Pvote Software Review Assurance Document

Ka-Ping Yee
ping@zesty.ca

March 22, 2007

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, the subject of this review, 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

In the following, **committed** means voter selections are finalized, either by printing them onto a paper ballot or recording them in a DRE. A **voting session** consists of the time from when a particular voter begins using a voting machine until the ballot is committed or the voter abandons the machine. The **ballot definition file** is a file that describes the ballot parameters and the voting user interface.

We intend to establish that Pvote can be relied upon to:

R1. Not abort during a voting session.
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 the same deterministic behaviour in all voting sessions that use the same ballot definition.
R6. Present instructions, contests, and options as specified in the ballot definition.
R7. Navigate among instructions, contests, and options as specified in the ballot definition.
R8. Select and deselect options according to user actions as specified in the ballot definition.
R9. Prevent overvotes.
R10. Correctly indicate whether options are selected when the ballot definition calls for such indication.
R11. Correctly indicate how many options are selected when the ballot definition calls for such indication.
R12. Correctly indicate which options are selected when the ballot definition calls for such indication.
R13. Commit the selections the voter made.

1.3 Assumptions

A1. The voting machine software (ostensibly Pvote) is handed over for review before the election.
A2. The software runs intact on the voting machines, unchanged from what was reviewed.
A3. Pvote is started once per voting session.
A4. Only authorized voters can begin voting sessions.
A5. Ballot definition files are published for review and testing before the election.
A6. For each voting session, the correct ballot definition is selected and provided to Pvote.
A7. Ballot definitions are provided to Pvote intact, unchanged from what was reviewed.
A8. In the ballot definition, there is at least one page, and each page has at least one state.
A9. No ballot definition contains more than two billion elements in any list (a specification of the ballot definition format is given below).
A10. The software platform functions correctly (specifications of the expected behaviour of the language and supporting software libraries are given below).
A11. 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 casting of multiple ballots (R3), allow casting of an invalid ballot (R9), mislead pollworkers about the casting of a ballot (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, R10, R11, R12, R13)?
- **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.
- **Faulty or subverted hardware.** This is a software review (A10).
- **Tampering with the software distribution.** We assume the software is not altered between review and use (A2).
- **Tampering with the ballot definition.** We assume the ballot definition is not altered between review and use (A7).
- **Tampering with cast vote records.** Protecting the integrity of vote records after ballots are committed is beyond Pvote's purview.
- **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, such as:
  - 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 implementors, 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 ballot definition 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 edge is triggered by a user action under specified conditions or by an idle timeout. Traversing an edge can cause options to be selected or deselected. Audio feedback sequences are associated with states and with edges 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 Serialization

This section describes how data types are recorded in the ballot definition file.

2.2.1 Primitive Types

The data structures are built up from the following types:

- **int**: An integer in the range from 0 to 4294967295 (`0xffffffff`) inclusive. Serialized as four bytes, most significant first.
- **intn**: An integer in the range from 0 to 4294967294 (`0xfffffffe`) 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 8-bit bytes with maximum length 4294967295. 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 4294967295 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**.

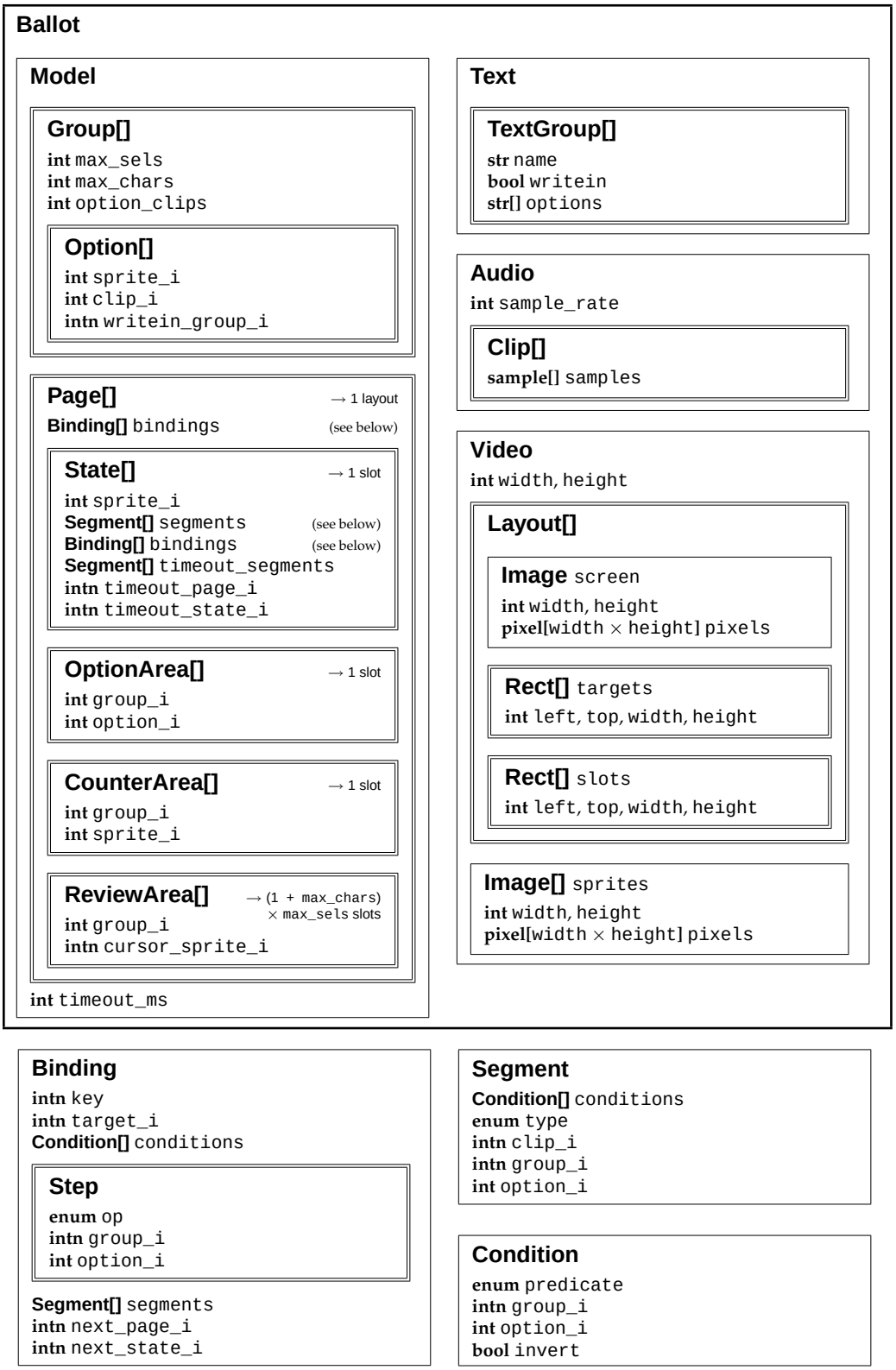

Figure 2.1: Ballot definition data structures. A double border around a subelement signifies a list of subelements of that type.

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. The ballot definition can specify an automatic transition or audio message to occur when there has been no user activity and no audio playing for `timeout_ms` milliseconds.

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 characters of the write-in. In all cases, the current selection for a group is a list of options (even though a contest selection has set 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`: The maximum allowed number of characters that can be entered for write-in options in the group. If this is zero, the group has no write-in options. This must be zero if the group is a write-in group.
- `option_clips`: The number of audio clips associated with each option.
- `options`: The list of options in the group.

Each option 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 the option is a write-in option, this field specifies the write-in group that will hold the text entered for this write-in. If the option is a regular option, this field is `None`.

Note the logical relationships among these fields. If `max_chars` is zero, then all the options should 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`.

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 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`: The index of a group in the **Model**'s `groups`.
- `option_i`: The index of an option in that **Group**'s `options`, which will appear in the option area's slot.

A **CounterArea** has two fields:

- `group_i`: The index of a group in the **Model**'s `groups`.
- `sprite_i`: The starting index of a set of sprites occupying indices from `sprite_i` to `sprite_i` + `max_sels` in `video.sprites`. 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`: The index of a group in the **Model**'s `groups`. For each of the selected options in group `group_i`, there is one slot for the option and `max_chars` slots for the characters of text for a write-in option, hence a total of (1 + `max_chars`) × `max_sels` slots.
- `cursor_sprite_i`: This field can be `None`; otherwise it specifies a sprite to be placed 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 pasted into the state's slot.
- `segments`: A list of audio segments played upon entry into this state.
- `bindings`: Bindings that apply to just this state. These can override page-level bindings; when the user presses a key or touches a target, an operative binding is sought first in the state's bindings and then in the page's bindings.
- `timeout_segments`: A list of audio segments to be played upon timeout.
- `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 array 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 it matches the stimulus and its condition is satisfied.

The response consists of three parts: selection operations (given as **Step**s), audio feedback, 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 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 an 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. Invoking a binding always interrupts any currently playing feedback and starts afresh. The segments for the binding, if any, are queued to be played first. The segments for the newly entered state, if a state transition takes place, are queued to be played 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 played; otherwise the segment is skipped and playback continues with the next segment in the list.
- `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 (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 group (for `SG_COUNT_SELS` or `SG_MAX_SELS`) or option (for `SG_OPTION` or `SG_LIST_SELS`) 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

The following specification of the data structures is annotated on the right with the constraints that have to be met in order for the ballot definition file to be valid. For brevity, the constraints are written with some unqualified names:

- `groups` and `pages` refer to fields of the **Model** object
- `clips` refers to the field of the **Audio** object
- `sprites` refers to the field of the **Video** object

Also, *length*(x) refers to the length of a list x, and the symbol $\hat{=}$ means "sizes match" (that is, $\texttt{a} \hat{=} \texttt{b} \Leftrightarrow \texttt{a.width} = \texttt{b.width}$ *and* $\texttt{a.height} = \texttt{b.height}$).

```
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[] 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              ∀ segment ∈ segments:
    Binding[] bindings                  segment.type = 0 or segment.use_step = false
    Segment[] timeout_segments      ∀ segment ∈ timeout_segments:
                                        segment.type = 0 or segment.use_step = false
    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
                                ∀ slot ∈ review area's character slots:
Binding:                            slot ≙ groups[group_i]'s character size
    intn key
    intn target_i
    Condition[] conditions
    Step[] steps                length(steps) = 0 ⇒ ∀ segment ∈ segments:
    Segment[] segments              segment.type = 0 or segment.use_step = false
    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
                                      g = 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                            name contains only printable characters
    bool writein
    str[] options                       ∀ option ∈ options:
                                            option contains only printable characters
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` array 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 not strictly necessary for Pvote to function. They are present due to a Pygame limitation: Pygame refuses to create zero-length sounds or zero-sized images.

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 have unlimited size. Integer literals are written in decimal.

Strings are variable-length arrays of 8-bit bytes. String literals are written exactly as in C.

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 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`
`o.`*field*	*field* is bound in `o`'s namespace	access a field in the object `o`'s namespace
`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`
`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` ≥ 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). For the expressions listed in this table, `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 any operand has an unacceptable type or the precondition is violated, a runtime 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, which are 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 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 runtime error if `i` is zero
`print s`		send `s` and a newline to the printer
`if i:` *block*		if `i` is nonzero, execute the first *block*
`else:` *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. In these descriptions, `i` is an integer, `s` is a string, `l` is a list, `o` is an object, and `x` is a value of any type.

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 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	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 that 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.

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. The event-related functions and event types used in Pvote are shown in Table 4.1.

Function	**Preconditions**	**Definition**
`pygame.event.wait()`		Wait for the next event and return an event object describing it.
`pygame.time.set_timer(`*event*`, `*delay*`)`	*event* is an integer event type code greater than or equal to `USEREVENT`. *delay* is an integer number of milliseconds.	Schedule an event of type *event* to occur after *delay* milliseconds (replacing any currently scheduled timer for an event of that type).

Event `type` value	**Definition**
`pygame.KEYDOWN` (2)	A keyboard 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.
`pygame.USEREVENT` (24)	Events with `type` equal to `USEREVENT` and above have user-defined meaning.

Table 4.1: Pygame event operations 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 not designed to use 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.
`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 for the given sound clip.

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 type code greater than or equal to `USEREVENT`.	Schedule an event of type *event* to occur when the currently playing sound clip stops playing, either because the end of the clip has been reached or playing has been stopped.

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`]`. $0 \leq x < x + width <$ width of surface. $0 \leq y < y + height <$ 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** (`pvote`): 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 that the ballot is well-formed. The purpose of the verifier is to ensure that a badly formed ballot will cause an immediate failure, so that a runtime error cannot occur after the user interface has started operating.

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`**`()`: Start playing the `timeout_segments` for the current state. Go to the page and state given by `timeout_page_i` and `timeout_state_i` if `timeout_page_i` is not `None`.

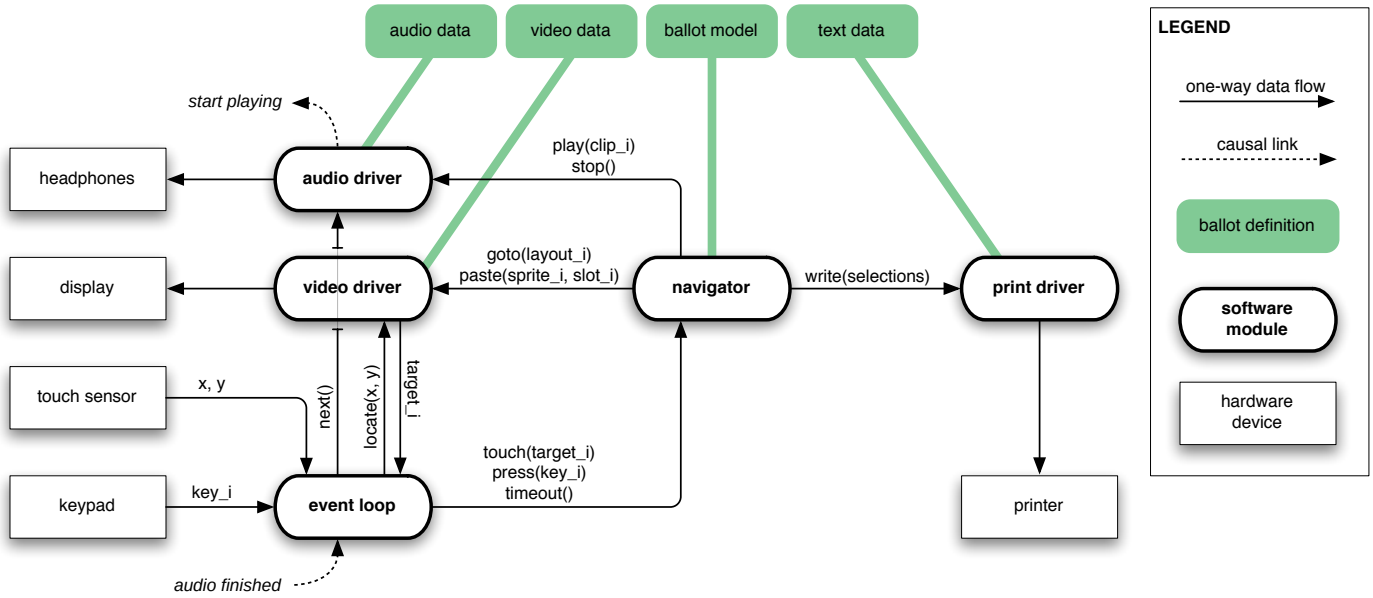

Figure 6.1: Block diagram of the virtual machine, which consists of the five software modules in bold. The arguments clip_i, layout_i, sprite_i, target_i, key_i, x, and y are integers; selections is an array of lists of integers.

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 array 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 schedules a notification event with the type constant `AUDIO_DONE` to 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 array.
- **`locate`**`(x, y)`: Find and return the index of the first target that contains the given point in the current layout's array of targets, or a failure code 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 future every time it receives any event.

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

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 **ASSUMPTIONS** column contains assumptions and preconditions for each line, function, or method. The **REASONS FOR VALIDITY** column gives an explanation of why each line will not cause a runtime error, or highlights potential causes of runtime error with the symbol ⚠ . The **POSTCONDITIONS** column outlines what is expected to be true after the line has been executed, or what is returned from a function or method.

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

	ASSUMPTIONS	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			`event` is a **pygame.Event**.
13	`pygame.KEYDOWN` is an int.	`event` is an **Event** (12) ⇒ `event.type` exists and is an int.	
14		`navigator` is a **Navigator** (9). `event` is a keypress (13) ⇒ `event.key` exists and is an int.	
15	`pygame.MOUSEBUTTONDOWN` is an int.	`event` is an **Event** (12) ⇒ `event.type` exists and is an int.	
16		`event` is a mouse click (15) ⇒ `event.pos` exists and is a list of two ints.	`x` and `y` are ints.
17		`video` is a **Video.Video** (7). `x` and `y` are ints (16).	`target_i` is an int or `None` (Video:18).
18			
19		`navigator` is a **Navigator** (9). `target_i` is an int (17, 18).	
20		`AUDIO_DONE` is an int (2). `event` is an **Event** (12) ⇒ `event.type` is an int.	
21		`audio` is an **Audio.Audio** (6).	
22		`TIMER_DONE` is an int (3). `event` is an **Event** (12) ⇒ `event.type` is an int.	
23		`navigator` is a **Navigator** (9).	
24		`TIMER_DONE` is an int (3). `ballot` is a **Ballot** (4) ⇒ `ballot.model.timeout_ms` is an int (Ballot:6, Ballot:19).	

6.2.1 pvote

```
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()
    event = pygame.event.wait()
    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()
    pygame.time.set_timer(TIMER_DONE, ballot.model.timeout_ms)
```

	ASSUMPTIONS	REASONS FOR VALIDITY	POSTCONDITIONS
1			`sha` is bound to the SHA module.
2			
3	`stream` is a readable stream.		
4		`stream` is a stream (3). ⚠ if file header not present.	
5		`sha` is the SHA module (1) ⇒ `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 loaded ballot definition data matches its concluding hash.
11	`length` is an int.		Returns a string (12, 14).
12		`self.stream` is a stream (5). `length` is an int (11).	`self.stream` is a stream (5) ⇒ `data` is a str.
13		`self.sha` is a **sha** (5). `data` is a str (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** (123).
18		`stream` is a stream (16). `Page` is a class (31).	`self.pages` is a list of **Page** (123).
19		`stream` is a stream (16).	`allow_none` = 0 ⇒ `self.timeout_ms` is an int (125).
20			
21	`stream` is a readable stream.		
22		`stream` is a stream (21).	`allow_none` = 0 ⇒ `self.max_sels` is an int (125).
23		`stream` is a stream (21).	`allow_none` = 0 ⇒ `self.max_chars` is an int (125).
24		`stream` is a stream (21).	`allow_none` = 0 ⇒ `self.option_clips` is an int (125).
25		`stream` is a stream (21). `Option` is a class (31).	`self.options` is a list of **Option** (123).
26			
27	`stream` is a readable stream.		
28		`stream` is a stream (27).	`self.sprite_i` is an int (125).
29		`stream` is a stream (27).	`self.clip_i` is an int (125).
30		`stream` is a stream (27).	`allow_none` = 1 ⇒ `self.writein_group_i` is int or `None` (125).
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** (123).
34		`stream` is a stream (32). `State` is a class (38).	`self.states` is a list of **State** (123).
35		`stream` is a stream (32). `OptionArea` is a class (46).	`self.option_areas` is a list of **OptionArea** (123).
36		`stream` is a stream (32). `CounterArea` is a class (50).	`self.counter_areas` is a list of **CounterArea** (123).
37		`stream` is a stream (32). `ReviewArea` is a class (54).	`self.review_areas` is a list of **ReviewArea** (123).
38			
39	`stream` is a readable stream.		
40		`stream` is a stream (39).	`allow_none` = 0 ⇒ `self.sprite_i` is an int (125).
41		`stream` is a stream (39). `Segment` is a class (78).	`self.segments` is a list of **Segment** (123).
42		`stream` is a stream (39). `Binding` is a class (58).	`self.bindings` is a list of **Binding** (123).
43		`stream` is a stream (39). `Segment` is a class (78).	`self.timeout_segments` is a list of **Segment** (123).
44		`stream` is a stream (39).	`allow_none` = 1 ⇒ `self.timeout_page_i` is int or `None` (125).
45		`stream` is a stream (39).	`allow_none` = 1 ⇒ `self.timeout_state_i` is int or `None` (125).

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, 1)
```

	ASSUMPTIONS	REASONS FOR VALIDITY	POSTCONDITIONS
46			
47	`stream` is a readable stream.		
48		`stream` is a stream (47).	`allow_none` = 0 ⇒ `self.group_i` is an int (125).
49		`stream` is a stream (47).	`allow_none` = 0 ⇒ `self.option_i` is an int (125).
50			
51	`stream` is a readable stream.		
52		`stream` is a stream (51).	`allow_none` = 0 ⇒ `self.group_i` is an int (125).
53		`stream` is a stream (51).	`allow_none` = 0 ⇒ `self.sprite_i` is an int (125).
54			
55	`stream` is a readable stream.		
56		`stream` is a stream (55).	`allow_none` = 0 ⇒ `self.group_i` is an int (125).
57		`stream` is a stream (55).	`allow_none` = 1 ⇒ `self.cursor_sprite_i` is int or `None` (125).
58			
59	`stream` is a readable stream.		
60		`stream` is a stream (59).	`allow_none` = 1 ⇒ `self.key` is int or `None` (125).
61		`stream` is a stream (59).	`allow_none` = 1 ⇒ `self.target_i` is int or `None` (125).
62		`stream` is a stream (59). `Condition` is a class (67).	`self.conditions` is a list of **Condition** (123).
63		`stream` is a stream (59). `Step` is a class (74).	`self.steps` is a list of **Step** (123).
64		`stream` is a stream (59). `Segment` is a class (78).	`self.segments` is a list of **Segment** (123).
65		`stream` is a stream (59).	`allow_none` = 1 ⇒ `self.next_page_i` is int or `None` (125).
66		`stream` is a stream (59).	`allow_none` = 1 ⇒ `self.next_state_i` is int or `None` (125).
67			
68	`stream` is a readable stream.		
69		`stream` is a stream (68).	`allow_none` = 0 ⇒ `self.predicate` is an int (125).
70		`stream` is a stream (68).	`allow_none` = 1 ⇒ `self.group_i` is int or `None` (125).
71		`stream` is a stream (68).	`allow_none` = 0 ⇒ `self.option_i` is an int (125).
72		`stream` is a stream (68).	`allow_none` = 0 ⇒ `self.invert` is an int (125).
73			
74	`stream` is a readable stream.		
75		`stream` is a stream (74).	`allow_none` = 0 ⇒ `self.op` is an int (125).
76		`stream` is a stream (74).	`allow_none` = 1 ⇒ `self.group_i` is int or `None` (125).
77		`stream` is a stream (74).	`allow_none` = 0 ⇒ `self.option_i` is an int (125).
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** (123).
81		`stream` is a stream (79).	`allow_none` = 0 ⇒ `self.type` is an int (125).
82		`stream` is a stream (79).	`allow_none` = 0 ⇒ `self.clip_i` is an int (125).
83		`stream` is a stream (79).	`allow_none` = 1 ⇒ `self.group_i` is int or `None` (125).
84		`stream` is a stream (79).	`allow_none` = 0 ⇒ `self.option_i` is an int (125).
85			
86	`stream` is a readable stream.		
87		`stream` is a stream (87). `TextGroup` is a class (89).	`self.groups` is a list of **TextGroup** (123).

Ballot.py (page 2 of 3)

```
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, 1)

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

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

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

	ASSUMPTIONS	REASONS FOR VALIDITY	POSTCONDITIONS
88			
89	`stream` is a readable stream.		
90		`stream` is a stream (90).	`self.name` is a str (129).
91		`stream` is a stream (90).	`allow_none` = 0 ⇒ `self.writein` is an int (125).
92		`stream` is a stream (90). `get_str` is a function (129).	`self.options` is a list of strings (123, 129).
93			
94	`stream` is a readable stream.		
95		`stream` is a stream (95).	`allow_none` = 0 ⇒ `self.sample_rate` is an int (125).
96		`stream` is a stream (85). `Clip` is a class (89).	`self.clips` is a list of **Clip** (123).
97			
98	`stream` is a readable stream.		
99		`stream` is a stream (99). `allow_none` = 0 ⇒ `get_int` returns an int (125).	`stream` is a stream (99) ⇒ `self.samples` is a str.
100			
101	`stream` is a readable stream.		
102		`stream` is a stream (102).	`allow_none` = 0 ⇒ `self.width` is an int (125).
103		`stream` is a stream (102).	`allow_none` = 0 ⇒ `self.height` is an int (125).
104		`stream` is a stream (102). `Layout` is a class (107).	`self.layouts` is a list of **Layout** (123).
105		`stream` is a stream (102). `Image` is a class (112).	`self.sprites` is a list of **Image** (123).
106			
107	`stream` is a readable stream.		
108		`Image` is a class (112). `stream` is a stream (102).	`self.screen` is a **Image**.
109		`stream` is a stream (102). `Rect` is a class (117).	`self.targets` is a list of **Rect** (123).
110		`stream` is a stream (102). `Rect` is a class (117).	`self.slots` is a list of **Rect** (123).
111			
112	`stream` is a readable stream.		
113		`stream` is a stream (113).	`allow_none` = 0 ⇒ `self.width` is an int (125).
114		`stream` is a stream (113).	`allow_none` = 0 ⇒ `self.height` is an int (125).
115		`stream` is a stream (113). `self.width` is an int (114). `self.height` is an int (115).	`stream` is a stream (113) ⇒ `self.pixels` is a str.
116			
117	`stream` is a readable stream.		
118		`stream` is a stream (118).	`allow_none` = 0 ⇒ `self.left` is an int (125).
119		`stream` is a stream (118).	`allow_none` = 0 ⇒ `self.top` is an int (125).
120		`stream` is a stream (118).	`allow_none` = 0 ⇒ `self.width` is an int (125).
121		`stream` is a stream (118).	`allow_none` = 0 ⇒ `self.height` is an int (125).
122	`stream` is a stream.		Returns a list of instances of `Class` (124).
123		`stream` is a stream (123). `allow_none` = 0 ⇒ `get_int` returns an int (125).	
124	`stream` is a stream. `allow_none` is 0 or 1.		Returns an int if `allow_none` is 0 (127, 128); else returns int or `None`.
125		`stream` is a stream (125). ⚠ if `read` returns less than 4 bytes. `list` returns a list of 4 1-byte strings.	`a`, `b`, `c`, `d` are 1-byte strings.
126		`allow_none` is an int (125). `a`, `b`, `c`, `d` are strings (126).	
127		`a`, `b`, `c`, `d` are 1-byte strings (126).	An int is returned.
128	`stream` is a stream.		Returns a string (130).
129		`stream` is a stream (129). `allow_none` = 0 ⇒ `get_int` returns an int (125).	A string is returned.

Ballot.py (page 3 of 3)

```
class TextGroup:
    def __init__(self, stream):
        self.name = get_str(stream)
        self.writein = get_int(stream, 0)
        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)

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

def get_int(stream, allow_none):
    [a, b, c, d] = list(stream.read(4))

    if not allow_none or a + b + c + d != "\xff\xff\xff\xff":
        return ord(a)*16777216 + ord(b)*65536 + ord(c)*256 + ord(d)

def get_str(stream):
    return stream.read(get_int(stream, 0))
```

	ASSUMPTIONS	REASONS FOR VALIDITY	POSTCONDITIONS
1	`ballot` is a **Ballot**.		`ballot` satisfies the validity constraints in Section 2.7.
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		`ballot.model.pages` is a list (1, Ballot:6, Ballot:18).	`page_i` is an int. `pages` is a list of **Page** (Ballot:18) ⇒ `page` is a **Page**.
8		`page_i` is a valid index in `model.pages` (7) ⇒ `page_i` is a valid index in `video.layouts` (6).	`layouts` is a list of **Layout** (Ballot:104) ⇒ `layout` is a **Layout**.
9		`page.bindings` is a list (7, Ballot:33).	`bindings` is a list of **Binding** (Ballot:33) ⇒ `binding` is a **Binding**.
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 an int. `states` is a list of **State** (Ballot:34) ⇒ `state` is a **State**.
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` fits the state's slot at `layout.slots[state_i]` (103).
14		`ballot` is a **Ballot** (2). `page` is a **Page** (7). `state` is a **State** (12) ⇒ `state.segments` is a list of **Segment** (Ballot:41).	Every element of `state.segments` is a valid **Segment** (85).
15		`state.bindings` is a list (Ballot:42).	`bindings` is a list of **Binding** (Ballot:42) ⇒ `binding` is a **Binding**.
16		`Ballot` is a **Ballot** (2). `page` is a **Page** (7). `binding` is a **Binding** (15).	`binding` is a valid **Binding** for this `page` (70).
17		`ballot` is a **Ballot** (2). `page` is a **Page** (7). `state` is a **State** (12) ⇒ `timeout_segments` is a list of **Segment** (Ballot:41).	Every element of `state.timeout_segments` is a valid **Segment** (85).
18		`ballot` is a **Ballot** (2). `timeout_page_i` is an int or `None` (Ballot:44). `timeout_state_i` is an int or `None` (Ballot:45).	Either `timeout_page_i` is `None`, or `timeout_page_i` and `timeout_state_i` are a valid page and state index (82).
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).	`option_areas` is a list of **OptionArea** (7, Ballot:35) ⇒ `area` is an **OptionArea**.
21		`ballot` is a **Ballot** (2). `page` is a **Page** (7). `area` is a **OptionArea** (20).	`area.group_i` is an int (Ballot:48) ⇒ `area.group_i` is a valid group index and `area.option_i` is a valid option index in that group (82).
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).	`counter_areas` is a list of **CounterArea** (7, Ballot:36) ⇒ `area` is a **CounterArea**.
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. The sprite at `area.sprite_i` + `i` fits the counter area's slot (103).
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
```

	ASSUMPTIONS	REASONS FOR VALIDITY	POSTCONDITIONS
28		`page.review_areas` is a list (7, Ballot:37).	`review_areas` is a list of **ReviewArea** (7, Ballot:37) ⇒ `area` is a **ReviewArea**.
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 enough option slots for this review area (29). `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 (33). `char_sizes` for this review area's group contains all of the review area's character slots.
34		`slot_i` is an int (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 an int. `groups` is a list of **Group** (2) ⇒ `group` is a **Group**.
38		`group.options` is a list (37, Ballot:25).	`group.options` is a list of **Option** (37, Ballot:25) ⇒ `option` is a **Option**.
39		`option_sizes` is a list of lists (3). `group_i` is a valid group index (36). ⚠ 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 (36). ⚠ 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		`ballot.audio.clips` is a list (1, Ballot:8, Ballot:96). ⚠ if `option.clip_i` + `group.option_clips` − 1 is out of bounds.	`ballot.audio.clips` contains at least `group.option_clips` elements starting at `option.clip_i`.
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 a **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.max_chars` > 0 ⇒ `group` cannot be the write-in group for any option (43–45).
47		`writein_group.options` is a list (44, Ballot:25).	`options` is a list of **Option** (44, Ballot:25) ⇒ `option` is an **Option**.
48		`char_sizes` is a list of lists (4). `group_i` is a valid group index (36). ⚠ if `option.sprite_i` is out of bounds.	All the 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).	`option_sizes[group_i]` is a list of **Slot** or **Sprite** objects (22, 30, 36, 39, 40) ⇒ `size` is a **Slot** or **Sprite**.
50		`group_i` is a valid group index (37). `size` is a **Slot** or **Sprite** (49). `option_sizes[group_i][0]` is a **Slot** or **Sprite** (49).	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).	`char_sizes[group_i]` is a list of **Slot** or **Sprite** objects (33, 48) ⇒ `size` is a **Slot** or **Sprite**.
52		`group_i` is a valid group index (37). `size` is a **Slot** or **Sprite** (51). `char_sizes[group_i][0]` is a **Slot** or **Sprite** (51).	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 size in option_sizes[group_i]:
            verify_size(size, option_sizes[group_i][0])
        for size in char_sizes[group_i]:
            verify_size(size, char_sizes[group_i][0])
```

	ASSUMPTIONS	REASONS FOR VALIDITY	POSTCONDITIONS
53		`ballot.text.groups` is a list (1, Ballot:7, Ballot:87).	`ballot.text.groups` and `groups` have the same length (5) ⇒ `group_i` is a valid group index. `ballot.text.groups` is a list of **TextGroup** (1, Ballot:7, Ballot:87) ⇒ `group` is a **TextGroup**.
54		`group.name` is a string (53, Ballot:90).	`char` is a 1-byte string.
55		`char` is a 1-byte string (54). ⚠ if assertion fails.	Every byte in `group.name` is printable.
56		`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).	This **TextGroup** `group` has the same number of options as its corresponding **Group** in `model.groups`.
57		`group.options` is a list (53, Ballot:92).	`group.options` is a list of strings (53, Ballot:92) ⇒ `option` is a string.
58		`option` is a string (57).	`char` is a 1-byte string.
59		`char` is a 1-byte string (58). ⚠ if assertion fails.	Every byte in `option` is printable.
60		`ballot.audio.clips` is a list (1, Ballot:8, Ballot:96).	`audio.clips` is a list of **Clip** (1, Ballot:8, Ballot:96) ⇒ `clip` is a **Clip**.
61		`clip.samples` is a string (60, Ballot:99). ⚠ if assertion fails.	Every **Clip** in `audio.clips` has a nonempty string of `samples`.
62		`ballot.video` is a **Video** (1, Ballot:9) ⇒ `width` and `height` are ints (Ballot:102, Ballot:103). ⚠ if assertion fails.	`ballot.video` has a nonzero `width` and nonzero `height`.
63		`ballot.video.layouts` is a list (1, Ballot:9, Ballot:104).	`layouts` is a list of **Layout** (1, Ballot:9, Ballot:104) ⇒ `layout` is a **Layout**.
64		`layout.screen` is an **Image** (63, 110). `ballot.video` is a **Video** (1, 9).	`layout.screen` has the same size as `ballot.video`.
65		`layout.targets` is a list of **Rect** (63, Ballot:109). `layout.slots` is a list of **Rect** (63, Ballot:110).	The sum of lists is a list of **Rect** ⇒ `rect` is a **Rect**.
66		`rect` is a **Rect** (65). `ballot.video` is a **Video** (1, Ballot:8).	`rect` does not extend beyond the right edge of the screen.
67		`rect` is a **Rect** (65). `ballot.video` is a **Video** (1, Ballot:8).	`rect` does not extend beyond the bottom edge of the screen.
68		`ballot.video.sprites` is a list (1, Ballot:8, Ballot:105).	`sprites` is a list of **Image** (1, Ballot:8, Ballot:105) ⇒ `sprite` is a **Image**.
69		`sprite` is an **Image**.	`sprite` has a nonzero width and height and the correct amount of pixel data for a `width` × `height` image.
70	`ballot` is a **Ballot**. `page` is a **Page**. `binding` is a **Binding**.		`binding` is a valid **Binding**.
71		`binding.conditions` is a list (70, Ballot:62).	`conditions` is a list of **Condition** (70, Ballot:62) ⇒ `condition` is a **Condition**.
72		`ballot` is a **Ballot** (70). `page` is a **Page** (70). `condition` is a **Condition** (71).	All the conditions in `binding.conditions` are valid.
73		`binding.steps` is a list (70, Ballot:63).	`binding.steps` is a list of **Step** (70, Ballot:63) ⇒ `steps` is a **Step**.
74		`step` is a **Step** (73). ⚠ if assertion fails.	`step.op` is 0, 1, 2, 3, or 4.
75		`ballot` is a **Ballot** (70). `page` is a **Page** (70). `step` is a **Step** (73).	`step.group_i` and `step.option_i` form a valid option reference.
76		`ballot` is a **Ballot** (70). `page` is a **Page** (70). `binding.segments` is a list of **Segment** (70, Ballot:64).	All the segments in `binding.segments` are valid for this page (85).
77		`ballot` is a **Ballot** (70). `binding.next_page_i` is an int (70, Ballot:65). `binding.next_state_i` is an int (70, 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 (82).
78	`ballot` is a **Ballot**. `page` is a **Page**. `condition` is a **Condition**.		`condition` is a valid **Condition**.
79		`condition.predicate` is int (78, Ballot:69). ⚠ if assertion fails.	`condition.predicate` is 0, 1, or 2.
80		`ballot` is a **Ballot** (78). `page` is a **Page** (78). `step` is a **Condition** (78).	`condition.group_i` and `condition.option_i` form a valid option reference.
81		`condition.invert` is an int (78, Ballot:72). ⚠ if assertion fails.	`condition.invert` is 0 or 1.
82	`ballot` is a **Ballot**. `page_i` and `state_i` are ints.		Either `page_i` is `None` (83), or `page_i` is a valid page index and `state_i` is a valid state index for that page (84).
83			
84		`ballot.model.pages` is a list of **Page** (83, 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.

verifier.py (page 3 of 4)

```
    for [group_i, group] in enumerate(ballot.text.groups):


        for char in list(group.name):
            assert 32 <= ord(char) < 127
        assert len(group.options) == len(groups[group_i].options)

        for option in group.options:
            for char in list(option):
                assert 32 <= ord(char) < 127

    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_condition(ballot, page, condition)

    for step in binding.steps:
        assert step.op in [0, 1, 2, 3, 4]
        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_condition(ballot, page, condition):
    assert condition.predicate in [0, 1, 2]
    verify_option_ref(ballot, page, condition)

    assert condition.invert in [0, 1]

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

	ASSUMPTIONS	REASONS FOR VALIDITY	POSTCONDITIONS
85	`ballot` is a **Ballot**. `page` is a **Page**. `segments` is a list of **Segment**.		Every segment in `segments` is a valid **Segment**.
86		`segments` is a list (85).	`segments` is a list of **Segment** (85) ⇒ `segment` is a **Segment**.
87		`segment.conditions` is a list (86, Ballot:80).	`segment.conditions` is a list of **Condition** (86, Ballot:80) ⇒ `condition` is a **Condition**.
88		`ballot` is a **Ballot** (85). `page` is a **Page** (85). `condition` is a **Condition** (87).	All the conditions in `segment.conditions` are valid.
89		`segment` is a **Segment** (86). ⚠ if assertion fails.	`segment.type` is 0, 1, 2, 3, or 4.
90		`ballot.audio.clips` is a list of **Clip** (1, Ballot:8, Ballot:96).	`segment.clip_i` is a valid clip index.
92		`ballot` is a **Ballot** (85). `page` is a **Page** (85). `segment` is a **Segment** (85).	The segment's `group_i` and `option_i` form a valid option reference. `group` is the referenced **Group**.
94		`segment.clip_i` is an int (86, Ballot:82). `group.option_clips` is an int (92, Ballot:24).	If `type` is 1 or 2, `segment.clip_i` is a valid option clip offset for the referenced group (92, 93).
96		`ballot.audio.clips` is a list (1, Ballot:8, Ballot:96). `segment.clip_i` is an int (86, Ballot:82). `group.max_sels` is an int (92, Ballot:22).	If `type` is 3 or 4, `segment.clip_i` + `max_sels` is a valid clip index for the referenced group (92, 95).
97	`ballot` is a **Ballot**. `page` is a **Page**. `object` is an **OptionArea**, **Condition**, **Step**, or **Segment**.		Either `object.group_i` is a valid group index and `object.option_i` is a valid option index in that group, or `object.group_i` is `None` and `object.option_i` is a valid option area index in `page`. Returns the **Group** referenced by the object's `group_i` and `option_i` (100, 102).
99		`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 the given page. `area` is an **OptionArea**.
100		`ballot.model.groups` is a list (1, Ballot:6, Ballot:17). `area` is an **OptionArea** (99). ⚠ if `area.group_i` is out of bounds.	The referenced option area's group is returned.
101		`ballot.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.	If `group_i` is not `None`, then `group_i` is a valid group index and `option_i` is a valid option index in that group.
102		`ballot.model.groups` is a list (1, Ballot:6, Ballot:17). `group_i` is a valid group index (101).	The referenced group is returned.
103	`a` is a **Video**, **Image**, or **Rect**. `b` is a **Video**, **Image**, or **Rect**.		`a` and `b` have equal `width` and equal `height`.
104		`a.width` and `b.width` are ints (108, Ballot:103, Ballot:114, Ballot:120). `a.height` and `b.height` are ints (108, Ballot:104, Ballot:115, Ballot:121).	

verifier.py (page 4 of 4)

```
def verify_segments(ballot, page, segments):
    for segment in segments:

        for condition in segment.conditions:

            verify_condition(ballot, page, condition)

        assert segment.type in [0, 1, 2, 3, 4]
        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]


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

INV1. `OP_ADD` = `0`, `OP_REMOVE` = `1`, `OP_APPEND` = `2`, `OP_POP` = `3`, `OP_CLEAR` = `4` (1).
INV2. `SG_CLIP` = `0`, `SG_OPTION` = `1`, `SG_LIST_SELS` = `2`, `SG_COUNT_SELS` = `3`, `SG_MAX_SELS` = `4` (2).
INV3. `PR_GROUP_EMPTY` = `0`, `PR_GROUP_FULL` = `1`, `PR_OPTION_SELECTED` = `2` (3).

In an initialized Navigator object:
INV4. `self.model` is a valid **Model** (6).
INV5. `self.audio` is a **Audio.Audio** (7).
INV6. `self.video` is a **Video.Video** (7).
INV7. `self.printer` is a **Printer** (7).
INV8. `self.selections` is a list of *length*(`model.groups`) lists (8).
INV9. `self.selections[i]` always contains at most `model.groups[i].max_sels` elements (8, 82, 83).
INV10. The elements of `self.selections[i]` are always valid indexes into `model.groups[i].options` (83).
INV11. `self.page_i` is a valid page index and `self.page` is the **Page** at `self.model.pages[self.page_i]` (16).
INV12. `self.state_i` is a valid state index in the page `self.page` and `self.state` is the **State** at `self.page.states[self.state_i]` (17).

	ASSUMPTIONS	REASONS FOR VALIDITY	POSTCONDITIONS
1			**INV1**.
2			**INV2**.
3			**INV3**.
4			
5	`model` is a valid **Model**. `audio` is a **Audio.Audio**. `video` is a **Video.Video**. `printer` is a **Printer.Printer**.		
6			**INV4**.
7			**INV5**, **INV6**, **INV7**.
8		`model.groups` is a list (**INV4**, Ballot:17).	**INV8**. `self.selections` is a list of *length*(`model.groups`) empty lists.
9			
10	There is at least one page and one state in the page.	`0` is a valid page or state index.	
11			
12	Either `page_i` is `None`, or `page_i` is a valid page index and `state_i` is a valid state index in that page.		
13		`self.model.pages` is a list (**INV4**, Ballot:18).	`page_i` and `state_i` are a valid page index and state index (12).
14			
15		**INV7**, **INV8**, **INV10**.	
16		`self.model.pages` is a list (**INV4**, Ballot:18). `page_i` is a valid page index (13).	**INV11**. `self.page_i` and `self.page` are the current page.
17		`self.page.states` is a list (**INV4**, Ballot:18). `state_i` is a valid state index (13).	**INV12**. `self.state_i` and `self.state` are the current state.
18		`self.state.segments` is a list of valid `Segments` (16, verifier:14).	
19			
20		**INV9** and `model.pages` and `video.layouts` have equal length (verifier:6) ⇒ `self.page_i` is a valid layout index.	
21		The sprite at `self.state.sprite_i` is the same size as the slot at `self.state_i` (verifier:13).	
22			`slot_i` points to the next available slot after the states' slots.
23		`self.page.option_areas` is a list (**INV9**, Ballot:35).	`option_areas` is a list of **OptionArea** ⇒ `area` is an **OptionArea**.
24		`area` is an **OptionArea** (23). **INV8**. `area.group_i` is a valid group index (verifier:21).	`unselected` is `0` if this option area contains a selected option, else it is `1`.
25		`area.group_i` is a valid group index (verifier:21).	
26		`area.group_i` and `area.option_i` are a valid option reference (verifier:21).	
27		`unselected` is `0` or `1` (24). `slot_i` is this option area's slot index (22, 23, 28).	
28		This option area's slot (verifier:22), sprite `option.sprite_i` (verifier:39) and sprite `option.sprite_i + 1` (verifier:40) all have the same size (verifier:50).	The unselected or selected sprite for this option is correctly displayed in this option area.

6.2.4 Navigator.py

```
[OP_ADD, OP_REMOVE, OP_APPEND, OP_POP, OP_CLEAR] = range(5)
[SG_CLIP, SG_OPTION, SG_LIST_SELS, SG_COUNT_SELS, SG_MAX_SELS] = range(5)
[PR_GROUP_EMPTY, PR_GROUP_FULL, PR_OPTION_SELECTED] = range(3)

class Navigator:
    def __init__(self, model, audio, video, printer):

        self.model = model
        [self.audio, self.video, self.printer] = [audio, video, printer]
        self.selections = [[] for group in model.groups]
        self.page_i = None
        self.goto(0, 0)
        self.update()

    def goto(self, page_i, state_i):
        if page_i != None and self.page_i != len(self.model.pages) - 1:
            if page_i == len(self.model.pages) - 1:
                self.printer.write(self.selections)
            [self.page_i, self.page] = [page_i, self.model.pages[page_i]]

            [self.state_i, self.state] = [state_i, self.page.states[state_i]]

            self.play(self.state.segments)

    def update(self):
        self.video.goto(self.page_i)

        self.video.paste(self.state.sprite_i, self.state_i)

        slot_i = len(self.page.states)

        for area in self.page.option_areas:

            unselected = area.option_i not in self.selections[area.group_i]

            group = self.model.groups[area.group_i]
            option = group.options[area.option_i]
            self.video.paste(option.sprite_i + unselected, slot_i)
            slot_i = slot_i + 1
```

	ASSUMPTIONS	REASONS FOR VALIDITY	POSTCONDITIONS
29		`self.page.counter_areas` is a list (**INV11**, Ballot:36).	`counter_areas` is a list of **CounterArea** ⇒ `area` is an **CounterArea**.
30		`area` is a **CounterArea** (29). **INV8**. `area.group_i` is a valid group index (verifier:21).	**INV9** ⇒ `count` is an int ≤ `groups[area.group_i].max_sels`.
31		`count` is an int from `0` to `max_sels`. `slot_i` is this counter area's slot index (22, 23, 28, 29, 32). This counter area's slot matches the size of all the sprites from `area.sprite_i` to `area.sprite_i` + `max_sels` (verifier:26).	The counter area displays the correct sprite indicating the number of selections in its group.
32			
33		`self.page.review_areas` is a list (**INV11**, Ballot:37).	`review_areas` is a list of **ReviewArea** ⇒ `area` is an **ReviewArea**.
34		`area.group_i` is a valid group index (verifier:29). `slot_i` is this review area's first slot index (22, 23, 28, 29, 32, 33, 34). $\forall\, k \in \{0, 1, \ldots, \texttt{max\_sels} - 1\}$, $\texttt{slot\_i} + k \times (1 + \texttt{max\_chars})$ is the valid index of a slot with size matching the group's options' sprites (verifier:30–34, verifier:39, verifier:49–50). `area.cursor_sprite_i` is a valid sprite index or `None` (verifier:36).	The review area is properly populated with options. `slot_i` is the first slot after this review area's slots.
35	`group_i` is a valid group index. $\forall\, k \in \{0, 1, \ldots, \texttt{max\_sels} - 1\}$, $\texttt{slot\_i} + k \times (1 + \texttt{max\_chars})$ is the valid index of a slot with size matching the group's options' sprites. `cursor_sprite_i` is `None` or a valid sprite index.		The review area shows the selections in its group, with write-in text for any selected write-in options. Returns $\texttt{slot\_i} + \texttt{max\_sels} \times (1 + \texttt{max\_chars})$ (47).
36		`group_i` is a valid group index (35).	`group` is the review area's group.
37		`group_i` is a valid group index (35).	`selections` is the group's selections.
38		`group.max_sels` is an int (35, Ballot:22).	`i` is an int from `0` to $\texttt{max\_sels} - 1$.
39		`i` is an int (38). `selections` is a list (37).	
40		`i` is a valid index into `selections` (39). `selections[i]` is a valid index into `group.options` (36, 37, **INV10**).	`option` is a selected **Option** in group `group_i` (Ballot:25).
41		`option.sprite_i` is a valid sprite index (verifier:39). `slot_i` is a valid slot index of equal size (35, 46).	The review area shows the sprites for the selected options in its group.
42			
43		`writein_group_i` is a valid group index (verifier:44, 42). That group has `max_chars` = `0` (verifier:43, verifier:45) and `max_sels` = `group.max_chars` (verifier:46). $\forall\, k \in \{0, 1, \ldots, \texttt{group.max\_chars} - 1\}$, $\texttt{slot\_i} + 1 + k$ is the valid index of a slot with size matching the write-in group's options' sprites (verifier:31–34, verifier:47–48, verifier:51–52).	The review area shows the write-in characters for this selected option.
44			
45		`cursor_sprite_i` is a valid sprite index (35, 44). The cursor sprite has the same size as slot `slot_i` (verifier:30, verifier:36, verifier:50).	
46			`slot_i` is the first slot for the next option in this review area.
47			$\texttt{slot\_i} + \texttt{max\_sels} \times (1 + \texttt{max\_chars})$ is returned (38, 46).
48	`key` is an int.		The operative binding, if any, for this keypress is invoked. Returns `None`.
49		`state.bindings` is a list (**INV11**, Ballot:42). `page.bindings` is a list (**INV10**, Ballot:33).	The lists contain only valid **Binding**s (verifier:10, verifier:16) ⇒ `binding` is a valid **Binding**.
50		`binding.key` is an int (49, Ballot:60). `binding.conditions` is a list of valid **Condition**s (49, Ballot:62, verifier:72).	
51		`binding` is a valid **Binding** (49).	If `binding` is operative, it is invoked. Returns `None` (69).
52	`target_i` is an int.		The operative binding, if any, for this target is invoked. Returns `None`.
53		`state.bindings` is a list (**INV11**, Ballot:42). `page.bindings` is a list (**INV10**, Ballot:33).	The lists contain only valid **Binding**s (verifier:10, verifier:16) ⇒ `binding` is a valid **Binding**.
54		`binding.target_i` is an int (53, Ballot:61). `binding.conditions` is a list of valid **Condition**s (53, Ballot:62, verifier:72).	
55		`binding` is a valid **Binding** (53).	If `binding` is operative, it is invoked. Returns `None` (69).

Navigator.py (page 2 of 4)

```
        for area in self.page.counter_areas:

            count = len(self.selections[area.group_i])

            self.video.paste(area.sprite_i + count, slot_i)
            slot_i = slot_i + 1

        for area in self.page.review_areas:

            slot_i = self.review(area.group_i, slot_i, area.cursor_sprite_i)


    def review(self, group_i, slot_i, cursor_sprite_i):


        group = self.model.groups[group_i]
        selections = self.selections[group_i]
        for i in range(group.max_sels):
            if i < len(selections):
                option = group.options[selections[i]]

                self.video.paste(option.sprite_i, slot_i)
                if option.writein_group_i != None:
                    self.review(option.writein_group_i, slot_i + 1, None)


            if i == len(selections) and cursor_sprite_i != None:
                self.video.paste(cursor_sprite_i, slot_i)
            slot_i = slot_i + 1 + group.max_chars

        return slot_i

    def press(self, key):

        for binding in self.state.bindings + self.page.bindings:

            if key == binding.key and self.test(binding.conditions):

                return self.invoke(binding)

    def touch(self, target_i):

        for binding in self.state.bindings + self.page.bindings:

            if target_i == binding.target_i and self.test(binding.conditions):

                return self.invoke(binding)
```

	ASSUMPTIONS	REASONS FOR VALIDITY	POSTCONDITIONS
56	`conditions` is a list of valid **Condition**s.		Returns `1` if all the conditions are met, otherwise `0` (67, 68).
57		`conditions` is a list (56).	`conditions` is a list of valid **Condition**s (56) ⇒ `cond` is a valid **Condition**.
58		`cond` is a valid **Condition** (58).	`group_i` and `option_i` are a valid group index and option index (118).
59		`cond.predicate` is `0`, `1`, or `2` (verifier:79). `PR_GROUP_EMPTY` is `0` (**INV3**).	
60		`group_i` is the valid index of a list in `self.selections` (56, **INV8**).	`result` is `1` if the group is empty, otherwise `0`.
61		`cond.predicate` is `0`, `1`, or `2` (verifier:79). `PR_GROUP_FULL` is `1` (**INV3**).	
62		`group_i` is a valid group index (58).	`max` is an int (**INV4**, Ballot:22).
63		`group_i` is the valid index of a list in `self.selections` (58, **INV8**).	`result` is `1` if the group is full, otherwise `0`.
64		`cond.predicate` is `0`, `1`, or `2` (verifier:79). `PR_OPTION_SELECTED` is `2` (**INV3**).	
65		`group_i` is the valid index of a list in `self.selections` (58, **INV8**).	`result` is `1` if the option is selected, otherwise `0`.
66		`cond.invert` is `0` or `1` (verifier:81).	
67			0 is returned if any condition is not met.
68			1 is returned if no condition is not met.
69	`binding` is a valid **Binding**.		`binding` is invoked. Returns `None`.
70		`binding.steps` is a list (Ballot:63).	`binding.steps` is a list of **Binding** (Ballot:63) ⇒ `step` is a **Step**.
71		`step` is a **Step** (70).	
72		**INV5**.	
73		`binding.segments` is a list of valid **Segment**s (69, verifier:76).	
74		Either `next_page_i` is `None` or `next_page_i` and	
75		`binding.next_state_i` are a valid page index and state index (69, verifier:77).	
76	`step` is a **Step**.		The step is executed. Returns `None`.
77		`step` is a **Step** (76) with a valid option reference (verifier:75).	`group_i` and `option_i` are the group and option referenced by `step` (118).
78		`group_i` is a valid group index (77).	`group` is the step's group.
79		`group_i` is a valid index into `self.selections` (77, **INV8**).	`selections` is the group's selections.
80		`option_i` is an int (77). `selections` is a list (79).	`selected` is `1` if the referenced option is selected, otherwise `0`.
81		`step.op` is `0`, `1`, `2`, `3`, or `4` (verifier:74). `OP_ADD` is `0` and `OP_APPEND` is `2` (**INV1**).	
82		`selections` is a list (79). `group.max_sels` is an int (78, Ballot:22).	
83		`selections` is a list (79). `option_i` is an int (77).	`option_i` is added to the selections for `group_i`.
84		`step.op` is `0`, `1`, `2`, `3`, or `4` (verifier:74). `OP_REMOVE` is `1` (**INV1**).	
85		`selections` is a list (79). `option_i` is an int (77).	`option_i` is removed from the selections for `group_i`.
86		`step.op` is `0`, `1`, `2`, `3`, or `4` (verifier:74). `OP_POP` is `3` (**INV1**).	
87		`selections` is a non-empty list (79, 86).	The last item is removed from this group's `selections`.
88		`step.op` is `0`, `1`, `2`, `3`, or `4` (verifier:74). `OP_CLEAR` is `4` (**INV1**).	
89		`group_i` is a valid index into `self.selections` (77, **INV8**).	This group's `selections` are cleared.

Navigator.py (page 3 of 4)

```
    def test(self, conditions):

        for cond in conditions:


            [group_i, option_i] = self.get_option(cond)

            if cond.predicate == PR_GROUP_EMPTY:
                result = len(self.selections[group_i]) == 0
            if cond.predicate == PR_GROUP_FULL:
                max = self.model.groups[group_i].max_sels

                result = len(self.selections[group_i]) == max
            if cond.predicate == PR_OPTION_SELECTED:
                result = option_i in self.selections[group_i]
            if cond.invert == result:
                return 0
        return 1

    def invoke(self, binding):
        for step in binding.steps:
            self.execute(step)
        self.audio.stop()
        self.play(binding.segments)
        self.goto(binding.next_page_i, binding.next_state_i)
        self.update()

    def execute(self, step):
        [group_i, option_i] = self.get_option(step)

        group = self.model.groups[group_i]
        selections = self.selections[group_i]
        selected = option_i in selections

        if step.op == OP_ADD and not selected or step.op == OP_APPEND:
            if len(selections) < group.max_sels:
                selections.append(option_i)
        if step.op == OP_REMOVE and selected:
            selections.remove(option_i)

        if step.op == OP_POP and len(selections) > 0:
            selections.pop()
        if step.op == OP_CLEAR:
            self.selections[group_i] = []
```

	ASSUMPTIONS	REASONS FOR VALIDITY	POSTCONDITIONS
90			The current state's timeout segments are played, if any, and transition is followed, if any.
91		`timeout_segments` is a list of valid **Segment**s (**INV12**, Ballot:43, verifier:17).	
92		Either `timeout_page_i` is `None` or `timeout_page_i` and `timeout_state_i` are a valid page index and state index (**INV12**, verifier:18).	
93			
94	`segments` is a list of valid **Segment**s.		The sequence of `segments` is played.
95		`segments` is a list (94).	`segments` is a list of valid **Segment**s (94) ⇒ `segment` is a valid **Segment**.
96		`segment.conditions` is a list of valid **Condition**s (95, Ballot:80). `self.test` returns 0 or 1 (verifier:54).	
97		`segment.type` is `0`, `1`, `2`, `3`, or `4` (verifier:89). `SG_CLIP` is `0` (**INV2**).	
98		`segment.clip_i` is a valid clip index (verifier:90).	
99			
100		`segment` is a valid **Segment** (95).	`group_i` and `option_i` are a valid group index and option index (118).
101		`group_i` is a valid group index (100).	`group` is the segment's group.
102		`group_i` is a valid group index (100).	`selections` is the group's selections.
103		`segment.type` is `0`, `1`, `2`, `3`, or `4` (verifier:89). `SG_OPTION` is `1` (**INV2**).	
104		`option_i` is a valid option index (100). `segment.clip_i` < `group.option_clips` (verifier:94).	
105		`segment.type` is `0`, `1`, `2`, `3`, or `4` (verifier:89). `SG_LIST_SELS` is `2` (**INV2**).	
106		`selections` is a list (102).	`selections` contains valid option indices (**INV10**) ⇒ `option_i` is a valid option index.
107		`option_i` is a valid option index (106). `segment.clip_i` < `group.option_clips` (verifier:94).	
108		`segment.type` is `0`, `1`, `2`, `3`, or `4` (verifier:89). `SG_COUNT_SELS` is `3` (**INV2**).	
109		*length*(`selections`) ≤ `max_sels` (**INV9**) and `segment.clip_i` + `max_sels` is a valid clip index (verifier:96) ⇒ `segment.clip_i` + *length*(`selections`) is a valid clip index.	
110		`segment.type` is `0`, `1`, `2`, `3`, or `4` (verifier:89). `SG_MAX_SELS` is `4` (**INV2**).	
111		`segment.clip_i` + `max_sels` is a valid clip index (verifier:96).	
112	`option` is an **Option**. `0` ≤ `offset` < `group.option_clips` for the option's group.		The clip for `option` at offset `offset` is played; if the option is a write-in option, the clips for the selected write-in characters at offset 0 are also played.
113		`option.clip_i` + `group.option_clips` − `1` is a valid clip index (verifier:42) and `offset` < `group.option_clips` (112) ⇒ `option.clip_i` + `offset` is a valid clip index.	
114			
115		`option.writein_group_i` is a valid group index (verifier:44).	`writein_group` is a **Group** (**INV4**, Ballot:17).
116		`option.writein_group_i` is a valid index into `self.selections` (verifier:44, **INV8**).	`option_i` is a valid option index in `writein_group` (**INV10**.)
117		`option_i` is a valid option index in `writein_group` (116). `clip_i` is a valid clip index (verifier:41, verifier:42).	
118	`object` is a valid **Condition**, **Step**, or **Segment** contained within `self.page`.		Returns a list of two ints `[group_i, option_i]` where `group_i` is a valid group index and `option_i` is a valid option index in that group (121, 122).
119		`object.group_i` is an int or `None` (118).	
120		`object.group_i` is `None` (119) and `object` is contained within `self.page` (118) ⇒ `object.option_i` is a valid option area index in `self.page` (verifier:75, verifier:80, verifier:92, verifier:99).	
121			`area.group_i` is an int (Ballot:48) ⇒ a valid group index and option index are returned (verifier:21).
122			`object.group_i` is an int (119) ⇒ a valid group index and option index are returned (verifier:75, verifier:80, verifier:92, verifier:99).

Navigator.py (page 4 of 4)

```
    def timeout(self):
        self.play(self.state.timeout_segments)
        self.goto(self.state.timeout_page_i, self.state.timeout_state_i)
        self.update()

    def play(self, segments):
        for segment in segments:
            if self.test(segment.conditions):

                if segment.type == SG_CLIP:
                    self.audio.play(segment.clip_i)
                else:
                    [group_i, option_i] = self.get_option(segment)

                    group = self.model.groups[group_i]
                    selections = self.selections[group_i]

                if segment.type == SG_OPTION:
                    self.play_option(group.options[option_i], segment.clip_i)

                if segment.type == SG_LIST_SELS:
                    for option_i in selections:
                        self.play_option(group.options[option_i], segment.clip_i)

                if segment.type == SG_COUNT_SELS:
                    self.audio.play(segment.clip_i + len(selections))


                if segment.type == SG_MAX_SELS:
                    self.audio.play(segment.clip_i + group.max_sels)

    def play_option(self, option, offset):

        self.audio.play(option.clip_i + offset)

        if option.writein_group_i != None:
            writein_group = self.model.groups[option.writein_group_i]

            for option_i in self.selections[option.writein_group_i]:

                self.audio.play(writein_group.options[option_i].clip_i)

    def get_option(self, object):

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

            return [area.group_i, area.option_i]


        return [object.group_i, object.option_i]
```

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).

	ASSUMPTIONS	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			`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 `play()` method of **Sound** returns a **Channel**. `AUDIO_DONE` is an int (2).	
17			
18			
19			
20	`rate` is an int. `data` is a string.		Returns a **Sound** (24).
21			
22		`rate` is an int (22). `putint` returns a string (31).	`fmt` is a string (21).
23		`fmt` and `data` are strings (24, 22).	
24		`file` is a string (25). `Buffer` yields an object with a `read` method. 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. `putint` returns a string (29).	
27	`n` is an int.		Returns a 4-byte string (29).
28			`n` is an int (27) ⇒ `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 a string of `length` bytes (35).
34		`self.pos` is an int (32).	
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 + putint(rate) + putint(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 + putint(len(contents)) + contents

def putint(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` (9).
INV5. `self.layout` is a **Layout** (13).

	ASSUMPTIONS	REASONS FOR VALIDITY	POSTCONDITIONS
1			`pygame` is bound to the Pygame module.
2	`im` is a **Ballot.Image**.		Returns a Pygame **Image**.
3		`im` is a **Ballot.Image** (2). `im.pixels` has length `im.width` × `im.height` × 3 (verifier:69). `im.width` and `im.height` are nonzero (verifier:69).	`pygame.image.fromstring` returns a Pygame **Image**.
4			
5	`video` is a **Ballot.Video**.		
6		`video` is a **Ballot.Video** (6).	`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:64).	
15	`sprite_i` is a valid sprite index. `slot_i` is a valid slot index in the current layout.		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:66–67).	
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**).	`self.layout.targets` is a list of **Target** (**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).

	ASSUMPTIONS	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 `options` is a list of valid option indices in that group (4).
6			
7		`group_i` is a valid index into `self.text.groups` (5, **INV1**, verifier:5).	`group` is a **TextGroup** (Ballot:87).
8		`line` is a string with length at least 55 (8, 9, 10).	
9			
10			
11		`group.writein` is an int (7, Ballot:91).	
12		`options` is a list (5).	`option` is a valid option index for group `group_i` (4).
13		`option` is a valid index into `group.options` (12, verifier:56). `group.options` is a list of strings (Ballot:92). `line` is a string (8, 10).	
14			
15			
16		`options` is a list (5).	`option` is a valid option index for group `group_i` (4).
17		`option` is a valid index into `group.options` (12, verifier:56). `group.options` is a list of strings (Ballot:92). `line` is a string (8, 10, 18).	
18			
19			
20			

6.2.7 Printer.py

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

    def write(self, selections):

        for [group_i, options] in enumerate(selections):

            if len(options):
                group = self.text.groups[group_i]
                line = group.name + ":"
                while len(line) < 55:
                    line = line + " "
                if group.writein:
                    for option in options:
                        line = line + group.options[option]
                    print line
                else:
                    for option in options:
                        print line + group.options[option]
                        line = " "*55
                print
        print "\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.

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:9). In the `goto` method, `page_i` is zero (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 (pvote:8). This `printer` is immediately passed to `navigator` and never referenced again in pvote.py. The **Navigator** assigns the incoming `printer` to `self.printer`, which is only ever referenced once (Navigator:15). 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:16). Thus, any transition to the last page must call `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 added to the current selection (Navigator:83).

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 current selection (Navigator:83). 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, the same option cannot appear twice in a group's selection list unless `OP_APPEND` is used. If the ballot definition is examined, it can be confirmed that `OP_APPEND` is used only in write-in groups but never in contest groups.

7.6 Summary of responsibilities established

R1. Not abort during a voting session.

The annotations in the source code identify all the possible places where a runtime error can occur. These appear in the verifier and in 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 runtime error cannot occur at a later point.

R2. Remain responsive during a voting session.

The only two looping constructs in Pthin are `while` and `for`.

There are only two occurrences of `while` in Pvote:

- The main event loop runs forever (pvote:10). But this does not cause unresponsiveness, since each time the loop executes it is responding to an event.
- `while len(line) < 55` runs until a string reaches 55 characters (Printer:9). Each iteration of the loop adds a space to the string and does nothing else, so this loop executes at most 55 times.

In the code that runs after a voting session has started (i. e. not including the ballot loader, verifier, audio initialization, or video initialization) there are the following uses of `for` loops:

- To update the display, the navigator iterates over the option areas (Navigator:23), performing one paste per option area.
- To update the display, the navigator iterates over the counter areas (Navigator:29), performing one paste per counter area.

- To update this display, the navigator iterates over the review areas (Navigator:33), calling `self.review` for each one.
- The `review` method in **Navigator** iterates up to the maximum number of selections in the group, performing at most one paste and making at most one recursive call to `self.review` each time.
- The recursive call to `review` passes a write-in group as `group_i`. Since a write-in group cannot have any options that themselves have write-in groups (verifier:46), recursion cannot proceed more than one level deep.
- The lists of bindings in the current state and page are scanned for a match to a keypress (Navigator:49) or a target touch (Navigator:53).
- The navigator iterates of a list of conditions to test all the conditions (Navigator:57).
- The navigator iterates over a list of steps to execute the steps (Navigator:70).
- The navigator iterates over a list of segments to play audio (Navigator:95), over a list of selections to play the audio for lists of options (Navigator:106) or for write-in options (Navigator:116). The audio is queued immediately without waiting for it to play.
- The printer iterates over groups to print their selections (Printer:5).
- The printer iterates over selected options to print contest options (Printer:12) or write-in characters (Printer:16).
- The video driver iterates over the current list of targets to determine whether the touched point falls within a target (Video:19).

The recursive call in the `review` method is the only recursive call. The call graph otherwise contains no cycles.

Therefore, to the extent permitted by the size of lists in the ballot definition, Pvote always responds to an event within a small and bounded amount of time.

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. When this happens, the navigator sets `self.page_i` to `len(self.model.pages) - 1` (Navigator:14–16). Thereafter, the page and state can never be changed again, because these changes can only happen (Navigator:16–17) if `self.page_i != len(self.model.pages) - 1`.

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` (pvote:14), `touch` (pvote:19), and `timeout` (pvote:23), 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.

R5. Exhibit the same deterministic behaviour in all voting sessions that use the same ballot definition.

By design, Pvote restarts for each voting session. It does not access the clock or any sources of randomness, so its behaviour is deterministic except for any non-determinism introduced by the incoming event stream. The event stream interleaves timers with user input, so it is sensitive to race conditions, but, given the same event stream and ballot definition, Pvote will always exhibit the same behaviour.

R6. Present instructions, contests, and options as specified in the ballot definition.

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.

R7. Navigate among instructions, contests, and options as specified in the ballot definition.

Navigation occurs only by the `goto` method, which is called whenever a binding is invoked (Navigator:74) and whenever a timeout is received (Navigator:92). 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:13–17).

R8. Select and deselect options according to user actions as specified in the ballot definition.

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:77–89).

R9. Prevent overvotes.

This is established in Section 7.4.

R10. Correctly indicate whether options are selected when the ballot definition calls for such indication.
R11. Correctly indicate how many options are selected when the ballot definition calls for such indication.
R12. Correctly indicate which options are selected when the ballot definition calls for such indication.

The navigator calls its own `update` method every time any binding is invoked (Navigator:75) or a timeout is received (Navigator:93). The `update` method always redraws everything on the screen. It first pastes the current layout's full-screen image (Navigator:20). 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 selects 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. This method pastes an option sprite in only one place (Navigator:41) and a cursor sprite in one place (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`.

R13. Commit the selections the voter made.

For this we must establish three things:

1. Selection and deselection of options indeed occurs correctly according to user actions. This is argued for R9.
2. Ballot commitment occurs when intended. To ensure this we can examine the ballot definition to see that keys and targets that cause transitions to the last page are clearly identified to the voter, and that there is adequate confirmation before a key or target that goes to the last page becomes available.
3. The printed selections are accurate. Printing occurs in the `write` method (Printer:5–18). Every group with a nonzero number of selected options causes the main clause to be executed (Printer:7–18). For a write-in group, the options are printed on the same line (Printer:13). For a contest group, the options are printed on separate lines (Printer:17).

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: Of a ballot, for the selection of votes to be complete. 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: Of a group, contest, or write-in, having no options selected.

full: Of a group, contest, or write-in, having the maximum options selected.

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

invoke: Of a binding, to carry out the response it specifies.

match: Of a binding, for its stimulus to match the user input actually received.

operative: Of a binding, to match user input and have its condition be 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 using a particular 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 $\times$ 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.