Pvote Software Review Assurance Document

Ka-Ping Yee
ping@zesty.ca

March 29, 2007

Acknowledgements

David Wagner reviewed this document extensively and made many suggestions that have led to substantial improvements. David Wagner, Marti Hearst, Noel Runyan, Scott Luebking, and Steven Bellovin all participated in discussions that have influenced the design of Pvote. I am grateful to Lisa Friedman for her support during the production of this document.

Contents

Chapter 1

Scope

This document is a preparatory guide for reviewers of the Pvote software for voting machines, which is based on the prerendered user interface approach. Pvote is implemented in a subset of Python.

Pvote is not a complete voting system. It is just the component responsible for presenting the ballot to the voter and recording the voter's selections. (The EVT paper on prerendered user interfaces for voting argues that this is a crucial component to get right because the voting interactions of individual voters must be kept secret, whereas other parts of the process can be made publishable.) Voter registration, vote tallying, and administrative functions are not part of Pvote.

The following sections set out expectations for the scope of the review based on Pvote's design assumptions and design intent. However, as reviewers, if you find it necessary to look beyond the scope suggested here, you should feel free to direct the course of the review as you see fit.

1.1 Overview

Pvote is intended to be small and not changed often. The election parameters and the voting user interface are described by a **ballot definition file** that Pvote accepts as input. Pvote is flexible enough to support a wide range of election types and interface designs, just by using different ballot definition files. It can be considered a virtual machine for a high-level user interface specification language.

Pvote could be used as the core user interface component of a cryptographically auditable voting system, an electronic ballot marking or printing device, a DRE with a paper or audio audit trail, or (gasp!) a paperless DRE.

1.2 Responsibilities

We say **committed** to mean voter selections are finalized as far as the machine is concerned — for a DRE this means "recorded" or "cast," but for a ballot printer this means "printed." A **voting session** consists of the time from when a voting machine starts interacting with a particular voter (e. g. when the first screen of the voting user interface comes up) until the ballot is committed or the voter abandons the machine. This does not include per-voter initialization steps by pollworkers.

We intend to establish that Pvote can be relied upon to do the following:

R1. Never abort during a voting session. (Specifically, given any particular ballot definition, Pvote should either always reject it as invalid and abort immediately without starting a voting session, or always accept it as valid and never abort during any voting session with that ballot definition.)
R2. Remain responsive during a voting session.
R3. Become inert after a ballot is committed.
R4. Display a completion screen when and only when a ballot is committed, and continue to display this screen until the next session begins.
R5. Exhibit behaviour in each session independent of any previous sessions.
R6. Exhibit behaviour independent of which parts of buttons are touched (all touch points within a target region should be equivalent).
R7. Exhibit behaviour that is determined entirely by the ballot definition and the stream of user input events and their timing.
R8. Commit valid selections (no overvotes and no invalid candidates or contests).
R9. Commit the ballot when and only when so requested by the voter.
R10. Correctly and unambiguously commit the selections the voter made.

Pvote should also do the following correctly, according to the ballot definition:

R11. Present instructions, contests, and options as specified.
R12. Navigate among instructions, contests, and options as specified.
R13. Select and deselect options according to user actions as specified.
R14. Correctly indicate which options are selected, when directed to do so.
R15. Correctly indicate whether options are selected, when directed to do so.
R16. Correctly indicate how many options are selected, when directed to do so.

1.3 Assumptions

A1. The voting machine software (ostensibly Pvote) is handed over for review before the election.
A2. The software that runs on the voting machines on election day is exactly what was reviewed.
A3. Pvote is started once per voting session.
A4. Only authorized voters are allowed to carry out voting sessions.
A5. Ballot definition files are published for review and testing before the election.
A6. The correct ballot definition is selected and used for each voting session.
A7. The ballot definitions used on election day are intact, exactly as they were reviewed.
A8. The programming language functions correctly (according to the behaviour specified in Chapter 3).
A9. The operating system and software libraries function correctly (according to the behaviour specified in Chapters 4 and 5).
A10. The voting machine hardware functions correctly.

1.4 Threats in scope

- **Voters.** Voters can interact with Pvote using the touchscreen and keypad. Is there any sequence of interactions that can cause Pvote to allow a voter to commit multiple ballots (R3), allow committing of an invalid ballot (R8), mislead pollworkers about whether a ballot has been committed (R4), or violate voter privacy (R5)?
- **Bugs.** Though bugs are not usually considered security threats in the sense of being willful attackers, they do threaten the integrity of the election. Can any valid ballot definitions or user interactions ever cause Pvote to behave incorrectly (R1, R2, R6, R7, R8, R9, R10, R11, R12, R13, R14, R15, R16)?
- **Insiders among voting software suppliers.** Pvote could be modified to contain backdoors or hidden weaknesses before being handed over for review and installation. Could an attacker make effective changes that would go unnoticed by reviewers? What effect does Pvote have on the difficulty of performing or detecting such subversion? (This is the "meta-threat" corresponding to the two preceding items.)
- **Insiders among election officials.** Ballot definitions could be designed or altered to contain the wrong information or bias the vote. Could an attacker subvert ballot definitions in a way that would go unnoticed by reviewers and testers? What effect does Pvote have on the difficulty of performing or detecting such subversion?

1.5 Threats out of scope

- **Insiders among pollworkers.** We are relying upon pollworkers not to give voters multiple sessions (A3), not to let unauthorized people vote (A4), and to select the correct ballot style for each voter (A6). We assume election procedures make it hard for an insider working alone to violate these rules.
- **Tampering with the software distribution.** We assume the software is not altered between review and use (A1, A2).
- **Tampering with the ballot definition.** We assume the ballot definition is not altered between review and use (A5, A7).
- **Tampering with cast vote records.** We assume that if Pvote is used in a DRE, some other mechanism will be responsible for protecting the integrity of the vote records that Pvote outputs.
- **Faulty or subverted non-voting-specific software.** We assume that the software components that are not specific to voting function correctly (A8, A9). The threat of attacks on Pvote via these software components can be largely eliminated by choosing to use versions of such software that were released before Pvote was created.
- **Faulty or subverted hardware.** This is a software review (A10).
- **Poor ballot design.** We don't claim that using Pvote eliminates accessibility or usability problems, though testing with the published ballot definitions might help reveal some of these problems in time to address them.

1.6 Other questions to consider

Depending on the time available, we may be able to look at a broader set of questions surrounding Pvote.

Testing is an issue closely related to security and reliability that may be worth examining. How would Pvote affect the testing process?

- Does Pvote change the required amount of testing?
- Does Pvote change the level of confidence attainable through testing?
- Does Pvote increase or decrease the effectiveness of existing kinds of testing? Some types of testing to consider are:
  - unit testing
  - system testing
  - manual testing
  - automated testing
  - parallel testing
  - logic and accuracy testing
- Does Pvote make feasible any new kinds of tests?

How does using Pvote affect the ability to mix and match components from different vendors, and what influence would this have on testing and reliability?

How does using Pvote affect the difficulty of reviewing the voting system?

Post-election audits are also an important diagnostic and recovery tool. How does Pvote affect the ability to audit the voting system?

Finally, there is the question of integration with existing and proposed systems and practices. Running an election requires many other components in addition to Pvote. How would or could Pvote interoperate with these other components? How does it compare with, and interoperate with, other software-independent approaches to electronic voting?

Chapter 2

Ballot Definition Format

The format of the ballot definition file is central to Pvote's design, as it specifies all the capabilities of the voting user interface. The ballot definition describes a state machine where each 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.

2.1 Overview

The ballot definition contains four parts:

- **Ballot model**: structure of the ballot and interaction flow of the user interface.
- **Text data**: information for the printer.
- **Audio data**: sound data for the audio driver.
- **Video data**: image and layout data for the video driver.

The ballot model consists of:

- **Groups**: sets of options for the voter to select.
- **Pages**: the coarse-grained unit of navigation; a full-screen display state. Pages contain finer-grained **states** for navigation within a page. Both pages and states have **bindings**, which map keypresses and screen touches to selection and navigation actions. Pages and states specify audio feedback in terms of sequences of audio **segments**. Both bindings and segments may be subject to **conditions** concerning the voter's current selections. Finally, **areas** are parts of the page that change according to the voter's selections.

The text data contains the names of the contests and candidates. The audio data contains a collection of sound clips. The video data contains:

- **Layouts**: the visual appearance of a page. Each layout corresponds to one page. The layout contains a full-screen image for the page. It also specifies the locations of **targets** (screen regions that respond to touch) and **slots** (screen regions where sprites are pasted). Targets invoke bindings; areas are associated with slots.
- **Sprites**: smaller images for pasting over variable parts of the display.

The next few sections will describe in more detail the contents of these data structures and what they mean, and set out the constraints that have to be met for a ballot definition file to be considered valid.

2.2 Data types and their serialization

2.2.1 Primitive types

The data structures are built up from the following types:

- **int**: An integer in the range from 0 to 2147483647 (`0x7fffffff`) inclusive. Serialized as four bytes, most significant first.
- **intn**: An integer in the range from 0 to 2147483647 (`0x7fffffff`) inclusive, or the special value `None`. An integer value is serialized as four bytes, most significant first; the value `None` is serialized as `"\xff\xff\xff\xff"`.
- **bool**: A Boolean value. Truth is serialized as `"\x00\x00\x00\x01"` and falsehood is serialized as `"\x00\x00\x00\x00"`.
- **enum**: A value from a finite set of identifiers. Each of the three uses of **enum** (in **Step**, **Segment**, and **Condition**) has a distinct domain. Values of an **enum** correspond to small integers and are serialized in the same way as an **int**.
- **str**: A string of ASCII bytes with length from 0 up to 2147483647, where each byte is at least 32 and at most 125. Serialized as a four-byte integer length followed by the bytes of the string.
- **pixel**: A pixel colour with red, green, and blue components, each ranging from 0 to 255 inclusive. Serialized as three bytes (red, green, blue).
- **sample**: An individual audio sample value ranging from -32768 to 32767 inclusive. Serialized as a 16-bit signed integer, most significant byte first.

2.2.2 Compound types

The top-level compound type for the entire ballot definition is **Ballot**. A ballot definition file consists of an 8-byte identifying header, followed by the serialized content of the **Ballot** structure, followed by the 20-byte SHA-1 digest of the serialized content. The header is `"Pvote\x00\x01\x00"`, where the last two bytes are the major and minor version number of the format.

Figure 2.1 depicts the exact structure of **Ballot**, which is shown as the heavy box at the top. Within this box, the internal structure of all its constituent types is revealed, except for **Binding** and **Segment**, which are described in the boxes below. The figure shows all the fields in the order they are serialized. Each compound type is serialized simply by concatenating its serialized fields with no padding.

Many fields contain lists of elements. A list can have from 0 to 2147483647 elements. A list is serialized as a four-byte integer length followed by all the elements serialized in order. All the list fields (those marked with []) are serialized in this fashion, except the `pixels` field of an **Image**. The `pixels` field is serialized with no length, since the length is already determined by the `width` and `height` fields of the **Image**.

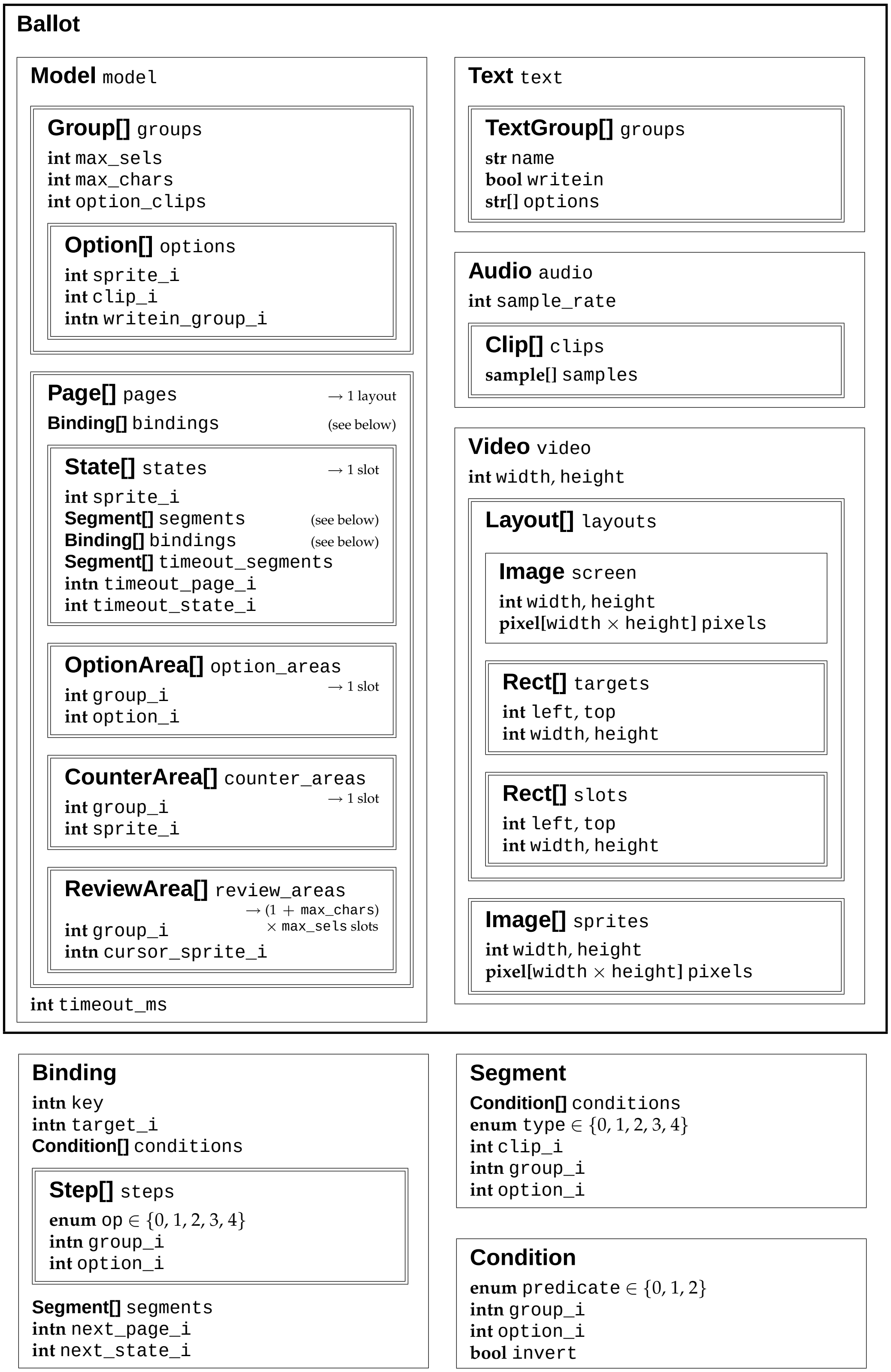

Figure 2.1: Ballot definition data structures. A double border around a subelement signifies a list of subelements of that type.

2.3 Model

The model contains **Group**s, which describe the ballot structure, and **Page**s, which describe the user interface. The model also has one integer field, `timeout_ms`, which specifies an idle timeout in milliseconds. A **timeout** is defined to occur when there has been no user activity and no audio playing for `timeout_ms` milliseconds. The ballot definition can specify an automatic transition or audio message to occur in case of a timeout.

Each field whose name ends with `_i` is an integer index that refers to an element of a list elsewhere in the structure.

2.3.1 Groups

A **Group** is a container of selectable options. Groups are used for two purposes: as *contest groups* and as *write-in groups*. A contest group represents an actual contest on the ballot; its options are options such as candidates. A write-in group represents a single write-in entry field; its options are the individual characters that can appear in the entry field. In all 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). The fields in a **Group** are:

- `max_sels`: The maximum allowed number of selections in the group.
- `max_chars`: For contest groups, this is the maximum number of characters that can be entered for any write-in option in the contest. If this is zero, the contest has no write-ins. For write-in groups, `max_chars` must be zero.
- `option_clips`: The number of audio clips associated with each option.
- `options`: The list of options in the group.

Each option (in any kind of group) is associated with exactly two sprites, one to display when the option is selected, and one to display when it is not selected. Each option can be associated with any number of audio clips (the same number for all options in a group, specified by `option_clips` in the **Group**). These come from the `sprites` and `clips` lists in the **Video** and **Audio** structures, respectively. There are three fields in an **Option**:

- `sprite_i`: An index into `video.sprites`. The sprite at index `sprite_i` is shown for the selected option, and the sprite at index `sprite_i` + `1` is shown for the unselected option.
- `clip_i`: An index into `audio.clips`. The clips with indices from `clip_i` to `clip_i` + `option_clips` − `1` are used to represent the option.
- `writein_group_i`: If this field is `None`, this option is a regular option. Otherwise, this option is a write-in option; `writein_group_i` specifies the write-in group that will hold the text entered for this write-in.

Note the logical relationships among these fields. If a group's `max_chars` is zero, then all its options must have `None` as their `writein_group_i`. Only in a contest group may `writein_group_i` can take on values other than `None`; these values must be the indices of write-in groups. The referenced write-in groups must have `max_chars` set to zero and `max_sels` equal to the contest group's `max_chars`. These relationships are summarized in the following table.

Type of group	Kinds of options it contains	**`writein_group_i`**
contest group (`max_chars` ≥ 0)	regular option	`None`
	write-in option	index of a write-in group
write-in group (`max_chars` $= 0$)	character option	`None`

Table 2.1: Types of groups and the options they contain.

2.3.2 Pages

The **Page** represents an overall display appearance such as a page of instructions or a selection page for a particular contest. The fields in a **Page** are as follows:

- `bindings`: Bindings that apply in all the states in this page.
- `states`: States that belong to this page (i. e. have this overall appearance).
- `option_areas`: Parts of the visual display that show specific options and indicate whether the options are selected or unselected.
- `counter_areas`: Parts of the visual display that change based on the number of options that are selected in a particular group.
- `review_areas`: Parts of the visual display that list all the selected options (with their write-in text, if any) in a particular group.

There is a one-to-one correspondence between pages and layouts: item *i* in the **Model**'s list of `pages` corresponds to item *i* in the **Video**'s list of `layouts`. The corresponding **Layout** gives the full-screen image for the page. The slots in the **Layout** also correspond to elements of the page: if there are *s* states, *o* option areas, *c* counter areas, and *r* review areas, then the states get the first *s* slots in the list, the option areas get the next *o* slots, the counter areas get the next *c* slots, and the review areas get the rest.

An **OptionArea** has two fields, `group_i` and `option_i`, which give the index of a group in the **Model**'s `groups` and the index of the option in that group's list of `options` that will appear in the option area's slot.

A **CounterArea** has two fields, `group_i` and `sprite_i`. The sprite at index `sprite_i` + *n* will appear in the counter area's slot, where *n* is the number of options currently selected in group `group_i`.

A **ReviewArea** has two fields, `group_i` and `cursor_sprite_i`. For each of the selected options in group `group_i`, the review area uses one slot for the option and `max_chars` slots for its write-in characters, for a total of (1 + `max_chars`) × `max_sels` slots. The `cursor_sprite_i` field can be `None`, or it can specify a sprite to be shown in the first unused option slot when the group is not full.

The actual states of the state machine are represented by the **State** data structure. The states are grouped into pages because several states often share a similar display appearance (e. g. states could highlight different user interface elements in a fixed layout of elements on the screen) and similar behaviours (e. g. the "next page" button, a common element of voting user interfaces, takes you to a new screen regardless of which element has the focus on the current screen). Organizing states into pages reduces redundancy and simplifies the work of ballot definition creators and reviewers.

A **State** has these fields:

- `sprite_i`: A sprite to be displayed in the state's slot while in this state.
- `segments`: A list of audio segments (see **Audio feedback** below).
- `bindings`: Bindings that apply to just this state. These override page-level bindings; when the user presses a key or touches a target, an operative binding is sought first in the **State** and then in the **Page**.
- `timeout_segments`: A list of audio segments to be played upon timeout (see **Audio feedback** below).
- `timeout_page_i`, `timeout_state_i`: The state to automatically enter upon timeout. If `timeout_page_i` is `None`, no automatic transition occurs.

User input

The lists of **Binding**s in pages and states specify behaviour in response to user input. Each binding specifies a triplet of stimulus, condition, and response.

There are two kinds of stimuli: keypresses, which are received as an integer key code, and screen touches, which are 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. A binding is said to *match* the stimulus if it specifies the pressed key or touched target.

The condition specifies constraints on the current selection state in order for the binding to apply. A binding is considered *operative* if its condition is satisfied.

The response consists of three parts: selection operations (given as **Step**s), audio feedback (given as **Segment**s), and navigation. To *invoke* a binding is to carry out the response. When the user provides a stimulus, at most one binding is invoked: the first matching, operative binding found in the current state or the current page.

The fields in a **Binding** are:

- `key`: A key code that this binding will match.
- `target_i`: A target index that this binding will match.
- `conditions`: A list of **Condition**s that must all be satisfied in order for this binding to be operative.
- `steps`: A list of **Step**s to be carried out when this binding is invoked.
- `segments`: A list of **Segment**s to be played when the binding is invoked.
- `next_page_i`, `next_state_i`: The state to enter when this binding is invoked. If `next_page_i` is `None`, no state transition occurs.

A **Condition** has four fields:

- `predicate`: One of the following predicate types.

  0. `PR_GROUP_EMPTY`: Satisfied when a group is empty.
  1. `PR_GROUP_FULL`: Satisfied when a group is full.
  2. `PR_OPTION_SELECTED`: Satisfied when a specific option is selected.

- `group_i`, `option_i`: Identifies the group or option to which the predicate is applied (see **Group and option references** below).
- `invert`: If true, the sense of the condition is inverted.

A **Step** has three fields:

- `op`: One of the following operation types.
  0. `OP_ADD`: Append the specified option to its group's selection list if not already present.
  1. `OP_REMOVE`: Remove the specified option from its group's selection list if it is present.
  2. `OP_APPEND`: Append the specified option to its group's selection list.
  3. `OP_POP`: Remove the last option from the specified group's selection list.
  4. `OP_CLEAR`: Clear the specified group's selection list.
- `group_i`, `option_i`: Identifies the group or option to which the operation is applied (see **Group and option references** below).

Audio feedback

Audio feedback is specified as a list of segments. Some segments simply play a particular clip; others can play different clips depending on the selection state.

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, unless the queue is empty.

Invoking a binding always interrupts any currently playing clip and clears the play queue. The segments for the binding, if any, are queued first; if a state transition occurs, the segments for the newly entered state are queued next.

The fields in a **Segment** are:

- `conditions`: A list of **Condition**s that must all be satisfied in order for this segment to be considered (otherwise, it is skipped). The conditions are evaluated when the segment list is being queued, immediately after executing the `steps` of a **Binding**, after entering a new **State**, or on timeout.
- `type`: One of the following segment types.
  0. `SG_CLIP`: Play the clip at `clip_i`.
  1. `SG_OPTION`: Play the clip at offset `clip_i` from the specified option's `clip_i`. If the option has a write-in group, also play the clips for all the selected options in the write-in group (when playing the character options for a write-in, use each option's `clip_i` with no offset).
  2. `SG_LIST_SELS`: For each selected option in the specified group, play the clip at offset `clip_i` from the selected option's `clip_i`. If the option has a write-in group, also play the clips for all the selected options in the write-in group (use each option's `clip_i` with no offset).
  3. `SG_COUNT_SELS`: Play the clip at offset *n* from the specified `clip_i`, where *n* is the number of selected options in the specified group.
  4. `SG_MAX_SELS`: Play the clip at offset *n* from the specified `clip_i`, where *n* is `max_sels` for the specified group.
- `clip_i`: A clip index or offset applied to a clip index, depending on `type`.
- `group_i`, `option_i`: Identifies the option or group for which a clip is played (see **Group and option references** below).

Group and option references

In a **Condition**, **Step**, or **Segment**, the pair of fields `group_i` and `option_i` is used to refer to a group or option. If `group_i` is `None`, then `option_i` is the index of an **OptionArea** on the current page; the pair (`group_i`, `option_i`) *indirectly* refers to the group or option of this **OptionArea**. Otherwise, the pair (`group_i`, `option_i`) *directly* refers to group `group_i` in the **Model**'s list of `groups` or option `option_i` in that **Group**'s list of `options`.

2.4 Text

The text data provides textual labels for groups and options so that the user's selections can be printed out. The **Text** structure has just one field, `groups`, which is a list of of **TextGroup**s. Each **TextGroup** has three fields:

- `name`: The name of the group.
- `writein`: If true, the group is to be printed as a write-in group. Otherwise, the group is to be printed as a contest group.
- `options`: A list of the names of the options in the group.

2.5 Audio

The **Audio** structure contains two fields:

- `sample_rate`: The playback rate in samples per second.
- `clips`: A list of audio clips, referenced by index in **Option** and **Segment** structures.

Each clip is a **Clip** structure, which contains just one field:

- `samples`: A list of signed 16-bit samples. Audio clips have one channel.

2.6 Video

The **Video** structure has the following fields:

- `width`, `height`: The display screen resolution.
- `layouts`: A list of **Layout**s, one for each **Page** in the **Model**.
- `sprites`: A list of **Image**s for pasting onto the display, referenced by index in **Option**, **State**, and **ReviewArea** structures.

The fields in a **Layout** are as follows:

- `screen`: The full-screen page image (over which sprites will be pasted).
- `targets`: A list of rectangular screen regions where touches will be detected and acted upon.
- `slots`: A list of rectangular screen regions where sprites will be pasted.

Images are specified as an integer `width` and integer `height` followed by pixel data (3 bytes per pixel). The rectangular regions for targets and slots are specified as four integers, `left`, `top`, `width`, and `height`.

2.7 Validity constraints

A ballot definition is considered **valid** if it meets the constraints in this section. These constraints are intended to be sufficient (though not necessary) to ensure that Pvote will not terminate abnormally through a fatal runtime error at the language level or illegal calls to library routines. Possible causes of such errors are pointed out in the specifications given in Chapters 3, 4, and 5. For example, these constraints try to ensure that list indices will be within bounds, but not that every option appears on the ballot. No rules can enforce the correctness of the user interface, so the job of helping humans evaluate ballot designs is left to other tools (which can apply usability recommendations or region-specific election rules).

The following specification of the data structures is annotated with validity constraints on the right. In these constraint expressions, all arithmetic is performed with mathematical integers. *length*(x) refers to the length of a list x, and the symbol $\circeq$ means "sizes match" (that is, `a` $\circeq$ `b` $\Leftrightarrow$ `a.width` $=$ `b.width` and `a.height` $=$ `b.height`). For brevity, some unqualified names are used:

- `groups` and `pages` refer to the fields of the **Model** object
- `group` and `page` refer to the **Group** or **Page** containing the current element
- `clips` refers to the field of the **Audio** object
- `sprites` refers to the field of the **Video** object

```
Ballot:
    Model model
    Text text
    Audio audio
    Video video

Model:
    Group[] groups              length(groups) = length(text.groups) > 0
    Page[] pages                length(pages) = length(layouts) > 0
    int timeout_ms

Group:
    int max_sels
    int max_chars
    int option_clips            option_clips > 0
    Option[] options

Option:
    int sprite_i                sprite_i + 1 < length(sprites)
                                sprites[sprite_i] ≗ sprites[sprite_i + 1] ≗ group's option size
    int clip_i                  clip_i + group.option_clips − 1 < length(clips)
    intn writein_group_i        writein_group_i ≠ None ⇒ (writein_group_i < length(groups) and
                                      groups[writein_group_i].max_chars = 0 and
                                      groups[writein_group_i].max_sels = group.max_chars > 0 and
                                      ∀ option ∈ groups[writein_group_i].options:
                                            sprites[option.sprite_i] ≗ group's character size)
Page:
    Binding[] bindings
    State[] states              length(states) > 0
    OptionArea[] option_areas
    CounterArea[] counter_areas
    ReviewArea[] review_areas
```

```
State:
    int sprite_i                    sprite_i < length(sprites)
                                    sprites[sprite_i] ≗ slots[state_i]
    Segment[] segments
    Binding[] bindings
    Segment[] timeout_segments
    intn timeout_page_i             timeout_page_i ≠ None ⇒ timeout_page_i < length(pages)
    intn timeout_state_i            timeout_page_i ≠ None ⇒ timeout_state_i < length(page[timeout_page_i].states)

OptionArea:
    int group_i                     group_i < length(groups)
    int option_i                    option_i < length(groups[group_i].options)
                                    option area's slot ≗ groups[group_i]'s option size
CounterArea:
    int group_i                     group_i < length(groups)
    int sprite_i                    sprite_i + groups[group_i].max_sels < length(sprites)
                                    ∀ i ∈ {0, 1, 2, ..., groups[group_i].max_sels}:
                                        counter area's slot ≗ sprites[sprite_i + i]
ReviewArea:
    int group_i                     group_i < length(groups)
    intn cursor_sprite_i            cursor_sprite_i ≠ None ⇒ (cursor_sprite_i < length(sprites) and
                                        sprites[cursor_sprite_i] ≗ groups[group_i]'s option size)
                                    review area's option slot ≗ groups[group_i]'s option size
Binding:                            ∀ slot ∈ review area's character slots:
    intn key                            slot ≗ groups[group_i]'s character size
    intn target_i
    Condition[] conditions
    Step[] steps
    Segment[] segments
    intn next_page_i                next_page_i ≠ None ⇒ next_page_i < length(pages)
    intn next_state_i               next_page_i ≠ None ⇒ next_state_i < length(page[next_page_i].states)

Condition:
    enum predicate                  predicate ∈ {0, 1, 2}
    intn group_i                    group_i ≠ None ⇒ group_i < length(groups)
    int option_i                    group_i = None ⇒ option_i < length(page.option_areas)
    bool invert                     group_i ≠ None ⇒ option_i < length(groups[group_i].options)

Step:
    enum op                         op ∈ {0, 1, 2, 3, 4}
    intn group_i                    group_i ≠ None ⇒ group_i < length(groups)
    int option_i                    group_i = None ⇒ option_i < length(page.option_areas)
                                    group_i ≠ None ⇒ option_i < length(groups[group_i].options)
Segment:
    Condition[] conditions
    enum type                       type ∈ {0, 1, 2, 3, 4}
    int clip_i                      type = 0 ⇒ clip_i < length(clips)
                                    type ∈ {1, 2} ⇒ clip_i < groups[g].option_clips
                                    type ∈ {3, 4} ⇒ clip_i + groups[g].max_sels < length(clips)
                                    where g = group_i if group_i ≠ None, or
                                        g = page.option_areas[option_i].group_i otherwise
    intn group_i                    group_i ≠ None ⇒ group_i < length(groups)
    int option_i                    type ≠ 0 and group_i = None ⇒ option_i < length(page.option_areas)
                                    type ≠ 0 and group_i ≠ None ⇒ option_i < length(groups[group_i].options)
```

```
Text:
    TextGroup[] groups              ∀ i ∈ {0, 1, 2, ..., length(groups) - 1}:
                                        length(groups[i].options) = length(model.groups[i].options)
TextGroup:
    str name                        length(name) ≤ 50
    bool writein
    str[] options                   ∀ s ∈ options, length(s) ≤ 50

Audio:
    int sample_rate
    Clip[] clips

Clip:
    sample[] samples                length(samples) > 0

Video:
    int width                       width > 0
    int height                      height > 0
    Layout[] layouts
    Image[] sprites

Layout:
    Image screen                    screen ≗ video
    Rect[] targets
    Rect[] slots

Image:
    int width                       width > 0
    int height                      height > 0
    pixel[width*height] pixels

Rect:
    int left
    int top
    int width                       left + width ≤ video.width
    int height                      top + height ≤ video.height
```

Some of the above constraints refer to the *option area's slot*, *counter area's slot*, and *review area's slots*, which are slots taken from the `slots` list of the page's corresponding **Layout** object, as described in Section 2.3.2.

The size constraints on sprites and slots also refer to the *option size* and *character size* of a group, even though the **Group** structure doesn't have fields for specifying option size and character size. This just means that all the objects that are required to match a particular group's *option size* must all have the same size, and all the objects that are required to match a particular group's *character size* must all have the same size.

The constraints requiring each **Clip** to have a nonzero length and each **Image** to have nonzero width and height are present due to a Pygame limitation: Pygame refuses to create zero-length sounds or zero-sized images. Were it not for this limitation, they would be unnecessary — it would be logical for playing a zero-length clip or pasting a zero-sized image to have no effect.

Chapter 3

Pthin

Though the implementation of Pvote is developed, tested, and demonstrated on the open-source Python interpreter (versions 2.3, 2.4, and 2.5), it only uses a small subset of the Python language. To limit the scope of the review and to save the reviewers from having to read the entire Python reference manual, this section defines "Pthin", a subset of Python sufficient to run Pvote. Differences between the Pthin specification and the behaviour of the Python interpreter are out of scope for this review.

3.1 Types

Values in Pthin are typed, but variables are not. There is a unique special value called `None` whose only supported operation is comparison to `None`. Aside from `None`, there are six types of values in Pthin: integers, strings, lists, functions, classes, and objects.

Integers are signed and unlimited in size. Integer literals are written in decimal.

Strings are variable-length arrays of 8-bit bytes. String literals are written exactly as in C. Null bytes have no special significance.

Lists are variable-length arrays of Pthin values. Lists can be heterogeneous and can contain values of any type as elements. List literals are written in square brackets with elements separated by commas.

Functions may take any number of arguments and always return one value. Functions are defined with the `def` keyword (see Section 3.4 for more on functions).

Classes contain method definitions and can be invoked to instantiate objects. Classes are defined with the `class` keyword (see Section 3.5 for more on classes).

Objects are instances of classes. Each object contains its own public namespace, accessed with a dot. For example, if `x` is an object, then `x.foo = 3` binds `foo` to 3 in the namespace belonging to `x`. An object's methods are simply functions residing in its namespace (see Section 3.5 for more on methods).

Table 1 is a summary of expressions involving these types. When the arguments to an operation are of unacceptable types, a fatal runtime error occurs.

Assignment binds a name to a reference, lists and object namespaces contain references, and arguments are passed by reference. (This works like Scheme or Java with objects only: all values are boxed, even integers.)

Expression	Preconditions	Definition
(*expr*)		evaluate *expr*
None		literal for the special value None
123		integer literal
"abc"		string literal
[*expr1*, *expr2*, ...]		list literal
[*expr* for *name* in l]		evaluate *expr* with *name* bound to each element of l
f(*arg1*, ...)	arguments match f's arity	call a function
c(*arg1*, ...)	arguments match c's arity	create an object that is an instance of c
o.*field*	*field* is bound in o's namespace	access a field in the object o's namespace
s[i:j]	$0 \leq i \leq j <$ length of s	get a substring (skip first i bytes, get next j - i bytes)
l[i]	$0 \leq i <$ length of l	get the element of l at index i (counting starts at zero)
i * j		multiply
i / j	$j \neq 0$	divide and round down
i % j	$j \neq 0$	i - j*(i/j)
s * i	$i \geq 0$	concatenate i copies of s to make a new string
i + j		add
i - j		subtract
l + m		concatenate two lists to make a new list
s + t		concatenate two strings to make a new string
Comparison operators can be chained (e. g. 10 <= x < 20*). The result is the conjunction of all the comparisons.*		
i == j		1 if $i = j$; 0 otherwise
i != j		1 if $i \neq j$; 0 otherwise
i == None		1 if i is None; 0 otherwise
i != None		1 if i is not None; 0 otherwise
i < j		1 if $i < j$; 0 otherwise
i > j		1 if $i > j$; 0 otherwise
i <= j		1 if $i \leq j$; 0 otherwise
i >= j		1 if $i \geq j$; 0 otherwise
s == t		1 if s and t are identical strings; 0 otherwise
s != t		1 if s and t are different strings; 0 otherwise
i in l		1 if i is an element of l; 0 otherwise
i not in l		1 if i is not an element of l; 0 otherwise
not i		1 if i is zero; 0 otherwise
i and j		1 if i and j are both nonzero; 0 otherwise
i or j		1 if i or j or both are nonzero; 0 otherwise

Table 3.1: Expression syntax, with operators grouped by precedence (highest at the top). The above expressions are only legal with the types of operands indicated: i and j are integers, s and t are strings, l and m are lists, f is a function, c is a class, o is an object, and x is a value of any type. If operands of unacceptable types are used in these expressions or a precondition is violated, a fatal error occurs.

3.2 Namespaces

Bindings are created by assignment statements, the `for` statement, function definitions, and class definitions. Bindings can exist in three types of namespaces: global namespaces, local namespaces, and object namespaces.

Each Pthin file has one **global namespace**. Whenever a function is invoked, a new **local namespace** is created for the execution frame, and it lasts until the frame is exited.

Pthin has lexical scoping with just two levels. When names are bound outside of a function, the binding is created in the global namespace. When names are bound inside of a function, the binding is created in the local namespace.

Within a function, names can refer to bindings in the global or local namespace. A name refers to a local binding if a binding to that name exists anywhere within the function definition. Otherwise, the name refers to a global binding.

Every object has its own public **object namespace**. Object namespaces are always accessed explicitly using the dot operator on the object.

3.3 Statements

Many kinds of statements contain blocks of code that are syntactically delimited by indentation. A block is introduced with a colon at the end of a line. The body of the block is indented with respect to its introducing line, and ends when the indentation level returns to match the indentation of the introducing line.

The `assert` statement evaluates an integer-valued expression and causes a fatal runtime error if the value is zero.

The `print` statement sends a string to the printer.

An `if` statement takes the form `if` *condition*`:` followed by an indented block. The condition must evaluate to an integer. The block is executed if the condition is nonzero. This can be optionally followed by `else:` (indented to match its `if`) and another indented block to be executed if the condition is zero.

A `while` loop takes the form `while` *condition*`:` followed by an indented block. The condition must evaluate to an integer. Just as in C, the block is repeatedly evaluated as long as the condition is nonzero.

A `for` loop takes the form `for` *name* `in` *expr*`:` followed by an indented block. The expression *expr* must evaluate to a list. The `for` loop binds *name* to each element of the list in turn, executing the body once for each element.

The `import` statement imports Pthin modules and makes them available in the current namespace. A Pthin module is just a text file containing Pthin code, with a filename ending in `.py`. The statement `import` *name* creates a new object to represent the module and executes *name*`.py` using that object's namespace as the global namespace. That is, all the global bindings in the file appear as bindings in the module object's namespace. The module object is then bound to *name*. If the module has already been imported, it is not executed again; *name* is bound to the already existing module object.

See Table 3.2 for a summary of these statement types.

Statement	Preconditions	Definition
name = x		create or replace a binding in the current namespace
o.*field* = x		create or replace a binding in the object o's namespace
l[i] = x	$0 \leq$ i $<$ length of l	set the element of l at index i to x
[$lvalue_1$, ..., $lvalue_n$] = l	n = length of l	assign to multiple lvalues (names, fields, or list items)
assert i		cause a fatal runtime error if i is zero
print s		send s and a newline to the printer

if i:
    block
else:
    block
		if i is nonzero, execute the first *block*
		
		otherwise execute the second *block*

while i: *block*		execute *block* repeatedly as long as i is nonzero
for *lvalue* in l: *block*		for each element of l, assign it to *lvalue* and execute *block*
import $name_1$, $name_2$, ...		import the modules $name_1$, $name_2$, ... from the files $name_1$.py, $name_2$.py, ... respectively
def *name*($param_1$, $param_2$, ...): *block*		create a function with parameters $param_1$, $param_2$, ...
return *expr*		exit a function, returning *expr* as the result

class *name*:
    def *method*($param_1$, $param_2$, ...):
        block
    ...
		create a class with the given methods

Table 3.2: Statements in Pthin. These are only legal with the types indicated: i is an integer, s is a string, l is a list, o is an object, and x is a value of any type. If an unacceptable type is supplied or a precondition is violated, a fatal error occurs.

3.4 Functions

A function is defined with the def keyword followed by the name of the function, a pair of parentheses surrounding a comma-separated list of parameter names, and a colon. The body of the function is an indented block. Executing a function definition binds the name to the newly created function. Here's an example:

```
def factorial(n):
    if n == 0 or n == 1:
        return 1
    return n * factorial(n - 1)
```

Calling a function creates a new local namespace in which the parameter names are bound to the arguments passed in. If the number of arguments does not match the number of parameters, a fatal runtime error occurs.

Within the body of a function, return *expr* exits the function with a return value. If no return statement is executed, the function returns None.

3.5 Classes and objects

A class is defined with the `class` keyword followed by the name of the class and a colon, then an indented block containing a series of method definitions. Each method definition is a function definition with at least one parameter. Since the object itself is always passed into a method as the first argument, the first parameter is conventionally named `self`.

Invoking a class creates a new object belonging to the class. The new object's namespace acquires a binding for each method in the class. Each method definition with n parameters in the class yields a function of the same name with $n - 1$ parameters in the object's namespace. Invoking this function with some arguments is equivalent to invoking the corresponding method with one extra argument, the object itself, prepended to the given argument list.

Immediately after the object is created, the function named `__init__` in its namespace is invoked with the arguments passed into the invocation of the class.

Here's an example of a simple class definition:

```
class Counter:
    def __init__(self, n):
        self.count = count
    def next(self):
        self.count = self.count + 1
        return self.count
```

`c = Counter(5)` would create a new Counter object with `c.count` initially bound to 5. Invoking `c.next()` would increment `c.count` to 6 and return 6.

3.6 Built-in functions and methods

The functions in Table 3.3 are available from any scope.

Expression	**Result**	**Preconditions**	**Definition**
`range(i)`	list	`i` ≥ 0	make a list of the `i` integers from 0 to `i - 1`
`chr(i)`	string	$0 \leq$ `i` ≤ 255	convert `i` to a one-byte string
`ord(s)`	integer	`len(s)` = 1	convert the first byte of `s` to an integer
`len(s)`	integer		get the number of bytes in `s`
`list(s)`	list		break `s` into a list of one-byte strings
`len(l)`	integer		get the number of elements in `l`
`enumerate(l)`	list		make a list of pairs `[i, x]` for each element `x` and its index `i`
`l.append(x)`	`None`		append `x` as one more element at the end of `l`
`l.remove(i)`	`None`	`i` is an element of `l`	find and remove the first element that equals `i` from `l`
`l.pop()`	any	`l` is not empty	remove and return the last element from `l`
`open(s)`	object	a file named `s` exists and is readable	open a file for reading, yielding a readable stream object

Table 3.3: Built-in functions and methods in Pthin. In these descriptions, `i` is an integer, `s` is a string, `l` is a list, and `x` is a value of any type.

3.7 Readable stream objects

The term "readable stream object" refers to any object with a `read` method that takes a single integer argument, `length`, and returns a string of up to `length` bytes. The underlying concept is that the object maintains a current position in a finitely long data stream, and each invocation of `read` returns the next `length` bytes from the data stream and advances the current position by `length` bytes in preparation for the next `read`. If there are fewer than `length` bytes remaining to be read, the result is a string containing whatever is left in the data stream; if the end of the stream has been reached, the result is an empty string.

Opening a file with the built-in `open` function returns an object that provides this protocol. Custom objects that provide this protocol can also be instantiated from class definitions that implement an appropriate `read` method.

3.8 Memory management

Pthin dynamically allocates memory for data and for the stack. Each Pthin file (the main program or any module being imported) is analyzed before it is executed.

3.8.1 Data

During the analysis phase, a Pthin file is scanned for literals. Memory is allocated just once for literals; their values are created and kept in a pool of constants. During execution, when an expression yields a value, memory is allocated for the value.

Every value has a reference count. The reference count is incremented whenever an assignment statement binds a global name, a local name, or a field of an object to the value. The reference count is also incremented whenever a reference to the value is placed in a list, either by assignment or the `append` method.

The reference count is decremented whenever a binding to the value is replaced or a name bound to the value goes out of scope. The reference count is also decremented whenever a reference to the value is removed from a list by replacement or the `remove` or `pop` methods.

Arguments and returned values are passed to and from functions and methods by pushing them onto an internal stack. Reference counts are also incremented and decremented when values are pushed onto or popped off of this stack.

When decrementing the reference count causes the count to reach zero, the memory for the value is deallocated. If the value is a list or object, the reference counts of all its elements or fields are also decremented.

3.8.2 Stack

During the analysis phase, the entire file is scanned for global names, and enough space is allocated to hold all of the global bindings. Then, every function and method declaration is scanned for local names, and the amount of space needed for each function or method's local bindings is recorded.

Whenever a function or method is called, memory is allocated for a new stack frame, which holds the local bindings and saves the parent execution context. When the function or method returns, its stack frame is deallocated.

Chapter 4

Pygame

Pvote uses the Pygame library for graphics, sound, and user input. This section specifies the parts of Pygame that Pvote uses and their expected behaviour.

4.1 Events

A Pygame program is built around a main event loop that processes incoming events one at a time. When events occur, Pygame adds them to an internal queue. Each call to `pygame.event.wait()` waits until the queue is nonempty, then removes and returns the first event from the queue. The returned event object always has an integer field `type` specifying the kind of event, and may have other fields for details of the event, depending on the type.

Function	Preconditions	Definition
`pygame.event.wait()`		Wait for the next event and return an event object describing it.
`pygame.time.set_timer(`*event*`,` *period*`)`	*event* is an integer event type code greater than or equal to `pygame.USEREVENT` = 24 and less than `pygame.NUMEVENTS` = 32. *period* is an integer number of milliseconds.	Set the timer period for event type *event* to *period*. If *period* > 0, an event of type *event* will be placed on the queue in *period* ms and again every *period* ms thereafter. Each event type has its own timer. If *period* $= 0$, the timer for event type *event* is disabled.

Event `type` value	Definition
`pygame.KEYDOWN` (2)	A key has been pressed. The integer key code is given in the `key` field of the event object.
`pygame.MOUSEBUTTONDOWN` (5)	A mouse button has been pressed. The coordinates of the mouse pointer are given in the `pos` field of the event object, which is a list of two integers.

Table 4.1: Pygame event operations and event types used by Pvote.

4.2 Audio

Pygame provides a `mixer` facility for playing audio. The mixer can play many sounds at once, though Pvote is designed specifically to avoid this capability. Sound clips are represented by **Sound** objects that can be told to `play()` themselves. Each time a **Sound** starts playing, it is assigned to an available **Channel**; the mixer mixes all the channels together (by default, there are 8 channels). A channel can be asked to trigger a notification event when its current sound clip finishes playing.

Table 4.2 summarizes the Pygame functions and methods that Pvote uses to implement audio playback.

Function	Preconditions	Definition
`pygame.mixer.init(`*rate*`, `*format*`, `*stereo*`)`	*rate* is a valid sample rate (11025, 22050, or 44100). *format* is 8 for unsigned 8-bit samples or -16 for signed 16-bit samples. *stereo* is 0 for mono or 1 for stereo.	Initialize the audio player with the given sample rate (in samples per second), sample format, and mono/stereo setting. Must be called before any other audio operations.
`pygame.mixer.stop()`		Stop any currently playing sounds on all channels. Any currently playing channels that previously had an end event set with `set_endevent` will immediately place their end events on the event queue.
`pygame.mixer.Sound(`*stream*`)`	*stream* is a readable stream object (see Section 3.7). When read, *stream* yields the contents of a valid WAV file (see Section C).	Create a **Sound** object from the audio data in the given WAV file.

Class	Method / Preconditions	Definition
Sound	`play()`	Start playing this sound clip and return the **Channel** on which the clip is playing.
Channel	`set_endevent(`*event*`)` *event* is an integer event code greater than or equal to `pygame.USEREVENT` = 24 and less than `pygame.NUMEVENTS` = 32.	Set the end event for this channel. Each channel can have its own end event type. If $event > 0$, then from now on, an event of type *event* will be placed on the event queue each time a sound clip stops playing on this channel. The end event can be triggered by playing to the end of the clip or by a call to `stop()`. If $event = 0$, then the sending of end events for this channel is turned off.

Table 4.2: Pygame audio operations used by Pvote.

4.3 Video

All drawing takes place on frame buffers represented by **Surface** objects. Initializing the video system yields a **Surface** for the display. After drawing on the surface, one must call the display's `update` method to copy the changed contents of the frame buffer to the visible display.

Pvote constructs its visual display entirely by pasting prerendered images onto the screen. It needs to use only one drawing method, `blit`, for this purpose.

Table 4.3 summarizes the functions and methods that Pvote uses for visual display.

Function	Preconditions	Definition
`pygame.display.set_mode(`*size*`,` *flags*`)`		
	size is a pair of integers `[`*width*`,` *height*`]`. *flags* is an integer.	Initialize the video display with a resolution of *width* × *height* pixels and return a **Surface** object. If *flags* is `pygame.FULLSCREEN`, the display fills the screen.
`pygame.display.update()`		
		Update the video display to reflect the contents of its surface object. (Drawing commands will alter the surface in memory, but the contents are not placed on the display until `update` is called.)
`pygame.image.fromstring(`*data*`,` *size*`, "RGB")`		
	size is a pair of integers `[`*width*`,` *height*`]`. *data* is a string of *width* × *height* × 3 bytes.	Create an **Image** object from the pixel data in *data*, which is ordered left to right, top to bottom, and has 3 unsigned bytes per pixel (red, green, and blue values respectively).

Class	Method Preconditions	Definition
Surface	`blit(`*image*`,` *pos*`)`	
	image is an **Image** object with size (*width*, *height*). *pos* is a list of two integers `[`x`,` y`]`. $0 \leq x < x + width \leq$ width of surface. $0 \leq y < y + height \leq$ height of surface.	Paste an image onto this surface with its top-left corner at the given (x, y) position.

Table 4.3: Pygame video operations used by Pvote.

Chapter 5

SHA

Pvote uses the Python SHA module to compute SHA-1 digests. After the module has been imported with the statement `import sha`, calling `sha.sha()` creates a new SHA hashing object. The SHA object supports progressively adding more input data with the `update` method; at any point the `digest` method can be called to obtain the digest of the data submitted to far.

Function	Preconditions	Definition
`sha.sha()`		Create a new SHA object.

Class	Method Preconditions	Definition
sha	`o.update(s)` `s` is a string.	Append `s` to the data being hashed.
sha	`o.digest()`	Return a 20-byte string with the SHA-1 digest of all the data sent to this object so far.

Table 5.1: SHA module operations used by Pvote.

Chapter 6

Pvote

6.1 Design

Pvote consists of seven components:

- **Main program and event loop** (`main`): Responsible for loading the other components and receiving and dispatching Pygame events.
- **Ballot loader** (`Ballot.py`): Responsible for deserializing the ballot definition file and verifying its header and digest.
- **Ballot verifier** (`verifier.py`): Responsible for checking the validity of the ballot definition according to the constraints described in Section 2.7.
- **Navigator** (`Navigator.py`): Responsible for keeping track of the user's selections and the current state of the user interface, and performing selection, navigation, or audio feedback in response to user actions.
- **Audio driver** (`Audio.py`): Responsible for queueing and playing audio.
- **Video driver** (`Video.py`): Responsible for drawing the visual display.
- **Printer driver** (`Printer.py`): Responsible for printing the committed ballot.

When Pvote starts up, the ballot loader is invoked to deserialize the ballot definition into memory, and then the verifier is invoked to check the ballot definition. The purpose of the verifier is to ensure immediate termination on an invalid ballot definition, so that a fatal error cannot be caused by an illegal type, precondition violation, or assertion failure after a voting session has begun.

The remaining five components form the virtual machine (Figure 6.1) that presents the voting user interface to the voter. Each component has limited responsibilities, and there are limited data flows between components.

The **navigator** keeps track of the current page and state and the current selections in each group. 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 `timeout_segments` for the current state to the play queue. Go to the page and state given by `timeout_page_i` and `timeout_state_i` if `timeout_page_i` is not `None`.

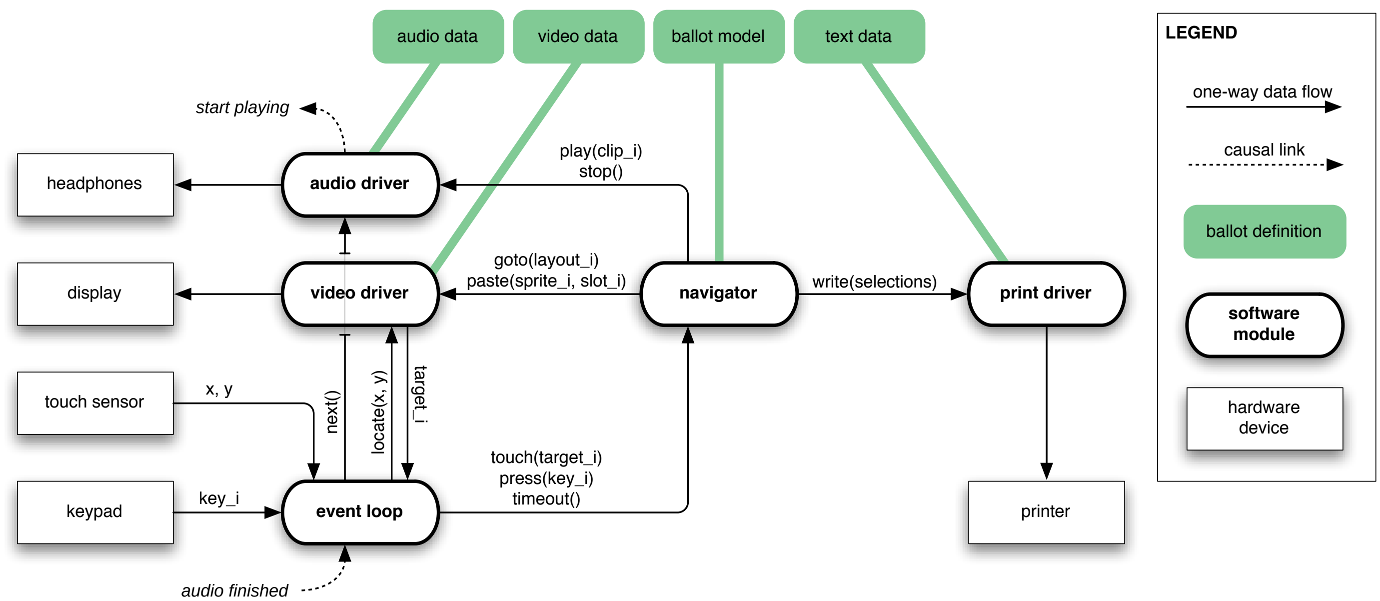

Figure 6.1: Block diagram of the virtual machine, which consists of the five software modules in bold. The arguments clip_i, layout_i, sprite_i, target_i, key_i, x, and y are integers; selections is a list of lists of integers.

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.
- **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.
- **play**(clip_i) is sent to the audio driver to queue a clip to be played on the headphones.
- **stop**() is sent to the audio driver to stop the currently playing clip.
- **write**(selections) is sent to the printer to commit the user's selections by printing the ballot.

The **audio driver** maintains a queue of audio clips to be played. It responds to two messages:

- **play**(clip_i): If nothing is currently playing, immediately begin playing the specified clip; otherwise queue the specified clip to be played. clip_i is an index into the list of clips in the **Audio** part of the ballot definition.
- **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 also exposes a field named playing that the main loop can read to determine whether a sound clip is currently being played. Whenever the audio driver starts playing a clip, it also ensures that a notification event with event type AUDIO_DONE will occur when the clip finishes playing.

The **video driver** maintains 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 set the current layout to layout_i.
- **paste**(sprite_i, slot_i): Copy the given sprite into the frame buffer at the position specified by slot slot_i in the current layout's slot list.
- **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 None if the point does not fall within any target.

The **print driver** maintains no state and responds to only one message:

- **write**(selections): Print out the voter's selections. selections is a list of lists (one for each group). The sublists contain the integer indices of selected options within each group.

The **event loop** receives four kinds of Pygame events:

- Keypresses (KEYDOWN): Upon receiving a keypress event, the event loop notifies the navigator with a **press** message.
- Mouse clicks (MOUSEBUTTONDOWN): Upon receiving a touch event, the event loop invokes **locate** on 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 (AUDIO_DONE): Upon receiving notification that a sound clip has finished playing, the event loop invokes **next** on the audio driver.
- Timer notifications (TIMER_DONE): Upon receiving notification that the timer has expired, if no sound clip is currently playing, the event loop sends **timeout** to the navigator to indicate that the ballot's specified timeout has passed with no activity.

The event loop also reschedules a TIMER_DONE event for timeout_ms milliseconds in the future every time it receives any event.

The audio driver, video driver, and printer driver are passive components: they only respond to received messages and initiate no messages of their own.

6.2 Source Code

The following sections display a complete listing of the source code to Pvote, with three columns of annotations on the left. The **PRECONDITIONS** column contains assumptions and preconditions for each line, function, or method. The **REASONS FOR VALIDITY** column explains why each line will not cause a fatal runtime error, or marks potential causes of a fatal error with the symbol ▲. The **POSTCONDITIONS** column identifies what is expected to be true after a line, function, or method has completed execution. The preconditions and postconditions for each function or method are given on the first line (the def line) to facilitate modular reasoning.

Assumptions and postconditions of other lines are cited as evidence. Small numbers in parentheses (123) refer to lines in the current file, and lines in other files are cited with the filename and a colon, as in (Navigator:123).

	PRECONDITIONS	REASONS FOR VALIDITY	POSTCONDITIONS
1			
2	pygame.USEREVENT is an int.		AUDIO_DONE is an int.
3	pygame.USEREVENT is an int.		TIMER_DONE is an int.
4	A file named ballot exists.	open() returns a readable stream object.	ballot is a **Ballot** object.
5		ballot is a **Ballot** object (4).	ballot is valid (verifier:1).
6		ballot.audio is a **Ballot.Audio** object (4, Ballot:8).	audio is an **Audio.Audio**.
7		ballot.video is a **Ballot.Video** object (4, Ballot:9).	video is a **Video.Video**.
8		ballot.text is a **Ballot.Text** object (4, Ballot:7).	printer is a **Printer**.
9		ballot.model is a **Ballot.Model** (4, Ballot:6). audio is an **Audio.Audio** (6). video is a **Video.Video** (7). printer is a **Printer** (8).	navigator is a **Navigator**.
10			
11			
12		TIMER_DONE is an int (3). ballot is a **Ballot** (4) ⇒	
13		ballot.model.timeout_ms is an int (Ballot:6, Ballot:19).	event is a **pygame.Event**.
14			
15	pygame.KEYDOWN is an int.	event is an **Event** (13) ⇒ event.type exists and is an int.	
16		navigator is a **Navigator** (9). event is a keypress (15) ⇒ event.key exists and is an int.	
17	pygame.MOUSEBUTTONDOWN is an int.	event is an **Event** (13) ⇒ event.type exists and is an int.	
18		event is a mouse click (17) ⇒ event.pos exists and is a list of two ints.	x and y are ints.
19		video is a **Video.Video** (7). x and y are ints (18).	target_i is an int or None (Video:18).
20			
21		navigator is a **Navigator** (9). target_i is an int (19, 20).	
22		AUDIO_DONE is an int (2). event is an **Event** (13) ⇒ event.type is an int.	
23		audio is an **Audio.Audio** (6).	
24		TIMER_DONE is an int (3). event is an **Event** (13) ⇒ event.type is an int.	
25		navigator is a **Navigator** (9).	

6.2.1 main.py

```
import Ballot, verifier, Audio, Video, Printer, Navigator, pygame

AUDIO_DONE = pygame.USEREVENT
TIMER_DONE = pygame.USEREVENT + 1

ballot = Ballot.Ballot(open("ballot"))
verifier.verify(ballot)
audio = Audio.Audio(ballot.audio)
video = Video.Video(ballot.video)
printer = Printer.Printer(ballot.text)
navigator = Navigator.Navigator(ballot.model, audio, video, printer)

while 1:
    pygame.display.update()
    pygame.time.set_timer(TIMER_DONE, ballot.model.timeout_ms)
    event = pygame.event.wait()
    pygame.time.set_timer(TIMER_DONE, 0)

    if event.type == pygame.KEYDOWN:
        navigator.press(event.key)

    if event.type == pygame.MOUSEBUTTONDOWN:
        [x, y] = event.pos
        target_i = video.locate(x, y)
        if target_i != None:
            navigator.touch(target_i)
    if event.type == AUDIO_DONE:
        audio.next()
    if event.type == TIMER_DONE and not audio.playing:
        navigator.timeout()
```

	PRECONDITIONS	REASONS FOR VALIDITY	POSTCONDITIONS
1			`sha` is bound to the SHA module.
2			
3	`stream` is a readable stream.		The ballot definition file is complete and the types of its fields are valid, or ⚠ .
4		`stream` is a stream (3). ⚠ if file header not present.	
5		`sha` is the SHA module (1) $\Rightarrow$ `sha.sha` is a function.	`self.stream` is a readable stream (3). `self.sha` is a **sha** object.
6		`self` is a readable stream (11).	`self.model` is a **Ballot.Model**.
7		`self` is a readable stream (11).	`self.text` is a **Ballot.Text**.
8		`self` is a readable stream (11).	`self.audio` is a **Ballot.Audio**.
9		`self` is a readable stream (11).	`self.video` is a **Ballot.Video**.
10		`self.sha` is a **sha** (5). ⚠ if hash does not match.	The ballot definition file is complete and the loaded ballot definition data matches its concluding hash.
11	`length` is an int.		Returns the next `length` bytes of the stream (12, 14).
12		`self.stream` is a stream (5). `length` is an int (11).	`self.stream` is a stream (5), so `data` is a string.
13		`self.sha` is a **sha** (5). `data` is a string (12).	
14			
15			
16	`stream` is a readable stream.		
17		`stream` is a stream (16). `Group` is a class (20).	`self.groups` is a list of **Group** (136).
18		`stream` is a stream (16). `Page` is a class (31).	`self.pages` is a list of **Page** (136).
19		`stream` is a stream (16).	`allow_none` = `0`, so `self.timeout_ms` is an int (122).
20			
21	`stream` is a readable stream.		
22		`stream` is a stream (21).	`allow_none` = `0`, so `self.max_sels` is an int (122).
23		`stream` is a stream (21).	`allow_none` = `0`, so `self.max_chars` is an int (122).
24		`stream` is a stream (21).	`allow_none` = `0`, so `self.option_clips` is an int (122).
25		`stream` is a stream (21). `Option` is a class (31).	`self.options` is a list of **Option** (136).
26			
27	`stream` is a readable stream.		
28		`stream` is a stream (27).	`self.sprite_i` is an int (122).
29		`stream` is a stream (27).	`self.clip_i` is an int (122).
30		`stream` is a stream (27).	`allow_none` = `1`, so `self.writein_group_i` is int or `None` (122).
31			
32	`stream` is a readable stream.		
33		`stream` is a stream (32). `Binding` is a class (58).	`self.bindings` is a list of **Binding** (136).
34		`stream` is a stream (32). `State` is a class (38).	`self.states` is a list of **State** (136).
35		`stream` is a stream (32). `OptionArea` is a class (46).	`self.option_areas` is a list of **OptionArea** (136).
36		`stream` is a stream (32). `CounterArea` is a class (50).	`self.counter_areas` is a list of **CounterArea** (136).
37		`stream` is a stream (32). `ReviewArea` is a class (54).	`self.review_areas` is a list of **ReviewArea** (136).
38			
39	`stream` is a readable stream.		
40		`stream` is a stream (39).	`allow_none` = `0`, so `self.sprite_i` is an int (122).
41		`stream` is a stream (39). `Segment` is a class (78).	`self.segments` is a list of **Segment** (136).
42		`stream` is a stream (39). `Binding` is a class (58).	`self.bindings` is a list of **Binding** (136).
43		`stream` is a stream (39). `Segment` is a class (78).	`self.timeout_segments` is a list of **Segment** (136).
44		`stream` is a stream (39).	`allow_none` = `1`, so `self.timeout_page_i` is int or `None` (122).
45		`stream` is a stream (39).	`allow_none` = `0`, so `self.timeout_state_i` is an int (122).

6.2.2 Ballot.py

```
import sha

class Ballot:
    def __init__(self, stream):
        assert stream.read(8) == "Pvote\x00\x01\x00"
        [self.stream, self.sha] = [stream, sha.sha()]
        self.model = Model(self)
        self.text = Text(self)
        self.audio = Audio(self)
        self.video = Video(self)
        assert self.sha.digest() == stream.read(20)

    def read(self, length):
        data = self.stream.read(length)
        self.sha.update(data)
        return data

class Model:
    def __init__(self, stream):
        self.groups = get_list(stream, Group)
        self.pages = get_list(stream, Page)
        self.timeout_ms = get_int(stream, 0)

class Group:
    def __init__(self, stream):
        self.max_sels = get_int(stream, 0)
        self.max_chars = get_int(stream, 0)
        self.option_clips = get_int(stream, 0)
        self.options = get_list(stream, Option)

class Option:
    def __init__(self, stream):
        self.sprite_i = get_int(stream, 0)
        self.clip_i = get_int(stream, 0)
        self.writein_group_i = get_int(stream, 1)

class Page:
    def __init__(self, stream):
        self.bindings = get_list(stream, Binding)
        self.states = get_list(stream, State)
        self.option_areas = get_list(stream, OptionArea)
        self.counter_areas = get_list(stream, CounterArea)
        self.review_areas = get_list(stream, ReviewArea)

class State:
    def __init__(self, stream):
        self.sprite_i = get_int(stream, 0)
        self.segments = get_list(stream, Segment)
        self.bindings = get_list(stream, Binding)
        self.timeout_segments = get_list(stream, Segment)
        self.timeout_page_i = get_int(stream, 1)
        self.timeout_state_i = get_int(stream, 0)
```

	PRECONDITIONS	REASONS FOR VALIDITY	POSTCONDITIONS
46			
47	`stream` is a readable stream.		
48		`stream` is a stream (47).	`allow_none` = `0`, so `self.group_i` is an int (122).
49		`stream` is a stream (47).	`allow_none` = `0`, so `self.option_i` is an int (122).
50			
51	`stream` is a readable stream.		
52		`stream` is a stream (51).	`allow_none` = `0`, so `self.group_i` is an int (122).
53		`stream` is a stream (51).	`allow_none` = `0`, so `self.sprite_i` is an int (122).
54			
55	`stream` is a readable stream.		
56		`stream` is a stream (55).	`allow_none` = `0`, so `self.group_i` is an int (122).
57		`stream` is a stream (55).	`allow_none` = `1`, so `self.cursor_sprite_i` is int or `None` (122).
58			
59	`stream` is a readable stream.		
60		`stream` is a stream (59).	`allow_none` = `1`, so `self.key` is int or `None` (122).
61		`stream` is a stream (59).	`allow_none` = `1`, so `self.target_i` is int or `None` (122).
62		`stream` is a stream (59). `Condition` is a class (67).	`self.conditions` is a list of **Condition** (136).
63		`stream` is a stream (59). `Step` is a class (73).	`self.steps` is a list of **Step** (136).
64		`stream` is a stream (59). `Segment` is a class (78).	`self.segments` is a list of **Segment** (136).
65		`stream` is a stream (59).	`allow_none` = `1`, so `self.next_page_i` is int or `None` (122).
66		`stream` is a stream (59).	`allow_none` = `0`, so `self.next_state_i` is an int (122).
67			
68	`stream` is a readable stream.		
69		`stream` is a stream (68).	`self.predicate` is `0`, `1`, or `2` (127).
70		`stream` is a stream (68).	`allow_none` = `1`, so `self.group_i` is int or `None` (122).
71		`stream` is a stream (68).	`allow_none` = `0`, so `self.option_i` is an int (122).
72		`stream` is a stream (68).	`self.invert` is `0` or `1` (127).
73			
74	`stream` is a readable stream.		
75		`stream` is a stream (74).	`self.op` is `0`, `1`, `2`, `3`, or `4` (127).
76		`stream` is a stream (74).	`allow_none` = `1`, so `self.group_i` is int or `None` (122).
77		`stream` is a stream (74).	`allow_none` = `0`, so `self.option_i` is an int (122).
78			
79	`stream` is a readable stream.		
80		`stream` is a stream (79). `Condition` is a class (67).	`self.conditions` is a list of **Condition** (136).
81		`stream` is a stream (79).	`self.type` is `0`, `1`, `2`, `3`, or `4` (127).
82		`stream` is a stream (79).	`allow_none` = `0`, so `self.clip_i` is an int (122).
83		`stream` is a stream (79).	`allow_none` = `1`, so `self.group_i` is int or `None` (122).
84		`stream` is a stream (79).	`allow_none` = `0`, so `self.option_i` is an int (122).

Ballot.py (page 2 of 4)

```
class OptionArea:
    def __init__(self, stream):
        self.group_i = get_int(stream, 0)
        self.option_i = get_int(stream, 0)

class CounterArea:
    def __init__(self, stream):
        self.group_i = get_int(stream, 0)
        self.sprite_i = get_int(stream, 0)

class ReviewArea:
    def __init__(self, stream):
        self.group_i = get_int(stream, 0)
        self.cursor_sprite_i = get_int(stream, 1)

class Binding:
    def __init__(self, stream):
        self.key = get_int(stream, 1)
        self.target_i = get_int(stream, 1)
        self.conditions = get_list(stream, Condition)
        self.steps = get_list(stream, Step)
        self.segments = get_list(stream, Segment)
        self.next_page_i = get_int(stream, 1)
        self.next_state_i = get_int(stream, 0)

class Condition:
    def __init__(self, stream):
        self.predicate = get_enum(stream, 3)
        self.group_i = get_int(stream, 1)
        self.option_i = get_int(stream, 0)
        self.invert = get_enum(stream, 2)

class Step:
    def __init__(self, stream):
        self.op = get_enum(stream, 5)
        self.group_i = get_int(stream, 1)
        self.option_i = get_int(stream, 0)

class Segment:
    def __init__(self, stream):
        self.conditions = get_list(stream, Condition)
        self.type = get_enum(stream, 5)
        self.clip_i = get_int(stream, 0)
        self.group_i = get_int(stream, 1)
        self.option_i = get_int(stream, 0)
```

	PRECONDITIONS	REASONS FOR VALIDITY	POSTCONDITIONS
85			
86	`stream` is a readable stream.		
87		`stream` is a stream (87). `TextGroup` is a class (88).	`self.groups` is a list of **TextGroup** (136).
88			
89	`stream` is a readable stream.		
90		`stream` is a stream (89).	`self.name` is a string (131).
91		`stream` is a stream (89).	`self.writein` is `0` or `1` (127).
92		`stream` is a stream (89). `get_str` is a function (131).	`self.options` is a list of strings (136, 131).
93			
94	`stream` is a readable stream.		
95		`stream` is a stream (94).	`allow_none` = `0`, so `self.sample_rate` is an int (122).
96		`stream` is a stream (94). `Clip` is a class (97).	`self.clips` is a list of **Clip** (136).
97			
98	`stream` is a readable stream.		
99		`stream` is a stream (98). `allow_none` = `0`, so `get_int` returns an int (122).	`stream` is a stream (98), so `self.samples` is a string.
100			
101	`stream` is a readable stream.		
102		`stream` is a stream (101).	`allow_none` = `0`, so `self.width` is an int (122).
103		`stream` is a stream (101).	`allow_none` = `0`, so `self.height` is an int (122).
104		`stream` is a stream (101). `Layout` is a class (106).	`self.layouts` is a list of **Layout** (136).
105		`stream` is a stream (101). `Image` is a class (111).	`self.sprites` is a list of **Image** (136).
106			
107	`stream` is a readable stream.		
108		`Image` is a class (111). `stream` is a stream (107).	`self.screen` is a **Image**.
109		`stream` is a stream (107). `Rect` is a class (116).	`self.targets` is a list of **Rect** (136).
110		`stream` is a stream (107). `Rect` is a class (116).	`self.slots` is a list of **Rect** (136).
111			
112	`stream` is a readable stream.		
113		`stream` is a stream (112).	`allow_none` = `0`, so `self.width` is an int (122).
114		`stream` is a stream (112).	`allow_none` = `0`, so `self.height` is an int (122).
115		`stream` is a stream (112). `self.width` is an int (113). `self.height` is an int (114).	`stream` is a stream (112), so `self.pixels` is a string.
116			
117	`stream` is a readable stream.		
118		`stream` is a stream (117).	`allow_none` = `0`, so `self.left` is an int (122).
119		`stream` is a stream (117).	`allow_none` = `0`, so `self.top` is an int (122).
120		`stream` is a stream (117).	`allow_none` = `0`, so `self.width` is an int (122).
121		`stream` is a stream (117).	`allow_none` = `0`, so `self.height` is an int (122).

Ballot.py (page 3 of 4)

```
class Text:
    def __init__(self, stream):
        self.groups = get_list(stream, TextGroup)

class TextGroup:
    def __init__(self, stream):
        self.name = get_str(stream)
        self.writein = get_enum(stream, 2)
        self.options = get_list(stream, get_str)

class Audio:
    def __init__(self, stream):
        self.sample_rate = get_int(stream, 0)
        self.clips = get_list(stream, Clip)

class Clip:
    def __init__(self, stream):
        self.samples = stream.read(get_int(stream, 0)*2)

class Video:
    def __init__(self, stream):
        self.width = get_int(stream, 0)
        self.height = get_int(stream, 0)
        self.layouts = get_list(stream, Layout)
        self.sprites = get_list(stream, Image)

class Layout:
    def __init__(self, stream):
        self.screen = Image(stream)
        self.targets = get_list(stream, Rect)
        self.slots = get_list(stream, Rect)

class Image:
    def __init__(self, stream):
        self.width = get_int(stream, 0)
        self.height = get_int(stream, 0)
        self.pixels = stream.read(self.width*self.height*3)

class Rect:
    def __init__(self, stream):
        self.left = get_int(stream, 0)
        self.top = get_int(stream, 0)
        self.width = get_int(stream, 0)
        self.height = get_int(stream, 0)
```

	PRECONDITIONS	REASONS FOR VALIDITY	POSTCONDITIONS
122	stream is a stream. allow_none is 0 or 1.		Returns an int if allow_none = 0 (125, 126); otherwise returns an int or None (126). ⚠ if the next 4 bytes read do not represent an int or None, or if they represent None but allow_none = 0.
123		stream is a stream (122). ⚠ if read returns less than 4 bytes. list returns a list of 4 length-1 strings.	a, b, c, d are 1-byte strings.
124			
125		a, b, c, d are 1-byte strings (126).	An int from 0 to 2147483647 is returned.
126		allow_none is an int (122). a, b, c, d are strings (126).	None can be returned only if allow_none $\neq$ 0.
127	stream is a stream. cardinality is an int.		Returns an int n where 0 $\leq$ n $<$ cardinality (129, 130). ⚠ if the next 4 bytes read do not represent an int in this range.
128		stream is a stream (127).	allow_none = 0, so value is an int (122).
129		⚠ if value is out of range.	
130			An int is returned.
131	stream is a stream.		Returns a string containing only bytes between 32 and 125 inclusive (133–135). ⚠ if the stream does not yield a valid string.
132		stream is a stream (131). allow_none = 0 $\Rightarrow$ get_int returns an int (122).	str is a string.
133			ch is a 1-byte string.
134		⚠ if any byte in str falls outside the range from 32 to 125 inclusive.	
135			A string is returned.
136	stream is a stream.		Returns a list of instances of Class (124).
137		stream is a stream (136). allow_none = 0 $\Rightarrow$ get_int returns an int (122).	

Ballot.py (page 4 of 4)

```
def get_int(stream, allow_none):


    [a, b, c, d] = list(stream.read(4))

    if ord(a) < 128:
        return ord(a)*16777216 + ord(b)*65536 + ord(c)*256 + ord(d)
    assert allow_none and a + b + c + d == "\xff\xff\xff\xff"

def get_enum(stream, cardinality):

    value = get_int(stream, 0)
    assert value < cardinality
    return value

def get_str(stream):

    str = stream.read(get_int(stream, 0))
    for ch in list(str):
        assert 32 <= ord(ch) <= 125
    return str

def get_list(stream, Class):
    return [Class(stream) for i in range(get_int(stream, 0))]
```

	PRECONDITIONS	REASONS FOR VALIDITY	POSTCONDITIONS
1	`ballot` is a **Ballot**.		`ballot` satisfies the validity constraints in Section 2.7, or ⚠ .
2		`ballot.model` is a **Model** (1, Ballot:6). `ballot.video` is a **Video** (1, Ballot:9).	`groups` is a list of **Group** (Ballot:17). `sprites` is a list of **Image** (Ballot:105).
3			`option_sizes` is a list of *length*(`groups`) empty lists.
4			`char_sizes` is a list of *length*(`groups`) empty lists.
5		`model.groups` is a list (1, Ballot:6, Ballot:17). `text.groups` is a list (1, Ballot:7, Ballot:87). ⚠ if assertion fails.	`model.groups` and `text.groups` have the same length > 0.
6		`model.pages` is a list (1, Ballot:6, Ballot:18). `video.layouts` is a list (1, Ballot:9, Ballot:104). ⚠ if assertion fails.	`model.pages` and `video.layouts` have the same length > 0.
7		`model.pages` is a list (1, Ballot:6, Ballot:18).	`page_i` is a valid page index. `page` is the associated **Page** (Ballot:18).
8		`page_i` is a valid index in `model.pages` (7) ⇒ `page_i` is a valid index in `video.layouts` (6).	`layout` is a **Layout** (Ballot:104).
9		`page.bindings` is a list (7, Ballot:33).	`binding` is a **Binding** (Ballot:33).
10		`ballot` is a **Ballot** (1). `page` is a **Page** (7). `binding` is a **Binding** (9).	`binding` is a valid **Binding** for this `page` (71).
11		`page.states` is a list (7, Ballot:34). ⚠ if assertion fails.	`page.states` is nonempty.
12		`page.states` is a list (Ballot:34).	`state_i` is a valid state index. `state` is the associated **State** (Ballot:34).
13		⚠ if `state.sprite_i` is out of bounds. ⚠ if `state_i` is out of bounds. `sprites` is a list of **Image** (2). `layout.slots` is a list of **Rect** (8, Ballot:110).	`state.sprite_i` is a valid index in `video.sprites` (2). `state_i` is a valid index in `layout.slots`. The state's sprite at `state.sprite_i` has the same size as the state's slot (95).
14		`ballot` is a **Ballot** (2). `page` is a **Page** (7). `state.segments` is a list of **Segment** (12, Ballot:41).	Every element of `state.segments` is a valid **Segment** (78).
15		`state.bindings` is a list (Ballot:42).	`binding` is a **Binding** (Ballot:42).
16		`Ballot` is a **Ballot** (2). `page` is a **Page** (7). `binding` is a **Binding** (15).	`binding` is a valid **Binding** for this `page` (68).
17		`ballot` is a **Ballot** (2). `page` is a **Page** (7). `state.timeout_segments` is a list of **Segment** (12, Ballot:43).	Every element of `state.timeout_segments` is a valid **Segment** (78).
18		`ballot` is a **Ballot** (2). `timeout_page_i` is an int or `None` (Ballot:44). `timeout_state_i` is an int (Ballot:45).	Either `timeout_page_i` is `None`, or `timeout_page_i` and `timeout_state_i` are a valid page and state index (75).
19			`slot_i` is the index of the first remaining slot after slots have been assigned to states.
20		`page.option_areas` is a list (7, Ballot:35).	`area` is an **OptionArea** (7, Ballot:35).
21		`ballot` is a **Ballot** (2). `page` is a **Page** (7). `area` is a **OptionArea** (20).	`area.group_i` is an int (Ballot:48), so `area.group_i` is a valid group index and `area.option_i` is a valid option index in that group (89).
22		`option_sizes` is a list of lists (3). `area.group_i` is a valid group index (21). `layout.slots` is a list (Ballot:110). ⚠ if `slot_i` is out of bounds.	This page's layout contains a slot for this option area. `option_sizes` for this option area's group contains this option area's slot.
23		`slot_i` is an int (19, 23).	`slot_i` is the index of the next available slot.
24		`page.counter_areas` is a list (7, Ballot:36).	`area` is a **CounterArea** (7, Ballot:36).
25		⚠ if `area.group_i` is out of bounds. `groups` is a list of **Group** (2) ⇒ `groups[area.group_i].max_sels` is an int (Ballot:22).	`area.group_i` is a valid group index. `i` is an int from `0` to `max_sels` inclusive.
26		`area.sprite_i` is an int (Ballot:53). ⚠ if `area.sprite_i` + `i` is out of bounds. ⚠ if `slot_i` is out of bounds.	This page's layout contains a slot for this counter area. `sprite_i` through `sprite_i` + `max_sels` are valid sprite indices. These sprites all fit the counter area's slot (95).
27		`slot_i` is an int (19, 23, 27).	`slot_i` is the index of the next available slot.

6.2.3 verifier.py

```
def verify(ballot):
    [groups, sprites] = [ballot.model.groups, ballot.video.sprites]
    option_sizes = [[] for group in groups]
    char_sizes = [[] for group in groups]

    assert len(ballot.model.groups) == len(ballot.text.groups) > 0

    assert len(ballot.model.pages) == len(ballot.video.layouts) > 0

    for [page_i, page] in enumerate(ballot.model.pages):
        layout = ballot.video.layouts[page_i]

        for binding in page.bindings:
            verify_binding(ballot, page, binding)

        assert len(page.states) > 0

        for [state_i, state] in enumerate(page.states):
            verify_size(sprites[state.sprite_i], layout.slots[state_i])


            verify_segments(ballot, page, state.segments)

            for binding in state.bindings:
                verify_binding(ballot, page, binding)

            verify_segments(ballot, page, state.timeout_segments)

            verify_goto(ballot, state.timeout_page_i, state.timeout_state_i)

        slot_i = len(page.states)

        for area in page.option_areas:
            verify_option_ref(ballot, page, area)

            option_sizes[area.group_i].append(layout.slots[slot_i])


            slot_i = slot_i + 1

        for area in page.counter_areas:
            for i in range(groups[area.group_i].max_sels + 1):


                verify_size(sprites[area.sprite_i + i], layout.slots[slot_i])


            slot_i = slot_i + 1
```

	PRECONDITIONS	REASONS FOR VALIDITY	POSTCONDITIONS
28		`page.review_areas` is a list (7, Ballot:37).	`area` is a **ReviewArea** (7, Ballot:37).
29		⚠ if `area.group_i` is out of bounds. `groups` is a list of **Group** (2) ⇒ `groups[area.group_i].max_sels` is an int (Ballot:22).	`area.group_i` is a valid group index. `i` is an int from 0 to `max_sels` − 1 inclusive.
30		`option_sizes` is a list of lists (3). `area.group_i` is a valid group index (29). ⚠ if `slot_i` is out of bounds.	This page's layout contains an option slot for this review area. `option_sizes` for this review area's group contains all of the review area's option slots.
31		`slot_i` is an int (19, 23, 27, 31).	`slot_i` is the index of the next available slot.
32		`area.group_i` is a valid group index (29). `groups` is a list of **Group** (2) ⇒ `groups[area.group_i].max_chars` is an int (Ballot:23).	`j` is an int from 0 to `max_chars` − 1 inclusive.
33		`char_sizes` is a list of lists (4). `area.group_i` is a valid group index (29). ⚠ if `slot_i` is out of bounds.	This page's layout contains enough character slots for this review area. `char_sizes` for this review area's group contains all of the review area's character slots.
34		`slot_i` is an int (19, 23, 27, 31, 34).	`slot_i` is the index of the next available slot.
35		`area.cursor_sprite_i` is an int or `None` (Ballot:57).	
36		`option_sizes` is a list of lists (3). `area.group_i` is a valid group index (29). ⚠ if `area.cursor_sprite_i` is out of bounds.	`area.cursor_sprite_i` is `None`, or it is a valid sprite index and `option_sizes` for this review area's group contains the review area's cursor sprite.
37		`groups` is a list (2).	`group_i` is a valid group index. `group` is the associated **Group** (2).
38		`group.options` is a list (37, Ballot:25).	`option` is an **Option** (37, Ballot:25).
39		`option_sizes` is a list of lists (3). `group_i` is a valid group index (37). ⚠ if `option.sprite_i` is out of bounds.	`option.sprite_i` is a valid sprite index. `option_sizes` for this group contains the option's selected sprite.
40		`option_sizes` is a list of lists (3). `group_i` is a valid group index (37). ⚠ if `option.sprite_i` + `1` is out of bounds.	`option.sprite_i` + `1` is a valid sprite index. `option_sizes` for this group contains the option's unselected sprite.
41		`group` is a **Group** (37). ⚠ if assertion fails.	`group.option_clips` is at least 1.
42		`audio.clips` is a list (1, Ballot:8, Ballot:96). ⚠ if `option.clip_i` + `group.option_clips` − `1` is out of bounds.	The integers `option.clip_i` through `option.clip_i` + `group.option_clips` − `1` are all valid clip indices.
43		`option` is a **Option** (38).	
44		`groups` is a list of **Group** (2). `option` is a **Option** (38). ⚠ if `option.writein_group_i` is out of bounds.	`option.writein_group_i` is `None` (43), or it is a valid group index and `writein_group` is the associated **Group**.
45		`writein_group` is a **Group** (44). ⚠ if assertion fails.	`max_chars` = 0 for this option's write-in group.
46		`writein_group` is a **Group** (44). `group` is a **Group**. ⚠ if assertion fails.	`max_sels` for this option's write-in group matches `max_chars` for this option's parent group. `group` cannot be the write-in group for any option (43–46).
47		`writein_group.options` is a list (44, Ballot:25).	`option` is an **Option** (44, Ballot:25).
48		`char_sizes` is a list of lists (4). `group_i` is a valid group index (37). ⚠ if `option.sprite_i` is out of bounds.	After loop: all options in the write-in group have valid sprite indices. `char_sizes` for the parent group contains all their sprites.
49		`group_i` is a valid group index (37). `option_sizes[group_i]` is a list (3).	`object` is a **Slot** or a **Sprite** (22, 30, 36, 39, 40).
50		`group_i` is a valid group index (37). Each of `object` and `option_sizes[group_i][0]` is a **Slot** or **Sprite** (49).	After loop: all the slots and sprites for options in this group have the same size.
51		`group_i` is a valid group index (37). `char_sizes[group_i]` is a list (3).	`object` is a **Slot** or a **Sprite** (33, 48).
52		`group_i` is a valid group index (37). Each of `object` and `char_sizes[group_i][0]` is a **Slot** or **Sprite** (51).	After loop: all the slots and sprites for characters in write-in options in this group have the same size.

verifier.py (page 2 of 4)

```
        for area in page.review_areas:
            for i in range(groups[area.group_i].max_sels):
                option_sizes[area.group_i].append(layout.slots[slot_i])
                slot_i = slot_i + 1
                for j in range(groups[area.group_i].max_chars):
                    char_sizes[area.group_i].append(layout.slots[slot_i])
                    slot_i = slot_i + 1
            if area.cursor_sprite_i != None:
                option_sizes[area.group_i].append(sprites[area.cursor_sprite_i])

    for [group_i, group] in enumerate(groups):
        for option in group.options:
            option_sizes[group_i].append(sprites[option.sprite_i])
            option_sizes[group_i].append(sprites[option.sprite_i + 1])
            assert group.option_clips > 0
            ballot.audio.clips[option.clip_i + group.option_clips - 1]
            if option.writein_group_i != None:
                writein_group = groups[option.writein_group_i]
                assert writein_group.max_chars == 0
                assert writein_group.max_sels == group.max_chars > 0
                for option in writein_group.options:
                    char_sizes[group_i].append(sprites[option.sprite_i])
        for object in option_sizes[group_i]:
            verify_size(object, option_sizes[group_i][0])
        for object in char_sizes[group_i]:
            verify_size(object, char_sizes[group_i][0])
```

	PRECONDITIONS	REASONS FOR VALIDITY	POSTCONDITIONS
53		`text.groups` is a list (1, Ballot:7, Ballot:87).	Since `text.groups` and `model.groups` have the same length (5), `group_i` is a valid group index and `group` is the associated **TextGroup** (1, Ballot:7, Ballot:87).
54		⚠ if assertion fails.	`group.name` is no more than 50 bytes long.
55		`group_i` is a valid group index (53). `group.options` is a list (53, Ballot:92). `groups[group_i].options` is a list (Ballot:25). ⚠ if assertion fails.	This **TextGroup** `group` has the same number of options as its corresponding **Group** in `model.groups`.
56		`group.options` is a list (53, Ballot:92).	`option` is a string (53, Ballot:92).
57		⚠ if assertion fails.	`option` is no more than 50 bytes long.
58		`audio.clips` is a list (1, Ballot:8, Ballot:96).	`clip` is a **Clip** (1, Ballot:8, Ballot:96).
59		`clip.samples` is a string (60, Ballot:99). ⚠ if assertion fails.	`clip` has a nonempty string of `samples`.
60		`video` is a **Video** (1, Ballot:9) ⇒ `width` and `height` are ints (Ballot:102, Ballot:103). ⚠ if assertion fails.	`video` has a nonzero `width` and nonzero `height`.
61		`video.layouts` is a list (1, Ballot:9, Ballot:104).	`layout` is a **Layout** (1, Ballot:9, Ballot:104).
62		`layout.screen` is an **Image** (61, Ballot:108). `video` is a **Video** (1, Ballot:9).	`layout.screen` has the same size as `video`.
63		`layout.targets` is a list of **Rect** (61, Ballot:109). `layout.slots` is a list of **Rect** (61, Ballot:110).	The sum of lists is a list of **Rect**, so `rect` is a **Rect**.
64		`rect` is a **Rect** (63). `video` is a **Video** (1, Ballot:8).	`rect` does not extend beyond the right edge of the screen.
65		`rect` is a **Rect** (63). `video` is a **Video** (1, Ballot:8).	`rect` does not extend beyond the bottom edge of the screen.
66		`video.sprites` is a list (1, Ballot:8, Ballot:105).	`sprite` is an **Image** (1, Ballot:8, Ballot:105).
67		`sprite` is an **Image**.	`sprite` has a nonzero width and height and the correct amount of pixel data for an image with size `width` × `height`.
68	`ballot` is a **Ballot**. `page` is a **Page**. `binding` is a **Binding**.		`binding` is a valid **Binding**, or ⚠.
69		`binding.conditions` is a list (68, Ballot:62).	`condition` is a **Condition** (68, Ballot:62).
70		`ballot` is a **Ballot** (68). `page` is a **Page** (68). `condition` is a **Condition** (69).	After loop: every element of `binding.conditions` is a valid **Condition**.
71		`binding.steps` is a list (68, Ballot:63).	`step` is a **Step** (68, Ballot:63).
72		`ballot` is a **Ballot** (68). `page` is a **Page** (68). `step` is a **Step** (71).	`step.group_i` and `step.option_i` form a valid option reference.
73		`ballot` is a **Ballot** (68). `page` is a **Page** (68). `binding.segments` is a list of **Segment** (68, Ballot:64).	All the segments in `binding.segments` are valid for this page (78).
74		`ballot` is a **Ballot** (68). `binding.next_page_i` is an int or `None` (68, Ballot:65). `binding.next_state_i` is an int (68, Ballot:66).	Either `next_page_i` is `None`, or `next_page_i` is a valid page index and `next_state_i` is a valid state index for that page (75).
75	`ballot` is a **Ballot**. `page_i` and `state_i` are ints.		Either `page_i` is `None` (76), or `page_i` is a valid page index and `state_i` is a valid state index for that page (77), or ⚠.
76			
77		`model.pages` is a list of **Page** (75, Ballot:6, Ballot:18). ⚠ if `page_i` is out of bounds. ⚠ if `state_i` is out of bounds.	`page_i` is a valid page index and `state_i` is a valid state index for that page.
78	`ballot` is a **Ballot**. `page` is a **Page**. `segments` is a list of **Segment**.		Every segment in `segments` is a valid **Segment**.
79		`segments` is a list (78).	`segment` is a **Segment** (78).
80		`segment.conditions` is a list (79, Ballot:80).	`condition` is a **Condition** (79, Ballot:80).
81		`ballot` is a **Ballot** (78). `page` is a **Page** (78). `condition` is a **Condition** (80).	After loop: every element of `segment.conditions` is a valid **Condition**.
82		`audio.clips` is a list of **Clip** (1, Ballot:8, Ballot:96).	`segment.clip_i` is a valid clip index.
83			
84		`ballot` is a **Ballot** (78). `page` is a **Page** (78). `segment` is a **Segment** (79).	The segment's `group_i` and `option_i` form a valid option reference.
85			`group` is the referenced **Group**.
86		`segment.clip_i` is an int (79, Ballot:82). `group.option_clips` is an int (84, Ballot:24).	If `type` is 1 or 2, `segment.clip_i` is a valid option clip offset for the referenced group (83, 85).
87			
88		`audio.clips` is a list (1, Ballot:8, Ballot:96). `segment.clip_i` is an int (79, Ballot:82). `group.max_sels` is an int (84, Ballot:22).	If `type` is 3 or 4, `segment.clip_i` + `max_sels` is a valid clip index for the referenced group (83, 87).

verifier.py (page 3 of 4)

```
    for [group_i, group] in enumerate(ballot.text.groups):
        assert len(group.name) <= 50
        assert len(group.options) == len(groups[group_i].options)
        for option in group.options:
            assert len(option) <= 50

    for clip in ballot.audio.clips:
        assert len(clip.samples) > 0

    assert ballot.video.width*ballot.video.height > 0

    for layout in ballot.video.layouts:
        verify_size(layout.screen, ballot.video)

        for rect in layout.targets + layout.slots:
            assert rect.left + rect.width <= ballot.video.width
            assert rect.top + rect.height <= ballot.video.height
    for sprite in ballot.video.sprites:
        assert len(sprite.pixels) == sprite.width*sprite.height*3 > 0

def verify_binding(ballot, page, binding):

    for condition in binding.conditions:
        verify_option_ref(ballot, page, condition)

    for step in binding.steps:
        verify_option_ref(ballot, page, step)
    verify_segments(ballot, page, binding.segments)

    verify_goto(ballot, binding.next_page_i, binding.next_state_i)

def verify_goto(ballot, page_i, state_i):
    if page_i != None:
        ballot.model.pages[page_i].states[state_i]

def verify_segments(ballot, page, segments):
    for segment in segments:
        for condition in segment.conditions:
            verify_option_ref(ballot, page, condition)

        ballot.audio.clips[segment.clip_i]
        if segment.type in [1, 2, 3, 4]:
            group = verify_option_ref(ballot, page, segment)
            if segment.type in [1, 2]:
                assert segment.clip_i < group.option_clips
            if segment.type in [3, 4]:
                ballot.audio.clips[segment.clip_i + group.max_sels]
```

	PRECONDITIONS	REASONS FOR VALIDITY	POSTCONDITIONS
89	`ballot` is a **Ballot**. `page` is a **Page**. `object` is an **OptionArea**, **Condition**, **Step**, or **Segment**.		`object.group_i` and `object.option_i` form a valid direct or indirect option reference, or ⚠. Returns the referenced **Group** (92, 94).
90			
91		`page.option_areas` is a list of **OptionArea** (7, Ballot:35). ⚠ if `object.option_i` is out of bounds.	If `group_i` is `None`, then `option_i` is a valid option area index for `page`. `area` is the associated **OptionArea**.
92		`model.groups` is a list (1, Ballot:6, Ballot:17). `area` is an **OptionArea** (91). ⚠ if `group_i` is out of bounds.	The referenced option area's group is returned.
93		`model.groups` is a list (1, Ballot:6, Ballot:17). ⚠ if `object.group_i` is out of bounds. `groups[object.group_i].options` is a list (25). ⚠ if `object.option_i` is out of bounds.	`group_i` is a valid group index and `option_i` is a valid option index in that group.
94		`model.groups` is a list (1, Ballot:6, Ballot:17). `group_i` is a valid group index (93).	The referenced group is returned.
95	`a` is a **Video**, **Image**, or **Rect**. `b` is a **Video**, **Image**, or **Rect**.		`a` and `b` have equal `width` and equal `height`.
96		`a.width` and `b.width` are ints (95, Ballot:102, Ballot:113, Ballot:120). `a.height` and `b.height` are ints (95, Ballot:103, Ballot:114, Ballot:121).	

verifier.py (page 4 of 4)

```
def verify_option_ref(ballot, page, object):

    if object.group_i == None:
        area = page.option_areas[object.option_i]

        return ballot.model.groups[area.group_i]

    ballot.model.groups[object.group_i].options[object.option_i]



    return ballot.model.groups[object.group_i]

def verify_size(a, b):
    assert a.width == b.width and a.height == b.height
```

INVARIANTS In an initialized **Audio.Audio** object:

INV1. `self.clips` is a list of **Sound** the same length as `ballot.audio.clips` (7).
INV2. `self.queue` is a list (8, 18).
INV3. Each element of `self.queue` is a valid index into `ballot.audio.clips` (10).
INV4. Each element of `self.queue` is a valid index into `self.clips` (by **INV1** and **INV3**.)
INV5. `self.playing` is an int (8, 14).
INV6. `self.playing` ≠ 0 ⇔ either audio is currently playing, or it has just been stopped and an `AUDIO_DONE` event is pending. (Audio is only started (16) immediately after setting `self.playing` (14). `self.playing` is updated (14) when audio stops (main:22–23).)
INV7. `self.queue` is not empty ⇒ `self.playing` ≠ 0. (Only `play()` adds to the queue (10); after doing so, it immediately updates `self.playing` (11, 12, 14).)

	PRECONDITIONS	REASONS FOR VALIDITY	POSTCONDITIONS
1			`pygame` is bound to the Pygame module.
2	`pygame.USEREVENT` is an int.		`AUDIO_DONE` is an int.
3			
4	`audio` is a **Ballot.Audio** object.		
5			Since `sample_rate` is an int (Ballot:123), `rate` is an int.
6		`rate` is an int (5). ⚠ if `rate` is not accepted as a valid sample rate.	
7		`audio` is a **Ballot.Audio** (4) ⇒ `audio.clips` is a list of **Ballot.Clip** (Ballot:49) ⇒ `clip.samples` is a string.	`self.clips` is a list of **Sound** with the same length as `audio.clips`.
8			
9	`clip_i` is a valid index into `ballot.audio.clips`.		
10		**INV2**.	
11		**INV5**.	
12			
13			
14		**INV2**.	
15		**INV2**.	
16		**INV4**. `self.queue` is nonempty (15). **INV1**. The audio system has been initialized (6). The `play()` method of **Sound** returns a **Channel**. `AUDIO_DONE` is an int (2).	
17			
18			
19		The audio system has been initialized (6). If audio is playing, this will trigger `AUDIO_DONE`.	
20	`rate` is an int. `data` is a string.		Returns a **Sound** object for the given audio data (24).
21			
22		`rate` is an int (20). `put_int` returns a string (27).	`fmt` is a string (21).
23		`fmt` is a string (22). `data` is a string (20).	
24		`file` is a string (23). `Buffer` yields a readable stream object (33). See Appendix C to verify that the WAV file passed to `Sound` is well-formed.	
25	`type` and `contents` are strings.		Returns a RIFF chunk as a string (26).
26		`type` and `contents` are strings (25). `len` returns an int. `put_int` returns a string (27).	
27	`n` is an int.		Returns the big-endian serialization of `n` (29).
28			`n` is an int (27), so `a`, `b`, `c`, and `d` are integers.
29		`a`, `b`, `c`, and `d` are integers (28).	
30			
31	`data` is a string.		
32			`self.data` is a string. `self.pos` is an int.
33	`length` is an int. The caller will not read past the end of `data`.		Returns the next `length` bytes of the buffer (35).
34		`self.pos` is an int (32, 34).	
35		`self.pos - length` is nonnegative (34). `self.data` is a string (32). `self.pos` is no larger than the length of `self.data` (33).	

6.2.5 Audio.py

```
import pygame
AUDIO_DONE = pygame.USEREVENT

class Audio:
    def __init__(self, audio):
        rate = audio.sample_rate
        pygame.mixer.init(rate, -16, 0)
        self.clips = [make_sound(rate, clip.samples) for clip in audio.clips]
        [self.queue, self.playing] = [[], 0]

    def play(self, clip_i):
        self.queue.append(clip_i)
        if not self.playing:
            self.next()

    def next(self):
        self.playing = len(self.queue)
        if len(self.queue):
            self.clips[self.queue.pop(0)].play().set_endevent(AUDIO_DONE)

    def stop(self):
        self.queue = []
        pygame.mixer.stop()

def make_sound(rate, data):
    [comp_channels, sample_size] = ["\x01\x00\x01\x00", "\x02\x00\x10\x00"]
    fmt = comp_channels + put_int(rate) + put_int(rate*2) + sample_size
    file = chunk("RIFF", "WAVE" + chunk("fmt ", fmt) + chunk("data", data))
    return pygame.mixer.Sound(Buffer(file))

def chunk(type, contents):
    return type + put_int(len(contents)) + contents

def put_int(n):
    [a, b, c, d] = [n/16777216, n/65536, n/256, n]
    return chr(d % 256) + chr(c % 256) + chr(b % 256) + chr(a % 256)

class Buffer:
    def __init__(self, data):
        [self.data, self.pos] = [data, 0]

    def read(self, length):
        self.pos = self.pos + length
        return self.data[self.pos - length:self.pos]
```

INVARIANTS In an initialized **Video.Video** object:

INV1. `self.surface` is a **Surface** (7).
INV2. `self.layouts` is a list of **Layout** (8).
INV3. `self.screens` is a list of Pygame **Image** objects the same length as `video.layouts` (9).
INV4. `self.sprites` is a list of Pygame **Image** objects the same length as `video.sprites` (10).
INV5. `self.layout` is a **Layout** (13).

	PRECONDITIONS	REASONS FOR VALIDITY	POSTCONDITIONS
1			`pygame` is bound to the Pygame module.
2	`im` is a **Ballot.Image**.		Converts raw pixel data to a Pygame **Image**.
3		`im` is a **Ballot.Image** (2). `im.pixels` has length `im.width` × `im.height` × `3` (verifier:67). `im.width` and `im.height` are nonzero (verifier:67).	`pygame.image.fromstring` returns a Pygame **Image**.
4			
5	`video` is a **Ballot.Video**.		
6		`video` is a **Ballot.Video** (5).	`size` is a list of two ints (Ballot:102, Ballot:103).
7		⚠ if `size` is not accepted as a valid resolution.	**INV1**.
8			**INV2**.
9		`video.layouts` is a list of **Layout** (Ballot:104) ⇒ `layout.screen` is a **Ballot.Image** (Ballot:108).	**INV3**.
10		`video.sprites` is a list of **Ballot.Image** (Ballot:105).	**INV4**.
11			
12	`layout_i` is a valid layout index.		`self.layout` is the referenced **Layout** and its screen is displayed.
13		`layout_i` is a valid layout index (12).	
14		`layout_i` is the valid index of a Pygame **Image** in `self.screens` (**INV3**). The **Image** has size equal to the screen resolution (verifier:62).	
15	`sprite_i` is a valid sprite index. `slot_i` is a valid slot index in the current layout. The sprite and slot have the same size.		The sprite is pasted into the slot.
16		`slot_i` is a valid slot index (15).	`slot` is a **Rect** (Ballot:110).
17		`sprite_i` is the valid index of a Pygame **Image** in `self.sprites` (**INV4**). The pasted sprite fits within screen bounds (verifier:64–65).	
18	`x` and `y` are ints.		Returns the index of the current layout's first target containing (`x`, `y`), or `None` (22).
19		`self.layout.targets` is a list (**INV5**).	By **INV5**, `i` is a valid target index and `target` is a **Target**.
20			
21			
22			`i` is returned if the target contains (`x`, `y`).

6.2.6 Video.py

```
import pygame

def make_image(im):
    return pygame.image.fromstring(im.pixels, (im.width, im.height), "RGB")

class Video:
    def __init__(self, video):
        size = [video.width, video.height]
        self.surface = pygame.display.set_mode(size, pygame.FULLSCREEN)
        self.layouts = video.layouts
        self.screens = [make_image(layout.screen) for layout in video.layouts]

        self.sprites = [make_image(sprite) for sprite in video.sprites]
        self.goto(0)

    def goto(self, layout_i):

        self.layout = self.layouts[layout_i]
        self.surface.blit(self.screens[layout_i], [0, 0])

    def paste(self, sprite_i, slot_i):

        slot = self.layout.slots[slot_i]
        self.surface.blit(self.sprites[sprite_i], [slot.left, slot.top])

    def locate(self, x, y):

        for [i, target] in enumerate(self.layout.targets):
            if target.left <= x and x < target.left + target.width:
                if target.top <= y and y < target.top + target.height:
                    return i
```

INVARIANTS In initialized Printer objects:

INV1. `self.text` is a **Ballot.Text** (3).

INV2. Wherever `line` is bound, the length of `line` never exceeds 60 bytes. (When `line` is lengthened (15), it length increases by at most 51 bytes (verifier:57). It is cleared immediately preceding this lengthening (14) if the lengthening would have increased its length to more than 60 bytes (12).

	PRECONDITIONS	REASONS FOR VALIDITY	POSTCONDITIONS
1			
2	`text` is a **Text**.		
3			**INV1**.
4	`selections` is a list of *length*(`model.groups`) lists, where each list contains only valid option indices for each group.		The selections are printed out.
5		`selections` is a list of lists (4).	`group_i` is a valid group index and `selection` is a list of valid option indices in that group (4).
6		`group_i` is a valid index into `self.text.groups` (5, **INV1**, verifier:5).	`group` is a **TextGroup** (Ballot:87).
7		`group.writein` is an int (7, Ballot:91).	
8		`selection` is a list (5).	
9		`group.name` is a string (6, Ballot:90).	
10			
11		`selection` is a list (5).	`group.options` has the same length as `model.groups[group_i].options` (verifier:55), so `option_i` is a valid index into `group.options` (5).
12		`group.options` is a list of strings (6, Ballot:92). `option_i` is a valid index into `group.options` (11).	
13		`line` is a string (10, 14, 15).	
14			
15		`line` is a string (10, 14, 15). `group.options` is a list of strings (6, Ballot:92). `option_i` is a valid index into `group.options` (11).	
16		`line` is a string (10, 14, 15).	
17			
18		`selection` is a list (5).	
19		`group.name` is a string (6, Ballot:90).	
20			`option_i` is a valid option index for group `group_i` (5). `group.options` is a list of strings (Ballot:92), so `option` is a string.
21		`option_i` is an int (20). `selection` is a list (5).	
22		`group.name` is a string (6, Ballot:90).	
23			
24			
25			

6.2.7 Printer.py

```
class Printer:
    def __init__(self, text):
        self.text = text

    def write(self, selections):
        for [group_i, selection] in enumerate(selections):
            group = self.text.groups[group_i]
            if group.writein:
                if len(selection):
                    print "\n+ " + group.name
                    line = ""
                    for option_i in selection:
                        if len(line) + len(group.options[option_i]) + 1 > 60:
                            print "= " + line
                            line = ""
                        line = line + group.options[option_i] + "~"
                    print "= " + line
            else:
                if len(selection):
                    print "\n* " + group.name
                    for [option_i, option] in enumerate(group.options):
                        if option_i in selection:
                            print "- " + option
                else:
                    print "\n* " + group.name + " ~ NO SELECTION"
        print "\n~\f"
```

Chapter 7

Correctness claims

7.1 No negative integers

A negative integer literal occurs only once in Pvote: Audio.py, line 6, as a constant supplied to `pygame.mixer.init`. The unary negation operator is never used, and the binary subtraction operator is used exactly twice in Pvote:

- `length` is subtracted from `self.pos` (Audio:35), which the preceding line ensures is greater than or equal to `length`.
- `1` is subtracted from `group.option_clips` (verifier:42), which the preceding line ensures is greater than or equal to `1`.

Therefore, no computations ever result in negative numbers and no variables ever take on negative values.

7.2 Navigator starts on page 0 in state 0

Initialization of the **Navigator** always calls `self.goto(0, 0)` (Navigator:10). In the `goto` method, `page_i` is 0 (not `None`) and `self.page_i` is `None` (which cannot equal an integer), so it proceeds to set `self.page_i` and `self.state_i` to 0, and set `self.page` and `self.state` to `model.pages[0]` and its `states[0]` respectively. Therefore, the navigator always starts on page 0 in state 0.

7.3 Ballot is committed on the last page

Only one **Printer** is ever instantiated (main:8). This `printer` is immediately passed to `navigator` and never referenced again in main.py. The **Navigator** assigns the incoming `printer` to `self.printer`, which is only ever referenced once (Navigator:14). This line can only be executed when `page_i` + `1` is equal to `len(self.model.pages)`, that is, on the last page.

Also, there is only one assignment to `self.page` anywhere in the **Navigator** (Navigator:15), which is immediately preceded by a call to `self.printer.write` if transitioning to the last page. Thus, any transition to the last page must call `self.printer.write`.

Therefore, the Navigator always commits the ballot, and only commits the ballot, when it transitions to the last page.

7.4 Overvoting is impossible

There is only one place where options are ever added to the current selection (Navigator:82). The immediately preceding line ensures that the group is not full (the number of selections is less than `max_sels`) at that point. Therefore, the number of selections in any group cannot exceed `max_sels` for that group.

7.5 Contest options cannot be selected twice

There is only one place where options are added to the selection (Navigator:82). This can only be reached with a `step.op` equal to `OP_ADD` or `OP_APPEND`. In the case of `OP_ADD`, this line cannot be reached if the option to be added is already selected. Therefore, no option can appear twice in a group's selection list unless `OP_APPEND` is used. The ballot definition can be examined to confirm that `OP_APPEND` is used only in write-in groups but never in contest groups.

7.6 Bounded function call depth

Figure 7.1 depicts all the ways Pvote routines can be called during the processing of an event received by the main event loop.

The call to `review` in the `review` method (Navigator:43) is the only recursive call. The recursive call passes a write-in group as the `group_i` argument to `review`. Since a write-in group cannot have any options that themselves have write-in groups (verifier:45–46), recursion cannot proceed more than one level deep.

The call graph otherwise contains no cycles. Inspection of the call graph shows that a call to `play` yields a depth of at most 4 calls; thus a call to `goto` yields a depth of at most 5 calls; thus a call to `invoke` yields a depth of at most 6 calls. Therefore, the processing of a single event cannot exceed a depth of 7 calls.

7.7 Bounded iteration

Pthin has two looping constructs, `while` and `for`. The call graph shown in Figure 7.1 is annotated with bubbles that mark every use of these constructs.

Observe that `invoke` and `goto` can each be called at most once, and `play` can be called at most twice. The number of iterations of any operation that can occur during the processing of a single event is therefore bounded by one of:

- a number of bindings × a number of conditions
- a number of targets
- a number of steps
- a number of groups × a number of selections
- a number of option areas
- a number of counter areas
- a number of review areas × a number of selections × a number of selections
- 2 × a number of segments × a number of conditions
- 2 × a number of segments × a number of selections × a number of selections

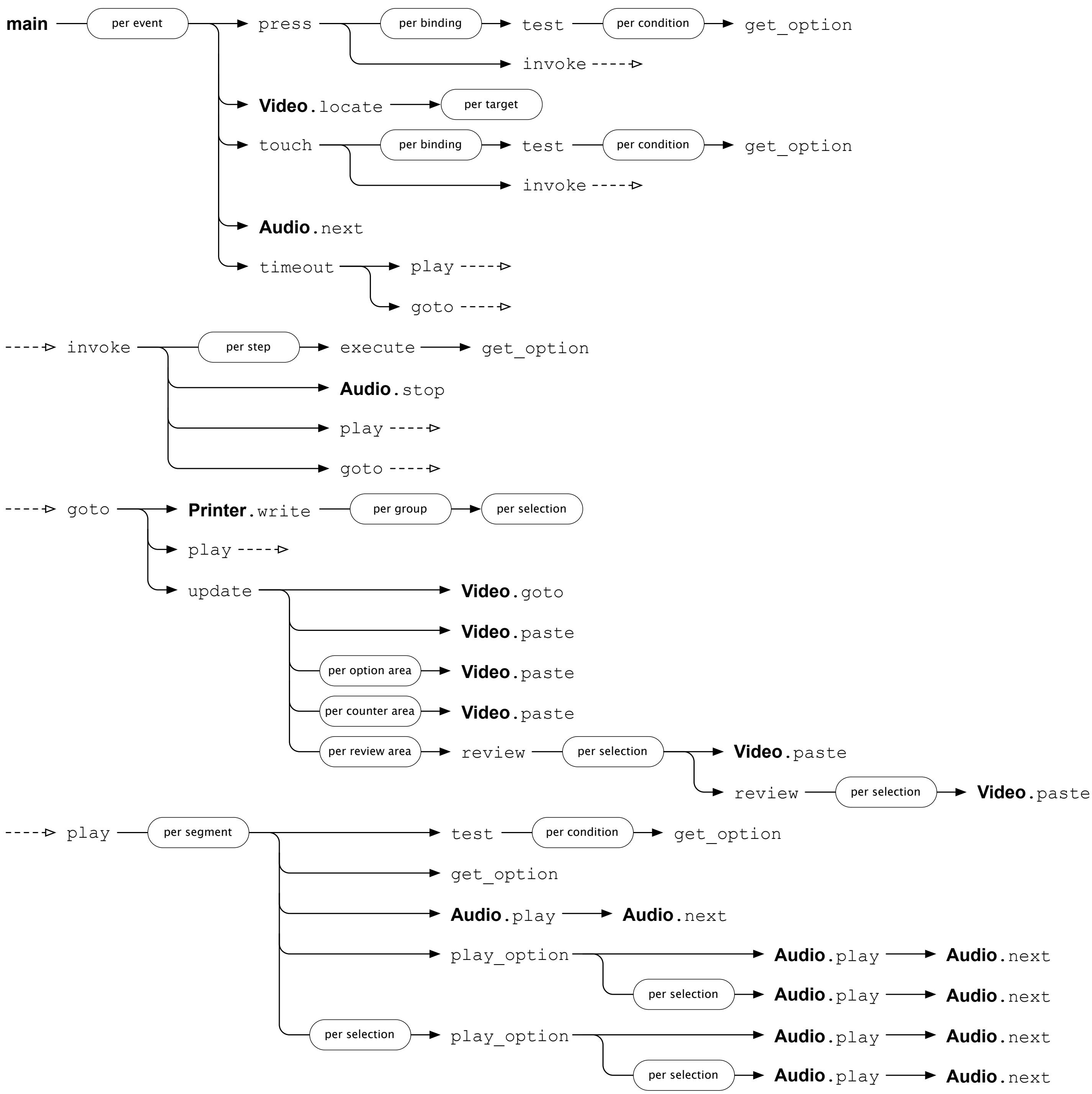

Figure 7.1: Call graph among Pvote routines, rooted from the event loop. Every use of iteration is indicated with a round bubble. Where a module name is not specified, the method belongs to **Navigator**.

7.8 At most one audio clip plays at a time

The Pygame audio system is capable of playing multiple audio clips mixed together, but we wish to avoid ever letting this happen.

To prove that Pvote only ever plays one clip at a time, imagine a token that conveys the permission to play audio. Only one token exists, and it is passed back and forth between Pvote's **Audio** object and the Pygame audio system. The `playing` field of the **Audio** object represents whether it possesses the token. If `playing` = 0, then Pvote has the token; otherwise, Pygame has the token. Starting playback of an audio clip with the `play` method of a **Sound** object should pass the token from Pvote to Pygame. Receiving an `AUDIO_DONE` event from Pygame should pass the token from Pygame to Pvote.

Now let us verify that Pvote actually upholds this model. The Pvote **Audio** object initializes `self.playing` to 0, so it initially holds the token (Audio:8). Sound playback is only ever initiated by `next` method (Audio:16), which can be called in exactly two ways. Either `play` calls `next` (Audio:12), which can only happen when `self.playing` is 0 (i. e. Pvote already holds the token); or the main loop calls `next` upon receiving an `AUDIO_DONE` event (main:22–23) (i. e. Pygame has just passed the token to Pvote). Thus, Pvote only initiates playback when it holds the token.

We can also confirm that Pvote accurately tracks whether it holds the token. Sound playback can be initiated only when `self.queue` is not empty (Audio:15–16), which means `self.playing` must be set to a nonzero value (Audio:14). Thus, Pvote relinquishes the token when it initiates audio playback.

`self.playing` is set only upon a call to `next` (Audio:14). If `next` is called by `play`, then Pvote must already have the token (Audio:11). The only other possibility is that `next` is called by the main loop due to the receipt of an `AUDIO_DONE` event. Thus, Pvote can acquire the token only when Pygame notifies it that audio playback has stopped.

7.9 Timeout occurs after `timeout_ms` ms of idle silence

If `timeout_ms` ms passes with no events and no audio output, the Pygame timer will send a `TIMER_DONE` event (main:12–13) and `audio.playing` will be zero,. so the timeout behaviour will be triggered (main:24–25).

The Pygame timer is set to run only while Pvote is waiting for an event (main:12, main:14), so a `TIMER_DONE` event can only occur when `timeout_ms` ms has passed with no other events occurring. In particular, this includes `AUDIO_DONE` events, so no sound can have finished playing during the last `timeout_ms` ms. For the timeout behaviour to be triggered there must not be a sound currently playing (main:24). Therefore, the audio output must have been silent for the last `timeout_ms` ms.

7.10 Ballot definition is never changed

Inspection of the code shows that, during event processing, assignments are never made and methods are never called on objects in the ballot definition.

7.11 Responsibilities established

R1. Never abort during a voting session.

Termination of a Pthin program can occur in the following ways:

1. Execution reaches the end of the main program.
2. Illegal types of operands are supplied to an expression.
3. A precondition for an expression is violated.
4. A precondition for a library routine is violated.
5. An incorrect number of arguments is passed to a function or method.
6. An assertion fails.
7. Memory is exhausted.

Cause 1: Due to the infinite event loop, execution never reaches the end of the main program (main:10).

Causes 2 through 6: The annotations in the source code identify all the possible places where these kinds of errors can occur. These appear in the ballot loader, the verifier, and the initialization routines for the audio driver and video driver, all of which execute on startup before the voting session begins. After these routines have successfully completed executing, it has been established (mainly by the verifier) that these kinds of errors cannot occur at a later point.

Cause 7: By the memory management rules in section 3.8, memory stays allocated only by binding values to names, placing values in lists, creating cyclic reference chains, or passing values as arguments. Static analysis of the program can determine the total number of global names, local names, and field names used, so the space used by bindings is bounded.

Only strings and lists have variable size. Strings are never manipulated during event processing except when printing the ballot; we will establish for R10 that **Printer** never constructs a string longer than 70 bytes. Lists are made longer only in two places: the `execute` method in **Navigator** appends to the current selection (Navigator:82), and the `play` method in **Audio** appends to the play queue (Audio:10). The lengths of selection lists are bounded by `max_sels` (Navigator:81). Examination of the call graph (Figure 7.1) shows that **Audio**`.play` is only called by **Navigator**`.play`, which can only be called by `timeout`, `invoke`, or `goto`, which is itself called only by `timeout` or `invoke`. `timeout` can only be called when the play queue is empty (main:24–25), and `invoke` clears the play queue before adding anything to it (Navigator:72). Since we can see that the number of calls to **Audio**`.play` from **Navigator**`.play` is bounded, the length of the play queue is bounded.

Cyclic reference chains can only created via list containment or object fields. During event processing, lists or objects are never placed into lists, and assignments to object fields (Audio:14, Audio:18, Navigator:15, Navigator:16, Video:13) always assign integers, empty lists, or elements of the ballot definition, which are never mutated.

Finally, Section 7.6 established a bound on the depth of the call stack, and the size of each stack frame is predetermined by the number of arguments and local names in each function or method.

Thus, Pvote's maximum memory usage is determined by the ballot definition.

R2. Remain responsive during a voting session.

For an interactive program, "responsive" means that the program is always ready to process user input within a reasonably short time. We are concerned specifically with the code that runs in a voting session — that is, just the main loop, not including the initialization steps that happen before it. Since Pvote's main loop alternates between waiting for events and processing events, responsiveness depends on the time required to process each event.

We assume that it takes a negligible time to evaluate individual expressions in Pthin. Of all the Pygame functions used during the main loop, only the **Surface** method `blit()` does a variable amount of work; its work is proportional to the area of the pasted sprite. The verifier ensures that sprites fit into their slots (verifier:{13, 22, 26, 30, 33, 36, 39–40, 48, 49–52}) and that slots can be no larger than the screen resolution (verifier:61–62), so the area of pasted sprites is bounded by the area of the screen. Thus, we also assume that it takes a negligible time to invoke individual Pygame functions.

There are less than 500 lines of code in Pvote, which isn't enough for straight-line execution to cause an appreciable delay; only loops could result in enough latency to make Pvote unresponsive. Loops can arise from Pthin's two looping statements (`while` and `for`) and from recursive function calls. Section 7.6 showed that function call depth is bounded by a constant, and Section 7.7 showed that iteration counts are bounded by parameters in the ballot definition file.

Therefore, there is an upper bound on the time it takes to process each event that depends on the length of lists in the ballot definition. Keeping the sizes of these lists small will ensure that Pvote always stays responsive.

R3. Become inert after a ballot is committed.

As established in Section 7.3, the ballot is only committed upon arrival at the last page, where `self.page_i` becomes `len(self.model.pages) - 1` (Navigator:13–15). Thereafter, the page and state can never change again, since they can only change if `self.page_i != len(self.model.pages) - 1` (Navigator:12, Navigator:15–16). Thus, the ballot can never be committed more than once.

To ensure that Pvote becomes totally inert, one could examine the ballot definition to see that there are no bindings defined for the last page. As the only incoming messages to the navigator are `press` (main:16), `touch` (main:21), and `timeout` (main:25), eliminating bindings would guarantee that only `timeout` would ever get called after that point. The `timeout` method can only play audio and call `goto`, which would not cause a page or state transition because `self.page_i == len(self.model.pages) - 1`.

R4. Display a completion screen when and only when a ballot is committed, and continue to display this screen until the next session begins.

As established in Section 7.3, the ballot is committed upon and only upon arrival at the last page. The last page's screen is the completion screen. Since no more transitions can happen after the last page is reached, this screen remains on the display until Pvote is restarted.

It is up to the author of the ballot definition to ensure that the completion screen has a distinct appearance.

R5. Exhibit behaviour in each session independent of any previous sessions.

By design, Pvote is restarted for each voting session and does not read from any external storage except for the ballot definition, which it never rewrites, so it cannot carry any state from previous voting sessions.

R6. Exhibit behaviour independent of which parts of buttons are touched.

Incoming touch events are processed only by a single clause in the main event loop (main:17–21). This clause translates the touch coordinates into a target index by calling `locate` on the **Video** object, which has no side effects (Video:19–22). Only this target index is then passed on to the **Navigator**. Therefore, within a given target, all touch coordinates have the same effect. It is up to the author of the ballot definition to ensure that the targets have reasonable sizes and locations.

R7. Exhibit behaviour that is determined entirely by the ballot definition and the stream of user input events and their timing.

Pvote is single-threaded, uses no shared memory, and does not access the clock or any sources of randomness, so its behaviour is deterministic except for information introduced by the incoming event stream. The incoming event stream contains user-generated events, `TIMER_DONE` events, and `AUDIO_DONE` events. `TIMER_DONE` events are determined entirely by the timing of user-generated events and the `timeout_ms` parameter. `AUDIO_DONE` events are determined entirely by the processing of other events and the length of audio clips in the ballot definition. Therefore, given the same sequence and timing of user input events and the same ballot definition, Pvote will always exhibit the same behaviour.

R8. Commit valid selections.

Invariant **INV8** in the **Navigator** establishes that `self.selections` is always a list of lists, with one list per group. This format ensures that the selection data passed to the **Printer** cannot express any groups (contests or write-ins) other than those specified in the ballot definition. Any option appended to a selection list is an option referenced in a **Step** (Navigator:76, Navigator:82). The verifier ensures that this option reference is valid, whether it is a direct reference (verifier:72) or an indirect reference through an option area (verifier:91, verifier:21). Section 7.4 establishes that overvotes cannot occur. It is up to the author of the ballot definition to ensure that the **Text** data accurately represents the contests and options.

R9. Commit the ballot when and only when so requested by the voter.

Section 7.3 established that the ballot is committed when and only when there is a transition to the last page. That is as much as can be upheld by technical means; only a human can verify that the voter's expectations about committing are met.

To ensure this, one must examine the ballot definition to see that all keys and targets that cause transitions to the last page are clearly identified to the voter (visually and aurally) that they will commit the ballot. Also, no other keys or targets should be presented in a way that implies they will commit the ballot, and no visual display or audio feedback should falsely indicate that the ballot has been committed when it has not.

To minimize the possibility of voter error, one can examine the ballot definition to see that there is adequate confirmation before entering any page or state with a binding that causes a transition to the last page.

R10. Correctly and unambiguously commit the selections the voter made.
This requires establishing four things:

1. Selection and deselection of options indeed occurs correctly according to user actions. This is argued below for R13.
2. The ballot is committed when and only when the voter so requests. This is argued above for R9.
3. The printed selections are accurate. Printing occurs in the `write` method (Printer:4–25).
   - Every write-in group with a nonzero number of selected options causes the first loop to be executed (Printer:11–15). This loop proceeds through the character options in the order they were selected, and adds each option in the write-in to `line` exactly once (Printer:15). Anything that is added to `line` is printed exactly once (Printer:13–14, Printer:16).
   - Every contest group with a nonzero number of selected options causes the second loop to be executed (Printer:20–22). This loop proceeds through options in their order in the **Group** (Printer:20). Since all the option indices in the selection list must be valid (R8), every option that is present in the selection list will be printed exactly once (Printer:21).
4. The printed selections are unambiguous. Since the `print` statement always finishes its output with a newline, everything printed by `print` starts on a new line. Because group and options names are at most 50 bytes long (verifier:54, verifier:57), no string constructed in **Printer** ever exceeds 70 bytes in length (Printer:9, Printer:12–15, Printer:16, Printer:19, Printer:22, Printer:24). All the strings sent to `print` contain only printable ASCII characters (Ballot:134). Therefore, as long as the printing hardware can fit 70 characters across the page, no group or option names will wrap. Thus, every printed line can be identified by its first character:
   - `+` introduces the name of a write-in group (Printer:9).
   - `=` introduces the content of a write-in (Printer:13, Printer:16).
   - `*` introduces the name of a contest group (Printer:19, Printer:24).
   - `-` introduces the name of an option (Printer:22).
   - ~ marks the end of the ballot.

   Therefore, if all the **TextGroup**s in the ballot definition have unique names, the groups can be uniquely identified on the printout. Also, options in a contest group are printed separated by newlines, and options in a write-in group are printed separated by tildes (ASCII 126), which are not allowed in option names (Ballot:134). Therefore, if all the options in each **TextGroup** have unique strings, the options can be uniquely identified on the printout.

R11. Present instructions, contests, and options as specified.
The instructions, contests, and options are prerendered images embedded in the ballot definition. Thus, as long as the text and other information in the images is correct, it will be displayed correctly.

R12. Navigate among instructions, contests, and options as specified.

Navigation occurs only by the `goto` method, which is called whenever a binding is invoked (Navigator:74) and whenever a timeout occurs (Navigator:91). As long as the destination page and state are specified correctly in the ballot definition, the transition will occur to the correct page and state (Navigator:12–16).

R13. Select and deselect options according to user actions as specified.

Selection and deselection occurs entirely within the `execute` method, which can only be called in response to the invocation of a binding (Navigator:71), and a binding can only be invoked in response to a user action (Navigator:51, Navigator:55). If the selection steps in bindings are specified correctly in the ballot definition, then the correct selection or deselection operations will take place (Navigator:76–88).

R14. Correctly indicate whether options are selected when directed to do so.
R15. Correctly indicate how many options are selected when directed to do so.
R16. Correctly indicate which options are selected when directed to do so.

The `update` method in **Navigator** is called whenever `goto` is called (Navigator:18), and `goto` is always called each time a binding is invoked (Navigator:74) or a timeout is received (Navigator:91). The `update` method always redraws everything on the screen. It first pastes the current layout's full-screen image (Navigator:20, Video:14). Then it pastes the state's sprite (Navigator:21).

The indication of whether options are selected is determined by the flag `unselected` (Navigator:24), which chooses between the selected and unselected sprites for each option area. As long as the option area points to the correct option and the option points to the correct `sprite_i`, this will be displayed correctly.

The indication of how many options are selected is determined by the `count` variable (Navigator:30), which is added to a counter area's `sprite_i` to select the sprite to display. As long as the counter area points to the correct group and sprite index, this will be displayed correctly.

The indication of which options are selected is done by the `review` method. Calls to `paste` appear exactly twice in this method: once for option sprites (Navigator:41) and once for the cursor sprite (Navigator:45). The option sprite is `option.sprite_i`, the selected sprite for an option, and the option is taken directly from the selection list (Navigator:40). So it cannot display any unselected options. On the other hand, the paste operation is executed once for every option in the selection list, since the number of selections cannot exceed `max_sels` and `i` takes on every value from `0` to `max_sels` − `1`.

Appendix A

Glossary

ballot style: A combination of contests and options (for a particular set of voters).

binding: A triple of stimulus, condition, and response.

committed: A ballot is *committed* when the selection of votes is finalized. For a DRE, a ballot is committed when it is recorded. For a ballot printing or marking device, a ballot is committed when it is printed.

condition: A logical predicate concerning the current selection state.

contest: A race or a proposition.

contest group: A group representing a contest on the ballot, where the options are candidates or referendum choices.

empty: A group, contest, or write-in is *empty* when it has no options selected.

full: A group, contest, or write-in is *full* when the maximum options are selected.

group: A set of options that can be selected (see *contest group* and *write-in group*).

invoke: To *invoke* a binding is to carry out the response it specifies.

match: A binding *matches* when its specified stimulus matches the input received.

operative: A binding is *operative* when all its conditions are satisfied.

option: A choice in a group (a candidate in a race for office, one of the choices for a proposition, or a character that can be entered for a write-in).

overvote: Selecting more than the maximum allowed number of selections in a particular contest.

response: A system behaviour in response to user input (e. g. changing a selection, navigating to another page, or playing audio).

selection: An option that is currently selected.

selection state: The list of options that are selected in each group.

stimulus: An instance of user input (e. g. a keypress or a screen touch).

undervote: Selecting fewer than the maximum allowed number of selections in a particular contest.

write-in group: A group representing the text written into a single write-in option, where the options are characters.

write-in option: An option that allows a candidate's name to be written in.

voting session: The period from when a voter starts interacting with a voting machine until a ballot is committed or the voter abandons the machine.

Appendix B

Deployment example

To evaluate Pvote, it may help to have in mind some context in which it will be used. Here is just one example of a possible deployment scenario for an electronic ballot printer based on Pvote.

B.1 Before election day

The ballot definition files are prepared and widely published, along with their hashes, before election day.

B.2 Election day before polls open

The polling place is divided into three areas: the *public area*, where anyone can stand, the *voting area*, which voters are permitted to enter after they have been authorized to vote by pollworkers, and the *private area*, which is accessible to pollworkers only.

The voting area contains any number of *voting stations*. Each voting station has a touchscreen, a pair of headphones, a keypad, and a printer. There is a shield or curtain around the station to protect the voter's privacy. The voting stations are stateless.

The private area contains a ballot scanner and a number of bins for flash cards (one bin for each ballot style to be used at that polling place). Before opening the polls, the pollworkers use a *flash station* to prepare some flash cards for each ballot style. The flash station can be an ordinary PC. For each ballot style, a pollworker carries out the following steps:

1. Load the ballot definition file onto the flash station. The flash station displays the hash of the file.
2. Verify the computed hash against the published hash.
3. Insert flash cards one by one. The flash station erases each card and copies the file onto the card.
4. Label each flash card according to its ballot style.
5. Deposit each flash card in the bin for its ballot style.

The pollworkers can then shut down the flash station, or leave it set up in case they want to be able to prepare flash cards on the fly with other ballot styles throughout the day (e. g. for the occasional voter at the wrong polling place). After the flash cards are prepared, the polling place is opened.

B.3 Election day with polls open

The voting procedure for each voter is as follows:

1. The voter lines up to be authorized to vote.
2. After checking that the voter is authorized and determining which ballot style the voter should get (which might depend on the voter's party affiliation or address), the pollworker takes a flash card from the appropriate bin.
3. The pollworker proceeds with the voter to any available voting station and inserts the card. The pollworker inserts a key into the station and turns it, which aborts and restarts Pvote. Pvote loads the ballot definition from the card on startup. Once the initial screen appears, the pollworker removes the card, walks away, and returns the card to its bin.
4. The voter privately interacts with Pvote to make selections on the ballot. When the final screen is reached, the voter's selections are printed out on a paper ballot.
5. The voter verifies the paper ballot.
6. The voter carries the paper ballot (covered in a privacy folder) to the ballot scanner and places it in the scanner. The scanner records the actual scanned image of the paper ballot.

B.4 Election day after polls close

The counts reported by the ballot scanner are posted locally at each polling place. Each polling place posts its counts on a public website.

Each polling place also posts encrypted files containing the scanned images of its paper ballots on the public website. An openly chosen random sample of the polling places, as well as any polling places with a sufficiently narrow margin of victory, post their scanned images of paper ballots without encryption. Members of the public can run their own OCR software to verify the counts.

After 3 years, the encryption keys are published so the entire election can be verified by the public.

Appendix C

WAV audio file format

The essential elements of the Microsoft WAV file format are as follows:

- All integers are represented in little-endian order.
- A *chunk* is a block of data preceded by an 8-byte header. The first 4 bytes of the header are a chunk type identifier, and the next 4 bytes give the length of the data block, not including the header.
- A WAV file consists of a chunk of type `"RIFF"` that contains the 4-byte string `"WAVE"` followed by other chunks.
- The minimal two required chunks are a `"fmt "` chunk and a `"data"` chunk.
- The `"fmt "` chunk contains this 16-byte structure:

Size	Contents
2 bytes	compression type (1 for none)
2 bytes	number of channels (1 for mono, 2 for stereo)
4 bytes	number of samples per second
4 bytes	number of bytes per second
2 bytes	number of bytes per sample × number of channels
2 bytes	number of bits per sample

- The `"data"` chunk contains the audio sample data. For 16-bit samples, each sample is a signed little-endian 16-bit value.