Designing a Voting Machine for Testing and Verification

Cynthia Sturton

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2012-253
http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-253.html

December 14, 2012

Acknowledgement

Many thanks to my collaborators: Susmit Jha for his help with the verification; Sanjit Seshia for his expertise in formal methods; and David Wagner for his overall advice, and in particular, his guidance on writing proofs.

Designing a Voting Machine for Testing and Verification

by Cynthia Sturton

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor David Wagner
Research Advisor

(Date)

* * * * * * *

Professor Sanjit A. Seshia
Second Reader

(Date)

Designing a Voting Machine for Testing and Verification

Cynthia Sturton
University of California, Berkeley
csturton@cs.berkeley.edu

Abstract

This work incorporates and builds on previous research done jointly with Susmit Jha, Sanjit Seshia, and David Wagner on designing an electronic voting machine with the goal of verification of correctness [28]. In that work we developed an approach of combining formal verification with user testing to verify an interactive machine and we demonstrated our technique with the design and implementation of a voting machine. This paper presents our work with a focus on the methodology and set of design principles we developed which made our approach possible. This also extends the functionality of our original voting machine to include a summary screen while still adhering to our methodology and design principles. We implement the new functionality and demonstrate that our original proof of correctness holds for the augmented voting machine.

1 Introduction

Electronic voting machines provide convenience and usability features that make them a popular choice among voting officials. One particular type of electronic voting machine is the Direct Recording Electronic voting machine (DRE). A DRE presents an electronic ballot to the voter as a series of screens representing different contests on the ballot. For each contest, the voter can make her selections using the DRE's input device (e.g., buttons or touch screen) and the DRE records her vote in electronic form.

DREs can provide a variety of features, which make them an appealing option for election officials choosing among voter-marked ballots, mechanical lever machines, and electronic voting machines [32]. A DRE can be easily configured for a particular precinct, presenting only those local contests relevant to that precinct. Each cast ballot is stored electronically, removing any ambiguity that can result from a voter incorrectly or incompletely marking a ballot. DREs can also provide usability features that paper and mechanical ballots can not. They can be programmed to present the ballot in a variety of languages; and alternate input and output devices can be provided to accommodate a variety of physical needs. For example, audio output can be provided for voters who have trouble seeing. These usability features make it possible for people who might otherwise require aid to now place their vote privately.

Statistics from VerifiedVoting.org, an online non-profit election watchdog, show that for the November 2010 U.S. elections, 33% of registered voters were using DREs [30]. With DREs in use for such a sizable portion of the electorate, it is important that we have confidence they are correctly recording votes according to the voters' intent. However, DREs are typically complex programs, often tens of thousands of lines of code [7, 32], and a single bug in the program can potentially lead to the DRE malfunctioning, misrecording votes, losing votes, or providing confusing feedback to the voter in the form of surprising or ambiguous displays. We have seen many examples of such malfunctions in the news in recent years [10, 29]. We present here a method for verifying the correctness of the voting machine. We use well known formal verification techniques to prove the correctness of our implementation. What is novel about our approach is our technique for defending against that third form of malfunction, an error in the user interface. We involve a tester in our verification and use a combination of formal verification plus testing that allows us to prove our DRE will behave correctly on election day for any sequence of inputs the voter might provide.

Formal verification techniques can guarantee an implementation meets its specification. However, it can be difficult to formalize the complete specification for an interactive device such as a DRE. For example, if the voter sees the name "Alice" on the screen and a shaded rectangular region next to "Alice," the voter might expect that pressing somewhere in the shaded region would select candidate Alice for that contest. If, after

pressing somewhere in the shaded region, the voter sees that "Alice" has become highlighted, the voter might interpret this to mean the voting machine has recorded her vote for Alice. Specifying those expectations formally is difficult and likely error-prone. Therefore we employ user testing to verify the interaction of the voting machine with voters.

Currently DREs undergo extensive testing prior to election day. Testing requires no formal specification and, by involving a human in the loop, can show that an interactive device is behaving as a user would expect. However, testing alone is not sufficient to prove the machine will behave correctly on election day. A ballot with N contests in which voters must choose 1 out of k candidates for each contest can be marked in k^N possible ways, which is too large to exhaustively test for typical values of k and N. Even if a tester could check all k^N test cases, and all tests pass, we have no guarantee that the machine will behave in the same way on election day as it did during testing. Even assuming for the moment we know the voting machine is deterministic on the set of input signals available to the tester, these tests are still insufficient to prove the correctness of the machine. At each contest the voter has the choice of moving to the next contest, moving back to the previous contest, or moving to the summary screen. These options combine to provide an infinite number of ways a voter might navigate through the ballot to the final cast state. With infinitely many possible input traces, it is impossible for any finite amount of testing to rule out the possibility of some hidden functionality existing for a particular input trace.

Instead of relying solely on testing to validate the behavior of the machine, we use a combination of formal verification plus testing. We design the DRE to be deterministic on its user inputs and verify it is so using an SMT solver. We structure the voting machine so that we can prove a finite number of user tests are sufficient to prove the correctness of the configured voting machine and use an SMT solver to show the machine is structured as required by our proof. We provide guidelines for identifying the set of test cases which will be sufficient for our proof. We show that the number of tests required is polynomial in the number of contests on the ballot.

The goal in this research is to build a provably correct DRE. However, this alone will not guarantee a secure voting system. Such a guarantee would require securing everything from the machines used to capture the voters' intent and the tabulator used to tally up the votes to the training of the poll workers on election day and the transport of ballots and machines to and from the polling place [16]. Still, proving the correctness of the DRE is a necessary step toward proving the security of any voting system using it.

1.1 Contributions

This work includes and builds on previously published research [28]. In this section we highlight the contributions of this report.

We provide in this report the full design and detailed specification of our implementation of a DRE, identifying the set of design principles we adhered to in order to make our use of verification and testing possible.

We extend the voting machine presented in our previous work to include a summary screen as found in currently deployed DREs. Maintaining our correctness guarantees requires some additional verification and new test cases be introduced into our procedure. We provide the details of these additional measures in Sections 8 and 9.

1.2 Notation and definitions

Before continuing, we briefly define some voting-related terms that are used throughout the discussion.

Contest: A single race, such as presidential, for which a voter will make a selection.

Ballot: The physical or electronic representation of all contests that a voter will be deciding on election day.

Candidate: A choice in a particular contest. The voter will typically choose from among two or more candidates for each contest on the ballot.

Voting Session: A voter's interaction with the machine from the time they are given a new ballot until the time their entire ballot is stored in non-volatile memory, i.e., until the time they cast the ballot.

Cast: Casting a vote refers to the action taken at the end of a voting session that causes the selections made in all contests to be irrevocably written to non-volatile memory. Making a selection in a particular contest and moving on to the next contest is *not* considered casting a vote.

Selection State: The state representing the set of all candidates currently selected in a particular contest.

Button: A (usually rectangular) region on the screen. Touching anywhere within this region activates a particular functionality of the machine. The corresponding part of the screen image is often designed to provide the appearance of a physical button. Navigation buttons help move from contest to contest. Selection buttons help control the selection state of the currently active contest.

Mode: In this work, a voting machine can be in one of three modes: main mode, in which selections for each contest are made; summary mode, in which the voter can see a summary of all selections in every contest; and cast mode, in which the vote has been cast and the voting session is over.

2 Methodology

Our goal is to prove the voting machine will behave correctly on election day. However, it is not obvious how best to define "correct" for an interactive machine such as a DRE. Typically, a correct machine is one which meets its specifications exactly, but for an interactive device this definition is not sufficient. An interactive machine is really only useful if it behaves in a way that meets users' expectations.

We start with the premise that during an interactive session with a particular DRE a voter will maintain some mental model for the internal state of the voting machine given the user interface (UI) displayed to the voter at each point in the session. The mental model will incorporate both the particular UI displayed to the voter and the voter's own preconceived assumptions about how a voting machine should work. The first step in our methodology is to define a set of properties we expect the voters' model to always have, regardless of the UI of any particular voting machine. An example of one such property is: selecting a candidate in one contest has no effect on the set of selected candidates for any other contest. In order to make our use of formal verification and testing feasible we have to assume every voter's mental model will satisfy these properties. We therefore want our list of properties to be minimal while still being sufficient to make our method of testing and formal verification possible. It is possible to imagine a voting machine and UI that lead the voter to a mental model that does not satisfy all of our properties. A realistic example is straight-party voting in which making a selection on the first "contest," i.e., selecting the party, *does* affect selections in all other contests. We currently do not handle such cases. The DRE we build does not provide such options and we assume the properties we require hold for any voter's mental model when interacting with our DRE.

The second step in our methodology is to construct an abstract model of a voting machine that satisfies the required properties we defined in step one. We call this a canonical voting machine $\mathcal{C}$. We design the canonical voting machine $\mathcal{C}$ such that the properties are satisfied by construction. We use this model to represent the voter's mental model in our proof of correctness. Note that our canonical voting machine represents only one possible design that satisfies the properties we require; yet, we fix $\mathcal{C}$ and assume the voter is using this mental model. However, $\mathcal{C}$ specifies nothing about the users' interpretation of the voting machine's input and output display and it is this interpretation that we leave unspecified and instead verify through testing. With a well defined model of voter's expectations in place, we move to the third step in our methodology: defining a notion of correctness that incorporates voters' expectations about how the DRE should behave.

The fourth step in our methodology is to design and implement an actual voting machine. The design for our voting machine is based closely on the canonical voting machine we developed in step two. Before starting the implementation, we fully specify the expected behavior of the voting machine under all possible input conditions. We implement the DRE in Verilog, a hardware description language, and synthesize our design onto an FPGA board with an attached touchscreen daughterboard. We use model checking to formally verify the implementation meets its specification and an SMT solver to formally prove our implementation satisfies our set of required properties. All of our verification is done directly on the source code of the machine, not on a model of the machine.

Our fifth and final step is to provide a formal proof that a voting machine satisfying the properties we identify is structured such that a finite amount of user testing is sufficient to prove the machine will behave according to voters' expectations on election day. In our proof we use the canonical voting machine we construct in step two of our methodology to represent the voter's mental model of a voting machine and the implementation we built in step four to be our prototypical voting machine whose correctness we are interested in proving. Our proof requires a set of user-based test cases satisfying certain criteria. We describe those criteria and give a satisfying example suite of test cases.

In the following sections we delve in to the details of each step in our methodology.

3 Required Properties

We define here a set of properties we believe any voter would implicitly expect to hold true. The purpose of these properties is two-fold:

- Provide a set of guidelines to use when building our DRE. If every voter would expect these properties to be true of any voting machine, it only makes sense to design the machine so that it provably obeys the properties.
- We use these properties to motivate the definition of the voter's expectations about the voting machine's behavior in the formulation of our proof of correctness.

Because each of these properties represents an assumption we are making about voters' expectations, we include only those properties that are reasonable to assume and are necessary for our proof. These properties are:

P_0: The voting machine is a deterministic transducer.

P_1: The state of a contest is updated independently of the state of any other contest.

P_2: If a navigation button is pressed, the selection state remains unchanged.

P_3: If a selection button is pressed, the current contest number remains unchanged; the selection state of the current contest might be altered, but not any other.

P_4: When interacting with the candidate selection interface, the output function is an injective function of the current mode of the machine, the current contest number, and the selection state of the current contest.

P_5: The electronic cast vote record is an accurate record of the selection state for each contest.

Note that Property P_4 makes no assumption about the particulars of the output function or the display presented to the voter. It only requires that the display for a given mode, contest number and selection state be consistent and unique to that state.

4 $\mathcal{C}$: The Canonical Voting Machine

We describe here canonical voting machine $\mathcal{C}$, which satisfies properties P_0 – P_4 above. We assume the tester's mental model matches $\mathcal{C}$ and use this in our proof.

The formalization of the canonical voting machine presented here is very similar to the model first described in previous work [28], with modifications made to include the notion of a summary screen.

We define $\mathcal{C}$ as a deterministic finite-state transducer. $\mathcal{C}$ is defined as a 6-tuple $(\mathcal{I}, \mathcal{O}, \mathcal{S}, \delta, \rho, s_{\text{init}})$ where

- $\mathcal{I}$ is the set of input signals from the voter,
- $\mathcal{O}$ is the set of outputs from the voting machine,
- $\mathcal{S}$ is the set of states of the voting machine,
- $\delta : \mathcal{S} \times \mathcal{I} \to \mathcal{S}$ is the transition function,
- $\rho : \mathcal{S} \to \mathcal{O}$ is the output function, and
- $s_{\text{init}} \in \mathcal{S}$ is the initial state of $\mathcal{C}$.

The structure of $\mathcal{C}$ is shown in Figure 1. $\mathcal{C}$ operates in three modes, *main*, *summary*, and *cast* and is composed of three corresponding transducers: T_{main}, T_{summary}, and T_{cast}. The first, T_{main}, is itself a

composition of $N+1$ transducers. $T_{\text{controller}}$ is the main transducer that is always active whenever T_{main} is active. It is responsible for the progress of the voting machine through each contest. There is one transducer, T_i, for each contest on the ballot. $T_{\text{controller}}$ activates only one T_i at a time. Each T_i is responsible for maintaining the state for contest i. The state for contest i can change according to the voter's input when, and only when, T_i is active. When T_{main} is active, the output is the current contest number and the state of that contest: (i, s_i). $\mathcal{C}$ starts in T_{main} with T_1 active and all selection states zeroed out. T_{main} can only pass control to T_{summary}. T_{summary} has a single state, and its output is a list of the state of every T_i: $[(1, s_1), (2, s_2), \ldots, (n, s_n)]$. T_{summary} can pass control back to T_{main} or on to T_{cast}. T_{cast} has a single state and outputs the permanent recording of the state of each T_i. The state of T_{cast} is the final state of $\mathcal{C}$. A reset is required to start the next voting session, but the reset event is not modeled as $\mathcal{C}$ models exactly one voting session.

The inputs are partitioned into two sets: $\mathcal{I} = \mathcal{I}_N \cup \mathcal{I}_S$. $\mathcal{I}_N$ is the set of navigation signals: those inputs that cause control to pass to a different transducer. In particular, $\mathcal{I}_N = \{\texttt{next}, \texttt{prev}, \texttt{summary}, \texttt{resume}, \texttt{cast}\}$. The set $\mathcal{I}_S$ defines the inputs that can be used to select or deselect candidates in each contest. Thus, $\mathcal{I}_S = \{0, 1, \ldots, k-1\}$ where k is the maximum number of candidates in any single contest.

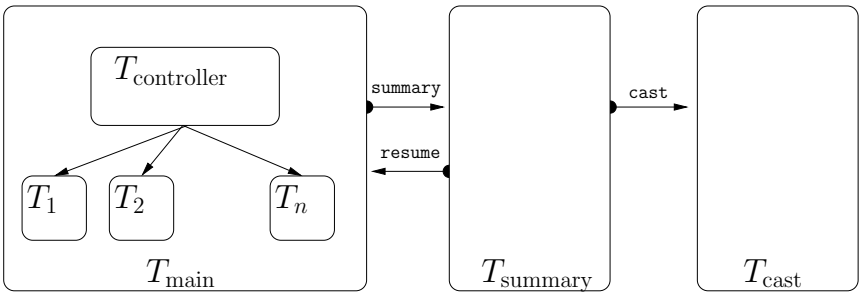


Figure 1: The canonical voting machine $\mathcal{C}$. We show the high-level structure of $\mathcal{C}$

4.1 Interpretation Function

In Section 2 we claimed a voter would maintain a mental model of the internal state of the DRE throughout a voting session and this model would be informed both by the voter's own assumptions about how the DRE should work, and by the output displayed by the DRE. It is through the DRE's output screen that the voter learns about the current internal state of the machine.

The canonical voting machine tracks the state of the voting machine in terms of (m, i, s_i), that is, mode, current contest number and state of that contest; $\mathcal{C}$ does not produce formatted output screens. A typical DRE does not explicitly output those three signals. Instead it displays a screen that the voter can read and interpret. Thus, we need some way to relate $\mathcal{C}$ to the actual DRE.

To draw this connection, we introduce the notion of an interpretation function I. The purpose of I is to relate the concrete signals (pixels displayed) produced by the DRE to the abstract signals of the voter's mental model $\mathcal{C}$. I encodes which (x, y) coordinate maps to which input buttons and how every possible screen display maps to internal states. We define I by reference to a thought experiment wherein a hypothetical voter interacts with a DRE. Suppose the voter is confronted with a screen z representing one contest on the ballot and is asked which contest the screen is displaying and which candidates on that contest have been selected. A typical voter would implicitly and immediately have some notion about the internal state of the machine given that screen. We define this as the voter's output interpretation function $I_{\mathcal{O}}(z) = (m, i, s_i)$. Similarly suppose someone pointed to an (x, y) coordinate on screen z and asked the voter which input button that location corresponded to. The voter would use the current display shown to interpret the meaning of a particular (x, y) coordinate and relate it to an abstract button $b \in \mathcal{I}$. This defines the voter's input interpretation function $I_{\mathcal{I}}(z, (x, y)) = b$.

We do not attempt to write down the interpretation function I explicitly, as it is election-dependent and likely to be very complex to specify. However, we do make one important assumption. We assume that there does exist a single interpretation function and every voter will have the same input and output

interpretation functions. In other words, for a given screen, every voter would interpret the current state of the DRE in the same way. This is a strong assumption to make and it is one we rely on for the construction of our proof of correctness. We do not claim the assumption is necessarily always valid, but by making the assumption clear, we underscore which parts of an implementation would need careful attention. We discuss some techniques that might increase the validity of the assumption in Section 10.2.

5 Defining Correctness

We consider a voting machine correct if it will behave in a way that comports with voters' expectations. To make this more precise, we assume the existence of the voter's mental model about the internal state of the machine and compare the voter's mental model to the actual internal state of the machine at every point in a voting session. Intuitively, we say the machine is correct if they match.

A trace of the canonical voting machine $\mathcal{C}$ is a sequence of outputs and inputs $(z_0, b_1, z_1, b_1, \ldots, z_l)$, where $b_i \in \mathcal{I}$ and $z_i \in \mathcal{O}$. A complete trace $\tau_\mathcal{C}$ of $\mathcal{C}$ is a trace $\tau_\mathcal{C} = (z_0, b_1, \ldots, z_l)$ where $b_i \in \mathcal{I}$, $z_0, \ldots, z_{l-2} \in \mathcal{O}_{\text{main}} \cup \mathcal{O}_{\text{summary}}$, $z_{l-1} \in \mathcal{O}_{\text{summary}}$, and $z_l \in \mathcal{O}_{\text{cast}}$. A complete trace $\tau_\mathcal{A}$ of the actual voting machine $\mathcal{A}$ is a sequence of outputs and inputs $\tau_\mathcal{A} = (z_0, b_1, \ldots, z_l)$ where each output $z_0, \ldots, z_{l-1}$ is a screen display, z_l is the final cast vote record, and each input b_j is a touch by the voter on the touch screen, represented as an (x, y) coordinate. Define $I(\tau_A)$ to be the application of the voter's interpretation function to the trace of $\mathcal{A}$:

$$I(\tau_A) = (I_\mathcal{O}(z_0), I_\mathcal{I}(b_1, z_1), I_\mathcal{O}(z_1), I_\mathcal{I}(b_2, z_2), I_\mathcal{O}(z_2), \ldots, I_\mathcal{I}(b_l, z_l), I_\mathcal{O}(z_l))$$

We say $\mathcal{A}$ is correct if, for every possible complete trace τ_A of $\mathcal{A}$, $I(\tau_A)$ is a valid complete trace of $\mathcal{C}$.

6 $\mathcal{A}$: Our Voting Machine

We used $\mathcal{C}$ as the guide for our design and implementation of a prototype voting machine $\mathcal{A}$. In this section we describe the details of that design. We start with an explanation of our design principles. We then describe the organization of the voting machine followed by a full specification of each module in the machine and the behavioral and structural properties of the composition of those modules. These properties help us verify that $\mathcal{A}$ is equivalent to $\mathcal{C}$.

6.1 Design Principles

Using $\mathcal{C}$ as our guide, we developed the following design principles.

- Keep the entire state of the voting machine small and well defined.
- The state for each contest should be controlled independently of the state of any other contest.
- Make clear which modules are active at any given time.
- Separate the UI from the core logic.
- Have a well-defined mapping from the (large) set of possible user inputs to the (small) set of signal inputs understood by the core logic.
- Have a well-defined mapping from the (small) set of output signals from the core logic to the (large) set of possible output screens to the user.

6.2 Design

Similar to $\mathcal{C}$, the actual voting machine $\mathcal{A}$ can be in one of three modes: *main*, *summary*, or *cast*. The entire state held by the voting machine consists of the current contest number, the selection state for each contest, and whether the machine is in main mode, cast vote mode, or summary screen mode. The machine is organized as one selection state module for each contest and a centralized controller module that controls which mode the machine is in and which contest (if any) is active. The selection state modules are each responsible for holding the state for a single contest. The controller maintains all other state of the voting machine. In addition to controller and selection state modules, there are three peripheral modules that

handle the input to and output from the voting machine: map, display, and cast. Each module is explained in depth in the following sections.

We use an LCD touch screen as the user interface to the voting machine. The (x, y) coordinates corresponding to a user's touch on the screen are the input to the voting machine. The output is the image displayed to the screen. In addition to the voter interface, the machine interfaces with non-volatile memory: it reads an election definition file (EDF) from read-only memory and writes the cast ballot to a separate memory bank at the end of each session.

There is an additional input, *reset*, which clears all register values to logic 0. It is intended that *reset* will be tied to a keyed mechanism that only a poll worker has access to. This allows the poll worker to prepare the voting machine for the next voter, after the previous voter has finished. Thus every voting session begins and ends with a reset. Resetting the state in this way guarantees that one voter's session can not affect any other session and that every voter will have the same experience [26].

In our implementation, a single ballot can have up to 7 contests, labeled 1–7, and each contest can have up to 10 candidates. To make the discussion more concrete, we will use these parameters, but an implementation could easily increase them if needed.

The EDF contains all the parameters for a particular election, for example, the list of contests and the candidates in each contest. The contents of the EDF are used by three modules, Map, SelectionState, and Display. The particulars of the EDF's content will be explained in the discussion of those three modules.

The full architecture of the voting machine implementation is shown in Figure 2.

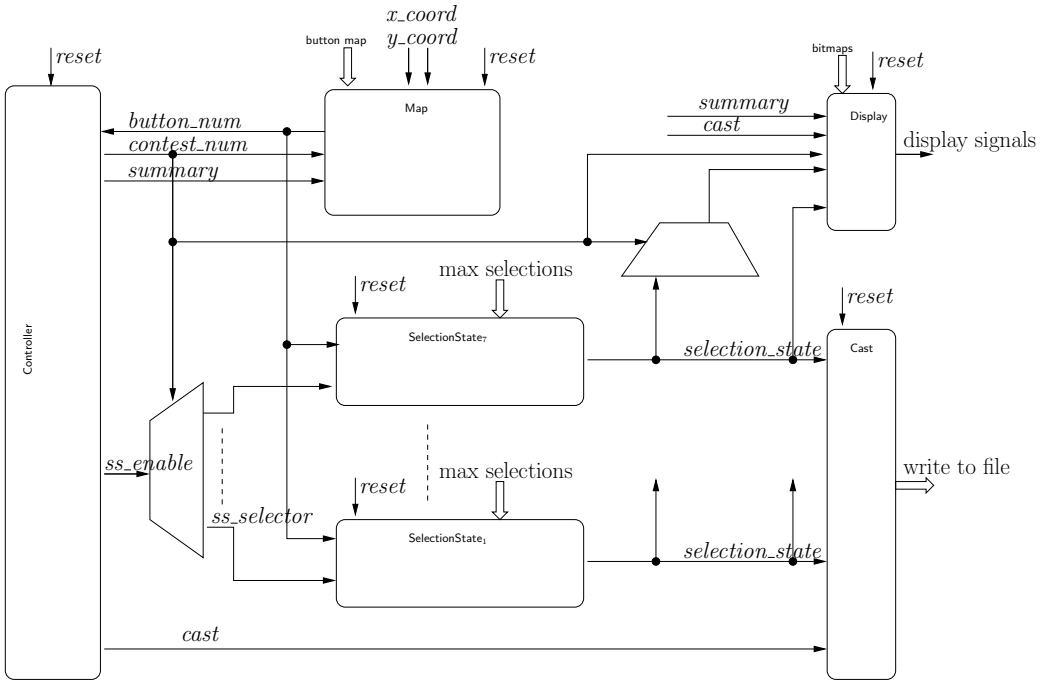


Figure 2: Design of the voting machine.

6.3 Component-Level Specifications

As part of the design process we fully specified each core component so that its behavior under all possible input combinations was well defined. Once implementation was complete, we were able to verify it against these specifications (see Section 8). The one exception was the Display module; its behavior was well specified, but we did not formally verify the implementation against the specification, in part because our prototype's display module is so simplified. In the following sections we provide the specification for each component.

6.3.1 Map

The Map module converts the (x, y) coordinate pair of the voter's touch on the screen to a signal, *button_num*, representing one of 16 logical buttons. For each candidate in a particular contest there will be a selectable region on the screen. The user touches somewhere in that region to select the candidate. That region is called a `select` button. In addition to the buttons for each candidate, every contest screen also has the navigation buttons `prev` and `next`, which let the voter move from contest to contest, and a `summary` button which allows the user to view the summary screen representing their entire ballot as it currently stands. From the summary screen the voter can select either the `resume` or the `cast` button. The former takes the voter back to the contest she was viewing prior to the summary screen. The latter casts the ballot irrevocably.

button_num represents the logical buttons as a number from 0 to 15, but it is easier to think of them using their names. In particular, a $button\_num = 0 \ldots 9$ corresponds to a $b_{\texttt{select}}$ button; $button\_num = 10$ is $b_{\texttt{resume}}$; $button\_num = 11$ is $b_{\texttt{summary}}$; $button\_num = 12$ is unused (it acts as a no-op); $button\_num = 13$ is $b_{\texttt{cast}}$; $button\_num = 14$ is $b_{\texttt{prev}}$; and $button\_num = 15$ is $b_{\texttt{next}}$.

In order to know the set of (x, y) coordinates covered by each button, Map reads a button map from the EDF that provides this information for each contest. The input signals *summary* and *contest_num* identify whether the machine is in summary mode or, if it is main mode, which contest is currently active so that Map can apply the correct mapping. In order for Map to work correctly, the button map has to be well-formatted; we formulated a precise mathematical expression defining a valid button map in our work, but intuitively it corresponds to saying each button is defined by two coordinates, the lower left and upper right corners of the rectangular region defining the buttons, and no two buttons may overlap.

By separating out the functionality required to convert an (x, y) signal to its associated logical button, we are able to more closely match the structure of $\mathcal{C}$ in the remainder of our design. This in turn makes the verification of our implementation simpler.

Specification

The signals to the Map module are defined as follows.

Input:	*reset*	{0, 1}
	x_coord	[0, 479]
	y_coord	[0, 799]
	contest_num	[1, 7]
	summary	{0, 1}
Output:	*button_num*	[0, 15]

On reset, the button map is read in from the EDF. The following describes its format.

button map: $(\mathrm{x}_0, \mathrm{y}_0, \mathrm{x}_1, \mathrm{y}_1)_{screen_0, button_0}, \ldots, (\mathrm{x}_0, \mathrm{y}_0, \mathrm{x}_1, \mathrm{y}_1)_{screen_n, button_m}$

$n =$ total number of contests + 1. The extra entry is used by the summary screen.
$m =$ ≤ 15 (variable by contest)

For two logical buttons a and b on screen i, defined in the button map by $a = (x_0, y_0, x_1, y_1)_{ia}$ and $b = (x_0, y_0, x_1, y_1)_{ib}$, the following always holds.

$$\neg(x_{0_{ia}} < x_{ib} < x_{1_{ia}} \bigwedge y_{0_{ia}} < y_{ib} < y_{1_{ia}}),$$

where $x_{ib} \in \{x_{0_{ib}}, x_{1_{ib}}\}$ and $y_{ib} \in \{y_{0_{ib}}, y_{1_{ib}}\}$

The value for the output signal is defined as follows.

$$button\_num = \begin{cases} \text{button}_a & \text{if } \exists\ (x_0, y_0, x_1, y_1)_{ia} \text{ in the button map s.t.} \\ & (x_0 < x\_coord < x_1) \text{ and} \\ & (y_0 < y\_coord < y_1) \text{ and} \\ & [(i = contest\_num \wedge \neg summary) \bigvee (i = n \wedge summary)] \\ 12 & \text{otherwise} \end{cases}$$

6.3.2 Controller

The Controller module controls which mode the machine is in and which contest is currently active.

Specification

The signals to the module are defined as follows.

Input:	*button_num*	[0, 15]
	reset	{0, 1}
Output:	*contest_num*	[1, 7]
	ss_enable	{0, 1}
	summary	{0, 1}
	cast	{0, 1}

The state maintained by the module and the corresponding output is as follows.

$$contest\_num = \begin{cases} contest\_num + 1 & \text{if } button\_num = b_{\texttt{next}} \wedge \neg cast \wedge \neg summary \wedge \neg reset \wedge contest\_num < 7 \\ contest\_num - 1 & \text{if } button\_num = b_{\texttt{prev}} \wedge \neg cast \wedge \neg summary \wedge \neg reset \wedge contest\_num > 1 \\ 1 & \text{if } reset \\ contest\_num & \text{otherwise} \end{cases}$$

$$ss\_enable = \begin{cases} 1 & \text{if } \neg reset \wedge \neg cast \wedge \neg summary \wedge button\_num \in \{b_{\texttt{select}}\} \\ 0 & \text{otherwise} \end{cases}$$

$$summary = \begin{cases} 1 & \text{if } \neg reset \wedge \neg cast \wedge button\_num = b_{\texttt{summary}} \\ 0 & reset \vee button\_num = b_{\texttt{resume}} \vee button\_num = b_{\texttt{cast}} \\ summary & \text{otherwise} \end{cases}$$

$$cast = \begin{cases} 1 & \text{if } \neg reset \wedge summary \wedge button\_num = b_{\texttt{cast}} \\ 0 & \text{if } reset \\ cast & \text{otherwise} \end{cases}$$

6.3.3 Selection State

There is one SelectionState module for each possible contest on the ballot: $\text{SelectionState}_1 \ldots \text{SelectionState}_7$. These correspond to the M_i state machines of $\mathcal{C}$. If an election contains fewer than 7 contests, the remaining SelectionState modules will simply go unused. The state of each module reflects the selections that have been made in that contest and is implemented as a 10-bit bitmap. The bit at index i is set if and only if the i^{th} candidate in that contest is currently selected.

The EDF includes a parameter indicating the maximum number of candidates a voter is allowed to select for that particular contest. If the voter tries to select more than the maximum allowed, *selection_state* will not change until one of the current choices is deselected.

Specification

The signals to the module are defined as follows.

Input:	*button_num*	[0, 15]
	ss_selector	{0, 1}
	max_selections	[1, 10]
	reset	{0, 1}
Output:	*selection_state*	$\{0, 1\}^{10}$

The selection state maintained by the module is a 10-bit bitmap, defined as follows.

$$\textit{selection\_state} \quad (x_0, \ldots, x_9)$$

$$x_b = \begin{cases} 0 & \text{if } (\textit{ss\_selector} \wedge \textit{button\_num} = b \wedge x_b = 1) \vee \textit{reset} \\ 1 & \text{if } \textit{ss\_selector} \wedge \textit{button\_num} = b \wedge x_b = 0 \wedge (x_0 + \cdots + x_9 < \textit{max\_selections}) \\ x_b & \text{otherwise} \end{cases}$$

6.3.4 Cast

The Cast module is responsible for writing the final values of the selection state for each contest to non-volatile memory. It does not maintain any state as the voter proceeds through the voting session, but once *cast* is set, the module freezes a snapshot of all the *selection_state* and writes these values to non-volatile memory. The Cast module corresponds to T_{cast} in $\mathcal{C}$; the transition to Cast is triggered when the voter presses the `cast` button on their screen.

The signals to the module are defined as follows

Input:	*cast*	{0, 1}	
	$\textit{selection\_state}_i$	$\{0, 1\}^{10}$	for $1 \leq i \leq 7$
Output:	*memory*	$\{0, 1\}^{10}_i$	for $1 \leq i \leq 7$

The state maintained by the module is a register *memory*. In our prototype we use *memory* to model the cast vote record that is stored in non-volatile memory. When *cast* is initially triggered, we write $\textit{selection\_state}_i$ into $\textit{memory}_i$, for each i; thereafter, *memory* remains unchanged.

$$\textit{memory}: \quad (x_0, \ldots, x_9)_1, \ldots, (x_0, \ldots, x_9)_7$$

$$(x_0, \ldots, x_9)_i = \begin{cases} (0, \ldots, 0) & \text{if } \textit{reset} \\ \textit{selection\_state}_i & \text{if } \textit{cast} \\ (x_0, \ldots, x_9)_i & \text{otherwise} \end{cases}$$

6.3.5 Display

Pvote showed that the use of pre-rendered screen images could greatly reduce the complexity of a voting machine [34]. We use this idea and include in the EDF a series of bitmap images for each contest. The base bitmap for a contest shows the buttons for each candidate as well as the navigation buttons. There is an additional overlay bitmap for each candidate in the contest. Each of these candidate overlays contains only highlighting in the region corresponding to that candidate's button. The screen output is produced by

displaying one overlay for each candidate that has been selected in that contest, on top of the base bitmap image.

For all screens, the output is partitioned into four sections. The upper left corner of the screen displays the current mode for the machine: main, summary, or cast. The upper right corner displays the current contest number when in main mode and is left blank when in summary or cast mode. The lower third of the screen displays the navigation buttons. Figure 3 illustrates the partitioning.

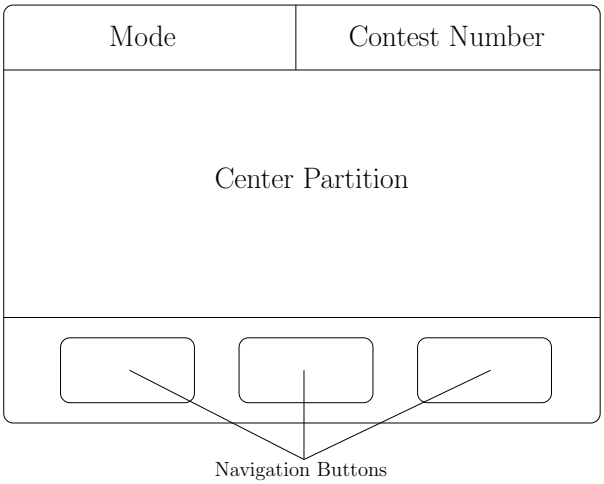


Figure 3: The partitions of the output display screen.

When the machine is in main mode, the center partition displays the candidates for the current contest and the available navigation buttons are `prev`, `next`, and `summary`. When in summary mode, the center partition is further divided into seven sections, one for each contest. The current state of each contest is displayed in its partition. The available navigation buttons in summary mode are `resume` and `cast`. In cast mode, the center partition displays a solid block of color and there are no navigation buttons available.

The Display module acts as the interface between the electronic voting machine and the LCD controller. A multiplexer provides the selection state of the active contest to the Display module, which then generates the correct output signals to display on the screen.

The voting machine is designed to work only with contest numbers, not contest names. However, voters will want to know the names of contests, not their numbers. A mapping of contest number to contest name (e.g., 1: Presidential) is kept by the EDF. It is up to the bitmaps of each contest to display the name for the voter to see. Any inconsistencies between the bitmaps and the mapping stored in the EDF will be caught during testing.

6.4 System-Level Behavioral Properties

We identified a number of properties which are critical for the correct behavior of the voting machine as a whole. For each property below, we provide an English description followed by a formal specification in linear temporal logic (LTL) [21]. In Section 8 we discuss how we verified these properties against our implementation. Formulating and verifying these properties was a useful exercise and we uncovered a number of bugs in the initial implementation of our design during the process.

1. At any given time, no more than one contest can be active.
   $G(reset \rightarrow XG(ss\_selector[0] + \cdots + ss\_selector[contest\_num - 1] \leq 1))$

2. A contest i is active if and only if the current contest number is i.
   $G(reset \rightarrow XG((contest\_num = i \wedge ss\_enable) \iff ss\_selector[i]))$

3. The total number of candidates selected for any contest is not more than the maximum allowed as given by the election definition file.
   $G(reset \rightarrow XG(total\_selections \leq max\_selections))$
   where $total\_selections = selection\_state[0] + \cdots + selection\_state[number\ of\ candidates - 1]$.

4. The selection state of a contest can not change if neither *ss_selector* nor *reset* are set. Note that in the case where $selection\_state_i$ starts low, it suffices to check that it remains low if *ss_selector* is not set regardless of the value of *reset*.
   $\forall i\ G(reset \rightarrow XG((\neg reset \bigwedge \neg ss\_selector \bigwedge selection\_state_i) \rightarrow X(selection\_state_i)))$
   $\forall i\ G(reset \rightarrow XG((\neg ss\_selector \bigwedge \neg selection\_state_i i) \rightarrow X(\neg selection\_state_i)))$

5. The selection state of a contest can not change if the pressed button is not within the set of valid selection buttons. Thus, the `next`, `prev`, `summary`, `resume`, and `cast` buttons cannot affect the selection state of any contest.
   $\forall i\ G(reset \rightarrow XG(\neg reset \bigwedge button\_num \notin \mathcal{I}_S \bigwedge selection\_state_i \rightarrow X(selection\_state_i)))$
   $\forall i\ G(reset \rightarrow XG((button\_num \notin \mathcal{I}_S \bigwedge \neg selection\_state_i) \rightarrow X(\neg selection\_state_i)))$

6. Setting *reset* clears the selection state for all contests.
   $\forall i\ G(reset \rightarrow X(\neg selection\_state_i))$

7. * Setting *reset* causes the *current_contest*, *summary*, and *cast* signals to be cleared and selections to be disabled.
   $G(reset \rightarrow X(\neg cast \bigwedge \neg ss\_enable \bigwedge \neg contest\_num \bigwedge \neg summary))$

8. * The voting machine enters cast mode only after a press of the `cast` button.
   $G(reset \rightarrow (XG((\neg cast \bigwedge button \neq b_{\texttt{cast}}) \rightarrow X(\neg cast))))$
   $G(reset \rightarrow (XG((\neg reset \bigwedge button = b_{\texttt{cast}}) \rightarrow X(cast))))$

9. Once the voting machine enters cast mode, *cast* is not cleared until the next cycle of the voting machine beginning with *reset*.
   $G(reset \rightarrow (XG(cast \rightarrow (cast\ U\ reset))))$

10. Once the voting machine enters cast mode, the selection states of all the contests become frozen and do not change until the next cycle beginning with *reset*.
    $G(reset \rightarrow (XG(cast \rightarrow (\neg ss\_enable\ U\ reset))))$

11. Selection of a candidate and casting of votes can not take place at the same time.
    $G(reset \rightarrow XG(\neg(cast \bigwedge ss\_enable)))$

12. * The voting machine enters summary mode only by by pressing the `summary` button.
    $G(reset \rightarrow (XG((\neg summary \bigwedge button \neq b_{\texttt{summary}}) \rightarrow X(\neg summary))))$
    $G(reset \rightarrow (XG((\neg reset \bigwedge \neg cast \bigwedge button = b_{\texttt{summary}}) \rightarrow X(summary))))$

13. * Once the voting machine enters summary mode, *summary* is not cleared if neither the `resume` nor `cast` button is pressed, nor the machine is reset.
    $G(reset \rightarrow XG((summary \bigwedge \neg reset \bigwedge button \neq b_{\texttt{resume}} \bigwedge button \neq b_{\texttt{cast}}) \rightarrow X(summary))))$

14. * Pressing the `resume` button will clear *summary*.
    $G(reset \rightarrow XG(b_{\texttt{resume}} \rightarrow X(\neg summary))$

15. * Selection of a candidate and viewing of the summary screen can not take place at the same time.
    $G(reset \rightarrow XG(\neg(summary \bigwedge ss\_enable)))$

16. * Summary mode and casting of votes can not take place at the same time.
    $G(reset \rightarrow XG(\neg(cast \bigwedge summary)))$

17. * The current contest number does not change when in summary screen mode.
    $G(reset \rightarrow XG((summary \bigwedge \neg reset \bigwedge contest\_num = i) \rightarrow X(contest\_num = i)))$

18. * The current contest number does not change if the pressed button is not one of the navigation buttons, `prev` or `next`.
$G(reset \rightarrow XG((\neg reset \bigwedge button \neq b_{\texttt{next}} \bigwedge button \neq b_{\texttt{prev}} \bigwedge contest\_num = i) \rightarrow X(contest\_num = i)))$

19. * The current mode does not change if the pressed button is not one of the navigation buttons.
$G(reset \rightarrow XG((\neg reset \bigwedge button \notin \{b_{\texttt{next}}, b_{\texttt{prev}}, b_{\texttt{summary}}, b_{\texttt{resume}}, b_{\texttt{cast}}\} \bigwedge summary) \rightarrow X(summary)))$
$G(reset \rightarrow XG((\neg reset \bigwedge button \notin \{b_{\texttt{next}}, b_{\texttt{prev}}, b_{\texttt{summary}}, b_{\texttt{resume}}, b_{\texttt{cast}}\} \bigwedge \neg summary) \rightarrow X(\neg summary)))$
$G(reset \rightarrow XG((\neg reset \bigwedge button \notin \{b_{\texttt{next}}, b_{\texttt{prev}}, b_{\texttt{summary}}, b_{\texttt{resume}}, b_{\texttt{cast}}\} \bigwedge cast) \rightarrow X(cast)))$
$G(reset \rightarrow XG((\neg reset \bigwedge button \notin \{b_{\texttt{next}}, b_{\texttt{prev}}, b_{\texttt{summary}}, b_{\texttt{resume}}, b_{\texttt{cast}}\} \bigwedge \neg cast) \rightarrow X(\neg cast)))$

The properties with a single asterisk (*) next to them are the properties required for proving the summary screen correct. These are either a modification of a property we used in our original voting machine [28] or they are entirely new to this work.

6.5 System-Level Structural Properties

Our voting machine is structured so that it satisfies our required properties, P_0 – P_5. This will allow us to establish the equivalence of our implementation to the canonical model $\mathcal{C}$ through testing. We list here the specific structural properties that combine to satisfy P_0 – P_5 as well as additional properties required by our proof of correctness. Section 8.3 discusses our verification of these properties.

1. The voting machine should be a deterministic finite state machine.
2. Contests should be independent of each other, i.e., the selection state of one contest should not have any influence on the evolution of the selection state of any other contest.
3. A contest's selection state after a single transition should depend only on that contest's previous selection state, the active contest number, and whether any selection button was pressed and if so which one.
4. If a navigation button is pressed, the next active contest number should depend only on the previous active contest number and which button was pressed. Otherwise, the active contest number should not change.
5. The final memory storing the selection state should be completely determined by the selection states of the contests before cast.
6. * The display module, which makes buttons visible, should partition the screen into four regions as shown in Figure 3. The display in the mode partition should be a deterministic function of the *cast* and *summary* signals; moreover, this function should be injective. The display in the current contest number partition should depend only on the *current_contest* signal and on the *cast* and *summary* signals. The display in the navigation partition should depend only on the *cast* and *summary* signals.
7. For any fixed EDF, when in main mode (i.e., when $\neg cast \wedge \neg summary$) the center partition of the output screen should be a deterministic function of the active contest number and the selection state of the current contest; moreover, this function should be injective.
8. * When in summary mode, the center partition of the output screen should be further partitioned into regions, one for each contest on the ballot. For any fixed EDF, the display in sub-partition i should be a deterministic function of the selection state of contest i; moreover, this function should be injective.
9. * The output of the Map module, which makes regions of the screen pressable, should not depend on the selection state of any contest. *button_num* may depend only on the current mode (*cast* and *summary* signals), the current contest number (*current_contest* signal), and the (x, y) coordinates pressed by the user.

The properties with an asterisk (*) were not required of our original voting machine and are new to this work; they are required for proving correctness of the summary screen.

7 Implementation

We implemented the above design in Verilog, a hardware description language for digital circuits. We synthesized our implementation onto the Altera FPGA, Nios II Embedded Evaluation Kit, Cyclone III

Edition with a touchscreen daughterboard. The core implementation is 1020 lines of code. The modules for interacting with the peripherals (the touchscreen and VGA video) add an additional 850 lines of code.

Our implementation differs from the design in one respect: our current prototype does not include an interface to non-volatile storage. While we would expect the EDF and cast vote records to be stored on flash memory in a finished implementation, our prototype uses volatile memory to simulate this functionality. This represents a limitation of our current engineering and is not a fundamental shortcoming of our approach. However, this limitation has several implications:

1. In our prototype, the EDF is hard-coded into the memory of the voting machine. Map has a register array containing a button map for a particular election and SelectionState has a register array storing the maximum number of candidates a voter can choose in each contest. In a finished implementation, this data might be read in from removable flash memory.
2. In our prototype, Cast writes the cast vote record to a register array called *memory* instead of to external storage. When we verify properties about the cast ballot, we verify them on *memory*. A finished implementation might write the cast vote record to external storage, such as a removable SD flash card. In that case we would also need to verify the interface to the SD card.
3. In our prototype, Display outputs an extremely simplified screen image indicating the candidates chosen for the current contest. The current screen images would not be usable by anyone other than the system developers. This limitation exists because our FPGA has a limited amount of on-chip memory available for storage of the images. In a finished implementation the EDF would be read from external storage, making it possible to store and use high-resolution images.

8 Formal Verification

Using formal verification techniques we show that our implementation follows our design specifications and satisfies the desired behavioral and structural properties.

8.1 Component-Level Specifications

In Section 6.3 we fully specified the behavior of each component of the machine under all possible inputs. For all but the Map module, we used Cadence SMV [17], a symbolic model checker, to verify the implementation conforms to these specifications. We used the SMV notion of a *layer*, a formal specification written in the SMV language, to express our component-level specifications. The model checker verifies that the implementation refines the layer, that is, that all possible behaviors of the implementation are consistent with the component-level specification.

We were unable to verify the component-level specifications for Map using Cadence SMV; the large register holding the EDF's button map made the state space too large to model check at the bit level. In our previous work we constructed an SMT instance (in the combination of the theories of uninterpreted functions and bit-vectors) encoding the assertion that the module's behavior matches its specification. The memory in Map was modeled as an uninterpreted function, and the Yices SMT solver [35] was used to complete the verification. Under the assumption that the EDF is valid (i.e., the button map is well-formed) we were able to verify that the Map module meets its component-level specification. The verification of the Map module that includes the *summary* input is work that remains to be done.

8.2 System-Level Behavioral Properties

We formulated each behavioral property from Section 6.4 as an LTL formula and used the SMV model checker to verify our implementation satisfies the property. The tool will perform any necessary Verilog-to-SMV translation, allowing us to run the verification directly on our Verilog implementation. Deriving the correct LTL formula for a given property was not always straightforward and we did not always get it right on the first try. However, the Verilog code, the SMV layers, and the LTL properties represent three independent means of describing our voting machine. Once mutually consistent, each one provides a cross-check on the other two and gives us increased confidence that they are each correct. Every property given in Section 6.4 was verified correct on the new voting machine implementation that includes the summary screen.

8.3 System-Level Structural Properties

We showed in our previous work how we verify the structural properties by formulating them as Boolean satisfiability (SAT) problems [28]. Each property is of the form: signals $y_1, \ldots, y_n$ depend only on signals $x_1, \ldots, x_n$. For the new implementation that includes the summary screen we performed a manual verification that the dependency constraints are maintained. For example, a visual inspection of the code shows that none of the selection states are an input to the Map module, nor are they an input to any of the modules that provide input, either directly or indirectly, to the Map module. A manual inspection does not replace a formal verification, but with a small design with only a few, well-defined modules, the manual inspection provides a measure of confidence that these properties are satisfied by the design. The formal verification is work that remains to be done.

The dependency graph shown in Figure 4 provides a subset of the results of our manual verification. For a given signal x at time t, x' denotes that same signal at time $t+1$. There is an edge from signal x to signal y if y depends on x. As the figure shows, a given selection state never depends on the value of any other selection state.

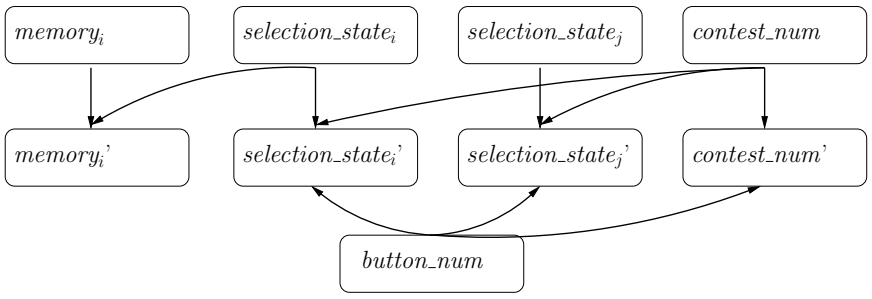


Figure 4: Dependency graph for a subset of the state in the voting machine. The first row shows the state variables at time t, the second row shows the state variables at time $t+1$, and the third row shows the input variables. An arrow from state x to state y indicates y's dependence on x.

9 Testing and Proving Correctness

Through user testing we prove our implementation, $\mathcal{A}$, is correct, i.e., is trace equivalent to $\mathcal{C}$. In our previous work we proved the correctness of an implementation that did not include a summary screen; in this section we prove that our new implementation with a summary screen added is correct.

Our proof of correctness relies on user testing. Testing proceeds by giving the tester a test case T, which is a sequence of inputs (navigation buttons and candidate selections) to apply: $T = (b_1, \ldots, b_l)$. The test begins with $\mathcal{A}$ in the initial state achieved by setting the *reset* signal. Starting with the first output screen z_0 and the first input in T, b_1, the tester determines a corresponding (x, y) coordinate pair such that her interpretation function would yield $I_{\mathcal{I}}(z_0, (x, y)) = b_1$. The tester presses the screen at position (x, y), potentially transitioning $\mathcal{A}$ to a new state and corresponding output screen. If the tester determines there is no (x, y) corresponding to b_1, she considers b_1 to be a no-op for $\mathcal{C}$, does not press the screen at any position, and moves on to the next input in the test case. Let $\tau_{\mathcal{A}}$ be the trace of the voting machine during test T and let $\tau_{\mathcal{C}}$ be the trace of the mental model the tester is implicitly using during the test. The test passes if $I(\tau_{\mathcal{A}}) = \tau_{\mathcal{C}}$. In other words, the test passes if at each point during the test, the state of the tester's mental model is equivalent to the tester's interpretation of the machine's output screen. Below, we describe two suites of test cases and show that if they both pass, then $\mathcal{A}$ is trace equivalent to $\mathcal{C}$, i.e., our implementation meets our definition of correctness (given in Section 5).

9.1 Election Definition File (EDF)

The EDF plays a crucial role in our proof of correctness. We have already seen that in many cases formal verification of a component or a property relies on a well-formatted EDF. In the following sections we will rely

on the EDF during user testing to provide the ground truth. For example, if the EDF lists five candidates for a particular contest, we accept that as correct and verify the voting machine provides the five candidates for that contest. If, in fact, the EDF is missing a candidate, our testing will not catch the error.

We argue these are reasonable requirements of our system. The well-formedness of the EDF can be checked by an automated tool. The specification for the EDF is well defined, so it is possible to imagine a tool that reads in the EDF and validates its contents against the specification. A single tool could be used to check all EDFs. The second requirement, that the contents of the EDF are correct, is more difficult to verify, but still reasonable. A single EDF may be loaded in to multiple machines so placing reliance on the EDF is an advantage over placing reliance on every voting machine. Furthermore, the contents of the EDF could be made public before election day so that its correctness could be checked by any interested third party as well as by election officials. In particular, we require the mapping of contest number to contest name to be correct and we require the button map used by the controller to be correct. We do not impose any requirements on the bitmaps used for display; any inconsistencies in those will be caught during testing.

9.2 Assumptions

Our proof of correctness makes the following assumptions:

A_0: For a given election definition file (EDF), for every state (m, i, s_i) and every (x, y) location on the touch screen, $I_{\mathcal{I}}(z, (x, y)) = \mathsf{Map}(m, i, (x, y))$, where $z = \rho(m, i, s_i)$.

A_1: We assume there exists a single interpretation function $I = (I_{\mathcal{I}}, I_{\mathcal{O}})$ such that, for every human tester, the human tester passes the voting machine on test T if and only if it is correct on test T.

Assumption A_0 states that for every screen produced by the voting machine, every (x, y) location for that screen will be interpreted by the voter in accordance with how it is mapped to internal buttons by the Map module. From property P_4 and Structural Properties 6–8, ρ is a deterministic function.

9.3 Test Suites

Our theorem requires two test suites, Navigation Coverage and Selection Coverage. Each suite comprises a series of test cases. The inputs given in each test case are defined with respect to $\mathcal{C}$.

The first test suite, Navigation Coverage, is used to test the contest number portion of $\mathcal{A}$'s state. The test cases are described in Table 1. For readability, the test cases are grouped into five sets: NC_1, NC_2, ... NC_5.

The second test suite, Selection Coverage, is used to test the selection state portion of $\mathcal{A}$'s state. The test cases are described in Table 2 and are grouped into sets for readability. Set SC^i_j is testing the selection state for contest i. There are $2k + 2$ sets of tests for each contest (SC^i_1, SC^i_2, ..., SC^i_{2k}, SC^i_{2k+1}, SC^i_{2k+2}), where k is the number of candidates in each contest and voters are allowed to choose a single candidate. In Section 9.6, we describe how to extend the test cases to handle elections where n out of k candidates can be chosen.

For both the Navigation and Selection suites, in each set, each line refers to a new test, which starts with both $\mathcal{C}$ and $\mathcal{A}$ in the initial state. In the tables, the input $\texttt{select}_i$ refers to selecting the i^{th} candidate in the current contest and the value N represents the total number of contests in $\mathcal{C}$.

9.4 Use of Required Properties

Our proof of correctness relies on both $\mathcal{A}$ and $\mathcal{C}$ satisfying properties P_0–P_4 as described in Section 3. In addition, $\mathcal{A}$ must satisfy property P_5 (P_5 does not apply to $\mathcal{C}$ as $\mathcal{C}$ has no notion of a cast vote record). The canonical voting machine satisfies properties P_0–P_4 by construction. Verification of $\mathcal{A}$'s structural properties (given in Section 6.5 and verified in Section 8.3) prove $\mathcal{A}$ satisfies properties P_0–P_4. Property P_5 follows from the verification of the structural and behavioral properties (Section 6.4) under the assumption that the machine's record of the cast votes is correctly output to persistent storage (e.g., paper) for the human to check.

NC_1	NC_2
$\texttt{select}_1, \texttt{summary}, \texttt{cast}$	$\texttt{prev}, \texttt{select}_1, \texttt{summary}, \texttt{cast}$
$\texttt{next}, \texttt{select}_1, \texttt{summary}, \texttt{cast}$	$\texttt{next}, \texttt{prev}, \texttt{select}_1, \texttt{summary}, \texttt{cast}$
$\texttt{next}, \texttt{next}, \texttt{select}_1, \texttt{summary}, \texttt{cast}$	$\texttt{next}, \texttt{next}, \texttt{prev}, \texttt{select}_1, \texttt{summary}, \texttt{cast}$
$\vdots$	$\vdots$
$\underbrace{\texttt{next}, \ldots, \texttt{next}}_{N}, \texttt{select}_1, \texttt{summary}, \texttt{cast}$	$\underbrace{\texttt{next}, \ldots, \texttt{next}}_{N}, \texttt{prev}, \texttt{select}_1, \texttt{summary}, \texttt{cast}$

NC_3
$\texttt{summary}, \texttt{resume}, \texttt{select}_1, \texttt{summary}, \texttt{cast}$
$\texttt{next}, \texttt{summary}, \texttt{resume}, \texttt{select}_1, \texttt{summary}, \texttt{cast}$
$\texttt{next}, \texttt{next}, \texttt{summary}, \texttt{resume}, \texttt{select}_1, \texttt{summary}, \texttt{cast}$
$\vdots$
$\underbrace{\texttt{next}, \ldots, \texttt{next}}_{N}, \texttt{summary}, \texttt{resume}, \texttt{select}_1, \texttt{summary}, \texttt{cast}$

NC_4
$\texttt{summary}, \texttt{next}, \texttt{resume}, \texttt{select}_1, \texttt{summary}, \texttt{cast}$
$\texttt{next}, \texttt{summary}, \texttt{next}, \texttt{resume}, \texttt{select}_1, \texttt{summary}, \texttt{cast}$
$\texttt{next}, \texttt{next}, \texttt{summary}, \texttt{next}, \texttt{resume}, \texttt{select}_1, \texttt{summary}, \texttt{cast}$
$\vdots$
$\underbrace{\texttt{next}, \ldots, \texttt{next}}_{N}, \texttt{summary}, \texttt{next}, \texttt{resume}, \texttt{select}_1, \texttt{summary}, \texttt{cast}$

NC_5
$\texttt{summary}, \texttt{prev}, \texttt{resume}, \texttt{select}_1, \texttt{summary}, \texttt{cast}$
$\texttt{next}, \texttt{summary}, \texttt{prev}, \texttt{resume}, \texttt{select}_1, \texttt{summary}, \texttt{cast}$
$\texttt{next}, \texttt{next}, \texttt{summary}, \texttt{prev}, \texttt{resume}, \texttt{select}_1, \texttt{summary}, \texttt{cast}$
$\vdots$
$\underbrace{\texttt{next}, \ldots, \texttt{next}}_{N}, \texttt{summary}, \texttt{prev}, \texttt{resume}, \texttt{select}_1, \texttt{summary}, \texttt{cast}$

Table 1: Navigation Coverage test suite

9.5 Theorem of Correctness

Theorem 1. *Assume A_0 and A_1 hold. Then test suites Navigation Coverage and Selection Coverage will pass if and only if $\mathcal{A}$ is correct.*

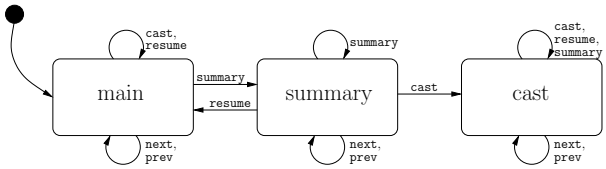
From Section 5, $\mathcal{A}$ is correct if, for every possible complete trace $\tau_{\mathcal{A}}$ of $\mathcal{A}$, $I(\tau_{\mathcal{A}})$ is a valid complete trace of $\mathcal{C}$.

The "if" portion of the theorem follows trivially. We prove the "only if" portion through the use of three lemmas. In the first and second, we separately consider the mode and contest number portion of the state and show that if the Navigation Coverage suite of tests pass, then for any complete trace of $\mathcal{A}$, the mode and current contest number of $\mathcal{A}$ is correct at each step of the trace. In the third lemma, we consider the selection state and show that if the Selection Coverage and Navigation Coverage suite of tests pass, then the selection state of $\mathcal{A}$ is also correct.

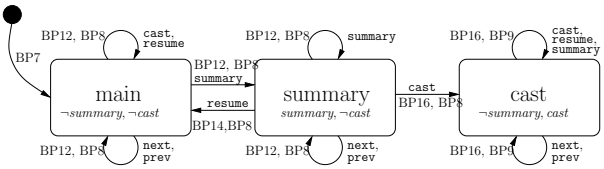
We say $\mathcal{A}$ or $\mathcal{C}$ is in state (m, i, s_i) to mean the mode part of its state is m, its current contest is i, and the selection state for the ith contest is s_i. As a shortcut, we abuse notation slightly and use b to refer to an input to both $\mathcal{A}$ and $\mathcal{C}$. When applied to $\mathcal{A}$, b is the logical-button signal corresponding to a particular (x, y) coordinate on a given output screen, i.e., the output of the Map module.

SC_1^1	SC_2^1
summary, cast	summary, resume, summary, cast
$\texttt{select}_1$, summary, cast	summary, resume, $\texttt{select}_1$, summary, cast
$\vdots$	$\vdots$
$\texttt{select}_k$, summary, cast	summary, resume, $\texttt{select}_k$, summary, cast
SC_3^1	SC_4^1
$\texttt{select}_1$, $\texttt{select}_1$, summary, cast	summary, resume, $\texttt{select}_1$, $\texttt{select}_1$, summary, cast
$\texttt{select}_1$, $\texttt{select}_2$, summary, cast	summary, resume, $\texttt{select}_1$, $\texttt{select}_2$, summary, cast
$\vdots$	$\vdots$
$\texttt{select}_1$, $\texttt{select}_k$, summary, cast	summary, resume, $\texttt{select}_1$, $\texttt{select}_k$, summary, cast
	$\vdots$
SC_{2k+1}^1	SC_{2k+2}^1
$\texttt{select}_k$, $\texttt{select}_1$, summary, cast	summary, resume, $\texttt{select}_k$, $\texttt{select}_1$, summary, cast
$\texttt{select}_k$, $\texttt{select}_2$, summary, cast	summary, resume, $\texttt{select}_k$, $\texttt{select}_2$, summary, cast
$\vdots$	$\vdots$
$\texttt{select}_k$, $\texttt{select}_k$, summary, cast	summary, resume, $\texttt{select}_k$, $\texttt{select}_k$, summary, cast
SC_1^2	SC_2^2
next, summary, cast	next, summary, resume, summary, cast
next, $\texttt{select}_1$, summary, cast	next, summary, resume, $\texttt{select}_1$, summary, cast
$\vdots$	$\vdots$
next, $\texttt{select}_k$, summary, cast	next, summary, resume, $\texttt{select}_k$, summary, cast
	$\vdots$
SC_1^N	SC_2^N
$\underbrace{\texttt{next}, \ldots, \texttt{next}}_{N-1}$, summary, cast	$\underbrace{\texttt{next}, \ldots, \texttt{next}}_{N-1}$, summary, resume, summary, cast
next, ..., next, $\texttt{select}_1$, summary, cast	next, ..., next, summary, resume, $\texttt{select}_1$, summary, cast
$\vdots$	$\vdots$
next, ..., next, $\texttt{select}_k$, summary, cast	next, ..., next, summary, resume, $\texttt{select}_k$, summary, cast

Table 2: Selection Coverage test suite



(a) The transitions of the canonical voting machine, $\mathcal{C}$, and their effect on its mode.



(b) The transitions of the actual voting machine, $\mathcal{A}$. In each state the valuation of signals *summary* and *cast* are shown. Each transition is labeled with the behavioral properties that guarantee the transition is the one taken for the given input.

Figure 5: The 'mode' portion of the state machines. Only transitions resulting from inputs in $\mathcal{I}_N$ are shown. By construction of $\mathcal{C}$ and by Behavioral Property 19 of $\mathcal{A}$, any inputs not in $\mathcal{I}_N$ do not update the mode; these self-loops are omitted for clarity.

9.6 Proof of Theorem 1

Lemma 1 (Mode)**.** *Suppose $\mathcal{A}$ passes the Navigation Coverage suite of tests. For any complete trace $\tau_{\mathcal{A}}$, for any step j, where $\mathcal{A}$ is in state (m, i, s_i) and displaying output screen $z = \rho(m, i, s_i)$, we have $I_{\mathcal{O}}(z) = (m, \_, \_)$ and moreover, if we run $\mathcal{C}$ on the sequence of inputs $I_{\mathcal{I}}(\tau_{\mathcal{A}})$, then at step j, $\mathcal{C}$ is in state $(m, \_, \_)$.*

Proof. Figure 5a shows $\mathcal{C}$'s transitions between main, summary, and cast mode. At each mode m, for every navigation input $b \in \mathcal{I}_N$, the transition (m, b, m') is shown. Figure 5b shows the transitions taken by $\mathcal{A}$ after abstracting away everything but the mode. At each mode m, for every navigation input b, the transition (m, b, m') is shown and is labeled with the behavioral properties that ensure the transition shown is correct and is the only transition allowed from mode m with input b. From the two figures it is clear that for each mode m, for every navigation input $b \in \mathcal{I}_N$, both $\mathcal{C}$ and $\mathcal{A}$ make the same transition: (m, b, m').

By construction, $\mathcal{C}$ starts in main mode. By Behavioral Property 7, $\mathcal{A}$ starts in main mode as well. $\mathcal{A}$ and $\mathcal{C}$ start in the same mode, and we know that if $\mathcal{A}$ and $\mathcal{C}$ start in the same mode and are given the same navigation input they will both transition to the same mode. Furthermore, by Behavioral Property 19, for any $b \notin \mathcal{I}_N$ the current mode does not change. By induction, it follows that for every trace starting from the initial state, if $\mathcal{A}$ and $\mathcal{C}$ are given the same inputs at each step, they will have the same mode at each step of the trace. By assumption A_0, the tester can correctly interpret the meaning of every button in $\mathcal{A}$ and so apply the same input to $\mathcal{A}$ and $\mathcal{C}$. Therefore at step j of trace $\tau_{\mathcal{A}}$, $\mathcal{A}$ and $\mathcal{C}$ both have state $(m, \_, \_)$. It remains to show that at step j, $I_{\mathcal{O}}(z) = (m, \_, \_)$.

There was a step in one of the navigation tests when $\mathcal{C}$ had state $(m, 1, \{1\})$. Let that be step k. Because the test passed, we know the tester's interpretation of $\mathcal{A}$'s output at step k, z_k, matched the state of $\mathcal{C}$: $I_{\mathcal{O}}(z_k) = (m, 1, \{1\})$. We also know, from the preceding paragraph, the mode portion of $\mathcal{A}$'s state at step k matches that of $\mathcal{C}$: let $(m, \hat{i}, s_{\hat{i}})$ be $\mathcal{A}$'s state at step k. Therefore $I_{\mathcal{O}}(\rho(m, \hat{i}, s_{\hat{i}})) = (m, 1, \{1\})$. Because the mode portion of the output screen depends only on the current mode and not on the contest number nor on any selections made (Structural Property 6), we assume the interpretation of the mode portion of state depends only on the mode and not on the contest number nor on any selections made. Therefore $I_{\mathcal{O}}(\rho(m, \_, \_)) = (m, \_, \_)$ for any contest number and selection state. Therefore at step j of $\tau_{\mathcal{A}}$, $I_{\mathcal{O}}(\rho(m, i, s_i)) = (m, \_, \_)$. We have shown that $\mathcal{A}$ is correct for the mode portion of state. □

Lemma 2 (Contest Number)**.** *Suppose $\mathcal{A}$ passes the Navigation Coverage suite of tests. For any complete trace $\tau_{\mathcal{A}}$ of $\mathcal{A}$, for any step j, where $\mathcal{A}$ is in state (m, i, s_i) and displaying output screen $z = \rho(m, i, s_i)$, we have $I_{\mathcal{O}}(z) = (m, i, \_)$ and moreover, if we run $\mathcal{C}$ on the sequence of inputs $I_{\mathcal{I}}(\tau_{\mathcal{A}})$, then at step j, $\mathcal{C}$ is in state $(m, i, \_)$.*

We prove Lemma 2 in two steps. In the first step we prove that at step j, the current contest number of $\mathcal{A}$ will equal the current contest number of $\mathcal{C}$. In the second step we complete the proof of Lemma 2 and show that $I_{\mathcal{O}}(z) = (m, i, \_)$. In both steps we rely on Lemma 1 to guarantee that the mode portion of state is correct.

Proof. Step 1
The first test case in NC_1 shows that $\mathcal{A}$ and $\mathcal{C}$ both start in an initial state with contest number equal to 1. Consider first the state of $\mathcal{C}$. By construction, $\mathcal{C}$ starts in contest 1. By property P_3 the $\texttt{select}_1$ button can make a selection in the current contest (contest 1) and no other contest. By property P_2 when the `summary` and `cast` buttons are pressed the selection state does not change. The final output of $\mathcal{C}$ (the cast vote record) after the first test case will be $\text{CVR}_{\mathcal{C}} = (\{1\}, \emptyset, \emptyset, \ldots, \emptyset)$. The first candidate is chosen in contest 1 and no other contests have any candidate chosen. Because the test passed, we know the cast vote records of $\mathcal{C}$ and $\mathcal{A}$ must match: $\text{CVR}_{\mathcal{C}} = \text{CVR}_{\mathcal{A}}$. By assumption A_0, $\mathcal{A}$ received the same inputs during the test as $\mathcal{C}$ did. By properties P_2, P_3, and P_5, only the current contest in $\mathcal{A}$'s initial state will have a non-empty selection state in $\text{CVR}_{\mathcal{A}}$. Since contest 1 of $\text{CVR}_{\mathcal{A}}$ is non-empty and all others are empty, $\mathcal{A}$ was in contest 1 when the $\texttt{select}_1$ button was pressed. We now know that $\mathcal{A}$ and $\mathcal{C}$ both start in contest 1.

By similar reasoning, because the second test case in NC_1 passed, we know that starting from contest 1 and pressing `next` will take both $\mathcal{A}$ and $\mathcal{C}$ into contest 2. In particular, $\mathcal{C}$ starts in contest 1 and pressing `next` takes it into contest 2. At the end of test 2, $\text{CVR}_{\mathcal{C}} = (\emptyset, \{1\}, \emptyset, \ldots, \emptyset)$. Because the test passed, $\text{CVR}_{\mathcal{C}}$ and $\text{CVR}_{\mathcal{A}}$ must match. Therefore, by properties P_2, P_3, and P_5, $\mathcal{A}$ must have been in contest 2 when the $\texttt{select}_1$ button was pressed. From the first test, we know $\mathcal{A}$ starts in contest 1. So, pressing `next` while in contest 1 transitions $\mathcal{A}$ to contest 2, just as it does for $\mathcal{C}$.

By continuing through the rest of the test cases in the Navigation Coverage test suite, we know that for any contest numbers i, j in $\mathcal{C}$, where pressing a navigation button b takes $\mathcal{C}$ from contest i to contest j, $\mathcal{A}$ would also transition to contest j if given input b while in contest i. This is true because for any contest i in $\mathcal{C}$, all navigation transitions from i in $\mathcal{C}$ have been tested and result in a state with the contest number of $\mathcal{C}$ equal to the contest number of $\mathcal{A}$. A transition of the contest number portion of state depends only on the current contest number and the navigation button pressed, not on the selection state or any other button (Structural Property 4), so all transitions are tested by the Navigation Coverage test suite.

Since $\mathcal{A}$ and $\mathcal{C}$ start with the same initial contest number, and since, if $\mathcal{A}$ and $\mathcal{C}$ start in a state with the same contest number, they will both transition in the same way on a given navigation button input, then by induction, for any trace starting from the initial state, $\mathcal{A}$ and $\mathcal{C}$ will always have equal contest numbers.

Step 2
There was a step in one of the navigation tests when $\mathcal{C}$ had state $(m, i, \{1\})$. Let that be step k. Because the test passed, we know the tester's interpretation of $\mathcal{A}$'s output at step k, z_k, matched the state of $\mathcal{C}$: $I_{\mathcal{O}}(z_k) = (m, i, \{1\})$. We also know, from Step 1 of this proof, that the contest number portion of $\mathcal{A}$'s state at step k matches that of $\mathcal{C}$. Let $(m, i, \hat{s}_i)$ be $\mathcal{A}$'s state at step k. By the definition of $\mathcal{A}$, $z_k = \rho(m, i, \hat{s}_i)$, so $I_{\mathcal{O}}(\rho(m, i, \hat{s}_i)) = (m, i, \{1\})$. Because the contest number portion of the output screen depends only on the contest number and not on any selections made (Structural Property 6), we assume the interpretation of the contest number portion of state depends only on the contest number part of the output screen and not on the selections made. Therefore, $I_{\mathcal{O}}(\rho(m, i, \_)) = (m, i, \_)$ for any selection state. Therefore at step j of $\tau_{\mathcal{A}}$, $I_{\mathcal{O}}(\rho(m, i, s_i)) = (m, i, \_)$.

We have shown that $\mathcal{A}$ is correct for the contest portion of state. □

Lemma 3 (Selection State)**.** *Suppose $\mathcal{A}$ passes the Navigation Coverage and Selection Coverage suite of tests. For any complete trace $\tau_{\mathcal{A}}$ of $\mathcal{A}$, for any step j, where $\mathcal{A}$ is in state (m, i, s_i) and displaying output screen $z = \rho(m, i, s_i)$, we have $I_{\mathcal{O}}(z) = (m, i, s_i)$ and moreover, if we run $\mathcal{C}$ on the sequence of inputs $I_{\mathcal{I}}(\tau_{\mathcal{A}})$, then at step j, $\mathcal{C}$ is in state (m, i, s_i).*

We prove Lemma 3 in two steps. In the first step we prove that at step j, the selection state of the current contest of $\mathcal{A}$ will equal the selection state of the current contest of $\mathcal{C}$. In the second step we complete the proof of Lemma 3 and show that $I_{\mathcal{O}}(z) = (m, i, s_i)$.

Proof. Step 1
The first test case in SC_1^1 shows that $\mathcal{A}$ and $\mathcal{C}$ both start in an initial state with no selections made in contest 1. Consider first the state of $\mathcal{C}$. By construction, $\mathcal{C}$ starts in contest 1 with no selections made. By property P_2 when the `summary` and `cast` buttons are pressed, the selection state does not change. The final output of $\mathcal{C}$ after the first test case will be $\text{CVR}_{\mathcal{C}} = (\emptyset, \emptyset, \ldots, \emptyset)$. No candidates are chosen in any contest. Because the test passed, we know $\text{CVR}_{\mathcal{C}}$ and $\text{CVR}_{\mathcal{A}}$ must match. By assumption A_0, $\mathcal{A}$ received the same inputs

during the test that $\mathcal{C}$ did. By Lemma 2, $\mathcal{A}$ was in the same contest as $\mathcal{C}$: contest 1. By property P_2, the selection state of $\mathcal{A}$ did not change during the test, so $\mathcal{A}$ started in contest 1 with no selections made.

By similar reasoning, because the second test case in SC_1^1 passed, we know that, starting from the initial state, pressing $\texttt{select}_1$ will select the first candidate in contest 1 for both $\mathcal{A}$ and $\mathcal{C}$. $\mathcal{C}$ starts in contest 1 with no selections made and pressing $\texttt{select}_1$ selects the first candidate, but leaves $\mathcal{C}$ in contest 1. At the end of test 2, $\text{CVR}_\mathcal{C} = (\{1\}, \emptyset, \emptyset, \ldots, \emptyset)$. Because the test passed, $\text{CVR}_\mathcal{C}$ and $\text{CVR}_\mathcal{A}$ must match. From the previous test, we know $\mathcal{A}$ started in an initial state with no selections made. By property P_2, only the $\texttt{select}_1$ input could have affected the selection state. Also, by Lemma 2, $\mathcal{A}$'s contest number will remain equal to $\mathcal{C}$'s contest number. Therefore, pressing $\texttt{select}_1$ in contest 1 while in a state with no selections made transitions $\mathcal{A}$ to be in contest 1 with the first candidate selected, just as it does for $\mathcal{C}$.

Continuing through the rest of the test cases in SC_1^1 shows that, for each candidate i in contest 1, starting from the initial state and pressing $\texttt{select}_i$ will transition both $\mathcal{A}$ and $\mathcal{C}$ to a state where candidate i is chosen in contest 1. By similar logic (and relying on Lemma 2 to ensure $\mathcal{C}$ and $\mathcal{A}$ are always in the same contest), the test cases in SC_1^2 show that, for each candidate i in contest 2, starting from the initial state and pressing $\texttt{select}_i$ will transition both $\mathcal{A}$ and $\mathcal{C}$ to a state where candidate i is chosen in contest 2. The same holds for contests $3 - N$, using test cases $SC_1^3, SC_1^4, \ldots, SC_1^N$.

Going back to contest 1, the test cases in SC_3^1 show that, when in contest 1, with the first candidate already selected, selecting any other candidate will transition both $\mathcal{A}$ and $\mathcal{C}$ in the same way. Consider the first test case in SC_3^1. After the first $\texttt{select}_1$ button is pressed, we know from the previous tests that both $\mathcal{A}$ and $\mathcal{C}$ will be in the first contest with the first candidate selected. After the $\texttt{select}_1$ button is pressed a second time, $\mathcal{C}$ will transition to a state where no candidates are selected in the first contest. Proceeding through `summary` and `cast` will not change any of the selected state and after the test $\text{CVR}_\mathcal{C} = (\emptyset, \emptyset, \ldots, \emptyset)$. Because the test passed, $\text{CVR}_\mathcal{A}$ and $\text{CVR}_\mathcal{C}$ must match. From previous tests we know $\mathcal{A}$ was in contest 1 with the first candidate selected after the first $\texttt{select}_1$ input. We also know, from property P_2, that only the second $\texttt{select}_1$ input could have changed the selection state. Therefore, when $\mathcal{A}$ is in contest 1 with the first candidate selected and receives the input $\texttt{select}_1$, it transitions to a state where no candidates are selected in contest 1, just as $\mathcal{C}$ does.

By continuing through the rest of the test cases in the Selection Coverage test suite, we know that for any two states (m, i, s_i) and (m, i, s_i') in $\mathcal{C}$, where pressing a selection button b takes $\mathcal{C}$ from state (m, i, s_i) to state (m, i, s_i'), $\mathcal{A}$ would also transition to (m, i, s_i') if given input b while in state (m, i, s_i). This is true because for any state (m, i, s_i) in $\mathcal{C}$, all selection transitions from (m, i, s_i) in $\mathcal{C}$ have been tested and result in a state with the selection state of the current contest equal to that of $\mathcal{A}$. A selection transition depends only on the current contest number, selection state of the current contest, and selection button pressed, not on the selection state of any other contest, nor on any other button (Structural Property 3), so all transitions are tested by the Selection Coverage test suite.

To extend the suite of Selection Coverage test cases for use in an election where n candidates out of k can be chosen in each contest, the following test cases must be added: For each contest i, for each selection state s_i seen in some test case, for each selection button b, add a new test case where $\mathcal{C}$ is first advanced to state (m, i, s_i), then input b is given, followed immediately by inputs `summary` and `cast`.

Since $\mathcal{A}$ and $\mathcal{C}$ start with the same initial selection state, and by Lemma 2 $\mathcal{A}$ and $\mathcal{C}$ always have the same current contest number, and since, if $\mathcal{A}$ and $\mathcal{C}$ start in a state with the same current contest number and selection state for the current contest, they will both transition in the same way on a given selection button input, then by induction, for any trace starting from the initial state, $\mathcal{A}$ and $\mathcal{C}$ will always have the same current contest number and selection state for the current contest. Furthermore, since the state of a contest is updated independently of the state of any other contest, and only the current contest can have its selection state updated (properties P_1 and P_3), for any trace starting in the initial state, $\mathcal{A}$ and $\mathcal{C}$ will always have the same selection state for all contests.

Step 2

There was a step in one of the selection tests when $\mathcal{C}$ had state (m, i, s_i). Let that be step k. Because the test passed, we know the tester's interpretation of $\mathcal{A}$'s output at step k, z_k, matched the state of $\mathcal{C}$: $I_\mathcal{O}(z_k) = (m, i, s_i)$. We also know, from Lemma 2 and Lemma 3, that $\mathcal{A}$'s state at step k is also (m, i, s_i). Therefore, $I_\mathcal{O}(\rho(m, i, s_i)) = (m, i, s_i)$. Because the output screen depends only on the current contest number and the state of the current contest, we know $I_\mathcal{O}(\rho(m, i, s_i) = (m, i, s_i)$ whenever $\mathcal{A}$ has current state (m, i, s_i), regardless of the selection states of any other contests. Therefore, at step j of $\tau_\mathcal{A}$, $I_\mathcal{O}(\rho(m, i, s_i)) = (m, i, s_i)$. □

The proof of Theorem 1 follows from Lemmas 1, 2, and 3.

Proof. Consider the complete trace $\tau_{\mathcal{A}} = z_0, (x_1, y_1), z_1, (x_2, y_2), z_2, \ldots, z_l$. Applying the interpretation function gives us:

$$I(\tau_{\mathcal{A}}) = (I_{\mathcal{O}}(z_0), I_{\mathcal{I}}(z_0, (x_1, y_1)), I_{\mathcal{O}}(z_1), I_{\mathcal{I}}(z_1, (x_2, y_2)), \ldots, I_{\mathcal{O}}(z_z))$$

The corresponding sequence of button presses is:

$$(I_{\mathcal{I}}(z_0, (x_1, y_1)), I_{\mathcal{I}}(z_1, (x_2, y_2)), \ldots, I_{\mathcal{I}}(z_{l-1}, (x_l, y_l)))$$

Let $b_i = I_{\mathcal{I}}(z_{i-1}, (x_i, y_i))$ and let $\tau_{\mathcal{C}}$ be the trace of $\mathcal{C}$ on inputs $(b_1, b_2, \ldots, b_l)$:

$$\tau_{\mathcal{C}} = ((m_0, i_0, s_{0,i}), b_1, (m_1, i_1, s_{1,i}), b_2, \ldots, (m_l, i_l, s_{l,i}))$$

By determinism of $\mathcal{C}$ (property P_0), we know $\tau_{\mathcal{C}}$ is uniquely determined by $(b_1, b_2, \ldots, b_l)$. We wish to prove $\tau_{\mathcal{C}} = I(\tau_{\mathcal{A}})$. In other words, we want to prove $I_{\mathcal{O}}(z_j) = (m_j, i_j, s_{j,i})$ for all j such that $0 \leq j \leq l$. From Lemmas 1, 2, and 3, we know that at step j, if $\mathcal{C}$ has state $(m_j, i_j, s_{j,i})$, then $\mathcal{A}$ also has state $(m_j, i_j, s_{j,i})$. We also know from Lemmas 1, 2, and 3 that $I_{\mathcal{O}}(m_j, i_j, s_{j,i}) = (m_j, i_j, s_{j,i})$. Therefore $I_{\mathcal{O}}(z_j) = (m_j, i_j, s_{j,i})$ for all j such that $0 \leq j \leq l$ and $\mathcal{A}$ is correct. □

10 Discussion

In the following sections we discuss limitations of our work and discuss the assumptions we rely on to meet our goal of correctness.

10.1 Limitations

The input and output interfaces of our design have a number of limitations that would need to be addressed before the design could be implemented for use in real-world elections in the United Sates.

One major limitation is the design's inability to capture a vote for a write-in candidate. Our current testing criteria require a tester to cover each possible output screen at least once. This is manageable since the number of output screens is limited by the number of candidates in each contest. If each contest included a write-in field of reasonable size (perhaps 20 characters), the number of possible output screens would quickly grow to an unmanageable size. Addressing this limitation would require developing a new verification and testing strategy for the write-in part of each contest.

A second limitation is the lack of alternate input and output devices for use by people with disabilities. For example, our testing procedure does not cover voting machines with audio output.

There are also some ballot styles in use in the United States which our design could not handle. For example, some jurisdictions allow straight-ballot voting in which an early contest screen allows the voter to choose a party instead of a particular candidate and then all future selections are made automatically according to the voter's choice of party. Our testing methodology relies heavily on the independence of individual contests and would not be sound if it was possible for one contest to affect the results of subsequent contests.

In addition to the limitations of our design, there are also some limitations of our prototype implementing the design. These result from a lack of interfaces to off-chip peripherals. These limitations may be seen as less critical as they may be addressed by increased engineering effort without requiring changes to the design, verification, or testing strategies. For example, we did not implement the hardware to interface with off-chip memory. This meant we were limited to using the small amount of on-chip memory available in our FPGA kit. As a result the images we used for displaying each screen were extremely simplified – it is fair to say that only the developers would be able to correctly interpret each output screen. A second consequence of our lack of any external interface was that the EDF could not be read in from external memory. Instead, we hard-coded in the values of a particular EDF to use during testing. Likewise the final cast vote record produced by each test was left on internal memory and read out programmatically.

10.2 Assumptions

Our definition of correctness makes a number of assumptions about the voting machine's environment and users. Those that our proof relies on explicitly have already been mentioned elsewhere in the paper. Here we describe those assumptions more fully and list some additional assumptions that are implicit in our correctness guarantee.

Our proof of correctness assumes the input and output interpretation functions employed by the tester are the same interpretation functions that will be used by every voter. In other words, if a tester says that a particular test passes, then any voter would say the same. This is a strong assumption to rely on and we make no claim that we have reason to believe it will always hold. However, there are some steps that might be taken to increase the validity of the assumption. The easiest thing might be to have the tester make a note of any screen that seems at all confusing or ambiguous. A further step might be to employ multiple testers for increased confidence that the interface is unambiguous. Finally, usability experts might be brought in to help design a clear and unambiguous interface.

A second assumption that some of our verified properties rely on is that the voting machine will be loaded with a well-formed EDF. There may be many EDFs in use for a particular election and each one needs to be formatted according to our specification in order for our correctness guarantees to apply. To gain confidence in the validity of this assumption, the process of checking EDFs could be automated. Since the specification for an EDF is complete and all EDFs must follow the same format, an external tool could be written to read in each EDF and check that it is well-formed.

There are a number of additional assumptions that our guarantee of correctness relies on which we state here without any discussion of the additional measures that might be taken to provide confidence in the validity of these assumptions. Identifying what those measures might be is outside the scope of this paper. We assume the FPGA synthesizer correctly implements the verified HDL code. We assume the DRE that is tested is the one that shows up on election day and that it has not been tampered with. We assume the DRE screen is calibrated during testing as it will be on election day. And finally, we assume there will be no hardware failures on election day.

11 Related Work

There are a handful of companies providing most of the commercial DREs in use today in the United States [31]. These include Election Systems & Software, Hart InterCivic, Premier Election Solutions (Diebold), and Sequoia Voting Systems. These systems provide features such as write-in candidates and straight party voting, which we do not provide and which would make our proof of correctness considerably more difficult. These features are often required by state law, but they make the design of the DRE considerably more complex. Many states require that all DREs undergo logic and accuracy testing before election day, but multiple security reviews have revealed that this testing is not enough [3, 6, 7]. To our knowledge, no commercial DRE in use today has been formally verified to be correct.

There have been other (non-commercial) DREs designed to provide greater assurance in the correctness of the voting system. Like ours, none of these machines are production-ready. The frog system proposed physically separating the vote generation component from the vote casting component into two separate voting machines. This reduces the trust required of the former and reduces the complexity of the latter [5]. Sastry et al. showed how to maintain the independence properties of physical separation within a single voting machine through the use of separate microprocessors with small, well-designed communication mechanisms and trusted I/O multiplexing. [26]. They also introduced the idea of physically resetting the voting machine between each voting session to provide a guarantee of non-interference between sessions. We use the idea of separation and independence between modules and extend it to provide separation between and independence of different contests on the same ballot. This enables our use of testing as part of our proof of correctness. Rather than physical separation, we use logical separation and use formal methods to prove the necessary independence and non-interference properties. We also use the idea of resetting the machine to an initial state before every voter.

Pvote [32–34] showed that is was possible to drastically reduce the complexity of an interactive DRE by pre-rendering the pages displayed to the voter. We use this idea, although we do not provide certain features, such as the ability to select a write-in candidate, which Pvote does.

Our design borrows many of the techniques introduced by these earlier machines to reduce the complexity of the DRE; we take advantage of the lessened complexity to provide formal verification of the machine, which none of the earlier work did.

An entirely different technique for gaining confidence in the voting system is to provide mechanisms to give confidence in the final vote totals, rather than in the machines capturing the votes. One such technique uses cryptography to provide end-to-end verification, which has the goal of showing that all cast votes were correctly captured and all captured votes were correctly counted and included in the final tally [1,4,9,11,12,15,20,22–25,27]. Today's end-to-end systems try to provide not just correctness guarantees, but also guarantees about privacy, coercion resistance, reliability, or usability. These additional properties are outside the scope of our work. One of the above end-to-end systems, VoteBox Nano [20], is, like our system, implemented on an FPGA in order to remove the complexities added by an OS or language runtime system.

Voter verified paper audit trails (VVPATs) also aim to provide assurance in the correctness of the total vote count, rather than in any individual machines or components [2, 14, 18, 19]. The idea is to provide a paper printout of the selections the voter has made. Before the voter casts her ballot, she can inspect the paper ballot and check all her selections are correctly marked. After the election is over, the tally provided by the DRE is audited by checking a percentage of the paper ballots to make sure they match the results returned by the DRE. There have been numerous publications in recent years proposing various auditing schemes; however, the audit relies on the VVPAT being correct and it is not clear that voters would catch any errors in the VVPAT. There is research showing that voters are not good at catching errors on the final election review screen of a DRE, although it is not known how this translates to voters' ability to catch discrepancies between a paper VVPAT and the electronic review system [8, 13].

Neither end-to-end verification nor VVPATs obviate the need for confidence in the correctness of the DRE itself. These techniques are designed to catch errors, if they occur, either during or after an election, but it is not always clear what actions should be taken in the event of an error occurring. Our approach is meant to preclude those errors from occurring in the first place and is therefore complementary to either of the above techniques and could be used in conjunction with either.

12 Conclusion

We have presented a Direct Recording Electronic (DRE) voting machine that is designed with the goal of making testing and verification possible. We show that by logically separating each contest into its own state machine and proving non-interference properties between contests, it is possible to conduct sufficient user testing to prove the voting machine is correct. We extend previous work on this design to include a summary screen for the DRE.

Acknowledgments

Many thanks to my collaborators: Susmit Jha for his help with the verification; Sanjit Seshia for his expertise in formal methods; and David Wagner for his overall advice, and in particular, his guidance on writing proofs.

References

[1] B. Adida. Helios: Web-based open-audit voting. In *Proceedings of the 17th USENIX Security Symposium*. USENIX Association, 2008.

[2] A. Appel. Effective audit policy for voter-verified paper ballots. Presented at 2007 Annual Meeting of the American Political Science Association, Sept. 2007. `http://www.cs.princeton.edu/~appel/papers/appel-audits.pdf`.

[3] A. W. Appel, M. Ginsburg, H. Hursti, B. W. Kernighan, C. D. Richards, G. Tan, and P. Venetis. The New Jersey voting-machine lawsuit and the AVC Advantage DRE voting machine. In *Proceedings of*

the conference on Electronic Voting Technology/Workshop on Trustworthy Elections, EVT/WOTE'09, pages 5–5, Berkeley, CA, USA, 2009. USENIX Association.

[4] J. Benaloh. Simple verifiable elections. In *Proceedings of the USENIX/ACCURATE Electronic Voting Technology Workshop*, pages 5–5, Berkeley, CA, USA, 2006. USENIX Association.

[5] S. Bruck, D. Jefferson, and R. L. Rivest. A modular voting architecture ("Frogs"). In *Workshop on Trustworthy Elections*, WOTE, August 2001.

[6] K. Butler, W. Enck, H. Hursti, S. McLaughlin, P. Traynor, and P. McDaniel. Systemic issues in the Hart InterCivic and Premier voting systems: Reflections following project EVEREST. In *Proceedings of the USENIX/ACCURATE Electronic Voting Technology Workshop*, EVT. USENIX Association, 2008.

[7] J. A. Calandrino, A. J. Feldman, J. A. Halderman, D. Wagner, H. Yu, and W. P. Zeller. Source code review of the Diebold voting system, July 2007. Report commissioned as part of the California Secretary of State's Top-To-Bottom Review of California voting systems.

[8] B. A. Campbell and M. D. Byrne. Now do voters notice review screen anomalies? A look at voting system usability. In *Proceedings of the Electronic Voting Technology/Workshop on Trustworthy Elections*, EVT/WOTE, 2009.

[9] D. Chaum, R. Carback, J. Clark, A. Essex, S. Popoveniuc, R. L. Rivest, P. Y. Ryan, E. Shen, and A. T. Sherman. Scantegrity II: End-to-end verifiability for optical scan election systems using invisible ink confirmation codes. In *Proceedings of the USENIX/ACCURATE Electronic Voting Technology Workshop*, EVT, July 2008.

[10] D. Cho. "Fairfax Judge Orders Logs of Voting Machines Inspected". *The Washington Post*, November 6 2003. `http://www.washingtonpost.com/ac2/wp-dyn/A6291-2003Nov5`.

[11] M. R. Clarkson, S. Chong, and A. C. Myers. Civitas: Toward a secure voting system. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2008.

[12] A. Essex, J. Clark, U. Hengartner, and C. Adams. Eperio: Mitigating technical complexity in cryptographic election verification. In *Proceedings of the Electronic Voting Technology/Workshop on Trustworthy Elections*, EVT/WOTE, 2010.

[13] S. P. Everett. *The Usability of Electronic Voting Machines and How Votes Can Be Changed Without Detection*. PhD thesis, Rice University, 2007.

[14] G. A. Gianelli, J. D. King, E. W. Felten, and W. P. Zeller. Software support for software-independent auditing. In *Proceedings of the Electronic Voting Technology/Workshop on Trustworthy Elections*, EVT/WOTE, 2009.

[15] R. Haenni and O. Spycher. Secure internet voting on limited devices with anonymized DSA public keys. In *Proceedings of the Electronic Voting Technology/Workshop on Trustworthy Elections*, EVT/WOTE, 2011.

[16] P. McDaniel, M. Blaze, G. Vigna, and et al. EVEREST: Evaluation and Validation of Election-Related Equipment, Standards and Testing, Dec 2007. `http://www.sos.state.oh.us/SOS/upload/everest/14-AcademicFinalEVERESTReport.pdf`.

[17] K. McMillan. Cadence SMV, 1998. `http://www.kenmcmil.com/`.

[18] R. Mercuri. *Electronic Vote Tabulation Checks & Balances*. PhD thesis, School of Engineering and Applied Science of the University of Pennsylvania, 2000.

[19] L. Norden, A. Burstein, J. L. Hall, and M. Chen. Post-Election Audits: Restoring Trust in Elections, Aug 2007. `http://www.brennancenter.org/page/-/d/download_file_50227.pdf`.

[20] E. Öksüzoğlu and D. S. Wallach. VoteBox Nano: A smaller, stronger FPGA-based voting machine. In *Proceedings of the USENIX/Accurate Electronic Voting Technology Workshop / Workshop on Trustworthy Elections*, EVT/WOTE, 2009.

[21] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[22] S. Popoveniuc, J. Kelsey, A. Regenscheid, and P. Vora. Performance requirements for end-to-end verifiable elections. In *Proceedings of the Electronic Voting Technology/Workshop on Trustworthy Elections*, EVT/WOTE, 2010.

[23] P. Y. A. Ryan. Prêt à voter with confirmation codes. In *Proceedings of the Electronic Voting Technology/Wokrshop on Trustworthy Elections*, EVT/WOTE, 2011.

[24] P. Y. A. Ryan and S. Schneider. Prêt à voter with re-encryption mixes. In *ESORICS*. Springer-Verlag, 2006.

[25] D. Sandler, K. Derr, and D. S. Wallach. VoteBox: a tamper-evident, verifiable electronic voting system. In *Proceedings of the 17th USENIX Security Symposium*. USENIX Association, 2008.

[26] N. Sastry, T. Kohno, and D. Wagner. Designing voting machines for verification. In *Proceedings of the 15th USENIX Security Symposium*, 2006.

[27] A. T. Sherman, R. A. Fink, R. Carback, and D. Chaum. Scantegrity III: Automatic trustworthy receipts, highlighting over/under votes, and full voter verifiability. In *Proceedings of the Electronic Voting Technology/Wokrshop on Trustworthy Elections*, EVT/WOTE, August 2011.

[28] C. Sturton, S. Jha, S. A. Seshia, and D. Wagner. On voting machine design for verification and testability. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS, pages 463–476, New York, NY, USA, 2009. ACM.

[29] C. Thompson. "Can You Count on Voting Machines?". *The New York Times Magazine*, January 6 2008. `http://www.nytimes.com/2008/01/06/magazine/06Vote-t.html`.

[30] Verified Voting. "America's Voting Systems in 2010". *VerifiedVoting.org*, Accessed Oct. 2011. `http://www.verifiedvoting.org`.

[31] Verified Voting. "State Election Equipment". *VerifiedVoting.org*, Accessed Oct. 2011. `http://www.verifiedvoting.org/verifier/`.

[32] K.-P. Yee. *Building Reliable Voting Machine Software*. PhD thesis, University of California Berkeley, 2007.

[33] K.-P. Yee. Extending prerendered-interface voting software to support accessibility and other ballot features. In *Proceedings of the USENIX/Accurate Electronic Voting Technology Workshop*, EVT, 2007.

[34] K.-P. Yee, D. Wagner, M. Hearst, and S. M. Bellovin. Prerendered user interfaces for higher-assurance electronic voting. In *Proceedings of the USENIX/Accurate Electronic Voting Technology Workshop*, EVT, 2006.

[35] Yices SMT solver. `http://yices.csl.sri.com/`.