# Extending prerendered-interface voting software to support accessibility and other ballot features

Ka-Ping Yee
*University of California, Berkeley*
*ping@zesty.ca*

## Abstract

This work builds on the prerendered user interface (PRUI) approach for high-assurance voting software with a new design that supports synchronized audio and visual output, as well as concurrent input from a touchscreen and an alternate input device. This new design offers access for voters with a range of disabilities while retaining the benefits of PRUI voting systems: the user interface can be designed and audited independently of the system software, and the software needing verification is relatively small and simple. This paper discusses the challenges of supporting accessibility in this architecture, explains how these challenges were addressed, and describes the resulting design. To demonstrate the feasibility of this approach and establish a point of reference for the simplicity of voting machine software, this design has been implemented as a program called Pvote, in 460 lines of Python.

## 1 Introduction

Electronic voting machines offer the promise of many advantages in the user interface offered to voters: ballots can be presented in more languages; the user interface can help voters avoid or correct mistakes; alternate input and output devices can enable voters with disabilities to vote independently. However, there have been severe drawbacks with electronic voting to date: lack of transparency, poor reliability, and vulnerability to attack.

$$\boxed{\textbf{assurance} = \textbf{disclosure} + \textbf{review}}$$

Legislative efforts are underway to require disclosure of the software code in electronic voting machines. However, disclosure does not in itself assure correctness of the voting system or the absence of malicious code; the disclosed code must also be reviewed.

Software security review, whether formal or informal, is notoriously difficult and labour-intensive. The size and complexity of the software is a major barrier to full confidence in voting systems. For example, the source code for the Diebold AccuVote TS machine consists of over 31,000 lines of C++ code and resource scripts (ignoring comments and blank lines). A prior paper [20] proposed an approach to the software complexity problem consisting of three complementary measures:

- Publish the ballot definition supplied to the voting machine, so that the software involved in generating the ballot definition does not need to be verified.
- Publish the anonymous votes recorded by the voting machine, so that the software involved in collecting and tallying the results does not need to be verified.
- Minimize the complexity of the voting machine software that does need to be verified by precomputing as much information as possible about the user interface and including it in the ballot definition.

In the prerendered user interface (PRUI) paradigm, the ballot definition is a platform-independent specification of the voting machine's user interface; the software running on the voting machine is a virtual machine (VM) that presents information to the user as specified by the ballot definition, reacts to user input, and records the user's selections. With a sufficiently small and simple VM, code review becomes tractable, and buggy or malicious code in the VM is less likely to go unnoticed. Making software review less time-consuming allows higher assurance to be attained through more thorough review or multiple independent reviews.

Publishing the ballot definition separately also yields other benefits: the voting user interface can be designed and verified independently of the voting machine software, the user interface design can be updated without requiring recertification of the software, and the user interface design can be tested by anyone on an ordinary personal computer, not just on the specialized voting equipment that will be used on election day.

While the original PRUI voting prototype [20] achieved significant strides in simplifying voting machine software and making it more amenable to verification, it supported only a touchscreen for input and output. Such an interface can only be used conveniently by voters who can see, who can read, and who have sufficient fine motor ability to accurately select items on the screen. But a major motivator for electronic voting machines in the first place is to support the accessibility requirements dictated by HAVA. By failing to support more accessible voting interfaces, the previous work left open the question of just how much software complexity is necessary to fulfill these machines' ostensible reason for existing. The purpose of the current work is to answer that question and to demonstrate that better verifiability can be achieved without sacrificing accessibility and useful functionality.

## 2 Goals

The software system described here is intended to serve as the core user interface component for a voting machine. It is designed so it could be used in many kinds of voting machines, such as electronic ballot markers or printers, DRE (direct recording electronic) machines with or without paper trails, or machines that support end-to-end cryptographic verification schemes. Every electronic voting system needs a reliable and auditable way to present the ballot to the voter, and this work is aimed at addressing that need.

In the PRUI paradigm, the ballot definition is central to the system design. Although the ballot definition format has more the flavour of a data structure than a grammar, nonetheless it is a programming language — a domain-specific "little language" [2] in which ballot definitions are the programs and the VM is the interpreter. The challenge is to design a language restrictive enough to enable meaningful security guarantees, yet flexible enough to support all needed functionality. The following sections set out these security and functionality goals.

### 2.1 Security

The essential task of a voting system is to obtain a fair and accurate measurement of the will of the electorate with respect to a number of multiple-choice questions. What it means for a voting system to be secure is that the system can be relied upon to produce the correct results in the face of determined attempts to subvert the outcome. Thus, security is closely tied to correctness; one could say that security is "correctness despite threats."

One of the most serious threats that is currently poorly addressed in voting systems is the insider threat from software developers. Intentionally placed bugs or backdoors are hard to detect even when software is carefully audited [5]. The persistent failure of the federal testing process to detect major security flaws [6] and the continuing revelations of security vulnerabilities in certified voting systems [15, 16, 3, 4, 1] suggest that voting software has not been audited anywhere near enough to defend against this threat.

Reducing the complexity of voting system software to facilitate thorough review directly addresses the insider threat. For the purposes of this work, correctness of the voting system consists of the following:

- The voting system should not crash or become unresponsive during a voting session.
- The behaviour of the system should be determined entirely by the ballot definition and the user's actions. In particular:
  - The behaviour in each voter's voting session should be independent of previous sessions, for the sake of fairness and voter privacy.
  - The behaviour should be exactly the same during an actual election as during testing.
- Each voter should only be able to cast one ballot.
- The ballot should be cast when and only when so requested by the voter.
- Overvoting should be prevented.
- The instructions, contests, and options on the ballot should be presented correctly.
- The feedback that a voter receives when making and reviewing selections should correctly match the selections that the voter made.
- The selections that are recorded should correctly match the selections that the voter made.

Some of these can be established just by correct implementation of the VM (for example, deterministic behaviour); others will also require correctness of the ballot definition. The security goal is that it must be possible (and preferably easy) for reviewers to verify to their satisfaction that the system guarantees all of these correctness properties, without relying on faith in the honesty or competence of the system's developers.

### 2.2 Accessibility

Initially, this work was aimed at creating a special "accessible version" of the system just for blind users, with a keypad for input, an audio-only interface, and no visual display. Before long, however, it became apparent that a *universal design* approach would be more fruitful.

Universal design [14] is the practice of designing artifacts that are flexible enough to support a wide range of users with and without disabilities, instead of separate artifacts or assistive devices for specific disabilities. A unified solution avoids stigmatizing people with disabilities, and the increased flexibility often yields benefits for all users. In this case, the unified solution is a single user interface with *synchronized audio and video*, rather than a visual interface for sighted voters and a separate audio-only interface for blind voters. The same information is presented concurrently in audio and video; user input always yields both audio and visual feedback.

The present work takes a universal design approach to input devices as well as output devices. The software design presented here is intended to support voting hardware with both a touchscreen for input and an alternate device that can transmit keypresses. The alternate device could be a regular keyboard, a numeric keypad, a set of hardware buttons designed for voting, a sip-and-puff device, or other accessible input device. The voter can decide whether to use the touchscreen or the alternate input device, and can mix them freely.

Noel Runyan, an expert on accessible technologies, recommended synchronized audio and video in discussions with the author during the early stages of this work. His recent report on voting interfaces [12] also makes this recommendation. Although not all of the electronic voting machines currently in use support synchronized audio and video, such a requirement is present both in the 2005 Voluntary Voting System Guidelines (VVSG) [17] (item 3.2.2.1f) and in a recent draft of the next generation of these guidelines [18] (item 3.3.2f).

A system with multimodal input and output is helpful not only for blind voters but also voters with low vision, voters who are illiterate, voters with cognitive disabilities, and voters with physical impairments that make it hard to use a touchscreen, as well as voters with multiple disabilities. Voters without disabilities could also benefit from audio confirmation of their choices [13].

## 2.3 Functionality

Voting systems should be highly usable by voters of all kinds, and their usability should be evaluated and improved through user testing. However, user testing of specific ballot designs is outside the scope of the present work; the aim here is to design not a particular ballot, or even a particular style of ballot, but a ballot definition language — one flexible enough that usability and accessibility experts can use it to create better and better ballots as our understanding of voting human factors improves. The PRUI paradigm opens up the process so ballot design can be done by expert ballot designers, not just voting system programmers.

If the ballot definition language is rich enough to replicate what existing voting machines do, then the resulting voting system will be capable of being at least as usable as today's voting systems. We can be assured of not having lost ground in usability, while throwing open the door to future ballot designs with better usability. Thus, the goals for the new ballot definition language are described in terms of standards of functionality:

- It should be possible, with an appropriate ballot definition and corresponding hardware, to produce a similar or better user experience compared to existing electronic voting systems, including those that support audio or synchronized audio and video.
- It should be possible to define a reasonably usable synchronized audio and video interface corresponding to a real ballot.
- It should be possible to create a single ballot definition that makes sense for a voter who can only hear the audio and also makes sense for a voter who can only see the visual display.
- It should be possible to implement most of the voting features needed for real elections, such as multiple-selection contests, write-ins, straight-party voting, eligibility for contests dependent on selections in other contests, restrictions on cross-endorsed candidates, and ranked voting.

## 3 Guiding principles

When one designs a programming language to anticipate and support many kinds of functionality, such as this ballot definition language, it is easy to get lost in a thicket of decisions and possibilities. The design experience led to the discovery of some guiding principles that helped to keep decisions well grounded. These principles would probably also be useful when taking the PRUI approach to high assurance in other domains as well as voting. The next few sections outline these principles, in order of decreasing priority.

## 3.1 Work from a concrete use case

It was helpful to examine a specific paper ballot (in this case, a sample ballot from the November 2006 election used in Contra Costa County, ballot style 167) and consider what would constitute an acceptable corresponding electronic ballot. Any faithful translation of this ballot into electronic form needs to present all of the information on the paper ballot, enable a voter to navigate through the ballot, keep the voter oriented as to their position in the ballot, allow access to all available options, and keep the voter aware of the current state of their selections. The electronic ballot must achieve all of these things for voters using only the visual display as well as voters using only the audio.

The paper ballot turned out to be invaluable for driving the design process. It was often a good idea to refer back to the paper ballot to work out exactly what should appear on the screen, what audio should be played, and the appropriate responses to all possible user inputs. The exercise of creating a specific ballot definition file revealed which features had to be supported by the ballot definition language and when it was necessary to add more capabilities to the VM.

## 3.2 Minimize VM complexity

The key goal of this work is to facilitate the review of the software that has to be verified — in this case, the VM. In general, the smaller and simpler the VM, the easier it is to verify. When faced with a design decision, it was helpful to keep returning to this goal and choose whichever option yielded a smaller or simpler VM. This principle was secondary only to including the necessary functionality to implement a real ballot, as described in the preceding section.

One consequence of this principle is that it is more important to avoid redundancy in the VM code than to avoid redundancy in the ballot data. For example, although the ballot definition file is certain to contain images that are highly compressible, they are not compressed, because that would require additional decompression code in the VM. Security reviews are expensive, but storage is cheap.

## 3.3 Maximize UI design flexibility

Other things being equal, it is better for the ballot definition language to allow a wider range of user interfaces to be specified. Giving more expressive power to the ballot definition makes the VM less likely to have to change to support new user interface designs. Since each change invalidates previous software reviews, future-proofing the VM yields real security benefits. Thus, when considering design options that do not significantly differ in the complexity of the VM or in the ability of the VM to enforce correctness constraints, the preferred option is the one that leads to a larger space of possible user interfaces.

One effective way to make the ballot definition language more expressive is to embrace orthogonality in language primitives. Replacing specialized high-level constructs with a combination of more general-purpose primitives can be doubly beneficial: the increased generality enables more possibilities to be expressed, while the increased uniformity makes the implementation in the VM more concise. For example, the new ballot definition language has no special cases to distinguish, say, review screens or write-in screens from other kinds of screens; all of these are just pages, and information can be freely arranged on each page.

The tradeoff is that using lower-level constructs sometimes makes the ballot definition more tedious to review. Switching to more general, lower-level constructs tends to be advantageous if it gives the UI designer more flexibility without creating new ways of violating correctness, and if the additional tediousness of reviewing ballot definitions can be mitigated by automated tools for reviewers.

## 3.4 Maximize UI review efficiency

In the PRUI paradigm, assurance is obtained through human review of the user interface specification (which, in this application, is the ballot definition). No design can eliminate the necessity of human involvement in evaluating the truthfulness of the user interface — whether a visual display or a spoken message is misleading is a judgement that can only be made by a human reviewer.

However, design choices can affect the level of confidence with which a human reviewer's observations can be generalized across all of the situations a user might encounter in using the voting interface. A well-designed ballot definition language can give human reviewers the leverage to draw broad conclusions from manageable amounts of review and testing.

In any system with even a modest number of variables, the number of states that the system can be in is probably so large that a human reviewer cannot observe the user interface in every possible state. But the ballot definition language can defend the human reviewer from this combinatorial explosion of states. The language can facilitate the creation of ballot definitions for which observing a limited number of states (for example, walking through the ballot making selections as in typical pre-election testing) is sufficient to extrapolate the UI presentation of all the states the system could come to be in.

For example, candidate's names are spoken in the audio interface in several contexts. When the voter selects Candidate X, there should be an audio confirmation message such as "Candidate X has been selected." When the voter is reviewing selections, the voter should hear a message such as "For President, your current selection is Candidate X."

Suppose that these two messages were each independently recorded as a single sound clip. In order to verify the correctness of the audio, a human reviewer would have to listen to each pair of messages to ensure that the candidate sounds the same in each pair — it would not do for the selection message to say "Candidate X" but for the review message to say "Candidate Y." In such a scheme, the number of messages to review would be roughly the number of candidates *times* the number of contexts in which they appear.

The reviewer's work can be made substantially easier by breaking up the messages into parts. The candidate's name can be recorded and stored once, then used for all the messages that have to do with that candidate. The remaining part (in our example, "has been selected") can be recorded once and used for all the selection messages across all candidates. The consistent reuse of audio clips can be checked mechanically, leaving the human reviewer with fewer audio clips to review (roughly the number of candidates *plus* the number of contexts).

## 4 Evolution of the design

The following sections describe the steps in thinking and the changes that were made to get from the original design [20] (referred to hereafter as the "touchscreen design" since it only supported a touchscreen for both input and output) to the new design (referred to hereafter as the "multimodal design" since it supports multiple input modes and multiple output modes). Many of these changes were driven by accessibility concerns, but some were motivated by other ballot features.

The ballot definition format had to become substantially more complex to support synchronized audio and video. Figure 1 compares the format of the ballot definition in the touchscreen design to the new format in the multimodal design. Only the main part of the ballot definition, called the *ballot model*, is shown. Precise specifications of the old format [20] and the new format [19] are available in other reports, so this paper does not go into every detail of these formats.

In the terminology used here, a **contest** is a race or a referendum put to the voters and an **option** is one of the choices available in a contest. The options in a race are candidates, whereas the options in a referendum are typically "yes" and "no." During voting, the **selection state** is the voter's current set of selections in all the contests. A contest is said to be **empty** if none of its options are selected, and **full** if the maximum allowed number of selections is selected. The **capacity** of a contest is its maximum allowed number of selections. To **undervote** in a contest is to leave the contest less than full; to **overvote** in a contest is to exceed its capacity.

### 4.1 Starting point

In the touchscreen design, the ballot definition specified a state machine where each state corresponded to a *page* with a general appearance and a visual layout. On each page, particular rectangular regions of the screen were designated for displaying options, indicating whether options were selected, and reviewing which options were currently selected. There was no restriction on how the options in a contest could be arranged; they could all be on one page or spread over many pages. Screen touches triggered transitions between pages or caused options to be selected or deselected.

Write-ins were handled as a special case in the touchscreen design. Individual options could be designated as write-ins; touching them took the user to a special type of page just for entering write-in text, with screen regions designated for special actions like appending or erasing a letter or accepting or cancelling the write-in.

The goal was a system with a superset of this functionality, so the design process sought ways to extend this touchscreen design into a multimodal design.

### 4.2 User focus

With respect to voting user interfaces, the visual channel has two advantages over audio. First, it can convey textual information at a higher bandwidth: for most people, reading a printed list of candidates' names is faster than listening to them spoken aloud. Second, a visual image can convey more information at once without an explicit navigation mechanism: although a page full of text probably exceeds what a person can hold in working memory, one can easily select and gather information of interest by looking at different parts of the page without needing to explicitly interact with the page.

A consequence of both of these properties is that audio-only voting interfaces require smaller units of navigation than video-only voting interfaces. Whereas an entire page can be visually "current" to the voter, only a few words can be aurally "current" at any given moment. For example, a visual interface can present an entire list of candidates at once but an audio interface must present the candidates one at a time. Therefore, a multimodal interface should support the notion of the user's focus at two different levels of hierarchy.

The new design adds *states* within pages to represent the second level of focus in the ballot definition. Because audio navigation units are finer-grained, audio information is primarily specified at the state level, whereas visual information is primarily specified at the page level. All the states that belong to the same page share the same overall appearance and layout, though they can vary in appearance. Behaviours in response to user input can be specified at either level; at the state level they apply to a single state; at the page level they apply to all the states in the page.

For example, in a typical ballot layout, a single page presents a list of candidates, and each state within that page highlights one of the candidates. The user presses a button to step through the candidates one at a time. In

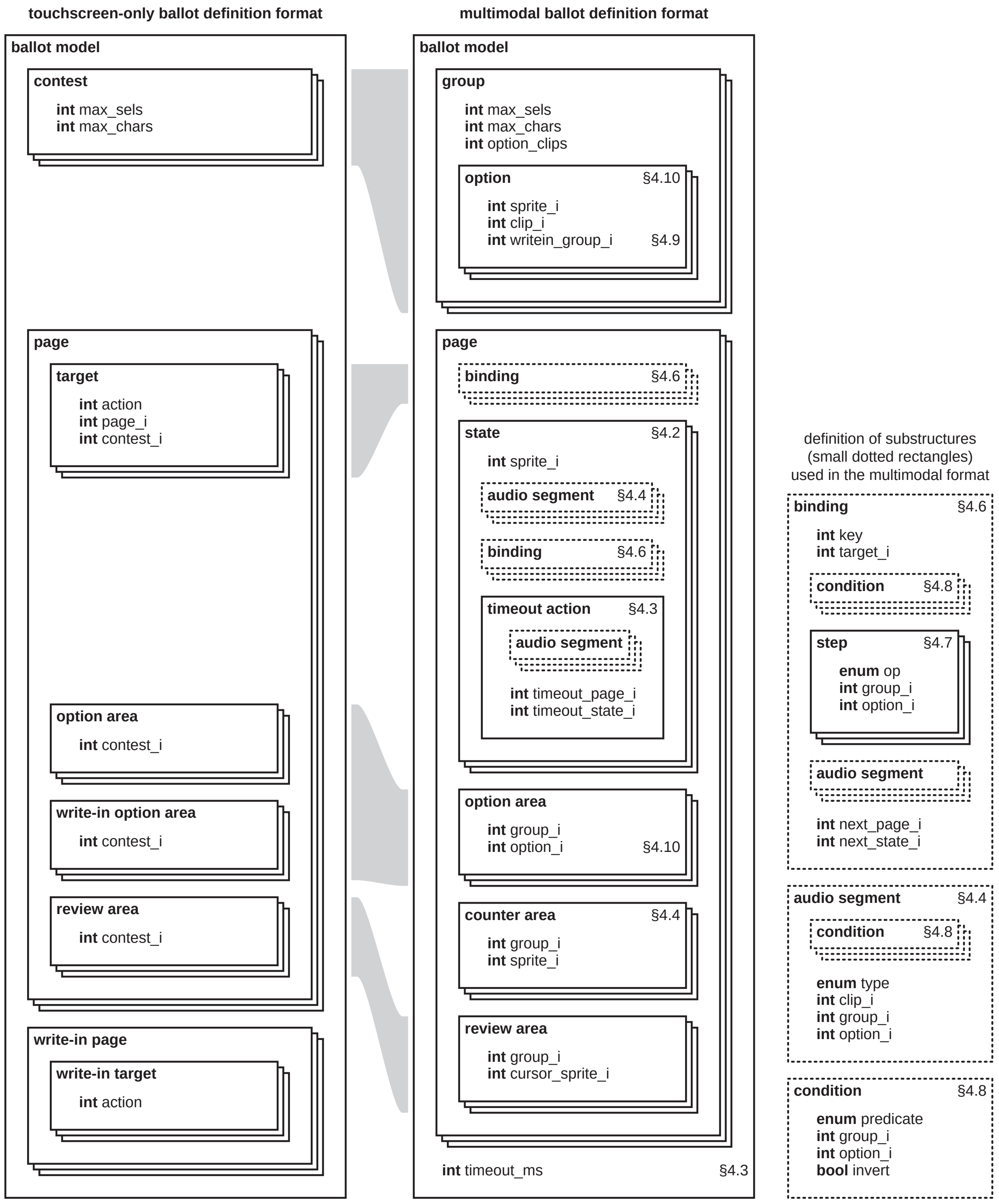

Figure 1: Side-by-side comparison of the old (touchscreen-only) and new (multimodal) ballot definition formats. Stacked rectangles represent arrays. The data structures shown in dotted rectangles are expanded on the right. Fields whose names end with _i contain array indices; in particular, sprite_i and clip_i point into arrays of images and audio clips in the video and audio sections of the ballot definition, which are not shown here. Most of the changes and additions in the new format are described in Section 4, and the relevant subsections are marked on the figure.

the state when a particular candidate becomes the focus, the audio for the candidate's name is played and the candidate's name is highlighted in the list on the screen. Selecting the currently highlighted candidate is a state-level behaviour, since the selection operation is different in each state, whereas moving on to the next contest is a page-level behaviour.

## 4.3 Staying oriented

Visual information can be presented passively, whereas presenting audio information requires continuous activity. Even an inert display can convey visual information, whereas silence conveys no audio information at all.

If a user is distracted while viewing static visual information, then getting reoriented is just a matter of looking over the information again. But if a user is distracted while listening to audio, then getting reoriented requires that the computer actively replay the audio. Therefore, an audio interface needs fallback mechanisms to trigger reorientation. The ballot definition needs to be able to specify a "where am I?" button that the user can press to recover context, as well as a way of triggering reorienting information after a period of inactivity, if the user is lost and doesn't know what button to press for help.

The former need, combined with the desire for other ballot features such as straight-party voting, led to a generalization in the way the ballot definition specifies behaviours in response to user input (see Section 4.7).

The latter need motivated the addition of a timeout parameter to the ballot definition, as well as a timeout audio sequence and an optional timeout transition for each state. When there has been no audio playing and no user input for the timeout period, the timeout audio sequence is automatically played and the timeout transition takes place, if any.

The draft human factors section for the next version of the VVSG [18] includes a requirement (item 3.2.5.1e) for a "defined and documented inactivity time" after which the system gives a warning. This new timeout functionality satisfies that requirement.

## 4.4 Conveying the selection state

The voting system is like a state machine (at any given moment, a particular "state" within a particular page is the current state), but it also keeps track of the set of current selections in each contest, which we'll call the *selection state*. (The terms "current state" and "selection state" are similar but refer to two distinct things; the voting system has only these two pieces of system state.)

In the touchscreen-only design, the selection state was exposed visually in two ways: an *option area* displayed a specified option, which would have one appearance if it was selected and another if it was not; and a *review area* displayed whatever options were selected in a specified contest.

To provide similar representation of state in audio, the multimodal design adds audio sequences consisting of one or more *audio segments*, where each segment can be constant or variable. A *constant segment* always plays the same audio clip independent of the selection state; a *variable segment* selects an audio clip to play as a function of the current selection state. Constant and variable segments are concatenated together to give the effect of filling in blanks in spoken prose, yielding a verbal description of the selection state. Variable segments can play the name of a specified option, analogous to an option area; or they can play the list of selected options in a specified contest, analogous to a review area.

For example, to tell the voter which candidates are selected for city council, an audio sequence might consist of two segments: first a constant segment that says "Your selections for city council are", then a variable segment that lists the voter's selections in the city council contest.

A constant segment is often insufficient to produce a grammatically natural description. If there is only one selection, the sentence should begin "Your selection for city council is", and if there are no selections, the audio description should say something like "You have no selections for city council." So, a third type of variable segment was added, which chooses an audio clip to play based on how many options are selected in a specified contest. By supplying an appropriate array of audio clips, a ballot definition can also use this type of variable segment to tell the voter whether they have undervoted in a particular contest.

The new design also adds a new type of variable display area, a *counter area*, analogous to this new type of variable audio segment. A counter area selects the image to display from an array of images, based on the number of options selected in a specified contest.

## 4.5 Providing feedback on actions

Not every user action succeeds. For example, the user should not be allowed to overvote. The touchscreen-only design enforced this rule, but provided no particular feedback; an attempt to select an additional candidate would simply have no effect when the contest is already full.

In a visual interface this might be considered acceptable behaviour, as the user can immediately see whether or not the attempt to select had an effect: either the candidate's name takes on a selected appearance, or it doesn't. But in an audio interface, there is no such direct feedback without an audio message describing what just happened. Therefore, to support audio-only voters, the ballot definition needs to be able to specify one audio message when selection succeeds and a different audio message when selection fails. There may be other messages as well, depending on the type of action the user was trying to perform.

In early versions of the multimodal design, each user action had several audio messages associated with it. Initially there were just two messages, one for success and one for failure; eventually this grew into five messages for each action: one to play if the action succeeded in selecting an option, one to play if the action succeeded in deselecting an option, one to play if the action failed because the contest was empty, one to play if the action failed because the contest was full, and one to play if the action had no effect. This tactic became unwieldy as the number of cases grew, and was finally discarded in favour of a generalized condition mechanism (see Section 4.8).

## 4.6 Generalized bindings

In the touchscreen design, the behaviours triggered by screen touches were specialized according to the type of the touched screen region. For example, option areas were hardcoded in the VM to react to a touch by toggling whether the associated option was selected, and write-in option areas were hardcoded to react to a touch by jumping to an associated write-in page.

This direct binding between screen regions and actions is inadequate for a multimodal design in several ways. First, direct binding doesn't make sense for input from hardware buttons: there aren't enough buttons to dedicate a button to each option. Second, the multimodal design has to allow for a "where am I?" button, as mentioned in Section 4.3, which could play many different audio messages depending on the current system state. Third, text entry in an audio-only interface is a nontrivial design problem. The touchscreen design could afford to hardcode text entry behaviour in the obvious way — a keyboard made of onscreen buttons, where touching each button types a letter. But there is no single obvious way to enter text in an audio-only interface. The text entry method is likely to vary widely depending on the hardware buttons available, so it should be left up to the ballot definition to specify.

For all these reasons, the multimodal design increases the flexibility of input handling by adding a layer of indirection: a list of bindings between input events and the actions they trigger.

## 4.7 Generalized actions

Introducing bindings meant there had to be a new data structure to represent the action triggered by an input event. This data structure initially specified a selection operation (select, toggle, or deselect) to perform on a single specific option. It was extended once with a list of contests to clear (to support automatically deselecting the current selection when making a new selection), and then extended again with a list of options to select in each contest (to support straight-party voting).

This data structure was later unified and simplified into a single list of *steps*, where each step performs a selection operation (select an option, deselect an option, remove the last selected option from a contest, or clear a contest). This change yielded a VM with less code but more flexible selection functionality. In the current design, there is no separate data structure for actions; the list of steps is embedded in the data structure for a binding.

## 4.8 Generalized conditions

The ballot definition needed a way not only to play different audio messages depending on the outcome of an action, as mentioned previously in Section 4.5, but also to perform different actions depending on the selection state. For example, consider what the voting system should do when the user touches an option. If the option is already selected, then one possible effect would be to deselect the option. If the option is not selected, and its contest is not full, then the option should become selected. And if the option is not selected but its contest is full, then the selection should not change. Each of these three cases should also have its own corresponding audio message. This capability was added by attaching to each binding a list of conditions concerning the selection state. Each condition can check whether a particular option is selected, a particular contest is full, or a particular contest is empty. The binding is triggerable only if all of its conditions are satisfied.

Conditions also turned out to be useful for constructing variable audio sequences. Prior to the addition of conditions, there were nine types of variable audio segments. Then a list of conditions was attached to each segment; each segment is played or skipped depending on whether all of its conditions were satisfied. Reusing conditions in this way increases the flexibility of audio feedback while simplifying the implementation: with conditions added, only three types of variable audio segments are necessary.

## 4.9 Groups: a dual-use data structure

Substantial economy in code complexity was achieved by using a single data structure, the *group*, for two purposes. In the multimodal design, a *group* is a container of selectable options; it can represent a contest (with options such as candidates) or a write-in entry field (where the options are the individual characters that can appear in the entry field). It made sense to combine these into a single data structure because of their common functionality:

- In both cases, the current selection for a group is a list of options (even though a contest selection has set-like semantics and a write-in selection has ordered sequence semantics).
- In both cases, user actions add and remove options to and from the selection (e. g. selecting candidates in a contest or typing letters into a write-in field).
- Reviewing the state of a group consists of displaying the list of selections in order (pasting the candidate images or the letter images into a sequence of equal-sized spaces on the screen) or playing back audio for the list of selections in order (reading off the list of selected candidates or speaking the letters in a write-in field one by one).

## 4.10 Candidate rotation support

In the touchscreen-only design, every option area was assumed to represent a distinct option. Thus, each option area only had to indicate which contest it belonged to; the number of options in a contest could be determined by scanning all the pages of the ballot definition and counting the option areas associated with that contest.

In the multimodal design, information about each option (such as its associated image and audio clip) is kept in an *option* structure under the option's group. The option areas refer to these option structures. Bindings that select options, audio segments that play option names, and conditions that examine options can either refer to options directly or refer to option areas, which themselves refer to options. This extra layer of indirection yields two kinds of flexibility:

- The same option can be displayed in more than one place on the ballot.
- Options can be rearranged by rearranging the references from option areas to options.

Without the extra layer of indirection, candidate rotation would be difficult to automate reliably because there would be no distinction between a reference to an option area and a reference to an option. This distinction is important because indirect references to options via option areas should change when options are shuffled, whereas direct references to options should not change when options are shuffled. When candidates are rotated, their screen position and order of audio presentation should change, but the set of candidates belonging to a party for a straight-party vote should not change.

This design feature makes it easy to rotate candidates by a simple manipulation of the ballot file. Rearranging the references from option areas to options does not change the option number assigned to each candidate. Thus, candidate rotation has no effect on the way voter selections are recorded, which helps to avoid the possibility of confusion in interpreting recorded votes.

One could produce several rotated variants of a ballot before the election and publish them all; it is straightforward to verify that two ballot definition files represent the same ballot except for reordering of the candidates. Alternatively, the voting machine could even perform candidate rotation on the fly for each voter, though the prototype implementation does not do this.

## 4.11 Large type and high contrast

A large-type mode and a high-contrast mode can be helpful for users with visual disabilities. Both the 2005 VVSG [17] (items 3.3.2.1b and c) and the draft new guidelines [18] (items 3.2.4e and j) require electronic voting displays to be capable of showing all information in at least two type sizes, 3.0–4.0 mm and 6.3–9.0 mm, and to have a high-contrast mode with a contrast ratio of at least 6:1 (on current voting machines this usually means a black-and-white mode).

The touchscreen-only design can already accommodate these requirements by providing multiple prerendered versions of the ballot in a single ballot definition file, with an initial page on which the voter can select the desired presentation mode.

This does have the consequence that contests and options are duplicated in each version of the ballot. In terms of the ballot definition data structures, the large-type contest and the normal-type contest for each office would be distinct contests with distinct options. This creates the possibility that the electronic record of a vote would reveal whether the voter selected a large-type candidate or a normal-type candidate. It is possible to avoid revealing this distinction with a workaround that exploits generalized actions: user selection of a candidate in any display mode automatically selects all the corresponding variants in the other display modes (e. g. touching the button for Jane Smith in normal print also selects Jane Smith in large print, Jane Smith in high contrast, etc.). Future versions of the ballot definition format may add features to obviate this workaround.

## 5 Design

This section describes the current design of the multimodal ballot definition format and the VM software that interprets it. (Readers not interested in the details can safely skip this section on a first reading.)

Pvote, the present implementation of this design, is intended for voting machines that are electronic ballot printers; thus, both the ballot definition and the VM software contain a component specifically to support ballot printing. An implementation targeted for other types of voting machines would substitute a different component for recording the cast votes, such as the tamper-evident direct recording mechanism of the prior PRUI voting prototype [20].

## 5.1 Ballot definition format

This section gives an overview of the new ballot definition format. For a detailed specification of the format, see the Pvote Assurance Document [19].

Just as in the touchscreen-only design, the ballot definition describes a state machine. Each state transition is triggered by a user action or by an idle timeout. Executing a transition can cause options to be selected or deselected. Audio feedback can be associated with states and with transitions between states.

The ballot definition contains three main sections:

- **Ballot model**: structure of the ballot and interaction flow of the user interface.
- **Audio data**: sound clips to play over the headphones.
- **Video data**: images to display on the screen, the locations at which to display them, and locations of touch-sensitive screen regions.

These three sections are separated so that each one can be supplied to a distinct module of the VM with distinct responsibilities. In addition, they can be separately updated — for example, one can translate the audio interface into a different language by recording audio clips for a new audio data section while leaving the other sections unchanged.

In Pvote, which is written specifically for a text-based electronic ballot printer, the ballot definition also includes a fourth section, the **text data**, which contains textual descriptions of the contests and candidates for the printer to print.

### 5.1.1 Audio data

The audio data section specifies the sample rate at which all audio is to be played and provides an array of sound clips. Other parts of the ballot definition refer to these clips by supplying indices into this array. The audio clips are uncompressed and monophonic, and each sample is a 16-bit signed integer. The clips can contain recordings of actual speech or of prerendered synthesized speech.

### 5.1.2 Video data

The video data section specifies the resolution of the video screen and includes an array of *layouts* and an array of *sprites*. A **sprite** is an image, smaller than the size of the entire screen, that will be pasted on the screen somewhere. A **layout** consists of a full-screen image, an array of *targets*, and an array of *slots*. A **target** is a rectangular region of the screen where a touch will have an effect; a **slot** is a rectangular region where a sprite can be pasted. Image data is stored uncompressed, with 3 bytes per pixel (red, green, and blue colour values).

### 5.1.3 Ballot model

The ballot model is the main specification of the state machine. It contains an array of *groups* and an array of *pages*. It also specifies an idle timeout in milliseconds.

**Groups and options**

A **group** is a set of choices from which the voter makes selections. There are two kinds of groups: *contest groups* and *write-in groups*. A **contest group** represents a race in which the options are candidates or a referendum question with options such as "yes" and "no". A **write-in group** represents the text entered in a write-in area within a contest, in which the options are the characters used to spell out the name of the write-in candidate. In the array of options within each group, images and sound clips are specified to represent each option by providing indices into the arrays of audio clips and sprites. Within a contest group, an option can also specify that it is a write-in option and identify the write-in group containing its write-in text.

Each group specifies its capacity (the maximum number of selections allowed in the group); for contest groups this prevents overvotes, and for write-in groups this limits the length of the entered text. All the write-in options within a contest must have the same maximum length for text entry.

**Pages and states**

The **page** is the basic unit of visual presentation; within each page is an array of *states*. The pages correspond, one-to-one, to the layouts in the video data. At any given moment, there is a current page and a current state. The user interface always begins on page 0 in state 0; when the VM executes a transition to the last page in the array of pages, the ballot is printed or cast with the voter's current selections. In addition to the array of states, each page contains arrays of *option areas*, *counter areas*, *review areas*, and *bindings*.

The **states** in a page are states in the state machine of the user interface. Each state specifies a sprite to be pasted over the main page image while the state is current. (For example, a page could show a list of several options, and the states within that page could display a focus highlight that moves from option to option. Each state would paste a focus highlight for its option over the page image.) Each state also has an array of *audio segments* to be played upon entering the state, and an array of its own bindings.

A state can also specify audio segments to be played upon a timeout and/or an automatic transition to another state upon a timeout. A timeout occurs when the audio has stopped playing and there has been no user activity for the timeout duration specified in the ballot model.

An **option area** is a screen region where an option will be displayed. Its fields identify the option that will appear there.

A **counter area** is a screen region that will indicate the number of options currently selected in a contest; this enables the interface to provide feedback on undervoting. A counter area is associated with a group and points to an array of sprites. The number of currently selected options in the group is used as an index to select a sprite from the array to display.

A **review area** is a screen region where currently selected options will be listed; it has a field to indicate the group whose selections will be shown. The review area must provide enough room for up to $j$ options to be displayed, where $j$ is the capacity of the group. A review area can also specify a "cursor sprite" to be displayed in the space for the next option when the group is not full. This allows a review area for a write-in group to serve as a text entry area, in which a cursor appears in the space where the next character will be added.

The screen locations for pasting all these sprites (overlays for states, options for option areas and review areas, and sprites for counter areas) are not given in the ballot model; they are specified in the array of *slots* in the page's corresponding layout. Each state, option area, and counter area uses one slot. Each review area uses $j \times (1 + k)$ slots, where $j$ is the capacity of the group and $k$ is the capacity of write-ins for options in the group. (A write-in group cannot itself contain write-in options; thus, for a review area for a write-in group, $k$ is zero.) Each block of $1 + k$ slots is used to display a selected option: the option's sprite goes in the first slot, and if the option is a write-in, the characters of the entered text go in the remaining $k$ slots, which are typically positioned within the first slot. If there are $i$ currently selected options in the group, option sprites appear in the first $i$ of the $j$ blocks. If there is a cursor sprite, it is pasted into the first slot of block $i + 1$ when the group is not full.

## Bindings

The lists of bindings in pages and states specify behaviour in response to user input. Each binding consists of three parts: stimulus, conditions, and response.

There are two kinds of stimuli: a keypress, which is represented as an integer key code, and a screen touch, which is translated into a target index by looking up the screen coordinates of the touch point in the layout's list of targets. A binding can specify either a key code or a target index or both.

Each binding can have a list of associated conditions; the binding applies only if all the conditions are satisfied. A condition can test whether a particular group is empty or full or whether a particular option is selected.

The response consists of three parts, all optional: selection operations, audio feedback, and navigation. The selection operations are specified as a series of *steps*, where a step selects or deselects an option, appends a character to a write-in, deletes the last character, or clears a group. The audio feedback is given as an array of audio segments to play. Navigation is specified as the index of a new page and state.

Bindings for the current state take precedence over bindings for the current page. When the user provides a stimulus, at most one binding is invoked: the bindings for the state and then the page are scanned in order, and the response is carried out for the first binding that matches the stimulus and has all its conditions satisfied.

## Audio segments

Audio feedback is specified as a list of segments. A segment can play a fixed clip, the clip associated with a specified option, all the clips associated with the options that are selected in a specified group, or a clip chosen based on the number of options that are selected in a specified group. When a clip associated with an option is played, if the option is a write-in option, the clip for each character in the contents of the write-in field is also played. More than one clip can be associated with an option (for example, each candidate could have a short description and a long description).

At any given moment, at most one clip can be playing at a time; there is a play queue for clips waiting to be played next. Whenever a clip finishes playing, the next clip from the queue immediately begins to play, until the queue is empty. Invoking a binding always interrupts any currently playing clip and clears the play queue. The audio segments for the binding, if any, are queued first; if a state transition occurs, the audio segments for the newly entered state are queued next.

Each segment has a list of conditions (the same as in a binding) that must all be satisfied in order for the segment to be queued; otherwise, the segment is skipped. The conditions are evaluated when the segment list is being queued (i. e. immediately after carrying out the selection steps of a binding, immediately after entering a new state, or when a timeout occurs).

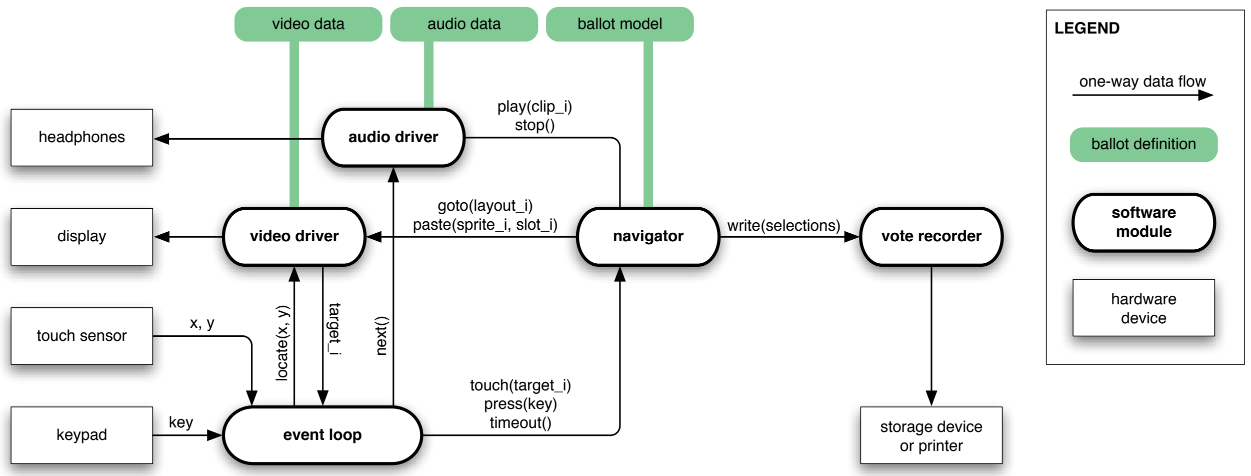

Figure 2: Block diagram of the virtual machine. The five software modules in bold generate and run the user interface. The arguments clip_i, layout_i, sprite_i, target_i, key, x, and y are integers; selections is an array of arrays of integers.

## 5.2 Virtual Machine

The VM is composed of five software modules: the *navigator*, the *audio driver*, the *video driver*, the *event loop*, and the *vote recorder* (Figure 2). Two additional components not visible in Figure 2 are the *ballot loader*, which deserializes the ballot definition into memory, and the *ballot verifier*, which checks the ballot definition. The loader and verifier complete their work before the voting session begins (i. e. before any interaction with the voter). The verifier is responsible for ensuring that the ballot definition is sufficiently well-formed that the VM will not crash or become unresponsive during the voting session.

Each component has limited responsibilities, and there are limited data flows between components.

The **event loop** maintains no state and handles all incoming events, which are of four types:

- Keypresses: Upon receiving a keypress event, the event loop sends a **press** message to the navigator.
- Screen touches: Upon receiving a touch event, the event loop sends a **locate** message to the video driver to translate the touch coordinates into a target index, then passes this target index to the navigator in a **touch** message.
- Audio notifications: Upon receiving notification that a sound clip has finished playing, the event loop sends a **next** message to the audio driver.
- Timer notifications: Upon receiving notification that the timer has expired, if no sound clip is currently playing, the event loop sends a **timeout** message to the navigator to indicate that the ballot's specified timeout has passed with no activity.

Whenever it receives any event, the event loop reschedules a timer notification event according to the timeout duration in the ballot definition.

The **navigator** keeps track of the current page and state and the current selections in each group, and has no other state. The navigator responds to three messages:

- **touch**(target_i): Find the first operative binding for the current state or page that matches the given target, and invoke it.
- **press**(key): Find the first operative binding for the current state or page that matches the given keypress, and invoke it.
- **timeout**(): Add the current state's timeout audio segments to the play queue, and follow the current state's timeout transition, if one is specified.

The navigator sends five messages to other modules:

- **goto**(layout_i) is sent to the video driver upon transition to a page. The layout index is the same as the page index (the array of layouts in the video data parallels the array of pages in the ballot model).
- **paste**(sprite_i, slot_i) is sent to the video driver to paste sprites into slots as necessary for states, option areas, counter areas, and review areas. sprite_i is the index of a sprite in the array of sprites in the video data; slot_i is the index of a slot in the current layout.
- **play**(clip_i) is sent to the audio driver to queue a clip to be played on the headphones. clip_i is the index of an audio clip in the array of clips in the audio data.
- **stop**() is sent to the audio driver to stop the currently playing clip.
- **write**(selections) is sent to the vote recorder to record the user's selections. selections is an array of

arrays of integers: one array for each group, listing the indices of the selected options in that group.

The **audio driver** maintains a queue of audio clips to be played, and has no other state. It responds to three messages:

- **play**(clip_i): If nothing is currently playing, immediately begin playing the specified clip; otherwise queue the specified clip to be played.
- **next**(): If there are any clips waiting in the queue, start playing the next one.
- **stop**(): Stop whatever is currently playing and clear the queue.

The audio driver sends no messages to other modules; however, whenever it starts playing a clip, it schedules an audio notification event for the event loop to receive when the clip finishes playing. The audio driver also exposes a flag that the event loop can read to determine whether a sound clip is currently being played.

The **video driver** maintains just one piece of state, the index of the current layout. It responds to three messages:

- **goto**(layout_i): Copy the full-screen image for the given layout into the video display's frame buffer and remember this as the current layout.
- **paste**(sprite_i, slot_i): Copy the given sprite into the frame buffer at the position specified by the given slot in the current layout.
- **locate**(x, y): Find and return the index of the first target that contains the given point in the current layout's list of targets, or an error code if the point does not fall within any target.

The video driver sends no messages to other modules.

The **vote recorder** maintains no state and responds to only one message:

- **write**(selections): Record the voter's selections.

The vote recorder records votes as appropriate for the type of voting machine (e. g. printing a ballot, marking a ballot, or directly recording votes in electronic storage).

## 6 Implementation

Pvote is a Python [10] implementation of the design described here. Pvote can run on Linux, MacOS, and Windows. Graphics and sound are handled by Pygame [9], an open-source multimedia library for Python. Touchscreen input is simulated using the mouse, and hardware button input is simulated using the keyboard.

Pvote is written to be deployed as an electronic ballot printer. In Pvote, the vote recorder prints out a textual description of the voter's selections. Each time Pvote runs, it prints at most one ballot (to standard output) and then enters a terminal state. The source code for Pvote and a sample ballot definition file are available online at `http://pvote.org/`.

### 6.1 Code Size

The entire Pvote implementation is 460 lines long, not counting comments and blank lines. The breakdown of module sizes is as follows:

| | | |
|---|---|---|
| ballot loader | 137 lines | |
| ballot verifier | 96 lines | |
| subtotal (pre-voting) | | 233 lines |
| event loop | 25 lines | |
| navigator | 120 lines | |
| audio driver | 35 lines | |
| video driver | 22 lines | |
| subtotal (voting) | | 202 lines |
| vote recorder | | 25 lines |
| total | | 460 lines |

### 6.2 Dependencies

Pvote is written in a small subset of Python 2.3 called Pthin, which is specified in the Pvote Assurance Document [19]. Pvote uses only one built-in collection type, the Python list, and only the following built-in functions:

- **open** and **read** to read the ballot definition file.
- **chr** and **ord** to convert integers to/from characters.
- **list** to convert strings to lists of characters.
- **enumerate** and **range** to iterate over lists.
- **len**, **append**, **remove**, and **pop** to manipulate lists.

The ballot loader imports the built-in SHA module to compute and verify a SHA-1 hash of the ballot definition. The audio and video driver use Pygame functions to play audio and display images. Aside from these, Pvote imports no other library modules.

### 6.3 Functionality

Pvote achieves the functionality goals set out in Section 2.3. Pvote can support a wide range of features in the voting user interface, including multimodal input and output and virtually complete flexibility in the style of audio and visual presentation. Because Pvote uses prerecorded audio and prerendered images, the ballot can be presented in any language.

With its generalized actions and conditions, Pvote offers much more flexibility in the handling of user input than its touchscreen-only predecessor. Unlike the previous prototype, Pvote can handle straight-party voting, dependencies among contests (e. g. in a recall election, voting for a replacement candidate conditional on voting "yes" for recalling the incumbent), and conditional navigation (e. g. displaying an undervote warning page when the voter has not made any selections in a contest). The ballot designer also has more freedom to define the interaction for selection and text entry.

To get a rough sense of Pvote's coverage of ballot design features, the author examined NIST's collection of sample ballots [8], consisting of 373 ballots from 40 U. S. states for elections from 1998 to 2006. The following table summarizes the ballot features found. All these features, and hence all the ballots in the collection, are supported by Pvote's ballot definition format.

| **Ballot feature** | **Ballots** |
|---|---:|
| Vote for 1 of $n$ | 373 |
| Vote for up to $k$ of $n$ ($k > 1$) | 195 |
| Vote for an image (e. g. a state flag) | 2 |
| Vote yes or no (referendum, confirmation) | 251 |
| Ranked choice (up to 3 choices) | 7 |
| Write-in candidate | 318 |
| Straight-party vote | 60 |
| Cross-endorsed candidates | 8 |
| Multi-party primary | 5 |
| Party logos | 21 |
| Chinese | 46 |
| Ilokano | 2 |
| Japanese | 1 |
| Korean | 1 |
| Spanish | 54 |
| Vietnamese | 1 |
| Multiple languages | 53 |

The longest ballot was a full-face Colorado ballot from 1998 containing 23 offices, 22 judicial confirmations, and 26 referenda. A ballot from New York allowed the most votes in a single contest (up to 11 selections).

## 6.4 Security

Pvote was the subject of a three-day software review conducted in March 2007. Five computer scientists with backgrounds in security research and/or electronic voting participated as reviewers. Four reviewers were present on the first two days, and three were present on the third day. The reviewers did not find any bugs in Pvote itself, though they did find some errors and omissions in the Pvote Assurance Document [19], which specified the design assumptions and security claims.

The reviewers all had positive things to say about their confidence in Pvote. Three reviewers had previous experience inspecting commercial voting systems. Among them, two declared greater confidence in Pvote's correctness as the UI component of a voting system than comparable parts of commercial systems; the third felt he had not spent sufficient time on the review to state a confidence level, but was convinced that Pvote's design would make it easier to argue for its correctness than for the correctness of other systems.

Pvote has probably received more scrutiny in terms of reviewer-hours per line of code than most existing voting software, so the lack of discovered bugs is cause for at least some confidence. Nonetheless, we can be certain that it did not receive enough scrutiny to establish full confidence, because the author intentionally hid three bugs in the code and the reviewers found only two of them. A forthcoming report will give further details on the review and the lessons and insights it yielded.

## 6.5 Ballot Definition File

Pvote was tested with a sample ballot definition file generated by a ballot compiler, also written in Python. The ballot compiler takes a textual description of the contests and options and produces the necessary images using the open-source ReportLab toolkit [11] for drawing, text rendering, and page layout. To construct the audio clips for the ballot definition, the compiler uses the same textual description to select fragments from a library of clips of recorded speech and concatenates the fragments together as needed. The audio clips in this sample ballot are recorded from live speech, which is usually preferred over synthesized speech.

The inclusion of screen images and audio recordings in the ballot definition yields a large file. The sample ballot contains five contests: two are single-selection races with six candidates each, one is a multiple-selection race with five candidates, and two are propositions. Each proposition has a description of about 100 words, and the audio of the entire description read aloud is included in the ballot. All of this compiles to a 69-megabyte ballot definition file, containing 17 pages at a resolution of 1024 $\times$ 768 pixels and 8 minutes of audio sampled at 22050 Hz. As a rough estimate, a ballot with 20 or 30 contests might occupy a few hundred megabytes.

File sizes this large might seem unwieldy in practice. However, ballot definition files can be compressed for transmission (bzip2 compresses this 69-megabyte ballot to 12.5 megabytes, which is better than a factor of 5), and ballot definitions can be loaded onto voting machines using inexpensive SD flash memory cards (one-gigabyte SD cards can now be purchased for about US$10).

## 7 Discussion

A theme running throughout this work is the management of complexity. Complexity is the enemy of correctness, but it cannot be completely avoided. Prerendering the user interface is a strategy for mobilizing the complexity in a voting system. Rather than embedding so much complexity in the voting machine, the designer gains the freedom to move complexity among three components: the tool that generates the ballot definition file, the ballot definition, and the VM in the voting machine. The allocation depends on design choices in the ballot definition language. For instance, in Pvote, the task of laying out text on the screen is no longer the job of the voting machine; it has moved to the ballot generator. The logic that decides when to play which audio message is no longer part of the voting machine; it has moved to the ballot definition.

It is worthwhile to ask what good this does us. Is real progress achieved by shifting complexity in this way, or are we merely playing a shell game, hiding complexity in components that we conveniently choose to ignore?

It matters where complexity resides. For example, why do software security reviews demand source code? Source code is certainly *easier* to review than an executable, but convenience alone is not a reason for confidence. If a review of the source code discovers no bugs, this does not assure the correctness of the executable unless the compiler is also correct. Figure 3 depicts the relationship between these three components.

Whenever there is a generative relationship such as this, with an input, a transform, and an output, reviewers have a choice: they can inspect the output, or they can inspect the input and the transform instead. In this example, the burden of establishing confidence in the executable is traded for the burden of establishing confidence in the source code *and* the compiler. Why is this trade considered helpful despite the compiler's massive complexity? Why trust compilers? For those who do, a typical reason would be that the compiler is general-purpose. An application-specific component with high complexity (the executable) has been traded for a component that is highly complex but general-purpose (the compiler), and a component that is application-specific but much less complex (the source code).

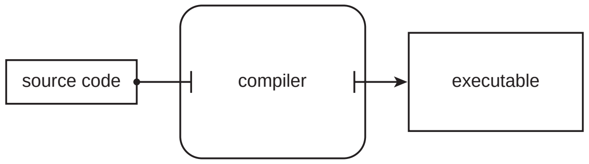

Figure 3: A compiler turns source code into an executable. Size indicates relative complexity. Rounded boxes are general-purpose components; sharp-cornered boxes are application-specific components.

As a framework for an assurance argument, it is interesting to map out the components in a system in terms of these generative relationships. Figure 4 shows such a map for a conventional voting system and for Pvote. Each arrow in Figure 4 represents a step in a hierarchical decomposition of the system. At each such step, one can choose to make an assurance argument about the output or about both input and transform.

Here are a few dimensions along which components can be evaluated with respect to the insider threat:

1. **Dictated or freely chosen?** Are relying parties forced to use a particular implementation of the component, or do they have the freedom to choose? Shifting complexity from a dictated component to a freely chosen component reduces barriers to confidence. For example, anyone can choose or write their own tools to decompile and analyze the ballot definition. In contrast, voters cannot choose to vote on any equipment they want; they must use the equipment dictated by election administrators.
2. **Hidden or disclosed?** Components that are undisclosed or inherently undisclosable (such as live running processes) are risky because their correctness cannot be verified. Shifting complexity to a disclosed component reduces barriers to confidence.
3. **Application-specific or general-purpose?** Shifting complexity to general-purpose components sometimes reduces barriers to confidence. Undetected bugs and backdoors may be less likely if the component is mature, widely used, or well tested by others, and the testing parallels the intended use. If one uses a version of the general-purpose component that was released before the voting system was conceived, it is harder to imagine how an insider could have subverted it to meaningfully influence the outcome.

In the interest of reducing the code that runs in the voting machine, Pvote trades one language for another: much of the user interface is specified in a specialized ballot definition language instead of a general-purpose programming language like C or Python. Section 3.3 suggested that generality in the ballot definition language is advantageous as it future-proofs the language definition. How much generality is beneficial? In the extreme, one could shrink the VM to nothing at all by declaring that ballot definitions are just machine code.

Shifting complexity from one programming language to another is useful only insofar as the target language provides security-relevant restrictions on what can be

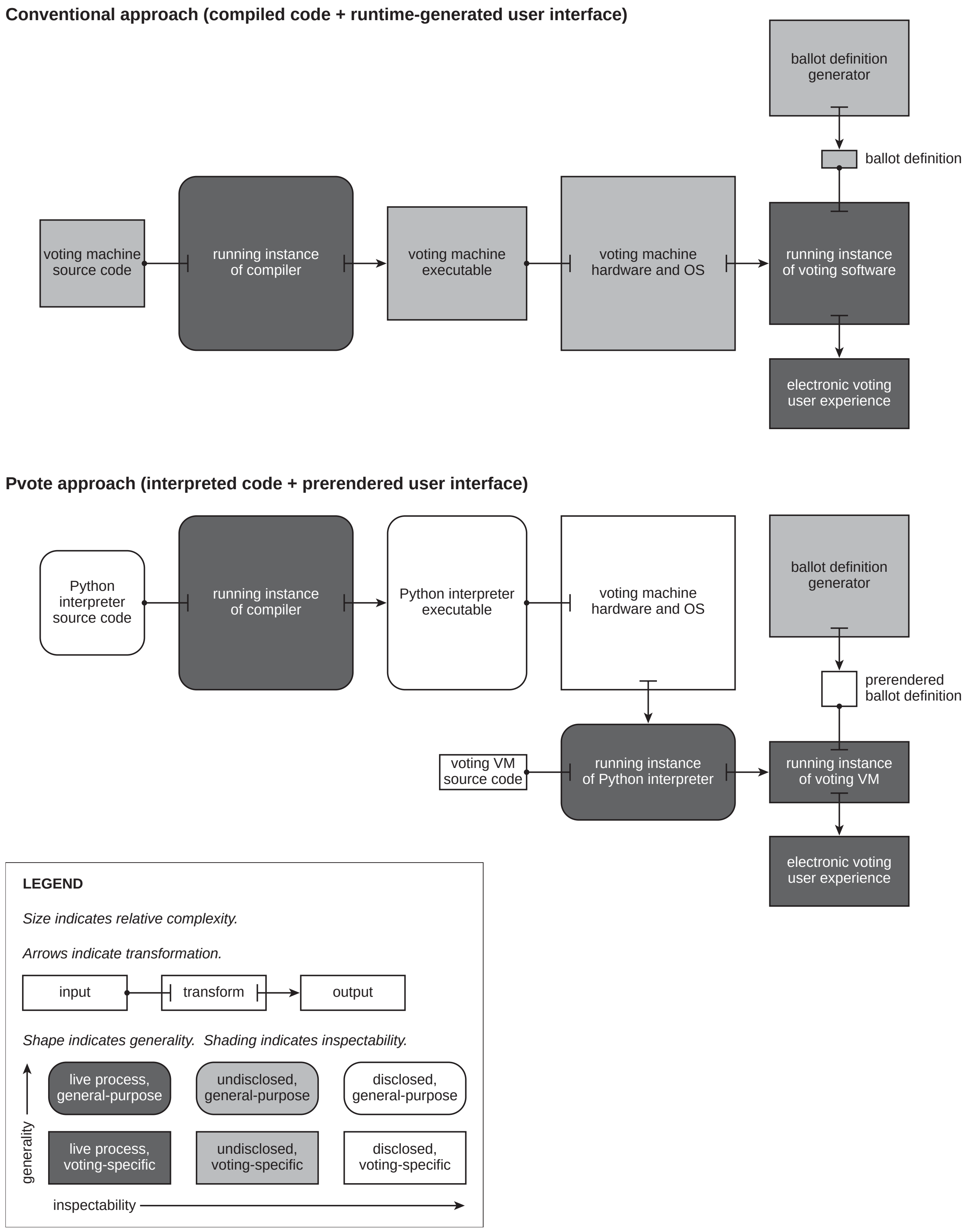
Conventional approach (compiled code + runtime-generated user interface)
ballot definition generator
ballot definition
voting machine source code
running instance of compiler
voting machine executable
voting machine hardware and OS
running instance of voting software
electronic voting user experience
Pvote approach (interpreted code + prerendered user interface)
Python interpreter source code
running instance of compiler
Python interpreter executable
voting machine hardware and OS
ballot definition generator
prerendered ballot definition
voting VM source code
running instance of Python interpreter
running instance of voting VM
electronic voting user experience
LEGEND
Size indicates relative complexity.
Arrows indicate transformation.
input
transform
output
Shape indicates generality. Shading indicates inspectability.
live process, general-purpose
undisclosed, general-purpose
disclosed, general-purpose
live process, voting-specific
undisclosed, voting-specific
disclosed, voting-specific
generality
inspectability

Figure 4: Maps of the origins of the components that determine the correctness of the user experience in electronic voting. Although relative differences in size are meant to roughly express relative differences in complexity, they are not to scale. For example, the voting machine source code in the conventional approach is over 60 times larger than the voting VM source code in Pvote, and the complexity of a C compiler is many times larger still. A notable omission from this discussion is the long chain of inputs leading to a running instance of a compiler, and its associated risks [7].

expressed. For example, the ballot definition language contains no concept of the current time and date, and in general, no way to express behaviour that will be different at testing time than on election day itself. This property is essential to the effectiveness of "logic and accuracy testing," in which pre-election test behaviour is assumed to reflect the machine's actual behaviour on election day. The PRUI approach greatly reduces the amount of code that has to be reviewed to establish this property, because non-determinism can only reside in the VM, not in the ballot definition. This experience suggests that restricted domain-specific languages and languages that support programming in restricted subsets are powerful tools for verifiable secure system design.

## 8 Conclusion

This paper has presented a design for voting machine software that supports user interfaces with synchronized audio and video, touchscreen input, and accessible device input. The software can be implemented in a very small amount of code compared to existing voting machines, while allowing a high degree of flexibility in the design of the user interface. This work validates the prerendered-UI approach by demonstrating that it can meet both accessibility and security goals.

A major area of future work consists of designing specific ballots using Pvote's ballot definition language and running user studies to measure their usability and accessibility. The PRUI paradigm offers the freedom to improve ballot designs based on such studies without having to change and recertify voting system software.

Also to be developed is a design tool for laying out ballots and producing ballot definition files.

## 9 Acknowledgements

I am grateful to David Wagner, Marti Hearst, and Scott Luebking for their guidance and editing suggestions on this paper. Scott Luebking and Noel Runyan gave helpful advice concerning the accessibility of this design. Matt Bishop, Tadayoshi Kohno, Mark Miller, Dan Sandler, and Dan Wallach generously volunteered their time to participate in the Pvote security review.

This work was funded in part by NSF CNS-0524252.

## References

[1] Alec Yasinsac, David Wagner, Matt Bishop, Ted Baker, Breno de Medeiros, Gary Tyson, Michael Shamos, Mike Burmester. Software Review and Security Analysis of the ES&S iVotronic 8.0.1.2 Voting Machine Firmware. Florida Department of State, 2007.

[2] Jon Bentley. Programming Pearls: Little Languages. *Communications of the ACM*, 29(8):711–721, Aug. 1986.

[3] David Wagner, David Jefferson, Matt Bishop, Chris Karlof, Naveen Sastry. Security Analysis of the Diebold AccuBasic Interpreter, Feb. 2006. `http://www.cs.berkeley.edu/~daw/papers/accubasic.pdf`.

[4] Harri Hursti. Diebold TSx Evaluation: Critical Security Issues with Diebold TSx, May 2006. `http://www.blackboxvoting.org/BBVtsxstudy.pdf`.

[5] Jonathan Bannet, David W. Price, Algis Rudys, Justin Singer, Dan S. Wallach. Hack-a-Vote: Security Issues with Electronic Voting Systems. *IEEE Security & Privacy*, 2(1):32–37, Jan./Feb. 2004.

[6] Doug Jones. Connecting Work on Threat Analysis to the Real World. Threat Analyses for Voting System Categories: A Workshop on Rating Voting Methods, 2006. `http://www.cs.uiowa.edu/~jones/voting/VSRW06.pdf`.

[7] Ken Thompson. Reflections on Trusting Trust. *Communications of the ACM*, 27(8):761–763, Aug. 1984.

[8] NIST and Richard G. Niemi. Sample State and Local Ballots. `http://vote.nist.gov/ballots.htm`.

[9] Pygame. `http://pygame.org/`.

[10] Python Software Foundation. Python. `http://www.python.org/`.

[11] ReportLab, Inc. ReportLab Toolkit. `http://www.reportlab.org/`.

[12] Noel H. Runyan. Improving Access to Voting: A Report on the Technology for Accessible Voting Systems. Demos and Voter Action, Feb. 2007. `http://demos.org/pubs/improving_access.pdf`.

[13] Ted Selker. Voting Technology: Election Auditing is an End-to-End Procedure. *Science*, 308(5730):1873–1874, Jun. 2005.

[14] Molly F. Story. Maximizing Usability: The Principles of Universal Design. *Assistive Technology*, 10(1):4–12, 1998.

[15] Tadayoshi Kohno, Adam Stubblefield, Aviel D. Rubin, Dan S. Wallach. Analysis of an Electronic Voting System. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2004.

[16] RABA Technologies. Trusted Agent Report: Diebold AccuVote-TS Voting System, Jan. 2004. `http://www.raba.com/press/TA_Report_AccuVote.pdf`.

[17] U. S. Election Assistance Commission. 2005 Voluntary Voting System Guidelines, Dec. 2005. `http://www.eac.gov/vvsg_intro.htm`.

[18] U. S. Election Assistance Commission. Draft Usability and Accessibility Requirements for 2007 Voluntary Voting System Guidelines. NIST, Dec. 2006. `http://vote.nist.gov/VVSG-HFP.pdf`.

[19] Ka-Ping Yee. Pvote Software Review Assurance Document. UC Berkeley EECS Technical Report 2007-40, Apr. 2007. `http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-40.html`.

[20] Ka-Ping Yee, David Wagner, Marti Hearst, and Steven Bellovin. Prerendered User Interfaces for Higher-Assurance Electronic Voting. In *Proceedings of the Electronic Voting Technology Workshop*, 2006.