

# DRAFT Formal Specification of the Colorado Risk Limiting Audit (CORLA) Tool

Joseph R. Kiniry and Daniel M. Zimmerman  
with input from  
Neal McBurnett, Stephanie Singer, and Joey Dodds

July 2017

## 1 Introduction

The RLA Tool is a computer system developed by Free & Fair for the Colorado Department of State (CDOS henceforth). The RLA Tool facilitates running a risk limiting audit across several jurisdictions. In the case of Colorado, it facilitates running risk limiting audits across all counties in the state simultaneously.

### 1.1 Colorado Statutes and Rules

The Colorado election law pertaining to election audits is Section 1-7-515, C.R.S., where “C.R.S.” denotes “Colorado Revised Statutes”.

The rules pertaining to election audits are found in Code of Colorado Regulations, Secretary of State, State of Colorado, Department of State Elections Rules **8 CCR 1505-1**. We will call that document *Rules 1505-1* for the purpose of traceability in this specification.

The contract for this system was solicited while these rules were still under development. The final rules were adopted during week seven of the project. The initial version of these specifications was written against “Revised Draft of Proposed Rules” (**CCR 1505-1**) dated July 6, 2017. We will call that document *RDPR-6-Jul-2017* for the purpose of traceability in this specification.

Section 25, Post-Election Audit, of these documents is the salient portion of the rules for our purposes.

This domain model is based upon that law, its rules, the procurement contract between CDOS and Free & Fair, our bibliography of publications about risk-limiting audits, etc.

The core of this document is a formal specification of the RLA tool. The specification documents our thinking and the system design, from domain modeling and engineering all of the way to formal specification and verification. In

keeping with the spirit of open source development, the publication of this specification should support any interested parties with the requisite technological skills to deploy, manage maintain or evolve the system.

This document is annotated to help readers who are not expert in formal system specification understand the scope and interpretations of the elements of the specification. The prose annotations will help elections domain experts assess the completeness and correctness of the RLA tool built for Colorado.

## 1.2 Technical Context

The RLA Tool is a client-server system. The server is implemented in Java and runs on servers hosted by CDOS. The client is implemented in TypeScript and JavaScript and runs in modern web browsers. CDOS provides the hosting servers, network, and a number of services in support of the tools deployment and management.

This document is written in the literate style promoted by Donald Knuth, using the PVS theorem prover. Writing system specifications in this fashion is called (formal) domain engineering. It allows the production of typeset books and interactive hypertext directly from the specification. In order to write literate PVS we use our old friend Adriaan de Groot's scripts, available at <http://www.cs.kun.nl/~adridg/research/PVS-literate.html>. The formal domain model is written in PVS's higher-order logic (HOL). There is a corresponding informal domain model written in the Free & Fair System Specification Language, or FAFSSL for short. FAFSSL is a daughter of Extended BON which was, in turn, a daughter of <http://www.bon-method.com/>, the Business Object Notation.

In order to relate PVS to FAFSSL, we must define a refinement relationship between their two type systems. Informally, that mapping is described in the following paragraph.

We map PVS public theories and their contents to FAFSSL constructs. The top-level corla theory maps to the FAFSSL system specification, theories map to clusters, (PVS) types map to (FAFSSL) types, and functions map to features.

The formal domain model is annotated with structured comments in precise natural language using a standard set of annotations. A shell script processes these annotations and generates a well-typed informal FAFSSL specification. See our Bibliography project at GitHub, <https://github.com/FreeAndFair/Bibliography>, for more information.

## 2 Formal Model Description

### 2.1 Basic Technical Infrastructure

It is necessary to summarize a number of background concepts from computer science in order to formalize the RLA Tool. The Free & Fair System Specification Language (FAFSSL) annotations we use are as follows:

**system** the FAFSSL name of the system

**cluster** the FAFSSL name of a cluster

**description** a short description of an artifact

**explanation** the (potentially longer) explanation for an artifact

**indexing-CLAUSE** a prefix for any indexing <CLAUSE>

These suffixes are commonly used for indexing:

**author** an author of an artifact

**organization** an organization responsible for an artifact

**keywords** a comma-separated list of keywords

**created** the creation date for an artifact

**github** the URL for the GitHub project containing an artifact

## 2.2 Kinds of Concepts

Three kinds of concepts, each of which is a part of speech, are introduced in any specification:

- **Nouns** are formalized by regular (composite) types. The set of types available in the prelude of PVS is large, ranging from booleans to real-world concepts like time and tokenizing leers. Composite types are product types such as tuples and records.
- **Verbs** are formalized by function types. PVS's HOL supports higher order dependent type declarations, including higher order function types. Thus, even the most complex verb forms can be formalized in the model.
- **Adjectives and adverbs** are formalized by defining function types that apply to formalizations of nouns and verbs, respectively. Often these types are either predicate types (i.e., something is true or not), metric types encoded in enumerations or ordered structures such as numbers (i.e., how heavy something is), or are enumeration types (i.e., what properties something has, but with no particular ordering, such as the color of something).

Consequently, when we formalize a concept from some background informal information, we identify all domain specific nouns, verbs, adjectives, and adverbs in that source material and capture the meaning of each and every idea relevant to the system we are defining. This identification and formalization process is iterative, with the precision of the formalization generally increasing as we refine the model of the system and being defining its implementation.

## 2.3 Refinement from Background Literature to Specification

The Colorado *statutes* (i.e., the law) pertaining to election audits is Section 1-7-515, of the C.R.S.. The *rules* (i.e., the interpretation of the statutes and an explanation for how to realize their goals) pertaining to such are found in the aforementioned *Rules 1505-1*. Section 25, Post-Election Audit, of that document is the salient portion of that document. We will call the draft version of that document against which we originally wrote this specification *RDPR-6-Jul-2017*, and the final version *Rules 1505-1*, for the purpose of traceability in this specification.

This domain model is based upon these statutes, their associated rules, our contract with CDOS for the RLA Tool or requirements stipulated in refinement of the contract through written exchanges with CDOS, and our bibliography of publications about risk-limiting audits and digital elections. Every concept introduced herein comes from one of those sources. That full collection of resources is known as the *background literature* for the system.

In general, if a concept is defined, its relationship to those non-technical artifacts (statue, rules, contract, etc.) is captured using a traceability annotation of the form

@trace <SOURCE ARTIFACT> Discussion of traceability relationship.

Such annotations exist directly in the source text of this document (the file `corla.pvs` in the `spec/pvs` folder of the ColoradoRLA GitHub project). They also will be included as margin comments with hyperlinks to the relevant target for refinement. If we are *refining from* requirements stated in statute or rules, then we link to those higher-level documents. If we are *refining to* implementation artifacts, then we link directly to those lower level artifacts as they exist in our source code repository.

If a concept is introduced and it has no explicitly annotated traceability relation, then the reason for its introduction should be spelled out in the discussion of that concept in the same section.

As discussed above, the set of all concepts defined and relevant to the system is the system's domain model. The short definition of each and every concept is part of what constitutes the *glossary* of this system. There may be other terms in the glossary introduced because they relate to, or are mentioned in, the background literature but need not be formalized to specify the RLA Tool's behavior.

Currently, our glossary is a hand-written Appendix in the RLA Tool Book, found in the `docs` directory of the project repository. Before the final system delivery, that book will be integrated with this formal specification and the glossary concepts that come from this formal specification will be automatically generated out of this specification. By integrating the artifacts in this fashion we more easily guarantee precision, correctness, completeness, and traceability of our specification, both formal and informal.

Another goal we have is making sure that our formal model is consistent with federal standards. In particular, the work of the NIST/EAC Interoperability Working group, Election Modeling subgroup, will likely be useful. In that work there is a BPMN2 model, pictures, XPDL, XML, and more at the NIST Election Modeling GitHub page, and the NIST main Election Modeling wiki. UML for some relevant concepts is also available via GitHub. We do not have time or resources for ensuring *consistency* between models now.<sup>1</sup>

## 2.4 Refinement from Specification to Implementation

In order to turn a specification like this one into an implementation we must take the following steps.

1. We must decide how each type is going to be concretized into the implementation the system. Generally, modules in the formalization refine into comparable modules in the implementation. For example, a cluster of ideas that must be persisted and all relate to each other often turn into a database table or an object graph. Simpler concepts often map directly to built-in primitive and library types. Function types, unsurprisingly, turn into procedures, functions, or methods.
2. We must ensure that all properties about types mandated by the specification are realized in the implementation. We use axiomatic definitions to encode types and their constraints, usually via dependent types and sometimes via literal axioms in the formal specification.

A simple example will provide an example of both of these steps. The concept of risk limit is that it is a percentage. Thus, it is a number between 0 and 100. It is a modeling decision how to represent such, but given the client has not mandated that risk limits must not be fractional percentages, such as 1.25 choose to use a real number to formalize the idea of risk limits. Thus, we encode the concept of risk limit as the type

```
risk_limit: TYPE = n : nonneg_real | n <= 100
```

which means “the concept of risk limit is equivalent to the set of all non-negative real numbers whose value is no greater than 100”.

3. We must ensure that all properties about the system and its components are realized by the system. System properties are specified using theorems. Each theorem is refined in the system to assurance artifacts. One form of assurance is testing. Runtime verification—in the form of runtime assertion checking, automatically generated tests, or hand-written tests—can check sets of cases of each theorem. Formal verification—in the form of

---

<sup>1</sup>What consistent usually means in this context is that there is a relation we can define between types in the models that is information preserving from the simpler model to the more complex one.

automatic static analysis or interactive theorem proving—can check properties wholesale in the system, ensuring that properties hold for all possible inputs in all possible environments.

Given the short period of performance for this first version of the CORLA system, we are formally annotating some of the code with JML. Doing so helps us trace and think carefully about refinement from our formal specification to our implementation. It will also permit us, after this initial burst of work is complete, automatically generate a test bench using JMLunitNG and automatically reason about the implementation using the OpenJML tools suite.

Add annotation functions  
coupled to milestones.  
-kiniry

```
FAFSSL: THEORY
BEGIN
  TBD: boolean = TRUE
  TBD_TYPE: TYPE
END FAFSSL
```

All of the data relevant to the audit is stored in a database. Databases contain tables that describe the information they contain. We need not formalize database elements in any more detail than what follows.

```
database: THEORY
BEGIN
  database_table: TYPE
  data: TYPE
  database: TYPE = setof[database_table]
  write: [database, database_table, data -> database]
  read: [database, database_table -> data]
END database
```

We also need to be able to talk about some ideas from information systems. For example, many files in election systems are syntactically just lists of comma-separated values. Email is used for communication between officials and with the public. Files are used to store information in a persistent fashion. Consequently, we introduce a few types for these ideas.

```
information_systems: THEORY
BEGIN
  comma_separated_value: TYPE
  csv: TYPE = comma_separated_value
  email: TYPE
  email_address: TYPE
  file: TYPE
```

```

network: TYPE
web_browser: TYPE
browser: TYPE = web_browser
javascript_code: TYPE
date_and_time: TYPE = string
END information_systems

```

Several kinds of servers are necessary to deploy this system. At the minimum, two of each of a web server, a time (NTP) server, a firewall, and a database server are necessary. Two servers of each class are required for redundancy to fulfill availability requirements. Preferably redundant servers would be hosted in separate facilities, with separate and independent power systems and networks.

An Election Night Reporting (ENR) server is mentioned repeatedly in the RDPR-6-Jul-2017, so we model it here.

The next few lines of the specification encode the process of establishing the necessary servers.

```

server: DATATYPE
  WITH SUBTYPES web_server,
                 application_server,
                 time_server,
                 firewall,
                 database_server,
                 enr_server
BEGIN
  IMPORTING database,
            elections,
            information_systems

  web_server?: make_web_server: web_server
  application_server?: make_application_server: application_server
  time_server?: make_time_server: time_server
  firewall?: make_firewall: firewall
  database_server?(databases: set[database]):
    make_database_server: database_server
  enr_server?: make_enr_server: enr_server
END server

```

## 2.5 Elections

A number of general elections concepts are necessary to specify the RLA Tool. Some are generic to all elections, and others are specific to Colorado.

```
elections: THEORY
BEGIN
```

Voters make the choices that determine the outcome of legitimate elections.

Voters have first and lastnames and a political party affiliation.

```
person: TYPE =
  [# firstname: string, lastname: string #]
political_party: TYPE = string
voter: TYPE = %FROM person
  [# firstname: string,
    lastname: string,
    party_affiliation: political_party #]
elector: TYPE = voter
```

Some voters are UOCAVA voters. What follows is one way to formalize such, simply stating that the type `uocava_voter` is a predicate on the type `voter`. Consequently, all voters are either UOCAVA or not.

```
uocava_voter: pred[voter]
```

Candidates run for office to represent voters. A person can be a voter, a candidate, both, or neither.

```
candidate: TYPE FROM person

option: TYPE =
  [# name: string, description: string #]
choice: TYPE = set[option]
```

Elections focus on contests, each of which represents a choice that voters can make. Making a legal choice ranges enormously across the Earth, from marking a vote for a candidate by filling in a single bubble to enumerating a total order on all choices in a contest. Within any single election, each contest has a unique name.

```
contest: TYPE =
  [# name: string,
    description: string,
    choices: choice,
    votes_allowed: nat #]

audited_contest: TYPE FROM contest
opportunistic_contest: TYPE FROM contest
full_hand_count_contest: TYPE FROM contest
```

Need to break the cycle between the contest type and the ballots theory. sfsinger suggests: contests are grouped into elections, not ballots. So define contests first. Then a ballot is created to represent a set of contests. -kiniry



After a tabulation, each contest has an outcome. For some contests, the outcome is the candidate who won; for others the outcome is a set of candidates (say, in a contest for five at-large seats where the top five vote-getters win); for ballot questions the outcome is either “yes” or “no”. The set of outcomes from all contests in a single election are the outcome of the election.

```
contest_outcome: TYPE
election_outcomes: TYPE = set[contest_outcome]
wrong_outcome: TYPE FROM election_outcomes
```

Add an axiom for contest name uniqueness. -kiniry

These types will be refined when we review the information provided by CDOS on 20 July 2017 about Dominion’s file formats. -kiniry

Tabulation results are calculated from the CVRs by the RLA tool. These outcomes and margins are used to drive the RLA algorithms. These outcomes should be checked against the `reported_tabulation_outcome`, and should match since they were generated at the same time.

```
cvr_tabulation_outcome: TYPE FROM election_outcomes
```

```
reported_tabulation_outcome: TYPE FROM election_outcomes
```

```
rla_tabulation_outcome: TYPE FROM election_outcomes
```

Revise this definition. -kiniry

Elections are defined across cohorts of voters partitioned in any number of ways—geographic, political, professional, and more. In Colorado, elections are organized across counties and the entire state.

```
county_id: TYPE = nat
county_name: TYPE = string
county: TYPE =
  [# name: county_name, id: county_id #]
county_status: TYPE =
  { no_data, cvrs_uploaded_successfully, error_in_uploaded_data }
state: TYPE = set[county]
nation: TYPE = set[state]
organization: TYPE
```

Each county has a name and a county number, both of which are unique across the state.

```
s: VAR state
c1, c2: VAR county
```

```
county_names_unique: AXIOM
(FORALL s, c1, c2:
```

```

        member(c1, s) AND member(c2, s) AND
            c1'name = c2'name
        IMPLIES c1 = c2)
county_numbers_unique: AXIOM
(FORALL s, c1, c2:
    member(c1, s) AND member(c2, s) AND
        c1'id = c2'id
    IMPLIES c1 = c2)

election_canvass: TYPE
audit_center: TYPE
ballot: TYPE % = [election, set[contest], ballot_id]
END elections

```

Elections come in many forms: public and private elections; national and local elections; etc. This RLA Tool focuses on state elections in Colorado, which we model as a set of county elections.

Note that county elections are actually *multi*-county elections, where the typical case is that the set of counties is a singleton. In other words, in the general case, contests are multi-county contests, which can have the same contest name in multiple counties, and county elections gather votes for contests from a single county.

```

election: DATATYPE
    WITH SUBTYPES county_election
BEGIN
    IMPORTING elections

    is_county_election?(name: string,
                        date: string,
                        counties: set[county],
                        contests: set[contest]):
        make_county_election: county_election
END election

```

## 2.6 Ballots

The second basic concept of elections that we need to specify is the ballot. A ballot is a (possibly flawed) representation of the choices made by a voter in all contests in the election. A ballot may be paper, or it may be electronic – for example in Colorado UOCAVA voters can return their voted ballots by email. In Colorado these electronic ballots are printed out, producing paper ballots.

In some situations, say, if a voter has marked a paper ballot with ink of a color that the scanners cannot read, Election Board workers will create a duplicate ballot from the voter's original. .

There are a variety of ways to store ballots. Some ballot storage containers are secure, others are not. Some facilitate an easy means by which to find a particular ballots, others do not. An important part of any election process is the creation of the storage containers for the ballots. We distinguish different types of containers because (a) we have witnessed different storage mechanisms in the field, (b) we understand there is not a uniform storage mechanism in Colorado, and (c) some mechanisms are secure and provide a chain-of-custody evidence and others are not. This latter point matters with regards to the meaning of *chain\_of\_custody\_secure?* and *verified?* below.

Find sources for these, other than "Dwight Shellman said so" -sfsinger

```
storage_container: DATATYPE
  WITH SUBTYPES box,
                bin,
                ballot_box
BEGIN
  IMPORTING elections

  box?(ballots:list[ballot]): make_box: box
  bin?(ballots:list[ballot]): make_bin: bin
  ballot_box?(ballots:list[ballot]): make_ballot_box: ballot_box
END storage_container
```

Another crucial element of an election is the vote-counting equipment.

```
elections_equipment: THEORY
BEGIN
  IMPORTING storage_container

  voting_system: TYPE
  dominion_voting_system: TYPE FROM voting_system

  scanner_id: TYPE = string
  scanner: TYPE =
    [# name: string, model: string, id: scanner_id #]

  vvpot: TYPE
  verified?: TYPE = pred[storage_container]
  chain_of_custody_secure?: TYPE = pred[storage_container]
```

Add an axiom for uniqueness of scanner id numbers. -kiniry

Voting system's logs are precisely defined in federal standards. Those standards are known as the "Voluntary Voting System Guidelines", or VVSG for short.

```
END elections_equipment
```

```
ballots: THEORY
BEGIN
  IMPORTING elections,
           elections_equipment,
           storage_container
```

Several of our concepts are simply whole numbers (denoted “nat”, for natural numbers, in the specification). For example, ballot identifiers (ids), batch numbers, and batch sizes are all just natural numbers, 0, 1, 2, etc.

```
ballot_id: TYPE = nat
batch_id: TYPE = string
batch_size: TYPE = nat
```

Ballots are marked by voters or ballot marking devices. A ballot mark is any kind of mark made on a ballot that is not found on a blank ballot. A voter\_marking is any mark made by a voter. An ambiguous mark is a mark for which there is ambiguity in its interpretation (voter or machine). A stray mark is a mark that is outside of the legitimate regions of a ballot, such as hesitation marks outside of the mark regions for a contest or marks made by coffee spilled on a paper ballot.

```
ballot_mark: TYPE
voter_marking: TYPE FROM ballot_mark
ambiguous_mark: TYPE FROM ballot_mark
stray_mark: TYPE FROM ballot_mark
```

Ballots come in several varieties. All ballots are represented in a digital fashion (a scan, PDF, etc.), or are paper ballots, or both. Ballots are also classified based upon where they originate, such as delivery via mail, UOCAVA ballots, early ballots, etc. Ballots are also either tabulated or not. Note that not all of these categories of ballots are mutually exclusive.

```
digital_ballot: TYPE FROM ballot
paper_ballot: TYPE FROM ballot
mail_ballot: TYPE FROM ballot
uocava_ballot: TYPE FROM ballot
early_ballot: TYPE FROM ballot
tabulated_ballot: TYPE FROM ballot
provisional_ballot: TYPE FROM ballot
property_owner_ballot: TYPE FROM ballot
original_ballot: TYPE FROM ballot
duplicated_ballot: TYPE FROM ballot
```

```

non_voter_verifiable_ballot: TYPE FROM ballot
voter_verifiable_ballot: TYPE FROM ballot
phantom_ballot: TYPE FROM ballot

```

Ballots can be in various stages of processing, as implicitly mentioned in the C.R.S.

```

verified_accepted?: pred[mail_ballot]

```

Each ballot has a single ballot style, which indicates (at least) the contests that are listed on that ballot. Ballot styles are usually encoded as natural numbers.

```

ballot_style: TYPE = nat
ballot_style?: [ballot -> ballot_style]

```

```

ballot_contest: TYPE = contest

```

Ballots are often grouped into batches.

```

batch: TYPE = set[ballot]

```

The county must secure and maintain in sealed ballot containers all tabulated ballots in the batches and order they are scanned. The county must maintain and document uninterrupted chain-of-custody for each ballot storage container. Sometimes ballots are processed, either manually or by machines, in such a way that an imprint is made on each ballot processed. By hand, a stamp and signature is sometimes used, e.g. Another example is that a scanner might automatically print a new ballot identifier on the corner of each ballot scanned. We call such a ballot an imprinted ballot.

```

imprinted_ballot: TYPE FROM ballot
ballot_certification: TYPE

```

```

number_elected: TYPE = posnat

```

```

votes_allowed: TYPE = posnat

```

END ballots

Need to clarify choices vs options vs votes. -kiniry

Need to add model information for write-in votes. -kiniry

## 2.7 Ballot Manifests

While tabulating ballots, the county must maintain an accurate ballot manifest in a form approved by the Secretary of State. At a minimum, the ballot manifest must uniquely identify for each tabulated ballot the scanner on which the ballot is scanned, the ballot batch of which the ballot is a part, the number of ballots in the batch, and the storage container in which the ballot batch is stored after tabulation. In the RLA Tool, a ballot manifest uploaded by a county must eventually be verified, using the cryptographic hash (described in a future section).

```
ballot_manifests: THEORY
BEGIN
    IMPORTING ballots,
            elections_equipment,
            information_systems

    ballot_manifest_info: TYPE =
        [county_id,
         scanner_id,
         batch_id,
         batch_size,
         storage_container]

    ballot_manifest: TYPE = list[ballot_manifest_info]
    verified?: pred[ballot_manifest]
    ballot_manifest_file: TYPE FROM file
    export_ballot_manifest: [voting_system -> ballot_manifest_file]

END ballot_manifests
```

Should this information also include what kind of ballot each ballot is? I.e., re-marked, phantom, etc.? -kiniry

These types will be refined when we review the information provided by CDOS on 20 July 2017 about Dominion's file formats. -kiniry

Ballot position number is still not explicitly specified. -kiniry

The interpretation of the meaning of a voter's choice in a single ballot contest is either a well-formed vote an overvote, or an undervote. In any contest there is a maximum number of selections a voter may make, usually one for Mayor, but often more than one for at-large seats on a Council. In a well-formed vote, the interpretation of the voter's marks is that the voter made the maximum number of allowable selections. In an undervote, the interpretation of the voter's marks is that the voter made fewer selections than allowed. Undervotes are legitimate, and the selections are added to the tallies. In an overvote, the interpretation of the voter's marks is that the voter made more selections than were allowed. In this case none of the voter's selections are added to the tallies.

```

vote: DATATYPE
  WITH SUBTYPES well_formed_vote,
                 overvote,
                 undervote,
                 no_consensus
BEGIN
  IMPORTING ballots
  well_formed_vote?(choices:set[choice]):
    make_well_formed_vote: well_formed_vote
  overvote?(choices:set[choice]): make_overvote: overvote

  undervote?(choices:set[choice]): make_undervote: undervote
  no_consensus?: make_no_consensus: no_consensus

```

Should we define undervote specifically as 0 votes? -nealmcb

Note that the Election Results Reporting project defines it as “Undervote: Occurs when the voter does not select a candidate in a 1-of-M contest or selects fewer than N candidates in an N-of-M contest.”

Should we model a blank vote separately from an undervote? Or do they simply use the term blank vote to denote an undervote? -kiniry

```

END vote

ballot_interpretation: THEORY
BEGIN
  IMPORTING ballots,
           vote

```

A Cast Vote Record (commonly called a “CVR”) is a catalog of choices from one or more ballots representing distinct voters. In the Colorado system, CVRs are produced as comma-separated-value files from the Dominion voting system.

```

cast_vote_record: TYPE = set[set[choice]]

END ballot_interpretation

ballots_collections: THEORY
BEGIN
  IMPORTING ballots

```

Ballots are often ordered, such as after they are hand or machine sorted, when they are kept in storage that maintains order, or an order is induced upon them by a random shuffle.

```

ballot_order: TYPE = sequence[ballot]

```

Sometimes, but not always, a total order is induced by ballot ids.

Introduce appropriate axiom for such. -kiniry

```
END ballots_collections
```

## 2.8 Instructions, Forms, and Reports

Elections have all kinds of different published instructions, forms, and reports. We enumerate a few here that are specific to Colorado, and relevant to the RLA Tool in particular.

```
instructions_forms_reports: THEORY
BEGIN
  IMPORTING audits,
            ballot_manifests,
            election,
            information_systems

  report: TYPE = [# name: string #]
  audit_investigation_report: TYPE =
    [# name: string, investigation_report: string #]
  empty_audit_investigation_report: audit_investigation_report

  audit_report: TYPE =
    [# name: string,
     election: election,
     audit_board: audit_board,
     county_administrator: county_administrator #]

  summary_results_report: TYPE =
    [overvotes: nat, undervotes: nat,
     blank_voted_contests: nat, valid_write_in_votes: nat]

  results_file: TYPE FROM file

  form: TYPE
  sos_audit_form: TYPE

  instructions: TYPE
  ballot_instructions: TYPE FROM instructions
  sos_voter_intent_guide: TYPE FROM instructions

  ballots_under_audit_instructions: TYPE =
    [ sequence[ballot], sequence[ballot_style],
      sequence[ballot_manifest_info] ]
```



```
END instructions_forms_reports
```

## 2.9 Roles

Within the RLA Tool, a person involved with the system can have a number of roles. Some of these roles are mutually exclusive for legal reasons.

```
roles: THEORY
BEGIN
  IMPORTING elections,
          FAFSSL
```

Audit board members are members of the public who come from different political parties. According to

```
audit_board_member: TYPE FROM person
canvass_board: TYPE = set[person]

colorado_department_of_state: TYPE
cdos: TYPE = colorado_department_of_state

sos: TYPE FROM person
state?: [sos -> state]

county_clerk: TYPE FROM person
county?: [county_clerk -> county]

administrator: TYPE FROM person
audit_supervisor: TYPE FROM person
county_administrator: TYPE FROM administrator
state_administrator: TYPE FROM administrator

system_administrator: TYPE FROM administrator

candidates_cannot_be_administrators: AXIOM TBD

END roles
```

```

canvass_boards: THEORY
BEGIN
  IMPORTING election,
          roles

  canvass_board?: [election -> canvass_board]
END canvass_boards

```

## 2.10 Cryptography

Cryptography is used, via hashing, to verify that the verified ballot manifests and CVRs uploaded by counties and stored in the file system and database have integrity and repudiation. I.e., checking a cryptographic hash of a file guarantees: (a) to the Secretary of State and state administrators that the uploaded files are the ones that county administrators thinks they are (thus, there was no error in data transmission), and (b) to the public that the uploaded files have not been altered in any fashion.

```

cryptography: THEORY
BEGIN
  digest: TYPE = bvec[256]
  sha256: [size: nat, bvec[size] -> digest]
END cryptography

```

## 2.11 Randomness

Choosing a random sample of ballots is key to the legitimacy of any risk-limiting audit. Legitimacy depends on a high-quality process for the random samples. The Colorado RLA Tool will produce a random order of all the ballots once and for all; then random samples of any size, or restricted to any County or set of Counties, can be created as needed by considering the ballots in that random order.

```

randomness: THEORY
BEGIN
  unit: TYPE

```

There are many algorithms available to produce a pseudorandom sequence of numbers. The relevant text from Colorado’s RDPR is quoted below. As stated in RDPR-6-Jul-2017 Section 25.2.2(K):

The Secretary of State will convene a public meeting on the tenth day after election day to establish a random seed for use with the Secretary of State’s RLA tool’s pseudo-random number generator based on Philip Stark’s online tool, *Pseudo-random Number Generator using SHA-256*. This material is incorporated by reference in the election rules and does not include later amendments or editions. The following material incorporated by reference is posted on the Secretary of State website and available for review by the public during regular business hours at the colorado secretary of state’s office: pseudo-random number generator using SHA-256 available at <https://www.stat.berkeley.edu/~stark/java/html/sha256rand.htm>. The Secretary of State will give public notice of the meeting at least seven calendar days in advance. The seed is a number consisting of at least 20 digits, and each digit will be selected in order by sequential rolls of a 10-sided die. The Secretary of State will randomly select members of the public who attend the meeting to take turns rolling the die, and designate one or more staff members to take turns rolling the die in the event that no members of the public attend the meeting. The Secretary of State will publish the seed on the audit center immediately after it is established.

As such, the seed of our pseudorandom algorithm must be a natural number that is at least twenty digits long.

```
seed: TYPE = {n: nat | 99999999999999999999 < n}
```

The random number generator itself must be seeded with the seed provided by the Department of State, and then after being seeded it must deterministically produce new pseudorandom numbers in a specified range.

```
rng: TYPE = [the_minimum: nat, the_maximum: nat ->
             {n: nat | the_minimum <= n AND n <= the_maximum}]
seed_prng: TYPE = [seed, with_replacement: boolean,
                   the_minimum: nat, the_maximum: nat -> rng]
```

The algorithm used by Rivest and Stark in the aforementioned sample code uses the SHA-256 cryptographic hash function in counter mode to obtain deterministic random input. Here is the relevant documentation from Rivest’s Python implementation, available via StarkRLAJS.

The cryptographic hash function SHA-256 is used in this program. This hash function maps arbitrary strings of input text to “pseudo-random” 256-bit integers. The pseudo-randomness of this function is of the highest quality: SHA-256 is a U.S. government standard and has passed the most stringent testing.

The SHA-256 hash function is used in “counter mode” to obtain the desired sample. The sample elements are picked one by one from a..b, with the i-th pick is generated by applying SHA-256 to the

Table 1: Example Refinement for CORLA Randomness Subsystem

Formal Specification Artifact	Implementation Artifact
<code>randomness</code> theory	Java class <code>us.freeandfair.corla.crypto.PseudoRandomNumber</code>
<code>seed</code>	<code>PseudoRandomNumberGenerator.my_seed</code>
<code>seed.prng(seed, replace, min, max)</code>	<code>new PseudoRandomNumberGenerator(seed, replace, min, ma</code>
<code>rng(min, max)</code>	<code>rng = new PseudoRandomNumberGenerator(seed, replace, m</code>

text string obtained by following the seed by a comma and then the decimal representation of `i`. This value reduced modulo  $(b-a+1)$  and added to `a` to obtain a value in the range `a..b`. This value is rejected if sampling is done without replacement and the value obtained is a duplicate of a previously obtained value.

We will use this `randomness` module as our first and simplest example of *refinement* from our formal specification to our implementation. The following table summarizes the refinement relationship between `randomness` and our implementation.

There are a few things to note about this refinement.

1. The size constraint on the type `seed` is expressed in the implementation via a precondition on its constructor, expressed both in JML and in an inline assert, and two class invariants, one which constrains seed's length and the other which characterizes its content. Note that the latter is necessary because we are mapping from a PVS `nat` (natural number, a non-negative integer) to a `java.lang.String` (which encodes arbitrary unicode characters).
2. A specification function, such as `rng`, is refined to a sequence of actions in the implementations. In this example, `rng` maps to a call to the constructor of `PseudoRandomNumberGenerator` followed by a call to `getRandomNumbers`.
3. The implementation often has more functionality than is specified in the formal specification. This is normal, as the specification is an abstraction of the *necessary* functionality

END `randomness`

## 2.12 Audits

Election audits come in many forms. The two main kinds of audits we focus on in this system are ballot polling audits and comparison audits, both of which

are risk-limiting audits.

```
audits: THEORY
BEGIN
  IMPORTING roles

  audit: TYPE
```

Audits are run by audit boards, whose members come from various constituencies and have various roles. Deciding who is on an audit board is also usually a matter of law, policy, and history.

```
audit_board: TYPE = set[audit_board_member]
audit_board_size: AXIOM TBD
audit_board_members_political_parties_disjoint: AXIOM TBD
```

These two terms are used but underdefined at this time. When the C.R.S. or CO law is further refined to explain audit investigations (i.e., the process by which a mismatch between the human adjudication and the machine interpretation is resolved), then we will have clarity on the notions of “audit investigation” and “audit progress”.

```
audit_investigation: TYPE
audit_progress: TYPE
```

The count of the total number of ballots to audit varies across audit types. For example, historically in Colorado random audits must audit 500 ballots or 5% of all ballots, whichever is smaller.

```
ballots_to_audit: TYPE = nat
```

We list various other terms relating to audits here. They will be defined and refined in later versions of the domain model.

```
contest_margin: TYPE = nat
margin: TYPE = real

digital_ballot_adjudication: TYPE
manual_ballot_adjudication: TYPE

diluted_margin: TYPE FROM margin
```

```

margin_overstatement: TYPE = nat

margin_understatement: TYPE = nat

random_audit: TYPE
END audits

```

### 2.13 Colorado RLAs

Audits ideally come after all the votes are tabulated, canvassed and reconciled. In Colorado, however, since the certification deadline comes shortly after the last date for voters to cure signature verification problems, etc., it is highly unlikely that a RLA could be postponed until after the canvass. An updated vote count may be released at the end of the tabulation and canvass, and the risk limit of the audit needs to apply to the updated outcome.

Given these constraints, it is generally best to audit conservatively. For example, we could assume that any late-tabulation ballots are cast for the losers. As discussed in [BanuelosEtAl12], counties should add a batch of phantom ballots to the manifest, one for each possible late-tabulation ballot. Another possibility, if it turns out that not enough phantom ballots were added, would be to use more flexible Bayes audit techniques to do a followup audit after the late-tabulation ballots and tabulations are available; however, that capability is not implemented at this time.

```

rlas: THEORY
BEGIN
  IMPORTING elections

```

First, we'll formalize the general idea of a risk limit from the scientific literature.

```

risk_limit: TYPE = {n : nonneg_real | n <= 100}

```

Next, we'll formalize what is stated in the current draft rules.

```

RDPR_risk_limit: TYPE = {n : nonneg_real | n <= 5}

```

Finally, according to the current draft rules, the Secretary of State has an “escape clause”, and one thing that they can choose to do is set any kind of risk limit they like.

```

RDPR_escape_clause_risk_limit: TYPE = {n : nonneg_real | n <= 100}

risk_limiting_audit: TYPE
RLA: TYPE = risk_limiting_audit

ballot_polling_audit: TYPE FROM risk_limiting_audit

comparison_audit: TYPE FROM risk_limiting_audit

```

These are some terms of the art that we will more carefully define as the model is refined.

```

discrepancy: TYPE
random: TYPE
sample_size: TYPE = nat

number_of_ballots_to_audit: [audited_contest -> nat]

```

END rlas

Still need to model core  
RLA algorithm(s). -  
kiniry

## 2.14 Cast Vote Records

Cast vote records, also known as CVRs, are the digital interpretations of paper ballot records by a computer. They frequently, but not always, contain an interpretation of all voter choices in all contests on a ballot. CVRs are sometimes syntactically written as comma-separated values, and other times in plain English. Some CVRs use an election-specific encoding scheme to represent choices (e.g., a ‘1’ means “John Doe”); others use plain English.

CVRs may (and in the case of CVRs exported from Colorado’s Dominion system, do) contain information about the ballot beyond the voter choices. For example, ballot style (which encodes, at a minimum, the set of contests on the ballot), precinct id and information about voting method (e.g., in-person vs. mail) may be included.

CDOS requirements mandate that CVRs are exported and uploaded to the RLA back-end as CSV files by county officials using their voting systems and the RLA Tool, respectively.

```

cast_vote_records: THEORY
BEGIN
  IMPORTING ballot_manifests,

```

For a given election, we cannot expect a 1-1 correspondence between its ballots and its CVRs. Phantom ballots and human error will result in a violation of such a property. -kiniry

```

        elections_equipment,
        information_systems,
        vote
    cvr: TYPE = [# votes: set[vote],
                description: ballot_manifest_info #]
    verified?: pred[cvr]
    as_csv: [cvr -> csv]
    cvr_file: TYPE FROM file
    export_cvr: [voting_system -> cvr_file]

    cvr_number: TYPE = nat
END cast_vote_records

```

It is unclear what a CVR number is, if anything. -kiniry

## 2.15 User Interfaces

The user interfaces of the system are the visible interactive parts of the application. There are three different user interfaces in the RLA Tool. The precise names of these interfaces are still under discussion; one is for CDOS personnel responsible for the audit at the state-level, one is for county personnel at the county-level, and one is for audit board members. Note that all these user interfaces both receive and provide information. Because the public dashboard only pushes information out, we do not consider it a “user interface”.

```

user_interface: DATATYPE
    WITH SUBTYPES uploading_interface,
                  cvr_uploading_interface,
                  county_auditing_interface,
                  audit_adjudication_interface,
                  public_interface
BEGIN
    uploading_interface?:
        make_uploading_interface: uploading_interface
    cvr_uploading_interface?:
        make_cvr_uploading_interface: cvr_uploading_interface
    county_auditing_interface?:
        make_county_auditing_interface: county_auditing_interface
    audit_adjudication_interface?:
        make_audit_adjudication_interface: audit_adjudication_interface

```

In the audit adjudication interface, it must be possible to classify a ballot as unauditable either due to not finding a voter-verifiable paper record or due to it being a phantom ballot. -kiniry



```

    public_interface?:
        make_public_interface: public_interface
END user_interface

```

After a county administrator attempts to upload artifacts to the RLA system's server, one of several different messages is shown. Each is self-explanatory in this domain model, and each must be used in a scenario of the system.

```

upload_system_message: DATATYPE
    WITH SUBTYPES upload_successful,
                    checking_hash,
                    hash_verified,
                    hash_mismatch,
                    file_type_wrong,
                    data_parsed,
                    data_transmission_interrupted,
                    too_late
BEGIN
    upload_successful?: make_upload_successful: upload_successful
    checking_hash?: make_checking_hash: checking_hash
    hash_verified?: make_hash_verified: hash_verified
    hash_mismatch?: make_hash_mismatch: hash_mismatch
    file_type_wrong?: make_file_type_wrong: file_type_wrong
    data_parsed?: make_data_parsed: data_parsed
    data_transmission_interrupted?: make_data_transmission_interrupted:
        data_transmission_interrupted
    too_late?: make_too_late: too_late
END upload_system_message

```

Authentication attempts can result in two different kinds of message: either a person authenticated successfully, or they did not.

```

authentication_message: DATATYPE
    WITH SUBTYPES successful_authentication,
                    unsuccessful_authentication
BEGIN
    successful_authentication?: make_successful_authentication:
        successful_authentication
    unsuccessful_authentication?: make_unsuccessful_authentication:
        unsuccessful_authentication
END authentication_message

```

## 2.16 Dashboards

The system has several dashboards aimed at specific users and stakeholders. Department of State officials use the state-wide dashboard; county officials use the county dashboard; audit board members use the audit board dashboard; and the general public uses the public dashboard.

```
dashboard: DATATYPE
  WITH SUBTYPES department_of_state_dashboard,
                  county_dashboard,
                  audit_board_dashboard,
                  public_dashboard
BEGIN
  IMPORTING elections
  department_of_state_dashboard?(counties: set[county]):
    make_department_of_state_dashboard:
      department_of_state_dashboard
  county_dashboard?(county: county):
    make_county_dashboard:
      county_dashboard
  audit_board_dashboard?(county: county):
    make_audit_board_dashboard:
      audit_board_dashboard
  public_dashboard?: make_public_dashboard: public_dashboard
END dashboard
```

## 2.17 Department of State Dashboard

The status of uploaded data will be summarized in a Department of State Dashboard, along with information on which counties have not yet uploaded their CVRs, and uploads that have formatting or content issues. The status of data, and results as audits are performed, will be provided for each contest to be audited.

```
department_of_state_dashboard: THEORY
BEGIN
  IMPORTING dashboard,
            elections,
            FAFSSL,
            instructions_forms_reports,
            randomness,
            rlas,
            ballot_interpretation
```

CDOS staff, after authenticating to the state-wide dashboard, can see the status of the entire election. Various static information about the election is displayed

along with dynamic information, the type and content of which is dependent upon the current audit stage.

```
audit_stage: TYPE = {
    pre_audit,
    audit_ready_to_start,
    audit_ongoing,
    audit_complete,
    audit_results_published
}
overall_audit_stage:
[department_of_state_dashboard ->
 [audit_stage, department_of_state_dashboard]]
```

State administrators can also pull up the dashboard for any county.

```
dashboard_for_county:
[department_of_state_dashboard, county ->
 county_dashboard]
```

The state-wide dashboard will provide a way for the Secretary of State to enter the risk limit(s), as required by RDPR-6-Jul-2017 Section 25.2.2(A).

No later than 30 days before Election Day, the Secretary of State will establish and publish on the audit center the risk limit(s) that will apply in RLAs for that election. The Secretary of State may establish different risk limits for comparison audits and ballot polling audits, but in no event will the risk limit exceed five percent.

```
establish_risk_limit_for_comparison_audits:
[department_of_state_dashboard, risk_limit ->
 department_of_state_dashboard]
```

Because ballot polling audits are mentioned in the current draft rules, we must formalize the associated concepts, even though we are not implementing support for ballot polling audits in this first version of the RLA Tool.

```
establish_risk_limit_for_ballot_polling_audits:
[department_of_state_dashboard, risk_limit ->
 department_of_state_dashboard]
```

The RLA status of each county is a part of the Department of State dashboard. At the moment, from a UX point of view, we are assuming that it is simply rolled into the overall dashboard, thus there is not a state administrator means by which to trigger an update of this data separate from all other election data.

```
upload_status: [department_of_state_dashboard -> set[county_status]]
```

The state-wide dashboard will provide a mechanism for the Secretary of State to enter and publish the list of contests to be audited.

No later than 5:00 PM MT on the Friday after Election Day, the Secretary of State will select for audit at least one statewide contest, and for each county at least one countywide contest. The Secretary of State will select other ballot contests for audit if in any particular election there is no statewide contest or a countywide contest in any county. The Secretary of State will publish a complete list of all audited contests on the audit center. The Secretary of State will consider the following factors in determining which contests to audit:

1. The closeness of the reported tabulation outcome of the contests;
2. The geographical scope of the contests;
3. Any cause for concern regarding the accuracy of the reported tabulation outcome of the contests;
4. Any benefits that may result from opportunistically auditing certain contests; and
5. The ability of the county clerks to complete the audit before the canvass deadline.

```
audit_reason: TYPE =
{ state_wide_contest, county_wide_contest, close_contest,
  geographical_scope, concern_regarding_accuracy,
  opportunistic_benefits, county_clerk_ability, no_audit }
select_contests_for_comparison_audit:
[department_of_state_dashboard, set[[contest, audit_reason]] ->
 [department_of_state_dashboard, set[audited_contest]]]
at_least_one_statewide_contest: AXIOM TBD
at_least_one_countywide_contest_per_county: AXIOM TBD
```

At the moment, CDOS has not contracted support for the selection of opportunistic audited contents. But, since we know that CDOS wishes to support this feature in the future, we model it for future support.

```
select_contest_for_opportunistic_audit:
[department_of_state_dashboard, contest, audit_reason ->
 [department_of_state_dashboard, opportunistic_contest]]
```

The state-wide dashboard will allow the Secretary of State to select ballots at random as required by law (assuming that a truly random seed has been entered). Ballots can be randomly selected for audit in two ways, either by:

1. permuting all ballots and auditing a prefix of ballots (thereby auditing ballots “with no replacement”); or
2. randomly selecting ballots ballot-by-ballot (thereby auditing ballots “with replacement” as the same ballot may be audited multiple times).

```

ballot_permutation:
    [set[ballot] -> list[ballot]]
random_list_of_ballots:
    [set[ballot] -> sequence[ballot]]

print_ballots_under_audit_list:
    [county, list[ballot] -> ballots_under_audit_instructions]

```

Obviously this type has to be strengthened considerably to guarantee permutation. That is, that both the set and the list are finite and converting the list into a set results in a set equivalent to the original set. -kiniry

The Secretary of State will randomly select the individual ballots to audit. The Secretary of State will use a pseudo-random number generator with the seed established under subsection (H) of this rule to identify individual ballots as reflected in the county ballot manifests. The Secretary of State will notify each county of, and publish on the audit center, the randomly selected ballots that each county must audit no later than 11:59 PM MT on the tenth day after Election Day.

Write a specification for publishing data to audit. -kiniry

```

publish_seed:
    [department_of_state_dashboard, seed -> department_of_state_dashboard]
publish_ballots_to_audit:
    [department_of_state_dashboard, set[cast_vote_record] ->
        [department_of_state_dashboard, list[[county, list[ballot]]]]]

```

The Secretary of State can indicate that a contest must be a full hand count contest.

```

indicate_full_hand_count_contest:
    [department_of_state_dashboard, contest ->
        [department_of_state_dashboard, full_hand_count_contest]]

```

Lastly, state administrators can simply get updates on the current state of the election under audit and its RLAs.

```

refresh: [department_of_state_dashboard ->
    [department_of_state_dashboard,
        audit_stage,
        risk_limit,
        set[[audited_contest, audit_reason]],
        set[county_status],
        boolean, % cvr_uploads_complete?
        boolean, % manifest_uploads_complete?
        seed,

        set[full_hand_count_contest]]]
END department_of_state_dashboard

```

### 2.17.1 County Dashboard

The County dashboard is used by county officials to communicate with the Secretary of State for the purpose of planning and executing risk-limiting audits.

```
county_dashboard: THEORY
BEGIN
    IMPORTING ballot_manifests,
              cast_vote_records,
              cryptography,
              dashboard,
              election,
              information_systems,
              roles,
              server,
              upload_system_message,
              department_of_state_dashboard
```

Some of the generation information contained in the county dashboard is stipulated by C.R.S.

```
general_static_information: string
```

The status of the county dashboard depends upon which stage of the county has reached in its audit process. The status is obtained via the following function. Briefly, either the county has uploaded no information at all (`no_data`), it has uploaded its verified CVRs successfully (`cvrs_uploaded_successfully`), or county administrators tried to upload their CVRs and there was an error of some kind (`error_in_uploaded_data`).

```
county_status:
    [county -> county_status]
```

```
establish_audit_board:
    [county_dashboard, set[elector] -> county_dashboard]
```

To prepare for uploading of artifacts to the Secretary of State, counties conducting a comparison audit must verify several properties (all of which are discussed in Section 3.2). After verifying those properties, counties must generate a digest of the CVR file using a hash designated by the Secretary of State. After verifying the accuracy of the CVR export, the county must apply a hash value to the CVR export file using the hash value utility provided by the Secretary of State.

Note that this function/feature is implemented by a tool provided by the Secretary of State. We have not been asked to produce such a tool, though

one could compute the hash of a file locally in a web browser, so this could be part of our system. From an assurance standpoint it is a better idea to use a completely separate tool, developed independently from us, to perform this hashing.

```
generate_cvr_digest: [cvr_file -> digest]
```

Each county performing a comparison audit must upload a hash (digest) of its ballot manifest to the RLA Tool.

```
generate_ballot_manifest_digest: [ballot_manifest_file -> digest]
```

Each county conducting a comparison audit must upload:

1. its verified and hashed ballot manifest to the RLA Tool;

```
county_upload_verified_ballot_manifest:
  [county_dashboard, ballot_manifest_file, digest ->
   [county_dashboard, email, upload_system_message]]
```

All ballot manifests are stored in a relation between counties and ballot manifest files. See `ballot_manifest_table` below.

2. its verified and hashed CVR export to the RLA Tool; and

```
upload_verified_cvr:
  [county_dashboard, cvr_file, digest ->
   [county_dashboard, email, upload_system_message]]
```

All CVRs are stored in a relation between counties and CVR files. See `cvr_table` below.

3. its RLA tabulation results export to the Secretary of State's election night reporting system.

```
upload_tabulation_results:
  [enr_server, cvr_tabulation_outcome -> enr_server]
```

After the audit board is established, all CVRs and ballot manifests are successfully uploaded, the county administrator can start the audit.

```
start_audit:
  [county_dashboard -> audit_board_dashboard]
```

Improve/strengthen the dependent types in these upload signatures below.  
-kiniry

@review cdos Must the Department of State give an explicit go-ahead, or can the county simply immediately start the audit?  
-kiniry

No later than 11:59 PM MT on the ninth day after Election Day, each county conducting a ballot polling audit must upload:

1. its verified and hashed ballot manifest to the RLA tool; and
2. its RLA tabulation results export to the Secretary of State's election night reporting system.

There is exactly one audit board dashboard for each county.

```
audit_board_dashboard:
  [county_dashboard -> audit_board_dashboard]
```

Note that both of these uploads are facilitated by the functions defined above, as their types do not mandate a particular kind of audit on the county. Lastly, county administrators can see updates on the current state of their audit.

```
refresh: [county_dashboard ->
  [county_dashboard,
    string, % general_static_information
    set[elector], % the audit board members
    digest, % the ballot manifest digest
    digest, % the CVR digest
    set[contest], % the contests on the ballot in this county
    set[[audited_contest, audit_reason]], % the contests under
    date_and_time, % start date and time of the audit
    nat, % number of ballots according to CVR
    nat, % total estimated number of ballots to audit according to RLA
    nat, % total number of ballots audited thus far
    nat, % total number of discrepancies (overstatements +
    nat] % total number of times the audit board could not
  ]
END county_dashboard
```

### 2.17.2 Audit Board Dashboard

There are a number of assumptions that the RLA Tool audit board dashboard makes with respect to the C.R.S. It also facilitates the comparison audit.

```
audit_board_dashboard: THEORY
BEGIN
  IMPORTING dashboard,
    audits,
    ballots,
    cast_vote_records,
    instructions_forms_reports,
    storage_container,
    department_of_state_dashboard
```



The audit board must locate and retrieve from the appropriate storage container each randomly selected ballot. The audit board must verify that the seals on the appropriate storage containers are those recorded on the applicable chain-of-custody logs.

```
ballots_to_audit_to_storage_container_list:
  [list[ballot] -> list[storage_container]]
verify_all_seals_on_storage_containers:
  [list[storage_container] -> list[storage_container]]
```

The audit board must examine each randomly selected ballot or VVPAT and report the voter markings or choices using the RLA Tool or other means specified by the Secretary of State.

```
next_ballot_for_audit:
  [audit_board_dashboard ->
   [ballot_manifest_info, audit_board_dashboard]]
```

A list of which ballots have been audited thus far is also available on the UI.

```
audit_cvrs:
  [audit_board_dashboard ->
   [list[cvr], audit_board_dashboard]]
```

We call the report of the markings on a ballot an Audit CVR, or aCVR for short.

```
aCVR: TYPE = list[ballot_mark]
report_markings:
  [audit_board_dashboard, ballot_manifest_info,
   aCVR -> audit_board_dashboard]
report_ballot_not_found:
  [audit_board_dashboard, phantom_ballot -> audit_board_dashboard]
```

If a ballot does not have a voter-verifiable paper ballot associated with it then the Audit Board reports the lack of voter-verifiable paper ballot.

```
report_ballot_has_no_voter_verifiable_paper_record:
  [audit_board_dashboard, non_voter_verifiable_ballot ->
   audit_board_dashboard]
```

If supported by the county's voting system, the audit board may refer to the digital image of the audited ballot captured by the voting system in order to confirm it has retrieved the correct ballot randomly selected for audit. If the scanned ballot was duplicated prior to tabulation, the audit board must also retrieve and compare the markings on the original ballot. The audit board must complete its reports of all ballots randomly selected for audit no later than 5:00 PM MT one business day before the canvass deadline.

The audit board must interpret voter markings on ballots selected for audit in accordance with the Secretary of State's voter intent guide.

To the extent applicable, the Secretary of State will compare the audit board's reports of the audited ballots to the corresponding CVRs and post the results of the comparison and any margin overstatements or understatements on the audit center.

```
compare_reported_markings_to_cvr:
  [ballot, cvr, list[ballot_mark] ->
    [margin_overstatement, margin_understatement]]
```

The RLA will continue until the risk limit for each audited contests is met or until a full hand count results. If the county audit reports reflect that the risk limit has not been satisfied in an audited contest, the Secretary of State will randomly select additional ballots for audit. We presume at the moment that if errors are made during the auditing process, we should capture information in the RLA Tool about those errors and their mitigation and resolution.

```
submit_audit_investigation_report:
  [audit_board_dashboard, audit_investigation_report ->
    audit_board_dashboard]
submit_audit_report:
  [audit_board_dashboard, audit_report -> audit_board_dashboard]
```

This function updates a set of relations between counties and audit reports.

```
audit_reports: set[[county, audit_report]]
```

We need to know when all audit reports have been submitted to transition the Department of State Dashboard to its DOS Audit Complete state.

```
audit_complete?: [audit_report -> boolean]
signoff_intermediate_audit_report:
  [audit_board_dashboard, audit_report -> audit_board_dashboard]
```

Lastly, audit board members see updates on the current state of their audit.

```
refresh: [audit_board_dashboard ->
  [audit_board_dashboard,
    string,
    set[[audited_contest, audit_reason]],
    county_status,
    list[[ballot, storage_container]],
    [ballot, ballot_id, ballot_style]
  ]
]
END audit_board_dashboard
```

### 2.17.3 Public Dashboard

```
public_dashboard: THEORY
BEGIN
    IMPORTING dashboard
```

We are currently proposing that the following set of data and reports be included on the public dashboard:

1. Target and Current Risk Limits, by Contest
2. Audit Board names by County
3. County Ballot Manifests, CVRs and Hashes (status & download links)
4. Seed for randomization
5. Ballot Order
6. List of Audit Rounds (number of ballots, status by County, download links)
7. Link to Final Audit Report

```
END public_dashboard
```

## 2.18 Authentication

Authentication is currently underspecified in our system design as we do not yet have information from CDOS on the nature and kind of their mandated two-factor authentication system. Consequently, we have only modeled the necessary concepts and features of any two-factor authentication system. Included in this model are the ideas of usernames, passwords, other authentication factors mentioned in CDOS documents (such as biometrics and physical tokens like smartcards and one-time authentication code books), etc.

Given that CDOS is handling two-factor authentication, it is unclear if they want any additional features such as a password reset.

```
authentication: THEORY
BEGIN
    IMPORTING ballots,
            cast_vote_records,
            cryptography,
```

```

        dashboard,
        dashboard,
        election,
        information_systems,
        roles,
        upload_system_message

credential: TYPE
username: TYPE FROM credential
password: TYPE FROM credential
complex_enough?: [password -> bool]
biometric: TYPE FROM credential
physical_token: TYPE FROM credential

authentication: TYPE FROM [person, set[credential]]
two_factor_authentication: TYPE =
    [[username, password], [physical_token + biometric]]

```

In order to obtain a new, valid credential, some authority must issue credentials to a specific person.

```

issue_credential: TYPE =
    [cdos, county_election, person -> [person, two_factor_authentication]]

```

What are the credentials that have been issued to this person?

```

credential?: [person -> two_factor_authentication]

```

Is this person authenticated?

```

authenticated: [person, two_factor_authentication -> bool]

```

What follows are several functions used to authenticate various roles to their respective dashboards.

Each dashboard has an abstract state machine that captures the dashboard's workflow and consequently identifies which features are visible to its users at various stages. Those abstract state machines and features are modeled in the dashboard modules below. We focus only on authentication here.

```

authenticate_county_administrator:
    [county_dashboard, county_administrator,
     two_factor_authentication -> bool]
authenticate_state_administrator:
    [department_of_state_dashboard, state_administrator,
     two_factor_authentication -> bool]

```

Introduce an authentication monad and a state monad for our ASMs. - kiniry

END authentication

## 3 Computer System

### 3.1 System Architecture

As mentioned early in this chapter, the RLA Tool is a client-server system. As usual for these kinds of systems, the server part of this architectural style is known as the back-end and the client part as the front-end.

```
system_architecture_component: DATATYPE
    WITH SUBTYPES back_end,
                  front_end
BEGIN
    IMPORTING database,
              information_systems,
              server

    back_end?(servers: set[server],
              networks: set[network],
              databases: set[database]): make_back_end: back_end
    front_end?(web_browser: web_browser,
              code: javascript_code): make_front_end: front_end
END system_architecture_component
```

The RLA system focuses on a single election and has a back-end and a front-end.

```
rla_tool: DATATYPE
BEGIN
    IMPORTING election,
              system_architecture_component

    rla_tool?(election: county_election,
              front_end: front_end,
              back_end: back_end): make_rla_tool
END rla_tool
```

The system architecture consists of:

1. several servers of different kinds deployed and configured in a redundant fashion as described elsewhere;

2. several databases whose tables and data are transactionally identical (this means that after each transaction completes, all databases are guaranteed to never witness to a client an inconsistent state relative to that transaction);
3. a JavaScript-based front-end whose code comes only from the system's web servers; and
4. HTTPS-based connections between the front-end and the back-end over which the web application transmits data, including new HTML pages, style sheets, data input by the web browser user, etc.

```

system_architecture: THEORY
BEGIN
  IMPORTING rla_tool,
            server,
            system_architecture_component

  rla: VAR rla_tool
  be: VAR back_end
  fe: VAR front_end
  db1, db2: VAR database_server

  make_rla_tool: [back_end, front_end -> rla_tool]

  javascript_code?: [front_end -> javascript_code]
  code_origins?: [javascript_code -> web_server]

```

The front-end, which is written in TypeScript and JavaScript, is provided to a web browser client directly from the web server.

```

browser_code_origins: AXIOM
(FORALL (rla):
  member(code_origins?(code(fe)), servers(be))
  WHERE fe = front_end(rla), be = back_end(rla))

transactionally_synchronized?: [set[database] -> bool]

```

The back-end consists of two or more servers, two or more networks, and two or more databases. At least two of each is necessary because the system must have redundancy to be fault tolerant and not have a single point of failure.

```

redundancy: AXIOM
(FORALL (be): 2 <= card(servers(be))
  AND 2 <= card(networks(be))
  AND 2 <= card(databases(be)))

```

Multiple databases need multiple synchronized database servers among the back-end's servers. Note that our model is simplified here insofar as we are presuming that all databases across synchronized database servers are synchronized. It is certainly possible to deploy synchronized databases in a fashion that does not fulfill this requirement, but for all databases relevant to this system, this requirement does hold.

```

database_redundancy: AXIOM
  (FORALL (be): EXISTS (db1, db2):
    member(db1, servers(be)) AND member(db2, servers(be))
  AND db1 /= db2
  AND transactionally_synchronized?(union(databases(db1),databases(db2))))
END system_architecture

```

### 3.2 System Assumptions

This system architecture includes a number of explicit assumptions derived from C.R.S.

```

system_assumptions: THEORY
BEGIN
  IMPORTING election,
    FAFSSL

  e: VAR county_election

```

According to the section entitled “CVR Export Verification”, counties conducting a comparison audit must verify that:

1. The number of individual CVRs in its CVR export equals the aggregate number of ballots reflected in the county's ballot manifest as of the ninth day after election day;

```

CVR_count_equals_ballot_manifest_count: AXIOM TBD

```

2. The number of individual CVRs in its CVR export equals the number of ballots tabulated as reflected in the summary results report for the RLA tabulation;

```

CVR_count_equals_summary_results_report_count: AXIOM TBD

```

3. The number of individual CVRs in its CVR export equals the number of in-person ballots issued plus the number of mail ballots in verified-accepted stage in SCORE, plus the number of provisional ballots and property owner ballots included in the RLA tabulation, if any; and

CVR\_count\_equals\_aggregate\_count\_over\_ballot\_kinds: AXIOM TBD

4. The vote totals for all choices in all ballot contests in the CVR export equals the vote totals in the summary results report for the RLA tabulation.

CVR\_vote\_totals\_equals\_summary\_results\_vote\_totals: AXIOM TBD

END system\_assumptions

### 3.3 System Logging

The system must log a variety of events. One reason to log information is to help understand how the system is operating, fix bugs post-facto, understand how users are using the system, etc. Another, reason to log in this context of this system is to provide an indelible record of administrator and auditor actions so that any audit can be “replayed” by any third party.

```
logging: THEORY
BEGIN
  log: TYPE = sequence[string]
  chain_of_custody_log: TYPE FROM log
  rla_tool_log: TYPE FROM log
END logging
```

### 3.4 Data Model

A single, mirrored relational database is used to store all persistent information for the RLA Tool. In order to define the data model of the system, we need to:

1. identify the *core concepts* that must be persistent in the system
2. identify the *relationships* that must be persistent in the system



3. derive from these pieces of information:
  - (a) the necessary *tables* that *collect concepts*,
  - (b) the *indexing concept* of each table, and
  - (c) definitions of the appropriate *joins* that realize relationships

```
data_model: THEORY
BEGIN
  IMPORTING FAFSSL,
            database,
            information_systems,
            roles,
            rla_tool,
            authentication,
            rlas,
            department_of_state_dashboard,
            cast_vote_records

  be: VAR back_end
  db: VAR database
```

The first set of data persistently stored relates to authentication. Some of this data will be stored in the two-factor authentication system provided by CDOS. All other data must be stored in the RLA Tool database. In general, each entry in the authentication table is simply an administrator and two-factor authentication credential pair.

```
authentication_table: TYPE =
  sequence[[administrator, two_factor_authentication]]
```

The Department of State dashboard permits State administrators to establish risk limits for each contest, specify which kinds of audits are used for which races and in which counties, etc. Most of the dynamic data available to State administrators is provided via County administrators using the County dashboard.

```
state_status_table: TYPE =
  sequence[# county: county,
            general_status: county_status,
            audit_status: audit_stage,
            date: string #]]
```

There is also some background information that relates to the RLA overall, much of which is relevant to the Department of State dashboard. In particular, geographic (such as the identity of the State and its Counties) and political (registered political parties) information must be stored, as must information about each election under audit.

Notice that these tables are simply a flattening of the election datatype. This is a standard pattern in mapping HOL specifications to relational data models.

```

geography_table: TYPE =
  sequence[[nation, state, county]]
political_party_table: TYPE =
  sequence[[political_party]]
county_contest_table: TYPE =
  sequence[[county, contest]]

public_meeting_to_determine_seed_table: TYPE =
  [# location: string, date: string #]
rla_information_table: TYPE =
  sequence[[RLA, audit_reason, contest, risk_limit]]

```

We formalize the list of ballots under audit and organize them both by contest and by county. The former is defined when the RLA algorithm randomly chooses ballots. The latter is populated by filtering the former by county, and it is what is shared with counties via the Audit Board Dashboard.

```

audited_ballot_by_contest_list_table: TYPE =
  sequence[[contest, sequence[ballot_manifest_info]]]
audited_ballot_by_county_list_table: TYPE =
  sequence[[county, sequence[ballot_manifest_info]]]

```

County administrator actions create and update several kinds of data. For example, they define who is on their audit boards, they generate digests of files critical to the audit (principally, ballot manifests and CVRs), they upload those files, etc.

For general county information, we currently model the data as property/value pairs in a record. For example, we imagine that each county will want to display who their County Clerk is, etc.

```

county_general_information_table: TYPE =
  sequence[[# property: string, value: string #]]
audit_board_table: TYPE =
  sequence[[county, audit_board_member]]
ballot_manifest_digest_table: TYPE =
  sequence[[# county: county,
            ballot_manifest_file_name: string,
            digest: string,
            data: ballot_manifest_file #]]
cvr_digest_table: TYPE =
  sequence[[# county: county,
            cvr_file_name: string,
            digest: string,
            data: cvr_file #]]

```

```

ballot_manifest_table: TYPE =
  sequence[[county, ballot_manifest_info]]

```

Cast vote records uploaded by counties contain CVRs of three kinds: CVRs for local (county) contests, CVRs for contests that span counties, and CVRs for contests that span the whole state. These latter two cases are identical from a modeling point of view, as the whole state is simply a spanning contest over all counties.

We can store and organize this information in several different ways. First, we can simply store the individual CVRs uploaded by a given county in a table.

```

cvr_table: TYPE =
  sequence[[county, cvr]]

```

We need a query to determine when the Department of State Dashboard can show the full set of contests in the state and permit the Secretary of State to choose which contests are under audit. This query is answered by examining every county's status (provided by the `refresh` endpoint of the DOS dashboard) to see that they all have value `cvrs_uploaded_successfully`.

```

cvr_uploads_complete?: [cvr_table -> boolean]
ballot_manifest_uploads_complete?: [ballot_manifest_table -> boolean]

```

But we also need to store all CVRs for each contest, aggregating all CVRs across the upload from all counties in which the contest was on the ballot.

```

contest_cvr_table: TYPE =
  sequence[[contest, cvr]]

```

Tabulation results as calculated from the CVRs.

```

tabulation_result_table: TYPE =
  sequence[[contest, cvr_tabulation_outcome]]

```

While running the RLA, the Audit Board uploads information about each ballot they audit. Also, if any investigations are made and remedied during the audit, that information must be stored as well. Finally, after an audit is complete an audit report is uploaded and stored.

```

ballots_under_audit_table: TYPE =
  sequence[[ballot,
    list[ballot_mark],
    cvr,
    margin_overstatement,
    margin_understatement]]
auditing_investigations_table: TYPE =
  sequence[[county, audit_investigation_report]]
audit_reports_table: TYPE =
  sequence[[county, audit_report]]
rla_summary_results_table: TYPE =
  sequence[[contest, summary_results_report]]

```

All of the information on the public dashboard is derived from the above data.

```
END data_model
```

Add election as first column to every table? Or have a different DB per election? -kiniry

## 4 Abstract State Machine

Each dashboard has its own abstract state machine (ASM). The RLA Tool's ASM is, at least in theory, the composition of the composed of the three dashboard ASMs. We specify ASM's transitions using the finite automatas' standard tabular format.

```
asm: THEORY
BEGIN
  IMPORTING FAFSSL,
    department_of_state_dashboard
```

In order to formalize ASMs in general, we need to define a few relevant types. Note that we are modeling *deterministic* ASMs, thus transitions can depend upon not only the current state and some event, but also an arbitrary boolean expression that typically involves system state. Since our ASMs are deterministic, transitions map to only a single target state, rather than a set of states as one would see in a non-deterministic ASM.

```
event: TYPE
asm_state: TYPE
asm_event: TYPE FROM event
asm_transition: TYPE =
  [# start_state: asm_state, event: asm_event, end_state: asm_state #]
asm_transition_function: TYPE = set[asm_transition]
```

Consequently, the core features of the ASM implementation in the business logic of the RLA Tool are:

- What is the current state of the system?

```
current_state: TYPE = [dashboard -> asm_state]
```

- What events are enabled?

```
enabled_events: TYPE = [dashboard -> set[asm_event]]
```

- Transition to the next state based upon this event.

```
step_event: TYPE = [[dashboard, asm_event] -> [dashboard, asm_state]]
```

- Transition to the next state based upon this transition.

```
step_transition: TYPE = [[dashboard, asm_transition] -> [dashboard, asm_state]]
```

- Are we in a initial state?

```
initial?: TYPE = [dashboard -> boolean]
```

- Are we in a final state?

```
final?: TYPE = [dashboard -> boolean]
```

```
asm: TYPE =
  [# identity: string,
   states: set[asm_state],
   events: set[asm_event],
   transitions: asm_transition_function,
   initial_state: asm_state,
   final_states: set[asm_state] #]
```

Explain ASM definition.  
-kiniry

We will first specify the ASM for the Department of State dashboard. Its states describe the sequence of steps that state administrators must take to start, observe, and complete an RLA. The state `initial` is the initial state of the ASM; the state `audit_results_published` is the final state of the ASM.

```
department_of_state_dashboard_state: TYPE = % SUBTYPE asm_state
{ di_s,    % dos_initial_state
  daa_s,    % dos_administrator_authenticated
  rls_s,    % risk_limits_set
  ctai_s,   % contests_to_audit_identified
  rsp_s,    % random_seed_published
  bod_s,    % ballot_order_defined
  art_s,    % audit_ready_to_start
  ao_s,     % audit_ongoing
  ac_s,     % audit_complete
  arp_s,    % audit_results_published
}
```

Various events, most of which are triggered by state administrators using the RLA Tool, cause state changes. These events are named according to their corresponding functional definitions in earlier theories. E.g., the `authenticate_state_administrator_event` relates to the `authenticate_state_administrator` function and its realization via an endpoint.

```

department_of_state_dashboard_event: TYPE = % SUBTYPE asm_event
{
  asa_e,      % 1. authenticate_state_administrator_event
  erlfca_e,   % 2. establish_risk_limit_for_comparison_audits_event
  scfca_e,    % 3. select_contests_for_comparison_audit_event
  pdta_e,     % 13. publish_data_to_audit @todo kiniry
  ps_e,       % 4. publish_seed_event
  pbta_e,     % 5. publish_ballots_to_audit_event
  rm_e,       % 7. 8. any report_markings call by any county
  ifhcc_e,    % 9. 10. indicate_full_hand_count_contest_event
  cac_e,      % 11. county_audit_complete_event
  par_e,      % 12. publish_audit_report
  r_e,        % 13. refresh_event
  s_e,        % 6. skip_event
  n_e,        % no_event
}

dos_states: set[asm_state]
dos_events: set[asm_event]
dos_transitions: asm_transition_function
dos_initial_state: asm_state
dos_final_states: set[asm_state]

dos_state: VAR asm_state
dos_event: VAR asm_event
dos_asm_function: VAR asm_transition_function

```

As there is only a single Department of State, then our system has only a single instance of this ASM.

```

dos_dashboard_asm: asm =
  (# identity := "DOS",
    states := dos_states,
    events := dos_events,
    transitions := dos_transitions,
    initial_state := dos_initial_state,
    final_states := dos_final_states #)

county_dashboard_state: TYPE = % SUBTYPE asm_state
{
  ci_s,      % county_initial_state
  caa_s,     % county_administrator_authenticated
  abe_s,     % audit_board_established_state

  bm_tl_s,   % upload_ballot_manifest: too_late
  bm_dti_s,  % upload_ballot_manifest: data_transmission_interrupted

```

Discuss up-  
load\_system\_message. -  
kiniry

```

        bm_us_s,    % upload_ballot_manifest: upload_sucessful
        bm_ch_s,    % upload_ballot_manifest: checking_hash
        bm_hm_s,    % upload_ballot_manifest: hash_mismatch
        bm_hv_s,    % upload_ballot_manifest: hash_verified
        bm_pd_s,    % upload_ballot_manifest: parsing_data
        bm_ftw_s,   % upload_ballot_manifest: file_type_wrong
        bm_dp_s,    % upload_ballot_manifest: data_parsed
        cvr_tl_s,   % upload_cvrs: too_late
        cvr_dti_s,  % upload_cvrs: data_transmission_interrupted
        cvr_us_s,   % upload_cvrs: upload_sucessful
        cvr_ch_s,   % upload_cvrs: checking_hash
        cvr_hm_s,   % upload_cvrs: hash_mismatch
        cvr_hv_s,   % upload_cvrs: hash_verified
        cvr_pd_s,   % upload_cvrs: parsing_data
        cvr_ftw_s,  % upload_cvrs: file_type_wrong
        cvr_dp_s,   % upload_cvrs: data_parsed
        cau_s,      % county_audit_underway
        cac_s,      % county_audit_complete
    }

county_dashboard_event: TYPE = % SUBTYPE asm_event
{
    aca_e,    % 1. authenticate_county_administrator_event
    eab_e,    % 2. establish_audit_board_event
    cuvbm_e,  % 3. 4. 5. 7. county_upload_ballot_manifest_event
    uvc_e,    % 16. 17. 19. 21. upload_cvrs_event
    sa_e,     % 29. start_audit_event
    r_e,      % 30. refresh_event
    s_e,      % 1. 6. 8. 9. 10. 11. 12. 13. 14. 15. 18. 20. skip_event
    n_e,      % no_event
}

county_dashboard_states: set[asm_state]
county_dashboard_events: set[asm_event]
county_dashboard_transitions: asm_transition_function
county_dashboard_initial_state: asm_state
county_dashboard_final_states: set[asm_state]

c_state: VAR county_dashboard_state
c_event: VAR county_dashboard_event
c_asm_function: VAR asm_transition_function

```

Each state has several counties, and each county has an ASM. Thus there is a relation between counties and county dashboard ASMs.

```
county_dashboard_asm: asm =
```

```

    (# identity := "county ID",
      states := county_dashboard_states,
      events := county_dashboard_events,
      transitions := county_dashboard_transitions,
      initial_state := county_dashboard_initial_state,
      final_states := county_dashboard_final_states #)
county_to_county_dashboard_asm_relation: TYPE =
  pred[[county, asm]]

```

Likewise, there is a similar relation between counties and audit board dashboard ASMs.

```

audit_board_dashboard_state: TYPE = % SUBTYPE asm_state
{ ai_s,    % audit_initial_state
  aip_s,   % audit_in_progress_state
  iars_s,  % intermediate_audit_report_submitted_state
  ars_s    % audit_report_submitted_state
}

audit_board_dashboard_event: TYPE = % SUBTYPE asm_event
{ rm_e,    % 2. report_markings_event
  rbnf_e,  % 3. report_ballot_not_found_event
  sair_e,  % 4. submit_audit_investigation_report_event
  siar_e,  % 5. submit_intermediate_audit_report_event
  sar_e,   % 6. submit_audit_report_event
  r_e,     % 8. audit_refresh_event
  as_e,    % 1. 7. audit_skip
  n_e      % no_event
}

audit_board_dashboard_states: set[asm_state]
audit_board_dashboard_events: set[asm_event]
audit_board_dashboard_transitions: asm_transition_function
audit_board_dashboard_initial_state: asm_state
audit_board_dashboard_final_states: set[asm_state]

ab_state: VAR audit_board_dashboard_state
ab_event: VAR audit_board_dashboard_event
ab_asm_function: VAR asm_transition_function

audit_board_dashboard_asm: asm =
  (# identity := "county ID",
    states := audit_board_dashboard_states,
    events := audit_board_dashboard_events,
    transitions := audit_board_dashboard_transitions,
    initial_state := audit_board_dashboard_initial_state,

```



```

        final_states := audit_board_dashboard_final_states #)
    county_to_audit_board_dashboard_asm_relation: TYPE =
        pred[[county, asm]]

END asm

```

## 5 Endpoints

```

endpoints: THEORY
BEGIN
    IMPORTING dashboard,
            asm,
    department_of_state_dashboard,
    county_dashboard,
    audit_board_dashboard

```

In order to relate the ASM to the core implementation constructs in a RESTful client-server architecture, we must relate UI events to ASM events and ASM events to server endpoints. Communication to and from those endpoints happens via JSON streams over HTTPS. Thus, for any given endpoint, there is an expected inbound JSON type and an expected outbound JSON type per HTTP response code.

```

    ui_event: TYPE = {
        LOGIN,
        FETCH_INITIAL_STATE_SEND,
        FETCH_INITIAL_STATE_RECEIVE,
        SELECT_NEXT_BALLOT,
        UPDATE_BOARD_MEMBER,
        UPDATE_BALLOT_MARKS,
        UNDEFINED
    }

```

We specify the content of data sent to/from the server either with a specification of a URL parameter-based communication or using a JSON object. We do not wish to precisely specify the JSON, as encoding is non-unique due to the existence of semantically equivalent encodings (as with any serialization technique) and across libraries. Instead, we specify data-in-transit implicitly via the function signatures in the theories above that focus on the endpoint callback (the various `dashboard` theories) and we name the semantically relevant fields using a list of strings using the `json_spec` type.

```

    json_spec: TYPE = list[string]

```

The `nothing` constant specifies that no content is transmitted for a given communication with the server. This commonly happens, e.g., when performing a GET on an endpoint.

```
nothing: json_spec = null
```

The `server_response` concept is the standard/normal response that the server gives to all public inbound events to permit the UI (the view of the MVC architecture) to know what state the server is in, which UI elements should be enabled (and thus what transitions are legal). We call this kind of behavior in the server “normal behavior”. In all normal behavior cases the server will answer with an HTTP response code of OK\_200.

```
server_response: json_spec =  
  (: "server_state", "ui_events" :)
```

The `upload_response` concept is used to encode the variety of response status the server can give to an upload. Note that the `upload_response` type is used as a response for both ballot manifest and CVR uploads. These responses can happen synchronously (our plan for *stage-1* deliverable) or asynchronously (our plan for a later deliverable). Asynchronous updates happen via a call to the `refresh...` endpoints.

```
upload_response: json_spec =
```

Legal values for `status` are: `too_late`, `data_transmission_interrupted`, `upload_sucessful`, `checking_hash`, `hash_wrong`, `hash_verified`, `parsing_data`, `file_type_wrong`, and `data_parsed`.

```
(: "ballot_manifest-status", "cvr-status" :)
```

There are several possible cases where a client can make an inappropriate call to the server. Our client will be guaranteed to never make such a call, but we cannot presume that only our client will be communicating with the server, as the server will be publicly accessible on the internet. We call these behaviors of the server “exceptional behavior”. All responses to exceptional behavior come in the form of HTTP client error (4xx) or server error (5xx) response codes.

```
http_response_code: TYPE = {  
  OK_200,  
  BAD_REQUEST_400,  
  UNAUTHORIZED_401,  
  FORBIDDEN_403,  
  NOT_FOUND_404,  
  UNSUPPORTED_MEDIA_TYPE_415,  
  UNPROCESSABLE_ENTITY_422,  
  INTERNAL_SERVER_ERROR_500  
}
```

The cases that we must cover are:

1. The server is simply not up and running properly. For example, if the database is not in a legal state. In these cases the server will answer with an HTTP response code of `INTERNAL_SERVER_ERROR_500`.
2. In instances where the server is asked to query for a particular piece of data from the database and that does not exist, the server will respond with a `NOT_FOUND_404`. This can happen, e.g., when a client uses an improper encoding for a piece of data, such as a county ID that does not exist.
3. A piece of data uploaded by a client (at this point in time, either a ballot manifest, a CVR file, or an Audit CVR) is malformed and cannot be interpreted. This situation can happen, for example, if the Election Management System used to export the file has a bug, if the file is improperly edited by someone, or if there is a bug in our parser. In this situation, the server will respond with an `UNPROCESSABLE_ENTITY_422` response.
4. *(This item is a potential future feature.)* Alternatively, if the file is of the wrong type (i.e., a PDF was uploaded when the server expected a CSV file), or if the uploaded file is too large, then the server will respond with a `UNSUPPORTED_MEDIA_TYPE_415` message.
5. If the client is unauthenticated, or if authentication fails, then the server will respond with a `UNAUTHORIZED_401` response.
6. If some element of a request is incorrect then the server responds with a `BAD_REQUEST_400` message. In particular, if the digest uploaded with a ballot manifest or a CVR file is incorrect, then the server will store the data and respond with this message.
7. Finally, if the endpoint called is improper given the state that the server is in—i.e., if calling the given endpoint implies an illegal state transition—then the server will respond with a `FORBIDDEN_403` message. This happens only when an authenticated client tried to violate the state machine of the server, and will either indicate a bug in our client code or malicious or experimental behavior on the part of some non-official party.

```
http_inbound_spec: TYPE = list[string]
empty_request: http_inbound_spec = null
```

```
authenticate_params: http_inbound_spec = % reused for County Dashboard
  (: "username", "password", "second_factor" :)
risk_limit_params: http_inbound_spec =
  (: "risk_limit" :)
select_contests_params: http_inbound_spec =
  (: "contests" :)
random_seed_params: http_inbound_spec =
```

Transport specs for Department of State Dashboard. -kiniry

```

        (: "seed" :)
ballots_to_audit_params: http_inbound_spec =
    (: "county" :)
full_hand_count_params: http_inbound_spec =
    (: "contest" :)
published_audit_data: json_spec =
    (: "TBD" :)
ballots_to_audit: json_spec =
    (: "ballots" :)
final_state_audit_report: json_spec =
    (: "audit_data", "county_audit_reports" :)
refresh_department_of_state_dashboard: json_spec =
    (: "audit_stage", "risk_limit", "audited_contests",
       "county_statuses", "seed", "full_hand_count_contests" :)

```

Transport specs for  
County Dashboard. -  
kiniry

```

audit_board: http_inbound_spec =
    (: "firstname", "lastname", "party" :)
ballot_manifest_parts: http_inbound_spec =
    (: "bmi_file", "county", "hash" :)
cvr_parts: http_inbound_spec =
    (: "cvr_file", "county", "hash" :)
refresh_county_dashboard: json_spec =
    (: "general_information", "audit_board_members",
       "ballot_manifest_digest", "CVR_digest",
       "contests_on_ballot", "contests_under_audit_with_reasons",
       "date_and_time_of_audit", "estimated_number_of_ballots_to_audit",
       "number_of_ballots_audited", "number_of_discrepancies",
       "number_of_disagreements" :)

```

Transport specs for Audit  
Board Dashboard. -kiniry

```

acvr_parts: http_inbound_spec =
    (: "audit_cvr" :)
ballot_not_found_params: http_inbound_spec =
    (: "ballot_id" :)
audit_investigation_report: json_spec =
    (: "report" :)
audit_report: json_spec =
    (: "reason" :)
refresh_audit_board_dashboard: json_spec =
    (: "general_information", "contests_under_audit",
       "county_status", "ballots_under_audit_with_locations",
       "current_ballot_under_audit" :)

```

Discuss HTTP endpoint  
types and endpoints. -  
kiniry

```

endpoint_type: TYPE = { GET, POST, PUT }
server_endpoint: TYPE =
  [# endpoint_type: endpoint_type,
    callback: [dashboard -> dashboard],
    uri: string,
    implementation_class: string,
    in_stream: http_inbound_spec,
    out_stream: set[[http_response_code, json_spec]],
    dos_event: department_of_state_dashboard_event,
    county_event: county_dashboard_event,
    audit_event: audit_board_dashboard_event
  #]
server_endpoints: TYPE = set[server_endpoint]
rla_tool_endpoints: VAR server_endpoints
undefined_callback: [dashboard -> dashboard]

```

Each RLA Tool system endpoint is specified below.

```

root_endpoint: server_endpoint = % DONE
  (# endpoint_type := GET,
    callback := undefined_callback,
    uri := "/",
    implementation_class := "Root",
    in_stream := empty_request,
    out_stream := singleton((OK_200, nothing)),
    dos_event := s_e,
    county_event := n_e,
    audit_event := n_e
  #)

```

## 5.1 Department of State Dashboard Endpoints

```

auth_state_admin: server_endpoint = % DONE
  (# endpoint_type := POST,
    callback := undefined_callback,
    uri := "/auth-state-admin",
    implementation_class := "AuthenticateStateAdministrator",
    in_stream := authenticate_params,
    out_stream := add((UNAUTHORIZED_401, nothing),
      singleton((OK_200, nothing))),
    dos_event := asa_e,
    county_event := n_e,
    audit_event := n_e
  #)
risk_limit: server_endpoint = % DONE
  (# endpoint_type := POST,
    callback := undefined_callback,

```

```

uri := "/risk-limit-comp-audits",
implementation_class := "EstablishRiskLimitForComparisonAudits",
in_stream := risk_limit_params,
out_stream := add((BAD_REQUEST_400, nothing),
                  add((UNAUTHORIZED_401, nothing),
                      singleton((OK_200, nothing)))),
dos_event := erlfca_e,
county_event := n_e,
audit_event := n_e
#)
select_contests: server_endpoint = % DONE
(# endpoint_type := POST,
 callback := undefined_callback,
 uri := "/select-contests",
 implementation_class := "SelectContestsForAudit",
 in_stream := select_contests_params,
 out_stream := add((BAD_REQUEST_400, nothing),
                  add((UNAUTHORIZED_401, nothing),
                      singleton((OK_200, nothing)))),
 dos_event := scfca_e,
 county_event := n_e,
 audit_event := n_e
#)
publish_data_to_audit: server_endpoint = % UNDERWAY
(# endpoint_type := POST,
 callback := undefined_callback,
 uri := "/publish-data-to-audit",
 implementation_class := "PublishDataToAudit",
 in_stream := empty_request,
 out_stream := add((UNAUTHORIZED_401, nothing),
                  singleton((OK_200, published_audit_data))),
 dos_event := pdta_e,
 county_event := n_e,
 audit_event := n_e
#)
publish_seed: server_endpoint = % DONE
(# endpoint_type := POST,
 callback := undefined_callback,
 uri := "/random-seed",
 implementation_class := "UploadRandomSeed",
 in_stream := random_seed_params,
 out_stream := add((BAD_REQUEST_400, nothing),
                  add((UNAUTHORIZED_401, nothing),
                      singleton((OK_200, nothing)))),
 dos_event := ps_e,
 county_event := n_e,

```

```

        audit_event := n_e
    #)
publish_ballots: server_endpoint = % UNDERWAY
    (# endpoint_type := GET,
        callback := undefined_callback,
        uri := "/ballots-to-audit",
        implementation_class := "PublishBallotsToAudit",
        in_stream := ballots_to_audit_params,
        out_stream := add((UNAUTHORIZED_401, nothing),
            singleton((OK_200, ballots_to_audit))),
        dos_event := pbta_e,
        county_event := n_e,
        audit_event := n_e
    #)
full_hand_count: server_endpoint = % UNDERWAY
    (# endpoint_type := POST,
        callback := undefined_callback,
        uri := "/hand-count",
        implementation_class := "IndicateHandCount",
        in_stream := full_hand_count_params,
        out_stream := add((BAD_REQUEST_400, nothing),
            add((UNAUTHORIZED_401, nothing),
                singleton((OK_200, nothing)))),
        dos_event := ifhcc_e,
        county_event := n_e,
        audit_event := n_e
    #)
publish_report: server_endpoint = % UNDERWAY
    (# endpoint_type := GET,
        callback := undefined_callback,
        uri := "/publish-report",
        implementation_class := "PublishAuditReport",
        in_stream := empty_request,
        out_stream := add((UNAUTHORIZED_401, nothing),
            singleton((OK_200, final_state_audit_report))),
        dos_event := par_e,
        county_event := n_e,
        audit_event := n_e
    #)
refresh_department_of_state_dashboard: server_endpoint = % DONE
    (# endpoint_type := GET,
        callback := undefined_callback,
        uri := "/dos-dashboard",
        implementation_class := "DoSDashboardRefresh",
        in_stream := empty_request,
        out_stream := add((UNAUTHORIZED_401, nothing),

```

```

                                singleton((OK_200, nothing))),
dos_event := r_e,
county_event := n_e,
audit_event := n_e
#)

```

## 5.2 County Dashboard Endpoints

```

auth_county_admin: server_endpoint = % DONE
(# endpoint_type := POST,
  callback := undefined_callback,
  uri := "/auth-county-admin",
  implementation_class := "AuthenticateCountyAdministrator",
  in_stream := authenticate_params,
  out_stream := add((UNAUTHORIZED_401, nothing),
                    singleton((OK_200, nothing))),
  dos_event := n_e,
  county_event := aca_e,
  audit_event := n_e
#)

establish_audit_board: server_endpoint = % UNDERWAY
(# endpoint_type := POST,
  callback := undefined_callback,
  uri := "/audit-board", % placeholder
  implementation_class := "EstablishAuditBoard", % placeholder
  in_stream := audit_board,
  out_stream := add((UNAUTHORIZED_401, nothing),
                    singleton((OK_200, nothing))),
  dos_event := n_e,
  county_event := eab_e,
  audit_event := n_e
#)

ballot_manifest_upload: server_endpoint = % DONE
(# endpoint_type := POST,
  callback := undefined_callback,
  uri := "/upload-ballot-manifest",
  implementation_class := "BallotManifestUpload",
  in_stream := empty_request,
  out_stream := add((BAD_REQUEST_400, upload_response),
                    add((UNAUTHORIZED_401, nothing),
                        singleton((OK_200, upload_response)))),
  dos_event := n_e,
  county_event := cuvbm_e,
  audit_event := n_e
#)

cvr_export_upload: server_endpoint = % DONE

```



```

(# endpoint_type := POST,
  callback := undefined_callback,
  uri := "/upload-cvr-export",
  implementation_class := "CVRExportUpload",
  in_stream := cvr_parts,
  out_stream := add((UNPROCESSABLE_ENTITY_422, upload_response),
    add((UNAUTHORIZED_401, nothing),
      singleton((OK_200, upload_response)))),
  dos_event := n_e,
  county_event := uvc_e,
  audit_event := n_e
#)
refresh_county_dashboard: server_endpoint = % DONE
(# endpoint_type := GET,
  callback := undefined_callback,
  uri := "/county-dashboard",
  implementation_class := "CountyDashboardRefresh",
  in_stream := empty_request,
  out_stream := add((UNAUTHORIZED_401, nothing),
    singleton((OK_200, refresh_county_dashboard))),
  dos_event := n_e,
  county_event := r_e,
  audit_event := n_e
#)

```

### 5.3 Audit Board Dashboard Endpoints

```

acvr_upload: server_endpoint = % DONE
(# endpoint_type := POST,
  callback := undefined_callback,
  uri := "/upload-audit-cvr",
  implementation_class := "ACVRUpload",
  in_stream := acvr_parts,
  out_stream := add((UNPROCESSABLE_ENTITY_422, upload_response),
    add((UNAUTHORIZED_401, nothing),
      singleton((OK_200, nothing)))),
  dos_event := n_e,
  county_event := n_e,
  audit_event := rm_e
#)
ballot_not_found: server_endpoint = % UNDERWAY
(# endpoint_type := POST,
  callback := undefined_callback,
  uri := "/ballot-not-found", % placeholder
  implementation_class := "BallotNotFound", % placeholder
  in_stream := ballot_not_found_params,

```

```

        out_stream := add((UNAUTHORIZED_401, nothing),
                           singleton((OK_200, nothing))),
        dos_event := n_e,
        county_event := n_e,
        audit_event := rbnf_e
    #)
audit_investigation_report: server_endpoint = % UNDERWAY
    (# endpoint_type := POST,
       callback := undefined_callback,
       uri := "/audit-investigation-report", % placeholder
       implementation_class := "AuditInvestigationReport", % placeholder
       in_stream := audit_investigation_report,
       out_stream := add((UNAUTHORIZED_401, nothing),
                          singleton((OK_200, nothing))),
       dos_event := n_e,
       county_event := n_e,
       audit_event := sair_e
    #)
intermediate_audit_report: server_endpoint = % UNDERWAY
    (# endpoint_type := POST,
       callback := undefined_callback, % signoff_intermediate_audit_report
       uri := "/intermediate-audit-report", % placeholder
       implementation_class := "IntermediateAuditReport", % placeholder
       in_stream := audit_report,
       out_stream := add((UNAUTHORIZED_401, nothing),
                          singleton((OK_200, nothing))),
       dos_event := n_e,
       county_event := n_e,
       audit_event := siar_e
    #)
audit_report: server_endpoint = % UNDERWAY
    (# endpoint_type := POST,
       callback := undefined_callback,
       uri := "/audit-report", % placeholder
       implementation_class := "AuditReport", % placeholder
       in_stream := audit_report,
       out_stream := add((UNAUTHORIZED_401, nothing),
                          singleton((OK_200, nothing))),
       dos_event := n_e,
       county_event := n_e,
       audit_event := sar_e
    #)
refresh_audit_board_dashboard: server_endpoint = % NOT STARTED
    (# endpoint_type := GET,
       callback := undefined_callback,
       uri := "/audit-board-dashboard", % placeholder

```

```

implementation_class := "AuditBoardDashboard", % placeholder
in_stream := empty_request,
out_stream := add((UNAUTHORIZED_401, nothing),
                  singleton((OK_200, refresh_audit_board_dashboard))),
dos_event := n_e,
county_event := n_e,
audit_event := r_e
#)

ui_to_asm_event_relation: pred[[ui_event, asm_event]]
asm_event_to_endpoint_relation: pred[[asm_event, server_endpoint]]
END endpoints

```

Explain these relations  
and their use. -kiniry

## 6 Protocol

As discussed in Section 3.1, the RLA Tool is a client-server system. In this section we describe the protocol used by the client and server to communicate with each other.

```

protocol: THEORY
BEGIN
  IMPORTING system_architecture,
            system_architecture_component

```

The client-side of the system is exclusively a web browser running an HTML/Javascript-based web application. Recall that in Section 3.1 we mandated that all client artifacts (static content, HTML, and Javascript) are provided directly from the (web) server.

```

client: VAR front_end
browser: VAR web_browser

```

The server-side of the system is a web server which communicates over HTTPS. It serves both static content, such as images and HTML fragments, and executable content, such as Javascript code. As such, this is a standard REST-based, multi-tier architecture.

To provide assurance about system availability, we deploy multiple instances of web and application servers providing content and databases providing persistent data storage. All of these artifacts are connected over multiple networks in CDOS's hosting service. We underspecify the details of the deployment network configuration here at the request of CDOS.

```

server: VAR back_end
servers: VAR set[server]
web_server: VAR web_server
application_server: VAR application_server
networks: VAR set[network]
databases: VAR set[database]

```

Multiple databases run in a failover configuration. Web application servers communicate with databases over a standard secure remote SQL connection. Web servers serve static content and delegate dynamic content services to the application servers.

All communication between the client web browsers and the server(s) is initiated by the client, as with normal REST-based architectures. As such, functions defined in the dashboards (Section 2.16) of the system are the endpoint of all communication in the system. The RLA Tool protocol is consequently the union of all of those endpoints.

## 6.1 Department of State Dashboard Protocol

As discussed in Section 2.17, the Department of State Dashboard has eight (8) features: one for authentication, six for providing information about a given election and RLA, and one for getting updates on the current RLA. Each feature is encoded in a function in the `department_of_state_dashboard` module.

1. `authenticate_state_administrator`: authentication of state administrators (the mandatory first step in the protocol),
2. `select_contests_for_comparison_audit`: defining which contests will be under audit using comparison audits,
3. `publish_seed`: the publication of the random seed for RLAs in the election under audit,
4. `establish_risk_limit_for_comparison_audits`: establishments of risk limits for the election under audit,
5. `public_ballots_to_audit`: publishing the list of ballots to be audited for each audited contest,
6. `indicate_full_hand_count_contest`: indicate that a contest must be a full hand count contest, and
7. `refresh`: update the dashboard on the current state of the election and its RLAs, including the status of all counties.

The first mandatory step in the protocol is authentication. All other features can be used in any order, though typically, as mandated by the C.R.S., risk limits are defined earlier than the selection of audited contests. Refreshes are

automatic and periodic, the periodicity of which will be determined via UX and load testing.

Each of these functions is a public feature of the system encoding what we call a “public inbound event” to the RLA Tool’s servers. By examining one such event in detail, we will elucidate how to read and understand its specification and derive from it our network data model.

## 6.2 Drill-down Example of a Public Inbound Event

We will examine the function `select_contests_for_comparison_audit` in detail in this section.

The first thing to note about this function is its *type signature*. The type signature tells us what types of values it depends upon (on the lefthand side of the arrow (“->”) and what types of values it produces (on the righthand side). The type signature of this function is

```
[department_of_state_dashboard, set[[contest, audit_reason]] ->
[department_of_state_dashboard, set[audited_contest]]]
```

As a consequence of the first type parameter mentioned in the signature, this function operates on the Department of State Dashboard, thus is visible to authenticated State administrators.

The second parameter states that the function expects to send a *set of pairs of contests and audit reasons* to the server. By virtue of the fact these two generic types (sets and pairs) are mentioned says that we must have a means by which to encode those generic notions over the network. Because we are using JSON as our general textual wire encoding format, we have a straightforward means by which to encode such. The mention of the domain-specific types `contest` and `audit_reason` mean that we must be able to encode values of these types as well on the wire.

To encode a value of a given type on the wire (and, commensurately, store a value in a database), one must be able to either or both *encode its constituent values on the wire* or *refer to a uniquely identified and encoded value that has been previously defined*. The type `contest` is defined as a record containing strings and choices, the latter of which ends up being encoded as a set of options which are strings. So, in the end, a contest value’s contents are nothing more than a structured assembly of strings—a straightforward thing to encode.

The type `audit_reason` is simply an *enumeration*, thus can be encoded in any number of ways. Commonly such structures are encoded by either string representations of the enumeration’s values (such as “`state_wide_contest`” and “`county_wide_contest`”, for example) or by using an encoding to natural numbers (e.g., 0 means `state_wide_contest`, etc.). The means of encoding enumerations does not matter, so long as it is a full bijective encoding (encoding covers all values and preserves all information and is reversible). We will discuss the precise means by which we define data types and derive their JSON encodings in ?? below.

The righthand side of the function's type signature says that the dashboard will be updated after the communication with the server completes (successfully or not). If we have sufficient information about potential exceptional failure cases for the function (e.g., we decide to deny the ability for state administrators to call this function twice to update the contests to audit more than once, we may need a response code to indicate such), then the codomain of the function must indicate such. See, for example, `county_upload_verified_ballot_manifest`.

Add citation or explanation of encoding of sets and pairs. -kiniry

### 6.3 County Dashboard Protocol

The County Dashboard has four (4) functions defined on it, all of which are public external events: `establish_audit_board`, `county_upload_verified_ballot_manifest`, `upload_verified_cvrs`, and `upload_tabulation_results`, the meaning and use of each is clear from its definition.

Add response code for all external public events. -kiniry

Explain how we are doing the definition of data types and the derivation of JSON encodings. -dmz

### 6.4 Audit Board Dashboard Protocol

The Audit Board Dashboard has seven (7) functions defined on it, all of which are public external events:

1. `ballots_to_audit_to_storage_container_list`, which is used to help audit board members find the ballots that they must audit,
2. `next_ballot_for_audit`, which provides the ballot manifest information necessary to inform the audit board members of the ballot style of the pulled ballot as well as the CVR number of the ballot under audit,
3. `report_markings`, which permits the audit board members to report the markings that they see on the current paper ballot under audit,
4. `report_ballot_not_found`, which permits the audit board members to report that they cannot find the ballot that they are supposed to audit,
5. `submit_audit_investigation_report`, which permits the audit board members to submit a report of an investigation into a given discrepancy in the RLA process,
6. `submit_audit_report`, which permits the audit board members to submit their final report on the RLA, and
7. `submit_intermediate_audit_report`, which permits the audit board members to submit an intermediate report on the RLA so that they can, e.g., take a break and return to the audit process later.

END protocol

## 7 Domain Model

The overall domain model and specification of this system is the sum total of all concepts introduced in this chapter.

```
specification_theories: THEORY
BEGIN
    IMPORTING
        FAFSSL
END specification_theories

background_terminology: THEORY
BEGIN
    IMPORTING
        database,
        information_systems,
        randomness,
        cryptography
END background_terminology

terminology: THEORY
BEGIN
    IMPORTING
        audits,
        authentication,
        ballot_interpretation,
        ballot_manifests,
        ballots,
        ballots_collections,
        canvass_boards,
        cast_vote_records,
        elections,
        elections_equipment,
        instructions_forms_reports,
        rlas,
        roles
END terminology

dashboards: THEORY
BEGIN
    IMPORTING
        dashboard,
        department_of_state_dashboard,
        county_dashboard,
        audit_board_dashboard,
        public_dashboard
```

```

END dashboards

system_specification: THEORY
BEGIN
  IMPORTING
    data_model,
    logging,
    protocol,
    system_architecture,
    system_assumptions

END system_specification










corla: THEORY
BEGIN
  IMPORTING
    specification_theories,
    background_terminology,
    terminology,
    dashboards,
    system_specification
END corla

```



## Todo list

■ Add annotation functions coupled to milestones. -kiniry . . . . .	6
■ Need to break the cycle between the contest type and the ballots theory. sfsinger suggests: contests are grouped into elections, not ballots. So define contests first. Then a ballot is created to represent a set of contests. -kiniry . . . . .	8
■ Add an axiom for contest name uniqueness. -kiniry . . . . .	9
■ These types will be refined when we review the information provided by CDOS on 20 July 2017 about Dominion’s file formats. -kiniry . .	9
■ Revise this definition. -kiniry . . . . .	9
■ Find sources for these, other than “Dwight Shellman said so” -sfsinger	11
■ Add an axiom for uniqueness of scanner id numbers. -kiniry . . . . .	11
■ Need to clarify choices vs options vs votes. -kiniry . . . . .	13
■ Need to add model information for write-in votes. -kiniry . . . . .	13
■ Should this information also include what kind of ballot each ballot is? I.e., re-marked, phantom, etc.? -kiniry . . . . .	14
■ These types will be refined when we review the information provided by CDOS on 20 July 2017 about Dominion’s file formats. -kiniry . .	14
■ Ballot position number is still not explicitly specified. -kiniry . . . . .	14
■ Should we define undervote specifically as 0 votes? -nealmcb . . . . .	15
■ Should we model a blank vote separately from an undervote? Or do they simply use the term blank vote to denote an undervote? -kiniry	15
■ Introduce appropriate axiom for such. -kiniry . . . . .	15
■ Still need to model core RLA algorithm(s). -kiniry . . . . .	23
■ For a given election, we cannot expect a 1-1 correspondence between its ballots and its CVRs. Phantom ballots and human error will result in a violation of such a property. -kiniry . . . . .	23
■ It is unclear what a CVR number is, if anything. -kiniry . . . . .	24
■ In the audit adjudication interface, it must be possible to classify a ballot as unauditable either due to not finding a voter-verifiable paper record or due to it being a phantom ballot. -kiniry . . . . .	24
■ Obviously this type has to be strengthened considerably to guarantee permutation. That is, that both the set and the list are finite and converting the list into a set results in a set equivalent to the original set. -kiniry . . . . .	29
■ Write a specification for publishing data to audit. -kiniry . . . . .	29
■ Improve/strengthen the dependent types in these upload signatures below. -kiniry . . . . .	31
■ @review cdos Must the Department of State give an explicit go-ahead, or can the county simply immediately start the audit? -kiniry . . . .	31
■ Introduce an authentication monad and a state monad for our ASMs. -kiniry . . . . .	36
■ Add election as first column to every table? Or have a different DB per election? -kiniry . . . . .	44
■ Explain ASM definition. -kiniry . . . . .	45

	Discuss upload_system_message. -kiniry . . . . .	46
	Transport specs for Department of State Dashboard. -kiniry . . . . .	51
	Transport specs for County Dashboard. -kiniry . . . . .	52
	Transport specs for Audit Board Dashboard. -kiniry . . . . .	52
	Discuss HTTP endpoint types and endpoints. -kiniry . . . . .	52
	Explain these relations and their use. -kiniry . . . . .	59
	Add citation or explanation of encoding of sets and pairs. -kiniry . . .	62
	Add response code for all external public events. -kiniry . . . . .	62
	Explan how we are doing the definition of data types and the derivation of JSON encodings. -dmz . . . . .	62