Specifying and Verifying Requirements for Election Processes

Borislava I. Simidchieva, Matthew S. Marzilli, Lori A. Clarke, Leon J. Osterweil
Laboratory for Advanced Software Engineering Research (LASER)
Computer Science Department
University of Massachusetts, Amherst
140 Governors Drive, Amherst, MA 01003
{bis | mmarzill | clarke | ljo}@cs.umass.edu

ABSTRACT
In this paper we outline an approach for modeling election processes and then performing rigorous analysis to verify that these process models meet selected behavioral requirements. We briefly outline some high-level requirements that an election process must satisfy and demonstrate how these are refined into a collection of lower-level properties that can be used as the basis for verification. We present a motivating example of an election process modeled using the Little-JIL process definition language, capture the lower-level properties using the PROPEL property elicitation tool, and perform formal analysis to verify that the process model adheres to these properties using the FLAVERS finite-state verifier. We illustrate how this approach can identify errors in the process model when a property is violated.

Keywords
Process, elections, requirements, properties, verification

1. INTRODUCTION
The ability to carry out fair elections has long been considered a cornerstone of democracy. Election defects and frauds have been alleged on many occasions in the past, and such allegations have been particularly prominent in recent elections in the United States. There have been many attempts to use technological approaches to try to assure that elections are fair. Recently these attempts have focused primarily upon the development and use of Direct Recording Electronic (DRE) machines. DREs are intended to assure that votes are captured and tabulated correctly. As such, they are addressing an important aspect of the election and warrant the scrutiny that they are now receiving. But we note that an election entails far more than just the casting and tabulating of votes. A full election process starts with such activities as the establishment of voter rolls and the qualification of poll workers, and extends through final tabulations, and, perhaps, recounts and challenges. We note that this far larger and more extensive election process creates possibilities for many types of errors that cannot be addressed by work that solely focuses on analyzing DREs.

In this paper we demonstrate technologies for reasoning rigorously about the presence or absence of errors during all phases of an election process. In particular, we suggest that it is feasible to define precisely all the phases of an election process, ranging from the earliest activities through recounts, to define the desired low-level requirements, or properties, to which such a process must adhere, and then to reason about whether the process adheres to those properties. Election processes are hard to define clearly and precisely, because they are large and complex, and thus we introduce Little-JIL [26], a rigorously defined language that seems adept at supporting the precise definition of such processes. The properties to which elections must adhere are diverse and hard to define precisely, and thus we introduce PROPEL [24], a system for facilitating the definition of such properties. Finally we note that Finite-State Verification (FSV) can be used as a vehicle for reasoning definitively about whether an election process model adheres to defined properties and describe how the FLAVERS FSV system [11] was used to verify the adherence of a defined election process model to defined properties. When FLAVERS determines the possibility that a process model could result in the violation a property it indicates an example of how the violation can occur. We demonstrate how this feedback can then be used to guide the improvement of the process model.

Thus, this paper indicates the potential of applying rigorous approaches to election processes to help assure that elections are fair. We show that for specified properties, the rigorously defined election process model can either be shown to be consistent with these properties or, if it is not consistent, example traces can be provided to show how the process model could violate the properties. These example traces can be used to help modify the process model so that it no longer violates the properties. If the error is actually associated with the real process, as opposed to merely being a mistake in the encoding of the process model or the properties, then the real-world process should be modified as well.

Section 2 summarizes previous work in improving elections and in using processes definitions (models) to support process improvement in other domains. Section 3 introduces the notion of rigorous process definition and uses the Little-JIL process definition language to define a small portion of an example election process. Section 4 shows how election process requirements may be first developed at a very high level, but must then be clarified by being refined and decomposed into precisely-stated, lower-level properties. We use the PROPEL property elicitation tool to specify these properties and generate a representation that is suitable basis for verification. Section 5 shows how the FLAVERS finite-state verifier is used to determine the adherence of the process model described in section 3 to properties described in section 4. Section 5 also shows how FSV can suggest ways to improve the process model. Section 6 summarizes our approach and suggests directions for future work.

2. RELATED WORK

As noted there has been major interest in studying the conduct of elections, especially with respect to electronic voting machines [5, 2]. Many aspects of elections have been studied. The risks associated with the use of DREs stemming from both errors in their operation and their susceptibility to various forms of fraud have recently been explored [1, 3]. Jones [15] provides a good survey of the various voting technologies that are used in elections, including DREs. Our work seeks to develop election process models at a higher level of abstraction, so that the actions of any vote recording mechanism are a small, although crucial, piece of the overall process.

Many states have begun to reevaluate the use of electronic voting machines because of security and reliability concerns. Florida, Maryland, and New York, among others, have performed extensive studies on the use of DREs. Most recently, Ohio commissioned a study to evaluate the voting machines used in the state. The study, called Project EVEREST (Evaluation & Validation of Election-Related Equipment, Standards & Testing), is the most comprehensive to date and included independent panels of industry and university experts who performed security assessments of the electronic voting machines used in Ohio. The report's findings [7] concludes that the e-voting machines used in Ohio (these machines include models from three different DRE vendors, namely Premier Election Systems–formerly Diebold, Election Systems & Software, and Hart InterCivic) "failed to adopt, implement and follow industry standard best practices in the development of the system."

Validation and verifiability of voting systems are actively researched; Mercuri and Neuman [18] give an overview of how electronic voting systems can be verified and emphasizes the importance of a verifiable paper trail, and Saltman [22] outlines different techniques for performing auditing and improving public confidence for both ballot and non-artifactual systems. Saltman, however, fails to identify the importance of static analysis and only relies on testing of DREs. While our approach also provides a framework for verification, we concentrate on the holistic election process rather than any one aspect of election technology or conduct. With this focus in mind, we attempt to verify properties that are general requirements of most election processes.

There has been some effort on specifying holistic election processes and their requirements, notably [4] advocates precise specification of election processes, which is similar to our approach on a high level but does not identify concrete technologies or formalisms that can be used to achieve this goal. Our work allows the process to be rigorously and unambiguously defined, and the framework we use allows us to perform formal verification. Raunak [21] uses the same approach to define election processes and reason about frauds and collusion, while we reason about correctness.

Significant research has been done on identifying requirements that elections must satisfy, such as privacy, anonymity, ensuring each eligible voter can cast at most one vote, accessibility, etc. Mitrou et al [19] outline several high level requirements that elections must satisfy and what implications these requirements might have on the voting technology used but, unlike our approach, these requirements are specified at a very abstract level, making it very difficult to reason whether or not an election process satisfies them. Lambrinoudakis et al [16] similarly identify a small set of requirements and reason about how these may interrelate or conflict with each other once they are elaborated into a lower level of abstraction but this reasoning is limited to manually examining the English definitions of the requirements, which are inherently imprecise. By precisely defining the properties as finite-state machines and using finite-state verification, we are able to perform a more rigorous evaluation of whether the requirements are satisfied.

This paper focuses on demonstrating a software framework for defining and reasoning about election processes. We define these processes in a manner that facilitates the use of software analysis techniques to reason about the presence or absence of errors in the process. This strategy builds upon previous work in which process definitions were used to reason about election vulnerability with respect to incorrect or fraudulent behaviors by election officials [21]. Process definitions have also been used as the basis for reasoning about processes in several other domains, including science[6, 20], medicine [8, 13], and business[12, 25].

3. DEFINING THE PROCESS

The example discussed in this paper is a subset of an extended, more detailed process definition, which is in itself a subset of a real-world process. For simplicity, this example process defines an election with two opposing candidates. The process also assumes a physical ballot and includes some steps that assume the existence of a voting machine, but no assumptions are made about the voting machine technology. In an attempt to make the process general enough to accommodate different voting machine technologies, it is defined at a high level of abstraction and does not include technology-specific details.

On a high level, the process outlined in Section 3.2 consists of three main phases–the pre-polling events that must occur before election day, the events that occur on election day, and the counting of the votes after the election is over. The example process focuses primarily on the events that occur on election day, and therefore some of the process details that are not directly relevant (for example the details of how registration is carried out or how recounts are handled) are omitted.

The pre-polling events include training the election officials and registering eligible voters. Before election day, the voting rolls (i.e. the lists of registered voters) are generated and distributed to the precincts. The events that occur on election day include the initial checks and setup of any voting machines that may be used and the actual voting process, which encompasses verifying the voter's credentials and allowing the voter to cast a ballot (in the case where the voter has been previously checked off as having voted, the voter casts a provisional ballot, which is kept separately from the regular ballots).

After the election is complete, the vote totals are recorded and transmitted to a higher-level election authority that may consolidate totals from multiple voting sites, may do verification, and declare official election results. A recount procedure is specified in exceptional situations, such as when the reported totals from a precinct do not pass a sum consistency check (i.e. the total number of votes for the first candidate plus the total number of votes for the second candidate does not equal the total number of votes cast). A high-level overview of the process as modeled in the Little-JIL process definition language appears in Figure 1. Before explaining the process in detail, we present a brief overview of the features of the Little-JIL language.

3.1 Overview of Little-JIL Features

Little-JIL[26] is a visual process definition language that provides rich semantics. In this discussion, we only describe the features pertinent to the example election process defined in section 3.2. A Little-JIL process coordination diagram, such as the one shown in Figure 1, consists of a hierarchical decomposition of steps. A step is denoted by a black bar, with the step name appearing above that bar. Associated with each step is an agent that is responsible for its execution. The behavior of a step consists of the behaviors of its children (the steps that connect to the lower left side of the parent step bar via edges) and the order in which they are executed. Each step that has children also has a sequence badge, which appears in the left half of the step bar and determines the order in which its children will be executed. For example, the root step `conduct election` in Figure 1 has an arrow, which means it is a sequential step, and hence all of its children will be executed in left to right order. The `pre-polling checks` step has an equal sign as its sequence badge indicating it is a parallel step, which means its children can be executed in any order, including any arbitrary interleaving.

A step without children is called a leaf step and responsibility for execution of such a step is left entirely to the step's agent. A step is reasonably thought of as a procedure that is invoked whenever there is a reference to that step from anywhere in the process.

In addition to the coordination diagram, a Little-JIL process definition includes agent specifications and artifact specifications. One of the important features of Little-JIL is that it allows the definition of both human and automated (executed by a hardware device or software system) agents and does not discriminate between them. For the election process, Voter, Election Official, and Registration Place are some examples of types of agents. Note that the former two are human agents while the latter can be an automated agent. Little-JIL processes only specify the type of agent (e.g. Voter) that should execute a specific step, rather than a specific agent instance (e.g. Jane Doe). Little-JIL employs a dynamic resource management system, which assigns the appropriate agent instances to types during process execution.

Little-JIL also provides comprehensive exception handling semantics. For example, the `do recount` step in Figure 1 connects to the **X** in the right half of the step bar of its parent, `conduct election`, which indicates that `do recount` is an exception handler. Exceptions in Little-JIL are typed and different exception handlers must be defined for each exception kind. `Do recount` is an exception handler for exceptions of type `Checksum` as the edge that connects `do recount` to its parent indicates.

The artifact specification consists of all the artifacts that are used in the process, which for this election process example includes Voter preference (this may be a paper ballot or some sort of a persistent record in a voting machine) and Voting roll (the list of all registered voters for a given precinct). Each step has artifact declarations to define which artifacts it will be accessing or providing. Artifacts are generally passed within the coordination hierarchy (from parents to children and vice versa). As steps can be thought of as procedures, this artifact passing is essentially a parameter passing mechanism.

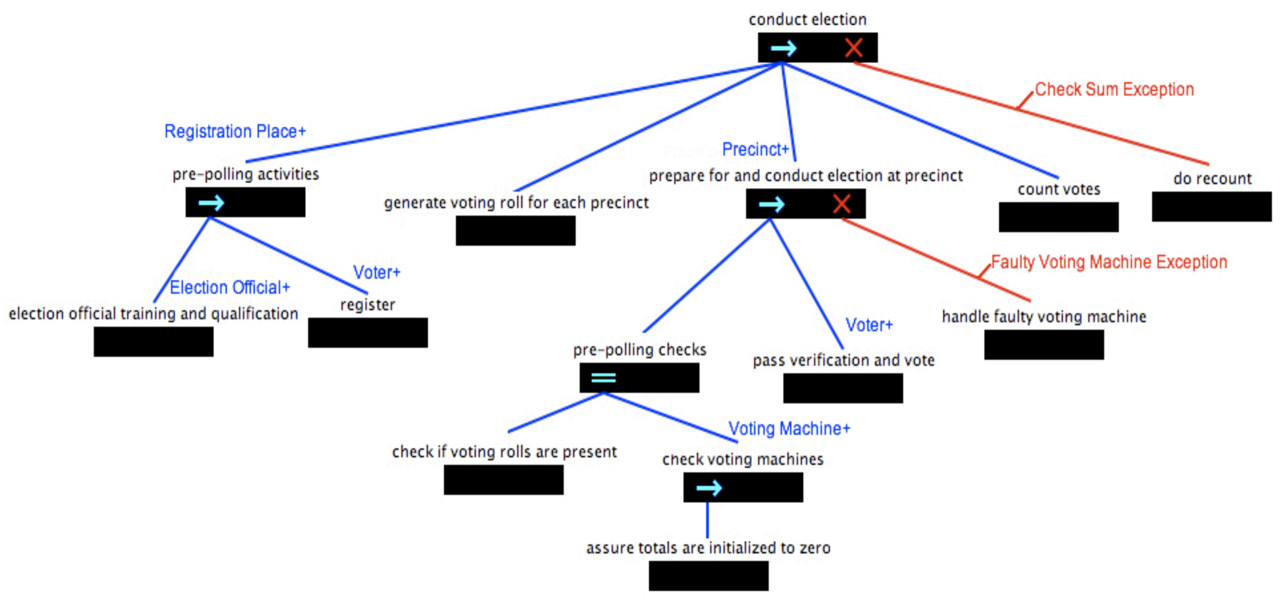
3.2 An Example Election Process

The diagram in Figure 1 outlines the high level election process used in this example. As noted earlier, the root step `conduct election` is a sequential step and so all of its children, namely `pre-polling activities`, `generate voting roll for each precinct`, `prepare for and conduct election at precinct`, and `count votes` are to be executed in this specific order. Note that the edge connecting `conduct election` to `pre-polling activities` has the notation `Registration Place +`. This indicates that the `pre-polling activities` step will be instantiated one or more times, once for each agent that executes it. In the case of the `pre-polling activities` step, the agent is specified to be of type Registration Place, so this step will be executed once for each agent instance of type Registration Place, which effectively enumerates all Registration Places specified in the system.

The `pre-polling activities` step is also a sequential step and consists of `election official training and qualification`, followed by the `register` step. Note that the edges to both children also have the `Election Official +` and `Voter +` notations, respectively, so they will be instantiated for each agent that executes them. Thus, the `election official training and qualification` step will be invoked once for each agent of type Election Official, and the `register` step will be executed once for each Voter type agent.

After the `pre-polling activities` are completed, `conduct election` proceeds by next executing `generate voting roll for each precinct`. Once the voting rolls are generated, the `prepare for and conduct election at precinct` step is executed. Since the edge leading to it again has the `Precinct +` notation, this step is also multiply executed, once for each executing agent, in this case specified to be of type Precinct. `Prepare for and conduct election at precinct` is a sequential step, and it consists of executing the `pre-polling checks` first, followed by the `pass verification and vote` step.

As noted earlier, `pre-polling checks` is a parallel step (it has an equal sign as its sequence badge) and its children can therefore be executed in any order, including parallel execution. The children in this case are the `check if voting`



conduct election
Check Sum Exception
Registration Place+
pre-polling activities
generate voting roll for each precinct
Precinct+
prepare for and conduct election at precinct
count votes
do recount
Faulty Voting Machine Exception
Election Official+
election official training and qualification
Voter+
register
handle faulty voting machine
pre-polling checks
Voter+
pass verification and vote
check if voting rolls are present
Voting Machine+
check voting machines
assure totals are initialized to zero

Figure 1: Little-JIL diagram for the `conduct election` process

`rolls are present` step and the **`check voting machines`** step, which is instantiated multiple times because it has the `Voting Machine +` notation (i.e. we perform this check once for every machine). The **`check voting machines`** step consists of completing the step **`assure totals are initialized to zero`**. Once **`pre-polling checks`** completes, **`prepare for and conduct election at precinct`** continues by executing its second child step, namely **`pass verification and vote`**, which is executed once for each agent of the type specified in the agent declaration, in this case a Voter agent (denoted by the `Voter +` notation).

Note that **`prepare for and conduct election at precinct`** has one more child, namely **`handle faulty voting machine`**, which connects to the **X** in the right half of the step bar via an edge. This step is an exception handler and illustrates Little-JIL's built-in support for dealing with exceptional behavior. In this case, the process specifies that the **`check voting machines`** step can trigger a **`Faulty Voting Machine`** exception during the **`pre-polling checks`**. A Faulty Voting Machine exception can also occur during the actual voting process, namely in the **`pass verification and vote`** step. Because of the fact that **`prepare for and conduct election at precinct`** is a common ancestor of both of these steps, one handler at the parent step can handle the exception regardless of which child triggers it. Once the election at all precincts completes, the execution continues with the **`count votes`** step, where the vote totals are tallied. The **`count votes`** step can trigger an exception in the case of a discrepancy, in which case the **`do recount`** exception handler of the **`conduct election`** step is invoked.

The **`pass verification and vote`** Little-JIL subprocess is further elaborated in Figure 2. The **`pass verification and vote`** step consists of executing the **`present ID`**, **`perform pre-vote verification`**, **`check off voter as voted,`** and, finally, **`record voter preference`** in exactly this order. The step **`present ID`** can trigger an exception, **Missing ID**, if no identification is presented, or an Inadmissible ID exception if the identification presented does not meet the established criteria. In both cases, the exceptions are handled by exception handlers emanating from below the **X** in the right side of the step bar of the **`pass verification and vote`** step. The behavior of these exception handlers specifies that in the case of either exception, process execution will continue as if the **`pass verification and vote`** step has completed (denoted by the white check mark in the dark circle). This exception-handling scenario effects the preemption of the voting process and thus prevents the voter from casting a vote in the case where the voter cannot provide an acceptable form of identification.

The **`perform pre-vote verification`** comprises three steps that are executed in parallel, namely **`confirm voter ID matches voter`**, **`confirm voter ID matches voting roll`**, and **`verify voter has not voted`**. If **`confirm voter ID matches voter`** or **`confirm voter ID matches voting roll`** fails, an `ID Mismatch` exception is triggered and similarly, in the case **`verify voter has not voted`** fails, a `Voter Already Checked Off` exception is triggered. The `ID Mismatch` exception is handled similarly to the exceptions raised in the `present ID` step outlined above–the process continues as if **`pass verification and vote`** has completed and thus the voter is unable to vote. The `Voter Already Checked Off` exception involves a more complicated compensation mechanism, specified in the **`handle voter already checked off exception`** exception handler. This exception handler consists of a single step, **`let voter vote with provisional ballot`**, which in turn reinvokes the **`record voter preference`** procedure. Thus, if a voter has already been checked off but presents

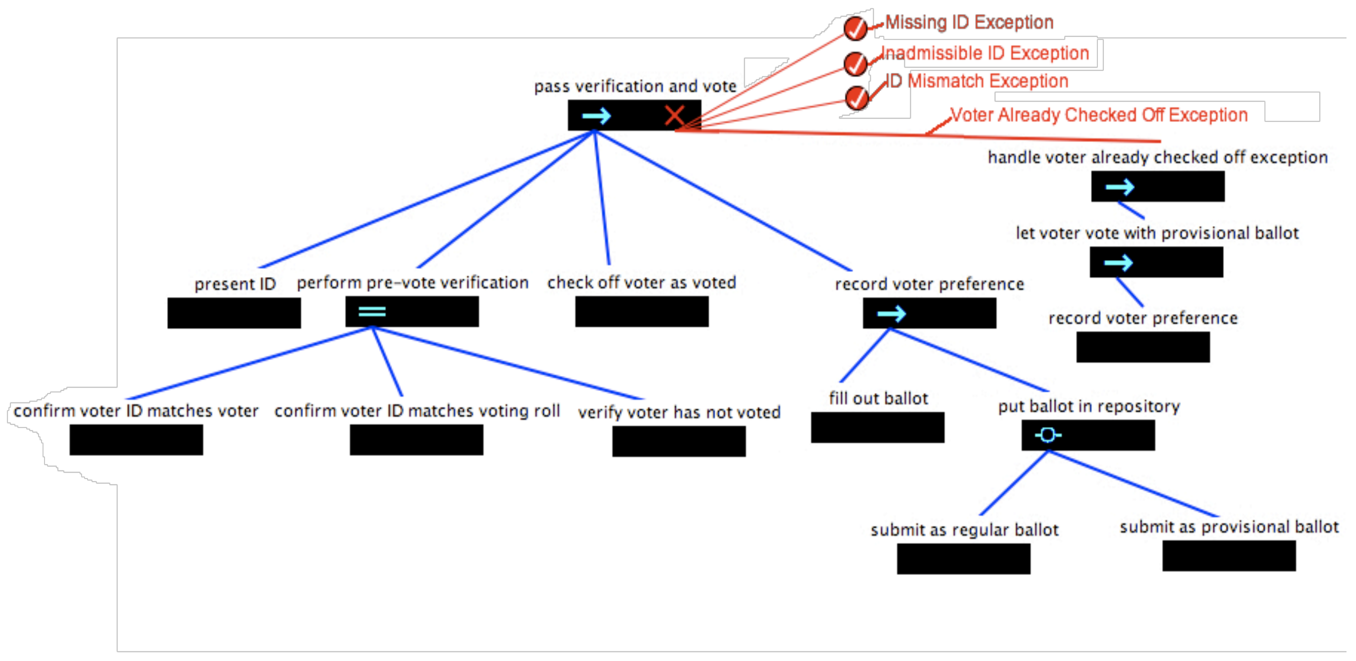

Missing ID Exception
Inadmissible ID Exception
ID Mismatch Exception
Voter Already Checked Off Exception
pass verification and vote
handle voter already checked off exception
let voter vote with provisional ballot
record voter preference
present ID
perform pre-vote verification
check off voter as voted
record voter preference
confirm voter ID matches voter
confirm voter ID matches voting roll
verify voter has not voted
fill out ballot
put ballot in repository
submit as regular ballot
submit as provisional ballot

Figure 2: Little-JIL diagram for the `pass verification and vote` subprocess

proper credentials, a provisional ballot is cast. This ballot is kept separate from the regular ballots.

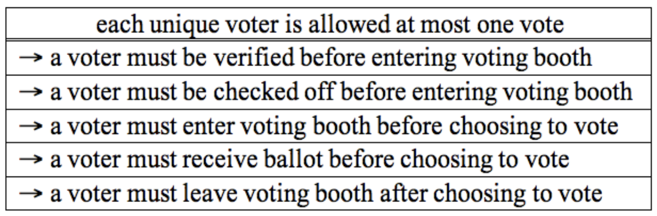
After a voter's ID has been verified successfully in the `perform pre-vote verification step`, the `check off voter as voted` step is executed. The voter is also given a ballot in this step. Lastly, the `record voter preference` step is a sequential step, which involves `fill out ballot` and then `put ballot in repository`. `Put ballot in repository` is a choice step (denoted by the blue circle with a bar going through it), which means that it can be executed by executing either one of its children, namely `submit as regular ballot` or `submit as provisional ballot`. A choice is made according to the circumstances–if the voter had not previously been checked off, then `submit as regular ballot` is chosen; if, on the other hand the voter is voting as a result of a ballot dispute, `submit as provisional ballot` is chosen.

4. REFINING THE REQUIREMENTS

To reason that our example process model is correct, it must be evaluated with respect to the requirements for the process. There are many requirements that a real-world election process may need to satisfy. Some examples of these are:

- each eligible voter is allowed at most one vote;
- the election is safe with respect to a voter's privacy and anonymity; and
- voters cannot be coerced (i.e. voters cannot be threatened or tricked into voting a certain way).

These requirements are specified at a high level of abstraction in an attempt to keep them independent from the specific manner in which an election is carried out. For example, an election process that uses physical ballots or allows voters to vote remotely by means of e-voting should still satisfy these requirements. While the specification of high-level requirements may be a useful starting point for specifying election intent, it is often difficult to formally verify that these requirements are satisfied by a particular process definition without first refining each requirement into a collection of lower-level, observable properties.



each unique voter is allowed at most one vote
→ a voter must be verified before entering voting booth
→ a voter must be checked off before entering voting booth
→ a voter must enter voting booth before choosing to vote
→ a voter must receive ballot before choosing to vote
→ a voter must leave voting booth after choosing to vote

Figure 3: Refinement of a high-level requirement into lower-level properties

Refined properties provide a more precise description of the overall requirement, but are sometimes specific to certain types of election processes. As an example, in Figure 3,we provide a refinement of the requirement "each eligible voter is allowed at most one vote" into a collection of lower-level properties. These properties assume an election where there are physical voting booths and physical ballots, and thus this refinement is specific to certain types of election processes. For e-voting elections, the example election requirements we have provided and others were described by Mitrou et. al [19]. The dependence of the properties in Figure 3 on specific election processes suggests that different refinements of the same high-level requirement would be necessary to support different election processes.

As previously mentioned, Lambrinoudakis et al [16] also considered a set of requirements for e-voting elections and discussed how they are related to a group of lower level abstractions necessary for formal verification. The requirements of [16] and the properties given in Figure 3 are defined in natural language, a notation that is often imprecise and ambiguous. For example, the property that states "a voter must be verified before entering voting booth" says nothing about whether a voter can be verified more than once before entering a voting booth or whether, after being verified, the voter can enter the voting booth again. To formally verify that a process definition satisfies the properties in our example will require a more precise property specification.

To provide precise specifications of properties we used PROPEL (PROPerty ELucidator [24]), a software tool that helps users formalize all the details associated with a property specification. PROPEL provides templates for commonly occurring property specification patterns [10]. Each template provides a set of pre-defined options that must be considered in order to define a property precisely. The templates are represented in three alternative, but equivalent, notations: a collection of disciplined English sentences where the user selects the appropriate phrases, a hierarchical series of questions where the user indicates the answers, or as an extended Finite-State Automaton (FSA) where the user selects transitions, labels, and accepting states as allowed by the options associated with a template. For each property in Figure 3, PROPEL was used to create a FSA that provides precise property definition of what is intended. PROPEL properties are specified independently from the Little-JIL language, reflecting the fact that an election process defined in any process definition language must satisfy these properties.

After all the options are specified, the detailed property specification can be quite long to state in natural language. The following is the PROPEL disciplined English representation of the property "voter must be verified before entering voting booth."

VoterEntersVotingBoth *cannot occur until after* ***VoterIsVerified*** *has occurred.* ***VoterIsVerified*** *is not required to occur, however.*

VoterIsVerified *can occur multiple times before the first subsequent* ***VoterEntersVotingBooth*** *occurs.*

After ***VoterIsVerified*** *occurs other events can occur before the first subsequent* ***VoterEntersVotingBooth*** *occurs*

After ***VoterEntersVotingBooth*** *occurs neither* ***VoterIsVerified*** *nor* ***VoterEntersVotingBooth*** *can occur again.*

The bolded terms in these disciplined English sentences are called events. This property is intended to specify the order in which events can occur in the process definition. The set of events that are relevant to a property is called the property alphabet. There are usually many different events that may occur in an election process, but each property is typically only concerned with a small subset of them. In this example the property alphabet consists of **VoterIsVerified** and **VoterEntersVotingBooth**. **VoterIsVerified** corresponds to the voter's ID being successfully checked by a poll worker, and **VoterEntersVotingBooth** corresponds to the voter entering the physical voting booth.

The FSA in Figure 4 is the equivalent representation of the property "voter must be verified before entering voting booth" given above. It defines the acceptable order in which events can occur for any trace through the process model. Each of the numbered nodes in this figure represents a state in the FSA The starting state is annotated with an arrowhead. Initially the current state is set to the starting state. The arrows between states are called state transitions. State transitions are labeled with an event name or a list of event names from the property alphabet. If one of the events on the transitions emanating from the current state occurs, then the current state is updated to the target state of that transition. An event annotation that has a negation sign ($\neg$) indicates that the transition will occur if any event within the property alphabet occurs other than the negated events. The FSA representation that is used is deterministic and total, meaning that for each state all of the events in the property alphabet must occur on one and only one transitions emanating from that state, If an event occurs that causes the sequence of events to be acceptable, then the current state should be updated to a special state, called the violation state. All the transitions from the violation state are self-loops, so if the current state becomes the violation state, it will remain so. To keep the FSA representation in Figure 4 small and easy to understand, the violation state and all the transitions to and from that state are not shown. Thus, whenever a state in this figure does not contain a transition for an event, there is in fact an unshown transition for that event to the violation state.

In Figure 4, state 1 is the starting state. If the event **VoterIsVerified** occurs, the current state is updated to state 2 since there is a transition from state 1 to state 2 labeled with the event **VoterIsVerified** occurs, meaning the next subsequent event will occur from state 2. On the other hand, if the current state is state 1 and the event **VoterEntersVotingBooth** occurs the current state is updated to the violation state since the event **VoterEntersVotingBooth** is not a label for either of the transitions emanating from state 1. This FSA therefore specifies the requirement that the voter must be verified before entering a voting booth and that once the voter has entered a voting booth, he or she cannot enter a voting booth or be verified again.

5. PROCESS MODEL VERIFICATION AND IMPROVEMENT

We use FSV to determine if the process definition satisfies each property associated with that definition. FSV constructs a finite model of the process that represents all the possible event sequences, for the events in a property, that could occur for all the possible traces through the process definition. It then attempts to determine if this model is consistent with the property specification[9, 17, 11, 14]. If the model is not consistent with the property, then a counter example trace through the model is shown so the user can see the event sequence associated with a trace that causes the

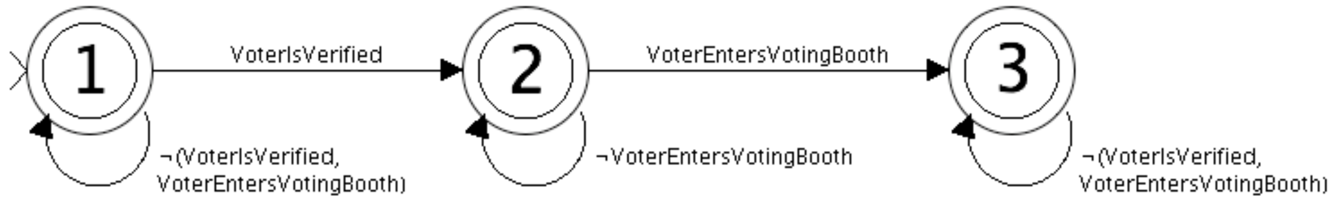

VoterIsVerified
VoterEntersVotingBooth
¬(VoterIsVerified, VoterEntersVotingBooth)
¬VoterEntersVotingBooth
¬(VoterIsVerified, VoterEntersVotingBooth)
1
2
3

Figure 4: Finite-state automaton corresponding to the "voter must be verified before entering voting booth" property

FSA to enter the violation state. This inconsistency could indicate imprecision within the finite model, an incorrectly defined property or process definition, or it could indicate an error in the actual process that was represented in the corresponding process definition.

Using FSV to formally verify that a process definition satisfies some property requires determining the correspondence between the process steps and the events within that property's alphabet. For example, the events in Figure 4 should correspond to specific steps that can occur in an election process definition. This correspondence is explicitly defined indicating a binding between process steps and property events.

5.1 Binding the Process Steps to the Property Events

The binding of events of a property alphabet to steps of a Little-JIL process definition involves not only matching event names to step names, but also determining the appropriate state of each step to bind to the event. Within the Little-JIL execution model, each step progresses through several states before it is considered to have completed successfully or to have failed to complete and triggered an exception. When a step first becomes available to an agent, its state is `POSTED`. When an agent starts to perform a step, its state is `STARTED`.When a step has been completed successfully, its state becomes `COMPLETED`. If, on the other hand, the step cannot complete successfully and triggers an exception, its state becomes `TERMINATED`.

A binding between event names and step states is necessary when it is important to reason about where a step is in its execution. Although it is often the case that an event can be bound to the `STARTED` state, sometimes finer grained control is needed. For example, we may want an event to occur only when an agent completes a step. When necessary an event can also bind to more than one step and more than one state of a step. Clearly, to specify bindings between event names and step states, one should have a deep understanding of the process definition.

Figure 5 provides example bindings between the event names used in Figure 4and step states in the process defined in Figure 2. In this example, the event **VoterIsVerified** is bound to the process step `perform pre-vote verification` with the specific state `COMPLETED`. This means that when a voter has successfully completed the step `perform pre-vote verification` the event **VoterIsVerified** occurs. Similarly

Event	Step State
VoterCheckedOff	`check off voter as voted.COMPLETED`
VoterEntersVotingBooth	`record voter preference.POSTED`
VoterIsGivenBallot	`check off voter as voted.COMPLETED`
VoterIsVerified	`perform pre-vote verification.COMPLETED`
VoterLeavesVotingBooth	`record voter preference.COMPLETED,` `record voter preference.TERMINATED`
VoterVotesOrDoesNotVote	`put ballot in repository.COMPLETED,` `put ballot in repository.TERMINATED`

Figure 5: Bindings of property events to step states

the event **VoterEntersVotingBooth** is bound to the step `record voter preference` with the specific state `POSTED`. This means that when this step first becomes available to an agent the event `record voter preference` occurs.

5.2 Applying FLAVERS

FLAVERS was used to verify that the example election process definition satisfies the properties in Figure3. FLAVERS was developed to verify Ada and then Java programs, but has been extended to include support for verifying Little-JIL process definitions. FLAVERS is able to produce traces through the process definition that could lead to unsatisfied properties. These traces can be helpful in identifying possible errors in the process definition. Moreover, as we demonstrate in the next section, analyzing the process definition and these traces can suggest process definition improvements capable of remedying these errors.

5.3 Example of Process Model Improvement

To develop a process model that defines the election process correctly and provides sufficient detail, it is helpful and convenient to make iterative process model improvements. This is especially true when processes are modeled in a process definition language that uses hierarchical decomposition. Little-JIL allows the process model to be defined at a very high level initially, then iteratively improved or elaborated to include more details. Most of the changes that were made to the election process model described in section 3 as it was being developed were such iterative improvements and concentrated on improving the level of abstraction of the process model to assure that there is a direct correspondence between steps in the process model and events in properties.

Sometimes, however, verifying a process model against a set of properties resulted in finding that it is possible for the process model to violate a property to which it is expected to adhere. This means there is an error in the process model, the real-world process, or the property. Carefully inspecting the FLAVERS trace of step states that could result in a violation helps to identify the source of the error. Once the source is identified, if the error is in the process definition or the property, appropriate corrections are made and verification is performed again to ensure that no other properties can be violated as a result of the changes. If the error detected is a manifestation of an error in the real-world process, then the process must be modified in consultation with domain experts. An improved process can then be proposed, and the corresponding process model and properties can be updated correspondingly and then reverified.

Through careful specification of the process model and properties and iterative refinement of both, we discovered a serious error in the process definition, which was not immediately obvious. As previously described, the `perform pre-vote verification` step in Figure 2 is executed in parallel. This process definition resulted from the belief that election officials may wish to perform the three verification steps, namely `confirm voter ID matches voter`, `confirm voter ID matches voting roll`, and `verify voter has not voted` in any order, including parallel execution (i.e. having three election officials execute the three checks at the same time). Since `pass verification and vote` is instantiated multiple times to accommodate multiple voters, we discovered that our process model includes a loophole that allows a voter to vote even if they are ineligible to do so.

Consider the following scenarios in which Jane Doe and Joan Smith go to vote. In the `pass verification and vote` step, Jane presents her identification, then during the `perform pre-vote verification` step election officials verify that the identification matches Jane and her registration information and that Jane has not yet voted. Jane is checked off as having voted in the `check voter as voted` step and upon the successful completion of that step, she is handed a ballot. She fills out and submits it as a regular ballot during the `record voter preference step` and, with that, the instantiation of the `pass verification and vote` step for voter Jane Doe successfully completes. Now Joan Smith, who is ineligible to vote, attempts to vote as Jane Doe. The `pass verification and vote` step gets instantiated and Joan presents her identification in the `present ID` step. Since Joan is an impostor, during the `pre-vote verification` step two exceptions arise–ID Mismatch exception because Joan's ID does not match Jane's information, as well as a Voter Already Checked Off exception since Jane successfully cast her ballot. Since the `pre-vote verification` step happens in parallel, both of these exceptions have to be handled. Since the ID does not match, the original instantiation of the `pass verification and vote` step for Joan is completed but since Jane was already checked off as having voted, the `handle voter already checked off exception` handler is invoked, allowing Joan Smith to cast a provisional ballot while impersonating Jane Doe. While this example may seem contrived and specific to this model, it is not hard to imagine how seemingly simple errors like this can occur in real life because of poor communication among the election officials or insufficient training on how to handle exceptional situations.

After discovering this error in the process model as a result of attempting to verify the specified properties outlined in Figure 3, the process model was changed to remedy this error. In this case, simply changing the `perform pre-vote verification` step from a parallel to a sequential execution solves the problem. By forcing the ID verification steps to occur first, any **ID Mismatch** exception is detected and handled before `verify voter has not voted` can even occur. Thus, the whole `pass verification and vote` process is completed if an **ID Mismatch** exception is triggered and an ineligible voter cannot vote with a provisional ballot under any circumstances. The property `voter must be verified before voting` was successfully verified after this change was made to the process model, and the model still adhered to the remaining properties.

5.4 Evaluative Observations

In this section, we present some observations about the challenges we encountered while defining the process model and the requirements, refining the requirements, binding the steps in the process model to property events, and performing finite-state verification on the resulting model.

It was somewhat difficult to define the process carefully, however, the fact that Little-JIL provides hierarchical decomposition made it easy to initially define the process at a very high level of abstraction and then iteratively elaborate it to lower levels of abstraction. Our approach facilitates detection of process definition errors as illustrated in the previous section. Additionally, since correcting the process model error described in section 5.3 did not result in a significant change in the coordination diagram, none of the bindings between the process model steps and the property events were changed. As a result, there was little overhead involved in running the FLAVERS system once again to assure that the new process model satisfied the original properties.

The initial high-level requirements that were defined in English required iterative refinement into precisely-defined, lower-level properties that contain enough detail to make them amenable to static analysis. As we illustrated, and as previous work indicates [19, 16], this refinement is often difficult. However, the PROPEL property elicitation tool significantly facilitated this process by providing guidance; the template of hierarchical series of questions provided by PROPEL was especially useful.

As noted above each high-level requirement was decomposed into a set of lower-level properties. In doing so we discovered that some lower-level properties occurred in more than one of these decompositions. This resulted in fewer lower-level properties than initially expected ([16] observed similar behavior). This allayed some of our early concerns that the total number of properties in need of verification in a realistic election process might be excessive. Originally, one of the properties specified, namely "voter must be verified before entering voting booth" was violated, but after correcting the detected error in the process model, all five properties (enumerated in Figure 3), which together define the high-level requirement that each unique eligible voter must be allowed at most one vote, were successfully verified.

The ability to perform formal verification of election processes was unquestionably useful. The formal verification approach we used also facilitated both error detection and localization–that is, it not only detected the presence of errors but also provided guidance in finding where the errors occur. Thus, it was extremely valuable in helping to improve the election process.

6. CONCLUSION AND FUTURE WORK

We have demonstrated an approach that allows election processes to be carefully defined and then evaluated with respect to specific correctness properties. The Little-JIL process definition language can be used to develop precise, rigorous process definitions of election processes, and the PROPEL tool supports the development of precise lower-level properties. Once the events of the PROPEL properties are bound to step states in the Little-JIL process definition, the FLAVERS system can be used to perform finite-state verification that attempts to determine if all traces through the process model are consistent with the properties. In the case where the process model violates a property, FLAVERS provides a textual counterexample, which traces a sequence of steps in the process model that may lead to the property violation. Such traces enable the process model developer to reevaluate the model and make improvements to avoid the property violation. To ensure that the improvement addresses the original error, but also that no additional errors are introduced, the corrected process definition and its respective properties are verified once more using finite-state verification.

Finite-state verification is commonly used to verify the correctness of hardware and software and we have begun to explore how this approach can be applied to verify election processes. Although this paper concentrated on only one aspect of the voting process, the approach is extendable to analysis of global properties and more complex processes. This approach allows a *holistic* view of the process, and is not restricted to concentrating on one specific part of an election, such as the part that deals with the use of the DRE.

Our approach also facilitates process model improvement by providing a specific counterexample when a property is violated and enables the improved process model to be verified again. In some cases, this reverification can be accomplished without the need to redefine the properties or the bindings between the properties and the process model step states (unless the process model changes mandate that the properties and bindings be changed). This approach is also easily extendable by allowing further elaboration of the process definition and properties to lower levels of abstraction. Indeed, as our approach involves the application of a verifier to an appropriately annotated graph, and a sufficiently well-defined property, it is applicable whenever such an annotated graph and property can be constructed. Such graphs can be built from source code such as is found in DREs. Thus, this approach can be extended to analyze and verify the source code contained in DREs against properties derived by decomposition of the properties of an overall election process. Thus, our approach, which begins by verifying properties of a holistic election process might then devolve into the verification of whether or not the actual code in a DRE supports the fulfillment of the desired properties of the holistic election process.

We have identified many avenues of future work in defining and analyzing election processes. Currently, when a property is violated, FLAVERS produces a textual trace of the counterexample. It seems beneficial to develop a visual trace representation also to assist the process model developer in identifying errors in the model. Although finite-state verification provides a useful framework for analyzing election processes and ensuring they comply with rigorously specified properties, there are other methods that can be used for verification. One of these methods that we plan to explore is fault tree analysis (FTA). Fault tree analysis considers how the incorrect processing of a step can cause an error and, thus, does not require pre-defined properties. FTA can be very useful in identifying single points of failure (i.e. single steps in the process model where an incorrect execution may lead to the election being compromised).

In the process of refining the requirements into lower-level properties, we encountered a couple of requirements that, once decomposed into properties, could potentially result in properties that may directly conflict with each other (with respect to a specific process definition). We would like to be able to determine if and when this is the case, and how to deal with such conflicts. For example, to remedy a provisional ballot that is later deemed valid, there must be a mechanism that allows for the retrieval of the ballot that was originally cast by that voter. However, that might conflict with a property that assures that voting is private and no one except the voter can obtain information about how the voter voted.

It also seems useful to be able to organize and leverage properties based on process families [23], collections of different processes that are significantly similar. For example, the privacy property might be associated with all members of an election process family, but might be represented differently depending on the specific mechanism used for storing ballots. We would also like to specify process desiderata that most election processes should meet (e.g. that they are resistant to certain types of frauds and collusion) and reason about the relationships among these desiderata.

There has been an abundance of work on process definition and analysis in other domains [6, 20, 8, 13, 12, 25]. In this paper, we demonstrate that rigorous scientific reasoning can be applied to the area of elections as well and described some technology for supporting this approach. Election processes are too important to democracy to not be carefully defined and evaluated.

7. ACKNOWLEDGEMENTS

The authors thank Bin Chen for his help with the verification and Alexander Wise for his help with Little-JIL. This research was supported in part by the U.S. National Mediation Board, and in part by the U.S. National Science Foundation under Grant Nos. IIS-0705772, IIS-0429297 and CCF-0427071. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied of the U.S. National Mediation Board, or the U.S. National Science Foundation, or the U.S. Government.

8. REFERENCES

[1] *ACCURATE A Center for Correct, Usable, Reliable, Auditable, and Transparent Elections.* http://accurate-voting.org/.

[2] *Brennan Center for Justice at NYU School of Law.* http://www.brennancenter.org/.

[3] *Caltech/MIT Voting Technology Project.* http://www.vote.caltech.edu/.

[4] *Election Assessment Hearing (June 29th, 2005).* http://www.electionassessment.org/.

[5] *Verified Voting Foundation.* http://www.verifiedvoting.org/.

[6] I. Altintas, C. Berkeley, E. Jaeger, M. Jones, B. Ludäscher, and S. Mock. Kepler: An extensible system for design and execution of scientific workflows. In *Proceedings of the 16th International Conference on Scientific and Statistical Database Management*, pages 423–424, 2004.

[7] J. L. Brunner. Project EVEREST: Evaluation & Validation of Election-Related Equipment, Standards & Testing Report of Findings. Technical report, Ohio Secretary of State Office, 2007.

[8] L. Clarke, Y. Chen, G. Avrunin, B. Chen, R. Cobleigh, K. Frederick, E. Henneman, and L. Osterweil. Process programming to support medical safety: A case study on blood transfusion. In *Proceedings of the Software Process Workshop*, pages 347–359. Springer-Verlag, 2005.

[9] J. C. Corbett and G. S. Avrunin. Using integer programming to verify general safety and liveness properties. *Formal Methods in System Design: An International Journal*, 6(1):97–123, January 1995.

[10] M. Dwyer, G. Avrunin, and J. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering*, pages 411–420, 1999.

[11] M. B. Dwyer, L. A. Clarke, J. M. Cobleigh, and G. Naumovich. Flow analysis for verifying properties of concurrent software systems. *ACM Trans. Softw. Eng. Methodol.*, 13(4):359–430, 2004.

[12] D. Georgakopoulos, M. F. Hornick, and A. P. Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and Parallel Databases*, 3(2):119–153, 1995.

[13] E. Henneman, R. Cobleigh, K. Frederick, E. Katz-Basset, G. Avrunin, L. Clarke, J. Osterweil, C. Andrzejewski, K. Merrigan, and P. Henneman. Increasing patient safety and efficiency in transfusion therapy using formal process definitions. Technical report, University of Massachusetts, Amherst, 2006.

[14] G. J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[15] D. Jones. *Evaluation of Voting Technologies*, pages 3–16. Advances in Information Security. Kluwer Academic Publishers, 2003.

[16] C. Lambrinoudakis, D. A. Gritzalis, V. Tsoumas, M. Karyda, and S. Ikonomopoulos. *Secure Electronic Voting: The Current Landscape*, pages 101–122. Advances in Information Security. Kluwer Academic Publishers, 2003.

[17] J. Magee and J. Kramer. *Concurrency: State Models and Java Programs.* Wiley, 2006.

[18] R. Mercuri and P. Neumann. *Verification for Electronic Balloting Systems*, pages 31–42. Advances in Information Security. Kluwer Academic Publishers, 2003.

[19] L. Mitrou, D. A. Gritzalis, S. Katsikas, and G. Quirchmayr. *Electronic Voting: Constitutional and legal requirements, and their technical implications*, pages 43–60. Advances in Information Security. Kluwer Academic Publishers, 2003.

[20] L. Osterweil, A. Wise, L. Clarke, A. Ellison, J. Hadley, E. Boose, and D. Foster. Process technology to facilitate the conduct of science. In *Proceedings of the Software Process Workshop*, pages 403–415. Springer-Verlag, 2005.

[21] M. S. Raunak, B. Chen, A. Elssamadisy, L. A. Clarke, and L. J. Osterweil. Definition and analysis of election processes. In *Proceedings of the Software Process Workshop*, volume 3966 of *LNCS*, pages 178–185. Springer-Verlag, 2006.

[22] R. Saltman. Public confidence and auditability in voting systems. In D. A. Gritzalis, editor, *Secure Electronic Voting*, Advances in Information Security. Kluwer Academic Publishers, 2003.

[23] B. I. Simidchieva, L. A. Clarke, and L. J. Osterweil. Representing process variation with a process family. In Q. Wang, D. Pfahl, and D. M. Raffo, editors, *Software Process Dynamics and Agility: Proceedings of the International Conference on Software Process*, volume 4470 of *LNCS*, pages 109–120. Springer, 2007.

[24] R. L. Smith, G. S. Avrunin, L. A. Clarke, and L. J. Osterweil. Propel: an approach supporting property elucidation. In *Proceedings of the 24th International Conference on Software Engineering*, pages 11–21, 2002.

[25] O. Weigert. *Business Process Modeling and Workflow Definition with UML.* 1998.

[26] A. Wise. Little-JIL 1.5 Language Report. Technical report, Department of Computer Science, University of Massachusetts, Amherst, MA, 2006.