Refinement: A constructive approach to formal software design for secure e-voting

Dominique Cansell[1], J. Paul Gibson[2], and Dominique Méry[3]

[1] Université de Metz, France (LORIA: MOSEL research group).
[2] Department of Computer Science, NUI Maynooth, Ireland (TASS research group).
Corresponding author: `pgibson@cs.nuim.ie`
[3] Université Henri Poincaré, Nancy, France (LORIA: MOSEL research group).

Abstract. Electronic voting machines have complex requirements. These machines should be developed following best practice with regards to the engineering of critical systems. The correctness and security of these systems is critical because an insecure system could be open to attack, potentially leading to an election returning an incorrect result or an election not being able to return any result. In the worst case scenario an incorrect result is returned — perhaps due to malicious intent — and this is not detected. We propose the use of formal methods as a means of ensuring that a machine counts correctly, and we demonstrate the use of the formal method B in guaranteeing simple safety properties of a voting machine implementing a common variation of the single transferable vote (STV) election process. The properties we examine are concerned with the collection and storage of only *valid* votes. We demonstrate that guaranteeing *validity* not only helps in the formal verification of the counting process, but also has an important role to play in making the machine more secure. Using the B-method, we apply an incremental refinement approach to verifying a sequence of designs for the collection and storage of votes, which we prove to be correct with respect to the simple requirement.

1 Introduction

Industrial-strength case studies help to validate ideas on realistic problems and to apply more theoretical advances in challenging environments. The problem of electronic vote counting (or tabulation) involves a wide range of issues, requiring expertise in science, engineering and technology. As such, it provides a good test bed for the application of formal methods.

1.1 E-voting: background and motivation

Applying state-of-the-art computer and information technology to "modernise" the voting process has the potential to make improvements over the existing paper (or mechanical) systems; but it also introduces new concerns with respect to secrecy, accuracy and security[16]. The debate over the advantages and disadvantages of e-voting is not a new one; and recent use of such systems in actual elections has led to their analysis from a number of viewpoints: usability[17], trustworthiness and safety criticality[21], transparency and openness[22, 25], and risks and threats[27].

The advantages are generally accepted: faster result tabulation, elimination of human error which occurs in manual vote tabulation, expansion of the franchise to those entitled to vote but who currently are unable to vote because of "special" needs, and improving the fairness of count systems that incorporate ("unfair") nondeterministic procedures.

Despite ungoing uncertainty over the trustworthiness of these systems — which is the major disadvantage associated with them — many countries have chosen to adopt e-voting. The main risks that have been clearly identified seem not to concern the politicians in their head-long rush to adopt new e-voting technologies.

In a recent paper, Kocher and Schneier[19] conclude by stating: "The threats are real, making openness and verifiability critical to election security." The formal methods community are experienced in chasing technological change in software engineering: and 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 chose to use the B method.

1.2 The B Method

B is a method[1] for specifying, designing and coding software systems. It is based on Zermelo-Fraenkel set theory with the axiom of choice, the concept of generalized substitution and structuring mechanisms. The concept of refinement[6, 5] is the key notion for developing B models of (software) systems in an incremental way. B models are accompanied by mathematical proofs that justify them. We start from an abstract model and each subsequent model is a refinement of the previous one. Proofs of B models help to convince the user (designer or specifier) that the (software) system is correct, since they demonstrate that the behavior of the last, and most concrete, system (software) respects the behaviour of the first, most abstract model (which we assume has already been validated).

The goal of a B development is to obtain a *proved model*. Since the development process leads to a large number of proof obligations, the mastering of proof complexity is a crucial issue. Even if a proof tool is available, its effective power is limited by classical results over logical theories and we must manage the complexity of proofs through the structure of the current development process, e.g. by refinement. Refinement has the potential to decrease the complexity of the proof process whilst allowing for tracability of requirements.

B models rarely need to make assumptions about the *size* of a system being modelled, e.g. the number of candidates in an election. This is in contrast to model checking approaches [11]. The price to pay is to face possibly complex mathematical theories and difficult proofs. The re-use of developed models and the structuring mechanisms available in B help in managing the complexity. Where B has been exercised on known difficult problems, the result has often been a simpler proof development than has been achieved by users of other more monolithic techniques.

Among case studies developed in B, we can mention the METEOR project [8] for controlling train traffic, the PCI protocol [9], the IEEE 1394 Tree Identify Protocol [4].

2 E-voting: formal methods, correctness and security

We propose to construct a formal, mathematical model of the e-voting problem in B. We argue that no e-voting system can be considered "safe" until the requirements of these systems are better defined; and introduce refinement as a method for supporting the correct design and implementation of "safe e-voting systems".

The importance of the procedural aspect of security in e-voting systems has already been identified[28], where analysis of exisiting systems identifies "several procedural security gaps". Clearly, some of these gaps cannot be easily addressed through the use of formal methods (as the methods are inappropriate for the modelling of the properties required); whilst other security gaps would never arise if formal methods were used.

Mercuri, in her thesis[23], identifies fundamental questions that need to be asked of any secure system, and of any electronic voting system. The two questions that are of most interest to us are: "How is vote tabulation correctness assured?", and "What are the data requirements, and how are these implemented and enforced?"

One could argue that the small errors that are most likely to occur in the tabulation process are very unlikely to have an impact on the correct candidates being elected. We disagree. Our view is that the tabulation must be developed formally: human error in the development of the tabulation software might have considerably more serious consequences than such errors in the manual system. Furthermore, we note from [14] that: "E-voting machines potentially make electoral fraud unprecedently simple. An election saboteur need only introduce a small change in the master copy of the voting software to be effective". We argue that even a minor design flaw in the way in which votes are presented for tabulation can lead to security weaknesses that could be exploited by an election saboteur.

3 Critical system development: formality and security

Critical Systems are ones where failure may lead to dire consequences. Requiring or expecting users to interact with such systems is conditional on them trusting the systems not to fail. The *cost of failure* is often used to further categorise these systems: e.g. *Safety Critical Systems* where failure may lead to loss of life, serious injury or extensive environmental damage; and *Mission Critical Systems* where there is no direct threat to life or limb, but where the cost of failure is not purely financial as it is likely to directly threaten the well-being of individuals or groups who rely on the system.

To motivate the use of formal methods, we argue that e-voting is at least *mission critical* and may, in some circumstances, be considered as *safety critical*[21].

For our argument we consider a “worst case scenario”, where the system fails to elect the correct candidates without the failure being identified. There is no meaningful way of equating this with a financial cost and its potential negative impact on the well-being of individuals and society is great. Thus, we must consider e-voting systems to be (at least) *mission critical* and we advocate the use of formal methods in their development as their application should ensure that the liklihood of failures due to modelling errors during design is reduced.

From a technological viewpoint we know that system design has an important role in security assurance. Mercuri[24] 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. One way of starting this process is by defining computer security with respect to need.” This supports our view that one must start with a simple model of the security needs 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 system* — and start our formal development from there.

We support an approach built on the concept of refinement that aids in the assurance of maintaining correctness properties throughout the whole development cycle. Our goal is to demonstrate the advantages of using a formal method in the verification of design decisions. Our goal is not to argue that a formal approach is always better; our goal is to show that a formal approach is feasible and could be usefully applied to parts of the design which are considered to be “critical” or vulnerable to attack.

4 Valid Votes: a STV case study

The Single Transferable Vote (STV) model on which we build this case study is regarded as a good democratic election process. However, it incorporates a complex, not necessarily determinisitic, tabulation (counting) procedure. In 2003, Farrell and McAllister[13] reported on how a subtle change in the implementation of the STV rules can lead to major changes in the results returned. In the article by Hill, Wichmann and Woodall[18], we see the modelling of an even more complex version of the STV following Gregory rules and using Meek’s method.

In this paper, the counting algorithm is not developed formally. However, a brief overview of the tabulation process will help us to develop our claim that it is critical that there is a formal verification of the property that only valid votes are counted.

4.1 Overview of the typical counting algorithm

In the overview that follows, our style of mixing natural language description with more formal notation is one that is common to official documentation of voting rules and procedures.

During an election, a candidate is elected if the number of votes they have is greater or equal to the *quota*. This quota, Q say, is a function of: the number of valid votes, V say, and the number of seats available for election, S say. It is usually defined by the equation: $Q = 1 + \frac{V}{S+1}$.

We note that the quota cannot be calculated without knowing the number of *valid* votes, and so its correct calculation is dependent on the notion of *validity* being correctly implemented. This notion is non-trivial as the STV election process allows voters to register support for more than one single candidate, by placing candidates in a preferred order. Thus on each vote, a candidate may or may not have an associated preference.

Informally, a vote is considered valid if and only if it shows a unique first preference. The means of specifying this property depends, of course, on a notation for representing a vote. Consider a constituency where there are three candidates: `A`, `B` and `C`, say. We could choose to represent a vote as a string of characters taken from the alphabet $\{A, B, C\}$. In such a string, we naturally interpret a character *ch* at index *i* in the vote string as stating that the *i*th preference of that particular vote is the candidate *ch*. Now, we can define a valid vote in this constituency by explicitly identifying the set of valid vote strings, for example:

$$\{A, B, C, AB, AC, BA, BC, CA, CB, ABC, ACB, BAC, BCA, CAB, CBA\}.$$

In such a definition we preclude, for example, the following strings from being considered as representations of valid votes:

1. The empty string — correctly excluded, we would argue, as there is no first preference.
2. The string AA — excluded, perhaps, because the candidate with the first preference has another associated preference. (We note that this was not explicit in the original informal definition, and it would be normal for this interpretation to be validated through additional discussion of this requirement with the customer.)
3. The string $ABBC$ — excluded, perhaps, because the number of preferences cannot be more than the number of candidates. (We note, again, that this was not explicit in the original informal definition.)

Of course, it is better[1] to define the notion of validity as a generic boolean function that takes any string of characters and decides on its validity, without having to explicitly construct all the members of the valid vote set. An even better approach, as taken with our abstract B model is to specify the set of all valid votes and to use refinement to be sure that a vote belongs to this set without having to actually construct it (as it exists in the abstraction).

4.2 Requirements for valid votes

Within any voting system, changing the definition of a valid vote can have major consequences for the election process and the results returned. In any STV

[1] Simple, naive, construction (listing) of the complete valid vote set is infeasible for elections involving even a moderate number of candidates.

system, for example, there is a major difference between *allowing* and *requiring* a voter to place all candidates in a preference order.

In the system that we chose to model, which is the most common in practice, a voter is facilitated in their wish to express preference for a few candidates by not forcing them to place all candidates in order. Consequently, in our model, it is required that a vote be considered valid if and only if it shows a unique first preference. This is a simple requirement, but one which can cause problems if it is not treated formally.

Clearly, if invalid votes manage to get passed to the tabulation process (to be counted) then there is a risk that this could break the counting process. For example, it would not be unreasonable to suggest that some of the tabulation methods make the assumption that the votes being counted are valid. However, without some degree of formal verification it is also likely that an invalid vote could — by accident — be counted and that this could lead to an incorrect result, a run-time error, or non-termination. Consequently, this weakness could also be exploited by an attacker to deliberately manipulate the election process.

Such a potential attack is similar to those mentioned in [26] where the security of votes stored in memory is addressed. In particular, the use of Trojan code to exploit vote data that has been tampered with is shown to be a real threat that requires elaborate schemes for the secure storage of votes. In most e-voting systems, there is a clear interface between the the storage of votes and the input of votes. We argue that the same degree of care must be taken in designing the vote interface to ensure that Trojan code cannot be used to exploit the input of invalid votes and their subsequent transfer to long term storage.

4.3 Validating votes in a typical implementation architecture

Typically, a voting machine has a simple, classic, layered architecture. We propose a generic, abstract model[2] in order to illustrate the need for formality in the processing of votes electronically:

- An *interface* facilitates the voter to input their preferences.
- A *store* records the preferences of all votes that have been input.
- A *tabulator* takes all the votes from the *store* and calculates the result.

We argue that an invalid vote in the tabulation process can compromise the security of the whole election. Thus, for an e-voting system to be considered secure, it is a requirement that the tabulator will not have to process invalid votes; and so it is a requirement that the store cannot transfer invalid votes into the tabulator. Consequently, it may be necessary to show that the store cannot receive invalid votes from the interface. There are clearly a number of design decisions that need to be taken with respect to the implementation of this simple architecture; and it is the responsibility of the designers to verify that their designs are correct with respect to the validity requirement.

[2] We deliberately choose not to model a specific machine.

5 Informal software design for vote validity

In this section we comment on typical design decisions that arise during implementation of our generic architecture. We do not claim to have seen such designs in real voting systems, but argue that such designs (and their flaws) are likely to arise in practice. We focus, for simplicity, on the first layer of our architecture: the *interface*. Simple analysis of the requirement for *only valid votes in the tabulation process* could lead one to designing a machine where the interface layer takes responsibility for guaranteeing that voters record only valid votes and rely on a secure store (like that proposed in [26]) to ensure that the vote record is tamper proof. In all the following designs (of the interface layer) we refer to buttons that the user can press and the information that is displayed to the voter. We abstract away from details of how these buttons and displays are implemented.

5.1 The rapid prototyping approach

We wish to verify that a given interface design implements the requirement that no invalid vote can be sent to the store. A standard technique for carrying out such a verification is to incrementally add rigour to the process, structuring the verification process in a number of layers. A typical 3-layer approach, which we used for the purpose of our study, is:

1. Run some voting scenarios through the natural language description and check, by hand, that there are no obvious examples of a scenario that leads to the requirement not being met.
2. Prototype an executable model of the design and test this executable model as thoroughly as possible.
3. Formalise the prototype model so that more formal model-checking or theorem-proving can be used to show that the design is *correct*.

In practice, with "simple" designs such as a simple vote interface, developers often see no need to progress to step (3). In the designs that follow in the remainder of this section, we follow a rapid-prototyping approach that never progresses to the 3rd layer of rigour. In section 6 we show how the B method and tools can be used to fully support layer 3 in this verification process.

As with all designs, it is necessary to abstract away from some details in order to focus on others. In this section we concentrate on the validity of a single vote. We note that the validity of a complete set of votes will require that each individual vote is valid. The validity of individual votes will be a necessary, but perhaps not sufficient condition for the validity of all votes. In this section we choose to work at the level of abstraction of a single vote — we leave it as an exercise for the readers to formulate the validity of all votes in the design language of their choice. (We have formulated this property, using B, in the next section.)

Before we consider verifying the property that the system contains only valid votes, it is necessary to formulate what we mean by a valid vote in a way that the design verification process can be automated. This means that, in our chosen modelling language, we have to choose a means of specifying the property that needs to be checked.

Consider the Java code which models the `Vote` (of a single voter) as an array of “Candidates” (identified by the array index value) associated to “Preferences” (identified by the value at the array index).

```
public class Vote{
int numberofCandidates;
int preferences [];
// Vote - construct empty Vote with no preferences
public Vote(int numCs){
    numberofCandidates = numCs;
    preferences = new int [numberofCandidates];
    for (int i = 0; i<numberofCandidates; i++) preferences[i] = 0;
}//END Vote Constructor
// ...
}// END CLASS Vote
```

We now formally specify a safety property that defines a valid vote as a `boolean` method of the `Vote` class.

```
// isValid *****************************************************
// The safety property to ensure the validity of a Vote
// A vote is valid iff there is a unique 1st preference recorded
public boolean isValid(){
    int numberof1s =0;
    for (int i = 0; i<numberofCandidates; i++)
      if (preferences[i]==1) numberof1s++;
    return (numberof1s==1);
}// END Vote.isValid
```

Of course, this invariant property could be modelled using *design-by-contract*[7] language and tool support. However, in this initial case study we choose to use only fundamental concepts that are part of the core programming language: we believe that little, if any, current voting software incorporates more rigorous design by contract methods.

Given the formal (constructive) specification of a valid vote (in Java) we can now proceed to the design of the first component of our generic architecture: the *interface*. The goal is to offer alternative interface designs and to verify the correctness, or otherwise, of each.

5.2 Design1: the simplest interface

We propose the following design for verification:

> "Every candidate is associated with a *candidate button*. Beside each candidate button, information is displayed to show the preference that is associated with that candidate. Initially, at the beginning of each vote, no preferences are displayed. When a voter presses a candidate button (for the 1st time) then it is assumed that the voter intended that candidate to be their first preference, and the number 1 is displayed beside that candidate's button. When a voter presses (for the second time) another candidate button then this is taken to represent their second preference, etc.... If they press a candidate's button that already has a preference associated with it then that press is ignored. When a voter is happy that they have recorded all their preferences, they press a *validate vote* button and the vote is sent to be stored. The *validate vote* button will only send the vote if at least one candidate button has been pressed."

We now model the design by a Java method (`pressCandButton`) which implements the behaviour of the system in response to the pressing of a candidate's button. We note that the choice of underlying data representation, with a vote as an array of Candidate's preferences, maps closely to the structure of the voting data as presented to the voter.

```
// Design1
// pressCandButton ******************************************
// Record next press as a preference, provided button is enabled
// Calls lastPref() to correctly assign the next preference value
// Calls buttonEnabled() to check if preference already allocated
public void pressCandButton(int button){
    if (!buttonEnabled(button-1)) return;
      else preferences[button-1] =lastPref()+1;
} // END Vote.pressCandButton
```

This `pressCandButton` method is dependent on two other methods: `buttonEnabled` for checking if the button being pressed is actually enabled (has not been pressed before), and `lastPref` for finding out the value of the last preference selected. The `buttonEnabled` behaviour is straightforward to implement, and left as an exercise for the reader. The `lastPref` method, as implemented below, examines all the preferences and deduces that the largest current preference value must have been the last preference made. We note, for future reference, that this is a correct (but inefficient) implementation of the design.

We argue that it is difficult to ensure the correctness of even this simple interface design without explicitly specifying and correctly implementing the `isValid()` method, as we have done. Working with reasonably experienced Java

```
// lastPref ****************************************************
// Called by pressCandButton()
protected int lastPref(){
    int largest =0;
    for (int i = 0; i<numberofCandidates; i++)
      if (preferences[i]> largest) largest = preferences[i];
    return largest;
} //END Vote.lastPref
```

programmers (students with at least 3 years of experience with Java) on this problem led us to observe a “poor design” that could compromise security.

5.3 Poor design may lead to security risks

A common approach to rapidly prototyping this “simple” first design is to realise that a vote is valid as soon as one of the candidate buttons is pressed. This was hinted at in the initial statement of the design:

> “The *validate vote* button will only send the vote if at least one candidate button has been pressed.”

It is a much quicker and simpler solution to hardcode this as an `enabled` boolean variable. In fact, this “solution” is correct but it is a poor design because it is not robust to changes in requirements. Consider a scenario where the interface requirements are extended to allow voters to reset their vote (in order, perhaps, to facilitate them in correcting an input error, or in changing their minds). The design of using a boolean `enabled` resulted in a significant number of students forgetting (during the coding of the new design feature) that they need to set this value to `false` when a vote is reset, and consequently led to their designs allowing an *empty vote* — with no preferences recorded — to be sent to the store. Furthermore, as this mistake was made, in general, by the weaker students a majority of them did not catch the error through their testing of the Java code. Could this sort of thing happen in a real e-voting system? Judging by analysis regarding the quality of the code (and development methods) used in systems that have been examined in detail[20], one could not be sure that “professional developers” would not make the same “simple” mistake.

It is difficult to make a judgement as to the severity of such a design fault. Clearly, it has the potential for voters to have their voting intentions incorrectly recorded. This could lead to an election returning the “wrong result”. A second risk is that invalid empty votes in the store may break the tabulation process: if tabulation methods (functions) work on the assumption that all votes have at least one unique first preference (and may have been tested under that assumption) then it is possible that tabulation may not even terminate if this assumption is broken. A third risk is that some attacker has managed to introduce code into the system that can manipulate the tabulation process but will only be called when an invalid vote (or password-like sequence of invalid votes) is passed to the store. Of course, rigorous design procedures should find these types of design flaws without the need to resort to formal methods. However, in *critical system* design "should" is not good enough.

5.4 Design2: a more sophisticated, *incorrect*, interface design

In the second design, we analyse how an extension to the requirements, in order to provide a more sophisticated interface, can pose specific problems. Imagine that we wish to provide a means of a voter changing their vote, without them having to reset all their preferences. In particular, we wish them to be able to cancel the last preference chosen (in the case that they accidentally pressed the wrong button). We propose the following design for verficiation:

> "if the voter presses a candidate button again then that preference is erased".

This is a faulty design, but without the use of formal methods are we sure to identify the fault before more costly implementation takes place?

The `VoteExt1` class uses the inheritance mechanism in Java to add this extra interface feature to the already existing `Vote` class.

```
// Design2
public class VoteExt1 extends Vote {
VoteExt1 (int numCs){ super(numCs);}
// pressCandButton ***********************************************
// The new feature requires over-riding of pressCandButton().
// Here is how it SHOULD NOT be done
public void pressCandButton(int button){
    if (!buttonEnabled(button-1)) preferences[button-1] = 0;
    else preferences[button-1] =lastPref()+1;
} // END VoteExt1.pressCandButton
} // END VoteExt1
```

In an ideal world, our development tool(s) should be able to automatically tell us that, either: the new functionality respects the safety property of the exisiting `Vote` class (that we have already formally verified) and provide the proof, or identify at least 1 scenario in which the new functionality breaks the safety property.

This code "nearly works": it implements the requirements correctly provided one can make the assumption that a voter will only press a button a second time (to cancel a preference) *immediately* after pressing that button for a first time, with no other preferences being recorded in the mean time. We model this using B in section 6; and demonstrate how the B tools automatically prove that the system is "broken" if this assumption about the voter's behaviour is invalid.

5.5 The risk of feature interactions in design

We note, again from observing our students, that making parallel changes to requirements and design models can lead to feature interactions similar to those well documented for telephone services[15]. Consider the interaction that arose between two features that by themselves did not break the safety property of the system but when combined (in an albeit informal manner) led to invalid votes being recorded:

1. The `lastPrefs` method was optimized by adding a counter value so that an iteration through the candidate list was not required each time a new preference was input.
2. A `reset` button was added to allow all preferences to be deleted.

Individually, each of these refinements does not compromise the system by allowing the introduction of previously invalid votes. However, together they can result in an invalid vote with no first preference being recorded even though some preferences have been selected. (With candidates `A`, `B` and `C`, for example, the sequence of button presses `A-B-reset-C` leads, under the new combined feature functionality, to an invalid vote where only the 3rd preference for `C` is recorded.)

5.6 Design3: a more sophisticated, *correct* interface design

Consider the interface design whereby a voter can press a candidate button a second time and this will delete that candidate's preference value, as in design2, above. However, it will also delete all the preference values lower than the preference just deleted.

```
// Design3
public class VoteExt2 extends Vote {
VoteExt2 (int numCs){ super(numCs);}
// The extension over-rides the pressCandButton method.
// pressCandButton ********************************************
public void pressCandButton(int button){
if (!buttonEnabled(button-1)) {
    int deletefrom = preferences[button-1];
    for (int i = 0; i<numberofCandidates; i++){
      if (preferences[i] >= deletefrom) preferences[i] = 0;
      else preferences[button-1] =lastPref()+1;}
} // END VoteExt2.pressCandButton
} // END VoteExt2
```

Informally, one can "verify its correctness" with the following reasoning: "This will guarantee that if the 1st preference is deleted then all the preferences are deleted and that the empty vote is the only invalid vote that can be found in the interface." We see, in the next section, how we prove the correctness of the design in a more formal manner. It is left as an exercise (and challenge) to the reader to verify the model as coded in Java.

6 Formal software design for vote validity

6.1 Incremental development and refinement

Proof-based development methods integrate formal proof techniques in the development of software systems. The main idea in our refinement based-approach is to start with a very abstract model of the system under development. We then gradually add details to this first model by building a sequence of more concrete ones. The relationship between two successive models in this sequence is that of *refinement* [6, 1, 10]. It is controlled by means of a number of, so-called, *proof obligations*, which guarantee the correctness of the development. Such proof obligations are proved by automatic (and interactive) proof procedures supported by a proof engine [12, 3]. The essence of the refinement relationship is that it preserves already proved *system properties* including safety properties and termination properties. The invariant of an abstract model plays a central rôle for deriving safety properties and the methodology focuses on the incremental discovery of the invariant; the goal is to obtain a formal statement of properties through the final invariant of the last refined abstract model.

Figure 1 gives set-theoretical notations of the B data modelling language and it borrows notations and concepts from Bourbaki's group. A complete introduction to B can be found in [1].

Name	Syntax	Definition
Binary Relation	$s \leftrightarrow t$	$\mathcal{P}(s \times t)$
Domain	$\mathsf{dom}(r)$	$\{a \mid a \in s \wedge \exists b.(b \in t \wedge a \mapsto b \in r)\}$
Codomain	$ran(r)$	$\mathsf{dom}(r^{-1})$
Restriction	$s \lhd r$	$\mathbf{id}(s); r$
Co-restriction	$r \rhd t$	$r;\ \mathbf{id}(s)$
Anti-co-restriction	$r \ \rhd\!\!- t$	$r \rhd (ran(r) - t)$
Image	$r[w]$	$ran(w \lhd r)$
$\cup r$		
Partial Function	$s \nrightarrow t$	$\{r \mid r \in s \leftrightarrow t \ \wedge\ (r^{-1}; r) \subseteq \mathbf{id}(t)\}$
Total Function	$s \rightarrow t$	$\{f \mid f \in s \nrightarrow t \ \wedge\ \mathsf{dom}(f) = s\}$
$\mathrm{f}^{-1} \in t \nrightarrow s\}$		
Total injection	$s \rightarrowtail t$	$\{f \mid f \in s \rightarrow t \ \wedge\ f^{-1} \in t \nrightarrow s\}$ height

Fig. 1. B set notations

The refinement of an event B model [2, 5] allows one to enrich the model in a *step-by-step* manner. Refinement provides a way to construct stronger invariants and also 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 essentially done by extending the list of state variables (possibly suppressing some of them), 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.

MODEL
$\quad AllVotesAtOnce$
SETS
$\quad ELECTOR; CAND$
CONSTANTS
$\quad nbc$
PROPERTIES
$\quad nbc = card(CAND)$
DEFINITIONS
$\quad valid(v) \ \widehat{=}\ \exists n \cdot \ (n \in 1..nbc \ \wedge \ v^{-1} \in 1..n \rightarrowtail CAND))$
VARIABLES
$\quad vote, nbv$
INVARIANT
$\quad vote \in ELECTOR \leftrightarrow (CAND \times (1..nbc)) \ \wedge$
$\quad \forall e \cdot \ (e \in \mathsf{dom}(vote) \Rightarrow valid(vote[\{e\}]) \ \wedge$
$\quad nbv \in \mathbb{N} \ \wedge$
$\quad nbv = card(\mathsf{dom}(vote))$
INITIALISATION
$\quad vote := \,||nbv := 0$
EVENTS
$\mathsf{Voting} \ \widehat{=}$
$\quad$ **begin**

$$vote, nbv : | \begin{pmatrix} vote \in ELECTOR \leftrightarrow (CAND \times (1..nbc)) \ \wedge \\ \forall e \cdot \ (e \in \mathsf{dom}(vote) \Rightarrow valid(vote[\{e\}])) \ \wedge \\ nbv \in \mathbb{N} \ \wedge \\ nbv = card(\mathsf{dom}(vote)) \end{pmatrix}$$

$\quad$ **end**

Following the traditional refinement process, the first model (`AllVotesAtOnce`) we propose is very high level: it abstracts away from individual votes and button presses. There is only one event — Voting — which models the votes of all electors who came to vote in *one shot*. A valid complete vote will be refined, in the next model, into the property that every individual vote is valid. For readers unfamiliar with the B modelling language, we note that:

– $ELECTOR$ is the set of all electors and $CAND$ is the set of all candidates,
– *vote* is the variable which contains votes of all the electors, and
– *nbv* is the cardinal of the domain of *vote* representing the total number of votes.

Instead of modelling all the votes in a single *one shot*, as above, in our next step we choose to refine the most abstract specification so that each individual vote is modelled, using the event `One_Vote` in the model `EachVoteAtOnce`. Without such a refinement a realisitic implementation cannot be constructed.

```
MODEL
   $EachVoteAtOnce$
REFINES
   $AllVotesAtOnce$
SET

   $STATE = \{voting, finish\}$

VARIABLES

   $vote, nbv,$
   $st, elector, vt, cvt$
INVARIANT

   $elector \subseteq ELECTOR \ \wedge$
   $vt \in elector \leftrightarrow (CAND \times (1..nbc)) \ \wedge$
   $\mathsf{dom}(vt) = elector \ \wedge$
   $\forall e \cdot \ (e \in \mathsf{dom}(vt) \Rightarrow valid(vt[\{e\}]) \ \wedge$
   $cvt \in \mathbb{N} \ \wedge$
   $cvt = card(elector) \ \wedge$
   $st \in STATE \ \wedge$
   $(st = finish \Rightarrow \mathsf{dom}(vote) = elector)$
```

```
INITIALISATION

   $vote := \emptyset \ || \ nbv := 0 \ ||$
   $st := voting \ || \ elector := \emptyset \ ||$
   $cvt := 0 \ || \ vt := \emptyset$
EVENTS

One_Vote  $\widehat{=}$
   any  $e, v, n$  where
      $st = voting$
      $e \in ELECTOR - elector$
      $n \in 1..nbc$
      $v \in 1..n \rightarrowtail CAND$
   then
      $vt := vt \ \cup \ (\{e\} \times v^{-1}) \quad ||$
      $elector := elector \ \cup \ \{e\} \ ||$
      $cvt := cvt + 1$
   end ;
Voting  $\widehat{=}$
   when
      $st = voting$
   then
      $vote, nbv := vt, cvt \quad ||$
      $st := finish$
   end
END
```

In this more concrete `EachVoteAtOnce` model, $STATE$ is an enumeration set which contains two values: *voting* to represent a voting system that is *open*, and $finish$ to represent when the voting is *closed*. The variable *st* contains one of these values, and *elector* is the subset of $ELECTOR$ recording the electors who have already voted, and *vt* is the variable which contains these votes, and *cvt* is its cardinal. All votes in *vt* are valid. The event `One_Vote` models the vote of a single elector in *one shot* (as a single event).

The B models that follow correspond to the three Java designs that we saw in the section 5. We show how the designs can be formally verified when modelled in B.

6.2 Design1: the simplest interface

In this `Vote` model — which corresponds to the Java `Vote` class — an elector e (who has not already voted) votes for candidates by pressing on the corresponding buttons.

```
MODEL
  $Vote$
REFINES
  $EachVoteAtOnce$
SET
  $One\_Voting = \{start, valid, no\_elec\}$
VARIABLES
  $vote, nbv,$
  $st, elector, vt, cvt$
  $e, n, v, sto$

INVARIANT

  $e \in ELECTOR \ \wedge$
  $v \in \mathbb{N} \leftrightarrow CAND \ \wedge$
  $n \in 0..nbc \ \wedge$
  $n = card(\mathsf{ran}(v)) \ \wedge$
  $sto \in One\_Voting \ \wedge$
  $(sto \neq no\_elec \Rightarrow$
    $v \in 1..n \rightarrowtail CAND) \ \wedge$
  $(sto \neq no\_elec \Rightarrow$
    $e \in ELECTOR - elector) \ \wedge$
  $(st = voting \ \wedge \ sto = valid \Rightarrow n \neq 0)$

INITIALISATION

  $vote := \emptyset \ || \ nbv := 0 \ ||$
  $st := voting \ || \ elector := \emptyset \ ||$
  $cvt := 0 \ || \ vt := \emptyset \ ||$
  $sto := no\_elec \ || \ e :\in ELECTOR \ ||$
  $v := \emptyset \ || \ n := 0$
```

```
EVENTS
Start_vote  $\widehat{=}$
  any  $E$  where
    $sto = no\_elec \ \wedge$
    $E \in ELECTOR - elector$
  then
    $sto := start \ \ ||$
    $e := E \ \ ||$
    $n := 0 \ \ ||$
    $v := \emptyset$
  end ;
Button_cand  $\widehat{=}$
  any  $c$  where
    $sto = start \ \wedge$
    $c \in CAND - \mathsf{ran}(v)$
  then
    $n := n + 1 \ \ ||$
    $v := v \ \cup \{n + 1 \mapsto c\}$
  end ;
Button_valid  $\widehat{=}$
  when
    $sto = start \ \wedge$
    $n \neq 0$
  then
    $sto := valid$
  end ;
One_Vote  $\widehat{=}$
  when
    $st = voting \ \wedge$
    $sto = valid$
  then
    $vt := vt \ \cup \ (\{e\} \times v^{-1}) \ \ ||$
    $elector := elector \ \cup \ \{e\} \ ||$
    $cvt := cvt + 1 \ \ ||$
    $sto := no\_elec$
  end ;
END
```

One_voting is an enumeration set which contains three values: *no_elec* when no electors are voting, and *start* when the a new elector e starts to vote, and *valid* when the elector e pushes the button to validate their vote. The variable *sto* contains one of these values. Variable e contains the current elector, vt is their current vote which is modified when a candidate button is pushed, and n is the number of the chosen candidate.

We remark that the guard of the event Button_valid requires that $n \neq 0$ and so we are sure that when an elector pushes this button then the partial vote v is not empty and so v is a valid vote. Remark also that we have no condition on n in the guard of the event Button_cand. When a candidate is not in the codomain of v we are sure[3] that $n < nbc$.

6.3 Enriching the interface: design2 and design3

In the previous model an elector cannot correct input mistakes: here we choose between alternative designs for correcting mistakes when a candidate button is pressed a second time: (1) delete that candidate's preference (corresponding to `VoteExt1` in the Java), or (2) delete that candidate's preference only if that was the last preference made (corresponding to the suggested "fix" that we wished to formally verify), or (3) delete that candidate's preference and all lower preferences (corresponding to `VoteExt2`).

In the first design, if we remove the corresponding vote and decrement our counter an unproved proof obligation is generated. This is formally treated by the *Button_cancel_incorrect_cand* event. In particular, the new event doesn't preserve the invariant which says that the partial vote v is an injection.

```
Button_cancel_incorrect_cand  $\widehat{=}$
  any  $c$  where
    $sto = start \ \wedge$
    $c \in \mathsf{ran}(v)$
  then
    $n := n - 1 \quad ||$
    $v := v \ \rhd\!\!\!\!- \ \{c\}$
  end ;
```

In section 5, we looked at a design that suggested a "fix" to the previous design. Using B, this "fix" can be formally modelled (in event *Button_cancel_last_cand*) and proven correct. In fact, there are no difficulties to prove the invariant preservation. All new proof obligation are discharged automatically by the prover.

An alternative "fix" to rich interface

Event *Button_cancel_last_cand* models the fix to `design2`.

```
Button_cancel_last_cand  $\widehat{=}$
  any  $c$  where
    $sto = start \ \wedge$
    $n \mapsto c \in v$
  then
    $n := n - 1 \quad ||$
    $v := v \ \rhd\!\!\!\!- \ \{c\}$
  end ;
```

[3] We have proven it thanks to the invariant property and the refinement construct. This proof, as with all others referred to in the paper, is available from the authors on request.

An alternative "correct" rich interface
Event *Button_cancel_cand* models `design3`.

```
Button_cancel_cand  $\widehat{=}$
  any  $c$  where
    $sto = start \wedge$
    $c \in \mathsf{ran}(v)$
  then
    $n := v^{-1}(c) - 1 \quad ||$
    $v := \{n | n \in \mathbb{N} \wedge n < v^{-1}(c)\} \lhd v$
  end ;
```

For this event there was only one difficulty in the proof process: we need to prove that $v^{-1}(c) - 1 = card(\mathsf{ran}(\{n|n \in \mathbb{N} \wedge n < v^{-1}(c)\} \lhd v))$. We have proved this assertion — for all c in the co-domain of v — by simple induction on the assertions clauses.

$$\boxed{(sto \neq no\_elec \Rightarrow v^{-1}(c) - 1 = card(\mathsf{ran}(\{n|n \in \mathbb{N} \wedge n < v^{-1}(c)\} \lhd v))}$$

We note that: in the underlying set theory the inductive theorem is designed in the following manner:

$$\boxed{\forall P \cdot \big(P \subseteq \mathbb{N} \ \wedge\ 0 \in P \ \wedge\ succ[P] \subseteq P \Rightarrow \mathbb{N} \subseteq P\big)}$$

7 The semi-automated proof process

The complexity of the development is evaluated through the number of proof obligations generated for the validation of each model or refinement; among generated proof obligations, a large number are automatically discharged by the tool [12]. In our simple case study, 44 proof obligations are automatically discharged, and 10 are interactively derived using the tool but with human help. A second aspect that should be taken into account is the distribution of proof obligations[4]

through the global process; the first model is very easy to prove since it is very abstract. The refinement model *EachVoteAtOnce* requires two interactive proofs which are quite simple, and only one difficult proof obligation stating the existence of some n which is related to the validity of the vote. The last refinement model is much more complex and the longest proof (requiring inductive

[4] The proof obligations for *Button_cancel_last_cand* and *Button_cancel_cand* are both included in these counts.

reasoning)is 34 steps long. The table illustrates how the complexity of the proof processes grows with each refinement of the model.

Model	Number of proof obligation	Interactively proved
AllVotesAtOnce	3	0
EachVoteAtOnce	16	2
Vote	35	8
TOTAL	54	10

8 Conclusions and Future Work

We have demonstrated the use of the formal method B in guaranteeing a simple safety property of an interface to an e-voting machine. We demonstrated that guaranteeing *validity* of votes recorded not only helps in the formal verification of the voting process, but also has an important role to play in making the machine more secure. The comparison between the B-method and the rapid-prototyping approach with Java is too narrow. We plan to examine other modelling approaches and apply them to the same formal design verification tasks. In particular, we aim to use: a number of *design by contract* tools, automatic code generation techniques, the Unified Modelling Language (UML) and associated verification tools, and at least one formal method where the verification is based on model-checking.

References

[1] J. Abrial. *The B Book - Assigning Programs to Meanings.* Cambridge University Press, 1996. ISBN 0-521-49619-5.

[2] J.-R. Abrial. B$^{\#}$: Toward a synthesis between z and b. In D. Bert, J. P. Bowen, S. King, and M. A. Waldén, editors, *ZB*, volume 2651 of *Lecture Notes in Computer Science*, pages 168–177. Springer, 2003.

[3] J.-R. Abrial and D. Cansell. Click'n prove: Interactive proofs within set theory. In D. A. Basin and B. Wolff, editors, *TPHOLs*, volume 2758 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2003.

[4] J.-R. Abrial, D. Cansell, and D. Méry. A mechanically proved and incremental development of IEEE 1394 Tree Identify Protocol. *Formal Aspects of Computing*, 14(3), Apr. 2003.

[5] J.-R. Abrial, D. Cansell, and D. Méry. Refinement and reachability in event_B. In H. Treharne, S. King, M. C. Henson, and S. Schneider, editors, *ZB*, volume 3455 of *Lecture Notes in Computer Science*, pages 222–241. Springer, 2005.

[6] R. J. R. Back. On correct refinement of programs. *Journal of Computer and System Sciences*, 23(1):49–68, 1979.

[7] I. Bate, R. Hawkins, and J. McDermid. A contract-based approach to designing safe systems. In *CRPIT '33: Proceedings of the 8th Australian workshop on Safety critical systems and software*, pages 25–36, Darlinghurst, Australia, Australia, 2003. Australian Computer Society, Inc.

[8] P. Behm, P. Benoit, A. Faivre, and J.-M.Meynadier. METEOR : A successful application of B in a large project. In *Proceedings of FM'99: World Congress on Formal Methods*, Lecture Notes in Computer Science, pages 369–387, 1999.

[9] D. Cansell, G. Gopalakrishnan, M. Jones, D. Méry, and A. Weinzoepflen. Incremental proof of the producer/consumer property for the PCI protocol. In D. Bert, editor, *Formal Specification and Development in Z and B - ZB'2002, Grenoble, France*, volume 2272 of *Lecture notes in computer science*. Springer Verlag, January 2002.

[10] K. M. Chandy and J. Misra. *Parallel Program Design A Foundation*. Addison-Wesley Publishing Company, 1988. ISBN 0-201-05866-9.

[11] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.

[12] ClearSy. *Web site B4free set of tools for development of B models*, 2004.

[13] D. Farrell and I. McAllister. "the 1983 change in surplus vote transfer procedures for the australian senate and its consequences for the single transferable vote". *Australian Journal of Political Science*, 38(3):479–491, 2003.

[14] A. D. Franco, A. Petro, E. Shear, and V. Vladimirov. Small vote manipulations can swing elections. *Commun. ACM*, 47(10):43–45, 2004.

[15] J. P. Gibson. Feature requirements models: Understanding interactions. In P. Dini, R. Boutaba, and L. Logrippo, editors, *FIW*, pages 46–60. IOS Press, 1997.

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

[17] 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.

[18] I. D. Hill, B. A. Wichmann, and D. R. Woodall. "single transferable vote by meek's method". *Computer Journal*, 30:277–281, 1987.

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

[20] T. Kohno, A. Stubblefield, A. D. Rubin, and D. S. Wallach. Analysis of an electronic voting system. In *IEEE Symposium on Security and Privacy (S&PŠ04)*. IEEE, 2004.

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

[22] 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.

[23] R. Mercuri. *Electronic Vote Tabulation Checks & Balances*. PhD thesis, University of Pennsylvania,School of Engineering and Applied Science, Department of Computer and Information Systems, 2000.

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

[25] R. T. Mercuri. Trusting in transparency. *Commun. ACM*, 48(5):15–19, 2005.

[26] D. Molnar, T. Kohno, N. Sastry, and D. Wagner. Taper-evident, history-independent, subliminal-free data structures on prom storage -or- how to store ballots on a voting machine. In *IEEE Symposium on Security and Privacy (S&P06)*. IEEE, 2006.

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

[28] A. Xenakis and A. Macintosh. Procedural security analysis of electronic voting. In *ICEC '04: Proceedings of the 6th international conference on Electronic commerce*, pages 541–546, New York, NY, USA, 2004. ACM Press.