Modeling Voting Machines

John R Hott

Advisor: Dr. David Coppit

December 8, 2005

Abstract

Voting machines provide an interesting focus to study with formal methods. People want to know that their vote is counted and that the voting machines they are using actually work the way they are supposed to, especially in the age of closed-sourced machines. This project uses PVS to formalize the requirements set forth by the Election Assistance Commission so that this specification can be used in the future to create voting machines with a provable base specification or test current and new voting machines to ensure they function properly under all circumstances. In the end, the model cannot be fully completed due to model size explosion and the need to formalize too much. Insights are given into the PVS verification tool, better ways and tools that can be used to specify the voting machine requirements, and where this specification can be used in the future.

1 Introduction to Electronic Voting

Many states are beginning to use electronic voting machines to capture votes on public election days. This enhances the availability of handicapped accessible voting mechanisms, and keeps human intervention out of vote counting (for the most part), hopefully allowing votes to be more accurate. Currently the United States Election Assistance Commission is working on specifying the requirements of polls and voting machines, and what should happen on voting day. They are updating these requirements to include electronic machines, and include the hardware and software guidelines and requirements of these machines to be certified for use in polls. Using Formal Methods and building a model from these requirements would be a manufacturers best first-move toward building a compatible system efficiently.

2 Why Formalize?

As the US depends more on electronic voting machines, it is imperative that the machines perform correctly. It is also imperative to ensure that the requirements make it into the actual design of systems. Currently, the requirements for voting software correctness only require a source code review and a check to ensure the system flows like the high level design created by the developer. This checking would be more sufficient if it included the comparison and proofs of a model of the software given.

3 What To Formalize?

The portion of the Voluntary Voting System Guidelines set out by the US Election Assistance Commission that would need the most assurance of correctness, at least to match their model, would be Section 2.4, the

functions that must happen at the polls on voting day, including Opening the polls, activating the ballot, casting of ballots, and the closing and counting of the polls. Of the guidelines, this section seems to be the most critical, because it includes everything that happens on voting day and the security involved in that voting.

Certain properties that should be formalized and proved include, but are not limited to, `only_one_ballot_per_eligible_voter`,
`voter_can_only_vote_on_ballot_entitled_to`,
`voter_cannot_vote_twice`,
`voter_can_select_party_affiliation_votes` (which would cast a vote for every member of that party),
`portions_of_ballot_voter_not_entitled_to_are_disabled`,
`system_cannot_reveal_how_a_particular_voter_voted`,
`voter_can_vote_during_failure` and `voter_can_vote_without_network` in accordance with power failures and telecommunications failures (Section 2.4.3.2.e-f),
`voter_can_only_make_legal_combination_of_choices`,
`voter_cannot_overvote`,
`voter_must_review_ballot_before_submit`,
`voter_cannot_access_unauthorized_information`,
`voter_notified_on_submit`,
`votes_stored_represent_votes_cast`,
`vote_cannot_change_after_submit`,
`ballots_unaccessable_until_polls_close`, and
`cannot_cast_ballots_after_polls_closed`.

Each of these properties is essential to ensure proper voting, and likewise, each can crop up in a poorly designed system, which dictates that they should be formalized and checked so that a system build with them cannot violate them. Also, after formalizing these ideas, that formal specification can be translated into code fairly quickly and efficiently.

4 Method

PVS was chosen as the method with which to formalize this voting machine specification. Upon writing and planning the system, it appears that a lighter-weight tool, such as Alloy, should have been chosen, but that will be described more later. Also, there is a method to the way the system should be built. The model should be built up incrementally. This system started with just `Person` and `Ballot` types and a `vote` function that took a person and gave their ballot. It was expanded to include the casting function and a store to keep a record of the votes cast. Then expanded to a way to review ballots, the inclusion of poll states and a poll worker, and finally to part of the user interface. Upon including the user interface, the model started to blow up in size, and so only part of it was included.

5 Model

The model itself is both simple and complex. It includes an abstract non-empty type, left abstract because it cannot be modeled. This is of course, `Person`. `Poll_Worker` and `Eligible_Voter` were extended from the `Person` type because there are people that work at polls and that can vote, but not every person can vote. `Candidate` should have been of type `Person`, but to ease the proofs, it was declared an enumerated type

with the candidates up for election. In my model, I used Washington and Monroe. Other enumerated types are `Poll_State` that tells whether polls are open or closed, the `Operation_State` which is unused but tells whether the system is under normal or failure mode of operation, and `UI_Input` which tells what input the user has issued: cancel, submit, and others.

There are also a few record types defined. `Ballot` is the basic record type, which has a selection of Candidate in it. This could be expanded to include President, Vice President, etc, but for simplicity of our model, we just have one selection the user can make. `Ballot_Store` is a step up from ballots, which is a record type that contains a count of the ballots it contains and a relation that contains all the ballots, mapped number to ballot, that the store contains.

5.1 Conventions

Some conventions must be made in order for our model to actually work. They are basically empty types, which means that either nothing is returned because it should not be allowed or not be counted. Blank, that is a ballot, is one of these conventions, because if a user doesn't submit they still have to return a ballot but it must not have a vote. Others include empty ballot stores and an empty candidate.

5.2 Functions

Many simple functions build up to the vote function that will be described in the next section. `Entitled_Ballot(v: Eligible_Voter): Ballot` is the first function, and it returns the ballot that the eligible voter is entitled to. It is left undefined because it cannot easily be modeled. Similar to this function is `Get_Candidates_In_Voters_Choice_Party(v: Eligible_Voter): Candidate` which takes a voter and returns the candidates in their party of choice. This design decision was taken because the initial design had People and Candidates having Parties, but the description of this function became too complex and this one was chosen. The simple choose function, `choose(v: Eligible_Voter): Ballot`, lets the voter choose what they want to pick on the ballot and return the ballot they have chosen. At the basic level, this is how a user votes.

5.3 Voting

To vote, a person must be an `Eligible_Voter`. The person calls the vote() function,

```
vote(v: Eligible_Voter): Ballot =
  voter_review(choose(v), submit)
```

which says that voting returns a ballot that the person chooses, however, they must choose who they want to vote for and review the ballot, pressing submit, for the ballot to be counted. This review function, shown below, returns the ballot the voter chose if they input submit, but if they don't, the blank ballot is returned, which means they're vote is not counted.

```
voter_review(b:Ballot, ui: UI_Input): Ballot =
  IF ui = submit THEN
    b
  ELSE
    blank
  ENDIF
```

This voting scheme must be done in mass, by multiple voters, and it has to be stored, or else the voting machine would be useless. The `Cast_Ballot` function below describes how this process takes place.

```
Cast_Ballot(ps: Poll_State, voter: Eligible_Voter, bs: Ballot_Store): Ballot_Store =
  IF ps = closed THEN
     bs
  ELSE
     (# count := bs`count + 1, ballots := bs`ballots WITH [(bs`count) := vote(voter)] #)
  ENDIF
```

Casting a ballot can only happen with an eligible voter, but it can also only happen if the polls are open. So, if the polls are closed the new vote is not counted and the old ballot store is returned. If the vote can be counted, the count in the store is updated and the user's vote is added to the store's ballots relation.

5.4 Review

The `Review_Ballots` function is similar to the `Cast_Ballot` function.

```
Review_Ballots(ps: Poll_State, bs: Ballot_Store, person: Person): Ballot_Store =
  IF ps = closed AND person = Poll_Worker THEN
     bs
  ELSE
     empty
  ENDIF
```

It takes a Person rather than an Eligible Voter because the Poll Worker can be just a Person. Also, this works opposite of the `Cast_Ballot` function. If the polls are closed and the person is the poll worker, then they can see the ballot store and it is returned. Otherwise, either the polls are open or there is a voter trying to access the votes, and they only get an empty store because they are not allowed to see the actual store.

6 Theorems

As the model was growing, lemmas were added based on the requirements of the Election Accessibility Commission to ensure that the model worked according to their guidelines. Each of these lemmas could be easily proven with PVS's `grind` function and one with `induct-and-simplify`. So, the proofs are uninteresting and can easily be recreated, so they will not be included. Also, we will examine some of the more interesting lemmas, but we will not go into detail over all of them since there are 16 proofs, which are too many to cover in great detail.

6.1 Time to Cast and Review Ballots

```
ballots_unaccessable_until_polls_close: LEMMA
  FORALL (bs: Ballot_Store, ballot_num: nat):
    Review_Ballots(open, bs, Poll_Worker) = empty

cannot_cast_ballots_after_polls_closed: LEMMA
  FORALL (voter: Eligible_Voter, bs: Ballot_Store):
```

```
    (Cast_Ballot(closed, voter, bs))`count = bs`count AND
        (Cast_Ballot(closed, voter, bs)`ballots = bs`ballots)
```

These two lemmas are very interrelated. They both deal with the passage of time, and ensuring that the model performs correctly under different times. The first lemma proves that the ballots can not be accessed until after the polls have closed. For every store of ballots, if the Poll_Worker tries to review the ballots while the Poll_State is open, they should only get the empty store, which means that they cannot access any ballots. One expansion would be to add `Review_Ballots(closed, bs, Poll_Worker) = bs`, which would check the opposite, that if they review the ballots during the closed state, they will get the ballot store to review. It, however, seemed simple, and was excluded.

The second lemma is similar to the first. It states that if a voter votes while the polls are closed, that the count and ballots of the ballot store do not change. That is, that the voter's vote is not counted. The opposite, voting in an open state, is not checked here because it is included in the next lemma we will look at, ballots stored represent the ballots that were cast. Both of these were easily proven with `grind`.

6.2 Ballots Stored Represent Ballots Cast

```
multiple_cast_ballot(bs: Ballot_Store, voters: list[Eligible_Voter]): RECURSIVE Ballot_Store =
  IF (voters = null) THEN
    bs
  ELSE
    LET (first_voter, remaining_voters) = (car(voters), cdr(voters)) IN
      Cast_Ballot(open, first_voter, multiple_cast_ballot(bs, remaining_voters))
  ENDIF
MEASURE length(voters)

votes_stored_represent_votes_cast2: LEMMA
  FORALL (voters: list[Eligible_Voter], bs: Ballot_Store):
    FORALL (v: Eligible_Voter):
      multiple_cast_ballot(bs, voters)`ballots
        WITH [(multiple_cast_ballot(bs, voters)`count) := vote(v)] =
        Cast_Ballot(open, v, multiple_cast_ballot(bs, voters))`ballots
```

Perhaps the most interesting lemma proven is this version of `votes_stored_represent_votes_cast`. This version of the lemma requires the `multiple_cast_ballot` function to ensure that all ballots are cast. Multiple cast ballot recursively applies votes from a list of voters, filling up a ballot store with ballots. When the length of the list voters reaches 0, the recursion will stop. The actual lemma checks to see that after all the votes are cast from `multiple_cast_ballot`, they are still in the ballot store after another vote has been cast. That ensures that the votes stored are the actual votes that were cast, and there are not any extra ballots that were added. This lemma took a call to `induct-and-simplify` to solve.

6.3 Vote Cannot Change After Submit

```
vote_cannot_change_after_submit: LEMMA
  FORALL (v: Eligible_Voter, bs: Ballot_Store):
    choose(v) = voter_review(choose(v),submit) AND
    choose(v) = (Cast_Ballot(open, v, bs))`ballots(Cast_Ballot(open, v, bs)`count-1)
```

We will examine this lemma because even though it is a simple lemma, it is fun to examine. Very simply, this lemma is supposed to ensure that the vote is not changed after it has been submitted. So, for every voter, they choose their choice ballot, `choose(v)`. After they review their ballot, `voter_review(choose(v), submit)`, and submit it, it should still be the same (the first line). Also, after it is cast, `(Cast_Ballot(open, v, bs))`, the ballot that is in the store should be the same as the one they chose (the second line). Since it examines the ballot in multiple different locations in the process of voting, it is an interesting lemma to examine, however, it can easily be proven correct by `grind`.

6.4 Voter Cannot Access Unauthorized Information

```
voter_cannot_access_unauthorized_information: LEMMA
  FORALL (v: Eligible_Voter, bs: Ballot_Store, ps: Poll_State):
    v /= Poll_Worker => Review_Ballots(ps, bs, v) = empty
```

Finally, we will examine the inability of a user to access unauthorized information. This lemma, which says that if any voter tries to review the ballots under any circumstances, and they are not the poll worker, then they only get the empty ballot and are not able to view the ballot store with the ballots in it. This probably should have been expanded to say that this is true for all people, but we made an assumption that only eligible voters would be allowed in to use the machine. Whether we state it as for all voters or for all people, though, it is easily and quickly proven with `grind`.

7 Surprises and Challenges

There were many surprises and challenges that arose in completing this model. Firstly, modeling in PVS is difficult. Understanding how the system should work and how to structure it are extremely difficult, and they both must be taken into consideration as the theorems and lemmas to prove are created; they are all related.

Also, implementing the model using an incremental method is very useful. It turns out that as more functionality is added to the model, the size of the model grows exponentially. For example, to increase the model from the point it is now to the next step, the User Interface, the system would have to take a ”step back” from what it is now. That is, to include the User Interface with all its functionality, it would have to be rewritten as a state machine to encompass a version of the model as it is now in each of the states. Choosing to vote and going to the Vote state would require the user to vote, calling the `Cast_Ballot` function but would also have to modify the `choose` function to interact with the user and then have the completion switch to a different state. The Review state would have a similar complexity. Therefore, as can be seen from this small example, as the model is grown, much more would have to be taken into account at each step. Because of this realization, not all of the lemmas proposed were actually completed – to implement them would require an exponential change to the model to complete and prove them.

8 Insights into PVS

PVS is a very powerful, yet very complex and confusing language. This project has given me some very interesting insights into the use of PVS in modeling of a real world system. Basically, I have learned that modeling in PVS is hard in specifying the model and yet easy when actually trying to prove that model.

First of all, creating a specification is difficult in PVS. The learning curve for PVS is large, but at the same time, once it is mastered, PVS can be extremely powerful. One difficulty is to move away from thinking like a programmer because PVS functions can only be defined with one expression. Therefore, a function cannot do multiple things like a program's function can do, but all the power of a function must be expressed in one mathematical expression. That can most easily be done with an IF ... ELSE ... ENDIF statement. Another barrier to overcome was to figure out how to model the system and not have it expand beyond comprehension. The more I modeled in PVS, the more I needed to model to keep the lemmas provable and the function calls also had to be expanded. I think PVS needs a definite design decision before building up the specification, because it cannot be changed or modified as easily as code.

Another insight into PVS is that general proofs are easy. Every lemma I wrote could easily be proven with `grind` or `induct-and-simplify`, PVS's big hammers. That means that once a system is formalized in the PVS syntax, PVS is extremely helpful and powerful at proving the lemmas that need to be proven. That realization was a blessing while working on this project. It meant that if the lemma could not be proven with `grind`, then there was either a problem with the lemma or a problem with the part of the model it was trying to prove.

9 Where To Go From Here

This model is undoubtedly incomplete. There is so much more that can be added to it, including a complete User Interface as well as including Network and Power running through the system so that it can deal with power and network failures. Adding a user interface can more easily be done in a state based system such as Alloy or by using UML because states are extremely difficult in PVS. Given the opportunity to recreate this project, a better design decision would have been to create the entire model in a lightweight language, such as Alloy, because it would have been easier to complete the model and include the User Interface without using multiple tools. However, once the model has been fully completed, we can use it for many very useful purposes.

9.1 Check Current Systems

Firstly, the model can be used to generate inputs to test current systems, such as the DieBold voting machines that have been employed but have closed source coded. The results would show that the system works without the source being necessary and could be published to ease voters' minds that their vote was counted. The inputs could be generated by hand or by an automatic generator, such as TestEra.

9.2 Build New Systems

The model could also be refined into code, which will generate a machine that we know would work right, according to the Election Accessibility Commission's guidelines, and has a proven base, this specification. We can still generate the inputs and tests to check its correctness, but the specification and proofs can also be public, allowing users to see that the code is correct.

Appendix

PVS Model

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Model
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Person: TYPE+
Candidate: TYPE+ = {Washington, Monroe, empty}
Party: TYPE+ = Candidate

Ballot: TYPE+ = [# selected: Candidate #]
blank: Ballot

Eligible_Voter: TYPE+ = Person
Entitled_Ballot(v: Eligible_Voter): Ballot
Get_Candidates_In_Voters_Choice_Party(v: Eligible_Voter): Candidate

Poll_Worker: Person

Poll_State: TYPE+ = {open, closed}
Operation_State: TYPE+ = {normal, failure}
UI_Input: TYPE+ = {submit, cancel, vote, read_ballots}

choose(v: Eligible_Voter): Ballot

voter_review(b:Ballot, ui: UI_Input): Ballot =
  IF ui = submit THEN
    b
  ELSE
    blank
  ENDIF

vote(v: Eligible_Voter): Ballot =
  voter_review(choose(v), submit)

Ballot_Store: TYPE+ = [# count: nat, ballots: [nat -> Ballot] #]
empty: Ballot_Store


Cast_Ballot(ps: Poll_State, voter: Eligible_Voter, bs: Ballot_Store): Ballot_Store =
  IF ps = closed THEN
     bs
  ELSE
     (# count := bs`count + 1, ballots := bs`ballots WITH [(bs`count) := vote(voter)] #)
  ENDIF
```

```
% Review Ballots at the end of the day (count them)
Review_Ballots(ps: Poll_State, bs: Ballot_Store, person: Person): Ballot_Store =
  IF ps = closed AND person = Poll_Worker THEN
     bs
  ELSE
     empty
  ENDIF
```

PVS Proofs

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Lemmas and Conjectures to prove
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

voters_vote_is_counted: LEMMA
  FORALL (voter: Eligible_Voter, bs: Ballot_Store):
     (Cast_Ballot(open, voter, bs))`count = bs`count + 1 AND
         (Cast_Ballot(open, voter, bs)`ballots(bs`count) = vote(voter))

only_one_ballot_per_eligible_voter: LEMMA
  FORALL (p1: Eligible_Voter):
    EXISTS (b1, b2: Ballot): (vote(p1) = b1 AND vote(p1) = b2) => b1 = b2

% Redundant, but gets the point across
voter_cannot_vote_twice: LEMMA
  FORALL (p1: Person):
    EXISTS (b1, b2: Ballot) : (vote(p1) = b1 AND vote(p1) = b2) => b1 = b2

voter_can_select_party_affiliation_votes: LEMMA
  FORALL (p1: Person):
    EXISTS (b1: Ballot) : (b1`selected = Get_Candidates_In_Voters_Choice_Party(p1)) =>
          vote(p1) = b1

voter_cannot_overvote: LEMMA
  FORALL (voter: Eligible_Voter):
     EXISTS (c1, c2: Candidate): (vote(voter))`selected = c1 AND (vote(voter))`selected = c2 =>
            c1 = c2

votes_stored_represent_votes_cast: LEMMA
  FORALL (voter: Eligible_Voter, bs: Ballot_Store):
    bs`ballots WITH [(bs`count):= vote(voter)] = Cast_Ballot(open, voter, bs)`ballots

multiple_cast_ballot(bs: Ballot_Store, voters: list[Eligible_Voter]): RECURSIVE Ballot_Store =
  IF (voters = null) THEN
    bs
```

```
  ELSE
    LET (first_voter, remaining_voters) = (car(voters), cdr(voters)) IN
      Cast_Ballot(open, first_voter, multiple_cast_ballot(bs, remaining_voters))
  ENDIF
MEASURE length(voters)

votes_stored_represent_votes_cast2: LEMMA
  FORALL (voters: list[Eligible_Voter], bs: Ballot_Store):
    FORALL (v: Eligible_Voter):
      multiple_cast_ballot(bs, voters)`ballots
        WITH [(multiple_cast_ballot(bs, voters)`count) := vote(v)] =
        Cast_Ballot(open, v, multiple_cast_ballot(bs, voters))`ballots

vote_cannot_change_after_submit: LEMMA
  FORALL (v: Eligible_Voter, bs: Ballot_Store):
    choose(v) = voter_review(choose(v),submit) AND
    choose(v) = (Cast_Ballot(open, v, bs))`ballots(Cast_Ballot(open, v, bs)`count-1)

ballots_unaccessable_until_polls_close: LEMMA
  FORALL (bs: Ballot_Store, ballot_num: nat):
    Review_Ballots(open, bs, Poll_Worker) = empty

cannot_cast_ballots_after_polls_closed: LEMMA
  FORALL (voter: Eligible_Voter, bs: Ballot_Store):
     (Cast_Ballot(closed, voter, bs))`count = bs`count AND
         (Cast_Ballot(closed, voter, bs)`ballots = bs`ballots)

voter_must_review_ballot_before_submit: LEMMA
  FORALL (v: Eligible_Voter):
    voter_review(choose(v), submit) = choose(v) AND
    voter_review(choose(v), cancel) = blank

voter_cannot_access_unauthorized_information: LEMMA
  FORALL (v: Eligible_Voter, bs: Ballot_Store, ps: Poll_State):
    v /= Poll_Worker => Review_Ballots(ps, bs, v) = empty

% Only entitled to a ballot if you're an eligible voter!
voter_can_only_vote_on_ballot_entitled_to: LEMMA
  FORALL (p1: Person):
    EXISTS (v: Eligible_Voter): p1 = v =>
      EXISTS (b1: Ballot) : choose(p1) = b1 => b1 = vote(p1)
```

Bibliography

Election Accessibility Commission, ”Voluntary Voting System Guidelines,”
http://www.glynn.com/eac_vvsg/inro.asp.