On Voting Machine Design for Verification and Testability

Cynthia Sturton Susmit Jha Sanjit A. Seshia David Wagner

EECS Department
University of California, Berkeley
Berkeley, CA 94720
{csturton,jha,sseshia,daw}@eecs.berkeley.edu

ABSTRACT

We present an approach for the design and analysis of an electronic voting machine based on a novel combination of formal verification and systematic testing. The system was designed specifically to enable verification and testing. In our architecture, the voting machine is a finite-state transducer that implements the bare essentials required for an election. We formally specify how each component of the machine is intended to work and formally verify that a Verilog implementation of our design meets this specification. However, it is more challenging to verify that the composition of these components will behave as a voter would expect, because formalizing human expectations is difficult. We show how systematic testing can be used to address this issue, and in particular to verify that the machine will behave correctly on election day.

Categories and Subject Descriptors

B.5.2 [**Register-Transfer-Level Implementation**]: Design Aids; D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.4 [**Software/Program Verification**]; H.1.2 [**Human factors**]

General Terms

Design, Security, Verification, Human Factors

1. INTRODUCTION

Electronic voting has brought with it concerns about reliability, accuracy, and trustworthiness. A challenge with using technology to run elections is that it is difficult to be sure that a complex computer system will perform correctly and as desired. Existing electronic voting systems provide a relatively low level of assurance. They are complex systems, often consisting of hundreds of thousands of lines of code, and a single bug anywhere in the code could potentially cause votes to be lost, misrecorded, or altered. As a result, it is difficult for independent evaluators to be confident that these systems will record and count the votes accurately. Moreover, in order to completely verify the voting machine, it is necessary to also verify the interface to human voters, i.e., that the operation of the voting machine is consistent with the behavior expected by voters.

In this paper, we present a new approach: design an electronic voting machine with assurance that it will work correctly. The novelty of our approach is in the manner in which we integrate design, formal verification, and systematic testing to certify correctness. We make three key contributions. First, we present the design of a voting machine where design decisions are made so as to ease verification and testability. Second, we formally and automatically verify that the implementation satisfies a number of low-level correctness properties. Third, we show how to verify that the machine will behave on election day in a manner consistent with the voters' expectations of correct operation, by using a combination of systematic testing and formal methods. We elaborate below on this integration of design, verification, and testing.

We follow the philosophy of "design for verification" and "design for testability." Rather than waiting until the implementation is finished to begin verification or testing, every design choice was made with an eye towards its impact on our ability to verify these properties, and the implementation was built with verification in mind from the start. In many cases, we started with the properties we wanted to prove, and then considered how to design the system so that these properties would be easy to verify.

We formally verify the correctness of each of the individual modules of the voting machine, as well as verifying some crucial correctness properties of their composition. For each module, we construct a formal specification that fully characterizes the intended behavior of that module. Also, we identify a number of structural and functional properties that the machine as a whole must satisfy. We use automated techniques such as model checking and satisfiability solving to verify that our Verilog implementation meets these specifications. We emphasize that we apply formal verification to the actual code that is executed, not just to a high-level abstract model. One of the contributions of this work is to demonstrate that formal verification of voting machine logic is feasible.

In addition, we use systematic manual testing to check that the machine will behave on election day in a way consistent with voter expectations. Verifying this with formal methods is challenging because it is difficult to formally specify a voter's expectations. Instead, we assume that an observant human tester can recognize incorrect behavior if it should occur (e.g., if the voting machine records a vote for a candidate the tester did not select, the tester will detect this fact). We generate a set of tests that, in combination with the formal verification, are provably sufficient to explore all possible behaviors of the machine, and we employ human testers to systematically check that the machine works correctly in these tests.

Testing is well-suited for checking election-specific behavior, such as that the machine is properly configured for this election with the correct set of contests and candidates. However, a limitation of conventional testing is that exhaustive testing takes too long (e.g. exponential time), so in practice testing can only find bugs; it cannot guarantee their absence. Perhaps surprisingly, we show that this limitation of testing can be eliminated if testing is combined with a limited amount of formal verification and if the machine is designed appropriately. In particular, we show that for an election with N contests in which each contest involves selecting one out of k candidates, we can verify correctness with just $O(kN)$ tests, instead of the $\Omega(k^N)$ tests exhaustive testing would require.

The kind of voting machine that we focus on in this paper is known as a direct-recording electronic (DRE) voting machine. A DRE voting machine is one where voters interact with the machine to make their selections and then the votes are recorded electronically. The most familiar example is a touchscreen voting machine, where the voter interacts with a graphical user interface displayed on the screen by software running on the voting machine. The voter presses at various locations on the screen to register her selections, and the voting software records the voter's selections once she is ready to cast her ballot. DREs are widely deployed throughout the US: for instance, in 2008 DREs were used by approximately 33% of registered voters [2].

DRE's are commonly thought to be large, complex machines, but we demonstrate that a small finite-state machine is sufficient to build a functional DRE. Our design is a finite-state machine that implements a bare-bones, stripped-down DRE. We implement the machine directly in hardware, in custom Verilog code, so that there is no operating system or runtime to verify. At present, one limitation of our implementation is that it supports only the minimum functionality needed to conduct an election, and does not support many features typically found in today's deployed DREs. However, using this stripped-down version helps to enable verification and has the added benefit that the complete state machine can be "held in your head", allowing for better design decisions than can be commonly achieved in commodity software.

In order to deem an electronic voting system secure, one must consider everything from the machine on which users make their selections and the tabulator that counts votes to the poll workers on election day and the layout of the polling place [16]. Our goal in this work is to provide a provably correct electronic voting machine that can provide a foundation for secure elections. We do not claim that a provably correct voting machine is sufficient for a secure voting system, but it is certainly necessary.

The paper is structured as follows. We mathematically specify the intended behavior of the machine in Section 2. We describe the design principles and architecture of our voting machine in Section 3, the implementation in Section 4 and then discuss in Sections 5 and 6 how we formally verified that the components of the implementation met their specifications. Section 7 explains how we use systematic testing to check that the composed machine behaves as desired. Finally, we conclude the paper with a discussion of related work and limitations of our approach.

2. SPECIFICATION

In order to validate a voting machine and guarantee its correctness, we need to formalize the specifications and properties that we are trying to prove. We focus on four verification goals:

1. Each individual component of the voting machine must work correctly (i.e., meet its specification) when considered in isolation;
2. When these components are composed, the resulting machine must satisfy certain behavioral properties that we would expect a correct voting machine to satisfy;
3. The voting machine must be structured in a way that makes our use of systematic testing sound; and,
4. When configured with a particular election definition file, the voting machine must display and allow selection of candidates in accordance with the election definition, and must behave and record votes in a way consistent with what a typical human voter would expect.

The first three are properties of the voting machine's design alone and thus can be verified once and for all, without any election-specific information. The fourth is election-specific and, in our approach, must be verified separately for each election.

To enable verification of the first three goals above, our specification includes the following three parts:

- for each component of the voting machine, a formal specification of the desired behavior of that component;
- behavioral properties of the voting machine, specified as statements in some formal logic; and,
- structural properties, specifying which input variables and state variables each state variable can depend upon.

Sections 3.2, 3.3, and 3.4 formalize these parts of the specification. In our approach, these are verified using formal verification techniques, namely, model checking [7] and satisfiability checking [32].

This leaves the question of how to formalize our fourth verification goal: that the voting machine must behave consistently with human expectations. This is much more difficult to cleanly specify in a precise, mathematical manner. For instance, if there is a rectangular region on the screen that displays "Thomas Jefferson" in some readable font, a human might expect that pressing that portion of the screen would select Jefferson, causing Jefferson's name to be highlighted and eventually causing a vote to be recorded for Jefferson if no other selection is subsequently made in this contest. However, because it involves semantic interpretation of the contents of a particular screen image by a human it is not clear how to specify this expected behavior in a precise, mathematical fashion. For instance, given a bitmap image, mechanically recognizing which portions of the screen a human would expect to correspond to a touchable region might require non-trivial image processing; moreover, mechanically determining that the touchable region should be associated with Thomas Jefferson might require OCR and other complex computation. Formalizing these kinds of human expectations in a formal logic would be horribly messy, and probably error-prone as well.

We take a different approach: we involve humans in the validation process. In particular, we ask human voters to cast test votes on the voting machine during pre-election testing. We ask them to check that the machine seems to be working correctly and recording their votes accurately. We assume that if the machine behaves in a way inconsistent with their expectations, they will notice and complain. Consequently, if the voting machine passes all of these tests, then at least we know that the voting machine has behaved in a way consistent with human expectations during those tests. We assume the voting machine will be used in the election only if it passes all of these tests.

In addition, we formally verify that the voting machine behaves deterministically. This ensures that the voting machine will behave the same way on election day as it did in pre-election testing. So, if a real voter interacts with the machine on election day in exactly the same way as some tester did during pre-election testing, then we can be confident that the machine will behave correctly and will record the voter's vote in accordance with the voter's intentions.

However, this alone is not enough to provide useful guarantees in practice, because the number of tests needed to exhaustively test all possible machine behaviors is astronomically large. For instance, in an election with N contests and k choices in each contest, the number of different ways to vote (assuming voters are only allowed to vote for a single candidate in each contest) is k^N, an exponential function of N. Taking into account the possibility to change one's selections in a contest as many times as one likes, the number of ways to interact with the voting machine becomes infinitely large. Clearly, we cannot exhaustively try all of these possibilities in pre-election testing: we need something more selective.

Our approach involves conducting many fewer tests: something like $O(kN)$ tests. We prove that, if the machine behaves as expected in each of these tests, then it will behave as expected for every possible interaction. Of course, this does not follow in general: for any fixed set of tests, one can devise a machine that works correctly on those tests but behaves incorrectly on some other interaction. We are able to show that if the voting machine has a particular structure, then a limited number of tests suffice.

Very roughly speaking, if the state and behavior for each contest is independent of the state of all other contests, it suffices to choose a test suite that attains 100% transition coverage in each individual contest and of navigation between contests, rather than 100% coverage of the whole voting machine's statespace. This can be achieved with $O(k)$ tests per contest, since the state space in a single contest is only of size $O(k)$ (whereas the statespace for the entire voting machine has size $O(k^N)$ and thus would require exponentially many tests to fully cover).

Therefore, we must verify that our voting machine has the appropriate structure needed in order to apply these results, e.g., that it behaves deterministically and that its state and behavior in each contest is independent of the state of all other contests. The structural properties, mentioned earlier and described in more detail in Section 3.4, are intended to capture these requirements and ensure that our use of systematic testing is sound.

It is also necessary to formalize what it means for the voting machine to behave as a human would expect. We model this in two pieces: a model of human expectations for how the voting machine should respond to inputs; and human interpretation of the meaning of each screen image produced by the voting machine. We formalize the former by defining a *specification voting machine* (Section 2.2), which captures our assumptions about how voters will expect the voting machine to update its internal state in response to inputs from the voter. The specification machine specifies, for instance, how the set of candidates currently selected should be updated when the voter presses a button. However, the specification machine does not specify what kinds of screen images should be produced by the voting machine: it is solely concerned with the evolution of the internal state of the machine.

Reasoning about the interface provided by the voting machine to human voters requires us to reason about how humans will interpret any particular screen image. Therefore, we assume the existence of an *interpretation function* I that maps screen images to their human interpretation (Section 2.3). For the purposes of this paper, we assume that all humans will interpret any given screen image in the same way, and thus a single function suffices; usability issues are outside the scope of this paper. In particular, we assume that testers will interpret screen images in the same way as voters. We make no attempt to specify I formally. Instead, we devise a set of tests that suffice to check that the screen images produced by the voting machine will be interpreted by humans in a way that accurately represents the internal state of the machine, no matter what I may be. The assumptions mentioned above cannot be mathematically proven; rather, they serve as a way to make precise what assumptions we do and do not make about the nature of human expectations.

Accordingly, the fourth part of our specification is:

- a formal model of a human's view of how the voting machine should operate, which we call the *specification voting machine*.

The fourth verification goal listed above then becomes to verify that the actual voting machine's behavior is consistent with the specification machine: i.e., that a human will interpret the screen images produced by the actual voting machine in a way consistent with how the specification machine mandates that its state should evolve. This is verified using systematic testing. As mentioned above, we provide formal verification of structural properties of the actual voting machine which, when combined with testing using a test suite that attains 100% transition coverage in each individual contest and of navigation between contests, is sufficient to guarantee trace correspondence between the voting machine and the specification voting machine.

In the rest of this section, we describe the specification machine in detail. The formal specification of each component, behavioral properties, and structural properties are presented in Section 3.

2.1 Notation and definitions

We begin by briefly defining some voting-related terms that are used throughout the discussion.

Contest: A single race, such as President, that a voter will vote on.

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.

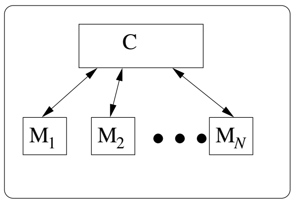
Button: A (usually rectangular) region on the screen. Touching anywhere within this region activates a particular functionality of the machine. The corresponding part of the screen image is often designed to provide the appearance of a physical button.

Next we provide definitions for the transducers that make up the specification voting machine.

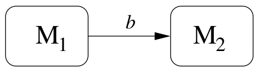
DEFINITION 1. *A deterministic finite-state transducer M is a 6-tuple $(\mathcal{I}, \mathcal{O}, \mathcal{S}, \delta, \rho, s_{init})$ where*

- *$\mathcal{I}$ is the set of input events,*
- *$\mathcal{O}$ is the set of outputs,*
- *$\mathcal{S}$ is the set of states of M,*
- *$\delta : \mathcal{S} \times \mathcal{I} \to \mathcal{S}$ is the transition function,*
- *$\rho : \mathcal{S} \to \mathcal{O}$ is the output function, and*
- *$s_{init} \in \mathcal{S}$ is the initial state of M.*

We introduce the notion of controlled composition, where we compose N transducers $M_1, \ldots, M_N$ and a controller C whose output set is $\{1, \ldots, N\}$. The output of C determines which transducer is active. Inactive transducers produce no output and do not transition between states. We assume that the M_i's share a common



(a) Controlled composition



(b) Sequential composition

Figure 1: Composition

input set $\mathcal{I}_M$, and that the input set of C is disjoint from $\mathcal{I}_M$. Any input from $\mathcal{I}_M$ will be routed to whichever transducer is currently active, and any other input will be provided to C. The output of the composition is taken from the output of whichever transducer is currently active. See Figure 1(a) for a visualization. Formally:

Definition 2. *Let $C, M_1, \ldots, M_N$ be deterministic finite-state transducers, given by*

- $C = (\mathcal{I}_C, \{1, \ldots, N\}, \mathcal{S}_C, \delta_C, \rho_C, s_C^{init})$ *and*
- $M_i = (\mathcal{I}_M, \mathcal{O}, \mathcal{S}_i, \delta_i, \rho_i, s_i^{init})$.

Suppose $\mathcal{I}_C \cap \mathcal{I}_M = \emptyset$. Then their controlled composition, denoted $(\!|C; M_1, \ldots, M_N|\!)$, is the transducer

$$(\mathcal{I}_C \cup \mathcal{I}_M, \mathcal{O}, \mathcal{S}_C \times \mathcal{S}_1 \times \cdots \times \mathcal{S}_N, \delta^*, \rho^*, (s_C^{init}, s_1^{init}, \ldots, s_N^{init})),$$

where δ^* and ρ^* are defined as

$$\delta^*((i, s_1, \ldots, s_N), b) = \begin{cases} (\delta_C(i, b), s_1, \ldots, s_N) & \text{if } b \in \mathcal{I}_C, \\ (i, s_1, \ldots, s_{i-1}, \delta_i(s_i, b), s_{i+1}, \ldots, s_N) & \text{if } b \in \mathcal{I}_M; \end{cases}$$

$$\rho^*(i, s_1, \ldots, s_N) = \rho_i(s_i).$$

We also use the notion of sequential composition of transducers. If M_1, M_2 are two transducers with common input and output sets $\mathcal{I}, \mathcal{O}$, and b is another input not in $\mathcal{I}$, then the sequential composition $M_1 \xrightarrow{b} M_2$ is a transducer that initially begins executing M_1. When it receives input b, it immediately transfers control to M_2, starting at the initial state of M_2. Equivalently, let C be a two-state transducer that begins executing in state 1 and transitions to state 2 upon receiving input b, and whose output function is the identity function. Then $M_1 \xrightarrow{b} M_2 = (\!|C; M_1, M_2|\!)$.

2.2 Specification Voting Machine

The specification voting machine formalizes how voters will expect the voting machine to respond to inputs from the voter, and how voters will expect the final votes recorded by the machine to correspond to these actions. Therefore, we start with a list of probable voter expectations:

1. If the voter presses a button for a candidate who is not already selected, then the effect will be to add that candidate to the list of selected candidates if this is legal (and nothing else will change). If the voter presses a button for a candidate who is already selected, the effect will be to remove that selection.
2. If the voter presses a button to navigate among contests, the voting machine will do so appropriately.
3. When the voter casts his/her ballot, the state of each contest equals the state of that contest the last time the voter saw that contest's screen.

From these expectations we develop a specification voting machine. Instead of outputting screen images (like the actual voting machine), it outputs only an abstract representation of what should be displayed upon the screen. Similarly, instead of receiving as input (x, y)-locations where the voter touched the screen (like the actual voting machine), the specification voting machine receives only an abstract representation of the button pressed. This abstract button number not distinguish between different locations corresponding to the same button.

The specification machine $\mathcal{P}$ is a deterministic finite-state transducer with a special structure, depicted in Figure 2. Conceptually, $\mathcal{P}$ operates in two modes in which its operation is respectively defined by two transducers M_{main} and M_{cast}. M_{main} represents the *main mode* of operation, in which $\mathcal{P}$ begins and processes voter selections, and M_{cast} is the *cast mode* in which the voter casts her ballot and the machine records the vote and resets itself for the next voter. M_{main} is itself the composition of $N + 1$ state machines, M_{nav}, M_1, M_2, ..., M_N, where M_{nav} controls navigation across contests, and M_i is the state machine responsible for processing votes for contest i.

Formally, $\mathcal{P}$ is the sequential composition $\mathcal{P} = M_{\text{main}} \xrightarrow{\texttt{cast}} M_{\text{cast}}$. M_{main} is the controlled composition $M_{\text{main}} = (\!|M_{\text{nav}}; M_1, \ldots, M_N|\!)$. M_{cast} is a transducer with only a single state and a self-loop on every input.

Definition 3. *$\mathcal{P}$ is a 6-tuple $(\mathcal{I}, \mathcal{O}, \mathcal{S}_\mathcal{P}, \delta_\mathcal{P}, \rho, s_{init})$ where*

- *$\mathcal{I}$ is the set of input events from the voter, corresponding to buttons that the voter can press,*
- *$\mathcal{O}$ is the set of outputs from the specification machine,*
- *$\mathcal{S}_\mathcal{P}$ is the set of states of the specification machine,*
- *$\delta_\mathcal{P} : \mathcal{S}_\mathcal{P} \times \mathcal{I} \to \mathcal{S}_\mathcal{P}$ is the transition function,*
- *$\rho : \mathcal{S}_\mathcal{P} \to \mathcal{O}$ is the output function, and*
- *$s_{init} \in \mathcal{S}_\mathcal{P}$ is the initial state of $\mathcal{P}$ for each voter.*

Note that this formulation requires $\mathcal{P}$ to be a *deterministic* finite-state transducer. It also requires the output to depend only upon the current state, not upon the input.

The output set $\mathcal{O}$ of the specification machine is partitioned into two kinds of outputs: $\mathcal{O} = \mathcal{O}_{\text{main}} \cup \mathcal{O}_{\text{cast}}$. Each element of $\mathcal{O}_{\text{main}}$ has the form (i, s_i), where i indicates the current contest and s_i is the set of candidates selected in that contest. This is an abstraction of the information that should be displayed on the screen at that point, and is output when the machine is in main mode. Each element of $\mathcal{O}_{\text{cast}}$ has the form $(s_1, \ldots, s_N)$, representing a record of the votes cast in all N contests, and is output when the machine is in cast mode.

The input set $\mathcal{I}$ of the specification machine is partitioned into two sets: $\mathcal{I} = \mathcal{I}_N \cup \mathcal{I}_S$. The set $\mathcal{I}_N$ corresponds to buttons that a voter can press to navigate between contests, while $\mathcal{I}_S$ corresponds to buttons that a voter can press to select or deselect the options within a contest. We use $\mathcal{I}_N = \{\texttt{next}, \texttt{prev}, \texttt{cast}\}$; `next` moves from contest i to contest $i+1$, `prev` moves from contest i to contest $i - 1$, and `cast` irrevocably casts the voter's ballot and moves to a final screen informing the voter that her vote has been recorded. Also we assume $\mathcal{I}_S = \{0, 1, \ldots, k - 1\}$, where k is an upper bound on the number of choices in any contest. The event $b \in \mathcal{I}_S$ corresponds to pressing a button to select/deselect the b^{th} candidate in the contest that is currently active.

Specification of M_{nav}. The controller M_{nav} can be specified formally as follows. Its state set is $\{1, 2, \ldots, N\}$, corresponding to the set of N contests, with 1 as its initial state, corresponding to the fact that the specification machine starts at the first contest. Its

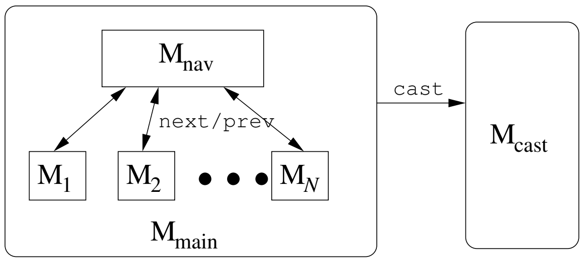


Figure 2: Structure of specification voting machine $\mathcal{P}$.

input set is $\{\texttt{next}, \texttt{prev}\}$. Its transition function is given by

$$\delta_{\text{nav}}(i, b) = \begin{cases} i+1 & \text{if } i < N \text{ and } b = \texttt{next} \\ i-1 & \text{if } i > 1 \text{ and } b = \texttt{prev} \\ i & \text{otherwise.} \end{cases}$$

Its output function is the identity function: $\rho_{\text{nav}}(i) = i$.

Specification of M_i, the machine for contest i. We specify the transducer M_i for an arbitrary contest i (for $1 \le i \le N$). This transducer is implicitly parametrized by two election-specific parameters: k_i, the number of candidates in contest i; and ℓ_i, the maximum number of candidates who can be selected at a time in contest i. For instance, in a contest where the candidate is entitled to vote for up to 2 candidates, out of a list of 10, we have $\ell_i = 2$ and $k_i = 10$.

A state s_i of M_i is a set of candidates who are currently selected in contest i: namely, $s_i \subseteq \{0, \dots, k_i - 1\}$ where $|s_i| \le \ell_i$; $j \in s_i$ indicates that the j^{th} candidate in contest i is currently selected. The initial state is $\emptyset$, indicating that no selection has been made in this contest. The input set is $\mathcal{I}_S$. The transition function is given by

$$\delta_i(s_i, b) = \begin{cases} s_i \cup \{b\} & \text{if } b \notin s_i, b < k_i, \text{ and } |s_i| < \ell_i \\ s_i \setminus \{b\} & \text{if } b \in s_i \\ s_i & \text{otherwise.} \end{cases}$$

The output function ρ_i is given by $\rho_i(s_i) = (i, s_i)$.

States of $\mathcal{P}$. Given the structure of $\mathcal{P}$ defined above, the overall set of states $\mathcal{S}_{\mathcal{P}}$ of $\mathcal{P}$ can be written as $\mathcal{S}_{\mathcal{P}} = \mathcal{S} \times \{Main, Cast\}$. The set $\mathcal{S}$ is in turn partitioned into states of $M_{\text{nav}}, M_1, M_2, \dots, M_N$: $\mathcal{S} = \{1, \dots, N\} \times S_1 \times \cdots \times S_N$. The overall initial state is $s_{\text{init}} = (s_0, Main)$ where $s_0 = (1, \emptyset, \dots, \emptyset)$. The transition function of $\mathcal{P}$ is constructed from δ_{nav} and δ_i as described in Section 2.1. The output function in cast mode is $\rho_{\text{cast}}(i, s_1, \dots, s_N) = (s_1, \dots, s_N)$, and the output function in main mode is constructed from ρ_i as defined in Section 2.1.

Meeting voter expectations. The specification machine models the expected behavior of the voting machine for a single voter. Above, we listed several voter expectations on which we based this specification. $\mathcal{P}$ was designed so that each of these expectations holds by construction. For instance, when the voter casts their ballot they expect the votes cast in each contest to match the voter's last view of that contest. This expectation follows from the separation between the M_is and by the sequential composition of M_{nav} and M_{cast}. The specification states that each M_i can only transition between states when it is active, which is exactly when the voter sees its output on the screen. Furthermore, the transition to M_{cast} can cause no transitions in any of the M_i.

2.3 Interpretation Functions

The touch-screen input-output interface of the voting machine plays a very important role since it is through this interface that a human voter perceives the execution of the voting machine.

Consider the output screen images. It is difficult to predict a priori how a human might interpret any particular screen image. Instead, we assume that everyone will interpret any screen image output by the voting machine in the same way, and introduce a function $I_{\mathcal{O}}$ that maps screen images to their abstract content. If z is a screen image, then $I_{\mathcal{O}}(z)$ is defined by the following thought experiment: we imagine showing z to a prototypical human; we ask the human which contest this screen is associated with, and let $i \in \{1, \dots, N\}$ denote the contest they identified; we ask the human which candidates are currently selected in this contest, and let s_i denote the set of candidates they identify; then $I_{\mathcal{O}}(z) = (i, s_i)$.

Similarly, we introduce an interpretation function $I_{\mathcal{I}}$ that maps a screen image z and an (x, y)-location on that screen to an input in $\mathcal{I}$. $I_{\mathcal{I}}$ formalizes how a prototypical human would map screen locations to buttons.

The crucial assumption we make is that everyone—all voters and testers alike—will use the same input/output interpretation functions. Testing procedures can partially validate this assumption: we can ask testers to check that each screen image output by the voting machine appears unambiguous, and if the voting machine ever outputs a screen image whose interpretation is ambiguous, we can declare the tests a failure. Nonetheless, we still must assume that, if all tests pass, then every voter will interpret each such screen image the same way. Consequently, this assumption cannot be fully rigorously verified and serves to formalize one of the assumptions underlying our approach.

Note that the interpretation functions $I_{\mathcal{O}}$ and $I_{\mathcal{I}}$ are not known a priori. We make no attempt to formally specify or explicitly reconstruct these functions. Instead, we show that if some interpretation functions exist that describes how all humans will interpret each screen image (an unverified assumption), then our testing process suffices.

Formal model. We now define our notion of correctness for the actual voting machine. A *trace* of the specification machine $\mathcal{P}$ is a sequence $(z_0, b_1, z_1, b_2, \dots, z_\ell)$ of outputs and inputs, where $z_j \in \mathcal{O}$ and $b_j \in \mathcal{I}$. A *complete trace* of $\mathcal{P}$ is a sequence $(z_0, \dots, z_\ell)$ where $z_0, \dots, z_{\ell-1} \in \mathcal{O}_{\text{main}}$ and $z_\ell \in \mathcal{O}_{\text{cast}}$.

A trace of the actual voting machine $\mathcal{A}$ is a sequence $(z_0, b_1, \dots, z_\ell)$ of outputs and inputs, where each z_j is a screen image or cast vote record and each b_j is an (x, y)-location where the voter pressed the screen; a complete trace is one where the $z_0, \dots, z_{\ell-1}$ are screen images and z_ℓ is a cast vote record.

If $\tau = (z_0, b_1, z_1, \dots, z_\ell)$ is a trace of $\mathcal{A}$, we define $I(\tau) = (I_{\mathcal{O}}(z_0), I_{\mathcal{I}}(z_0, b_1), I_{\mathcal{O}}(z_1), I_{\mathcal{I}}(z_1, b_2), \dots, I_{\mathcal{O}}(z_\ell))$. For a given I, we say that a trace $\tau_{\mathcal{A}} = (z_0, b_1, z_1, \dots, z_\ell)$ of the actual voting machine $\mathcal{A}$ is *correct* if the trace $\tau_{\mathcal{P}}$ of $\mathcal{P}$ on the input sequence $I_{\mathcal{I}}(z_0, b_1), I_{\mathcal{I}}(z_1, b_2), I_{\mathcal{I}}(z_2, b_3) \dots$, satisfies the relation $\tau_{\mathcal{P}} = I(\tau_{\mathcal{A}})$. Equivalently, $\tau_{\mathcal{A}}$ is correct if and only if $I(\tau_{\mathcal{A}})$ is a valid trace of $\mathcal{P}$. Let $\text{Tr}(\mathcal{P})$ denote the set of traces of $\mathcal{P}$, and $\text{Tr}(\mathcal{A})$ the set of traces of the actual voting machine $\mathcal{A}$. We consider the actual voting machine *correct* if $\text{Tr}(\mathcal{P}) = \{I(\tau) : \tau \in \text{Tr}(\mathcal{A})\}$. Equivalently, $\mathcal{A}$ is correct if and only if every feasible trace of $\mathcal{A}$ is correct. Our testing procedure (Section 7) is designed to prove that the actual voting machine is correct.

3. DESIGN

We designed and implemented a prototype voting machine $\mathcal{A}$; in this section we describe the details of that design. We start with an explanation of the organization of the voting machine followed by a full specification of each module in the machine. We then describe the behavioral and structural properties of the composition of those modules. These properties help us verify that $\mathcal{A}$ is equivalent to $\mathcal{P}$.

3.1 Component Level Design

We use a 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 on 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 8 contests, labeled 0–7, and each contest can have up to 12 candidates. To make the discussion more concrete, we will use these parameters, but an implementation could easily increase them if needed. The full architecture is shown in Figure 3.

Election Definition File. 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, Selection_State, and Display. The particulars of the EDF's content will be explained in the discussion of those three modules.

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 15 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 button. In addition to the buttons for each candidate, every screen also has the navigation buttons `prev` and `next`, which let the voter move from contest to contest, and a `cast` button which allows the user to cast their entire ballot as it currently stands.

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 signal $contest\_num$ identifies 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, lower left and upper right, 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 $\mathcal{P}$ in the remainder of our design. This in turn makes the verification of our implementation simpler.

Controller. The Controller module controls which contest is currently active. It is analogous to M_{nav} in $\mathcal{P}$. The inputs to the module are $button\_num$, $touch$, and $reset$; the outputs are $cast$, $contest\_num$, and ss_enable. The only state maintained by the module is $contest\_num$, the index of the currently active contest. This value changes accordingly as the voter navigates from contest to contest. ss_enable is set when (and only when) the voter presses a button that is valid for selecting or deselecting a candidate in the current contest. $cast$ is set when the voter presses the `cast` button and remains set thereafter, until the machine is reset.

Selection_State. There is one Selection_State module for each possible contest on the ballot: Selection_State_0 . . . Selection_State_7. These correspond to the M_i state machines of $\mathcal{P}$. If an election contains fewer than 8 contests, the remaining Selection_State modules will simply go unused. The state of each module, $selection\_state$, reflects the selections that have been made in that contest and is implemented as a 12-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.

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 M_{cast} in $\mathcal{P}$; the transition to Cast is triggered when the voter presses the `cast` button on their screen.

Display. Pvote showed that pre-rendering of screen images could greatly reduce the complexity of a voting machine [30]. We use this idea and include in our definition of 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. When the user presses the cast button a final screen is displayed indicating that the ballot has been cast.

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

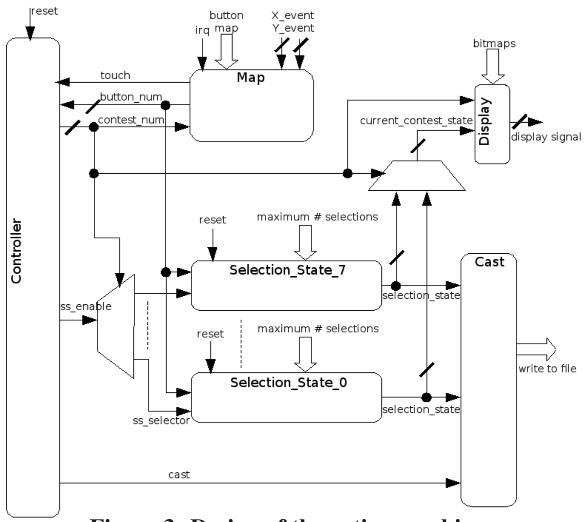


Figure 3: Design of the voting machine.

3.2 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 5). 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. We briefly summarize the specification for each component here.

Map. The $touch$ is set if and only if a touch event occurs (the voter presses somewhere on the touchscreen) *and* the touch is within the boundaries of a touchable button for the current screen as defined by the EDF. If a touch event occurs which causes the $touch$ to be set then the value of $button\_num$ will be the logical button number corresponding to the touchable region pressed by the voter.

Controller. The three output signals should behave as follows:

- $cast$ should be set whenever a valid touch for $button\_num$ = 13 occurs. Once set, cast remains unchanged until $reset$ is set.
- ss_enable should be low whenever $reset$ or $cast$ is high. When neither of those are set then ss_enable should be set when a valid touch for a candidate selection button is touched.
- $contest\_num$ should be low whenever $reset$ is triggered. The value of the signal should increment whenever a valid `next` button is touched and decrements if a valid `prev` button is touched. Otherwise the value of $contest\_num$ should remain unchanged. In addition, the `next` and `prev` buttons must not cause the value of $contest\_num$ to overflow or underflow.

Selection_State. If $ss\_selector$ is high for an instance of the Selection_State module and the button i has been touched, then the i^{th} bit of $selection\_state$ will be toggled as long as this does not cause the total number of bits set in $selection\_state$ to exceed the maximum number of candidates that can be selected in this contest.

Cast. 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 $selection\_state[i]$ into $memory_i$, for each i; thereafter, $memory$ remains unchanged.

3.3 System-Level Behavioral Properties

We identify a number of properties pertaining to the behavior of the voting machine as a whole. These properties are necessary but not sufficient for correct behavior. See Appendix A for the formal specification of these properties in linear temporal logic (LTL) [23]. In Section 5 we discuss how we verified these properties against our implementation.

1. At any given time, no more than one contest can be active.
2. A contest i is active if and only if the current contest number is 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.
4. The selection state of a contest can not change if neither $reset$ nor $ss\_selector$ are set.
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`, and `cast` buttons cannot affect the selection state of any contest.
6. Setting $reset$ clears the selection state for all contests.
7. On reset, the current contest number and cast are reset and selection mode is disabled.
8. Once the voting machine enters $cast$ mode, $cast$ is not cleared until the next cycle of the voting machine beginning with $reset$.
9. 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 $reset$.
10. Selection of a candidate and casting of votes can not take place at the same time.

3.4 Structural Properties

The voting machine is structured to provide several properties, chosen so that testing will suffice to establish the equivalence of the implemented voting machine with the specification voting machine. Section 6 discusses how we verified all 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. For any fixed EDF, when in main mode (i.e., before the ballot is cast), 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 bijective.
6. The final memory storing the selection state should be completely determined by the selection states of the contests before cast.

4. IMPLEMENTION

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

Our implementation differs from the design in one respect: our current prototype does not include an interface to non-volatile storage. In particular, 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 (e.g., registers) to simulate this functionality. This represents a limitation of our current engineering and is not a fundamental shortcoming of our approach.

This limitation has several implications:

1. In our prototype, the EDF is hard-coded into register arrays. Map has a register array containing a button map for a particular election and Selection_State 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; then we'd 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.

As a result, the current screen images would not be usable by anyone other than the system developers. This limitation exists because our FPGA has only a limited amount of on-chip memory available for storage of the bitmap images. In a finished implementation, the EDF would be read from external storage, making it possible to store and use high-resolution images.

5. FORMAL VERIFICATION

Using formal verification techniques we show that our implementation follows our design specifications and satisfies the desired behavioral properties. We used Cadence SMV [17], a symbolic model checker, for verification. The tool includes a Verilog-to-SMV translator so that we were able to run the verification directly on our Verilog implementation.

5.1 Component-Level Specifications

As mentioned in Section 3.2, with the exception of the Display module, we fully specified the behavior of each component in the machine under all possible inputs. For all but the Map module, we used Cadence SMV to verify that the implementation conforms to these specifications. Specifically, we used the SMV notion of a *layer*, a formal specification written in the SMV language. 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, since the large register holding the EDF's button map made the state space too large to model check at the bit level. Instead, 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 [31] was used to complete the verification. Under the assumption that the EDF is valid, we were able to verify that the Map module meets its component-level specification.

5.2 System-Level Behavioral Properties

We formulated each behavioral property from Section 3.3 as an LTL formula and then ran the SMV model checker to verify the property holds true for our implementation. The LTL formulation of each property is given in Appendix A. 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 crosscheck on the other two and gives us increased confidence that they are each correct.

6. VERIFYING STRUCTURAL PROPERTIES

Next, we describe how we verified that the implementation of our voting machine meets the structural properties articulated earlier (Section 3.4). These properties all involve checking that a variable v depends only on some specified set $W = \{w_1, \ldots, w_n\}$ of variables, and nothing else. In other words, we must verify that v can be expressed as a deterministic function of the other variables: $v = f(w_1, \ldots, w_n)$, for some function f, or in shorthand, $v = f(W)$. Put another way, we want to check that v deterministically depends on W, and only W, i.e., for every other variable $x \notin W$, v is conditionally independent of x given W. We verify this kind of property by formulating it as a Boolean satisfiability (SAT) problem.

6.1 Approach

The first step is to express the transducer as a Boolean system. We begin by introducing some notation. We assume there is a set S of state variables, so that each valuation of values to these variables corresponds to a state of the system. Similarly, let I be a set of input variables, and O a set of output variables. For each state variable s, let the variable s' denote the previous value of s; let S' denote the set of these variables. Then we can write the transition relation as a function δ, which expresses the state as a function of the previous state and the input via the relation $S = \delta(S', I)$. (This is shorthand for $s_i = \delta_i(s'_1, \ldots, s'_k, i_1, \ldots, i_\ell)$ for $i = 1, \ldots, k$, assuming $S = \{s_1, \ldots, s_k\}$ and $I = \{i_1, \ldots, i_\ell\}$.) Similarly, we assume the output function is modelled as a function ρ, via the relation $O = \rho(S)$. Thus, we can model the transducer by the formula

$$\phi(S, S', I, O) \equiv \quad S = \delta(S', I) \wedge O = \rho(S).$$

Now suppose we wish to check that state or output variable v is a deterministic function of a set W of state or input variables. Let S_1, S_2 be two copies of the state variables, I_1, I_2 be two copies of I, and O_1, O_2 be two copies of O. Consider the formula

$$\begin{aligned}
&\psi(S_1, S'_1, I_1, O_1, S_2, S'_2, I_2, O_2) \equiv \\
&\quad \phi(S_1, S'_1, I_1, O_1) \wedge \phi(S_2, S'_2, I_2, O_2) \wedge \\
&\quad v_1 \neq v_2 \wedge \forall w \in W \,.\, w_1 = w_2.
\end{aligned}$$

Effectively, we make two copies of our model of the system. We then check whether it is possible for v to take on two different values in the two copies, while all variables in W take on the same value in both copies; the answer reveals whether v depends deterministically upon W. In particular, v can be expressed as a deterministic function of W ($v = f(W)$) if and only if ψ is unsatisfiable. Figure 4(a) illustrates this idea. This approach to checking dependence is similar to the technique of using self-composition for checking information flow [27]. The key idea is to formulate non-interference as a 2-safety property.

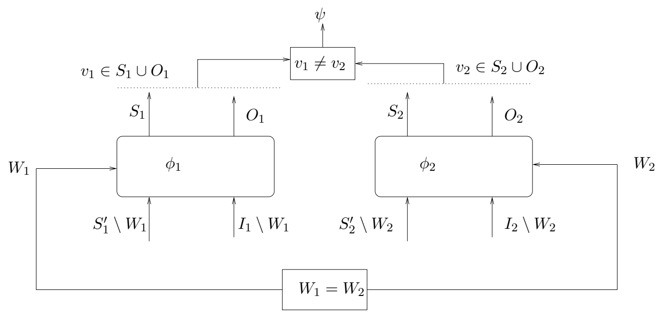
6.2 Application

We verify the structural properties from Section 3.4 by checking deterministic dependence properties. For instance, to help verify structural property 4, we verify that the active contest number depends deterministically upon the previous contest number and the button pressed.

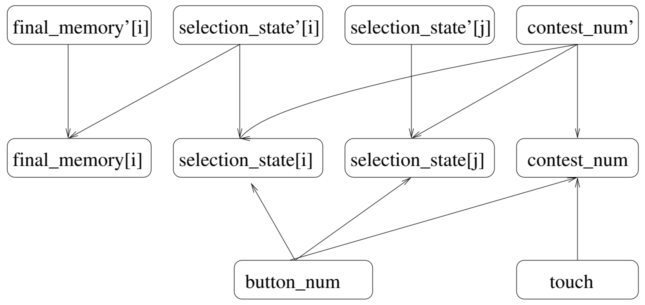
We applied the approach sketched above to the Verilog implementation of our voting machine, using the Beaver SMT solver [4, 13] to check satisfiability.

We illustrate some of the results of our dependency analysis in Figure 4(b). We draw an edge from state/input variable r to state variable s if the value of s after a single transition depends on the value of r before the transition. If there is no edge from r to s, then the new value of s does not depend upon the prior value of r. We excluded the Map and Display modules from this figure for clarity, and so the relevant input variables are $touch$ and $button\_num$, and the state variables are the $selection\_state[i]$ and $memory[i]$ for each contest i as well as $contest\_num$. We include the selection state for two sample contests, $selection\_state[i]$ and $selection\_state[j]$ (where $i \neq j$), to help demonstrate the independence between contests.

We applied dependency analysis to the entire machine and to every module. For instance, we verified that the $selection\_state$ for a contest i is independent of $selection\_state$ for any other contest. Also, $memory$ for contest i depends only on the $selection\_state$ for the same contest and is independent of any other contest.



(a) Satisfiability problem for checking that v deterministically depends on W



(b) Dependency graph for a subset of the state variables in the voting machine. The first row shows the states at time t, the second shows the states at time $t+1$, and the third row shows the input variables.

Figure 4: Checking independence and deterministic dependence

Thus, we were able to verify the independence of contests from each other. For the display module we verified that the current display is a *bijective* function of $contest\_num$ and the current $selection\_state$ when in main mode. This bijectivity will be important when we discuss testing in section 7. In addition, we verified Map is dependent only on its explicit input signals and contains no non-deterministic behavior. In this way, we were able to verify all the structural properties specified in Section 3.4.

6.3 Structural Decomposition

We also verify that the Verilog implementation takes the form of a controlled composition of finite-state transducers, as defined in Section 2.1. In particular, we show that it can be expressed in the form $(\!|M_{\text{nav}}; M_1, \ldots, M_N|\!) \overset{\texttt{cast}}{\rightarrow} M_{\text{cast}}$, augmented with a transition on `reset` back to its initial state.

We do this by showing that the transition relation $S = \delta(S', I)$ can be partitioned into four disjoint cases, representing the four types of transitions that can occur in this controlled composition: navigation (which corresponds to transitions in M_{nav}), contest i (transitions M_i, for some i with $1 \leq i \leq N$), cast (the transition to M_{cast} on `cast`), and reset (the transition on `reset`). Associated with each case x there is a guard g_x, which identifies the condition under which the case is active, a set V_x of state variables updated in this case, and an update rule $v := f(D'_v)$ for each state variable $v \in V_x$. The state variables not in V_x remain unchanged: $v := v'$ for each state variable $v \notin V_x$. We verified that the transition relation can be written as

$$\delta(S', I) = \begin{cases} \delta_{\text{navigate}}(S', I) & \text{if } g_{\text{navigate}}(S', I) \\ \delta_i(S', I) & \text{if } g_i(S', I) \\ \delta_{\text{cast}}(S', I) & \text{if } g_{\text{cast}}(S', I) \\ \delta_{\text{reset}}(S', I) & \text{if } g_{\text{reset}}(S', I). \end{cases}$$

The guards are disjoint: only one of the four guard predicates can be true at any one time. See Table 1 in the appendix.

Also, each state variable can be updated by only one of the four cases: V_{navigate}, V_i, V_{cast}, and V_{reset} are pairwise disjoint. In other words, at each time step, each state variable is controlled by exactly one component. For instance, $contest\_num$ can change only during navigation, $selection\_state[i]$ can change only when contest number i is active and a selection button is pressed, and $memory$ can not change except in cast mode.

This decomposition helps us to verify the structural properties. For instance, we can conclude that the next value of $selection\_state[i]$ may depend upon the previous value of $selection\_state[i]$ but not upon the previous value of any other $selection\_state[j]$, as asserted in structural property 3. The structural properties plays a very important role in testing the voting machine, as discussed in the next section.

7. TESTABILITY

We describe how systematic testing is used in conjunction with the formal verification performed earlier to ensure that the input-output behavior exhibited by the actual voting machine is identical to that exhibited by the specification voting machine.

Specifically, our goal is to give a testing protocol such that if all tests pass, we can conclude that $\mathcal{A}$ is correct as defined in Section 2.3, i.e., that $\mathcal{A}$ and $\mathcal{P}$ are trace-equivalent after application of the interpretation function I to traces of $\mathcal{A}$.

7.1 Preliminaries

We are concerned with testing finite-length behaviors of the voting machine. A *test input* (or just *test*) is a sequence of inputs to the specification voting machine $\mathcal{P}$ of the form $b_1, b_2, \ldots, b_\ell$ where $b_j \in \mathcal{I} \setminus \{\texttt{cast}\}$ for $1 \leq j < \ell$ and $b_\ell \in \mathcal{I}$. A *complete test input* (or *complete test*) is a test input whose last element b_ℓ corresponds to casting the vote; i.e., $b_\ell = \texttt{cast}$.

Recall from Section 2.3 the definitions of *interpretation functions* I, $I_\mathcal{O}$, $I_\mathcal{I}$, and of *traces* of the actual and specification voting machines. We make the following assumption about $I_\mathcal{I}$:

A0: For a given election definition file (EDF), for every screen z and every (x, y) location on the touch screen, $I_\mathcal{I}(z, (x, y)) = \mathsf{Map}(z, (x, y))$.

In the above statement, Map denotes the Map module of $\mathcal{A}$. Note that we have considered the range of $I_\mathcal{I}$ and Map to be the same; strictly speaking, although the ranges are isomorphic to each other, they could be different sets.

Intuitively, a human tester would administer a test $T = (b_1, \ldots, b_\ell)$ by "inverting" the input interpretation function $I_\mathcal{I}$ on each b_j to find a corresponding (x, y)-position a_j to press on. To formalize it a bit more, if the initial screen is z_0 and the first button press in the test sequence is b_1, then we assume that the tester computes a_1 such that $I_\mathcal{I}(z_0, a_1) = b_1$, and then presses the screen at (x, y)-position a_1. The tester then observes the next screen image, say z_1, finds a_2 such that $I_\mathcal{I}(z_1, a_2) = b_2$, and presses the screen at (x, y)-position a_2. This process continues, yielding a sequence $T_\mathcal{A} = a_1, a_2, \ldots, a_\ell$ of inputs to the actual voting machine $\mathcal{A}$.

Let $\tau_\mathcal{A}$ be the trace exhibited by $\mathcal{A}$ on input $T_\mathcal{A}$, and let $\tau_\mathcal{P}$ be the trace exhibited by $\mathcal{P}$ on input T. As noted earlier, we ensure by design and formal verification that $\mathcal{A}$ and $\mathcal{P}$ are deterministic, meaning that for any T, there exists exactly one $\tau_\mathcal{A}$ and exactly one

$\tau_{\mathcal{P}}$. If $I(\tau_{\mathcal{A}}) = \tau_{\mathcal{P}}$, we say that $\mathcal{A}$ is *correct on test* T or that test T *passes*.

Intuitively, at each step, the tester will check the output screen to make sure that the voting machine appears to have responded correctly, according to their expectations about correct behavior (e.g., after selecting a candidate, the candidate should be highlighted or otherwise appear to be selected). After casting their ballot, the tester will inspect the cast vote record produced by the voting machine and check that it appears to be correct (i.e., it is consistent with the selections the tester has made during this test, according to their interpretation of the test inputs). If any of these checks fail, the human tester will fail the machine; otherwise, the human tester will pass the machine. Based upon our assumptions about human expectations, as formalized in Section 2.3, we assume that the behavior of human testers can be modelled as follows:

A1: We assume that 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.

A *test suite* $\mathcal{T}$ is a set of complete tests. We say that $\mathcal{T}$ passes if every $T \in \mathcal{T}$ passes.

We assume that if any test fails, the voting system will not be used in an election. Therefore, we wish to identify a condition on $\mathcal{T}$ so that if every test in $\mathcal{T}$ passes, then we can be assured that $\mathcal{A}$ is correct in the sense defined in Section 2.3: i.e., it is trace-equivalent to $\mathcal{P}$ after application of the interpretation function. We identify such a sufficient condition on $\mathcal{T}$ in Section 7.2.

Such a result is only possible if we know that the voting machine has a certain structure. We rely upon the following properties of the actual and specification voting machines:

P0: The output function of the voting machine is a bijective function of the contest number and selection state of the current contest.
P1: The voting machine is a deterministic transducer.
P2: The state of a contest is updated independently of the state of other contests.
P3: If a navigation button is pressed, the selection state remains unchanged.
P4: If a selection button is pressed, the current contest number stays unchanged.

These five properties have been formally verified for $\mathcal{A}$, as described in Sections 3, 5 and 6. For $\mathcal{P}$, properties P0–P4 follow from the specification given in Section 2.2.

In addition, we require another property of $\mathcal{A}$:

P5: The electronic cast vote record that is produced when we cast the ballot is an accurate copy of the selection state for each contest.

Property P5 has been formally verified under the assumption that the machine's record of the cast vote is correctly output on persistent storage (e.g., paper) for the human to check.

7.2 Coverage Criteria

We say that a test suite $\mathcal{T}$ satisfies our coverage criteria if the resulting set of traces of $\mathcal{P}$ satisfies the following conditions:

C0: (*Initial State Coverage*) There is a test in which, from the initial output screen z_0, $\mathcal{P}$ receives the `cast` input.
C1: (*Transition Coverage*)
- **(a)** (*Selection transitions*) For every contest i, every selection state s_i within contest i, and every input $b \in \mathcal{I}_S$, there is some trace where $\mathcal{P}$ receives b in a state (i, s) where the ith component of s is s_i.
- **(b)** (*Navigation transitions*) For every contest i, and every input $b \in \mathcal{I}_N$, there is some trace where $\mathcal{P}$ receives b in a state of the form (i, s).

C2: (*Output Screen Coverage*) For every contest i and every selection state s_i of $\mathcal{P}$ within contest i, there is some trace of $\mathcal{P}$ where *the last transition* within contest i ended at s_i and then at some point thereafter $\mathcal{P}$ receives the `cast` input.

Criterion C0 specifies that a human tester must verify that $\mathcal{A}$ and $\mathcal{P}$ start with the same selection state.

Criterion C1 specifies that we must cover all transitions of each M_i within $\mathcal{P}$ (for every i), and all transitions of M_{nav} within $\mathcal{P}$.

Criterion C2 ensures that for every (i, s_i), the tester gets an opportunity to view the output screen for that s_i as the last step in contest i. This in turn ensures (through properties P2 and P3) that this selection state of $\mathcal{A}$ for contest i appears on the cast vote record. The main purpose of this criterion is to check that the interpretation function $I_{\mathcal{O}}$ is consistent with $\mathcal{A}$'s output function, call it $\rho_{\mathcal{A}}$, as defined by $\mathcal{A}$'s `Display` module. Formally, $I_{\mathcal{O}}$ must be the inverse of $\mathcal{A}$'s output function $\rho_{\mathcal{A}}$. We know from Property P0 that $\rho_{\mathcal{A}}$ is invertible, and that in contest i, it is only a function of (i, s_i), not of any s_j for $i \neq j$. Thus, effectively, C2 ensures that for every (i, s_i) pair, the human tester computes $(i', s_i') = I_{\mathcal{O}}(\rho_{\mathcal{A}}(i, s_i))$. The test passes only if $i = i'$ and $s_i = s_i'$; i.e., only if $I_{\mathcal{O}}$ is the inverse of $\rho_{\mathcal{A}}$. We formalize this result in Appendix C.

7.3 Main Theorem

We now state our main theorem that shows how our test coverage criteria and results of formal verification combine to ensure trace-equivalence of $\mathcal{A}$ and $\mathcal{P}$ (up to an application of I).

Recall from Section 3.2 that $\mathcal{A}$ is *correct* iff $\text{Tr}(\mathcal{P}) = \{I(\tau) : \tau \in \text{Tr}(\mathcal{A})\}$.

THEOREM 1. *Consider a test suite $\mathcal{T}$ that satisfies coverage criteria C0–C2. Then, $\mathcal{T}$ passes if and only if $\mathcal{A}$ is correct.*

The "if" part of the above theorem follows trivially. For brevity, we only sketch out the proof of the "only if" part here, and include the full proof in Appendix C.

PROOF. (sketch) Briefly, the proof works by induction on the length of the input sequence to the voting machine.

The idea is as follows. Consider an arbitrary input sequence T to $\mathcal{A}$. Due to determinism, we know that there is a single trace $\tau_{\mathcal{A}}$ of $\mathcal{A}$ on T. Correspondingly, we can extract the sequence $I(\tau_{\mathcal{A}})$, the sequence of button presses for $\mathcal{P}$, and $\mathcal{P}$'s trace $\tau_{\mathcal{P}}$.

Due to coverage criterion C1, we know that each transition in $\tau_{\mathcal{P}}$, for each (i, s_i) pair, has been covered by some test. We case-split on the input supplied on this transition and show that in each case, the verified properties P0–P5 ensure that the contest numbers and selection states in $I(\tau_{\mathcal{A}})$ are identical to those in $\tau_{\mathcal{P}}$. $\square$

7.4 A Sample Testing Protocol

We present one way to meet the coverage criteria C0–C2 stated above. We assume that it is possible to provide a `cast` input from any screen of the voting machine. This is clearly possible for $\mathcal{P}$, by definition. It is also possible in the voting machine we have designed.

The testing protocol builds upon the following ideas:

1. Due to properties P2–P4, we can cover transitions of M_{nav} and each M_i by separate tests. Note that transitions can be partitioned into two sets depending on the inputs labeling those transitions (in $\mathcal{I}_S$ or $\mathcal{I}_N$).

2. Each trace makes selections in at most one contest. For each i and for each transition of M_i, there is a trace that makes no selections in any contest other than i (simply following `next` button presses to reach contest i) and at some point follows this transition in M_i.
3. Similarly, for each transition of M_{nav}, we explore the shortest trace that ends in that transition and then a `cast`.

Operationally, human testers would do the following:

1. Supply the input sequences (`prev`), (`next`), (`next`, `prev`), (`next`, `next`), (`next`, `next`, `prev`), etc., each followed by a `cast`, and then check that the correct output is received. Each of these tests contains 0 to $N+1$ `next`s, followed possibly by a `prev`, followed by a `cast`. These tests cover the transitions of M_{nav}, and in particular, note that they satisfy coverage criterion C0. For N contests, there are $O(N)$ tests of this form.
2. Recall that in contest i, the selection state s_i is a set $s_i \subseteq \{0, 1, \ldots, k_i - 1\}$, subject to the constraint $|s_i| \leq \ell_i$, where contest i involves selecting at most ℓ_i out of k_i possible candidates. For each contest i and for each valid selection state s_i in this contest, perform the test
$$(\texttt{next}^i, c_1, \ldots, c_j, 0, 0, 1, 1, 2, 2, \ldots, k-1, k-1, \texttt{cast}),$$
where $s_i = \{c_1, \ldots, c_j\}$. Intuitively, we navigate to contest i using i `next`s; we select the candidates specified by s_i; we try de-selecting and subsequently re-selecting each selected candidate; we try selecting and subsequently de-selecting each unselected candidate; and finally we cast the ballot. If the machine is working properly, de-selecting and immediately re-selecting a candidate (or vice versa) should leave the selection state unchanged, and thus return one to the same output screen as before. In the special case of selecting exactly ℓ_i candidates, i.e., where $|s_i| = \ell_i$, selecting an unselected candidate should have no effect, so doing that twice should also have no effect.
Note that these tests satisfy coverage criteria C1 and C2. If every contest involves selecting at most one candidate (i.e., $\ell_1 = \cdots = \ell_N = 1$), there are $O(N \cdot k)$ tests of this form.

Thus, if the voting machine is correct, this protocol certifies its correctness with $O(N \cdot k)$ tests (assuming $\ell_1 = \cdots = \ell_N = 1$).

8. DISCUSSION

Related Work. There has been considerable prior work on using formal verification to build high-assurance systems. In many cases, the designers manually constructed a model of the system and then formally verified that the model satisfies desirable properties. In comparison, we directly verify the source code itself, which provides higher assurance. Like much prior work on high-assurance systems, we too have carefully chosen our design to be modular and to reduce the size of the correctness-critical portion of the code (the TCB) [15].

We are not the first to propose a new architecture for voting machines, with the goal of greater assurance. The "frog" architecture is based upon separating vote-selection (which is performed on one device) from vote-confirmation (which is performed on another device), to reduce the trust that is needed in the vote-selection device [5]. Later, Sastry et al. showed how to provide this functional separation with a single device, and also introduced the idea of forcibly resetting the system after each voter finishes, to ensure independence of voter sessions [26]. We borrow the idea of using resets for independence. Also, Yee et al. proposed pre-rendering of the user interface as a technique to reduce the size of the TCB, and showed that this makes it possible to build a voting machine with a rich UI in only 300–500 lines of Python code [28–30]. We adopt the pre-rendering approach to simplify interpretation of user inputs and generation of screen images. These systems were built in a general-purpose programming language and thus rely upon the correctness of an OS, language runtime/interpreter, and language libraries; in contrast, because our system is implemented directly in Verilog, we eliminate the need for these elements (although we trust the tools that synthesize a circuit from Verilog) and thus further reduce the size of the TCB. More recently, others have built a voting machine on a FPGA platform [22]. However, none of these systems were subjected to formal verification.

Our use of determinism to help verify complex, application-specific properties was inspired by other work on verification of functional purity [10].

Many authors have explored the use of independent, orthogonal mechanisms to verify the vote totals. Today, one widely deployed example is the use of a voter-verified paper audit trail (VVPAT) printer attached to a DRE, combined with post-election audits of the VVPAT records [3, 12, 14, 18, 19, 21]. However, one recent study showed that about two-thirds of voters failed to notice errors on the summary screen [9], raising questions about the effectiveness of VVPAT records. Many researchers have studied cryptographic mechanisms for end-to-end verification that the voter's vote has been recorded and counted accurately [1, 6, 8, 20, 24, 25]. These techniques provide a way to detect problems after the fact, but may not provide any way to recover from some failures and do not proactively prevent election-day failures. For instance, if the voting machine displays the set of options inaccurately—e.g., inverting the explanation of a bond measure—the voter might be tricked into voting contrary to her intentions (previously dubbed a presentation attack [11]). Preventing these kinds of failures requires verifying the user interface logic to a high degree of assurance, as our work does. On the other hand, our approach provides no way for an ordinary voter to verify, for herself, without trust in complex technology, that her vote was recorded and counted accurately; that requires some form of independent verification. Consequently, we believe that our techniques are complementary to end-to-end verification measures: it would make sense to use both.

Analysis of Limitations. Our general philosophy is to pare a voting machine down to the simplest possible core, and as a result our design provides only a bare minimum of functionality. We do support contests, ballot measures, propositions, and any contest that can be expressed in terms of selecting at most ℓ out of a list of k options. We do not support write-ins, straight-party voting, controlled contests, or cross-endorsement. Some of these are obscure or arguably unnecessary features, but the lack of support for write-ins is a significant limitation.

Also, our system is not as flexible, in terms of the kinds of user interface designs that can be supported, as a system written in a general-purpose programming language. We require that contests be presented one per screen: contests cannot be split across two pages, and one page cannot contain more than one contest. We currently do not support alternative languages, audio output, some kinds of accessible input mechanisms (e.g., sip-and-puff devices), zoomable display, or high-contrast or reverse-color modes. Many of these would be needed before our system could be considered accessible to voters with disabilities. We do not provide a summary screen that voters can use to confirm their selections before casting their ballot; this is a significant limitation.

We have not yet implemented a module to interface with external memory; this will be necessary to store votes in non-volatile storage (e.g., on flash). It also has consequences for both the touch screen and the video output. Our synthesized voting machine includes a module to translate analog inputs from the touchscreen to (x, y) coordinates and a module to drive an LCD display however, both these modules require access to data in the EDF. As a temporary measure, we hard-coded in values needed by the touch screen module and the video output module. In addition, in the case of the video output module we use a very simplified display. We expect the memory interface to be relatively straightforward to implement, and consequently we view these gaps as primarily a shortcoming of our implementation, rather than a fundamental limitation of our architecture.

We do not provide any kind of administrative functionality, e.g., for poll workers to configure the machine. We do not support casting of provisional ballots.

It is reasonable to suppose our architecture could be extended to provide verifiably correct implementations of some of these features, such as alternative languages and summary screens, more easily than others, such as write-ins or audio. Of course, it would be straightforward to extend our design with unverified implementations of these additional features, but we believe that it would be preferable to provide the same high level of assurance for all modes of operation of the voting machine.

9. CONCLUSION

We have designed a simple finite-state voting machine guided by the goals of verification and testability. A voter's view of correct operation was formalized using the concept of a specification voting machine. We used the results of formal verification to develop coverage criteria with which human testers can provably assure, using a reasonable (polynomial) number of tests, the correct operation of a voting machine prior to an election. The code and verification results are available online at http://uclid.eecs.berkeley.edu/vvm/. Although several features remain to be addressed, the presented framework is an important step towards a fully formally-verified voting machine.

Acknowledgments

We thank Brian Lam and Kristian Lyngbaek for their work on the implementation of the voting machine prototype. We also thank David Molnar and the anonymous reviewers for helpful comments on an earlier draft of this paper. This research was supported by NSF grants CNS-0524745, CNS-0644436, and CNS-0627734, SRC contract 1355.001, a University of California Chancellor's Fellowship, and an Alfred P. Sloan Research Fellowship.

10. REFERENCES

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

[2] K. Alexander and P. Smith. Verifying the vote in 2008 presidential election battleground states, Nov. 2008. `http://www.calvoter.org/issues/votingtech/pub/pres2008_ev.html`.

[3] 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`.

[4] Beaver SMT solver for bit-vector arithmetic. `http://uclid.eecs.berkeley.edu/beaver/`.

[5] S. Bruck, D. Jefferson, and R. L. Rivest. A modular voting architecture ("Frogs"). In *Workshop on Trustworthy Elections*, August 2001. `http://www.vote.caltech.edu/wote01/pdfs/amva.pdf`.

[6] 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 *EVT'08: Proceedings of the 2008 USENIX/Accurate Electronic Voting Technology Workshop*.

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

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

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

[10] M. Finifter, A. Mettler, N. Sastry, and D. Wagner. Verifiable functional purity in Java. In *ACM CCS 2008*.

[11] D. Jefferson. New concerns about electronic voting: What VVVPAT cannot fix, Apr. 2004. Personal communication.

[12] D. Jefferson, E. Ginnold, K. Midstokke, K. Alexander, P. Stark, and A. Lehmkuhl. Evaluation of Audit Sampling Models and Options for Strengthening California's Manual Count, July 2007. `http://www.sos.ca.gov/elections/peas/final_peaswg_report.pdf`.

[13] S. Jha, R. Limaye, and S. A. Seshia. Beaver: Engineering an efficient SMT solver for bit-vector arithmetic. In *Proc. Computer-Aided Verification (CAV)*, LNCS 5643. Springer, 2009.

[14] D. W. Jones. Auditing elections. *Communications of the ACM*, 47(10):46–50, Oct. 2004. `http://www.cs.uiowa.edu/~jones/voting/cacm2004.shtml`.

[15] D. W. Jones. *Minimizing the Trusted Base*, Dec. 2004. Presentation for the CSTB Framework for Understanding Electronic Voting, `http://www.cs.uiowa.edu/~jones/voting/nas-cstb2004a.shtml`.

[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] R. Mercuri. A better ballot box? *IEEE Spectrum*, 39(10):46–50, Oct 2002.

[20] C. A. Neff. A verifiable secret shuffle and its application to e-voting. In *8th ACM Conference on Computer and Communications Security (CCS 2001)*.

[21] 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`.

[22] E. Öksüzoğlu and D. S. Wallach. VoteBox Nano: A Smaller, Stronger FPGA-based Voting Machine. In *EVT/WOTE '09: Proceedings of the 2009 USENIX/Accurate Electronic Voting Technology Workshop / Workshop on Trustworthy Elections*.

[23] A. Pnueli. The Temporal Logic of Programs. In *FOCS*, IEEE Press, pp. 46–57, 1977.

[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 *USENIX Security Symposium*. USENIX Association, 2008.

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

[27] T. Terauchi and A. Aiken. Secure information flow as a safety problem. Technical Report UCB/CSD-05-1396, EECS Department, University of California, Berkeley, Jun 2005.

[28] K.-P. Yee. Extending prerendered-interface voting software to support accessibility and other ballot features. In *EVT'07: Proceedings of the 2007 USENIX/Accurate Electronic Voting Technology Workshop*.

[29] K.-P. Yee. *Building Reliable Voting Machine Software*. PhD thesis, UC Berkeley, 2007.

[30] K.-P. Yee, D. Wagner, M. Hearst, and S. M. Bellovin. Prerendered user interfaces for higher-assurance electronic voting. In *EVT'06: Proceedings of the 2006 USENIX/Accurate Electronic Voting Technology Workshop*.

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

[32] L. Zhang and S. Malik. The quest for efficient boolean satisfiability solvers. In *CADE-18: Proceedings of the 18th International Conference on Automated Deduction*, London, UK, 2002. Springer-Verlag.

APPENDIX

A. BEHAVIORAL PROPERTIES

We formalize the behavioral properties from Section 3.3 in LTL.

1. At any given time, no more than one contest can be active.
   $G(ss\_selector[0] + \cdots + ss\_selector[contest\ number - 1] \leq 1)$
2. A contest i is active if and only if the current contest number is i.
   $G((contest\_num = i \wedge ss\_enable) \iff ss\_selector[i])$
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 $ss\_selector$ and $reset$ are not 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 $reset$.
   $\forall i\ G((\neg reset \bigwedge \neg ss\_selector \bigwedge selection\_state[i]) \rightarrow X(selection\_state[i]))$
   $\forall i\ G((\neg ss\_selector \bigwedge \neg selection\_state[i]) \rightarrow X(\neg selection\_state[i]))$
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`, and `cast` buttons cannot affect the selection state of any contest.
   $\forall i\ G(\neg reset \bigwedge button\_num \notin \mathcal{B}_{sel} \bigwedge selection\_state[i] \rightarrow X(selection\_state[i]))$
   $\forall i\ G((button\_num \notin \mathcal{B}_{sel} \bigwedge \neg selection\_state[i]) \rightarrow X(\neg selection\_state[i]))$
6. Setting $reset$ clears the selection state for all contests.
   $\forall i\ G(reset \rightarrow X(\neg selection\_state[i]))$
7. On reset, the current contest number and cast are reset and selection mode is disabled.
   $G((reset \rightarrow X(\neg cast \bigwedge \neg ss\_enable \bigwedge \neg contest\_num))$
8. Once the voting machine enters $cast$ mode, $cast$ is not cleared until the next cycle of the voting machine beginning with $reset$.
   $G(reset \rightarrow (XG(cast \rightarrow cast\ U\ reset)))$
9. 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 $reset$.
   $G(reset \rightarrow (XG(cast \rightarrow \neg ss\_enable\ U\ reset)))$
10. Selection of a candidate and casting of votes can not take place at the same time.
    $reset \rightarrow XG(\neg(cast \bigwedge ss\_enable))$

B. THE TRANSITION RELATION

In Table 1, we show how each of the component of the transition relation can be expressed as a set of guarded update rules. Further, these components are disjoint in their guards. If none of the guards are true, there is no change in the state of the voting machine. Also, in one voting cycle between resets, any state variable is updated only by one component of the transition relation.

C. PROOF OF THEOREM 1

We prove Theorem 1 (from Section 7.3) using induction on the length of the input sequence.
Our proof makes use of the following lemma.

LEMMA 1. *Suppose that a test suite satisfying coverage criterion C2 passes. If there exists a trace of $\mathcal{A}$ s.t. its state at an arbitrary step j has contest number i and selection state s_i in contest i, and the output screen of $\mathcal{A}$ at step j is z, then $I_{\mathcal{O}}(z) = (i, s_i)$.*

PROOF. Denote the output function of $\mathcal{A}$ by $\rho_{\mathcal{A}}$. From Property P0, we know that $\rho_{\mathcal{A}}$ is a function only of i and s_i. Thus, $z = \rho_{\mathcal{A}}(i, s_i)$.
By Coverage Criterion C2, there exists some test T in which the last screen in contest i is z and then some time later the vote is cast. By Property P2 and P3, s_i should appear on the cast vote record as the selection state of the ith contest. If $\mathcal{A}$ is correct on test T, it implies that the selection state of $\mathcal{P}$ in contest i at step j is also s_i, matching the cast vote record. Moreover, since T passed, it must also hold that $I_{\mathcal{O}}(z)$ matches $\mathcal{P}$'s output at step j. Thus, $I_{\mathcal{O}}(z) = (i, s_i)$. $\square$

Since we have proved the above lemma for arbitrary i, s_i, and z, the following corollary is also obtained:

COROLLARY 1. *If a test suite satisfying coverage criterion C2 passes, then $I_{\mathcal{O}}$ is the inverse of $\rho_{\mathcal{A}}$.*

We now return to the proof of the main theorem.

PROOF. (Theorem 1)
Consider an input sequence $A = (a_1, a_2, \ldots, a_\ell)$ to $\mathcal{A}$ of finite but arbitrary length. Each a_j is an (x, y)-location on the touch screen. Let $\tau_{\mathcal{A}} = (z_0, a_1, z_1, a_2, z_2, \ldots, z_\ell)$ be the trace of $\mathcal{A}$ on this input sequence. By determinism of $\mathcal{A}$ (Property P1), we know that $\tau_{\mathcal{A}}$ is unique. Also, we have

$$I(\tau_{\mathcal{A}}) = (I_{\mathcal{O}}(z_0), I_{\mathcal{I}}(z_0, a_1), I_{\mathcal{O}}(z_1), I_{\mathcal{I}}(z_1, a_2), \ldots, I_{\mathcal{O}}(z_\ell)).$$

The sequence of button presses corresponding to A is

$$(I_{\mathcal{I}}(z_0, a_1), I_{\mathcal{I}}(z_1, a_2), \ldots, I_{\mathcal{I}}(z_{\ell-1}, a_\ell)).$$

type of transition (x)	guard (g_x)	update rule
navigation	$(\neg reset \wedge touch \wedge \neg cast$	$contest\_num := f_1(contest\_num, navigation\_buttons)$
cast	$(\neg reset \wedge touch \wedge cast\_button)$	$cast := f_2(cast\_button)$ $\quad final\_memory[i] := f_3(selection\_state[i])$ for all i
contest i	$(\neg reset \wedge touch \wedge \neg cast$	$selection\_state[i] := f_4(selection\_state[i], selection\_buttons, max\_selections)$
reset	$reset$	clear $contest\_num$, $cast$, $selection\_state$, and $final\_memory$

Table 1: Structural decomposition of the prototype voting machine's transition function (see Section 6.3)

Let $b_i = I_{\mathcal{I}}(z_{i-1}, a_i)$. Suppose that $\tau_{\mathcal{P}}$ is the trace of $\mathcal{P}$ on $T = (b_1, b_2, \ldots, b_\ell)$. Let $\tau_{\mathcal{P}} = (z'_0, b_1, z'_1, b_2, \ldots, z'_\ell)$. By determinism of $\mathcal{P}$ (Property P1), we know that $\tau_{\mathcal{P}}$ is unique.

We wish to prove that $\tau_{\mathcal{P}} = I(\tau_{\mathcal{A}})$. In other words, we want to prove that $I_{\mathcal{O}}(z_i) = z'_i$ for all i s.t. $0 \le i \le \ell$.

In fact, we will prove that, in addition to the above equality, the sequences of selection states of $\mathcal{A}$ and $\mathcal{P}$ corresponding to the above input sequence are the same. (We know that these sequences are unique due to determinism of $\mathcal{A}$ and $\mathcal{P}$.) Specifically, if $(s^0, s^1, s^2, \ldots, s^\ell)$ is the sequence of selection states for $\mathcal{A}$ and $({s^0}', {s^1}', {s^2}', \ldots, {s^\ell}')$ is the sequence for $\mathcal{P}$, then $s^j = {s^j}'$ for all j. This result will be used as an "auxiliary invariant" in proving the statement of the theorem.

Base case:

Consider the empty input sequence $A = ()$. Thus, $\tau_{\mathcal{A}} = (z_0)$. From coverage criterion C0, we know that the test $T = (\texttt{cast})$ passed. The traces $\tau_{\mathcal{A}}, \tau_{\mathcal{P}}$ are a prefix of that passing test, so it follows that $I_{\mathcal{O}}(z_0) = z'_0$. Also, since T passed, we know that after T, the selection state of $\mathcal{A}$ was $(\emptyset, \emptyset, \ldots, \emptyset)$. Since `cast` does not change any selection state, this was also the initial selection state. Therefore, $s^0 = (\emptyset, \ldots, \emptyset) = {s^0}'$.

Inductive step:

Suppose that $I_{\mathcal{O}}(z_j) = z'_j$ for all j s.t. $0 \le j < m$. We show that $I_{\mathcal{O}}(z_m) = z'_m$. For convenience, we abbreviate the selection state at the $m-1$th step, s^{m-1}, simply as s.

Let $z'_{m-1} = (i, s_i)$ where i is the contest number and s_i is the state of the ith contest. The full selection state is $s = (s_1, \ldots, s_N)$ where the ith entry is s_i. From the induction hypothesis we know that, at the $m-1$th step, i is the contest number and s is the selection state of both $\mathcal{P}$ and $\mathcal{A}$.

Recall that b_m is the button pressed on the mth step. We will case-split on the form of b_m.

- *Case 1:* $b_m \in \mathcal{I}_N \setminus \{\texttt{cast}\}$.
  Since $b_m \in \{\texttt{next}, \texttt{prev}\}$, by Property P3 and Assumption A0, we know that the selection state s remains unchanged on this transition in T for both $\mathcal{A}$ and $\mathcal{P}$. Thus, $s^m = {s^m}' = s$.
  Suppose that $\mathcal{P}$ transitions on b_m from (i, s) to some state (i', s), as per its definition in Section 2.2. Thus, $z'_m = (i', s_{i'})$.
  By coverage criterion C1(b), there exists some passing test $\hat{T}$ that covers the transition of $\mathcal{P}$ on b_m from some state $(i, \hat{s})$ where the ith component of $\hat{s}$ is s_i. Note that $\mathcal{P}$'s contest number would also change to i' on this transition.
  Consider any input a'_m to $\mathcal{A}$ corresponding to b_m. Consider the transition on a'_m in $\mathcal{A}$ from a state corresponding to contest number i and selection state $\hat{s}$. Since $\hat{T}$ passed, we know that the contest number component of $I_{\mathcal{O}}(z_m)$ is i', matching the contest number of $\mathcal{P}$.
  Thus, by Lemma 1, and since $\mathcal{A}$'s selection state for contest i' is $s_{i'}$, $I_{\mathcal{O}}(z_m) = (i', s_{i'}) = z'_m$.
- *Case 2:* $b_m \in \mathcal{I}_S$.
  Since $b_m \in \mathcal{I}_S$, by Property P4 and Assumption A0, we know that this transition in T leaves the contest number unchanged at i for both $\mathcal{A}$ and $\mathcal{P}$. Also, by Property P2, we know that this transition can only modify the selection states s^m and ${s^m}'$ in their ith components.
  Suppose that $\mathcal{P}$ transitions on b_m from (i, s) to some state (i, s'), as per its definition in Section 2.2. Let $s' = (s_1, s_2, \ldots, s'_i, \ldots, s_N)$. Then, $z'_m = (i, s'_i)$.
  By coverage criterion C1(a), there exists some passing test $\hat{T}$ that covers the transition of $\mathcal{P}$ on b_m from some state $(i, \hat{s})$ where the ith component of $\hat{s}$ is s_i. Note that $\mathcal{P}$'s selection state in the ith contest would also change to s'_i on this transition.
  Consider any input a'_m to $\mathcal{A}$ corresponding to b_m. Consider the transition on a'_m in $\mathcal{A}$ from a state corresponding to contest number i and selection state $\hat{s}$. Since $\hat{T}$ passed, we know that the ith entry in the selection state component of $I_{\mathcal{O}}(z_m)$ is s'_i, matching the corresponding entry for $\mathcal{P}$. Since the value of s'_i does not depend on the state of any contest other than i, this implies that the $\mathcal{A}$'s selection state at step m in $\tau_{\mathcal{A}}$, s^m, is s'. Thus, $s^m = {s^m}' = s'$.
  By Lemma 1, since $\mathcal{A}$'s selection state is s' and contest number is i, $I_{\mathcal{O}}(z_m) = (i, s'_i) = z'_m$.
- *Case 3:* $b_m = \texttt{cast}$.
  Consider the final `cast` input. From the formal verification of the properties in Sections 3.2 and 3.3, and Assumption A0 (that the tester presses a button that $\mathcal{A}$ interprets as `cast`), we know that the contest number i and selection state s of $\mathcal{A}$ remain unchanged on `cast`. Similarly, from Section 2.2, we also know that (i, s) remains unchanged on `cast` by $\mathcal{P}$. Moreover, we know that the final output of $\mathcal{P}$ is s, while by design of $\mathcal{A}$ and Property P5, the cast vote record of $\mathcal{A}$ generates an accurate copy of s. Thus, $I_{\mathcal{O}}(z_m) = z'_m = s$.

In all cases, we have shown that $I_{\mathcal{O}}(z_m) = z'_m$ and that $s^m = (s^m)'$. Thus, by induction we have shown that $I_{\mathcal{O}}(z_i) = z'_i$ for all i s.t. $0 \le i \le \ell$. In other words, $\tau_{\mathcal{P}} = I(\tau_{\mathcal{A}})$. $\square$