits, as perceived by the public[10], have been properly analysed and debated.

This paper contributes to answering two important questions: firstly, whether the public's mistrust in the security of e-voting systems is well-founded; and secondly, whether formal methods have a role to play in addressing the problem of mistrust. With respect to the first aspect, Kocher and Schneier[12] state: "The threats are real, making openness and verifiability critical to election security." As to the second aspect, expecting the public to trust the adoption of such new technology is only possible if they can be convinced that it rests on firm trustworthy foundations. The formal methods community have an important role to play in this respect: the use of formal notations provides a fundamental foundation upon which the complexities of ever-changing technologies can be managed[7]. We argue that

*Works of Dominique Cansell and Dominique Méry are supported by grant No. ANR-06-SETI-015-03 awarded by the Agence Nationale de la Recherche.

without the adoption and promotion of formal methods as the foundations of software engineering, developing trustworthy e-voting systems will not necessarily guarantee that they will be trusted.

This paper proposes that, in general, already existing formal techniques can help to alleviate many of the verification problems that the adoption of new e-voting technologies can introduce. For the specific modelling and verification in our study we use the event-B method[1], based on the B notation. We argue that it is unreasonable to expect the public to trust a system (or part of a system) to behave correctly just because it is developed using a formal method (like event-B)[1]. Instead, we propose that we must first establish a set of quality standards for reliable, re-usable, trustworthy tools and techniques that have proven themselves in the formal development of correct systems. Then, provided the public are properly informed, it is not unreasonable to expect systems built in this way to be both more trustworthy *and* more trusted. The correct-by-construction approach in this paper illustrates the type of standard process to which we are alluding.

1.2. Formal methods and vote storage

Public opinion, arising from detailed debate of the issues, would suggest that for e-voting machines to be acceptable they should be developed following best practice with regards to the engineering of critical systems. Media reports would also suggest that the secure storage of votes is one of the issues that is most mistrusted by the public.

We propose the use of formal methods as a means of ensuring that a machine securely stores votes, and we propose to demonstrate the utility of formal methods through guaranteeing simple safety properties of a voting machine store. The main property that we examine is concerned with the need for *tamper-evident* storage, which addresses the risk of unauthorised tampering of vote data after it has been correctly registered and stored. In *Analysis of an electronic voting system*[13], we see that such a security weakness already exists in one of the most widely procured voting systems:

> "...an adversary could alter election results by modifying ballot definition files, and ...it leaves no evidence that an attack was ever mounted"

Here, the "adversary" is most likely to be a single insider (election official) with access to the storage device. We argue that it is the responsibility of the storage designers to guarantee the security of the votes stored without having to make an assumption about the behaviour or intent of such officials.

In order to illustrate how a guarantee could be made, we use event-B and apply an incremental refinement approach to verifying a sequence of designs for the storage of votes, which we prove to be correct-through-construction with respect to the simple requirement that the vote storage is tamper-evident.

1.3. Refinement and genericity: a formal design pattern

>From a technological viewpoint we know that system design has an important role in security assurance. Mercuri[17] addresses the theme of quality in the process of engineering security:

> "By encouraging artistry and applying craftsmanship to our security problems, viable solutions will emerge."

This supports our view that one must start with a simple model of the vote store requirements and refine that model, during design, towards a correct implementation. For this reason we chose a simple security requirement — that only valid votes can be found in the vote store and that these cannot be tampered with without detection — and start our formal development from there. The use of formal methods to guarantee that only valid votes are passed from the machine interface to the store has already been presented[5]. The work presented in this paper, which addresses the tamper-evident requirement, is complementary in nature: event-B is the common modelling language, and correctness through construction is the common formal design approach.

As one of the long-term goals of the formal methods community is to simplify the verification process for engineers[4, 11], we support the view that re-usable verification design patterns, similar in nature to the work by Mehlitz and Penix[16], as a potential solution to this problem. This paper identifies a good candidate for such a re-usable pattern, combining genericity and refinement to provide a *correct-by-construction pattern* (see Section 4).

2. Manchester Encoding: formalising the design of a secure vote PROM

The main design that is modelled and verified in this paper is taken directly from the work by Molnar, Kohno, Sastry and Wagner[18]. Their proposed solution to providing tamper-evident storage involves the application of Manchester codes[21] and a write-once data PROM store. The encoding simply represents a 0 as a 01 and a 1 as a 10. Thus, when validating votes stored as pairs of bits there are

[1] This is analagous to the current common situation where the public have been asked to trust the e-voting machines because they have been independently tested by some appropriately accredited body. Experience now shows that such trust was misplaced.

2 additional pair cases to be considered, where (because our memory allows only 1s to be overwritten as 0s): 11 corresponds to unwritten memory and 00 corresponds to an invalid memory that has been tampered with.

Before we formally specify and verify the proposed solution, we briefly note that there is a real pragmatic need for *tamper-evident* rather than *tamper-proof* writeable storage. The *tamper-proof* requirement can be met only by some security mechanism ensuring authorised-only update of the vote store. This security mechanism would probably be implemented as some combination of physical constraints, together with hardware and software checks. It would most likely involve some complex encryption technique and it is not clear whether one could, or should, expect voters to trust such a complex system. Contrastingly, guaranteeing the *tamper-evident* requirement is a much simpler problem that — if done well — could be both trustworthy and trusted.

Implementing storage using a write-once data store has many obvious advantages when we consider tampering: obviously, any vote that has already been written cannot be overwritten? In fact, without a more formal model of the store, this is not guaranteed to be true. For example, one form of write-once storage could allow the flipping of an initial bit state to be done once and once only. This does not necessarily guarantee that a recorded vote cannot be overwritten as individual bits of a vote will not have been flipped when a vote is recorded. In fact, as with all storage mechanisms, the (encoding) protocol used for writing information to such a store will be the deciding factor in whether the tampering requirements are met. Furthermore, there are many reasonable variations of the tampering requirement. Without a precise statement, it is not clear whether we will be able to verify whether a given system (the store properties, together with the encoding protocol) is correct.

The key property of the encoding that we shall model is that if any (sub)set of 1 bits in a stored codeword are flipped to 0s then the result is no longer a valid code word. We then wish to establish that anyone with read access to the voting store can detect an invalid memory state, where at least one codeword is invalid, and consequently any tampering after[2] the election has been completed. The verification of this safety property requires modelling of the write-once behaviour in the chosen PROM implementation (checking that 1s can be re-written as 0s but that 0s cannot be changed) in conjunction with the encoding mechanism. It also requires the use of a special *election over* bit (bit pair in PROM) to signify that the election is over, and which must be unset and untampered with for new votes to be recorded (otherwise anyone with access to the voting machine could add unauthorised votes after the election, an attack known as ballot stuffing). We chose not to include the *election-over* behaviour in the model presented in this paper.

We note that this system is not tamper proof: attackers with write access to the vote store can still invalidate the election by overwriting vote data. However, this attacks would be easily identified by procedures for validating the storage state during and after the vote.

The main advantages of doing this design formally, in event-B, are development oriented:

- an abstract model can be easily validated as correctly expressing the requirement,
- the actual design model can be constructed incrementally through refinement of the abstraction,
- the refinement process can continue through to modelling at very fine grain levels of detail that correspond to the chosen low-level implementation architecture,
- we can more easily reason about different variations and combinations of encodings and storage media, and
- we can analyse possible problems of integrating this requirement with other requirements of the e-voting system, in general, and the vote storage, in particular.

Thus, we are more likely to develop a trustworthy storage.

A secondary benefit arises when we consider the issue of how to build public trust in our formally developed trustworthy system. We argue that the *correct-by-construction* technique, embodied in a reusable design pattern, will become more and more trusted as it is used to develop more and more systems that prove themselves to be trustworthy. As a consequence, using such a standard technique (and associated tools) in constructing critical systems will increase confidence in the systems' correctness, from both the developers and the public users.

With tool support for automatically checking our verification proof we have another advantage: if our proof tool is trustworthy then the design is sure to be correct provided the property that we have established, in the initial abstract model, is an accurate statement of the high level requirement. To make this transparent to the users (voters) it is essential that an initial abstract model is easy to understand and validate, and that they have good reason not to mistrust our proof tool and techniques. Our design approach facilitates this type of openness and transparency.

3. Overview of event-B development by stepwise refinement

3.1. Event-based modelling

Our event-driven approach [1] is based on the B notation. It extends the methodological scope of basic concepts in or-

[2] It is trivial to extend our model to dynamically detect tampering during an election but for simplicity and conciseness we do not present details of this variation of tamper-evidence in this paper.

der to take into account the idea of *formal models*. Roughly speaking, a formal model is characterized by a (finite) list x of *state variables* possibly modified by a (finite) list of *events*; an invariant $I(x)$ states properties that must always be satisfied by the variables x and *maintained* by the activation of the events. In the following, we briefly recall definitions and principles of formal models and explain how they can be managed by tools [2, 6].

Generalized substitutions are borrowed from the B notation. They provide a means to express changes to state variable values. In its simple form, $x := E(x)$, a generalized substitution looks like an assignment statement. In this construct, x denotes a vector built on the set of state variables of the model, and $E(x)$ a vector of expressions. The interpretation we shall give here to this statement is not however that of an assignment statement. We interpret it as a *logical simultaneous substitution* of each variable of the vector x by the corresponding expression of the vector $E(x)$. There exists a more general normal form of this, denoted by the construct $x : P(x_0, x)$. This should be read: "x is modified in such a way that the predicate $P(x_0, x)$ holds", where x denotes the *new value* of the vector and x_0 denotes its *old value*. This is clearly non-deterministic in general.

In the following, the so-called before-after predicate $BA(x, x')$ describes an event as a logical predicate expressing the relationship linking the values of the state variables just before (x) and just after (x') the "execution" of event evt[3]. Each event has two main parts: a *guard*, which is a predicate built on the state variables, and an *action*, which is a generalized substitution.

Proof obligations are produced from events in order to state that an invariant condition $I(x)$ is preserved. Their general form follows immediately from the definition of the before-after predicate, $BA(x, x')$, of each event:

$$I(x) \ \wedge \ BA(x, x') \ \Rightarrow \ I(x')$$

Note that it follows from the two guarded forms of the events that this obligation is trivially discharged when the guard of the event is false.

3.2. Model Refinement

The refinement of a formal model allows us to enrich a model in a *step-by-step* approach, and is the foundation of our *correct-by-construction* approach. Refinement provides a way to strengthen invariants and to add details to a model. It is also used to transform an abstract model into a more concrete version by modifying the state description. This is done by extending the list of state variables, by refining each abstract event into a corresponding concrete version, and by adding new events. The abstract state variables, x, and the concrete ones, y, are linked together by means of a, so-called, *gluing invariant* $J(x, y)$. A number of proof obligations ensure that (1) each abstract event is correctly refined by its corresponding concrete version, (2) each new event refines $skip$, (3) no new event takes control for ever, and (4) relative deadlock-freeness is preserved. Details of the formulation of these proofs follows.

We suppose that an abstract model AM with variables x and invariant $I(x)$ is refined by a concrete model CM with variables y and gluing invariant $J(x, y)$. If $BAA(x, x')$ and $BAC(y, y')$ are respectively the abstract and concrete before-after predicates of the same event, we have to prove the following statement, corresponding to proof obligation (1):

$$\begin{array}{l} I(x) \ \wedge \ J(x, y) \ \wedge \ BAC(y, y') \\ \Rightarrow \ \exists x' \cdot (BAA(x, x') \ \wedge \ J(x', y')) \end{array}$$

Now, proof obligation (2) states that $BA(y, y')$ must refine $skip$ ($x' = x$), generating the following simple statement to prove (2):

$$I(x) \ \wedge \ J(x, y) \ \wedge \ BA(y, y') \ \Rightarrow \ J(x, y')$$

For the third proof obligation, we formalise the notion of the system advancing in its execution; a standard technique is to introduce a variant $V(y)$ that is decreased by each new event (to guarantee that an abstract step may occur). This leads to the following statement to prove (3):

$$I(x) \ \wedge \ J(x, y) \ \wedge \ BA(y, y') \ \Rightarrow \ V(y') < V(y)$$

Finally, to prove that the concrete model does not introduce additional deadlocks, we give formalisms for reasoning about the event guards in the concrete and abstract models: $\mathsf{grds}(AM)$ represents the disjunction of the guards of the events of the abstract model, and $\mathsf{grds}(CM)$ represents the disjunction of the guards of the events of the concrete model. Relative deadlock freeness is now easily formalised as the following proof obligation (4):

$$I(x) \ \wedge \ J(x, y) \ \wedge \ \mathsf{grds}(AM) \ \Rightarrow \ \mathsf{grds}(CM)$$

To review, refinement guarantees that the set of traces of the refined model contains (modulo stuttering) the traces of the resulting model.

4. Genericity and refinement in event-B: a formal design pattern

The event-B method provides a framework for developing generic models of systems, where a *problem* can be defined using parameters to be instantiated. Intuitively, this

[3] The prime notation, where we represent the value of a variable x, say, after an event by x' is a fundamental part of the modelling language and is used throughout all the models that follow.

means that we are able to relate the current *problem* to be solved to an abstract *problem* solved by an already existing generic B development (theory). Following our approcah, in the existing generic solution we must find the mathematical framework that is common to both *problems*, together with some constants which need to be instantiated. Consequently, in formulating the solution to the new *problem* the main work is to check that the instantiated parameters satisfy the constraints of the generic *problem* theory.

4.1. Projects

The development of a fully formal generic modelling mechanism for the B event-based method is work in progress. In the following, we indicate how the current framework can be used for implementing the instantiation.

The key concept is that of a validated project: a collection of models, either machine or refinement or implementation, which are completely verified through type checking and theorem proving. For simplicity, and without loss of generality, we assume that there is only one machine in the current project, allowing us to focus on the re-usability of developed models. Hence, a project $\mathbb{G}$ is roughly speaking an acyclic directed graph of models related by the refinement relationship: $\mathbb{G} = (\{G, ..., G_n\}, \longrightarrow)$.

4.2. The General Model

In the following, in order to avoid confusion among names, we use different fonts for designating problems, models and projects. The creation and the development of the project $\mathbb{G}$ follows the event B methodology. We assume that $\mathbb{G}$ is an existing project corresponding to a given *generic* problem, denoted $\mathcal{G}$. The model G (see the template specification on the left of figure 1) is, in fact, the formal statement of the generic problem: it incorporates relevant aspects of the generic problem — at a high level of abstraction — in an initial model. This initial abstraction can be thought of as defining the scope of the problem and the behavioural properties that require validation.

The model G provides a general framework for the current problem; the problem is characterized by a theory defined by the clauses SETS, CONSTANTS and PROPERTIES. The unique $event$ helps in solving the problem by defining the problem in an abstract form: saying what is required rather than how the solution is to be implemented. Intuitively this corresponds to the problem being viewed as pre/post-condition relation between an initial state and a final state which is arrived at after executing the single $event$. Of course, refinement permits us to move from this "magical" one-step functional view of the required system's behaviour to a richer multi-step view.

MODEL G SETS s CONSTANTS c PROPERTIES $P(s, c)$ VARIABLES x INVARIANT $J(x)$ ASSERTIONS $A(x)$ INITIALISATION $S(x)$ EVENTS $event = L(x)$ END	MODEL H SETS t CONSTANTS d PROPERTIES $Q(t, d)$ VARIABLES y INVARIANT $I(y)$ ASSERTIONS $C(y)$ INITIALISATION $spec\_init(y)$ EVENTS $spec\_event = K(y)$ END

Figure 1. Definition of models

The generic model G is the starting point of the development of the project $\mathbb{G}$: it solves the problem $\mathcal{G}$; and the project is formally checked by the theorem prover. Constants in G can be instantiated, but proof obligations must be established to ensure the validity of properties in the instantiated model, which correspond to theorems in an instantiation.

Before we introduce the instantiation and refinement steps in our design pattern, we motivate the need for such a pattern. In the general *correct-by-construction* refinement-based development process, working with concrete models often leads to refinements generating large numbers of proof obligations that cannot be discharged automatically. The main goal during development is to find a *good* refinement path: a short sequence of refinement steps where a small number of proof obligations are generated at each step and which are discharged automatically. Finding such a path usually requires reformulating or restructuring of invariants, together with changes to the degree of detail (abstraction) in the models, and is very challenging for non-experts. Thus, we would like to package such expertise in a re-usable construct, which we call a formal design pattern.

In order to manage the complexity of the refinement path, one common approach (used by experts) is to find a more generic representation of their initial problem where details are hidden by constants requiring instantiation. Then, in general, the refinement path is much simpler to establish as it requires fewer interactive proofs and leads to a correct concrete generic solution. To prove that this generic

solution can be re-used, through instantiation, to solve the initial concrete problem, two final steps are required. First, one must show that the initial problem model is a correct instantiation of the generic problem model; and secondly, one must show that the final solution model is a correct instantiation of the generic solution model.

In the best case, the generic refinement path has already been established and can be re-used directly. In the worst case, this path has to be developed from scratch. However, even in this worst case, developing the path at a higher level of abstraction (and proving 2 instantiations to be correct) is much easier[4] than developing the path at the lower level of abstraction.

Thus, our formal design pattern is a re-usable solution to a common design problem that can be exploited by formal developers who are not necessarily expert. This re-use requires only that the developers understand instantiation and refinement.

4.3. Instantiation

Consider a specific problem $\mathcal{H}$ in project $\mathbb{H}$, say. We specify the specific requirements as a new model H (see the template specification on the right of figure 1). In order to exploit our design pattern, we wish to establish that an instantiation of the generic project $\mathbb{G}$ corresponds to the given problem $\mathcal{H}$;

The instantiation of the generic project $\mathbb{G}$ for the generic problem $\mathcal{G}$ to solve the specific problem $\mathcal{H}$ consists of exhibiting a set term $\sigma(t, d)$, defined in terms of the set t and constant of d, and also a similar constant term $\gamma(t, d)$ for instantiating the constant c of G. Thus, the instantiation consists of repainting G with $\sigma(t, d)$ and $\gamma(t, d)$ and to invoke it as $G\,(\sigma(t, d), \gamma(t, d))$. We must also rename each variable (resp. event) of G by a unique variable (resp. event) of H[5]. This instantiation must resolve the specific problem $\mathcal{H}$ and we propose to instantiate a development path through the refinement of H.

4.4. Proof Obligation of an Instantiation

Now, proof obligations of any subsequent refinement assume that the instantiated development solves the specific problem $\mathcal{H}$. The next refinement step captures the semantics of how the specific problem $\mathcal{H}$ is solved in the same way as the problem $\mathcal{G}$, after a suitable instantiation. When the instantiation is proved to be correct, we freely obtain a complete instantiated development for the new problem $\mathcal{H}$.

[4]We have no formal metric for the complexity of a refinement path; however, intuitively a path is simpler if there are fewer proof obligations that require interactive proof as they cannot be discharged automatically.

[5]We can assume that H and G have no common parameters: x is different from y and events names are different.

An instantiation requires one only to prove that the properties of the system G are theorems with respect to the properties of H. We do this in two steps:

- (1) The properties of the system G, i.e. axioms defining the theory of G, are theorems in the new theory defined by the problem H:

$$Q(t, d) \;\Rightarrow\; P(\sigma(t, d), \gamma(t, d))$$

- (2) both models are solving the same problem and the event $spec\_event$ of H is refined by the instance of the event $event$ of G for the problem H:

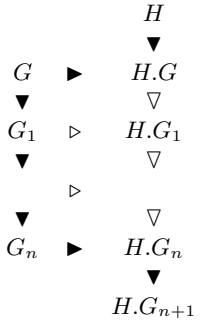
$$\begin{array}{l} Q(t, d) \wedge P(\sigma(t, d), \gamma(t, d)) \wedge I(y) \wedge [s, c := \\ \sigma(t, d), \gamma(t, d)]J(y1) \wedge y = y1 \wedge R(L)(y1, y1') \\ \Rightarrow \\ \exists y'.(I(y') \wedge R(K)(y, y')) \end{array}$$

Once we establish these two steps, we have a formal verification of the correctness of our concrete solution with respect to the already existing generic project:

Property 1 *When the given (previous) proof obligations are proved, the new problem $\mathcal{H}$ is solved by the development of the problem $\mathcal{G}$, up to renaming and instantiation.*

4.5. A formal design pattern

When the refinement is proved, the new problem $\mathcal{H}$ is solved by the development of the problem $\mathcal{G}$, upto renaming and instantiation. A new project $\mathbb{H}$ is created from the project $\mathbb{G}$ of the problem $\mathcal{G}$: events are renamed, variables are renamed, instantiations are done. Parameters are not necessarily completely instantiated or renamed: if a parameter is not instantiated then it keeps properties stated in the general model and no new proof obligation is generated. We illustrate this formal design pattern in the following diagram:



The diagram tells us where proof obligations must be proved: filled (triangular arrow) symbols show that new

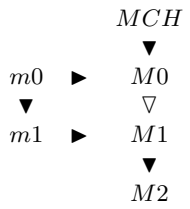
proof obligations are generated and require proving; non-filled symbols show that proof obligations have been generated but their proofs are inherited from the previous project ($\mathbb{G}$). Horizontal arrows represent instantiation. Vertical arrows represent refinement. The proof that model $H.G_{n+1}$ is a correct solution to the problem H is simplified by re-use of the refinement path in project $\mathbb{G}$.

5. The formal development in B

For conciseness, we choose to instantiate our formal design pattern in its simplest form, where the generic refinement path is a single refinement step.

5.1. Applying the design pattern

We have two generic models ($m1$ is a solution to problem $m0$) and a specific model (MCH) which is an instantiation from both generic models; then both specific model ($M0$ and $M1$) are obtained by this instantiation. Finally, we refine $M1$ to a final implementation model $M2$. This is illustrated in the diagram, below:



In the following, we do not have space to include complete code for all models and so we include snippets of B code which illustrate the main aspects of the application of the design pattern to the e-voting tamper-evident storage requirement. (Complete code for these projects is available from the authors on request.)

5.2. Project $\mathbb{G}$

5.2.1 The generic problem model $m0$

The most abstract generic model must capture the essence of the problem: a vote (v) in storage can be tampered with (be corrupted) but we can detect this as a bad vote. Storage that has not been tampered with contains only $good$ votes. A bad vote cannot be tampered with in order for it to be made $good$. This suggests modelling a single abstract $corrupt$ event.

corrupt $\widehat{=}$
any v **where**
$v \in good$
then
$good := good - \{v\}$
$bad := bad \cup \{v\}$
end

The invariant property specifies that $good$ and bad are subsets of an abstract $VOTE$ set and that they have no elements in common (so that a concrete vote cannot be both good and bad at the same time). We note that, in order to establish the invariant, $m0$ generated 5 simple proof obligations[6] that were automatically discharged.

At this point, one may ask what happens when a vote that is corrupt is corrupted again. The implicit $skip$ operation models the abstract event where the state of the model is not changed. This is precisely what we require: when we corrupt a vote that is bad we require that it stays bad. Thus, re-corruption may appear in more concrete refinements of $m0$ provided they refine skip of the abstract model $m0$. This is precisely what we intend to do in our next model $m1$.

5.2.2 The generic problem model $m1$

The refinement in model $m1$ introduces an abstract mechanism for encoding votes. Constants G_C (for GoodCode) and B_C (for BadCode) are two subsets of the set $CODE$. These sets are disjoint but don't neccessary cover the set $CODE$. The constant $code$ is a bijection between $VOTES$ and G_C.
The last constant chg is a relation between $CODE$s. The most important property of the relation chg is that a good or bad code can only be changed to a bad one. In B we specify these as $PROPERTIES$ of the model.

$G\_C \subseteq CODE$
$B\_C \subseteq CODE$
$G\_C \cap B\_C = \emptyset$
$code \in VOTE \rightarrowtail\!\!\!\!\!\!\rightarrow G\_C$
$chg \in CODE \leftrightarrow CODE$
$chg[G\_C \cup B\_C] \subseteq B\_C$

We refine the $corrupt$ event to ensure that any encoded vote that has been changed can be recognised as being bad. Furthermore, we refine the $skip$ event to say that if we now allow encoded votes to be changed then a bad vote is guaranteed to stay bad.

corrupt $\widehat{=}$
any v, c, b **where**
$v \in vt$
$c \in G\_C$
$b \in CODE$
$v \mapsto c \in Cv$
$c \mapsto b \in chg$
then
$Cv(v) := b$
end

corrupt_again $\widehat{=}$
any v, c, b **where**
$v \in vt$
$c \in B\_C$
$b \in CODE$
$v \mapsto c \in Cv$
$c \mapsto b \in chg$
then
$Cv(v) := b$
end

[6] An example proof obligation is $v \in good \wedge good \cap bad = \emptyset \implies (good - \{v\}) \cap (bad \cup \{v\}) = \emptyset$.

5.2.3 $m1$ refines $m0$

There is some additional work in proving that $m1$ refines $m0$ as we need to "glue together" the abstract and concrete models using a gluing invariant.

$$
\begin{array}{l}
Cv \in vt \rightarrow G\_C \cup B\_C \\
good = \mathsf{dom}(Cv \rhd G\_C) \\
bad = \mathsf{dom}(Cv \rhd B\_C)
\end{array}
$$

To glue together the actual set of votes (vt) with the $CODE$ we introduce Cv. Then, the abstract variables $good$ and bad can be defined in the concrete model using Cv.

We note that $m1$ generated 11 proof obligations and all but one were automatically discharged, with the single remaining obligation easily discharged through interaction with the theorem prover.

5.3. Project $\mathbb{H}$

In project $\mathbb{G}$ we have abstracted away from how a vote is represented. In project $\mathbb{H}$ we work with concrete representations of the votes (within the generic structure of project $\mathbb{G}$) by instantiating parameters of the models in $\mathbb{G}$.

5.3.1 The problem to be solved — MCH

In our pattern, the new problem to be solved is expressed by the model called MCH; which contains all specific constants, and with identical variables and identical event names as can be found in $m0$. $M0$ (resp. $M1$) is the model $m0$ (resp. $m1$) instantiated using the new constants of MCH and our new project is $\mathbb{MCH}$. MCH is the basis for our iterative refinement development process. The main point of interest is that the refinement between the model MCH and the model $M0$, an instance of $m0$, allows us to guarantee that our specific problem MCH is solved or refined by our intantiated model $M0$. For convenience (and space) we present all specific constants in two steps when presenting $M0$ and $M1$.

5.3.2 $M0$ refines MCH and instantiates $m0$

We start the development with an abstract model where a vote is represented by k bits. In fact, the abstract $VOTE$ set in model $m0$ is replaced (instantiated) by our more concrete $VOTES$:

$$
\begin{array}{l}
k \in \mathbb{N} \\
VOTES = 1..k \rightarrow 0..1
\end{array}
$$

We have nothing to prove to verify that this instantiation is correct (there are no additional properties on abstract set $VOTE$). The proof obligation that $M0$ refines MCH is obvious (with the same abstract and concrete events) and done automatically.

5.3.3 $M1$ — an intermediate design step

For this step we enrich the representation of a vote by doubling the number of bits. Each bit in the original vote representation is paired with its inverse value. For example, a vote that was represented as 10010 will now be represented by $(10010, 01101)$

$$
\begin{array}{l}
inv \in VOTES \rightarrow VOTES \\
\forall (v,i) \cdot \left(\begin{array}{l} v \in VOTES \ \ \wedge \ \ i \in 1..k \\ \implies \\ inv(v)(i) = 1 - v(i) \end{array} \right) \\
CODE = VOTES \times VOTES \\
code \in VOTES \rightarrow CODE \\
\forall v \cdot (v \in VOTES \implies code(v) = v \mapsto inv(v))
\end{array}
$$

For the new model, we instantiate the constants of the generic model $m0$: GC instantiates G_C and BC instantiates B_C and $chgv \cup chgi$ instantiates chg.

GC and BC are specified as follows:

$$
\begin{array}{l}
GC = \{v \mapsto w | v \mapsto w \in CODE \ \wedge \ w = inv(v)\} \\
BC = \{c | c \in CODE \ \wedge \ PC(c)\} - GC
\end{array}
$$

Note: in the definition of BC we use a predicate over codes, PC, that is defined to identify all "possible" codes. This predicate is true except in the case where a pair of bits in an encoded vote are both 1. This encoding is not possible because we allow only bits to change from 1 to 0 (and not from 0 to 1) and because all votes are initially coded as good codes (with pairs 01 or 10). PC is defined as follows:

$$
PC(c) = \exists (v,w) \cdot \left(\begin{array}{l} c = v \mapsto w \ \wedge \\ \forall i \cdot \left(\begin{array}{l} i \in 1..k \ \wedge \\ v(i) = 1 \\ \implies \\ w(i) = 0 \end{array} \right) \end{array} \right)
$$

Now we wish to specify that when a change is made to a single bit of a vote's representation (in either of the pair elements) then only a bit 1 can change to the bit 0. We do this by defining $chgv$ to specify how the vote part of the pair can change, and $chgi$ to specify how (symmetrically) the inverse part of the pair can change. The specification of $chgi$ is given below:

$$
\left(\begin{array}{l} (v1 \mapsto w1) \mapsto (v2 \mapsto w2) \in chgi \\ \Leftrightarrow \\ v1 = v2 \ \wedge \\ \exists i \cdot \left(\begin{array}{l} i \in 1..k \ \wedge \\ w1(i) = 1 \ \wedge \ w2(i) = 0 \ \wedge \\ \forall j \cdot \left(\begin{array}{l} j \in 1..k \ \wedge \ i \neq j \\ \implies \\ w1(j) = w2(j) \end{array} \right) \end{array} \right) \end{array} \right)
$$

The specification of $chgv$ is symmetrically defined.

5.4. $M1$ instantiates $m1$

We have instantiated[7] our previous generic model replacing: G_C with GC, B_C with BC and chg with $chgv \cup chgi$. Then we need to prove the instantiation to be correct by establishing the following proof obligation (from the invariant of model $m1$).

$$
\begin{array}{l}
GC \subseteq CODE \\
BC \subseteq CODE \\
GC \cap BC = \emptyset \\
code \in VOTE \rightarrowtail\!\!\!\!\!\!\rightarrow GC \\
chgv \cup chgi \in CODE \leftrightarrow CODE \\
(chgv \cup chgi)[GC \cup BC] \subseteq BC
\end{array}
$$

For convenience we structure the proof based on the two symmetric cases, depending on whether a change is made using $chgi$ or $chgv$. To do this, we split both events corrupt and corrupt_again into corruptv and corruptv_again, and corrupti and corrupti_again.

For brevity, we give the definition of only one half of the symmetric pair, the $chgv$ case:

corruptv $\widehat{=}$
any v, c, b **where**
$v \in vt$
$c \in G\_C$
$b \in CODE$
$v \mapsto c \in Cv$
$c \mapsto b \in chgv$
then
$Cv(v) := b$
end

corruptv_again $\widehat{=}$
any v, c, b **where**
$v \in vt$
$c \in B\_C$
$b \in CODE$
$v \mapsto c \in Cv$
$c \mapsto b \in chgv$
then
$Cv(v) := b$
end

We note that for this step we had 7 proof obligations (for the instantiation), 3 of which required interactive proofs.

5.5. A final implementation model — $M2$

A manchester code is a sequences of $2 \times k$ bits where the oddly ranked bits give the representation of the original k bits of an unencoded vote, and the even rank gives the bit-wise inverse. This is defined by $MNCH$, below:

$$
\begin{array}{l}
MNCH = 1..2 \times k \rightarrow 0..1 \\
mcode \in VOTES \rightarrow MNCH \\
\forall (v, i) \cdot \left(\begin{array}{l} v \in VOTES \ \land \ i \in 1..k \\ \Longrightarrow \\ mcode(v)(2 \times i - 1) = v(i) \ \land \\ mcode(v)(2 \times i) = 1 - v(i) \end{array} \right)
\end{array}
$$

We also define two constants, mv and mi, to extract a vote and its inverse from the manchester encoded $(2 \times k)$ bits:

[7] In our design pattern, $M1$ is shown as simply an instantiation of $m1$. In fact, in this paper, $M1$ is constructed as a refinement of the instantiation. For the sake of brevity, we have combined a horizontal instantiation with a vertical refinement in a single development step.

$$
\begin{array}{l}
mv \in MNCH \rightarrow VOTES \\
mi \in MNCH \rightarrow VOTES \\
\forall (c, i) \cdot \left(\begin{array}{l} c \in MNCH \ \land \ i \in 1..k \\ \Longrightarrow \\ mv(c)(i) = c(2 \times i) - 1 \ \land \\ mi(c)(i) = c(2 \times i)) \end{array} \right)
\end{array}
$$

5.5.1 $M2$ refines $M1$

It should be obvious that the Manchester code is a correct implementation of our requirements since it is clearly a correct implementation of $M1$. Intuitively, $M2$ refines $M1$ by changing the way in which the votes are encoded. In $M1$ they are encoded as a pair of bitsequences; in $M2$ they are single bit sequences where the original pair values have been interleaved. For example, the vote 101 is encoded as $(101, 010)$ in $M1$ but as 100110 in $M2$.

In order to formally proof this, we establish that the invariant in $M1$ is true in the refinement $M2$. In order to do this, we replace (instantiate) Cv with $Mchv$.

$$
\begin{array}{l}
Mchv \in vt \rightarrow MCH \\
\forall (v, m) \cdot \left(\begin{array}{l} v \in vt \ \land \\ m \in MNCH \ \land \\ v \mapsto m \in Mchv \\ \Longrightarrow \\ v \mapsto (mv(m) \mapsto mi(m) \in Cv \end{array} \right)
\end{array}
$$

Then we introduce two more constants:

corruptx $\widehat{=}$
any v, c, a **where**
$v \in vt$
$v \mapsto c \in Mchv$
GD
$a \in 1..2 \times k$
$x(a)$
$c(a) = 1$
then
$Mchv(v)(a) := 0$
end

corruptx_again $\widehat{=}$
any v, c, a **where**
$v \in vt$
$v \mapsto c \in Mchv$
$\neg GD$
$a \in 1..2 \times k$
$x(a)$
$c(a) = 1$
then
$Mchv(v)(a) := 0$
end

where:

$$GD = \forall i \cdot \left(\begin{array}{l} i \in 1..k \\ \Longrightarrow \\ c(2 \times i - 1) \neq c(2 \times i) \end{array} \right), \text{ and}$$

$v(a) = odd(a)$ and $i(a) = even(a)$.

Without going into details, we note that in this step there are 15 proof obligations, 5 of which required interactive proofs as they could not be discharged automatically by the tool.

6. Conclusions

We have argued that without the adoption and promotion of formal methods, as the foundations of software engineering, developing trustworthy e-voting systems will not necessarily guarantee that they will be trusted. We have demonstrated the application of the formal methods event-B for guaranteeing, through construction, the correctness of a vote store with respect to the requirement for *tamper-evident* storage. We illustrated the utility of our

refinement-based approach by verifying — through the application of a reusable formal design pattern — a store design that uses a specific PROM technology and applies a specific Manchester encoding mechanism. The formal design pattern is a reusable solution to a common design problem — of how genericity can help to structure the refinement proof process — that can be exploited by formal developers who are not necessarily expert.

Future work is mainly concerned with maintainability and extensibility, for example:

- **Strongly tamper-evident storage** — The design that we have presented in this paper guarantees that the store is weakly tamper evident. It is said to be weak because tampering can be detected once an election is complete. In fact, with minor modifications to the design we can meet the requirement for a store that is strongly tamper evident: so that tampering can be detected during the voting process.
- **Election closed bit** — There is a separate requirement that no more votes can be recorded once an election is closed. Clearly, the implementation of this requirement will involve some extension to the storage of votes so that the store is protected against any further addition of votes after the voting process is terminated (known as vote stuffing). A proposed design is to add an election closed bit to the store and to check that this is not set as a guard for the writing of a vote to the store. Of course, with our encoding mechanism we can detect when this bit has been tampered with. However, without formal modelling it is difficult to reason about the consequences of such a design with respect to a potential denial of service attack where the bit is set before the election has really terminated.
- **History independent storage** – The requirement that the physical order of the votes recorded in the store cannot be used to deduce any information about the vote of a particular voter.

We will analyse the different structuring mechanisms in event-B and the ways in which they can be used to extend our storage requirements.

References

[1] J.-R. Abrial. B$^{\#}$: Toward a synthesis between Z and B. In D. Bert and M. Walden, editors, *3nd International Conference of B and Z Users - ZB 2003, Turku, Finland*, Lecture Notes in Computer Science. Springer Verlag, June 2003.

[2] J.-R. Abrial and D. Cansell. Click'n'prove: Interactive proofs within set theory. In D. Basin and B. Wolff, editors, *16th Intl. Conf. Theorem Proving in Higher Order Logics (TPHOLs'2003)*, volume 2758 of *Lecture Notes in Computer Science*, pages 1–24. Springer Verlag, Sept. 2003.

[3] I. Brown. Who is enfranchised by remote voting? In *COMPSAC '05: Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC'05) Volume 1*, page 500, Washington, DC, USA, 2005. IEEE Computer Society.

[4] J. L. Caldwell. Formal methods technology-transfer: a view from NASA. In S. Gnesi and D. Latella, editors, *Proceedings of the ERCIM Workshop on Formal Methods for Industrial Critical Systems*, Oxford England, March 1996.

[5] D. Cansell, J. Paul Gibson, and D. Méry. Refinement: a constructive approach to formal software design for a secure e-voting interface. In *Proceedings, International Workshop on Formal Methods for Interactive Systems (FMIS), Macao SAR China*, 2006.

[6] ClearSy, Aix-en-Provence (France). *B4FREE*, 2004. http://www.b4free.com.

[7] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of software engineering*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[8] D. Gritzalis, editor. *Secure Electronic Voting*, volume 7 of *Advances in Information Security*. Springer, 2003.

[9] P. S. Herrnson, B. B. Bederson, B. Lee, P. L. Francia, R. M. Sherman, F. G. Conrad, M. Traugott, and R. G. Niemi. Early appraisals of electronic voting. *Soc. Sci. Comput. Rev.*, 23(3):274–292, 2005.

[10] M. Horst, M. Kuttschreuter, and J. M. Gutteling. Perceived usefulness, personal experiences, risk perception and trust as determinants of adoption of e-government services in The Netherlands. *Comput. Hum. Behav.*, 23(4):1838–1852, 2007.

[11] J. Paul Gibson. Formal requirements engineering: Learning from the students. In *Australian Software Engineering Conference*, pages 171–180. IEEE Computer Society, 2000.

[12] P. Kocher and B. Schneier. Insider risks in elections. *Commun. ACM*, 47(7):104, 2004.

[13] T. Kohno, A. Stubblefield, A. D. Rubin, and D. S. Wallach. Analysis of an electronic voting system. In *IEEE Symposium on Security and Privacy*, pages 27–40. IEEE, 2004.

[14] M. McGaley and J. Paul Gibson. E-Voting: A Safety Critical System. Technical Report NUIM-CS-TR-2003-02, NUI Maynooth, Comp. Sci. Dept., 2003.

[15] M. McGaley and J. McCarthy. Transparency and e-Voting: Democratic vs. Commercial Interests. In *Electronic Voting in Europe - Technology, Law, Politics and Society*, pages 153 – 163. European Science Foundation, July 2004.

[16] P. Mehlitz and J. Penix. Design for verification - using design patterns to build reliable systems. In *Proceedings of 6th ICSE workshop on component based software engineering*, Portland, Oregan, May 2003.

[17] R. T. Mercuri. Computer security: quality rather than quantity. *Commun. ACM*, 45(10):11–14, 2002.

[18] D. Molnar, T. Kohno, N. Sastry, and D. Wagner. Tamper-evident, history-independent, subliminal-free data structures on PROM storage —or— how to store ballots on a voting machine. In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 365–370, Washington, DC, USA, 2006. IEEE Computer Society.

[19] P. G. Neumann. Inside risks: risks in computerized elections. *Commun. ACM*, 33(11):170, 1990.

[20] W. L. Scherlis and J. Eisenberg. IT research, innovation, and e-government. *Commun. ACM*, 46(1):67–68, 2003.

[21] W. Stallings. *Data and computer communications*. Macmillan Publishing Co., Inc., Indianapolis, IN, USA, 1985.

[22] J. Svensson and R. Leenes. E-voting in europe: Divergent democratic practice. *Information Polity*, 8(1):3–15, 2003.

[23] N. Tideman and D. Richardson. Better voting methods through technology: The refinement-manageability trade-off in the single transferable vote. *Public Choice*, 103(1):13–34, April 2000.