August 2015

Variation in Human-Intensive Systems: a Conceptual Framework for Characterizing, Modeling, and Analyzing Families of Systems

Borislava I. Simidchieva
University of Massachusetts Amherst

Recommended Citation

Simidchieva, Borislava I., "Variation in Human-Intensive Systems: a Conceptual Framework for Characterizing, Modeling, and Analyzing Families of Systems" (2015). *Doctoral Dissertations*. 405.
https://scholarworks.umass.edu/dissertations_2/405

VARIATION IN HUMAN-INTENSIVE SYSTEMS: A CONCEPTUAL FRAMEWORK FOR CHARACTERIZING, MODELING, AND ANALYZING FAMILIES OF SYSTEMS

A Dissertation Presented

by

BORISLAVA I. SIMIDCHIEVA

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

May 2015

School of Computer Science

VARIATION IN HUMAN-INTENSIVE SYSTEMS: A CONCEPTUAL FRAMEWORK FOR CHARACTERIZING, MODELING, AND ANALYZING FAMILIES OF SYSTEMS

A Dissertation Presented

by

BORISLAVA I. SIMIDCHIEVA

Approved as to style and content by:

____

Leon J. Osterweil, Chair

____

George S. Avrunin, Member

____

Lori A. Clarke, Member

____

Jane E. Fountain, Member

____

Lori A. Clarke, Chair
School of Computer Science

DEDICATION

За мама, тате и кака.

ACKNOWLEDGMENTS

First and foremost, thanks go out to my adviser, Professor Leon Osterweil, and fellow members of my thesis committee, Professor Lori Clarke of Computer Science, Professor George Avrunin of Mathematics, and Professor Jane Fountain of Political Science. Professor Osterweil has worked hard to shape and direct the framework presented in this thesis with me for the past few years and I have grown tremendously as a researcher and scientist thanks to him. Professors Clarke and Avrunin, co-directors of the Laboratory for Advanced Software Engineering Research (LASER) with Dr. Osterweil have both heavily influenced my work throughout my graduate career and their help and feedback has been very valuable. Professor Fountain brought an important external perspective on this work, evaluating it through the lens of all her expertise in policy and digital government, and contributing new connections to existing problems that have informed and improved this thesis.

I would be remiss not to acknowledge that there are many other mentors who have helped me throughout my graduate career. Professors Rick Adrion, Emery Berger, and Brian Levine of Computer Science have all provided valuable career advice and have helped with navigating difficult situations. External collaborators whom I met through my work at LASER, such as NBM Chief of Staff Daniel Rainey, Dr. Sean Peisert of the Lawrence Berkeley National Laboratory, and Professor Matthew Bishop of UC Davis, are professionals whose feedback I value and seek. Professor Bettina Bair of Ohio State University was my first "official" mentor, and has been a confidant and adviser for many years now. Although I only see her occasionally at professional meetings, she has been more helpful than she will ever know.

I spent the summer of 2010 in magical Ireland, in the city of Limerick, at Lero - the Irish Software Engineering Research Centre. Although brief, the visit was a profound learning experience and it has helped to shape my research interests and the directions this thesis pursued. Special thanks to Drs. Bashar Nuseibeh and Goetz Botterweck for their guidance and many hours of stimulating conversations, and to all the lifelong friends I made at Lero.

Thanks also to all the other great people who have helped along the way—LASERites past and present, especially (soon-to-be) Dr. Heather Conboy, Alexander Wise of Click-Share, and Professor Barbara Lerner of Mount Holyoke College. Thanks to the scientists who took the time to talk shop with me after a technical presentation of theirs a few years ago and with whom I have the privilege to work today—including Dr. Joseph Loyall and Mr. Michael Atighetchi of BBN Technologies, among others.

I will be forever grateful to the wonderful and dedicated people who run the School of Computer Science behind the scenes. They make the lives of graduate students easier on a daily basis, and we all really appreciate it. Ms. Barbara Sutherland is the metaphorical ray of sunshine on a rainy day, and I have never met anyone else more helpful. Ms. Leeanne Leclerc is graduate program manager extraordinaire who is fantastic at liaising with the university and solving problems on behalf of students. Thanks also to Ms. Deborah Bergeron, the LASER Admin, and to the lovely folks at the CSCF, including Mr. Paul Sihvonen-Binder and Mr. Andrew Berkvist. Special thank you to Ms. Jean Joyce, editor of the School's newsletter, *Significant Bits*, which I had the pleasure to help with for a few years.

On a personal note, I feel very fortunate to have met and befriended many lovely people who enrich my life and encourage me to be better on a daily basis. Stefan is one of my closest friends and a trusted confidant and for a couple of years was the closest thing to family on this side of either ocean–thank you for always being there. Dirk seems to always understand what I am thinking without me having to explain. Gal, Megan, and Tim are the Holy Trinity of supportive friends. I miss spending time with Brendan and Laura dearly and look forward to visiting them soon. Scott and Rachael regularly restore my fading faith in humanity. Bruno and George taught me to embrace my inner Spock[1]. Maria, Katie, Alex, and Jamie are the best girlfriends a girl could wish for. Also thanks to Marc, Ilene, Huong, Laura S., Katerina, Jackie and Henry, Laura D., Francesca, Yoonheui, Alan, and all the people I inevitably forgot to mention.

It has only been a couple of years since Jan became a part of my life, but it feels like forever (in a good way). Thank you, boo, for your unconditional support, for being my anchor and my partner in life and silliness, and for being the wonderful person that you are. Last but most important, thanks to my amazing family to whom this thesis is dedicated. Another bunch of people more kind and supportive you will not find. I love you and would not be the person I am today had it not been for your love, encouragement, and help.

[1] Tip of the hat to Mr. Leonard Nimoy, may he rest in peace.

ABSTRACT

VARIATION IN HUMAN-INTENSIVE SYSTEMS: A CONCEPTUAL FRAMEWORK FOR CHARACTERIZING, MODELING, AND ANALYZING FAMILIES OF SYSTEMS

MAY 2015

BORISLAVA I. SIMIDCHIEVA

B.S., Computer Science, STATE UNIVERSITY OF NEW YORK, COLLEGE AT BROCKPORT

B.S., Computational Science, STATE UNIVERSITY OF NEW YORK, COLLEGE AT BROCKPORT

M.S., Computer Science, UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Leon J. Osterweil

A system model—namely a formal definition of the coordination of people, hardware devices, and software components performing activities, using resources and artifacts, and producing various outputs—can aid understanding of the real-world system it models. Complex real-world systems, however, exhibit considerable amounts of variation that can be difficult or impossible to represent within a single model. This dissertation evaluates the hypothesis that the careful characterization and representation of system variation can aid in the generation and analysis of concrete system instances related to one another in specified ways and manifesting different kinds of variation.

When a set of closely related systems can be characterized by a compelling set-membership criterion, it is often useful and appropriate to characterize the set as a *family* of systems. In this dissertation, a variety of system variation requirements and corresponding needs for family specification criteria are identified. We focus on two specific kinds of variation, namely functional and agent variation, and suggest an approach for meeting these needs both at the level of requirements specification (problem-level variation), as well as at the level of implementation specification (solution-level variation).

We present a framework for generating and analyzing new system instances, using the Little-JIL process definition language as an experimental vehicle to study what process definition language capabilities are necessary to support the explicit modeling of variation at the solution level, and thereby to address needs at the problem level. We define a formal notation for specifying functional and agent variation in human-intensive processes and describe a prototype system to accommodate this specification within an existing modeling framework. Once a family of systems is formally defined and characterized at the solution level, different analysis techniques can be applied to make assurances that all members of the family share certain kinds of properties. These analysis results can then be used to inform variation needs at the problem level.

To evaluate the applicability of the approach, we study and model the variation observed in two real-world, human-intensive systems from the domains of conflict resolution and elections. Both case study domains have been observed to employ functional variants of their processes, and, given their complex coordination of human and software agents, both domains require agent variation, therefore fostering a fruitful application of our approach.

CONTENTS

Page

CHAPTER

APPENDICES

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

This dissertation examines variation in families of systems. For the purpose of the research questions and challenges presented in this dissertation, we define a "system," as a collection of processes that orchestrate the complex coordination of activities performed by human and automated agents who use some collection of resources and use, modify, delete, or create some collection of artifacts. Specifically, the kinds of systems that are of interest are complex, real-world, distributed, human-intensive systems. A human-intensive system (HIS) is a collection of processes where humans are responsible for completing critical tasks, such as handling exceptional situations, and making sophisticated decisions throughout the process. Each of this processes can in turn be thought of as a human-intensive system, and we tend to use the terms system and process interchangeably in this thesis. This explicit inclusion of humans within the system boundary introduces additional levels of variation not usually manifested in traditional software systems. These new kinds of variation call for novel techniques for modeling variation and reasoning about it. In fact, our previous research has demonstrated that models of real-world processes have requirements for variation that are similar, but indeed may be still more diverse and challenging than those in software systems [67–70]. This similarity indicates that solutions to process variation problems may draw useful inspiration from solutions that are effective for system variation, but that meeting the need for variation in processes is likely to also require some new approaches.

A successful system- or software-development project rarely aims to produce a single system or a single piece of software. If the project is successful, then its resulting software product or service will provide a capability that will need to run in many different environments, is likely to present itself to different types of users in different ways, may need to run faster for some users, and have different sets of features for different kinds of users. In such cases it can be very ungainly to respond to differing requirements by building a single system or piece of software that is capable of meeting all of these needs. Instead, these needs are typically met by building different *variants* of the system or software, with each of the variants being developed to meet different combinations of these needs. Variants may differ from one another in the functionality they provide, their speed, robustness, or in any of a number of other aspects; this difference between variants is called *variation*.

Note that variation may occur in a single or along multiple dimensions, including the ways in which activities within and between processes are coordinated, the hardware and software subsystems and components that are used, or the choices of agents and services that are made. Despite increasingly complex and demanding requirements that lead to the creation of ever larger sets of such variants, it is important that the different variants retain well-understood relations to each other. If such well-understood relations exist, the variants, comprising what we refer to as a *family*, can be expected to share certain potentially useful properties. We will refer to families of variants of software artifacts as *software families*, families of variants of processes as *process families*, and families of variants of systems as *system families*. If variants share some high-level architectural or process specification, we call this specification a *common core*. The common core contains one or more *variation points*. A variation point is only abstractly specified in the common core and it is left up to different variants to provide detailed implementations. These different ways in which variants can then extend the variation points within common core are referred to as *abstractions* or *elaborations* once they have been instantiated.

A certain amount of variation exists in virtually all real-world systems and processes. Such systems and processes either 1) incorporate certain variation explicitly, or 2) anticipate acceptable variability through abstracting out the details of a part that may be expected to vary, or 3) provide mechanisms for the exclusion of certain variants typically in an attempt to assure that permitted variants all share certain desirable properties. For example, in the election domain, which is the domain of some of the evaluative case studies to be explored in this dissertation research, many variations have been observed among processes from different jurisdictions within the same state, or even within the same jurisdiction, based on context and circumstances. The research approach presented in this dissertation aims to support the ability to specify process variation (and resulting process families) sufficiently precisely so that it is possible to determine whether all family members conform to certain pre-specified properties, or, if not, to help determine what family members may violate the property. Doing this requires a model that faithfully represents the real world by explicitly representing existing variation, even across multiple dimensions, in ways that are accurate and amenable to reasoning. This usually requires elevating variation to be a first-class construct within the representation being used. A key goal of this research is to investigate approaches that are effective in supporting the modeling of such variation and the families thereby defined. Moreover, we seek to identify analysis techniques that could be used to verify that all members of suitably defined process families must necessarily adhere to certain kind of properties in dimensions such as functionality, security, and safety.

We suggest that semantic relations can define what variants are members of a system family (i.e., relations that define membership constraints and criteria), and, if that is the case, that it may be possible to then generate new variants as transformations of existing variants. Moreover, if such a transformation exists, the generation activity may be amenable to constraints that would assure that any newly created variant must have certain desirable properties by construction. Once the relations among variants are well defined, then it may be possible to go a step further and make assurances about the entire collection of variants and whether the family itself and all its variants conform to certain properties. In the case of elections, for example, one property that every variant in the family would have to satisfy is that every eligible voter is allowed to cast no more than one vote.

A key goal of this dissertation is to demonstrate that there are numerous kinds of variation relations that might be helpful in supporting such reasoning, and to indicate that different kinds of variation relations lead to families whose members share different properties. As suggested above, some kinds of variation relations define families whose members differ only in performance. Other kinds of variation define families whose members offer functionality that differs only in relatively small, well-circumscribed ways. Still other kinds of variation relations define families whose members all achieve similar, or identical, goals, but do so in very different ways. In earlier work [70], we suggested that there are different canonical approaches that are useful in meeting such diverse requirements for variation. In the research presented here we will explore ways to assure that, while the members of a process family may differ in a variety of different ways, they may also share important similarities. One way to evaluate similarity is to ensure that they all satisfy certain properties along dimensions of interest such as safety and security. Thus this research explores two main research goals:

- RG1: *Generation*—how can variation be characterized in a way that facilitates the creation of new family members at the solution level that best satisfy a set of given variation needs at the problem level, and also meet the membership criteria of the family with respect to the dimensions of variation it aims to model? and

- RG2: *Analysis*—how can the family representation be leveraged to facilitate the application of different reasoning techniques to the entire family, for example to prove that all variants have desired properties? How can the generation activity be constrained to ascertain that newly created variants satisfy certain properties by construction?

Specifically, we hypothesize that the careful definition, characterization, and categorization of different variation relationships would shine light on such questions and facilitate achieving both of these research goals. For generation, the formal definition of either a shared common core or a set of properties that all members of a family must satisfy may facilitate different generation approaches. For analysis, the relationship among variants may determine what analysis approaches the family is amenable to, or kinds of assertions can be proven about current or future variants in the family. A particular approach to creating a family might facilitate the fulfillment of only one of these research goals. Thus, for example, some approaches may make it easy to generate a new family member but may impede analysis. Conversely, a family might be defined in such a way that reasoning about certain properties of the family as a whole is easy, but generation of future members is limited to variants that are known to be "safe," which may impose onerous restrictions on the size and nature of the family. We believe that there is considerable value in identifying different approaches to generating families, and to understanding which types of variation requirements they meet under what circumstances, what kinds of reasoning they support or facilitate, and what advantages they offer.

The rest of this dissertation is organized as follows. Chapter 2 describes the conceptual framework for variation modeling and management at two levels of abstraction. Chapter 3 contains two motivating case studies, one on variation within a process guidance example in negotiations, and the other on variation within security and privacy requirements in elections. Chapter 4 presents the technical framework used in supporting the application and evaluation of the approach. Chapter 5 details the results of applying our approach and methodology to some real-world problems, including an extensive case study. These results are discussed in Chapter 6. In Chapter 7, we discuss the limitations of the approach and propose some possible mitigation strategies. Chapter 8 then provides an overview of related work and Chapter 9 outlines some promising avenues for future research.

CHAPTER 2
CONCEPTUAL FRAMEWORK

2.1 Problem- vs solution-level variation dichotomy

To aid understanding, management, and implementation, variation can be viewed from two perspectives, namely a problem or requirements perspective akin to domain engineering in software product line engineering (SPLE), and a solution or system perspective similar to application engineering in SPLE [68]. From the problem perspective, as in the case of software families, process and system families often must meet requirements for variation in functionality, variation in performance, variation in the platform on which they must run, variation in security assurance, and variation in robustness. In addition, however, complex systems and their processes often must also satisfy requirements for variation in the types of services that they will employ, the modes of interactions with the humans participating in the system or process, and the strategies that they will employ in using needed resources [70]. These variation requirements may overlap, and may even be inconsistent.

To begin addressing how these requirements might be met, we need to consider them from the solution perspective. Note that as these are variation requirements, they should be technology-independent and focus on defining the needs for variation, not the details of how that variation can be implemented. The solution perspective focuses on studying how specific approaches to achieving variation can help to generate new variants and to support analyses that apply to all variants in a family. Because the solution perspective begins to discuss specific approaches, it inevitably leads to specific technologies as well. However, this variation framework is designed to be widely applicable irrespective of the technology used. For demonstration purposes, a certain representation will be used to present solution-level variation in this dissertation, but the approach aims to be representation-agnostic. A key element of this approach is the proposition that a problem-level variation metamodel can be used to connect needs for different kinds of requirements variation to appropriate solution-level family implementation approaches. A brief, non-exhaustive classification of some of the kinds of problem-level variation we have observed in the two case studies of negotiation processes and election processes follows.

- *Functional Variation:* Variants differ in the details of one or more of the different functional capabilities specified.

- *Functional Invariance:* Variants are equivalent (for a pre-specified equivalence function) in their functionality, but their underlying implementations differ.

- *Goal Invariance:* Variants achieve the same goal (for a pre-specified goal condition, e.g., produce a specific set of artifacts), but the process to achieve the goal varies among variants.

- *Robustness Variation:* Variants differ in the ways in which they are able to recover from incorrect or abusive use.

- *Performance Variation:* Variants all provide the same functionality, but differ in the speed with which they execute.

- *Interaction-Based Variation:* Variants provide identical functionality but interact with users in different ways.

- *Service*[1] *Variation:* Variants differ from each other in the services or agents they utilize to provide different functional capabilities.

[1] A direct parallel can be drawn between agents in a process and service providers in a system built using Service-Oriented Architecture (SOA) principles so we use the terms *agent* variation and *service* variation interchangeably. However, human agents introduce additional complexities and existing SOA and SPLE techniques must be carefully adapted and extended to accommodate such differences.

- *Security Variation:* Variants differ in the system security they provide.
- *Privacy Variation:* Variants differ in the level of privacy protection they accomplish.

This dissertation will focus on exploring two of these problem-level variation dimensions, namely functional variation and agent variation, and follow their realization at the solution level. Functional variation has been extensively modeled and studied in the software domain and is in fact what most software product line approaches focus on. Functional variation is frequently achieved through the composition of different *features* that implement variation in the low-level behavior (i.e. in the details of the behavior) of the software being generated. Functional variation has also been studied in the context of processes and workflows, though not quite as extensively. Agent variation, on the other hand, has not been thoroughly studied but is of pivotal importance when it comes to human-intensive systems, such as negotiations and elections. We hope to be able to apply some of the techniques that have been found to be particularly useful for feature modeling and analysis in software to processes and systems and to use this experience to inform the problem of modeling and analyzing process and system families exhibiting agent variation.

Note that in addition to the dimensions listed above, there are other ways to define relationships that may be the bases for process families. Some of these relationships may indeed be more clearly defined by specifying what stays constant among variants within a family, as opposed to what varies. Such invariance could be ascertained through the verification of all variants in the family against a set of carefully predetermined properties. A couple of such *invariance* relationships that seem to form the basis for specifying useful families include functional invariance and security invariance. In the case of functional invariance, variants provide exactly the same functionality, but this functionality is implemented in different ways. Membership in a security invariance family on the other hand can be determined through the verification of all family members against a set of security properties. Newly created variants can then be verified against these properties to decide if they can be safely added to the family, or their generation may be constrained so that they are security-invariant by construction (for example by not including any events that may result in the violation of the properties).

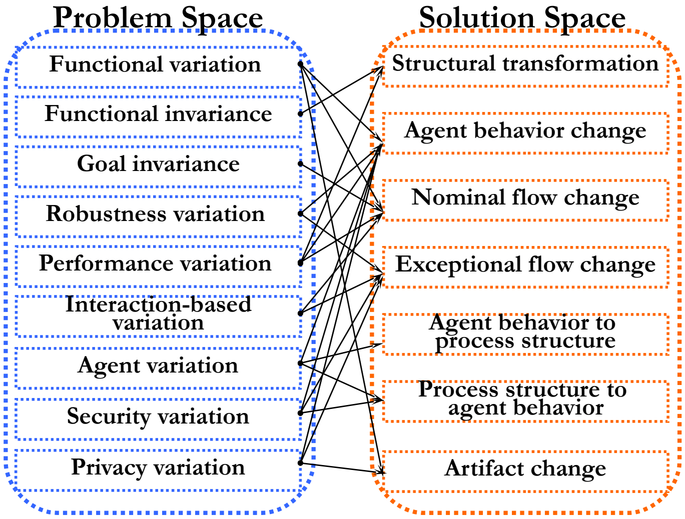


Figure 2.1. Example mappings from the abstract problem space to a solution space in a process definition language.

In order to represent processes faithfully, a process modeling notation with rich semantics is necessary. Such a notation must be well-defined and broad enough to allow for the precise specification of the many different aspects of processes that may vary. The notation must support the specification of both nominal and exceptional control flow as well as provide capabilities for rigorous data flow specification. Given the necessity for abstraction to support the specification of architecture-level common core, a notation with good provisions for abstraction specification and hierarchical dependencies is paramount. Since this dissertation places a heavy focus on human-intensive systems, support for agent assignments that allow for both software components and human actors would greatly facilitate modeling and reasoning about service variation. Lastly, most existing representations and notations do not support variation natively, so the ability to easily extend an existing notation would be helpful. Extending a notation to elevate variation to a first-class construct becomes easier when a good separation of concerns along the different dimensions of variation is already part of the language specification. Providing initial support for creating families of process variants, which cover many commonly used workflow constructs, should lead to useful insights about supporting variation in different process definition languages and other system representations as well. Some example representations that could be pursued include service-oriented architecture (SOA) systems, component-based distributed systems, and other human-intensive systems, such as mobile deployed applications, process-guidance applications, etc.

Figure 2.1 illustrates the dichotomy between problem-level and solution-level variation discussed in this section. Within the solution space on the right, process-specific techniques are identified that may address one or more of the variation needs identified in the problem space on the left. We assume a representation with a good separation of concerns between the comprising components of its process definitions, namely the coordination of activities, the services or agents carrying out these activities, different security policies, the input and output artifacts of procedures or components, and so on. Given such a representation, some of these dimensions of solution variation may be amenable to orthogonal changes. For example, changes in the coordination specification may be specified independently and then combined with variation based on changes in the agent behaviors.

Each problem-level variation dimension may be met by one or more solution-level variation implementation technique. For example, consider an election process requirement for functional variation based on jurisdiction, where casting a ballot may entail different steps, involve different technology, and necessitate different levels of assistance from an election official. The different variants in this functional variation family correspond to different elaborations of a ballot-casting activity that are invoked in specific higher-level contexts within the election process. Such different elaborations of the ballot-casting subprocess entail differences in the nominal flow of the overall election process definition.

Hence in Figure 2.1 we connect the *Functional variation* dimension in the problem space to the *Nominal flow change* dimension in the solution space.

In fact, nominal flow changes at the solution level, and specifically elaborations of certain steps with different details, seem to have the potential to address both functional and agent problem-level variation requirements quite well and may potentially provide partial support for some other problem-level variation dimensions. Therefore, elaboration will be the initial kind of solution-level variation explored in this dissertation.

2.2 Formalizing Variation

Given this dichotomy in variation, between problem-level variation driven by requirements, and solution-level variation driven by implementation choices, our conceptual framework accordingly also necessitates the specification of variation at both levels.

2.2.1 Specifying problem-level families

At the problem-level, we can specify families at a high level, by defining a characterization that applies to all members. Two simple approaches to doing this are to specify the family by induction (i.e. determine a starting process and build members by adding onto this process, which effectively defines the family by construction), or to specify the family according to certain properties that all of its members must meet (thus effectively defining membership criteria that can be applied to newly generated variants retroactively to determine if they are members of the family or not).

To build families by induction, one can specify an initial common process definition core C, and a set of elaborators, E, and define a family Φ to be all process definitions that can be generated by a sequence of applications of elaborators from E to C initially, and inductively to any variant generated by applying a sequence of elaborators to C. E.g., let C be a single high-level process definition, and E be the replacement of any unelaborated step in C by any elaborating subprocess. Such a family may contain a variant in which a core function f_A, is decomposed into a set of subfunctions, corresponding perhaps to subprocesses, say f_{A1}, f_{A2}, and f_{A3}, whose composed capabilities, when transformed by some other function g, achieve the required functionality of f_A (i.e. $f_A \rightarrow f_A = g * (f_{A1}, f_{A2}, f_{A3})$. Although such a process family can easily be generated if the function decomposition can be identified, it is not clear what analysis approaches could readily be used to determine whether all members of this family satisfy some given properties. Depending on C and E, it may indeed be possible to construct variants that violate the well-formedness rules of the language in which C was written. This difficulty could be addressed by restricting family membership only to those generated processes that are well-formed. But adding well-formedness constraint checking to this approach complicates variant generation, although it can facilitate analyses whose results are applicable to all variants.

A way to address this tension is to consider using induction to initially generate process family members, but also applying the idea of a set of membership criteria as described above, possibly in the form of conservative comparisons for trace equivalence, to decide when generated variants are to be allowed to become family members. If it is possible to specify the behavior we want to enforce with very few events of interest, and define a regular language for the acceptable sequences of these events, we can build a finite-state automaton (FSA), T, to recognize this language (perhaps using a tool to facilitate this process such as PROPEL [74]). We could further annotate each family member's possible executions with these events for example by using an annotated control flow graph, or translating it to a more suitable representation, such as a trace flow graph (TFG). A TFG, as employed in FLAVERS [31], leaves out parts of the CFG that do not contain events of interest or change the control flow and includes consideration for different interleavings of events of interest. We could then consider a functional variation process family Φ to be a collection of process variants p_i, and we could determine whether they are all trace-equivalent with respect to the property, T, if they all satisfy the property (meaning that there does not exist a trace through any one variant's TFG that leaves the FSA T in a non-accepting state at the end of that trace). By this definition, for example, all variants that can never cause an event included in T will be considered trace-equivalent. Similarly, all members of a family defined as trace-equivalent will have the same attribute of adherence (or if not, the same sequence of violation) to T. I.e. $\forall \{p_i, p_j \mid i \neq j\} \in \Phi, p_i \approxeq_T p_j$ ($\approxeq_T$ denotes trace equivalence with respect to T). This is so because under this definition only events pertinent to the property are considered in determining trace equivalence, and thus two variants must be trace-equivalent if and only if none of their elaborations contains events of interest with respect to the property, or the order of events coincides for both variants over all traces. More complex rules for rapidly determining family membership could perhaps be developed for cases where elaborations incorporate the possibility of certain kinds of event sequences that demonstrably cannot lead to differences in adherence to T.

2.2.2 Specifying solution-level families

In order to generate and analyze solution-level families, a specific representation must be used. We focus on the Little-JIL process definition language as a case study because Little-JIL is a good example of a process modeling notation that meets the requirements outlined earlier. Little-JIL already supports a considerable amount of variation within a single process definition, and it allows for the specification of complex coordination and concurrency, abstraction, resources, and agents, among other features [86]. However, we have found that attempting to model too much variation in a single Little-JIL process definition leads to process definitions that can be turgid and worryingly complex, especially in cases where one choice early in the process dictates choices later in the process. A single process definition without consideration for modeling these choices as variants and attempting to capture their interactions may fail to communicate this trend. Investigation of other process definition languages suggests that this difficulty is not unique to Little-JIL, and that, moreover, it may be a problem common to all attempts to represent broad process families with a single process definition, regardless of the process language used. Accordingly, we explore the use of these Little-JIL language features as the basis for creating process families that provide broader amounts of variation that can readily be provided by a single process, and can do so more transparently and simply.

Note that, while our expectation is that the outlined methodology can be applicable in the abstract to many different process specification languages (perhaps with minor modifications necessary to accommodate specific differences in semantics) we expect to gain important specificity of results by restricting our attention to one specific language. Little-JIL exhibits many of the desirable properties that a process definition language may need, such as rigor, rich semantics, and considerable flexibility, and so it seems like an appropriate choice for us to use as our example language. Our preliminary investigation had suggested that the language's semantics would have to be expanded and amended to accommodate the specification of variation within a process as a first-class construct. Accordingly, the language was extended correspondingly as outlined later in Chapter 4.

To allow for easier tracking of variation within a Little-JIL process definition, we devise a formal specification of the elements comprising a Little-JIL process as follows. This formalization can be easily defined for other process and system representations by identifying clear correspondences between language components. For clarity, the notation below uses capitalized names to denote sets and all-lower-case names to indicate set elements.

A Little-JIL process LJP can be specified as a three-tuple comprising a coordination structure, $Coord$, a set of artifacts, $Arts$, and a set of resources, Res:

$LJP = \{Coord, Arts, Res\}$

Each of these elements is further defined as follows:

$Coord = \{Step, Dcmp, root\}$, where

- $root \in Step$ is a handle to the root step of this coordination structure; and $Step, Dcmp$ together define a set of $step$ elements structured by the $Dcmp$ hierarchy relation:

- $Step = \{(path, Attr)\}$, namely a set of $step$ elements where each element is defined by its $path$ (the name of the step it describes, preceded by a complete or partial path[2] that points to a specific appearance[3] of the step within the process definition)[4], and a set of one or more attributes. Note that because of this path specification each $step$ element by default specifies one concrete appearance of a step type within a process definition and, furthermore, we can use this specification to perform operations on the path, such as path concatenation. The attributes are further defined as

  * $Attr = \{(name, value)\}$ (each $attr \in Attr$ is a $name$, $value$ pair). Some expected attributes in the case of Little-JIL include the step name, step kind, the agent responsible for the step's execution, as well as other resources necessary, incoming, outgoing, and local parameters, exception specifications and so on.

- $Dcmp = \{(s_1, s_2), \textbf{nominal}|\textbf{exceptional}\}, s_1, s_2 \in Step$ where $step$ elements are organized in a structure by the ordered binary parent/child relation (i.e. s_1 is the parent of s_2 and the edge connecting them is either blue (a nominal flow edge) or red (an exceptional flow edge)). Note s_1 and s_2 are step appearances according to the definition of $Step$, therefore multiple references to the same step type (i.e. multiple appearances of a step with the same name) can be resolved unambiguously.

$Arts = \{entity\}$ (each member, $entity$, of the artifact set $Arts$ may have a different internal structure, which is abstracted away in this specification. Each $entity$ specified

[2] The paths will be specified using the XPath specification language [3]

[3] Step "appearance" refers to a concrete invocation of a step type within the static coordination specification of the process definition. We refrain from using the term step "instance" because we reserve that term for a specific invocation of a step within the dynamic structure of an executing process.

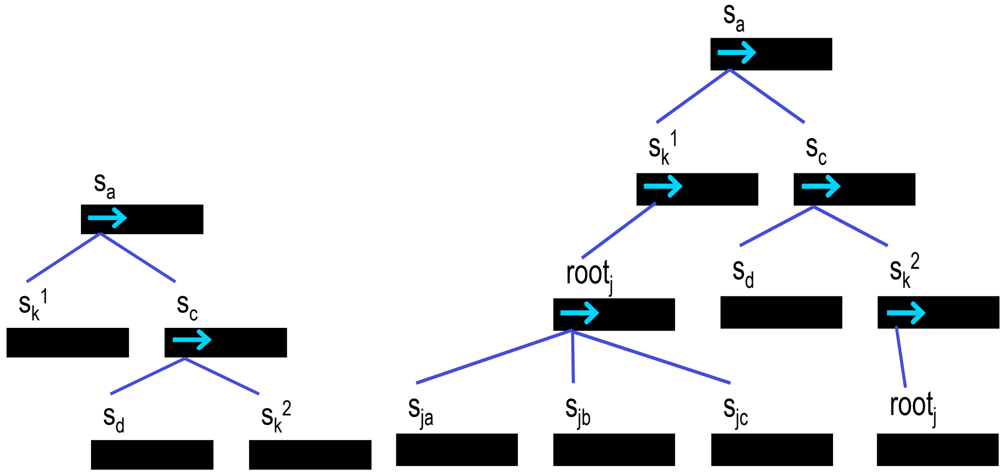
[4] A more complete explanation of the notion of context is given in Chapter 4

is also considered to be an artifact *type*, so the notation does not support differentiating among multiple instances of the same artifact type. This is a direction that may be explored as future work.)

$Res = \{(Caps, Attr)\}$(each resource entity is made out of a set of capabilities and a set of attributes. Note that this description abstracts away many of the features and advantages that the current resource management system in Little-JIL can provide, but suffices for elaboration specifications.) Capabilities are further specified as

- $Caps = \{(step, \{LJP\})\}, \forall LJP_i \in \{LJP\}, |Res_i| = 1$ (each capability is a tuple made out of a step corresponding to a step that some agent is capable of performing (note once again that steps are specified at the appearance level, allowing for different appearances to have different agents bound to them), as well as a set of agent behavior elaborations, each one of which is a separate Little-JIL process definition (because the definition of LJP is general enough to accommodate the specification of a subprocess rooted at a specified step). Note further that unlike in the general definition of a process fragment elaboration where the coordination of several different agents may be required, agent behavior elaboration implies that a single agent is responsible for the execution of all steps within the elaboration. Therefore, to implement this restriction, we specify the size of the resource set of the subprocess, Res, to be one. This distinction is explored in detail in Chapter 4, but for now just note that we can distinguish between an elaboration specifying a process fragment coordinating multiple agents or an elaboration describing a single agent's behavior through restricting the size of the Res set.)

Once a Little-JIL process is defined using this specification, we can start to define different kinds of families based on different elaboration techniques, as briefly described in the agent behavior specification above. Below we present two increasingly complex cases of elaboration, namely elaborating one or multiple (including all) *appearances* of an elaboration step (at a prespecified path within the process definition) with exactly one given elaboration and then extending this definition to allow for elaborating one or multiple (including all) *appearances* of a given elaboration step with more than one predefined elaboration. For example, a functional variation family LJP_{new} where two appearances of the step s_k, s_k^1 and s_k^2, within a process definition LJP_i will be elaborated with a given variant LJP_j is defined as follows:



(a) Core process with two step appearances to be elaborated, s_k^1 and s_k^2.

(b) Core process with the elaboration $root_j$ grafted on.

Figure 2.2. Illustration of $Elab_{mase}$ elaboration scenario.

$LJP_{new} = Elab_{mase}$[5]$(LJP_i, LJP_j, S_k), \{S_k \subset Steps_i \wedge \forall s_k \in S_k, s_k$ is an elaboration step[6]$\} = (Coord_{new}, Arts_{new}, Res_{new})$ where

- $Coord_{new} = \{Step_{new}, Dcmp_{new}, root_i\}$
- $Step_{new} = \{Step_i \cup Step_j\}$

[5] $Mase$ stands for "multiple appearances, single elaboration." Note that a most likely use for this kind of elaboration would be in the case where all appearances of steps having the same name are to be elaborated in the same way.

[6] This can be further formalized as a check on the prespecified attribute that contains the step kind.

- $Dcmp_{new} = \{Dcmp_i \cup Dcmp_j \cup Dcmp_k\}$, where $Dcmp_k = \{\forall s_k \in S_k, (s_k, root_j,$ **nominal**$)\}$ i.e., as depicted in Figure 2.2, the hierarchy for the new process comprises the original process (shown in Figure 2.2(a)) unioned with the elaboration, plus extra added edges between each step appearance to be elaborated and the subprocess describing the elaboration (which is the same for all $s_k \in S_k$ and is rooted at $root_j$), as shown together in Figure 2.2(b). Note that superscript numbers in these figures are used to illustrate multiple appearances of the same step, not different step specifications (subscripts, on the other hand, are part of the step name, which serves as a unique identifier).

- $Arts_{new} = Arts_i \cup Arts_j$

- $Res_{new} = Res_i \cup Res_j$

The elaboration $root_j$ is shown as grafted explicitly (i.e. with its corresponding subprocess definition) under its first appearance in Figure 2.2(b) for simplicity of presentation, but in the underlying Little-JIL representation it is actually grafted on as a separate subprocess and then simply referred to from each eponymous elaboration step. This removes the need from considering the first appearance as a special case, and simplifies the case of elaborating a single step appearance by allowing it to be handled identically.

Finally, we tackle the case where different alternative process variants may be considered as possible elaborations of different step appearances. This kind of elaboration seems to be very useful for studying ways to address issues in RG1: Generation because users would be able to quickly see all the possible elaborations for a certain step, which should help them to identify specific behaviors that could be what is needed in order to generate a needed process variant. Perhaps more important, this kind of elaboration would facilitate RG2: Analysis by allowing for a direct application of different analyses to the different possible variants, which should help to determine what variants satisfy certain properties, what other variants may be unsafe, and in what way. At the high level, in order to specify that any of several different variants, say s_{ia}, s_{ib}, and s_{ic} might be used to elaborate the step s_i, the transformation that is applied is that the elaboration step s_i is associated an intermediate choice[7] step, and that choice step has all variants, in this case s_{ia}, s_{ib}, and s_{ic}, as its children. Effectively, this states that s_i can be performed by performing exactly one of the alternatives s_{ia}, s_{ib}, and s_{ic}, which is exactly the goal. Note that as previously explained, care must be taken to avoid generating large process families with multiple elaborations grafted at each elaboration step to avoid complexity. Large process families may be helpful for performing analysis on the entire family, but this approach also affords the process developer the flexibility to generate single, targeted process instances containing appropriate variants. This type of elaboration can be thought of as being performed in two major steps. First, we essentially generate an augmented LJP_i that has one additional choice step for every step appearance to be elaborated, then all possible variants $LJP_{j1}, LJP_{j2}, \ldots LJP_{jn}$ are grafted as children of every new choice step, s_k^{choice}, thus creating an "elaboration of elaborations."

Using the above notation, this kind of elaboration is specified formally as follows:

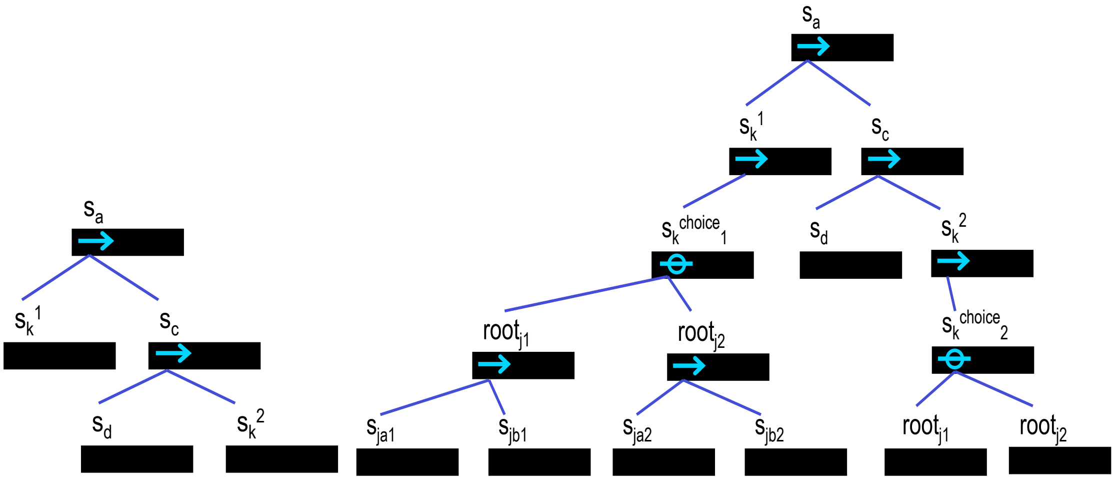
$LJP_{new} = Elab_{mame}{}^8(LJP_i, \{LJP_j\}, S_k), \{S_k \subset Steps_i \wedge \forall s_k \in S_k, s_k$ is an elaboration step$\} = (Coord_{new}, Arts_{new}, Res_{new})$ where

- $Coord_{new} = \{Step_{new}, Dcmp_{new}, root_i\}$
- $Step_{new} = \{\{Step_i \cup Step_j^{\cup} \cup S_k^{choice}\}$, where $Step_j^{\cup} = \forall LJP_j \{\bigcup \{Step_j\}\}$, and a newly-created[9] set S_k^{choice}.

[7] We use a choice step instead of creating a new step sequence paradigm because, conceptually, it accurately represents that only one option will be selected as dictated by Little-JIL semantics. However, there are other means to achieve the same result and they are discussed in Chapter 7.

[8] $Mame$ stands for "multiple appearances, multiple elaborations." Note we are allowing for more than one appearance of a step to have multiple elaborations, but for the case when only one appearance should be elaborated with multiple variants, this definition can be trivially modified by restricting the cardinality of S_k to 1, analogously to how $Elab_{mase}$ would handle a single appearance.

[9] S_k^{choice} is a new set constructed according to the following definition: $S_k^{choice} = \forall s_k \in S_k, \{s_k^{choice} = s_k \otimes (/s_k^{choice}) \wedge s_k^{choice}.\text{kind} = choice\}$.



(a) Core process with two step appearances to be elaborated, s_k^1 and s_k^2.

(b) Core process with the elaborations $root_{j1}$ and $root_{j2}$ grafted onto additionally added choice steps.

Figure 2.3. Illustration of $Elab_{mame}$ elaboration scenario.

I.e., an additional set of steps are added, that are essentially duplicates of all step appearances to be elaborated, but they are choice steps, which will allow for the attachment of multiple elaborations (illustrated in Figure 2.3(b). Moreover, each newly-created step appearance is created with the correct path by concatenating the corresponding step appearance's path with the new extra scope. This will allow for the newly-created choice steps to be attached as children of their corresponding elaboration steps, with the appropriate $Dcmp$ relation.

- $Dcmp_{new} = \{Dcmp_i \cup Dcmp_j^{\cup} \cup Dcmp_k^{choice} \cup Dcmp_k\}$, where $Dcmp_j^{\cup} = \forall LJP_j$ $\{\bigcup\{Dcmp_j\}\}$, $Dcmp_k^{choice} = \forall s_k \in S_k, \{((s_k, s_k^{choice}), \textbf{nominal})\}$ (this adds the new choice steps as children of the original elaboration steps), and finally, $Dcmp_k =$ $\forall s_k^{choice} \in S_k^{choice} \times \forall LJP_j \in \{LJP_j\}(s_k^{choice}, root_j, \textbf{nominal})\}$ i.e. as illustrated in Figure 2.3, the hierarchy for the new process comprises the original process (shown by itself in Figure 2.3(a)), extended with choice steps attached to every step appearance to be elaborated, and in turn, extra edges between every newly-created choice step and all subprocesses describing the elaboration (in this case rooted at $root_{j1}$ and $root_{j2}$, respectively), as shown in Figure 2.3(b). We use Cartesian product of the two sets of choice steps (S_k^{choice}) and elaborations ($\{LJP_j\}$), respectively, to ensure correct enumeration of all possible elaboration scenarios.

- $Arts_{new} = Arts_i \cup Arts_j$
- $Res_{new} = Res_i \cup Res_j$

Using this notation to define processes represented in Little-JIL as well as their potential composition into new families allows for the exploration of many interesting research questions raised within the two domain areas that seem to be approachable in ways that should shed light on issues in our two research goal areas. The next chapter describes the two case study domains in detail and provides motivating examples illustrating situations where the above variation relations have been observed.

2.3 Analysis and Variability Management

Depending on the problem-level variation needs, we expect that some solution-level approaches will better support analysis and variability management than others. This dissertation explores what analysis techniques can be applied to which solution-level variation approaches. It seems that compositional analysis techniques should be effective in supporting the analysis of process families that are built by function elaboration, as all members of such a family share a common core and vary from each other only by different elaborations of specific functionality.

The analysis of families of processes that are functionally invariant, on the other hand, seems more suitable for other kinds of analysis such as limited trace comparisons (only with respect to certain events of interest) as outlined earlier. In that case, all variants within a family can be compared according to a set of predefined important properties that the family must satisfy to determine if the variants are functionally equivalent with respect to the properties that matter. Our research will focus on two kinds of problem-level variation and a solution-level technique to support them, and study the different kinds of analysis that can be applied.

Variability management would also differ based on the variation relation. Different approaches such as variation points can be applied at the problem level to specify variants that elaborate the same functionality differently, or additional annotations, constraints, and obligations to specify variation dimensions that focus more on resources and artifacts rather than the activities within a process definition.

CHAPTER 3

MOTIVATION

This chapter describes variation needs observed in the two case studies featuring two different domains and, more important, two very different sets of requirements. The first case study is in the domain of negotiation and alternative dispute resolution, with a focus on the need for supporting process guidance and how variation can facilitate better training and even allow for the inception of self-adaptive process guidance systems. security and privacy.

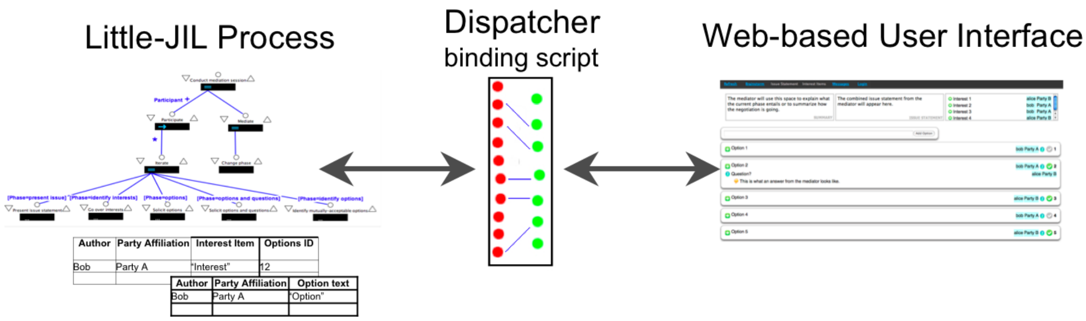
3.1 Case Study I: Variation in the context of process guidance

Needs for variation in negotiation processes were initially observed during the elicitation of an Online Dispute Resolution (ODR) process from multiple mediators at the National Mediation Board (NMB). Although all mediators had been trained to use the same process, it was apparent that there were differences in the way individual mediators handled certain parts of the process that were not clearly specified in the training materials or that required flexibility based on group dynamics, negotiation setting, and other factors. Details of the process were elicited from the mediators as the basis for creating a precise definition of NMB's core ODR process. The process thereby defined was then made the basis for a system called STORM2 that executes the defined process for interest-based negotiation[1] as carried out by the NMB.

[1] Interest-based negotiation is a process based on Interest-Based Bargaining (IBB), but adapted for grievance mediation.

3.1.1 The STORM2 Online Dispute Resolution (ODR) System

The STORM2 ODR system was designed to provide very flexible support for guiding disputing parties towards the resolution of their disputes. Our approach to providing flexibility was based principally on defining dispute resolution as a precisely specified process definition that could be edited to create different ODR systems.



Author	Party Affiliation	Interest Item	Options ID
Bob	Party A	“Interest”	12

Author	Party Affiliation	Option text
Bob	Party A	“Option”

The mediator will use this space to explain what the current phase entails or to summarize how the negotiation is going.

The combined issue statement from the mediator will appear here.

This is what an answer from the mediator looks like.

Figure 3.1. Three major components comprise the STORM2 system.

Figure 3.1 shows a very high-level sketch of the architecture of STORM2. The STORM2 framework consists of a precise and rigorous definition of a negotiation process that guides execution (shown on the left), a user interface (right), and middleware to integrate the two (center). Much of the negotiation process definition focuses on how data artifacts (e.g. issue statements, opinions, and questions offered by participants) are created by some users at some times, and then routed from the users that have created these data artifacts to other users and to data management tools that require them in order to perform their own tasks. STORM2's users (i.e. the disputants and mediator) are asked to submit inputs at specified times, and allowed to submit their replies at other times. STORM2 is responsible for being sure that submitted inputs are routed correctly and in a timely way without violating any restrictions on confidentiality, and without violating agreements about anonymity. The right-hand box in Figure 3.1 represents the screens through which STORM2 users interact with the system. The initial implementation of this interface is based on the Tapestry framework [1], which makes the interface user-accessible online through a web browser without requiring the user to install any software. A new variant of the STORM2 system is currently being created where the interface is based upon the Google Web Toolkit (GWT) [2] framework to provide more flexibility in the user experience without compromising portability. This is an excellent example of interaction-based variation, which is one of the problem-level variation kinds we have identified. Our preliminary assessment is that this kind of problem-variation is not likely to be very well addressed by the solution-level variation approaches that we will investigate initially in this dissertation. We will, however, revisit this assessment as our work proceeds.

The middle box in the diagram represents a Little-JIL middleware system, formerly called the Dispatcher, and currently being improved and replaced by its successor, Janus, named after the two-faced Roman god. This middleware component communicates requests and responses between the executing process on the left and its users through their user interfaces on the right. The Dispatcher and Janus are both written in the Java programming language, and integrate the process definition with the interface by passing messages and commands between them, based on a binding script that provides a mapping between the two. This three-component architecture, consisting of the Little-JIL process definition, the online user interface, and a message-passing intermediary, makes STORM2 very flexible and is intended to allow for easy modifications to explicitly model and accommodate different kinds of variations in the negotiation process. Thus, it seems to be a very promising vehicle for supporting the research we present here.

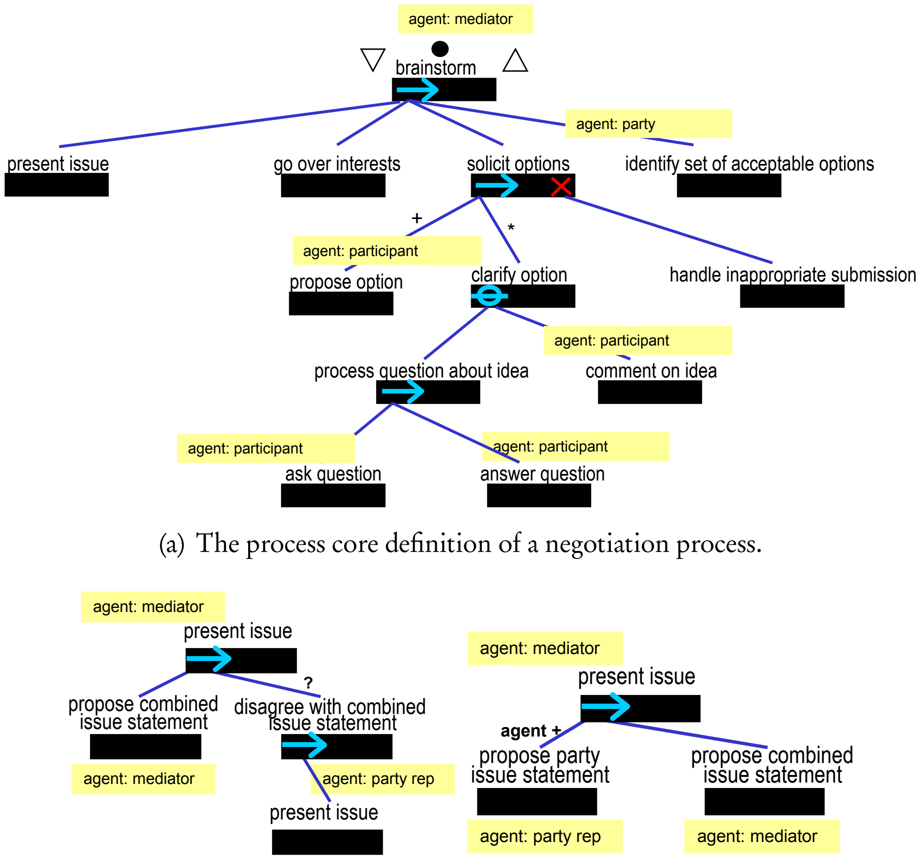
3.1.2 Process Guidance

At a high level, STORM2 provides software support for performing negotiations in order to achieve dispute resolution or bargaining between two parties. To do so, it enforces the dictates of a predefined process (part of which is illustrated in Figure 3.2(a)) that guides the negotiations, being able to provide important guidance to the mediator when needed.

It is important to note, however, that STORM2 provides *tailored* process guidance. That is, since the way that the system executes can be changed by simply changing the process definition, STORM2 can easily be customized to support negotiation processes that can differ depending upon whatever seems appropriate. This can be done by designing and defining a custom-built negotiation process, but more efficiently by making changes and modifications to an existing process. STORM2 thus does not mandate that a mediator follow a rigid process that is unsuitable for the current situation, but only provides the type of guidance that is felt to be needed, and only under circumstances that have been specified to be appropriate in accordance with the specifications of the underlying mediation process definition. This process definition can be customized with different functional variants or agent variants as illustrated next.

At a high level, the process consists of four main stages, or phases, that generally happen one after the other. Initially, the two negotiating parties must agree on an "issue statement," representing a summary of the issue that is being disputed. Each party then generates a collection of interests, or facts that they would like to be true when the negotiation is complete. The main brainstorming phase is then entered, during which a participant in the dispute first performs propose option to generate a list of possible solutions to the issue at stake, or, after an initial period of brainstorming, choose to clarify option by asking a clarifying question about someone else's submission. Finally, the fourth stage consists of the two party leaders voting for the set of options that their respective parties have found acceptable.

Several problem-level variation requirements have emerged out of consideration of the family of negotiation processes that seem useful in guiding STORM2. Examples of functional and agent variation follow.



(a) The process core definition of a negotiation process.

(b) One possible elaboration of present issue.

(c) Another elaboration of present issue.

Figure 3.2. Two sample process fragment elaborations for a given process core.

3.1.3 Functional variation in issue statement elaboration

The first phase of the interest-based negotiation process, the activity of presenting an issue statement, involves soliciting issue statements from the representatives of both negotiating parties and then constructing a combined issue statement that encompasses both sides' initial statements. This subprocess can be performed differently by different mediators. Some mediators iteratively refine the combined issue statement using explicit votes from all party representatives, while others create a single combined statement and go with it unless a party representative objects.

At the solution-level, this functional variation family can be modeled with a Little-JIL elaboration family as illustrated in Figure 3.2(a), which shows a high-level specification of the negotiation process, where the present issue step is an elaboration step. Two sample elaborations of the present issue step are presented in Figures 3.2(b) and 3.2(c), respectively. Figure 3.2(b) specifies the iterative approach where the mediator performs the step propose combined issue statement to suggest a starting issue statement representing both parties' interests, and each party representative can optionally (indicated by the question mark) choose to disagree with combined issue statement, resulting in a recursive reinvocation of present issue, so that the mediator can keep revising the initial statement until both party representatives agree with it. In Figure 3.2(c), on the other hand, each party representative (indicated by the agent+ notation) will propose party issue statement appropriate to the party he or she is representing, and afterward the mediator will propose combined issue statement, reflecting the input received from all parties. Either approach could be utilized in order to successfully complete the present issue stage of the negotiation process.

3.1.4 Agent variation to provide different levels of anonymity

Given the differences and similarities between process fragment elaboration and agent behavior elaboration, in some cases both approaches may be equally suitable and it may be up to the process developer to choose which approach better represents the observed variation, better suits anticipated future variation needs, and better fits with existing abstraction specifications within the process model. As an example of agent behavior variation that has been used to model observed variation in the negotiation domain, consider Figure 3.3.

The STORM2 ODR system discussed in Chapter 3.1 provides several anonymity settings for data that participants submit to the system. This anonymization is carried out in the process entirely within a software agent called the Redactor, which, according to the mediator's choice, either conceals or reveals the author. Two of the anonymization settings, encapsulated in different Redactor agent behaviors, are illustrated in Figures 3.3(a) and 3.3(b). Figure 3.3(a) illustrates the case when the mediator chooses to set the negotiation process to fully-attributed in which case the Redactor passes on information as submitted without making changes. On the other hand, Figure 3.3(b) details the case when anonymization is required and set to fully anonymous, in which case the Redactor removes author information from each piece of information before passing it on.

With respect to orthogonality between agent variation and process fragment variation, let us consider one more example. Recall the two subprocesses in Figures 3.2(b) and 3.2(c) illustrating different elaborations of the present issue step. Because we have different types of agents involved, process fragment elaboration is clearly needed to specify the two functional variants for the present issue step. However, note that both subprocesses specify a step named propose combined issue statement, executed by an agent who can fill the role of a mediator. Although variants for this step may be specified as process fragment elaborations, agent behavior elaboration is more appropriate here because all steps within the subprocess will be executed by the same agent, namely the mediator. The mediator must act differently for the two abstractions–in Figure 3.2(b), a new issue statement must be *composed* from scratch for the party representatives to approve, while in Figure 3.2(c), the party representatives propose issue statements first that is then *combined* by the mediator. These different agent behaviors can be grafted onto the different process fragments to create nested families (i.e. families where an agent behavior elaboration step can be specified within a process fragment elaboration variant). Specifically, recall that in Figure 3.2(b), either party representative can cause the reinvocation of the present issue, in which case the mediator would revise the original issue statement and proceed to check if both party representatives agree with it. Therefore, in the case of a reinvocation, it is the second agent behavior, where the statement is *combined*, that would need to be grafted onto the first process fragment to achieve the intended effect[2] and create a nested agent variation family from the existing functional variation family.

3.2 Case Study II: Variation in the context of secure and private elections

Elections are a cornerstone of the democratic process in countries such as the United States. Every citizen of the US above the age of 18 is entitled to participate in elections by casting a vote. Although all elections held in the US must satisfy many general requirements (e.g., no eligible citizen may cast more than one vote), there are some additional requirements that some election districts place on their elections, and these additional requirements may vary from one district to another. For example, all voters must always identify themselves to an election official, but different districts perform different processes for voter identification. In some districts voters are required to register their votes using electronic vote recording devices, and in other districts voting is done by placing marks on sheets of paper, which are then read either by election officials or by automated scanners. Thus it seems that election processes in the US can clearly be partitioned into different problem-level families, which could then be used to create different solution-level families. The solution-level families can be analyzed in order to reason about different security and privacy properties, which are critical in this domain. This case study is based on an ongoing collaboration with researchers at the University of California, Davis and election officials in Yolo and Marin counties in the state of California and shows how Little-JIL can be used to support the representation of some different kinds of such families, and indicates how these family representations can support our goals of generation and analysis to varying degrees.

[2] Note that in this case, agent variation at the problem-level, and the corresponding agent behavior elaboration at the solution-level, seem to be particularly suitable vehicles for explicitly modeling how rework is achieved through multiple subsequent executions of an activity that was not completed successfully the first time, as described in [17].

3.2.1 Challenges due to conflicting requirements and needs for variation

Some of the problem-level families concern different requirements in the privacy and security dimensions, either for variation or for invariance. Some desirable problem-level families are in fact based on properties that are in direct conflict with each other, so this case study provides illuminating insights on whether different dimensions of problem-space variation can be reconciled and whether they can be addressed by the same solution-space approach (namely elaboration), or how we can reason about more than one relation at a time. This is not unlike the kinds of reasoning identified and discussed in [63]. An example of conflicting requirements is that the voter confidentiality privacy requirement (no one but the voter should know how the voter voted) may raise potential conflicts with the one person-one vote security requirement (an eligible voter must be able to vote no more than once); different approaches to addressing these problem-space variation requirements may lead to the desirability of different approaches to solution-level variation in process variants. Specifically, we are interested how such interactions can be managed effectively, how families may be defined to satisfy the need for variation in some dimensions (e.g. agent variation to accommodate different voting machines) while maintaining invariance in other dimensions (e.g., security invariance to guarantee every machine conforms to a set of predefined properties). Thus, the members of a family of processes may all satisfy the same security property, and this may be the membership criterion for the family. Such invariance could be assured through the application of carefully selected analysis techniques that focus on predetermined properties and relevant events in the variants within the family. Note that such techniques clearly support our goal of supporting the analyzability of families as variants within the process family would clearly all satisfy given properties. However, although newly created variants could be checked for conformance to the properties quite straightforwardly, their initial generation may not be supported very well by these techniques.

3.2.2 Functional variation in ballot casting and provisional voting

Figure 3.4 presents how an example functional variation process family is modeled with a solution-level family consisting of a process core and two possible pairs of abstraction elaborations. The diagrammatic representation of a simplified, partial common core process for holding an election is shown in Figure 3.4(a) (for illustrative purposes, only a small part of the overall election process is shown here). This part of the election process indicates how a voter is identified and authenticated to cast a ballot and under what circumstances a provisional ballot is used instead.

As illustrated by the Little-JIL diagram, in order for a voter to pass authentication and vote, he or she must first present ID, then an election official will perform pre-vote authentication, and check off voter as voted, and, finally, the voting system of choice will record voter preference, as indicated by the name over each black step bar. The perform pre-vote authentication step indicates that the voter must pass a certain authentication procedure in order to be allowed to cast a ballot. If as part of that procedure it is determined that the voter has already been checked off as voted, then the authentication will fail and an exception of the type Voter Already Checked Off will be thrown.

The Voter Already Checked Off exception is handled by a handler that is attached via a red edge to the pass authentication and vote step. This exception is handled by performing the let voter vote with a provisional ballot step; the check mark in the annotation indicates that after invoking the exception handler, the process will continue to execute by regarding the entire pass authentication and vote step to be completed, with the result being that the step record voter preference is not executed and thus the voter is not allowed to cast a regular ballot in addition to the provisional.

Note that Figure 3.4(a) does not specify the details of how a regular or provisional ballot is to be cast. That is because there can be different *variants* on how to perform these steps. For example, Figures 3.4(b) and 3.4(c) demonstrate different ways a voter can cast a regular ballot, respectively by filling out a paper ballot and submitting it into a ballot box in Figure 3.4(b), or by filling out an electronic ballot, which is rendered by a DRE machine, confirmed by the voter, and then submitted to the memory of a DRE machine (the Help America Vote Act (HAVA) mandates that all DRE machines employed in the United States require voters to confirm their choices before a ballot can be cast) in Figure 3.4(c). Two different elaborations of the let voter vote with a provisional ballot are also illustrated in Figures 3.4(d) and 3.4(e). Note that when a provisional ballot is being cast, these two variants require that an election official perform the final step, submit provisional ballot, regardless of the kind of voting technology being utilized (a paper provisional ballot being cast into a provisional ballot box in Figure 3.4(d) or an electronic provisional ballot in Figure 3.4(d)).

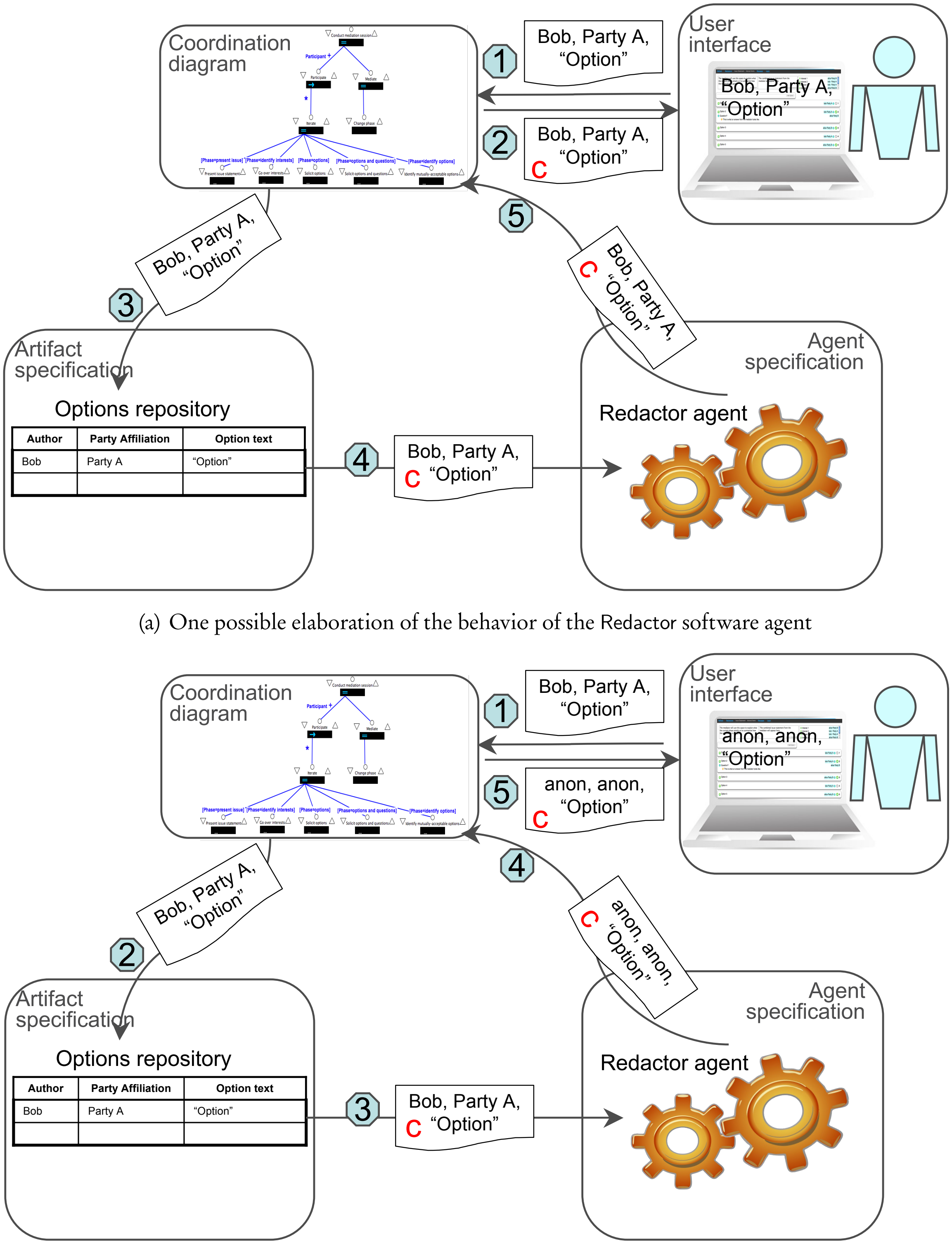
3.2.3 Agent variation to facilitate reasoning about different voting devices, detect fraud and collusion, and formalize attack modeling

Requirements to incorporate the use of different devices for the recording of votes creates the need for agent or service variation process families within this domain as well. Specifically, the submit ballot step that is the last child of the record voter preference step from Figure 3.4(b), is to be performed by an appropriate voting device. There are many such devices in use today, some of which are electronic (e.g. Direct-Recording Electronic (DRE) machines), and some of which are of other types (e.g. mark-sense reader devices). The activity decomposition structures of the process definitions within which these different devices are used do not differ from each other, but the processes are different in that the agent behaviors for the submit ballot step are different, as shown in Figure 3.5.

All DRE machines must remember the voter's intent by committing the completed ballot to memory, so all three behaviors include that step, and the behavior in Figure 3.5(a) completes after this step. However, most states require that if DRE machines are employed, they issue a Voter-Verified Paper Audit Trail (VVPAT), or a receipt stating that the ballot has been recorded. A behavior requiring this additional step is shown in Figure

3.5(b). Finally, instead of a VVPAT, some approaches have been proposed indicating that ballots can be assigned unique identification using cryptography that would allow a voter to verify if her ballot has been counted in the total tally or not without revealing her preference, as illustrated in Figure 3.5(c). Note that while these three variants could be represented as one behavior with appropriate optional steps, that would not faithfully represent the real world because a machine is either VVPT-compliant or not and that attribute should not change from one voter casting a ballot to the next if they are both using the same machine.

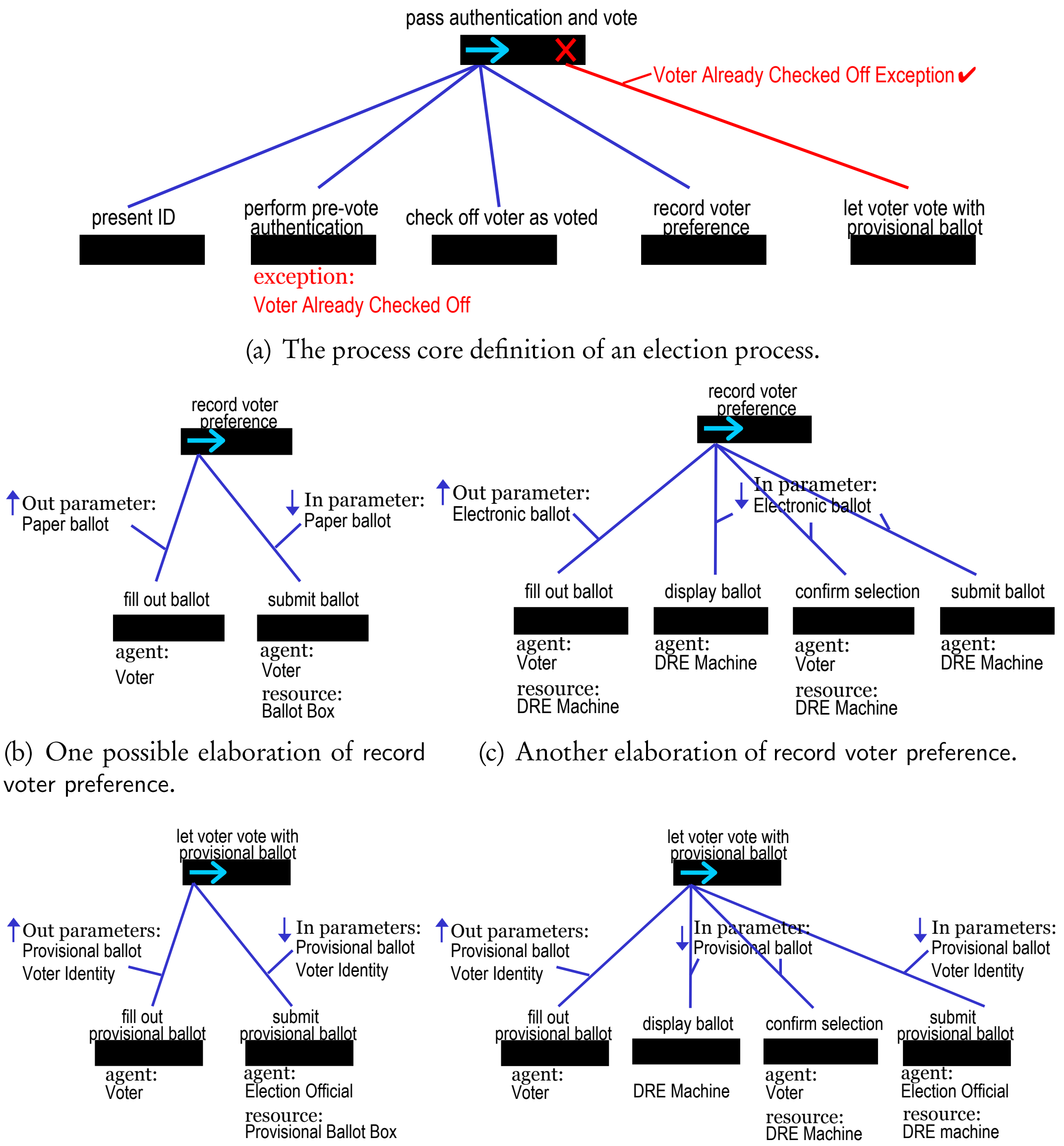
As illustrated by the examples in this chapter, many cases of observed variation from our real-world case study domains present themselves as functional or agent variation. We have indicated how this variation can be modeled and analyzed at the solution level through the exploitation of a common core that variants share with variant-specific elaborations that can be attached as appropriate. The conceptual framework for generating different kinds of Little-JIL process elaboration families presented in Chapter 2 is deconstructed into its technical components next.



(a) One possible elaboration of the behavior of the Redactor software agent

(b) Another elaboration of the Redactor's behavior

Figure 3.3. Two sample behavior variant specifications for a given software agent.



(a) The process core definition of an election process.

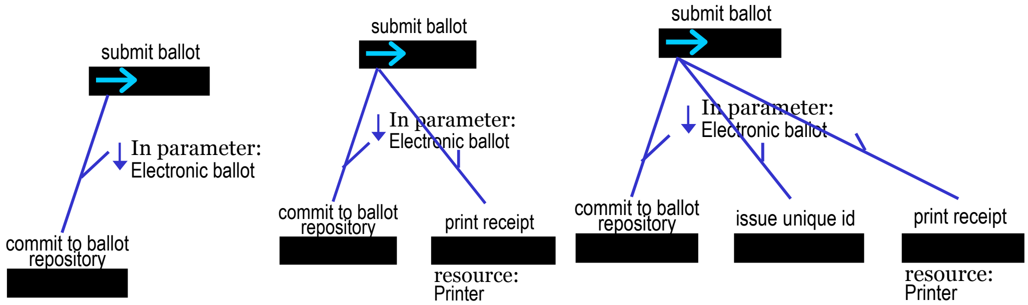
(b) One possible elaboration of record voter preference.

(c) Another elaboration of record voter preference.

(d) A possible elaboration of let voter vote with provisional ballot.

(e) Another elaboration of let voter vote with provisional ballot.

Figure 3.4. Two pairs of sample abstraction elaborations (in this case process fragments) for a given election process core.



(a) Basic submit ballot behavior. (b) Some DRE machines have a VVPT. (c) Ballots may be traceable by the voter.

Figure 3.5. Three sample agent behavior elaborations for DRE machines performing the submit ballot subprocess.

CHAPTER 4

TECHNICAL APPROACH

This dissertation outlines an approach for modeling, understanding, and reasoning about system variation to support the two research goals of RG1: *Generation* of new system variants based on provided desiderata and RG2: *Analysis* of entire families of systems based on different variation relations. Two main contributions are presented: 1) a modeling framework for specifying process variation and process families to support RG1 and 2) an analysis harness for reasoning about entire families of processes with respect to different properties they need to satisfy to fulfill RG2.

The outlined specification of Little-JIL process definitions from Chapter 2 is helpful in formally defining different kinds of elaboration-based families; the successful implementation of this framework, however, also necessitates the provision of adequate tool support. This chapter provides a specification of the system implementation to achieve the goals of RG1 and RG2.

4.1 Generating Families with The Little-JIL Elaborator

This section details an implementation plan with respect to the conceptual framework outlined above. The research prototype tool presented, the Little-JIL Elaborator, is guided by the research challenges and variation needs encountered in our two case study domains—negotiations and elections. In Appendix A, we outline a vision for an ambitious framework, called Process Line Analysis, Generation, and Evaluation (PLAGE), for carefully specifying process families using different constraints and conditional requests–and perhaps employing different repositories–for abstractions, whether they be agent behaviors specifications or process fragments. In this chapter, we focus on the finished prototype product, the Little-JIL Elaborator, which can generate process families based on developer-guided selections of abstractions, and supports their analysis. The appendix details how the prototype fits within the overall PLAGE system architecture and outlines all the system components and interfaces that have been implemented to support the future integration.

The Little-JIL Elaborator is a research prototype that supports the implementation of some solution-space approaches and allows for empirical evaluation of the usefulness and applicability of the described approach to meet the stated hypotheses of improved support for generation (RG1) and analysis (RG2). The Little-JIL Elaborator is implemented as an extension to the existing Visual-JIL framework, and allows for the definition of families of Little-JIL processes along several different variation relations. As a research prototype, the Elaborator support the specification of a certain kind of process families, namely ones that all share a common core and differ at lower levels of abstraction. This class encompasses process families that seem to be potentially capable of addressing the need for several different kinds of problem-level variation discussed in this dissertation, including, but not limited to, functional variation, robustness variation, performance variation, service variation, and interaction-based variation. For each of these kinds of variation relations, we have observed variation needs in one or both of the case study domains that would successfully be modeled with a process family that is built around a common core (some example families are outlined below as well as in Chapter 3 above.

Specifically, the Little-JIL Elaborator is designed to provide the following functionality to address the indicated key research goals by supporting two main kinds of elaboration specification, namely process fragment elaboration and agent behavior elaboration. This set of functionality is expected to provide substantial support for exploring several fundamental research questions that are described in more detail below.

4.1.1 Process Fragment Elaboration

Currently, Visual-JIL supports six different step kinds as building blocks to a process definition, namely *sequential*, *parallel*, *choice*, *try*, *leaf* and *reference*. A *reference* step is used as an invocation mechanism to any of the other five step kinds. We have added support for an *elaboration* annotation for the reference step kind (hereafter referred to as an *elaboration* step kind), which works similarly to a reference but instead of resolving to a single step definition allows the *elaboration* step to be replaced by any of a number of different pre-specified abstraction elaborations. This enables a step that may have previously been modeled as a leaf step to be replaced with a placeholder elaboration step indicating that elaborative details are to be provided by appending a new abstraction to it. This capability greatly facilitates the generation of new process variants, thereby supporting Research Goal #1 (RG1).

In order to facilitate RG1, process family generation, from a usability standpoint, these *elaboration* steps can be modified to contain interface information and therefore allow for the specification of variants and process navigation. As currently defined, reference steps do not have an interface specification accessible to the process developer, but the step declaration that the reference resolves to can easily be accessed on-demand and inspected to see those interfaces. Since an *elaboration* step by definition may resolve to multiple elaborations, this approach no longer seems feasible or user-friendly. Additionally, current Little-JIL developers have indicated that such a globally-accessible interface specification would be helpful for normal *reference* steps as well, and this usability enhancement is considered as future work.

4.1.1.1 Examples from the case study domain

As presented in Section 3.1, mediators can choose to perform the present issue step differently when executing the overall negotiation process. The suggested process core and two possible process fragment elaborations for present issue are shown in Figure 3.2.

On the other hand, in the domain of elections, voters can also have their voter preference recorded differently, as shown in Figure 3.4(a).

In both of these case, functional variation seems to be accommodated well by the different elaborations of the same high-level step (present issue or record voter preference). The variants presented exemplify some possible applications of functional flow changes to achieve functional variation. Note that in the case of negotiations, agent behavior variation may also be utilized to model lower-level behavior differences for the mediator or party representatives, but that does not seem to be the most appropriate approach to meeting this specific problem-space variation requirement.

To allow for "elaborating" a process core with different abstractions implementing different variants, there are several problems that must be considered. First, the variants must satisfy syntactic and semantic constraints to ensure compatibility between the declared interfaces of the *elaboration* step and any process fragment or agent behavior decorating that step. The existing Little-JIL semantic checkers extend to *reference* steps but do not currently allow for reasoning beyond one single definition of a step that a reference invokes. Management of such multiple definitions is one of the main goals of the Elaborator in order to meet RG1. Second, in order to address RG2, the abstraction elaborations must be checked for adherence to different properties (e.g., safety, robustness, or security properties), to ensure that any variant that can be generated is an acceptable member of the family with respect to the pre-specified membership criteria. For the negotiation example presented, all the party representatives and the mediator must be available for the successful completion of any possible elaboration, and each elaboration should result in the creation of an issue statement artifact capturing what the parties will be negotiating. For the election example, both the voter and an appropriate voting device must be available for the successful completion of any elaboration and additionally, an election official is required to cast a provisional ballot. Each elaboration should result in the creation of a ballot artifact capturing the voter intent. The Little-JIL notation seems particularly adept at supporting the specification of such a family, as this requires only that there be a specification of a set of service providers that have the capabilities needed to carry out the submit ballot step. As outlined in Chapter 2, these variants can all be grafted onto the appropriate core process as the children of a choice step connected to the step being elaborated. This would allow the user to explore the adoption of different variants or perform analysis encompassing all possible alternatives.

Furthermore, because the agent and coordination specifications in Little-JIL are independent, agent variation families provide the opportunity for interesting analyses looking into agent collusion scenarios to detect corrupt election officials, or juxtaposing the agent behaviors vs the process specification to identify vulnerabilities against malicious agents and devise protections against them. As noted in Figure 2.1, given Little-JIL as the system representation used within the solution space, agent behavior can often be transformed into process structure (where the behavior is specified as a process fragment and the agent assigned to each step within that fragment is the original agent), and process fragments can sometimes be abstracted out as agent behaviors (where the root step of the process fragment becomes a leaf step and the details of its elaboration become the specification of the agent behavior). Additionally, note that process fragment elaboration and agent behavior elaboration as described and illustrated in Figures 3.4 and 3.5 are not necessarily orthogonal to one another. Indeed, they intersect, and nested families can be created recursively by applying different elaborations of the submit ballot agent behaviors to different process fragment elaborations of record voter preference as noted. Therefore, we often use the term abstractions, or abstraction elaborations, to refer to any elaboration that would semantically *fit* onto an elaboration step.

Some examples clearly indicate which approach is more suitable; note that in the example shown in Figure 3.4, there are multiple types of agents responsible for the execution of the substeps of both the record voter preference and the let voter vote with a provisional ballot steps, and therefore abstracting these subprocesses as single-agent behaviors seems problematic. In fact, for initial versions of the Little-JIL Elaborator, we define agent behavior variants to be a subset of process fragment variants such that the executing agent for every step within the abstraction is identical. Moreover, we impose the restriction that agent behavior elaborations can only be performed at elaboration steps where the executing agent is specified as a new resource acquisition, as opposed to a resource use[1].

Even though functional variation tends to be quite well modeled and accommodated by traditional software product line engineering (SPLE) approaches, there has been considerably less attention targeted at service, or agent variation. One of the strengths of the Little-JIL process definition language as compared to other variation modeling platforms is the fact that it does not discriminate between human and computer agents, and, furthermore, allows for a very flexible definition of agent behaviors. Providing support for agent variation is therefore a major focus of this work. Agent variation raises many challenges, both from a research and implementation standpoints. From a research standpoint, there are two different approaches to considering agent variation, namely whether behavior variation is considered on an instance or type basis (that is, if a step specifies that two different agent behavior variants may be used to execute it, does that mean that one behavior will be chosen every time that step is executed, or will one behavior be chosen some of the time, and the other at other times?; would the choice be constrained by the context within which the step is invoked?; would a choice of agent behavior at one step influence a choice at another step because of overall process constraints? Some of these questions are revisited in the request language specification later in this section). From an implementation standpoint, the representation used to specify agent behaviors becomes quite important.

[1] Given the hierarchical nature of Little-JIL and many other process definition languages, every step that is elaborated is effectively replaced by the composition of its children and therefore no “real work” is performed at non-leaf steps. Therefore it is not clear what might be meant by specifying agent behavior variation for a non-leaf step, and thus restricting agent behavior variation to newly acquired agents seems reasonable.

4.1.2 Architecture and Separation of Concerns

The Elaborator prototype provides initial support for the specification of abstractions, usually represented as process fragment elaborations for analyzability[2]. Since agent behavior specification in Little-JIL is orthogonal from coordination specification, the Elaborator must maintain appropriate separation of concerns while allowing for maximal reuse. The current architecture of the tool is shown in Figure 4.1 and shows three main components, namely the current Visual-JIL editor, a query engine middleware component, and an abstraction repository. Each component is described below in more detail.

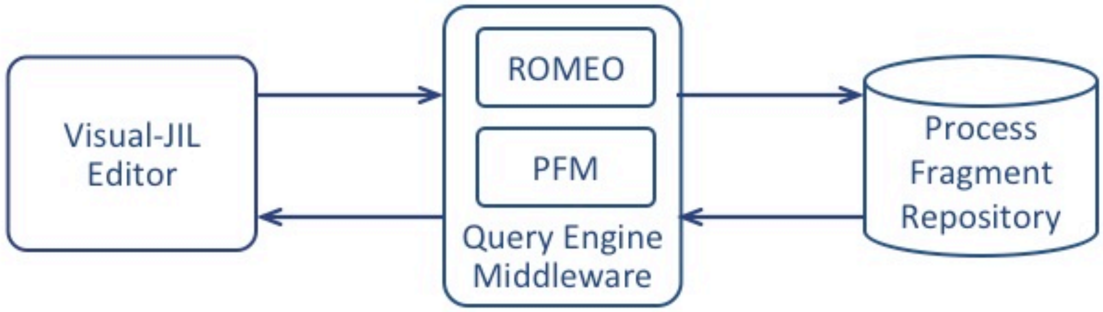


Figure 4.1. High-level architecture sketch of the Little-JIL Elaborator.

4.1.2.1 The Abstraction Repository

The abstraction repository is a data store for different variant elaborations. Since in the prototype versions of this framework, agent behaviors are specified as Little-JIL subprocess elaborations to facilitate analysis, an agent behavior becomes a special case of a process fragment where all substeps are assigned to the same agent (the agent whose behavior this subprocess is describing). Moreover, to support broad analysis (RG2), both process fragments and agent behaviors are used to decorate a Little-JIL process definition. Accordingly, storing both abstractions in the same repository seems to not only effectively

[2] We have previously discussed how agent behavior can be elaborated as part of the process hierarchy and parts of the process hierarchy can sometimes be abstracted away as agent behavior, provided the entire abstraction is performed by the same agent

support reuse but also some of the solution-level variation accommodation techniques shown in Figure 2.1, such as "agent behavior to process specification" and "process specification to agent behavior." More important, specifying the repository as an abstraction repository allows for easy extension to accommodate desired variant specifications discussed in future work, such as using code fragments encoded in heterogenous languages, or sockets to communicate with URLs for online service providers to plug in as agent behaviors.

The high-level schema for the repository is specified as:

$\{step, type, \underline{id}, [attribute]*, access\}$

A description of each element of the schema follows:

- $step$ (a character string) is the full or partial path specification[3] of a step appearance for which this abstraction specification is an elaboration. The step appearance specification can be used as the primary way to search for appropriate variants when there are multiple matches as described below.

- $type$ (an enumerated value) is the type of abstraction that this specification describes. The two types currently supported are "process fragment" and "agent behavior" as specified above.

- $\underline{id}$ (a numeric) is the primary key for the schema and is an automatically-generated number to uniquely identify an abstraction.

- $[attribute]*$ (a list of character string tuples) is a list of zero or more attributes defined as $\{type, value\}$ tuples, stored separately, where examples of $type$ might be "agent name," "performance score," "corrupt flag," "input constraint," "output constraint," and so on, and the values would be the corresponding settings.

[3] Partial specification refers to the ability to specify multiple appearances of a step within a process definition using the XPath language, as outlined in Chapter 2.

- *access* (a pointer or reference) is a mechanism to allow access to the actual abstraction stored in memory.

4.1.2.2 The Visual-JIL editor and Query engine middleware

Little-JIL process definitions consist of three main parts–a coordination diagram, a resource specification, and an artifact specification. These specifications are independent from one another but get combined through the specification of interface declarations within the coordination diagram.

Process fragment elaboration is achieved through the elaboration step mechanism described above. Specifically, when an elaboration step in a coordination diagram is specified, this results in the editor consulting the query engine middleware, which in turn will consult its Process Fragment Manager (PFM) subcomponent. In adding a new variant, the PFM will conduct the necessary well-formedness checks (these are called Critics within the Eclipse Visual-JIL development environment) to ensure that the new abstraction conforms to Little-JIL syntax and that its interface specification is appropriate given the elaboration step it will be decorating. The PFM will then add the new fragment to the abstraction repository annotated with any appropriate attributes as described in the variant specification. In retrieving existing variants, the PFM will provide the repository with a set of attributes that the variants must match, and return the results it gets. By default, all abstractions matching the name of the elaboration step will be returned, but additional mechanisms for filtering depending on additional attributes can be provided through the extensions described in Chapter A. For example, referring back to Figure 3.4, both variants will be returned by default if the PFM asks for variants for the step present issue, however in cases when there is a limited amount of time to carry out a negotiation, indicating the need for a certain performance variant, then only the variant presented in Figure 3.2(b) would be returned because it tends to lead to a faster completion of the "present issue" phase[4].

The PFM will be integrated within the existing Little-JIL infrastructure to augment the current resolution activity that conducts a search over the coordination diagram definition to resolve reference steps to their unique explicit elaborations. The new Resolution activity for reference steps annotated as elaboration steps will find and return all elaborations of a reference instead of a unique elaboration as currently defined. Upon request, when the user is ready to perform analysis (RG2) on the whole process family, the elaboration step will be elaborated as a choice step and all variants matching the user request will be grafted onto the coordination diagram as children of that choice step (as shown in Figure 2.3).

From the user's perspective, agent behavior is currently handled the same way. However, in future iterations, the system can be extended to integrate the ROMEO resource manager [60] as a plugin and instead handle agent behavior variation through the current resource specification mechanism within Visual-JIL. Within a Little-JIL process definition, at every step where an executing agent is declared, the specification of the agent declaration is used by a resource manager to identify the appropriate agent for the execution of the step. When there is no agent specification, a default query is launched to get all agents that are capable of performing the specified step. The query engine middleware could be modified to hand any new agent behavior variant specification to a resource manager to store in the abstraction repository, and when retrieving existing agent behavior variants, ROMEO could be modified to identify all agent behaviors matching the user request and return the appropriate behavior specifications, with the same defaults as for the PFM.

[4] This preference has been indicated by mediators in multiple experiments when a time limit is imposed.

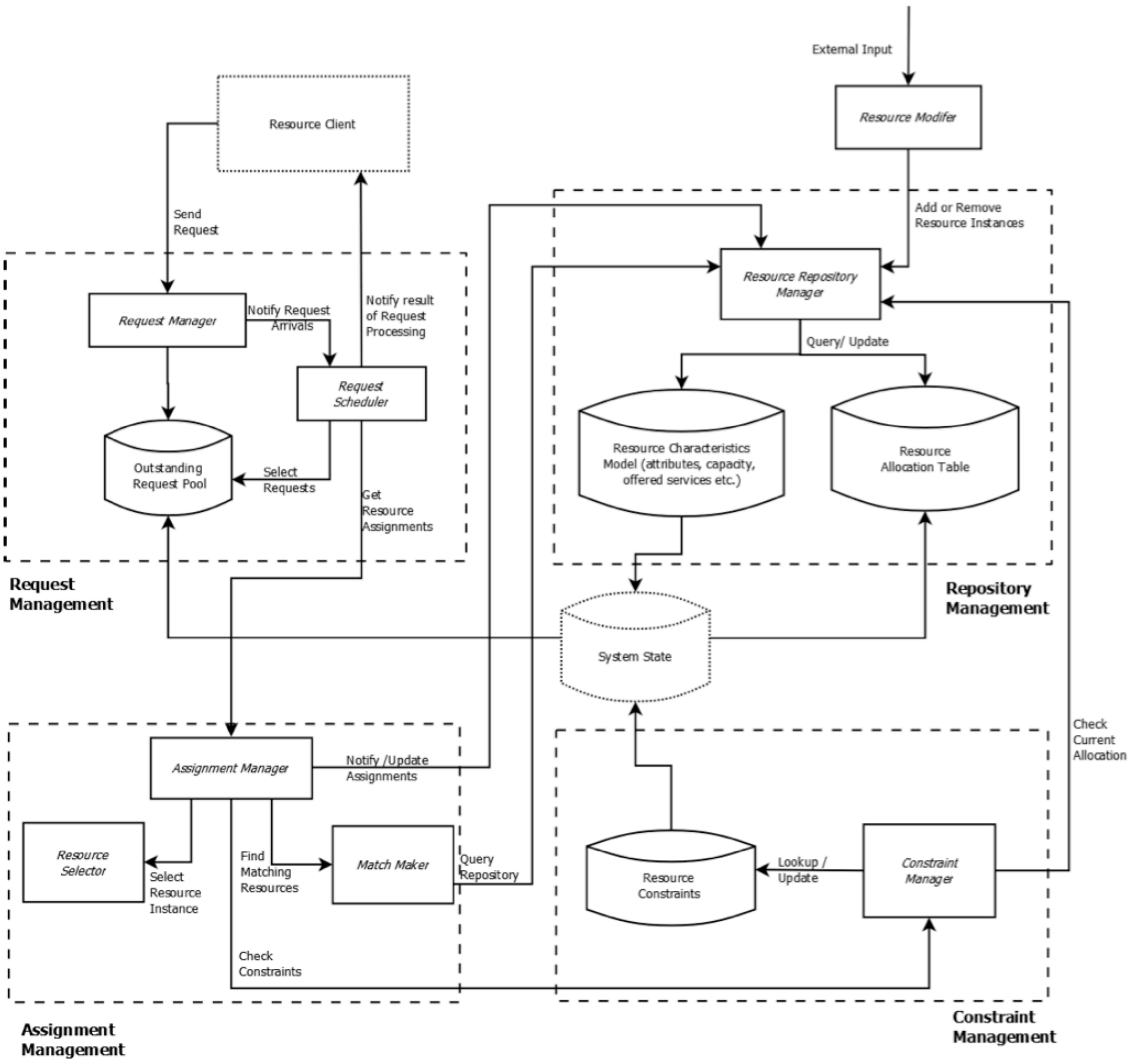


Figure 4.2. Current architecture of the ROMEO resource manager, as specified in [60].

The current architecture of ROMEO is shown in Figure 4.2. Currently, the Assignment Management component is configured to select exactly one agent instance that best matches the request received from the Resource Client. The Assignment Management component would therefore be bypassed in order to allow the user to retrieve any number of agent behaviors that match the desirable criteria. Moreover, the Request Manager and Request Scheduler would have to be modified to reflect the need for returning all matching agent behavior specifications and the abstraction repository would be used as the Resource Characteristics Model. However, unlike process fragment elaboration, agent behaviors would not be analyzable within a Little-JIL process specification given the current constraints on the analysis framework, so extensive additional changes to the analysis harness would need to be performed to make such families easier to reason about.

4.1.3 Storing, Managing, and Organizing Abstractions

The Elaborator will facilitate the specification of multiple variants of several kinds and will provide support for the storage and organization of these variants within repositories. Variants may be defined according to the different variation relations already discussed. For example, there may be a repository of process elaboration fragments, another repository for different agent behaviors, and a third repository for collections of properties that an entire family must satisfy. Variant repositories will be organized by the constraints and membership criteria defined by the variation relationships among variants. (RG1)

There are apparent implementation challenges that need to be considered in order to provide adequate support for the storage and navigation of process variants, such as appropriate representation, performance, and sufficient querying capabilities. However, from a research standpoint, one of the most interesting questions would be the application of appropriate membership criteria to constrain what variants should be allowed to be members of a certain repository, and, moreover, selecting variants from a given repository based on desiderata they must satisfy (to support RG1). If variants can be selected based on desiderata related to the properties they are guaranteed to satisfy (while also meeting problem-space variation needs), then newly generated variants may under certain circumstances be guaranteed to be safe by construction, thus facilitating RG2.

4.1.4 Request language specification

Specifically, in order to address some of these challenges and retrieve only the appropriate variants from the Abstraction Repository based on a set of user-supplied desiderata, we define the following request language specification. A request for variants will have the general form:

$request\{name, type, context*, attribute*, best*\}$

A description of each element of the request follows:

- $name$ (a character string) is ether the (complete or partial) path specification of the step for which variants are being requested, or the name of the type of agent whose behavior the user wants to vary, or both. Figure 3.4 illustrates the case in which lookup would be performed by step appearance, whereas in the case of Figure 3.3 both a step appearance specification and an agent name would be required. Using only an agent name as the $name$ allows for the "corrupt election official" use case described above where an additional flag may be set indicating that certain agent behaviors are corrupt, indicating the need for collusion analysis.

- $type$ (an enumerated type) specifies whether the variants requested should be process fragments or agent behaviors

- $context*$ is a list of zero or more partial or complete paths to specific step appearances within a process definition. The purpose of $context$ is to allow for the elaboration of only certain appearances of elaboration steps depending on where in the process definition they are invoked. If no context is specified, then all elaboration steps matching the original path specification of $name$ are elaborated with the resulting set of variants by default, as is the case with reference steps currently.

- $attribute*$ is a list of zero or more attributes that the set of variants should satisfy. If no attributes are specified, all variants matching the $name$ and optional $context$ will be returned.

- $best$ is an optional flag for restricting the set of returned variants to exactly one. This option is available specifically to allow for the generation of explicit variants either for the purposes of dynamic execution, simulation, or analysis, and is meant to mimic the ROMEO Assignment Management module that is being bypassed.

Note that the term “best” is used liberally and does not imply optimality, but only that a *single* variant that matches the criteria will be returned.

4.2 Analyzing Families with the Little-JIL Analysis Toolset

4.2.1 Finite-State Verification

Finite-State Verification (FSV) is an analysis technique that considers a conservative representation of a process definition or program to determine if there exists a trace that could violate a predefined property. To verify whether a Little-JIL process definition satisfies or violates a given property, we use the FLAVERS [31] analysis engine. In order to determine if all traces through the process definition adhere to the property specification, FLAVERS must know the correspondence between the alphabet of the property (i.e. the events of interest), and the steps of the process.

Therefore, analyzing a Little-JIL process definition with FLAVERS consists of three main steps: 1) creating a formal specification of the “legal” property, defined as a set of sequences of events, 2) specifying a set of bindings that indicate the correspondence between steps in the process definition and events in the property’s alphabet, and 3) running the analysis engine to determine if there is a trace through the process definition that can possibly generate a sequence of events that is not a member of the “legal” sequence. If such a trace is found, FLAVERS provides an example trace as a counterexample.

System requirements are usually high-level desiderata that are most useful when decomposed into multiple lower-level, more precisely defined properties that can be formally specified to support automated analysis, and to avoid any possible ambiguity. We use the PROPEL tool [74] to define properties. PROPEL provides guidance for property specification by means of an English-language question tree, and automatically generates a finite-state automaton (FSA), based on the answers to the questions. PROPEL properties are specified independently from the Little-JIL process definition, and represent the set of allowable event sequences against which process traces are to be compared.

FLAVERS constructs a finite model of the process definition that represents all the possible event sequences, for the events in the property's alphabet, that could occur over all possible traces through the process definition. It then determines if this model is consistent with the property specification. 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 process execution trace and corresponding event sequence that causes the property FSA to fail to terminate with an accepting state (e.g. by entering a 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. If the process definition being analyzed represents a process family, a violation may indicate that one or more variants lead to a violation, and a process developer may use this information to iteratively remove identified culprits from the process family to determine if a smaller family may satisfy the property.

To analyze a process family that has been specified as a process core and elaborations of different kinds at the pre-specified *elaboration* steps, we could use the Little-JIL Elaborator to explicitly graft all possible elaborations (variants) onto elaboration steps. Note that not all possible combinations of elaborations may be appropriate; for example, the use of one variant at one elaboration step appearance may preclude another variant from being used elsewhere in the process. The PLAGE architecture includes provisions allowing for such constraints to be specified as attributes in the request to the Little-JIL Elaborator to ensure that only reasonable combinations of variants are explicitly grafted on for analysis, avoiding nonsensical combinations that may lead to spurious violations. The resulting process would then contain the possible traces for all appropriate variants, and could then be analyzed using the existing FLAVERS finite-state verifier for Little-JIL to achieve RG2. For large numbers of possible grafts (and indeed it is possible to contemplate an infinite number of grafts) this approach becomes either impractical or impossible.

Thus, we consider approaches that help with analyzing larger families in discussions of future work. One obvious step is to prune all agent and functional variants that do not contain events from the property alphabet, as they cannot influence the outcome of the analysis. The analysis engine already performs some such optimizations, but pruning at the process definition level would facilitate addressing other concerns, such as automatically generating instances for self-adaptive systems. Another, more interesting technique would be to allow the Little-JIL Elaborator to selectively graft variants based on different constraints that may mandate that the same agent behavior cannot be used in more than one step appearance, for example, or that a certain functional variant is always used with a certain agent variant (this is actually a common case in elections, where different voting machines–i.e. different voting machine agent behaviors–mandate different functional variants for ballot casting). Such sophisticated constraints are anticipated, but not currently supported within PLAGE.

4.2.2 Fault Tree Analysis

In addition to finite-state verification, there is another form of analysis called fault tree[5] analysis (FTA) [16, 19, 32], which determines how hazards may occur as a result of the incorrect performance of certain activities in the process. A hazard is an undesirable or unsafe state of the system that may allow for a critical failure or accident to occur. At a high level, a fault tree is a hierarchical structure that has a root corresponding to this hazard, which is then decomposed into events that may lead to the hazard occurring if some combination of the events occurs. The decomposition semantics are defined using logical gates, much like a circuit, where an AND gate indicates all subevents must occur for the composite event to occur, whereas an OR gate means at least one subevent must occur. The structure is defined inductively starting at the root and decomposing each

[5] Fault trees are similar to attack trees [64], however, we do not assume a malicious intent and consider all possible ways that could lead to a hazard, including accidents or negligence.

level of the tree until the leaves of the tree correspond to leaf steps, or simple events in the process.

The FTA Toolset for Little-JIL provides automatic derivation of fault trees given a Little-JIL process and a hazard specification. Once a fault tree is derived, it can be reduced to an algebraic equation consisting of the root event on the left and the decomposition on the right (recursively substituting each event with the expression for its subevents connected with the appropriate logical operator. If the root is assigned a truth value, this equation can then be solved using standard satisfiability techniques to determine the minimal combinations of variables on the right that would need to be true for the hazard to occur. Given that the variables on the right correspond to events (e.g., the occurrence of an exception and how it is handled or not handled, incorrect performance of a step, propagation of a key artifact, and others) from the process definition, these reduce to minimal combinations of events that may cause the hazard will occur. Each combination is called a minimal cut set (MCS), and a MCS of size one indicates a single point of failure.

To determine where in the process definition key artifacts are created and modified—and therefore at risk for being corrupted—FTA analyzes the data flow in the model leading to the step associated with the hazard being examined. The FTA Toolset should not require modifications in order to be applied to an elaborated process, since the process would just contain more options for how artifacts can flow through it. Such a process can be automatically generated in PLAGE with the Little-JIL Elaborator explicitly grafting the variants of interest onto their respective elaboration steps.

FTA may be useful in detecting agent collusion within a process through studying the steps in each MCS to determine if they can all be executed by the same agent, and whether a "corrupt" agent behavior may be used to elaborate the agent in a variant. This technique may also be used to increase the robustness of a process through adding redundancy and limiting agent responsibilities in such a way that there are no single points of failure or single agents who can affect the integrity of the overall process (for example, that there is not a single election official with enough power to overturn an election without colluding with other agents, or causing a voting machine to fail, or other external factors).

The results from FTA can be used to determine what variants can contribute to a hazard occurring (those whose steps appear in the resulting fault tree), and what variants can be considered safe. The Little-JIL Elaborator can then be used to generate families that are provably resistant to certain vulnerabilities, or identify combinations of variants that mitigate other vulnerabilities (for example, single points of failure can often be mitigated through the addition of redundancy in the process definition, and PLAGE could be used to identify variants that may provide that kind of redundancy). Note that the converse is also true—PLAGE could be trained to select the combination of variants for analysis based on lateral relationships among these variants, such as grafting two abstractions at different elaboration steps that are known to often occur together (for example, in elections, an electronic voting machine agent behavior abstraction usually correlates to a specific vote-counting functional abstraction).

CHAPTER 5
EVALUATION

In order to determine if the anticipated contributions outlined in Chapters 2 and 4 have been met satisfactorily, we undertake the following evaluation approach. We describe the evaluation of the two research goals, Generation (RG1) and Analysis (RG2) separately, starting with quantitative evaluation and then specifying some qualitative metrics as well. Throughout this section, we use a case study of a large family of election processes describing different remote voting schemes, including scenarios for voting over the Internet.

5.1 Experimental Setup

We focus on evaluating the scalability of PLAGE, and specifically the applicability of process family generation and analysis techniques to an extensive case study with more variation points and more variants than previously considered. We use a specially produced family of families (i.e., a process family that includes multiple variation points that interact with one another) that focuses on remote voting. This process family was developed as a result of collaboration with the National Institute for Standards and Technology, NIST, as a potential security assessment framework for Internet voting technologies. The complete family comprises 79 unique steps, including three variation points and twelve variants with dozens of intricate interactions among them. The total number of variant combinations exceeds one hundred. You can find the complete specification in Appendix E, along with coordination diagrams for intermediate steps that are not included in this chapter.

There are clearly different aspects of generation and analysis that necessitate different evaluation strategies. We have previously studied the applicability of the approach to detecting malicious agents through the novel application of finite-state verification to a vote-in-person election process family based on the complete Yolo county process. For that case study, a subset of which was presented and evaluated for the 2014 Software Product Line Conference (SPLC) in [67], we considered how malicious or incorrect agent behaviors could be leveraged within a process family for correctness analysis. The extended SPLC case study uses a slightly larger process family of 98 unique steps, but there are only nine variants and two variation points. We omit the details here, but the full specification is available to the interested reader in Appendix C. Despite the small number of total variants, the case study still surpassed the capabilities of the then existing fault tree analysis framework and led to the improvements that have made this set of experiments possible. Scalability and performance results from that extended case study are included for completeness with the rest of the experiments below.

We focus on two specific kinds of solution-level variation observed in the process family, namely functional and agent variation. Agent variation is frequently a special case of functional variation, so all variants within our case study are presented together as observed activity variation in Table 5.1. The ballot marking subfamily is based on agent variation because its execution depends solely on the voter agent behavior and the voting modality used. The rest of the subfamilies are examples of functional variation because they include teams of election officials as well as voters in some cases. Note that Table 5.1 presents only the nominal, expected agent behaviors. Possible malicious agent behaviors are shown in Table 5.2 for completeness, but are not considered for this scalability assessment. Such specifications can be evaluated using the approach presented in [67], and would provide interesting insights into socio-technical vulnerability assessments in future work.

Process Variation Point	Vote by Mail (VBM)	Vote by Fax	Vote by Email	Vote by Ballot-on-Demand
Ballot distribution	Mail out	Fax out	Email out	Email or mail link to ballot
Ballot marking	On paper, special ballot stock marked with precinct number	On paper, regular printer paper	Mark then print or print then mark, depends on the ballot	Log in, mark pdf-like version of ballot online and submit it
Ballot collection	Mail back	Fax back, along with signed waiver	Fax or mail back a printout	Log in and access voters' submitted ballots
Other		Fax back to acknowledge receipt of ballot	Voter sign waiver if faxing; Election official fax back	
Ballot counting	Standard procedure	First print all electronic/facsimile/printout ballots that are received on ballot stock, then follow standard procedure for VBM ballots		

Table 5.1. Variation points in different activities in the Remote Voting process family.

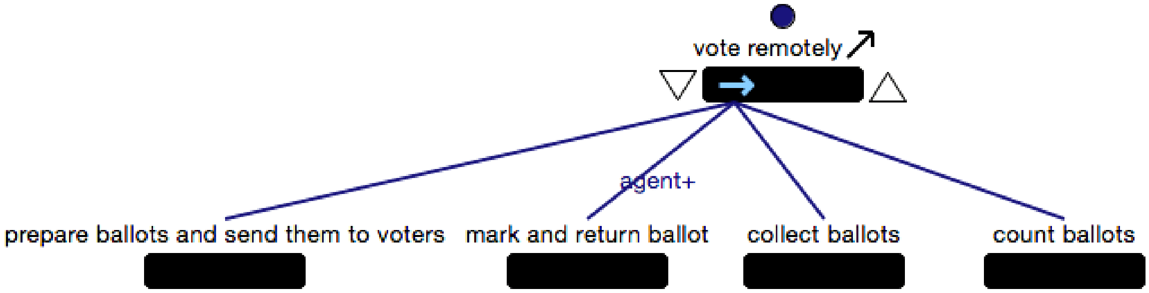


Figure 5.1. The main diagram of the vote remotely process family. The mark and return ballot and collect ballots references in this diagram have been defined as elaboration steps. Their corresponding abstractions are shown in Figures 5.4 and 5.5, respectively.

At the high level, the vote remotely process family is shown in Figure 5.1, with the eponymous root step, which is sequential as denoted by the arrow in its step bar. Its first substep, prepare ballots and send them to voters, is shown in the following Figure 5.2. The prepare ballots and send them to voters step is itself sequential, comprising acquire list of voters and distribute ballots. Within the Little-JIL internal form, this step is defined to be an elaboration step.

Figure 5.3 shows all the abstractions that can elaborate distribute ballots in this case study. They provide the mechanisms to distribute ballots to voters using modalities for voting by mail, fax, email, of ballot-on-demand submissions. The full declarations for steps referenced in these diagrams are available to the interested reader in Appendix E.

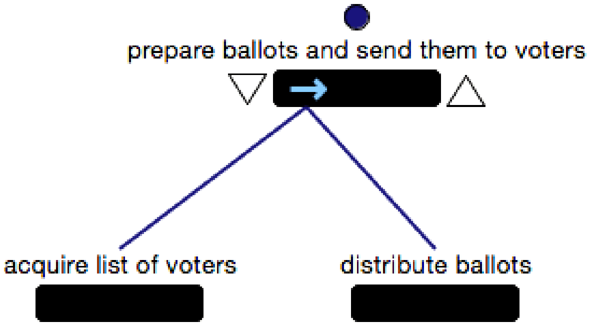
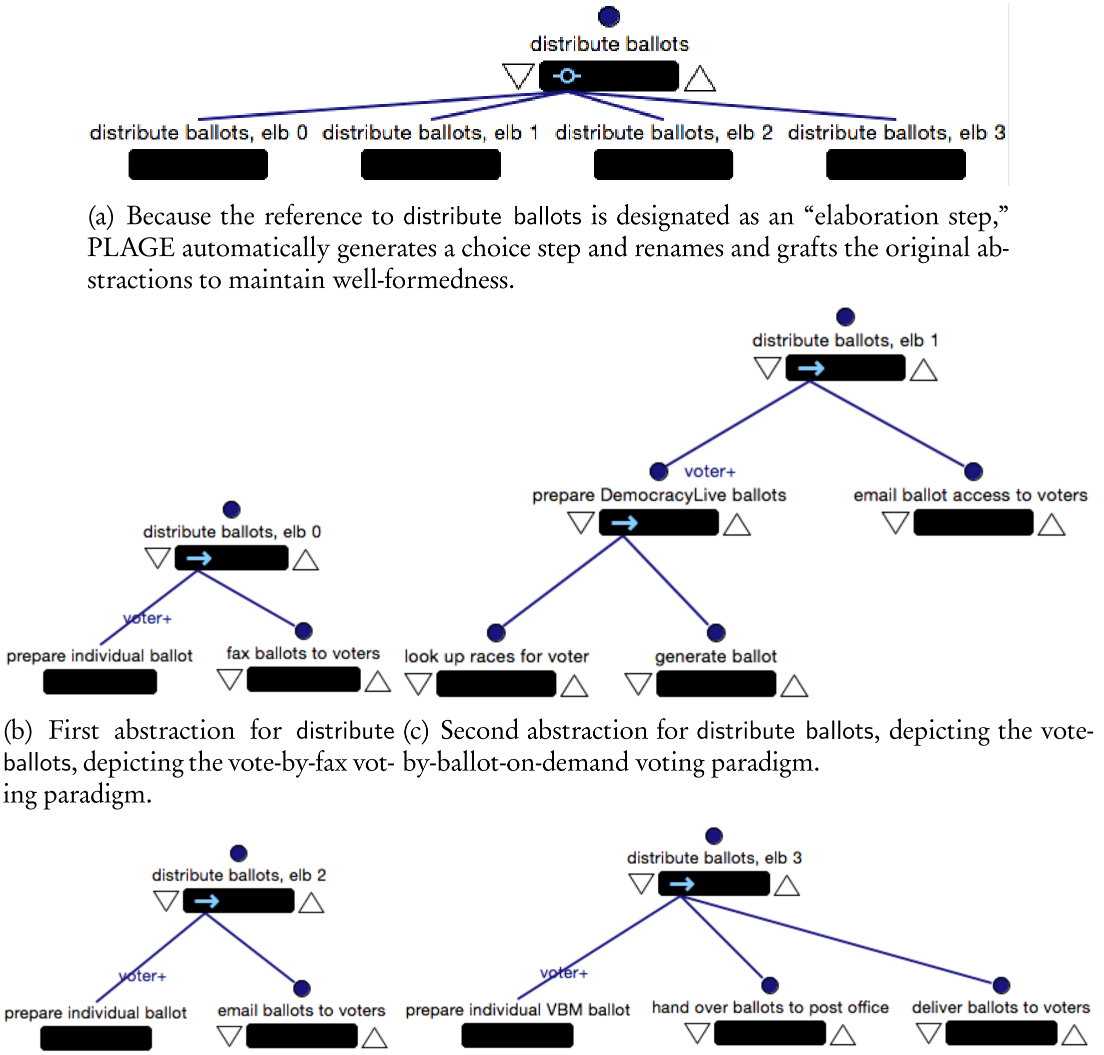


Figure 5.2. The subprocess for preparing and sending out ballots to voters. The distribute ballots reference in this diagram has been designated an elaboration step, and its corresponding abstractions are shown in Figure 5.3

The second substep of the vote remotely step in Figure 5.1 is mark and return ballot, a step that will be executed once for each agent of type Voter known in the system, as indicated by the agent + notation on the edge leading to the step as well as its interface declaration specification. This step is also designated an elaboration step in the process definition, and Figure 5.4 contains all the different ways for a voter to perform it, depending on which remote voting scheme is being used.

The third substep of vote remotely, collect ballots is also an elaboration step, and the last high-level variation point in this case study. The different ways for election official teams to collect and process ballots from voters are specified in Figure 5.5.

Currently, there are few United States jurisdictions we are familiar with that employ voting over the internet or more generally remote voting through the use of non-standard ballots that are not printed on ballot stock. In these jurisdictions, all ballots get duplicated or printed onto ballot stock before being counted. The ballots are then all counted using the same procedure as the one established for absentee, vote by mail (VBM) ballots. Our case study reflects this practice, and the last substep of the vote remotely step, count votes is shown in Figure 5.6. Note that this is an oversimplified subprocess, but it is used here instead of the full definition in order to facilitate reasoning about the abstraction elabo-


distribute ballots
distribute ballots, elb 0
distribute ballots, elb 1
distribute ballots, elb 2
distribute ballots, elb 3
distribute ballots, elb 1
voter+
prepare DemocracyLive ballots
email ballot access to voters
distribute ballots, elb 0
voter+
prepare individual ballot
fax ballots to voters
look up races for voter
generate ballot
distribute ballots, elb 2
voter+
prepare individual ballot
email ballots to voters
distribute ballots, elb 3
voter+
prepare individual VBM ballot
hand over ballots to post office
deliver ballots to voters

(a) Because the reference to distribute ballots is designated as an "elaboration step," PLAGE automatically generates a choice step and renames and grafts the original abstractions to maintain well-formedness.

(b) First abstraction for distribute ballots, depicting the vote-by-fax voting paradigm.

(c) Second abstraction for distribute ballots, depicting the vote-by-ballot-on-demand voting paradigm.

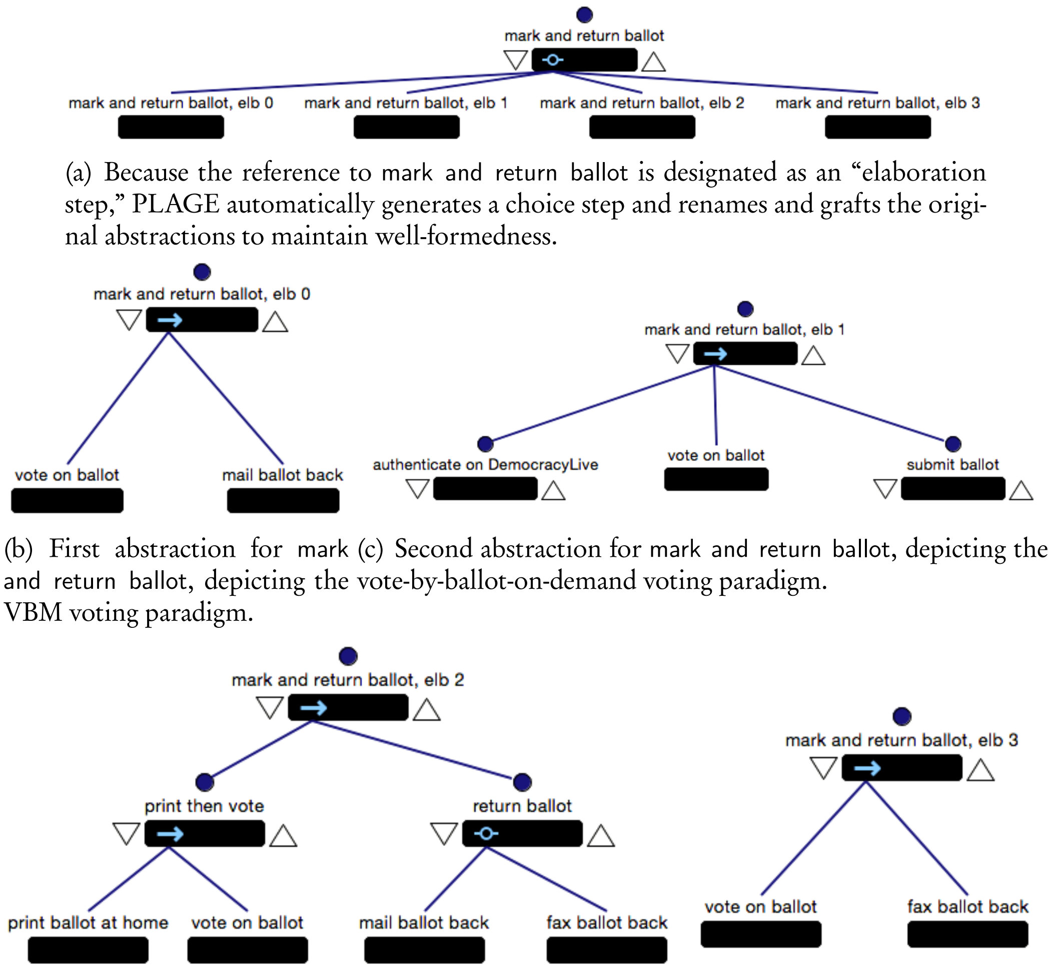
(d) Third abstraction for distribute ballots, depicting the vote-by-email voting paradigm.

(e) Fourth abstraction for distribute ballots, depicting the vote-by-mail (VBM) voting paradigm.

Figure 5.3. Abstractions and automatically generated elaboration choice step within the distribute ballots process family. The complete specifications of the subprocesses for ballot preparation, i.e., prepare individual ballot for digital transmissions and prepare individual VBM ballot for the VBM counterpart are shown in Figure E.1 in Appendix E.

rations that are employed earlier in the process rather than complicating the results with extraneous steps that are shared among all variants.

Note that different activities for each variation point may involve different agents depending on the variant selected. This information is presented fully in the family specifi-


mark and return ballot
mark and return ballot, elb 0
mark and return ballot, elb 1
mark and return ballot, elb 2
mark and return ballot, elb 3
mark and return ballot, elb 0
vote on ballot
mail ballot back
mark and return ballot, elb 1
authenticate on DemocracyLive
vote on ballot
submit ballot
mark and return ballot, elb 2
print then vote
return ballot
print ballot at home
vote on ballot
mail ballot back
fax ballot back
mark and return ballot, elb 3
vote on ballot
fax ballot back

(a) Because the reference to mark and return ballot is designated as an "elaboration step," PLAGE automatically generates a choice step and renames and grafts the original abstractions to maintain well-formedness.

(b) First abstraction for mark and return ballot, depicting the VBM voting paradigm.

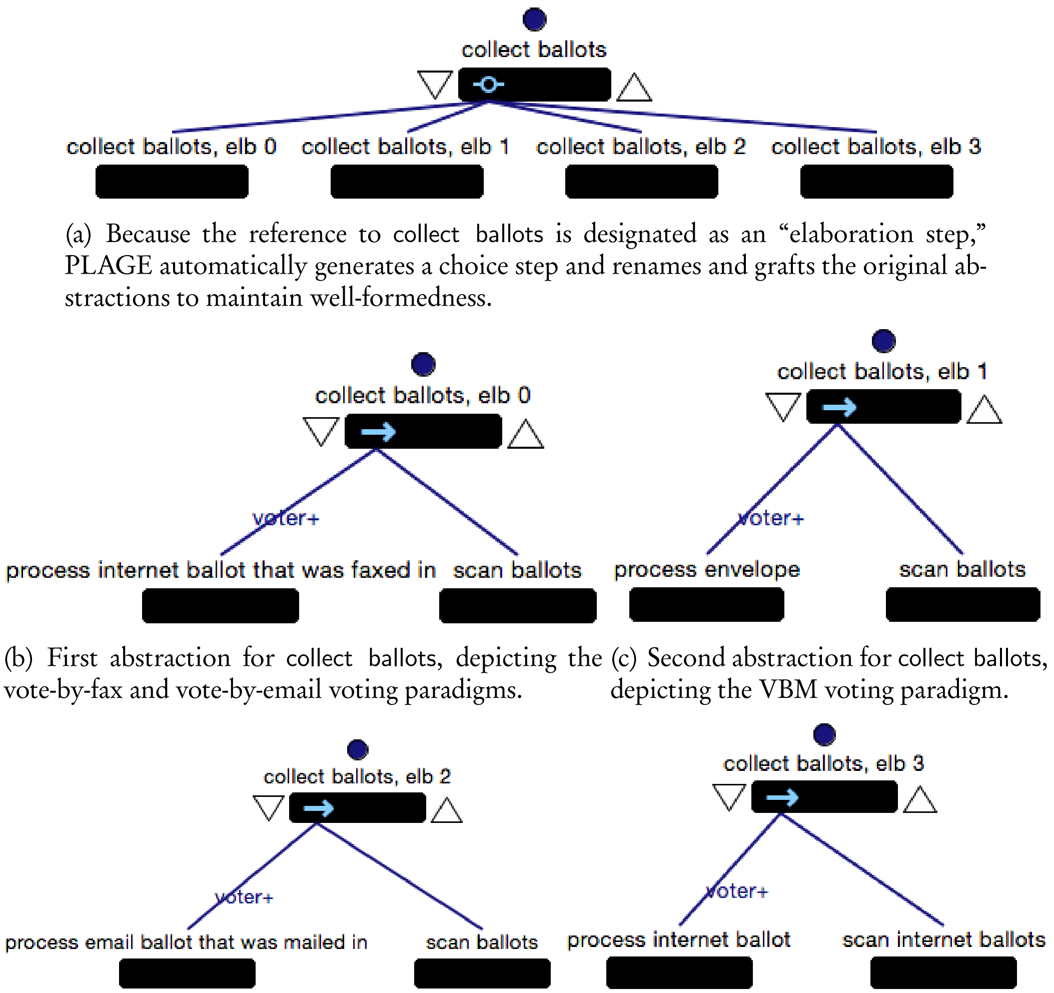
(c) Second abstraction for mark and return ballot, depicting the vote-by-ballot-on-demand voting paradigm.

(d) Third abstraction for mark and return ballot, depicting the vote-by-email voting paradigm.

(e) Fourth abstraction for mark and return ballot, depicting the vote-by-fax voting paradigm.

Figure 5.4. Abstractions and automatically generated elaboration choice step within the mark and return ballot process family. These abstractions show agent variation, as all activities within an abstraction are performed by a unique voter agent. The complete specifications of the subprocesses for ballot transmission, i.e., mail ballot back and fax ballot back are shown in Figure E.2 in Appendix E.

cation in Appendix E. For example, ballot distribution is always performed by different teams of election officials and is therefore considered functional variation, but the ballot marking is the responsibility of every individual voter and can therefore be presented as service or agent behavior variation. Further, service variation encoding the different ways in which agents could corrupt the integrity of elections–either through malicious intent



(a) Because the reference to collect ballots is designated as an "elaboration step," PLAGE automatically generates a choice step and renames and grafts the original abstractions to maintain well-formedness.

(b) First abstraction for collect ballots, depicting the vote-by-fax and vote-by-email voting paradigms.

(c) Second abstraction for collect ballots, depicting the VBM voting paradigm.

(d) Third abstraction for collect ballots, depicting the vote-by-email voting paradigm.

(e) Fourth abstraction for collect ballots, depicting the vote-by-ballot-on-demand voting paradigm.

Figure 5.5. Abstractions and automatically generated elaboration choice step within the collect ballots process family. The complete specifications of the subprocesses for ballot processing, e.g., process envelope or process internet ballot are shown in Figure E.3 in Appendix E.

or through accidental misperformance can also be specified for the different subfamilies, as outlined in Table 5.2.

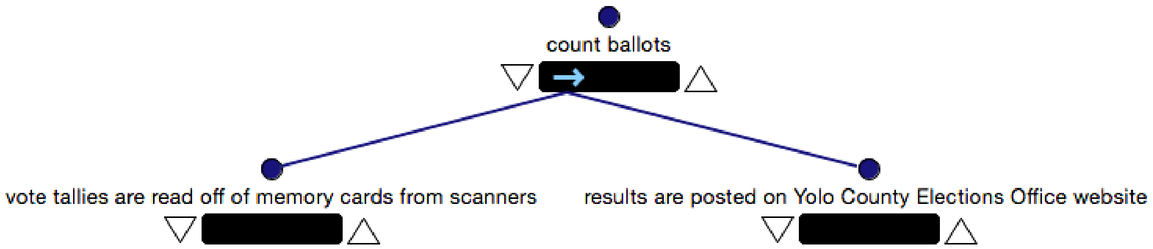

count ballots
vote tallies are read off of memory cards from scanners
results are posted on Yolo County Elections Office website

Figure 5.6. The subprocess for `count ballots`. As per existing practices, all ballots are counted using an identical procedure, but electronic ballots are reproduced onto ballot stock as paper ballots first.

Process Context Agents	Accessible		Standard	
Voting Machine	DRE Dial		Digital Scan	
	WYSIWYG	Skim votes	Scan correctly	Skim votes
Voter	Select and confirm on touch screen		Make selections on paper ballot	
	Vote as intended	Tamper with machine or paper trail	Vote as intended	Stuff ballots/impostor
Election Official	Read results after election, keep paper trail as official record for audit		Scan ballots during election, do vote tabulation after election.	
	Read results as expected	Tamper with paper trail or misreport results	Tabulate ballots and votes as expected	Tamper with ballots or misreport results

Table 5.2. Variation points for agent behavior variants in the Remote Voting process family.

5.2 Generation of Process Families

We apply the PLAGE conceptual framework and specifically the Little-JIL Elaborator toolset to model observed variation in the election domain.

We would like to evaluate whether the framework is suitable for generating the kinds of families we have observed in our work in elections so we focus on a real-world example of a well-defined large family with many variants—remote voting. The goal of applying the framework to a realistic case study is to ensure that the described variability management mechanisms are sufficient for accommodating the different dimensions of variation naturally encountered in the case study domains. We can generate solution-level families of variants that correspond to problem-level variation requirements within the case study domains and evaluate the suitability (in terms of effectiveness, clarity, ease of use, and other qualitative metrics) for generating new solution-level variants. This may also allow us to outline the general kinds of families that the framework seems suitable for, and other kinds of families that are not well accommodated.

In addition to the qualitative attributes described above, there are also certain quantitative metrics that we consider in this evaluation. For example, performance is a significant concern, and following are some questions we would like to evaluate quantitatively. When generating the families of processes outlined above, it would be interesting to keep track of how many variants a typical family has as a metric of the family's size. Note that depending on the number of elaboration steps within the core process, the answer to this question grows exponentially in the number of variants at each elaboration step. For example, in this case study there is no nesting of variation and there are only a dozen different abstractions, but this results in over a hundred overall combinations of variants. Therefore, a more suitable metric may be how many unique steps are declared within each process instance or family. The request language of the PLAGE framework has been designed to be flexible and precise enough to support easy answers to these questions through the application of different queries (for example, a query for abstraction elaborations that does not specify any additional constraints in addition to the step name would return all suitable elaborations and thus give an instant answer to the question how many elaborations this elaboration step has). Note that the amount of memory a process family needs includes the storage necessary for the common core, as well as all abstractions. Since the common core is shared, this provides some memory savings, but at the same time the management of abstractions may lead to significant overheads.

Since memory storage is rarely a bottleneck when it comes to storing Little-JIL process definitions, perhaps more importantly than memory, we may consider timing to evaluate the performance of the system. Specifically, we are interested in the overhead and performance issues associated with generating instances and accessing individual elaborations when new families are generated and how this size later affects the speediness of analysis.

Note that in the case of intersecting families (i.e. families that use both process fragment and agent behavior abstraction elaborations), there are clearly additional benefits to be gained in addition to the potential for more effective and widely-applicable reuse. Namely, the organization of variants into a formal family may provide better organization and management, apart from the quantitative performance gains. For this case study, we consider two functional variation subfamilies (one for ballot distribution and another for ballot collection) intersecting with a service variation subfamily (for a voter marking and returning a ballot). We then use another case study for performance comparisons.

We generated the following artifacts for fault tree analysis:

- The remote voting process family including all four variants for remote voting grafted onto all elaboration steps. This process family is henceforth referred to as ALL.
- An instance of the remote voting process for vote by mail (VBM).
- An instance of the remote voting process for vote by fax.
- An instance of the remote voting process for vote by email .
- An instance of the remote voting process for vote by ballot-on-demand (BoD).
- A complete process family modeling an extended case study of the one presented in [67], henceforth referred to as SPLC extended. Note that although there is less variation in this case study, the process is also larger and has a significant amount of branching, which allows us to really tax the analysis engines and ensure that they have the capacity to scale.

We generated the following additional artifacts for finite-state verification:

- The remote voting process family with the vote-by-fax variant removed (i.e., including only the variants for VBM, email, and BoD), referred to as ALL-Fax.

- The remote voting process family with the vote-by-fax and vote-by-ballot-on-demand variants removed (i.e., including only the variants for VBM and email), referred to as ALL-Fax-BoD.

- The remote voting process family with the vote-by-fax, vote-by-ballot-on-demand, and vote-by-email variants removed (i.e., including only the variant for VBM). This was hypothesized to be equivalent to the VBM instance, with the exception that it contains three additional choice steps inserted by PLAGE while the Little-JIL Elaborator was creating the original family. We confirmed that the results were consistent with the VBM instance as hypothesized, so this artifact is omitted from the comparative analysis below for clarity.

All sets of results obtained are presented in the following section. We experienced no delay in the generation phase for any of the families generated. Every single resolution activity completed within a small fraction of a second and appeared instantaneous from the point of view of the user. All of these test cases included a handful of abstractions to resolve, each containing a handful of interface declarations and binding specifications to be considered for transfer. Much larger numbers of abstractions to graft or other special cases that may lead to noticeable delays are discussed under Limitations.

5.3 Analysis of Process Families

Analysis is the second major goal addressed in this dissertation, and can be handled at either the problem or at the solution level. We focus on the solution level, which contains concrete implementations of process families and their comprising variants that could be studied, executed, analyzed, and so on. The variation management approach embodied by the Little-JIL Elaborator should be amenable to family-level analysis, and some corresponding analysis techniques that can reason about variation are outlined for evaluation below.

The most fundamental question we are interested in answering is whether the framework is suitable for performing the kinds of analyses we are interested in. Specifically, there are several kinds of analyses that would be considered useful in the context of families, such as making assurances or proving properties and theorems about entire families, as opposed to single process variants. Several concrete scenarios to be evaluated from the domain of elections are provided below.

In addition to asking whether the framework is capable of supporting the kinds of analysis we are interested in, these analyses would only be truly useful if their performance is reasonable and can provide results in a time frame comparable to the time it would take to analyze individual family members one by one. Thus, ideally, if a process family is fully elaborated to explicitly generate all legal, well-formed variants, and these variants are individually analyzed for adherence to certain properties, then performance gains may be made possible by optimizing the analysis so that when it is applied to a process family, it would reuse results whenever possible and take advantage of the parts of the process that variants share in common, or parts of the process that do not contain relevant events. In addition to ensuring that the analysis results are correct (i.e. the results for the family coincide with individual results obtained for variants), the analysis of entire families of variants may be more efficient that individually analyzing every variant within that family in order as outlined above. The way to achieve both efficiency and improved reasoning in this evaluation would be to take full advantage of the way that some solution-level variation encountered can be safely abstracted at the elaboration step level, and that the "common core" should not be analyzed multiple times; only differences would have to be considered carefully. Most analysis frameworks would do this automatically if the family were modeled using an appropriate representation. Additionally, in the case when a family of processes is being analyzed for a property whose events do not occur in a large portion of the abstraction elaborations, significant optimizations can be accrued from ignoring those elaborations as they cannot contribute to the property being violated. Of course, care must be taken not to avoid cases when an abstraction may not contain an event of interest itself, but may change the control flow in a way that impacts the overall results.

5.3.1 Fault tree analysis

We first present the results from running fault tree analysis on the six artifacts presented in the previous subsection–the five NIST remote voting artifacts including four instances and the family generated from considering all of them at once, and the SPLC extended use case. The hazard under consideration for the NIST case study is the step results are posted on Yolo County Elections Office website receiving the wrong voteCount artifact (of type VoteCount.java, specified as a JavaBean using the eponymous artifact mode in Visual-JIL) as input. The hazard under consideration for the extended SPLC case study is the step report vote totals to Secretary of State receiving the wrong tallies artifact, specified as a JavaBean of type VoteCount.java (the two hazards were selected to be similar in semantic meaning for the two case studies—in each case, the wrong totals have been computed after vote count, and although the dataflow specifications differ, they share an underlying set of artifacts and those artifacts propagate throughout the different election processes in similar ways). Each set of experiments was performed on a 3GHz dual-core Intel Core i7 processor with 16GB of physical RAM. All experiments were run under an Eclipse virtual machine (VM) with 4,096MB–16,384MB memory allowance. For the fault tree analyses performed, the Little-JIL Analysis Toolset translator was also given a 4,096MB–16,384MB memory allowance.

Table 5.3 presents the scalability results. Each row corresponds to a metric being observed, respectively:

- Size (# unique steps): the size of the process or process family under consideration, as determined by the number of *unique* step *declarations* used.

Subject Metric	*ALL*	*VBM*	*Fax*	*Email*	*BoD*	*SPLC extended*
Size (# unique steps)	79	41	29	41	24	98
TFG Build Time (s)	17.307	3.350	2.157	4.109	1.738	116.294
# FTA Events Original	705	331	169	303	97	914
# FTA Events No Duplicates	580	282	134	243	79	650
# FTA Events Optimized	154	76	32	68	14	225
# FTA Gates Original	645	302	155	276	90	650
# FTA Gates No Duplicates	524	255	122	219	73	575
# FTA Gates Optimized	98	49	20	44	8	150
# Minimal Cut Sets	70	61	12	26	8	85
# Minimal Cut Set Groups	44	34	9	18	5	45
# Single Points of Failure	10	3	4	7	2	0

Table 5.3. Fault tree analysis results for the NIST Remote Voting process family (ALL), as compared to each process instance and an orthogonal evaluation, SPLC extended.

- TFG Build Time (s): the number of seconds it took the translator to build the Trace Flow Graph (TFG) for this analysis problem.
- # FTA Events Original: the total number of events (simple or decomposable events) in the generated fault tree.
- # FTA Events No Duplicates: the new number of events in the fault tree, after duplicate events are removed.
- # FTA Events Optimized: the final number of events, after optimization, that get displayed as a fault tree (which is a directed acyclic graph (DAG), and no longer a tree) in which one event node may now correspond to more than one event of the same kind in the trace as long as it does not introduce a cycle. Inferred step states also get removed during optimization.
- # FTA Gates Original: the total number of gates (AND, OR, or NOT gates that describe how events are related) in the generated fault tree.

- # FTA Gates No Duplicates: the new number of gates in the fault tree, after duplicate gates are removed.

- # FTA Gates Optimized: the final number of gates, after optimization, in which one gate may now correspond to more than one gate from the previous list as long as it does not introduce a cycle in the fault tree.

- # Minimal Cut Sets: the minimal sets of events that may lead to a hazard occurring.

- # Minimal Cut Set Groups: in previous work, Phan et al. proposed a presentation of MCSs for easier exploration, where similar cut sets (for example differing by whether an exception of interest is thrown or not thrown) get grouped and numbered as related. This is the total number of such groups of MCSs found.

- # Single Points of Failure: total number of MCSs of size one.

Figure 5.7 visualizes the relationship between the time it took the translator to build a fault tree and the number of process steps in the process or process family for which that tree is being built. Note that the time is plotted on a logarithmic scale, base 10.

Figure 5.8 plots the relationship between how large a process or process family is, in terms of number of steps, and how large a resulting fault tree becomes. The number of nodes in a fault tree corresponds to the sum of its gate and event elements (# FTA Events Original + # FTA Gates Original from Table 5.3).

5.3.2 Finite-state verification

We present eight sets of results in this section, obtained from running finite-state verification on the same six artifacts considered in the previous subsection and the two additional partial families obtained from iteratively removing first the vote-by-fax variants and consequently also the vote-by-ballot-on-demand variants from the ALL process family. Each set of experiments was performed on a 3GHz dual-core Intel Core i7 processor with 16GB of physical RAM. All experiments were run under an Eclipse virtual machine

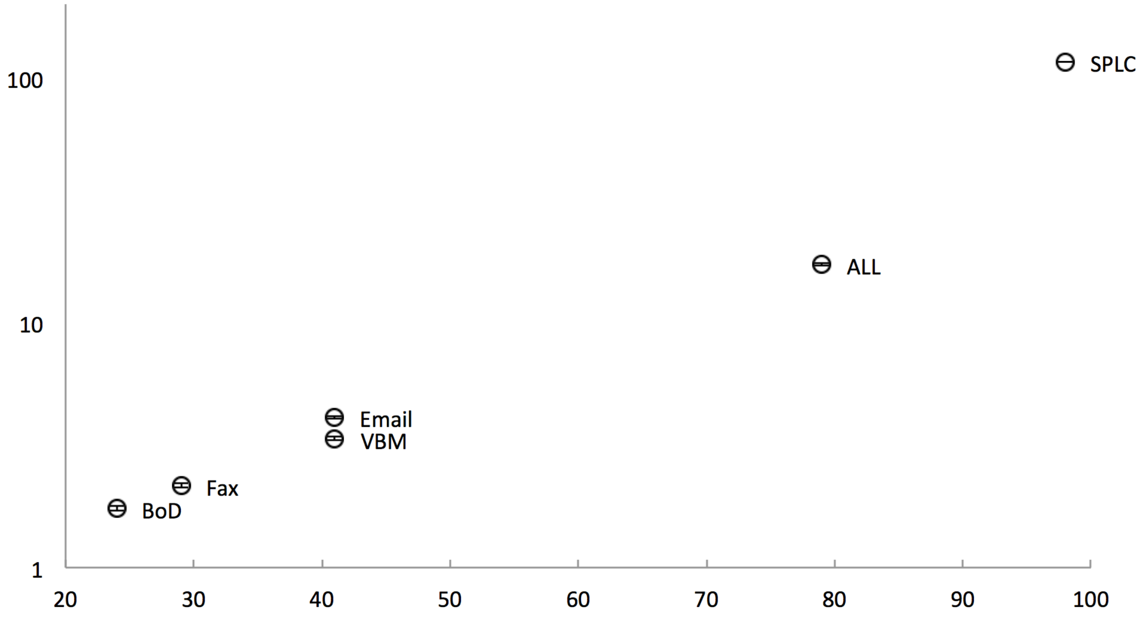


Figure 5.7. The time (in seconds, on a log_{10} scale) it takes to build a fault tree along the y axis versus the size (in number of steps) of the corresponding process or process family along the x axis.

(VM) with 4,096MB–16,384MB memory allowance, and these finite-state verification results were also obtained by allocating 4,096MB–16,384MB of memory to the Little-JIL Analysis Toolset translator.

The property we use loosely specifies that the ballot should correctly record the voter's intent. Voter "intent" is a very controversial term among election researchers; therefore we define "intent" simply to mean that if the voter made a selection on the ballot and then cast it, then the selection actually made by the voter is the voter's "intent". It is therefore vital that no changes can be made to that ballot after it is case and before it is counted. We determine that there are three events of interest to us, a *voter marks ballot* event, an *insider marks ballot* event, and a *count ballots* event. We use PROPEL (PROPerty ELucidator [74]), a software tool that helps users formalize all the details associated with a certain high-level requirement, to precisely define this property. PROPEL gives users several different, editable views of the same underlying property so users may edit

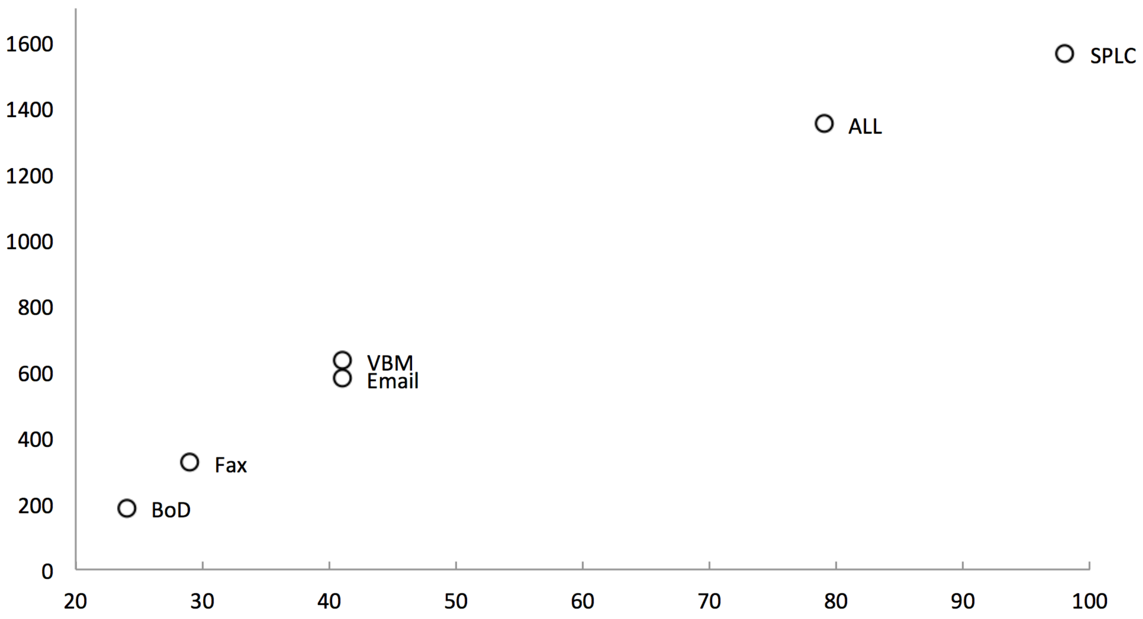


Figure 5.8. The total number of nodes (events + gates) in an unprocessed, unoptimized fault tree along the y axis versus the size (in number of steps) of the process or process family for which the fault tree was built along the x axis.

a structured English description, a finite state automaton (FSA), or answer a series of questions organized hierarchically until the property has been completely and correctly specified.

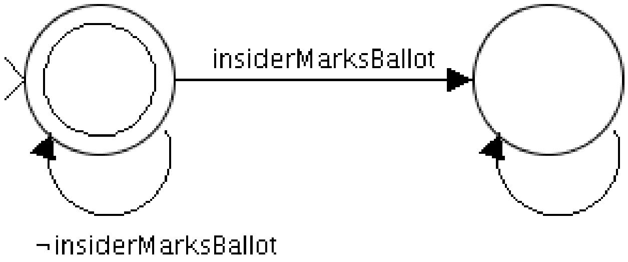
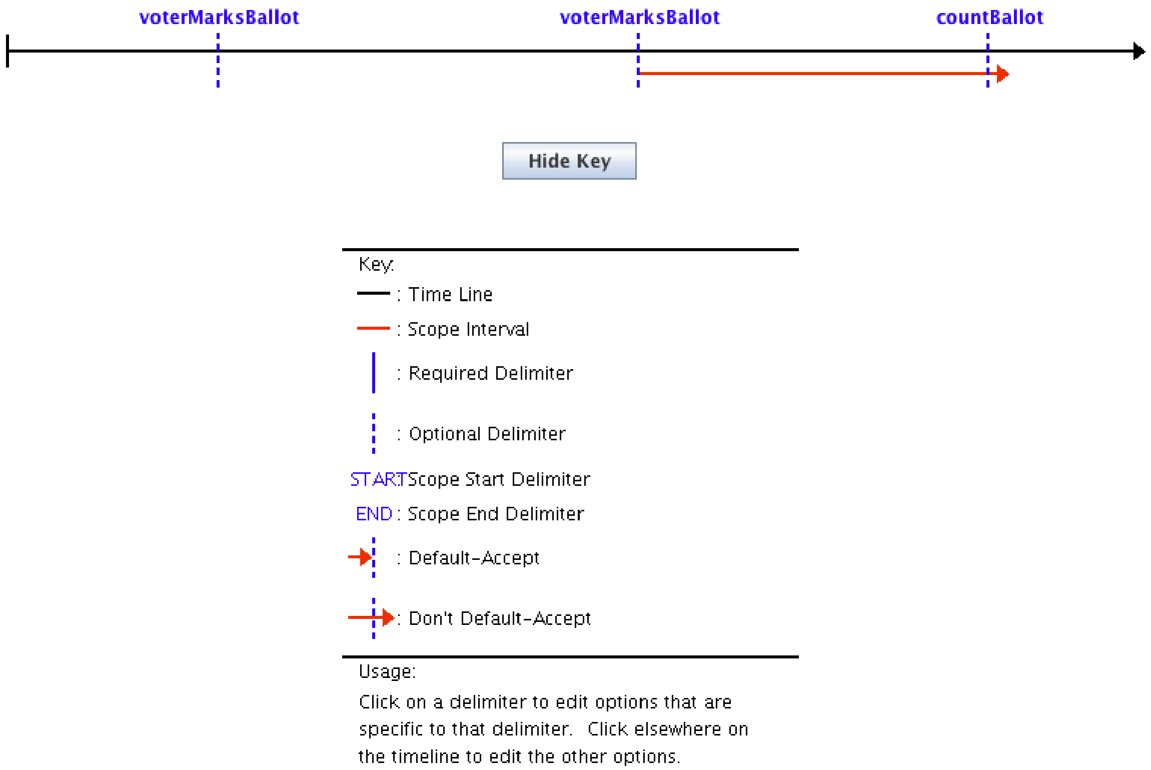


Figure 5.9. FSA for the property "After a *voter marks ballot* event, no *insider marks ballot* event can occur until *count ballots* occurs."

Because there are three events of interest in our property, we choose to define the property as a single event, *insider marks ballot*, that should not be allowed to occur within a predefined scope, in this case from the last occurrence of *voter marks ballot*



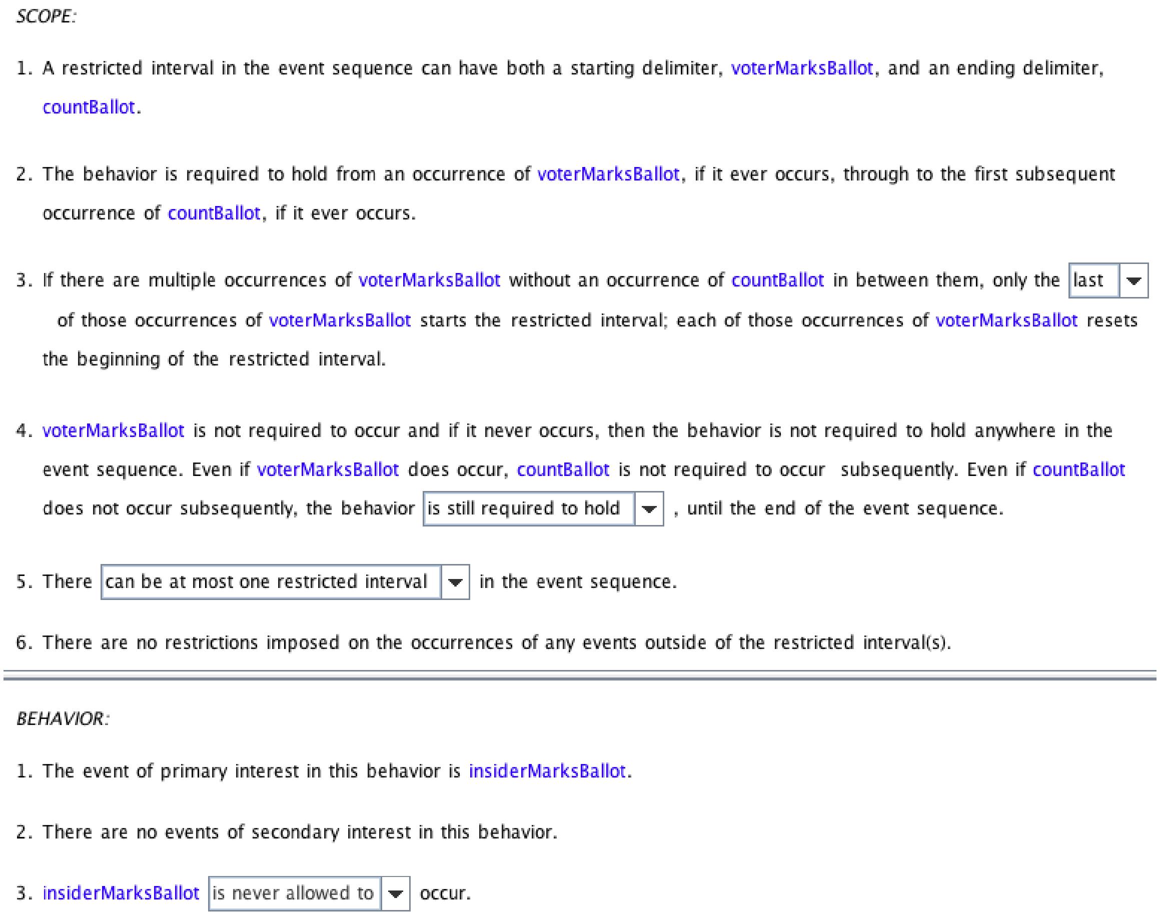
Key:
— : Time Line
— : Scope Interval
| : Required Delimiter
⋮ : Optional Delimiter
START:Scope Start Delimiter
END : Scope End Delimiter
→⋮ : Default-Accept
—⋮→: Don't Default-Accept

Usage:
Click on a delimiter to edit options that are specific to that delimiter. Click elsewhere on the timeline to edit the other options.

Figure 5.10. FSA for the property “After a *voter marks ballot* event, no *insider marks ballot* event can occur until *count ballots* occurs.”

until *count ballots* occurs. The FSA representing that *insider marks ballot* should not be allowed to occur is shown in Figure 5.9, while Figure 5.10 indicates the scope during which this property must hold. Finally, the property is explained in disciplined natural language in Figure 5.11. The question trees used for the scope and behavior definition are included in Appendix F for completeness.

Each of the two nodes in Figure 5.9 represents a state in the FSA. The double circles on the start state (on the left, denoted by an open arrowhead) indicate an accepting state, and the single circle on the state on the right indicates a non-accepting state. Arrows are state transitions, and each is annotated with a corresponding “event” that triggers it. When a transition is triggered, the current state is updated to the target state of that transition. There are many different events that may occur in an election, but each property is typically only concerned with a small subset of them (the property alphabet). In this



SCOPE:

1. A restricted interval in the event sequence can have both a starting delimiter, voterMarksBallot, and an ending delimiter, countBallot.
2. The behavior is required to hold from an occurrence of voterMarksBallot, if it ever occurs, through to the first subsequent occurrence of countBallot, if it ever occurs.
3. If there are multiple occurrences of voterMarksBallot without an occurrence of countBallot in between them, only the last ▾ of those occurrences of voterMarksBallot starts the restricted interval; each of those occurrences of voterMarksBallot resets the beginning of the restricted interval.
4. voterMarksBallot is not required to occur and if it never occurs, then the behavior is not required to hold anywhere in the event sequence. Even if voterMarksBallot does occur, countBallot is not required to occur subsequently. Even if countBallot does not occur subsequently, the behavior is still required to hold ▾ , until the end of the event sequence.
5. There can be at most one restricted interval ▾ in the event sequence.
6. There are no restrictions imposed on the occurrences of any events outside of the restricted interval(s).

BEHAVIOR:

1. The event of primary interest in this behavior is insiderMarksBallot.
2. There are no events of secondary interest in this behavior.
3. insiderMarksBallot is never allowed to ▾ occur.

Figure 5.11. FSA for the property “After a *voter marks ballot* event, no *insider marks ballot* event can occur until *count ballots* occurs.”

example the behavior specification of this property alphabet consists of insuring that if **insider marks ballot**, indicating that a mark was made by an insider to the election process (someone other than the voter, such as a rogue election official or a compromised voting or counting machine), the FSA will transition to a non-accepting violation state. An FSA used for verification must be deterministic and total, meaning that for each state every event in the property alphabet must occur on exactly one transition leaving the state. If an event occurs that causes the sequence of events to be unacceptable, then the current state is updated to a non-accepting state, called the violation state from which all transitions are self-loops, causing the FSA to remain in that state until the process terminates. In Figure 5.9, the violation state is show on the right, and it has an unlabeled self-loop, indicating that any event that occurs in the violation state will indeed result in the FSA remaining in that state.

The scope definition in Figure 5.10 supplements the FSA by defining the scope within which it should be considered. Note that the scope begins only after the *last* occurrence of **voter marks ballot**, indicating that the voter might have the chance to fix a discrepancy or marking error before the ballot is cast. Once the **voter marks ballot**, **insider marks ballot** should never be allowed to occur until **count ballots** occurs, indicating the end of the scope interval.

A sequence of events that would drive this property to the violation state would indicate a malicious agent compromising the system. In order to be able to perform finite state verification and check if a given process or process family satisfies this property, we must determine the correspondence between the events comprising the property's alphabet and specific step states in specific variants within each of the election process families under consideration. This correspondence is explicitly and manually defined through a binding between process step states and property events.

Property Event	Step Name
voter marks ballot	vote on ballot
insider marks ballot	duplicate ballot on ballot stock
count ballots	vote tallies are read off of memory cards from scanners

Table 5.4. Bindings between steps (all steps are specified to be in state COMPLETED) in the Little-JIL process family definition for the NIST remote voting case study and events in the property alphabet for the property presented in Figure 5.11.

Table 5.4 lists the bindings between the property alphabet events and steps in the NIST case study process family. The event **voter marks ballot** is bound to the vote on ballot step. This means that when an invocation of this step completes successfully (depending on which variant is chosen), the event **voter marks ballot** occurs. Note that all variants references a single definition of this step, passing different kinds of ballots as appropriate for a specific variant, so we can use this binding across the entire family and for each variant, without having to specify correspondences to multiple step declarations. Similarly, the event **insider marks ballot** is bound to the step duplicate ballot on ballot stock and **count ballots** is bound to the vote tallies are read off of memory cards from scanners step, which occurs later during the counting phase of the NIST remote voting election process.

Subject Metric	*ALL*	*VBM*	*Fax*	*Email*	*BoD*	*ALL-Fax*	*ALL-Fax-BoD*	*SPLC extended*
Size (# unique steps)	79	41	29	41	24	69	55	98
TFG Build Time (s)	0.845	0.371	0.469	0.622	0.480	0.709	0.582	3.937
Verification Time (s)	1.154	0.464	0.577	0.764	0.595	0.877	0.712	12.565
# TFG Nodes	84	6	16	37	12	59	38	613
# TFG Edges	1,255	5	19	268	13	530	155	67,451
# TFG MIP Edges	1,148	0	0	221	0	456	108	66,450
Violation Found?	yes	no	yes	yes	yes	yes	yes	yes

Table 5.5. Finite state verification analysis results for the NIST vote remotely process family, the different process instances, and families with some variants removed.

Table 5.5 presents the performance results from running the FLAVERS [31] FSV engine on each respective process or process family against the property presented in Figure 5.11 with pre-specified bindings as presented in Tables 5.4 and D.1 for the NIST remote voting case study and the extended SPLC case study, respectively. For each case when a property violation was found, the counterexample trace from FLAVERS is included along with the rest of the analysis evaluation artifacts in Appendices F and D. Each row in the table corresponds to a metric being observed, respectively:

- Size (# unique steps): the size of the process or process family under consideration, as determined by the number of *unique* step *declarations* used. For process families, this includes the automatically generated choice steps that elaboration steps resolve to.

- TFG Build Time (s): the number of seconds it took the translator to build the Trace Flow Graph (TFG) for this analysis problem.
- Verification Time (s): the time it took FLAVERS to confirm that the artifact being evaluated either satisfied the property or, in case it did not, to produce a counterexample trace showing how a violation could occur. This time includes the time to build the TFG from the previous column.
- # TFG Nodes: the number of nodes in the TFG.
- # TFG Edges: the number of edges in the TFG.
- # TFG MIP Edges: the number of May Immediately Precede (MIP) edges in the TFG.
- Violation Found?: whether the property held (*no* violation found) or was violated (*yes*).

Figure 5.12 visualizes the relationship between the time it took to run the verification (Verification Time in Table 5.5), in seconds, and the number of process steps in the process or process family being verified. Note that the time is plotted on a logarithmic scale, base 10.

Figure 5.13 plots the relationship between the verification time in seconds again, this time plotted against the total number of MIP edges in the TFG for the artifact being verified. Note that time is plotted on a logarithmic scale, base 10, and since $log_{10}(0)$ is not defined, there are no values for processes or process families that had 0 MIP edges in their TFGs (indicating no alternate interleavings of events). MIPs are also plotted on a logarithmic scale, base 10.

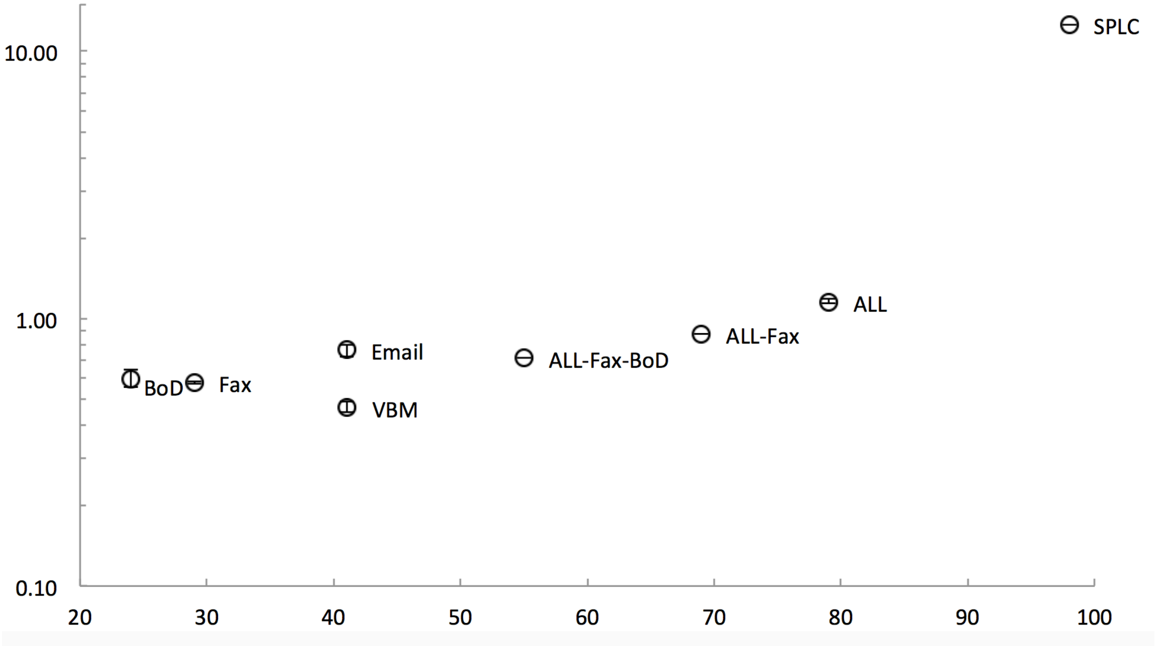


Figure 5.12. The time (in seconds, on a log_{10} scale) it took for the verification to run along the y axis versus the size (in number of steps) of the corresponding process or process family along the x axis.

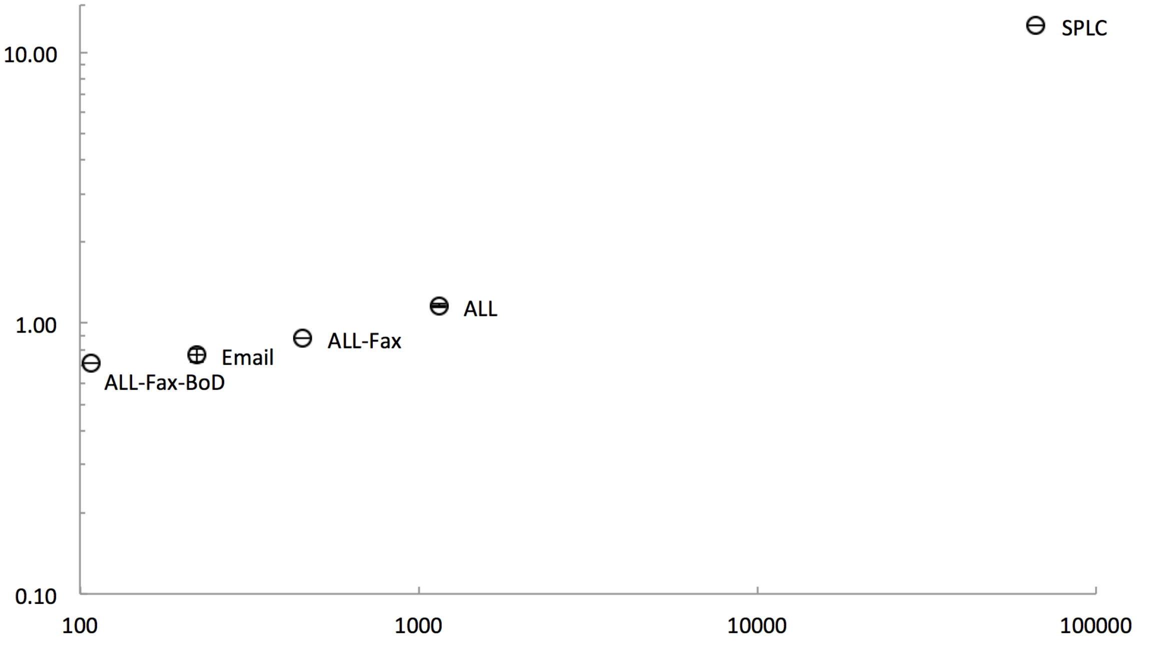


Figure 5.13. The time (in seconds, on a log_{10} scale) it took for the verification to run along the y axis versus the number of MIP edges (on a log_{10} scale) in the TFG of the corresponding process or process family along the x axis.

CHAPTER 6
DISCUSSION

6.1 Generation

Quantitatively, from a performance and scalability standpoint, all the generation activities performed for the evaluation presented in the previous chapter happened instantaneously from the point of view of the user. There are conceivable cases that would lead to observable latency, and they are discussed in the next chapter.

Qualitatively, there are two aspects of process family generation that crystallized as a result of the evaluation. The former is generality versus specificity, and the latter is aided variant selection. The issue of generality versus specificity has to do with how much or how little of a process family is encoded within that family's abstract common core (ACC), and what stays in the abstractions that will get grafted as elaborations later? For example, consider the mark and return ballot process subfamily presented in Figure 5.4. As noted earlier, every single elaboration refers to the same step declaration for the vote on ballot step, and the step's interface is defined so that it would accept both electronic and paper ballots, depending on the instance of the artifact with which it is invoked. Similarly, there are single definitions of commonly used fragments throughout the remote voting process, such as mail ballot back, or fax ballot back, two steps also occurring in the mark and return ballot variants. This modeling decision is an example of "pulling the variation down," intuitively, meaning there are now references to shared elements that repeat between abstractions. There are some reasons why this approach is appropriate—it is now harder to understand the ACC (meaning that the ACC for the remote voting process family is actually very small and does not communicate much detail about the family), but it is also much easier to see each abstraction elaboration and visualize the variation among them.

Given some technology limitations discussed in the next chapter, if we want to consider a whole family, this approach may allow for the exclusion of some false positives. From a usability perspective, ideally there may exist a separate repository of PLAGE modules to be used as "default" abstractions to elaboration steps referencing a commonly needed capability (such as vote on ballot).

Alternatively, we could instead choose to "push the variation up" meaning as much as possible goes into the ACC, and we create possibly more but smaller variants. In the mark and return ballot subfamily, looking at Figure 5.4, you might note that each abstraction elaboration could be summarized as a voter optionally performing some authentication or access step to obtain a ballot, invariably followed by vote on ballot, invariably followed by some method of ballot transmission. Therefore, this subfamily could instead be modeled as a part of the ACC, where the first (ballot access) and last (ballot transmission) steps become variation points where smaller elaborations can now be attached as appropriate. We might get certain qualitative advantages from making the common core bigger, as it may be easier to explain and learn from a communications standpoint, or make it easier to see the interactions between pieces for evolution and changes. Consequently, if we use an analysis technique that makes use of summaries, this approach might lead to some performance gains. We have not empirically evaluated which approach would provide better results, and the results presented in the previous section are all based on the former approach of "pulling variation down" and having fewer, larger elaborations to create variants with. This results in fewer variants, and, in turn, fewer possible interleavings for the analyzers to consider. Given the observed impact of MIP edges on FSV performance, the chosen modeling approach likely yielded faster performance.

The second issue that became apparent during evaluation was the importance of variant selection. Currently, the Little-JIL Elaborator grafts all matching abstractions as elaborations, and, after careful consideration, transfers over the appropriate interface declarations and bindings found in the union of the elaborating subprocesses' roots. In future work, it would be helpful if the PLAGE framework could suggest new variants that are, in some user-defined respect, better. In addition to generating the observed variants in elections as a "common core" process and a set of abstraction elaborations, it would also be interesting to observe if newly generated variants from the given abstractions could maybe suggest process improvements (through satisfying a given set of properties with fewer resources, for example). An evaluation scenario from the election case study domain follows.

In elections, there are different voting machines that may be needed in different contexts. This is a good example of agent behavior elaboration and would present an apt opportunity to explore the flexibility of the framework when it comes to requesting agent behavior specifications matching a certain set of desiderata. For example, the repository may contain many voting machines, but we may only be interested in the ones that can support a VVPAT. Additionally, this presents an excellent opportunity for exploring one of the future research directions, namely interface variation; for example, one voting machine may output a provisional ballot while another may output a regular ballot, and the framework would have to know how to handle the two different, but related, artifact types correctly.

6.2 Analysis

6.2.1 FTA

The fault tree analysis scalability results, presented in Table 5.3 hint at some interesting trends. In terms of performance, the analysis ran in near-realtime for all instances, and ranged from a few seconds for the entire NIST remote voting process family to a couple of minutes for the SPLC extended process family. The runtime invariably increased as the size of the process increased, as shown in Figure 5.7, and the trend appears exponential for the limited data sample we have.

The size of the fault tree grew in proportion to the size of the process being analyzed, as shown in Figure 5.8. The fault tree analyzer was able to generate a fault tree and calculate its corresponding MCSs even for the largest artifact, the SPLC extended process family, in under two minutes. The resulting fault tree has a total of 375 nodes, made up of 225 events and 150 gates, and is an unwieldy structure. Navigating a fault tree of this size manually to find a problem would be nearly impossible, especially given a hazard such as the one we considered, where a key artifact is incorrect toward the end of the process. With several variation points occurring earlier in the process, manually tracing a sequence or collection of steps within the fault tree that includes an elaboration is not feasible. The automatically calculated minimal cut sets become especially valuable for identifying problematic areas of the process. However, this large process family is based on a vetted in-person voting process, and the fault tree analysis did not identify any single points of failure, making interpretation of the results cumbersome and difficult.

Although the fault tree for the NIST process family had fewer single points of failure than the sum of its four variant instances (this is because some combinations of steps resulted in the same minimal cuts sets in general and single points of failure in particular), the results for the variant instances were easier to understand and interpret and may facilitate with the mitigation of single points of failure better than a conglomerate. Specifically, an electronic ballot or a paper ballot that was printed at home and mailed or faxed in has to be duplicated onto ballot stock before being counted, and that was identified as a single point of failure for every variant that employed this step (i.e. email and fax ballots that were faxed or mailed in). This was also identified as a single point of failure (twice, with different traces) for the process family, but the tree was both harder to navigate and understand.

6.2.2 FSV

Table 5.5 summarizes the scalability experiments results for the FSV analysis done with FLAVERS. There are more artifacts being evaluated here because the original NIST family did not satisfy the requirement under consideration, so we attempted to remove offending variants and iteratively reverify the resulting, smaller family until the property had been satisfied. Unfortunately, given that the binding indicate that the step duplicate ballot on ballot stock corresponds to the event **insider marks ballot**, all variants except VBM are bound to violate the property, and the additional experiments confirmed this. This step was chosen for this family because it was already identified as a single point of failure during the fault tree analysis, and also because it seems like a prime candidate for a place in the process where a corrupt insider could in fact tamper with or modify a ballot.

We have previously pointed our that FTA and FSV can be used as complementary approaches, and this is an application of that idea. To move the integration even further, it may be useful to consider an approach for automating the bindings specification, or at least providing semi-automated guidance to a process developer. When analyzing process families in particular, but also process instances in general, deciding what step state to bind to what property event can often be a complicated decision. In the case of process families, it is plausible that bindings may be omitted because the process developer forgot that a certain elaboration in another variant may have similar implications. Similar to the property templates, we may consider the application of binding templates, that specify for example that every step that could modify an artifact of type ballot and requires its agent to have the capability to carry out election official duties ought to be bound to the event **insider marks ballot**. Automating the binding process in such a way may results in more false positives, but would also make process family analysis faster and easier to execute.

Figure 5.12 visualizes the verification time charted against the size of the process. As with fault tree generation, the larger the process was, the longer it took to run the verification. However, because step interleavings are an integral part of identifying which traces through a process are safe and which are not, the process size was not the determining factor for verification time. This can be clearly seen in the figure by comparing the data points for the vote-by-mail and vote-by-email process instances. At 41 unique step declarations, the two processes are of equivalent size, but verification for the email process took 168% as long, on average, as the VBM process. The main different is that the vote-by-email process has some 221 MIP edges in its TFG (these are due to the fact that the voter may choose whether to return their ballot by mail or fax, which introduces different interleavings), while VBM has none (as all steps are strictly sequential in that variant instance).

This relationship is more apparent in Figure 5.13, which shows how the verification time increases with respect to the number of MIP edges in the TFG for an artifact being analyzed. Both axes are plotted on logarithmic scales because the number of MIP edges in a TFG increases very rapidly with more possible interleavings (either due to a larger process or to the sheer number of variants), and, in turn, a large number of MIPs in the TFG leads to significantly longer verification times.

There are certain general properties about elections, such as “one voter one vote” that are universally applicable to all election processes in the United States. Such properties can be examined for entire families using techniques such as finite-state verification based on the presence of certain events from the alphabet (such as “voter casts ballot”) within every variant under consideration. Moreover, certain assurances about all variants within a family should be quite straightforward to make; trivially, variants not containing the event of interest are automatically safe to attach, whereas reasoning about variants that do contain the event in question may necessitate the application of increasingly complex compositional analysis techniques.

Note that there are also properties that are not globally applicable to the entire family. In the domain of elections, there are certain voting situations, such as processing provisional voters, that require different handling from the general process. In provisional voting, in fact, the authentication requirements are quite different from the authentication requirements for regular voters. Therefore, an organization into a process family would be extremely helpful for the management of such properties because the process developer could then use some sort of annotation to tag abstractions that ought to adhere to certain properties, and other ones that may be excluded. This may greatly reduce the hassle of dealing with false positives, or cases when a process appears to violate a requirement even though the specific circumstances prevent the requirement from being applicable (such as requiring a voter to produce certain identification if he or she is casting a provisional ballot).

Furthermore, there are properties that may be applicable only to a certain part of a process definition, or a certain agent responsible for executing steps within that process. This example is easy to illustrate with voting machines within the context of the election process. Related work by Conboy et al [23] shows how using overall process requirements, one can derive requirements for a specific agent. These agent-specific properties can clearly be useful for doing piecewise analysis on all agent behavior elaborations that may be bound to an elaboration step, thus saving time and space by constraining the search to a much smaller process. On the other hand, the reverse procedure can be used to consider all the agent behavior elaborations that are part of a family, and then try to derive process- or family-wide requirements as compositions of the individual constraints.

As outlined in the previous subsection, properties may apply to only certain members of a process family, and not the entire family. It would be interesting to try to quantify how many of the properties that we have modeled for the election domain appear to be applicable to entire families, and how many are context-dependent and need only be satisfied by some variants and not others. Additionally, such a quantification may be very useful for the generation of an ontology of global election properties, or properties that are not only applicable to an entire family, but also to multiple related families, intersecting families, and so on.

Whether a property applies to all variants within a process family, or only some of them, the system must be able to clearly identify variants that violate the property. Currently, the finite-state verification system, FLAVERS, attempts to construct a single counter-example to indicate how a process definition may violate a property. This would no longer suffice since it may only identify a single variant within the family that may violate the property, based on what events occur in the variants that have been grafted. Instead, it would be useful to be able to collect more extensive information about which variants led to a violation occurring so that a new family can be created with only the variants that were proven safe. Multiple counterexample traces would be useful both for identifying offending variants, as well as for diagnostics of how those variants allow for the property to be violated. Since generating counterexample traces is a difficult problem, it may be possible to use information about the family and its variants to guide the generation of particularly interesting counterexamples. Providing multiple counterexamples poses additional challenges from a user-experience perspective; for example, if a large percentage of a family's variants fails to meet the property, generating many counterexamples may take a very long time, leading to unacceptable performance. Moreover, a large number of failing variants may also indicate that the property is not framed correctly or that it should not be applied to all the variants it is being applied to, in which case it may be more appropriate to change the property specification or choose fewer variants for analysis. These considerations indicate that generating additional counterexample traces may be provided as an on-demand feature, to avoid the risk of inundating the user with multiple counterexamples, which may be overwhelming and confusing.

CHAPTER 7
LIMITATIONS

There are several threats to the validity of the approach presented in this dissertation, as well as some limitations stemming from the infrastructure environment and experimental platform within which the PLAGE framework resides and operates. We address these from two complementary viewpoints, namely modeling and analysis.

7.1 Modeling constraints

There are several constraints that ought to be considered when evaluating the effectiveness of the proposed modeling approach with respect to its fidelity to the real-world systems and families of systems it attempts to model and analyze. The three main expressiveness limitations we focus on are 1) the closed world assumption, 2) accommodating heterogenous elements, and 3) the configuration specifications.

7.1.1 Expressiveness limitations

A main assumption of the PLAGE framework and the Little-JIL Elaborator in particular is the existence of variants that can serve as elaborations for different families. This hinges on the specification of interchangeable modules that can be composed into different combinations, or can be used to elaborate different abstract cores. The specification of such modules, as the specification of most human-intensive process, requires a significant investment of time and effort both from the process developers as well as the domain experts. The domain experts are frequently capable of identifying parts of the abstract common core but need help enunciating it as a precise entity because they are usually not trained in modular thinking or abstract specifications. Once the abstract core is defined, domain experts can usually describe existing variants for each elaboration point, but may not be considering all possible variants, the precise consequences of selecting one variant over another, or how the interaction between variants at multiple elaboration points may affect the outcome of the process.

7.1.1.1 Closed-world assumption

Because we rely to a large extent on domain expert knowledge to create the different variants that we use for family generation, we operate within a closed world of prespecified possibilities. The prototype implementation of this approach uses Little-JIL for solution-level variation; although the language boasts impressive flexibility for coordinating both human and automated agents together without discriminating between the two, if a human agent is executing a certain step and their behavior is not constrained or defined within an agent behavior variant, that information still remains external to the process model. Furthermore, it is impossible to generate all possible elaborations for any elaboration point or set of attributes, so we can only consider variants that are common, plausible, and have been predefined by a process developer (hopefully based on some specification that the domain experts consider both important and interesting).

We suggest several possible research avenues that may alleviate this limitation, but these important questions remain beyond the scope of this dissertation. Although not currently used for this purpose, there are some existing approaches for code generation and code mutation that appear immediately applicable to the closed-world problem and may allow us to consider variants that were not initially specified by the developer or identified by the domain expert. For example, there is work on reverse-engineering hierarchical feature models of product lines both using genetic algorithms to arrive at a baseline [30, 53] as well as combining those with inference techniques based on search [18], and a set of provided examples [51]. These approaches are currently applied to a set of products in order to identify the commonalities among them, but may be easily extensible to derive new possible products by reversing the direction of the implication for the search-based approaches to generate new variants or to genetically mutate existing variants into related, but different, ones (similar to current product line mutation testing techniques, e.g. [29, 50]. Although program synthesis remains an open challenge, some approaches in software engineering already take advantage of corporate code bases and the vast global repository of open source projects to automatically correct errors in existing code like GenProg [36], or soon attempt to devise entirely new functionality [39]. Such approaches can be used to generate new variants or mutate existing variants by injecting faults to evaluate robustness and correctness, and different rules could be specified to enable the creation of variants along different dimensions of variation.

7.1.1.2 Heterogenous elements

Our approach aims to accommodate variation in human-intensive systems where humans are considered to be active participants within the system, not external users of the system. This means that, by definition, we have to model human behavior, which can be different from the prespecified protocol in unpredictable ways. In our prototype implementation, using the Little-JIL language, we handle some of the intrinsic variation through abstraction because human agents can execute steps in any way they deem appropriate as long as they provide the required outputs, or can throw exceptions if they cannot complete an activity. With this approach, some variation in human behavior remains external to the system. We could predefine some human behavior specifications as process fragments in Little-JIL to gain more control over how activities are carried out. We could also plug in agents through external applications by interfacing and providing the necessary coordination and artifact information. This is similar to interfacing with automated agents, so we discuss the two together. At the solution-level, service variation in Little-JIL can be handled easily by considering different services that can be used at each variation point. External automated agents can be specified in a variety of languages as long as an appropriate interface is provided, and in fact the STORM2 online dispute resolution system presented in Chapter 3 uses several software agents written in the Java programming language.

Since the approach aims to be technology-agnostic, the ability to accommodate service variation for heterogeneous agents and services is an important contribution. From a modeling perspective, the Little-JIL Elaborator can easily handle such requests without any changes, and the request specification language is robust enough to handle the precise description and retrieval of such externally specified behaviors. However, it is also important to note that by removing the specification from the coordination model, we lose control over several aspects of variants, including subtle coordination, or crosscutting communication with the rest of the agents involved in this human-intensive system. These capabilities can of course be achieved within the external implementation of software components, but that would diminish the return on investment of using Little-JIL by not taking advantage of features that the language is especially adept at handling, and the process family approach at large by making variation external to the variant specification. Moreover, as discussed in more detail in the next section of this chapter, making this information external to the Little-JIL process family coordination specification has important ramifications for the analyzability of such families.

7.1.1.3 Configuration specification constraints

In many of the process families to which we have applied this approach, we have noticed that there are crosscutting constraints that affect what variants can be appropriately selected at one variation point with respect to variants selected as another variation point that occurs earlier or later in the process. E.g. if a voter casts a vote on a paper ballot, then later in the process when that vote is counted, it ought to be counted following the correct procedure for a paper ballot. When there are such crosscutting constraints affecting multiple variation points throughout the process model, appropriate variants must be selected and the request specification should be flexible enough to support the expression of complex constraints. The theoretical framework presented in this dissertation allows for appearance-level requests at different elaboration steps that can include various desiderata. Many of the cross-cutting constraints we have encountered can be easily and naturally accommodated within the existing framework through a tagging approach, where abstractions that depend on similar abstractions elsewhere in the process can be tagged as such (e.g., "paper-voting" for the above scenario) and exclusions can be specified similarly. Note that because of the limitations of the experimental platform, this capability is not fully implemented yet. Within the existing implementation, such constraints can be easily specified post-generation as edge predicates. There are also some occasional configuration constraints that seem more easily expressible as language constructs than within the request specification, for example steps that are optional. Since edge predicates are fully supported by the Little-JIL Interpreter, inferring these automatically or with little human guidance from the original request specifications would be an exciting future direction to pursue for supporting self-adapting systems.

Another issue arising directly from configuration constraints has to do with step nomenclature and the evolving nature of process families that combine several different variation dimensions. To clearly illustrate the severity of this problem, consider an elaboration step A and a variant elaborating that step that, within its nominal specification throws an exception that is handled by reinvoking that variant. As soon as resolution happens, the elaboration step A will become an eponymous choice step, and the aforementioned variant will become its child and be renamed A, elb n for some value of n, depending on the number of variants available and the order in which they are considered. The exception within A, elb n that previously reinvoked that variant now points to the elaboration step, A. This may no longer be the correct scope for the exception, and, moreover, allows for the selection of any variant of A during the reinvocation, which is clearly not semantically equivalent to the original intent. This is a fundamental limitation of the approach and it is unclear what the best way to handle it is. Variants must be renamed in order to maintain well-formedness and allow for the analyzability PLAGE strives to accommodate, bur renaming a variant along one variation dimension (in the example above, a functional detail variant) has clear ramifications to variants in other variation dimensions (robustness variation variant in the case presented). Currently, such discrepancies are handler by the process developer on a case-by-case basis. Automatically resolving such naming issues may be aided by examining renaming patterns used for automated refactoring, but will likely remain a limitation of the approach and require human involvement. Another mitigation strategy would be to identify certain precedence relations that may exist among different variation dimensions, and automatically rename subjugated references to refer to the "correct" variant. In the case of functional variants and robustness variants above, the functional variants may be considered primary because they vary the nominal flow of the process, and their corresponding robustness variants may therefore automatically be renamed to refer to the original variant by traversing the AST upward until the first variant name is encountered.

7.1.2 Scalability

There are at least two considerations for modeling scalability, namely technological limitations and human-factor issues. Since the Little-JIL Elaborator and the PLAGE framework within which it resides extend the existing Little-JIL and Visual-JIL toolsets, we operate within the internal form representation of the language and we facilitate backward compatibility by making elaboration steps be a special case of reference steps. As explained in detail in Chapter 4, the main extension point for the system is the resolution activity, which for elaboration steps retrieves and grafts *all* matching abstractions whereas the original resolution activity throws an exception if more than one exists. This modified resolution activity is massively parallelized just like the original and spawns multiple threads while traversing the tree, resulting in non-deterministic sequences for any non-constrained abstractions. These limitations are hence inherited from the original internal form architecture but exacerbated by the much heavier processing load that comes with having multiple matching abstractions to graft onto an elaboration step. The resolution response time for a regular reference step tends to be instantaneous from the point of view of the user, whereas an elaboration step may cause noticeable delay to resolve depending on the number of matching abstractions. For all case studies performed thus far, there has been no noticeable delay observed. The new resolution activity is still highly parallelized and stops searching as soon as all matching abstractions are found, after which the grafting process is instantaneous for all practical purposes. Within the conceptual PLAGE architecture, if abstraction requests are handled and accessed through a database system as proposed in the API specifications, those accesses could then significantly slow down the resolution activity. That performance overhead would be incurred even if the Elaborator were used just for remote access and not to generate families with multiple variants.

From a human-factors standpoint, families are valuable to generate only if they aid in the understanding of the process. With multiple variation points at multiple elaboration steps in the process, and multiple variants that can be grafted to each of these, a family can easily and quickly become unwieldy and difficult to navigate. To help alleviate this problem, two different but complementary approaches can be adopted–the whole family can be generated for analysis purposes, as done in Chapter 5, but an expert developer should also be able to specify requests excluding some abstractions so that the resulting families with fewer members can be used to support the navigation, or ideally, so that individual highly-customized variants can be generated on the spot, providing another opportunity for dynamic variation accommodation. This use case of PLAGE would make it similar to a configuration management system for HISs, allowing process administrators to create process instances on demand based on current needs. The fact that service variation is explicitly defined within this framework would allow for fine-grained control over the process instances and facilitate configuration management of agents. Process models could be generated for agents who have different access control or different levels of expertise to follow. By extracting only those parts of the process family that are relevant for certain performers, role-based process instances could prove to be valuable (for a discussion of role-based vs narrative processes in previous work see [71]). This customized generation capability is not currently automated, but the four processes from Chapter 5 specifying the four different remote voting modalities demonstrate this approach.

7.2 Analysis limitations

One of the main research goals of our approach is analyzability. Although we have successfully generated and analyzed several sizable process families in case studies from a couple of domains, we have also identified some shortcomings in the approach when it comes to generating large families that provide valuable analysis results. We have considered and applied two analysis techniques, namely finite-state verification (FSV) and fault tree analysis (FTA). As with the modeling limitations, some problems stem from fundamental limitations of the scope of the approach, while others are the direct result of legacy constraints inherited from the experimental platform within which the Little-JIL Elaborator and PLAGE are being evaluated.

7.2.1 Representation limitations

As noted in Chapter 2, the intermediate step that gets grafted in the *mame* elaboration scenario is a choice step, a preexisting Little-JIL step sequencing construct denoting that only one of a choice step's substeps will be completed in order for the choice step to complete. Further, the semantics dictate that the choice of which substep to pick lies with the agents responsible for the substeps, i.e. the agent that carries out the choice step is not the one to make the choice but instead delegates this responsibility to the collection of agents responsible for all the substeps. This seemed to conveniently align with the family generation scenario, where there is not one agent (the choice step's assigned agent) who decides which elaboration to choose during a certain execution or in different analysis scenarios. However, using a choice step introduces some additional complexities that a newly defined step sequencing paradigm would have avoided. Conceptually, process developers may find it confusing that the same choice step is used for automatically generated *mame* families as they themselves use to specify choice. However, there is anecdotal evidence that inventing a new paradigm may have also introduced confusion because process developers who needed to encode lower-level variation into their processes have deliberately used choice steps, presumably because it was the sequencing paradigm that best represented such variation. Using a choice step also has ramifications for the performance of the analysis toolsets with which the PLAGE framework was evaluated. Choice step alternatives lead to the creation of may immediately precede (MIP) edges in the trace flow graph (TFG), which significantly impact performance. A new sequencing paradigm could be defined to prevent MIP edges from being introduced, and they could also be avoided through specifying analyzer-specific cross-lateral constraints pre-encoding what variants should not be considered together, or translating the process using a different sequencing paradigm that does not introduce MIP edges, such as a try step with all but its last substep defined to be optional. Although this depicts an ordered alternative and is therefore not an accurate representation of an elaboration step, it would result in all possibilities being considered for analysis purposed without introducing additional MIP edges thanks to the artificially imposed order.

7.2.2 Translation constraints

For both analysis techniques that we have used, the Little-JIL specification of the process family first gets translated into a suitable representation for analysis, in this case a TFG. The technical challenges pertinent to this translation scheme are described elsewhere [19, 31]. Within the Little-JIL language and the Visual-JIL modeling framework, a process developer can specify edge predicates indicating conditions that must evaluate to true in order for the connecting steps to be executed by the Little-JIL interpreter. We have previously used these edge predicates to facilitate control flow in a fully executable process-guided dispute resolution system, STORM2. With respect to the PLAGE infrastructure, it is important to note that the aforementioned edge predicates are omitted during translation and the representation of the process family that gets analyzed will therefore not contain these lateral or cross-cutting constraint specifications. Since a TFG is already an over-approximation of system behavior, when combined with the omission of the additional constraints, it inevitably leads to false positives, as evident in the analysis results presented in Chapter 5 and discussed in Chapter 6. Thus, the system may report that a property is violated when a variant containing an undesirable event is selected, even though the constraints on that variant's edge should have removed it from consideration if they had been evaluated correctly. In such cases, the developer can remove the variant causing the spurious results and rerun the analysis to get more insights into the safety of the process family. Additionally, the process developer can force the FLAVERS analyzer not to consider certain combinations or elaborations by specifying cross-lateral constraints as enumerated types in the dataflow.

False positives, however, are a fundamental limitation of the approach and would happen even if the translation accurately carried over constraints. For example, the selection of service variation abstractions can be limited based on their attributes, but if these are external to the coordination model (e.g., encoded within the abstraction specifications or resource repository), the analysis toolset would still not consider them. Additionally, for large families, it may be unwieldy to specify all the cross-cutting constraints in advance, or there may be concerns about how variants interact that are discovered throughout the course of process improvement and analysis and are thus unknown a priori. Another concern is the possibility of false negatives where variants are not considered together because of some constraint but they could in fact exist together and cause unsafe behaviors. Both of these limitations would be in part mitigated if some bindings could be automatically inferred in the future based on the interface specifications of steps (e.g., if a step is declared to have the ability to modify a certain artifact, then that step may be automatically included in the possible bindings for the corresponding event).

7.2.3 Scalability

The combinatorial nature of variant selection taxes the analysis toolset as families result in much larger state spaces to explore. Based on the limited number of data points observed in Chapter 5, however, we were able to obtain reasonable performance (verification completion within a couple of minutes for the largest problems and a few seconds in all other cases). Although the trace flow graph constructed for the entire process family was significantly larger (in terms of number of nodes, edges, and MIP edges) than the sum of its four variants, the verification time was also significantly smaller than that sum, indicating that the common core did provide some performance advantage. Additionally, the biggest culprit for FLAVERS seemed to be the sheer number of interleavings that elaboration points can result in; these are encoded with MIP edges in the TFG, and significantly increase the verification time. FLAVERS does not utilize summaries, and that may be a future direction for optimization, along with other techniques to transfer constraints from the process family specifications into the analyzable representation of that family to restrict the number of combinations that must be considered. The Little-JIL Elaborator was a main reason for upgrading the FTA platform after the internal espresso library, which was used for the minimal cut set calculations, was overwhelmed by the size of the fault trees produced. At the time, the fault trees generated for the Yolo Election process with only twelve possible combinations of seven total variants exceeded the size of any previous fault tree by a factor of 3+. Since then, yet bigger families have been generated, with the case study highlighted in Chapter 5 containing three variation points, twelve variants, and over 100 different combinations, and the newly upgraded FTA tool can generate the appropriate fault trees and calculate MCSs with close to realtime performance. The finite-state verification system may encounter similar limitations stemming from the sheer size of the process families under consideration.

As with modeling, the human factor scalability constraints must be considered as well. The extremely large fault trees generated for the Yolo election process (over 700 nodes in the highly-optimized DAG version) were difficult to navigate and it would be nearly impossible to identify problematic areas in the process without the very disciplined use of the MCSs and a deep knowledge of the process. Given the rich potential of process family analysis in unearthing previously unrecognized vulnerabilities, efforts on facilitating the interpretation of the results and making sense of the vast amounts of analysis data, such as related work by Phan et al, would become paramount.

CHAPTER 8

RELATED WORK

There has been extensive work on creating and managing software product lines and software families [22, 41, 58, 59, 62, 84].

8.1 Problem-level variation

Different approaches for variability management suggest handling variability at different stages in the software development lifecycle. For example, [10] propose explicit modeling and management of architecture variability, since the architecture stage is very ripe for addressing variation; this work also outlines different sources of variation, some of which correspond to some of the kinds of variation that we have observed and outlined here in this dissertation, such as functional variation, performance variation, agent/service variation, and functional invariance, though the terminology differs. This work focuses mostly on the problem level; our approach is similar, but contributes additional, and important, dimensions of variation, and strives to provide support for solution-level mappings to effect variability management and facilitate *generation* and *analysis*. Conventional modeling approaches, such as the use of UML to identify patterns in architectures [34], have also been applied to model variation in system architectures. There are clear benefits to such approaches for modeling variation and variability throughout the development life cycle in both domain engineering and application engineering, or the problem and solution levels.

Feature-Oriented Domain Analysis, FODA [25, 42] advocates the development of a domain model representing the family of systems, which can then be configured and instantiated to produce a specification for a single system implementation. Similarly to other compositional approaches at the problem level, this encourages reuse and the identification of components that can be shared across variants.

Feature diagrams (e.g. [43, 65]) are widely used to model different feature configurations through variation points and different semantics for composing and combining features based on predefined constraints. Features closely correspond to the functional variation presented. There are several approaches that focus on the problem-level specification of variation through features, such as using domain-specific feature graphs for specifying variability within a model (e.g., [11, 12, 43]) or deriving the specification of a product through several different stages of configuration (e.g., [26]). Similarly, decision models (e.g. KobrA [9] and FAST [84]) can be used for instance generation and variant modeling, where variation points are indicated as decisions and, based on the selection, the model is extended with different sets of features. Feature graphs and decision models are explicit enumerations of mandatory and optional features. Such approaches can lead to very large feature graphs when all elaborations are included.

The approach presented in this dissertation could likely benefit by being augmented with explicit enumeration approaches in some cases to enunciate how each product or process variant within a family can be derived. Since the underlying goal of our work, however, is to better understand the different kinds of variation that exist in systems, we suggest the potential necessity to model these different kinds of variation both implicitly and explicitly, especially since our preliminary work suggests that some approaches may be more effective in defining and managing families characterized by different kinds of variation. Features are very helpful for describing the functionality and behavior of a family of systems, but may be insufficient for addressing variation driven by non-functional requirements, or variation in quality attribute, such as robustness, performance, or agent variation, which may necessitate the application of a different modeling approach as some of these attributes may not fir neatly into features. Moreover, some of the kinds of variation addressed here may be difficult to model using a single approach. In fact, they may necessitate the application of several different problem-level and solution-level specification approaches.

8.2 Solution-level variation

There are many different techniques aiming to aid the generation of software variants at the solution level. These can be loosely categorized into positive variation techniques, where families are generated by combining different features into a product, or negative variation techniques where one big product is initially created and then features are removed from it to achieve a specific configuration. Positive variation techniques include generation approaches for creating software product lines based on either a programming language or accompanying tool support, techniques and approaches for safely composing product lines out of preexisting features, components, or both. Negative variation techniques usually focus on creating one conglomerate product and then removing unnecessary features based on annotations or tags which can be added manually by developers or sometimes inferred.

8.2.1 Positive variation (generation and compositional approaches)

For example, in the domain of software families, component-based and generation approaches have been used to specify initial configuration specifications of components and then to apply generation techniques for parameterization [24, 33, 48]; there are also techniques for product line implementation such as component reuse [82], feature-oriented programming (FOP) and related modeling approaches [13,77], mixins [73] aspect-oriented programming [5, 33, 47] and its applicability and suitability for implementing different kinds of variation [4]. In [75], a taxonomy is presented of different techniques that can be used for variability realization in different scenarios. Most of these approaches are compositional in nature, meaning that a shared common core is gradually extended with more functionality to build a specific variant, facilitating generation.

8.2.2 Annotation approaches

There are also annotation and pruning techniques [45, 46], which take the opposite approach of building a union of the desired products in which different components or features are annotated differently, and then pruning away what is not considered to be necessary as specified by the requirements of the product configuration. Such techniques and tools are clearly useful for generating different members of a software family. These approaches do not explicitly address the needs of process families and the management of variation in processes, while the approach presented here advocates careful modeling and reasoning based upon different process artifacts; some of these artifacts have clear parallels to artifacts produced in the software development lifecycle, such as specific implementation artifacts, while others, such as specific assurances about required analytic properties that are met by all family members, do not.

8.3 Hybrid approaches

Other approaches focus on supporting variability modeling and management throughout different stages of the software development lifecycle through combining problem-level modeling of variation with solution-level product derivation. For example, the COVAMOF variability modeling framework [72] promotes the careful modeling of variation points and the different dependencies that may exist among variants, addressing different aspects of generation and analysis. The pure::variants tool [14] similarly provides support for the generation of new variants by supporting a configuration specification including different constraints at the domain engineering level (similar to problem-level variation not discussed here), and consequently derived configuration specifications for the implementation of variants at the application engineering level (similar to solution-level variation). Hybrid approaches have successfully combined feature-oriented programming (FOP) and model-driven development to demonstrate how software product lines can be modeled (with a focus on features) at the problem level, and then at the solution level products can be derived from these models using FOP [77].

Hybrid approaches based on the original FODA idea have also been developed. For example, in [28], feature diagrams are used to specify the overall system family, and then Model-Driven Software Development (MDSD) techniques are applied to come up with a hybrid representation of feature based model templates which could then be used as input to automatic software generation approaches.

The Koala framework [81, 82] has also been applied to support the reuse of components for different products within a family, and between different software families, with a focus on shared functionality. Apel et al [6] have shown how model superimposition can be applied to different UML models: models are first decomposed into features, then appropriate model are composed together using superimposition to produce models for single variants within a product line.

8.4 Process lines

Several approaches have been proposed to address the need for variation specifically within processes and process models. Armbrust et al [7] describe different scoping mechanisms to support explicit decision making on what existing processes to compose into process lines and thus provide future cost savings and faster product development. Washizaki [83] proposes a similar approach of considering existing processes, but focuses on defining a high-level process line architecture in which a core process can be extended at various variation points to derive product-specific processes composed from the bottom up (this technique uses the OMG's Software Process Engineering Metamodel, SPEM, and UML activity diagrams). Martinez-Ruiz et al [54] in their SPRINTT approach [55] also provide a process variation specification approach for process models through variation points and variants in an extension for SPEM, vSPEM. Simple constraints, relationships, and dependencies are supported. The Provop (PROcess Variants by OPtions) framework by Hallerbach et al [38] supports the specification and management of process families written in the Business Process Modeling Notation, BPMN [85]. Provop lets developers apply simple operations to a base process, such as deleting, inserting or moving different process fragments to create variants. Van der Aalst et al [79, 80] also focus on configuring individual process models for specific product development, but propose an approach to evaluate the correctness of the derived processes. An approach for generating different workflow instances for scientific experiments is presented in [24], where each scientific experiment corresponds to a family member and is represented as a workflow pattern, allowing the authors to define relationships between workflow events so that multiple experiments can be defined by composing several activities in different ways; at the same time, a provenance model is maintained so that a scientist can trace back where a specific "cartridge," or variant, was chosen from, and what other variants were available. Although this work suggests that relationships between activities within an experiment can be defined and explored, it does not focus on understanding the relationships between different experiments or suggest guidance to the process (experiment) line developer on how different com- positions might be derived from existing ones.

In [61], the authors propose to use the PROMENADE process modeling notation to flexibly model and derive processes through the definition of different precedence in relationships between components. PROMENADE provides flexible support for specifying dataflow and is supported as a UML profile. Similarly, in [15], development processes are generated through the prism of the Spiral model, as different variants of applying the Spiral Model at different stages of the software development process. Traditional domain modeling techniques have been applied to model variability in workflows as well, which could then be used to derive a specific process instance. In [35], the authors present another approach for generating concrete software development processes out of an overall configuration based on the Spiral Model and enhanced with ideas from the Capability Maturity Model.

Some variation dimensions identified in this dissertation have received more attention than others. Although most of the work in software product lines focuses on what we refer to as functional detail variation, there are approaches that address other dimensions as well. Service variation is addressed in [8] where a family is first represented using a modal transition system and then verified with respect to requirements specified in vaCTL. Asirelli et al. define service variation to be different web service providers (in this case, websites for flight and hotel bookings), whereas service variation in our approach encompasses those kinds of variation but also allows for variation in human behavior modeled within the system boundary of an HIS.

In [78], the authors correctly observe that in order to be useful, workflows must provide enough flexibility to allow for variation, while being declarative and prescriptive enough to provide usefulness as guides. This work is further extended in [79], where a generative approach is presented based on formal definitions of the models as Petri nets. A conglomerate model is used to represent the collection of processes that may be desirable, and individual process instances are then derived from it. Specifically, starting with a reference model (similar to the concept of a common core elaborated with all possible elaborations), certain transitions in the Petri net can be skipped or removed to generate configured instances. Propositional logic constraints are imposed at every step of the configuration process to ensure syntactic and semantic correctness of the derived model. Semantic correctness here includes properties such as removing unreachable states and guaranteeing that given a sound reference model, the derived one will also be sound, but focus on proving that the derived instances will be generated correctly from the reference model, as opposed to proving additional properties about the instances or the entire family.

As a recent literature review [56] notes, most existing approaches have serious limitations because they either only address one aspect of variation (most often what we describe here as functional detail variation), or, when including data-flow and role-based variation these are usually not varied orthogonally to the activity specification. Most of these approaches focus on modeling notations and representations that do not have semantics that are rigorous enough to support the kinds of analyses discussed in this dissertation. We provide a framework that can address several of these aspects orthogonally and in a rigorous way, allowing for the composition and nesting of families based on different variation relations.

8.5 Analysis

The approach presented in [40] addresses problem-level variation, with a focus mostly on what is referred to as functional variation in this dissertation, through the use of UML activity diagrams. The notation is augmented with formal Petri net semantics that allow for the careful and precise documentation of variability in activity diagrams. This facilitates quality assurance in the domain engineering phase of SPLE, which closely corresponds to the problem-level variation needs discussed in this dissertation. A similar approach to address analysis at the domain engineering level is presented in [49], where the authors formalize domain artifacts as I/O-automata which are then model-checked against properties specified in computational tree logic.

In [27], the authors extend the idea of feature-based model templates ([28] as discussed above) and extend their semantics to support automated reasoning. Requirements for the templates are specified using the Object-Constraint Language (OCL), and can be automatically applied to ensure that templates derived from the overall feature configuration specification are well-formed according to a set of predefined constraints. Thus, no ill-specified product instance can be derived or built.

In [78, 79], a similar approach is presented for presenting the correctness of process models by construction through the use of formalism such as Petri Nets and event-driven process chains. The derivation of individual business process models from the overall configuration model is then constrained to guarantee well-formedness. While well-formedness is a very important first step in ensuring the usefulness and correctness of a solution-level system family, the kinds of reasoning we hope to support extend to additional kinds of correctness specifications beyond syntax.

Different approaches address the problem of making assurances about an entire product line or software family at the solution level differently. Some focus on placing restrictions on the creation of new variants, thus impeding generation, but providing well-formedness assurances for all variants that are allowed to be generated (e.g., [44, 72, 76]). Others focus on the traceability of features to subsets of components from the core assets and can reason about those relationships using QSAT (a SAT solver modified to handle quantified Boolean formulae) [57], or, for more sophisticated analysis capabilities the system can first be modeled in product line CCS and then checked against multi-valued modal Kripke structures to determine legal configurations that satisfy the requirements [37]. Although this is congruent with the analysis aspect of variation emphasized in this dissertation, it only addresses well-formedness constraints, and not other types of properties, such as safety, performance or robustness properties, that a developer may want to analyze an entire software family against.

Another line of work that is similar to the approach we intend to take is presented in [52], where analysis is aided by imposing certain obligations at each variation point (similar to elaboration step in our approach). This allows the authors to guarantee that if the feature (elaboration) that is to be attached at a variation point satisfied the obligation, then the composition itself will also satisfy the obligation. Families can therefore be ascertained to be safe by construction, and analysis results can be reused from one configuration to another provided the configuration is done incrementally so that obligations can be recomputed if necessary after a feature has been added to a variation point. The request language of the Little-JIL Elaborator may provide enough flexibility that some such constraints and obligations may be specifiable at each elaboration step, thus aiding the analysis of a process family through its safe generation.

In [21], the authors extend label transition systems with features, and use the resulting formalism to describe the behavior of a family of systems, determine the feature differences between variants, and apply model checking to determine if the family adheres to prespecified temporal properties, or if there are variants that violate the properties. In the case of a property violation, the system is able to use the featured transition system to identify the variants that fail to satisfy the property. This work is extended in [20] where further functional specifications of the system are written in the Text-based Variability Language (TVL) and fPromela (a featured extension of Promela) and then model-checked against fLTL properties to determine what products satisfy the properties. This approach is very similar to the conceptual framework presented in this dissertation, and maps closely to both the problem-level functional variation (which most closely maps to features), as well as the solution-level nominal flow change. The similarities are both in the modeling aspect (i.e. with respect to *generation* through the possible use of the featured transition system), as well as the *analysis* aspect. However, we would like to be able to reason about more than feature variation; therefore, considering features as the defining difference between variants may be necessary to address functional variation but may not be sufficient for the other dimensions.

CHAPTER 9

FUTURE WORK

There are many promising directions for future work. Here we discuss possible extensions to the conceptual framework and how they align with supporting both research goals of improved generation and analysis of process and system families.

Conceptually, one obvious direction to pursue is to develop a more complete list of types of problem space variations and to devise an approach to classifying them and formally defining them. We are particularly interested in understanding the ways in which these different problem space variation families overlap and intersect with each other and, similarly, how their defining characteristics interact with one another and whether there are transformation functions that can be specified for composing different families together or morphing a family based on one variation dimension into a different one. Formal definitions of the variation relations that define different software families would encourage and perhaps facilitate such reasoning about how different families that share variant members may interact, especially in the case of variation relations that are not orthogonal (such as composing functional elaboration variation and robustness variation families). Such interfamilial interactions may also inform architectural decisions since some kinds of variation relations seem to accommodate the sharing of a common architecture, while others seem to result in variants that are structurally different and require different architectures.

We are also interested in identifying a broad range of approaches to generating solution space families from these problem-level variation requirements in the suggested taxonomy. In order for this variation taxonomy to be truly useful, however, the underlying relations that define family set membership need to be defined formally in a way that would support automatic construction of new architectures and canonical transformations from one dimension of variation to another. Although we have provided initial support for creating families of process variants only in the Little-JIL process definition language, we believe the domains and case studies we have examined cover many commonly used process language features, and should therefore lead to useful insights about supporting variation in different process definition languages and in other system representations. This approach aspires to be technology-agnostic.

In terms of generation at the solution level, since Little-JIL is a fully-executable language, the language interpreter could be modified to support executing process lines to potentially support real-time deployments of self-adapting families. This approach would be applicable to any process definition language that has the semantics to support dynamic binding. Executable process lines would be especially helpful for process guidance systems, which could then self-optimize depending on the context or self-heal if exceptional situations arise that require the switch to a different robustness variant within the family. Additionally, for relationships that are amenable to a common core architecture, our approach allows for the specifications of reusable process modules and components. For example, elections, one of the domains discussed herein, are rich in real-world process modules that are shared among jurisdictions. A library of standard procedures could support the generation of process variants through composing pre-specified (and pre-verified) modules for voter registration, ballot counting, and so on. Thus far, we have considered the election processes of Yolo and Marin counties in California, which are very different, but employ similar procedures for certain subprocesses.

Analysis facilitation is another key goal of our research, aimed at identifying generation approaches that support specific kinds of reasoning about whole families of variants. In Chapter 2, we described how induction constrained by trace equivalence can be used to generate families whose members are all equivalent with respect to a pre-specified property. We also suggest that dynamic approaches such as simulation may be suitable for supporting analytic reasoning about performance variation families. Process families can be leveraged for different kinds of analyses besides. We have demonstrated sample applications of finite-state verification and fault tree analysis in this dissertation. Extending previous work [66], we have successfully applied fault tree analysis (FTA) to very large families of election processes. Being able to analyze and interpret the resulting fault trees can aid in detecting fraudulent or colluding agents, as well as specific variants within which the misperformances of steps can have disastrous consequences.

We would like to further study this issue of agent collusion. In previous work [67], we have explored how election security may be compromised if a rogue election official is employed. There are additional studies that can investigate this problem more carefully. As outlined in the Chapter 4, minimal cut sets derived from fault trees can be used to model a scenario where one corrupt official is allowed to execute as many steps as allowed by the agent's capabilities to explore whether the resulting variant is more vulnerable to pre-identified attacks (i.e. whether the agent is able to find enough steps to misperform so that an entire MCS is compromised, leading to the hazard to occur) or fails to satisfy some security or privacy properties that a variant using only non-corrupt agents would (by looking at variant constraints used in combination with the finite-state verification). Additionally, the same approach can be applied to different scenarios involving two or more collaborating rogue agents to see if there are general attributes of the resource allocation that may lead to better or worse resilience against agent collusion, or to identify ways to expose pairs of corrupt agents, who may appear to be performing correctly when examined individually. Such wide-scale attacks that are not easily detected are of great interest to election officials, and, moreover, there are specific parts of the process that officials may consider vulnerable to such attacks. The ability to perform formal analysis using multiple different variants to demonstrate different possible courses of attacks would therefore presumably be very valuable to the domain experts.

Using FSV and FTA as complementary approaches and finding a way to combine the analysis results would be especially helpful. Finally, process families can be used to evaluate different agent behaviors along multiple different aspects, such as performance metrics or expertise assignments, by running simulations. Simulations with random agent assignments can help to derive the impact of certain steps being executed by a rogue agent, or the likelihood that two steps can be executed by the same agent or multiple colluding agents for attack prevention.

APPENDIX A

SYSTEM DESIGN AND IMPLEMENTATION DOCUMENTATION

In this appendix, we detail the architecture of the existed Little-JIL Elaborator system and explain its implementation with respect to the existing infrastructure. We then explain how the Little-JIL Elaborator fits within a larger conceptual framework that includes a parsing engine and sophisticated repository access. The parsing engine can process elaboration request specifications written in the request specification language defined in Chapter 4, including context constraints specifying which appearances of elaboration steps should be instantiated with abstractions. The repository access is configured in such a way that it could accommodate different databases depending on the abstraction needs and these databases can be easily replaced even in a dynamic setting. In order to understand the architecture of the Little-JIL Elaborator and the larger PLAGE system, we begin first with an overview of the existing infrastructure of the Little-JIL internal form, whose current intended architecture is illustrated in Figure A.1.

A.1 Architecture of current Little-JIL infrastructure

This figure represents the notion that very agent element has a corresponding agenda, which contains zero or more agenda items. Agenda items, which in term correspond to steps, are handled by the interpreter, which consults with a configuration object. The configuration object (before the execution of a process can begin) establishes connections to necessary services, such as a resource manager for example. Each agenda item represents a step instance (this is where the architecture is very difficult to represent because

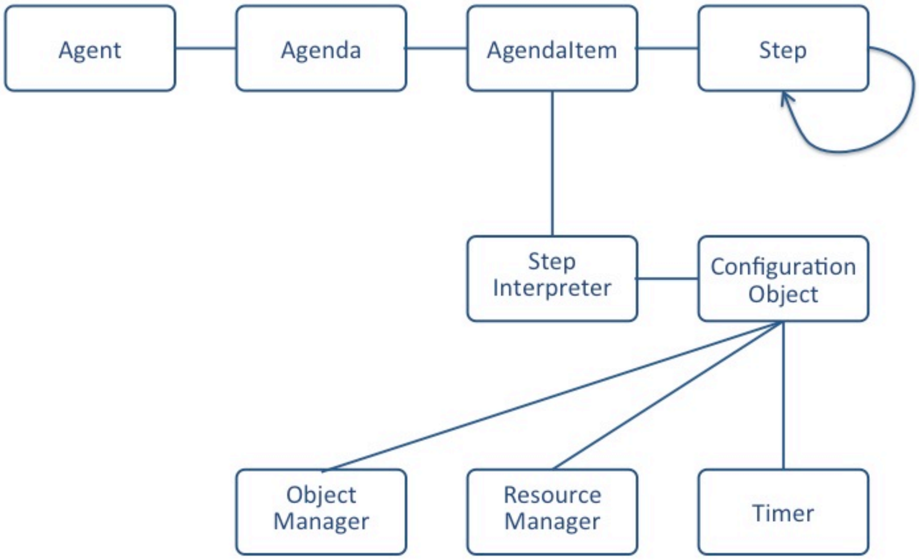


Figure A.1. Intended architecture of the Little-JIL internal form.

most of this architecture sketch is inherently referring to a dynamic structure, and thus for example agenda items refer to dynamic elements, but the steps that agenda items correspond to cross over to the static representation of the process that is defined in a Little-JIL coordination model). We now focus on the step element in the upper right, and elaborate it further in Figure A.2.

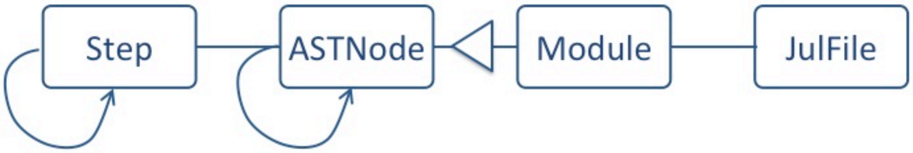


Figure A.2. "Step" element elaboration.

For the purposes of the PLAGE architecture and design in general, the element of most interest is the ASTNode, which is, as the name suggests, a node in the Little-JIL abstract syntax tree (AST). Given that the syntax tree of a Little-JIL process is built from a collection of steps, the node intuitively resolves to a step when the tree is statically checked for well-formedness or converted into a form suitable for execution. Also important from the point of view of process fragment storage and management are the JulFile and the Module. Each of these elements is described in more detail below.

A.1.1 A Little-JIL Process behind the Scenes

A Little-JIL process conceptually comprises a coordination diagram, an artifact specification, and a resource specification (which includes agents and other actors and resources). However, this is represented in Visual-JIL somewhat differently. A Little-JIL process is, at a high-level, a .jul file.

A *.jul file* consists of a .ljx representation along with any additional specifications (e.g. agent behaviors, artifacts, or exceptions if they are declared as Java classes) as well as additional required configuration files (for example for VSRM agent setup).

A *.ljx file* is an encoding of the coordination diagram, using an XML schema. The geometry of components is used to determine sequencing (for sequential steps), and all the other information about the coordination (for example, connectors, type of step, interface declarations, etc.) is stored as properties in the metadata. This file contains one or more Module.

A *Module* contains a collection of diagrams in the Visual-JIL editor. Diagrams, however, are semantically meaningless and therefore not represented separately in the class hierarchy (explained in more detail below).

We can now begin to explain the Little-JIL Elaborator architecture and then outline the overarching PLAGE architecture.

A.2 Overview of the Little-JIL Elaborator Architecture

As a reminder, at the high level, the Little-JIL Elaborator system architecture conceptually the Elaborator contains three main modules, as shown in Figure 4.1, namely:

1. A backend service which involves data storage, organization, and management in some kind of repository,

2. A middleware module that contains the business logic, in this case split up into two main areas of concern, namely
    (a) integration with the underlying Little-JIL internal form on the one hand, and
    (b) communication, coordination, and integration with the repository via the Process Fragment Manager/PFM submodule above or, optionally, ROMEO if used as a plugin service.
3. A frontend service that integrates with the existing Visual-JIL framework in an extensible manner.

The Elaborator provides additional functionality to the existing Little-JIL internal form that governs syntax and the Visual-JIL graphical editing environment, and these interfaces and extensions make its architecture more complicated in reality than the conceptual model-view-controller three-piece sketch of the internal representation, the graphical projection, and the middleware, respectively, shown earlier. To illustrate the basic interactions with the existing framework as well as planned extensions and revisions, Figure A.3 details existing parts of the system that will only require cosmetic changes (in dotted lines), as well as significant extensions (in solid lines).

From an implementation standpoint, there are several existing projects to interface with that correspond loosely to the dotted-line components: the Little-JIL internal form comprises several projects, but the two of most relevance here are the eponymous Little-JIL Internal Form (the Little-JIL abstract syntax tree lives here), and especially the "resolved" package within; and the Little-JIL XML project, and again especially the "resolved" package for backend and logic integration. In terms of the Visual-JIL Editor, the two projects that require changes and extensions are Little-JIL Visual-JIL plugin and possibly the Little-JIL Search Plugin for some extensions to specific GUI features. Specific interfaces for the different features are outlined later in Section A.4

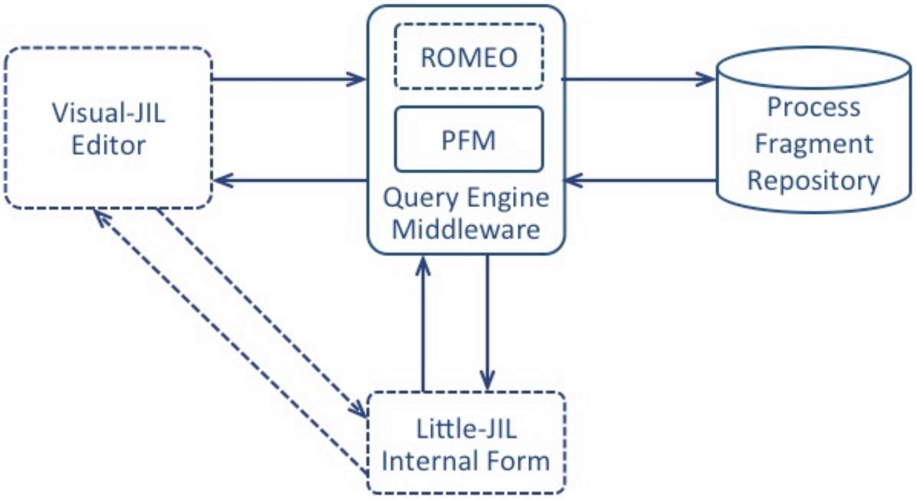


Figure A.3. More realistic architecture of the Little-JIL Elaborator.

A.3 Process Lines Analysis, Generation, and Evaluation (PLAGE)

We now present the architecture for a system that encompasses more sophisticated capabilities and neatly encapsulates the existing Little-JIL Elaborator. More than a set of additional features to supplement the Elaborator's functionality, PLAGE aspires to provide a holistic framework for designing, generating, and analyzing process families, and guiding their evaluation and continuous improvement through iterative changes.

Let us consider the changes necessary to accommodate the extension points and support the additional functionality of PLAGE. Figure A.4 presents the first step toward this goal; it is visually quite similar to the detailed architecture of the Little-JIL Elaborator architecture shown above in Figure A.3, however, it is important to note that the Internal Form itself will have to access the Process Fragment Repository and interact with it (this interaction will be explained in more detail in the following diagram), and that the Internal Form will now take in a configuration file that contains the elaboration step definitions and requests. With these changes in place, providing the additional functionality needed in PLAGE becomes simpler. Another important implication of these changes is the fact that we now begin to deal with dynamic processes and allow for the elaboration of

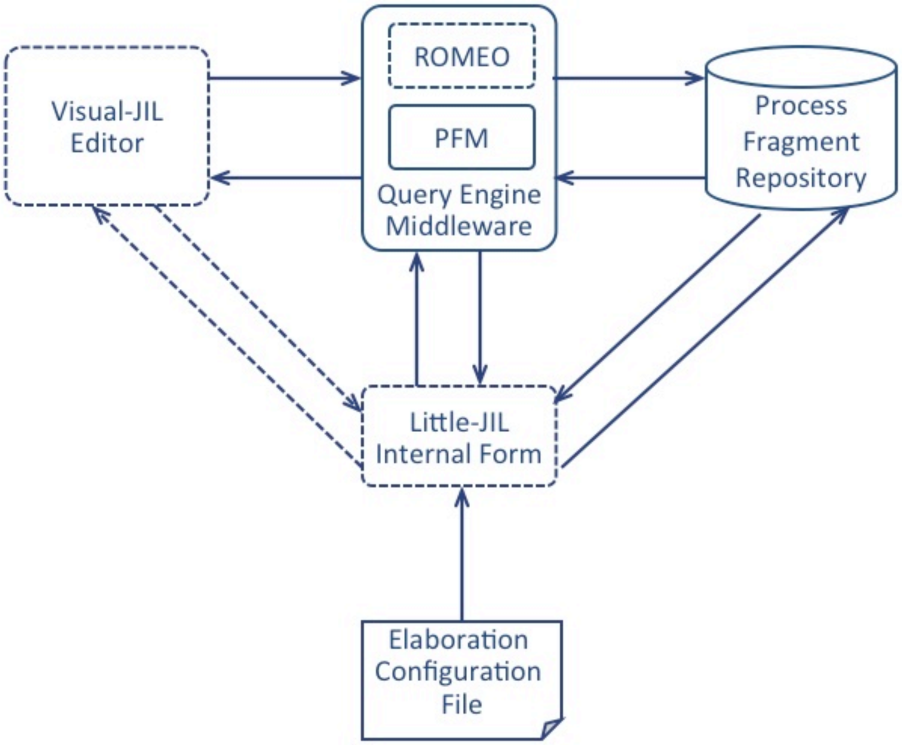


Figure A.4. Extending the Little-JIL Elaborator architecture to accommodate PLAGE.

any step appearance. This inevitably means that some elaborations PLAGE can support cannot be visualized because the existing Visual-JIL environment is strictly static-based. Therefore, if the configuration file specifies that only the first of several invocations of a certain elaboration step is to be elaborated with a certain cion, PLAGE could support that elaboration within the extended internal form of the process definition, but that elaboration could not be reasonably visualized.

To make the changes easier to understand, If we were to explore them in more detail and focus on the use case of applying formal analysis to process families—one of the major research goals of this dissertation—the scenario would look as portrayed in Figure A.5. As before, solid lines are new extensions, dashed lines are modifications to existing modules, components, and interactions, and dotted lines here imply data-only communication.

For this diagram, transforming components are oval, and artifacts are rectangles. When a user wants to generate or analyze a process family, this family is constructed dynam-

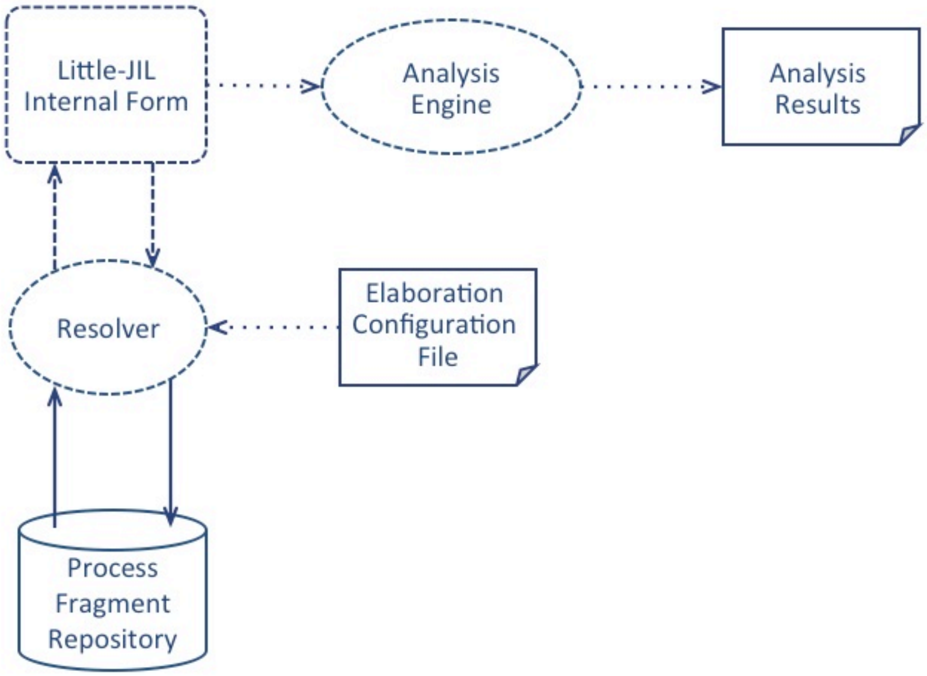


Figure A.5. Process family analysis use case for the initial PLAGE architecture.

ically. A representation of the elaborated common core is constructed in the internal form and can be visualized with the Little-JIL Elaborator on demand (within the static constraints outlined above). To achieve the elaboration, the Internal Form consults the Resolver, whose current job is to find matching subprocess and replace each and every reference with a fully expanded process fragment. This resolution activity has already been extended to deal with finding multiple abstractions and grafting multiple cions for the Little-JIL Elaborator. This resolution activity produces a fully resolved process containing multiple coins grafted onto choice steps as currently done by the Little-JIL Elaborator, but for PLAGE, it also takes a configuration file that restricts what appearances of elaboration steps are to be elaborated and what abstractions are to be considered as "suitable" cions depending on the desiderata specified. This complete dynamic representation can then be passed off to the Analysis Engine (could be either FLAVERS for FSV or the Fault Tree Analyzer) (which already uses the fully resolved version of the process and thus requires no changes) and can then produce the relevant analysis results.

The major changes implied by this extended architecture are twofold:

1. this development plan requires a much more heavy interaction with the Eclipse plugin development framework and heavier integration with the Interpreter, and

2. because the Elaborator now fully resides on the dynamic side of Little-JIL processes and the way that processes are represented, producing a visual representation of elaborated processes becomes an entirely new research question because it is no longer an issue of just modifying the user interface and "gluing" diagrams together, but also requires deconstructing the dynamic representation of the process into a set of static coordination diagrams that can then be displayed. How to reasonably display these to users in an understandable and intuitive manner would be a promising avenue for future work.

It is also important to understand that the Little-JIL Elaborator is no longer a standalone software as it has been, but becomes a component in the overall PLAGE system, where elaboration is a small but important piece. To illustrate, the architecture for the entire PLAGE framework is shown in Figure A.6.

There are four separate areas of concern when it comes to the new set of functionality PLAGE strives to provide. These are, in order of importance: resolution, abstraction selection, repository access, and an intuitive user experience (UX). Each area is discussed in more detail below before being deconstructed into features and components in the next section.

A.3.1 Resolution

Resolution happens within the Little-JIL Internal Form engine in Figure A.6. As a reminder, when a Little-JIL process is drawn on the screen, the diagram has no semantic meaning. Therefore, a reference on a diagram is actually meaningless and gets "resolved" behind the scenes to a subprocess, which essentially gets grafted onto every placeholder where a reference used to be. A fully resolved process therefore has no references.

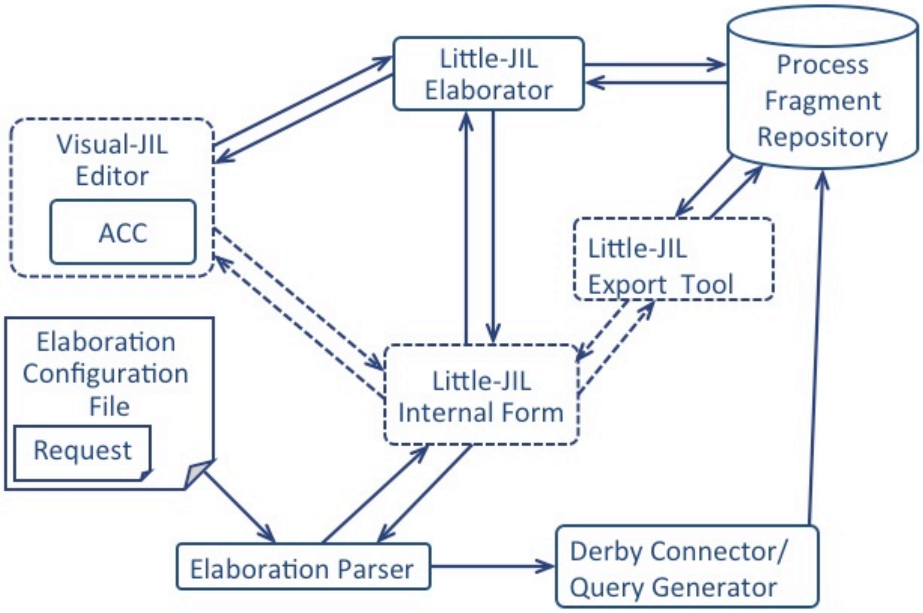


Figure A.6. Finalized detailed architecture of PLAGE, including the parsing engine and database access.

The resolution activity works by doing a tree traversal until it finds a matching subprocess. If no matching subprocess is found in the current process definition (for Visual-JIL, this means the current "module," where each module consists of one or more "diagrams"—refer back to A.1.1 for the detailed explanation), then Eclipse provides a way to specify precedence for other modules, and then other projects, and that is the order in which a traversal is to be performed. If a reference resolves to more than one subprocess within a single module, a MultipleResolutionsException is thrown. The Little-JIL Elaborator overrides this capability for reference steps that have been specified as elaboration steps. As explained in Chapter 4, when an elaboration step resolves to more than one abstraction, all appropriate cions are grafted onto a newly created choice step. When only one abstraction fits the elaboration specification, the resolution process remains the same as for regular reference steps.

A.3.2 Abstraction selection

Although there is a UX component elaboration selection, here we focus on the critical responsibilities, which are specifying desiderata for elaboration steps. In Figure A.6, the specification of the desiderata is represented in the elaboration configuration file as a series of request specifications for all elaboration steps. This specification cannot be fulfilled without being parsed first, so it is passed to the Elaboration Parser to be translated into a repository access request. More details on the parsing activity are provided in the next section. In addition to the pertinent features, there are graphic user interface suggestions to facilitate the initial elaboration selection specification that would be nice to have but are non-critical to the functionality of the system. They are therefore specified below as incremental and pending on the initial critical features.

In order to indicate which reference steps of a process are to be considered elaboration steps, we use a configuration file that would reside in the jul file specification of the core process that is to be elaborated. Such configuration files are already in use for integration with multiple parts of the system, and a simple example is the VSRM specification of resources. Currently, elaboration steps are indicated by modifying the .ljx file describing a Little-JIL process in xml by setting a Boolean flag that accompanies all reference step appearances to true for those appearances that should allow for elaboration. Once the configuration file approach is implemented, transferring over the selection to a process that has not yet been annotated is trivial.

The desiderata for each step is then specified as part of this configuration file. An example configuration file may look something like this:

```
name-1
      elaboration-type (process or agent)
      context*
attribute*
best
```

```
name-2
     . . .
```

Each element in this specification corresponds to the elements described in the request language in Chapter 4 (remember that *name* is a partial or complete path, not just a step name). Note that in order to integrate with the existing Little-JIL and Visual-JIL infrastructures, such a configuration file would need be constructed even if the interaction were entirely GUI-based in a scenario where the user navigates the diagram and selects elaboration steps and specifies requests.

Requests specified using either approach would likely be encoded exactly like this, specifying all the necessary components of the schema as described in Chapter 4. Once this configuration is specified, it needs to be read and parsed. To provide for additional flexibility (such as providing extension points to support different process definition notations in the future) and to allow for the parsing of complex requests (e.g. both the context and attribute elements may contain multiple entries that may need to be parsed differently depending on what properties of the fragment they are describing), a sufficiently expressive language should be developed, and a sufficiently adept parsing mechanism should be applied. We address these problems from a technical standpoint in detail in Section A.4.

A.3.3 Repository access

In Figure A.6, repository access is represented in two modules, the Derby Connector and Query Generator that takes the translated elaboration requests from the parser and passes them off to the second module, the Process Fragment Repository. This is a repository of abstractions, currently focused primarily on process fragments, but with a general schema that can accommodate a variety of representations. These two modules reflect the two main concerns when it comes to the collection of process fragments (be they subprocesses or agent behavior specifications): on the one hand backend storage and organization (in the repository), and on the other frontend accessibility through the specified request language (in the query generator).

For backend storage, the following organization seems ideal:

- Every elaboration step would have one .ljx file associated with it. Thus, if we wanted to elaborate the step “submit ballot,” all possible elaboration for it would reside in submitBallot.ljx. Within that ljx file, every elaboration would have one module associated with it. These can be named according to the unique identifier assigned to each elaboration.

There are several advantages to this approach. First, it allows for the explicit browsing of process fragments, which was indicated as a desirable feature during the proposal defense. Second, it allows for an organization approach that should easily port to other representations (each module could contain a pointer to a process definition in a process definition language different from Little-JIL if an integration mechanism is specified). Third, it allows for easy identification of only specific elaborations to be copied over when a new process family gets generated.

The abstract common core (ACC) gets copied over to the new family along with all the necessary cions. There are pros and cons to this technique. On the plus side, explicit cloning allows for multiple families to be generated, analyzed, and manipulated at once. This is important for easy reuse of predefined abstractions from the repository without worrying about side effects from trying to rename abstractions when they are converted into cions. Most important, cloning the abstractions allows for the selective grafting of cions at specific step appearances within the progenitor (whereas an elaboration step with the same name as another could request a different cion through appearance-level specification in PLAGE).

The constraint specification is flexible enough that runtime late-binding-like grafting of cions is a natural extension to consider for future work.

The cons of coping over the ACC and necessary abstractions are the same as problems related to code cloning. However, these are mitigated by two factors:

- only the necessary abstractions are used in cion construction, which in most cases would constrain how much of the repository is copied.
- both the ACC and abstractions are very small typically and the advantages of the ability to create and analyze multiple different families far outweighs the costs.

Notes on current resource management

Currently, the resource model for a Little-JIL process is modeled as separate and external to the .ljx representation that Eclipse knows about (i.e. the resource model lives outside of Visual-JIL and is not viewable or browsable from the process coordination diagram). Each resource has a type (e.g., Doctor), a set of attributes (e.g., name = Phil, shift = morning), as well as a capacity (default is 1), which determines how many agenda items may be assigned to each resource at any one point. Each resource has different capabilities corresponding to the activities it can perform. Note, for the Little-JIL Elaborator, the simplified abstraction selection relies on the elaboration step names as capabilities. For PLAGE, this can be extended to match the more general ROMEO convention.

Currently, each capability has a skill level associated with it, as well as an effort value for each job (effort here takes precedence over capacity, meaning that an agent that can perform three easy tasks all at once may only be able to perform one difficult task at a time). Capability currently corresponds to a “queryname” field that can be specified for each resource interface specification in Visual-JIL. What this means is that within a step interface in Visual-JIL, we can specify a query request for an agent and say that we need an agent that has the capability to give a shot, which would typically be a nurse, but can be substituted for a doctor in different circumstances (depending on skill level and availability). In VSRM, this corresponds directly to the resource type, and agents there can similarly have more than one “type” (e.g. negotiation participant *and* party representative), but only one is their primary type.

All of this is quite similar to the simplified resource model used by the Elaborator, however, there are major differences that limit any reuse of the current implementation. The current ROMEO resource model is completely external to the .ljx representation of the process and is represented externally in a hard-coded xml schema file. There is a database specification that interacts with that schema file to execute the queries from the .ljx file, however, there is no way to add instances to the xml schema through the database specification, or to make changes to the model from the database view. That precludes any use cases that involve a user specifying a new behavior or changing an existing behavior from working. Additionally, and more important, the current resource model does not specify any agent behaviors. There are some hard-coded constraints on how long an agent might take for a certain capability within the xml schema, for example, but there is no connection between any specific behaviors, or any apparent way to easily extend this specification to allow the schema to interact with other .ljx files (in fact, that seems difficult to do given that the schema is external to what we would refer to as the “common core” process specification as well).

A.3.4 User Experience

To easily identify elaboration steps, we suggest using a checkbox in a special “Properties” dialog of each reference step to indicate whether it is a usual reference (and therefore expected to resolve to exactly one specification), or whether it is an elaboration step (and therefore can reasonably be expected to resolve to more than one elaboration, and if such was the case, all appropriate abstraction cions would be grafted onto a choice step, which would in term be grafted onto the elaboration step). This would effectively set the Boolean flag that accompanies reference steps in the internal form. However, the current Visual-JIL infrastructure does not treat reference steps as other steps and consequently provides no properties dialog, so one must be implemented. For elaboration steps, the request language specified in Chapter 4 allows the process developer to indicate the desired properties of the process fragments being requested. Although writing an external specification is functionally sufficient to support PLAGE, the eventual development of a GUI-based editor (perhaps within the new Properties frame), would be desirable. For the purposes of providing an incremental development plan, such UX features should be secondary to a working prototype. The development plan in the next section indicates a reasonable alternative through the manual specification of the configuration file (an identical configuration file would be created behind the scenes in the case of a GUI editor, so the parsing activity remains unchanged). The suggestion is not to omit the prospect of developing a GUI editor in the future, but to make the file human-readable and modifiable in the meanwhile. This allows for iterative development where an initial working prototype may only rely on the configuration file being manually edited in a text editor, and later being generated through interaction with the coordination diagram.

A.4 Detailed Design and Implementation Plan

This section presents the detailed implementation plan for the Little-JIL Elaborator and the PLAGE framework and it contains the proposed interface design for each module. We begin with some high-level notes about integrating with the existing infrastructure whenever applicable, and present and an outline of features. These are, whenever possible, laid out in incremental fashion so in most cases a feature A that precedes a feature B indicates that B depends on A in some way (the semantics of these connections, e.g. uses, extends, aggregates, and so on, are further clarified when necessary). Moreover, features are ordered by criticality, so that the first few are necessary for a functional prototype, while latter ones are more concerned with improving the user interface (UI) or user experience (UX) rather than the functionality of the system. Points of interface with the existing framework are indicated in italics in the writing below.

A.4.1 Middleware: Little-JIL Internal Form integration

Feature: resolving an elaboration step to one or more elaborations. Recall that there is a resolution activity that happens every time a Little-JIL diagram uses a reference step, and this resolution activity needs exactly one declaration of the reference step and throws a MultipleResolutionsException if there is more than one within declaration a single module. For the purposes of the Elaborator, if a reference step is designated as an elaboration step, then the resolution activity is overridden to allow for this case specifically. The Resolver is designed to resolve based on a String input, so there were no changes made to the interface for the Little-JIL Elaborator. The modified resolution activity still returns a single item to conform to interface well-formedness constraints. For requests where a single fragment is requested (using the best flag as described) or matches the specification, this is trivial. For requests allowing multiple fragments, a newly constructed choice step is returned, onto which all matching cions are grafted as outlined in Chapter 4.

Interfaces with: Little-JIL Internal Form: laser.lj.ast.resolved; Little-JIL XML: laser.lj.xml.resolved, requires extension of laser.lj.xml.resolved.Resolver

A.4.2 Database access

Using the one ljx file per elaboration step, one module per elaboration scheme outlined above allows for straightforward indexing using a frontend relational database. As discussed, the process fragment repository would provide an access mechanism to any and all elaborations as needed, but this backend storage schema provides additional convenience for the process developers. When adding new elaborations to the repository, there is already a clear place for the new fragment to go (check if there exists a process definition for that step according to the predefined naming scheme and if so, add a new module to it; else create a new appropriately named process definition to match the step being elaborated first, then create a new within). When retrieving fragments, the access pointers to all fragments would be appropriately stored in the database that would return a set of subprocesses that match the request. A relational database seems like a reasonable way to store the attributes of the different elaborations and pointers to them. Since the file names can be constructed according to the schema outlined earlier, the prototype can be tested incrementally without initially using a database server as an intermediary and using direct access to process fragments. Given process fragments may have multiple attributes, a new schema was developed to ensure that all attributes are stored separately and that when necessary, a LEFT JOIN operation on the two tables allows for fast retrieval and useful indexing.

Interfaces with: Little-JIL XML laser/lj/xml/PersistentStoreInterfaceImpl

A.4.3 Elaboration Selection

After looking into several different parsing technologies including ANTLR, Xerces, and Java DOM, we selected ANTLR. ANTLR (ANother Tool for Language Recognition) was used to build a customized parser based on a well-defined grammar for the configuration file. ANTLR also supports the specification of arbitrarily complex grammars and provides a suite of tools both for grammar and for parser development and testing. Another option was to specify the file using a predefined XML schema in line with current Little-JIL implementation (note that most of the information about a process is stored as metadata and not as structured XML and therefore the XML representation of a Little-JIL process is not human-readable). Defining an XML schema would have indicated using an XML-specific parsing tool such as Xerces. Xerces provides flexible parsing and wrappers for a lot of the parsing functionality found in Java. Since Java already provides some built-in support for XML parsing, using a DOM (Document Object Model) XML may have been sufficient. The problem with choosing either of these technologies was that they would not be easily extensible with respect to future representations, so if the process or any of the elaboration specifications were to be encoded in another language, the system would break. Perhaps more important, looking forward, using a simpler technology would confine PLAGE to being a Little-JIL-only family generator. ANTLR, on the other hand, provides for an extensible solution and would allow for the system to be more easily tailored to other process definition languages if desired.

Interfaces with: JulietteRuntime, requires changes to laser/juliette/runtime/Runtime Configurator

Requires extensions to the configuration object; requires development of simple grammar and parser that the RuntimeConfigurator will connect to as services.

A.4.4 Frontend and UX concerns

Once the configuration file works, the graphic editor would have to do four main things:

1. Implement a properties window for references, which is currently not supported *Interfaces with: Visual-JIL plugin, requires changes to src/laser/little-jil/eclipse/property-sources*

2. Provide a checkbox in the properties window to mark a reference step as an elaboration step *Interfaces with: Visual-JIL plugin, requires changes to src/laser/little-jil/eclipse/property-sources*

3. Provide a wizard to specify the request for each elaboration step. *Interfaces with: Visual-JIL plugin, requires changes to src/laser/little-jil/eclipse/dialogs*

4. Navigate to a step declaration by clicking on a reference (already supported in the latest version of Visual-JIL) *Interfaces with: Little-JIL Search plugin, requires changes to src/laser/little-jil/eclipse/search*

5. Generate a diagram which includes the specified elaborations by grafting a choice tree onto an elaboration step (optional—already supported in the Little-JIL Elaborator)

Interfaces with: Little-JIL Search plugin, requires changes to src/laser/little-jil/eclipse/search, Visual-JIL plugin, requires changes to src/laser/little-jil/eclipse/views, src/laser/little-jil/eclipse/views/resolved

Notes on the software development plan and hierarchical decomposition by feature

At a high level, the prioritization for the development plan of PLAGE would look like this:

1. Resolution
2. Elaboration selection
3. Repository access
4. UI/UX

Where each later activity is lower precedence than all of the earlier activities, and depends on some or all earlier features to function correctly. More detailed notes on each activity are included in the previous section, the following plan focuses largely on the order or incremental development. Most of the interface specifications have already been completed. Major modules of PLAGE are also in place as indicated below but need the middleware to be integrated into the system.

1. Resolution
    (a) Extend laser.lj.xml.resolved.Resolver to override the resolution activity for reference steps that has been annotated to be elaboration steps so that they resolve to more than one option. *Changes in the Little-JIL Internal Form: laser.lj.ast.resolved and significant changes to Little-JIL XML: laser.lj.xml.resolved.*
        i. Case I: Allow only one elaboration and ensure that existing test cases pass (this is no different from regular reference steps). *Implementation complete.*

ii. Case II: Allow more than one elaboration and create additional test cases to ensure that multiple elaborations are getting grafted correctly. *Implementation complete.*

    A. Create a generation activity that generates the choice tree that is to be grafted. *Implementation complete.*

    B. Copy/union the interface declarations for the children steps *Implementation complete.*

    C. For integration testing, developer can hardcode in test elaborations so that the configuration file is not necessary yet and pass those into the Resolver

2. Elaboration selection

   (a) Develop a grammar matching the specification language for the selection of appropriate elaborations. *Implementation complete.*

   (b) Develop test cases to ensure selections are being made correctly. *Implementation complete.*

   (c) Develop parser for the grammar. Ensure parser parses requests correctly, resulting in correct selections (these selections can be made from memory without a database to begin with). *Implementation complete.*

      i. Create a setup as outlined above of .ljx files and corresponding modules to contain the elaborations we are testing with.

   (d) Integrate elaboration selection with resolution.

      i. Create a configuration file containing elaboration steps.

      ii. The file should specify steps and request specifications to be parsed.

      iii. Integrate file into .jul file (Juliette Runtime laser.juliette.runtime.Runtime Configurator.java) → this is needed only for runtime and exporting the process specification and is not necessary for a functional PLAGE system

(e) Eclipse plugin development

- Modifying the resolver *Implementation complete.*
- Adding a preference pane/wizard for specifying configuration file
- Building configuration file editor
- Grafting cions to the common core and debugging configuration setup *Implementation complete.*
- Building static visualizer of the resolved structure of fragments, having fragments refer back to original diagram specifications *Implementation complete.*

3. Repository access

(a) Setup a database schema. *Implementation complete.*

(b) Ensure the database can be accessed by the Process Fragment Manager.

(c) Integrate the database access with the existing .ljx files and corresponding modules from the previous activity.

(d) Integrate database access with elaboration selection.

(e) Integrate database access and elaboration selection with the resolution activity.

4. UI/UX

(a) Build a properties window for reference steps, which are currently treated as different from other step kinds in the Visual-JIL editor and do not have user-modifiable properties *(changes to src/laser/little-jil/eclipse/property-sources, src/laser/little-jil/eclipse/dialogs).*

(b) Provide a checkbox in the properties window to mark a reference step as an elaboration step *(changes to src/laser/little-jil/eclipse/property-sources).*

(c) Integrate with the elaboration selection activity to allow for the parser to access this information from the GUI and augment the configuration file with it when parsing a request.

(d) Provide a wizard to specify the request for each elaboration step *(changes to src/laser/little-jil/eclipse/dialogs)*.

(e) Integrate with the elaboration selection activity to allow for the parser to access the request specification from the GUI and build the elaboration selection configuration file from scratch.

(f) Navigate to a step declaration by clicking on a reference *(supported by latest Visual-JIL)*.

(g) Generate a diagram which includes the specified elaborations by grafting a choice tree onto an elaboration step—in the Little-JIL Elaborator this functionality is already supported with a simple drag-and-drop *Implementation complete*.

APPENDIX B

DEFINITIONS OF TECHNICAL TERMS

- An *abstract common core (ACC)* is an incomplete process definition where one or more reference steps are denoted to be elaboration steps.

- An *elaboration step* is a special kind of reference step that can resolve to more than one declaration. It stitch can specify any number of artifacts as long as they cannot conflict with the declaration to which the reference resolves.

- *Resolution* is the process of replacing reference steps with explicit declarations. Regular reference steps resolve to a single declaration. Elaboration steps involve an elaboration activity specified as follows:

  - A process family is populated through a *generation activity*. The generation activity takes in an abstract common core as a progenitor, a family request specification, and a repository of abstractions that will be used to satisfy the request.

  - A *cion* is an abstraction specification that is "grafted" onto an elaboration step in the abstract common core in the generation activity. Once all appropriate (i.e. matching the request specification) cions have been grafted, a *process family* has been generated or instantiated. Note that different families can be generated from the same progenitor abstract common core by varying the request specification, the repository used, or both.

    A cion takes one of two forms: it is either a single abstraction from the repository (in the case when "best" was requested or that abstraction was only one matching the specification), or it is a choice step newly generated to have all matching abstractions as its children. In both cases, the cion's interfaces are bound to match the elaboration step's inputs and outputs unlike an abstraction from the repository.

Note the exceptional case when there may be no abstractions that match the requested specification and therefore no abstractions to generate a cion: in that case, one should generate a "dummy" cion essentially consisting of a single leaf step with the appropriate inputs and outputs and a matching name (one could think of a reference step being replaced by a simple leaf step, which essentially tells the process that the details of how this step is to be executed are left up to the assigned agent).

- The *abstraction repository* contains two types of abstractions: process fragment elaborations and agent behavior elaborations. Each abstraction has a formal interface that is not bound to artifact instances (i.e. to specific values of inputs and outputs) until the abstraction becomes (part of) a cion.

APPENDIX C

THE SPLC CASE STUDY EXTENDED PROCESS FAMILY

The extended process family based on the SPLC case study used as a reference point in the scalability experiments presented in Chapter 5 is presented in its entirety here as an HTML narration. This is the complete family including all elaborations used for the evaluation, and is presented as HTML in order to show its artifact and resource specifications. The table of contents including an indented outline of the entire process follows.

Table Of Contents

- submit ballot, elb 2
  - fake ballot
  - record one ballot
    - record voter ballot
    - record election official ballot
- handle spoiled paper ballot
  - let voter have another paper ballot
    - issue correct ballot type
    - fill out paper ballot
  - let voter have third paper ballot
    - let voter have another paper ballot

- record voter preference, elb 1
  - fill out electronic ballot
    - make selections
    - confirm selections
  - submit e-ballot
    - submit e-ballot, elb 0
      - commit to repository
      - issue unique id
      - print receipt
    - submit e-ballot, elb 1
      - commit to repository
      - print receipt
    - submit e-ballot, elb 2
      - commit to repository
    - submit e-ballot, elb 3
      - fake e-ballot
      - record one e-ballot
        - record voter e-ballot
        - record DRE e-ballot
      - print receipt
  - handle faulty voting machine exception
    - let voter have another electronic ballot
      - issue correct ballot type
      - fill out electronic ballot
    - let voter vote at another machine
      - record voter preference

- ⊖ handle spoiled electronic ballot
  - → let voter have another electronic ballot
  - → let voter have third ballot
    - → let voter have another electronic ballot
- ✓ handle wrong candidate selected
  - → record voter preference, elb 1

- ✓ handle voter already checked off exception
  - → let voter vote with provisional ballot
    - ■ issue provisional ballot
    - ○ record voter preference
- ✓ handle voter not on voting roll exception
  - → let voter vote with provisional ballot

- ■ add unused ballots to repository
- ■ count all ballots in teams of three and reconcile with ballot cover sheet
- ✓ handle faulty voting machine

- → count votes
  - → count votes from all precincts
    - → perform ballot and vote count
      - → perform reconciliations
        - ■ reconcile voting roll and cover sheet
        - ■ reconcile total ballots and counted ballots
      - ■ scan votes
      - ■ confirm tallies match
      - ■ add vote count to vote total
      - ⊖ handle discrepancy
        - → rescan
          - ■ scan votes
          - ■ override software
        - → manually count votes
          - ■ read out voter preference
          - ■ confirm voter preference is read correctly
          - = tally votes
            - ■ increment and announce appropriate tally
            - ■ increment and announce appropriate tally
    - → perform random audit
      - ■ select precincts for 1% mandatory manual audit
      - → manually count votes

- ▪ confirm audit tallies are consistent
- ▪ report vote totals to Secretary of State

- ✓ recount votes
  - → recount selected ballots
    - → count paper ballots in teams of three
      - → manually count votes
    - → check sum consistency
      - ▪ confirm sum matches total
    - ⊖ handle recount discrepancy
  - ▪ report vote totals to Secretary of State

Additionally, all the steps within the extended SPLC case study process family are defined along with their corresponding artifact and resource specifications, and any exceptional control flow in detail below.

Legend   Index of step names

Conduct Election

● Before starting "conduct election", the resource *agent* must be acquired.

→ To "conduct election", pre-polling activities (This step must be done at least once.) (The cardinality of this step is controlled by the following parameter: agent.) , then prepare for and conduct election at precinct (This step must be done at least once.) (The cardinality of this step is controlled by the following parameter: agent.) , and finally count votes.

Pre-Polling Activities

↕ The *votingRoll* is required to "pre-polling activities" and may be modified during this step.

● Before starting "pre-polling activities", the resource *agent* must be acquired.

→ To "pre-polling activities", register one voter (This step must be done at least once.) (The cardinality of this step is controlled by the following parameter: agent.) .

Register One Voter

↕ The *votingRoll* is required to "register one voter" and may be modified during this step.

● Before starting "register one voter", the resource *agent* must be acquired.

→ To "register one voter", verify eligible and then complete the registration.

Verify Eligible

● Before starting "verify eligible", the resource *agent* must be acquired.

○ The resource *voter* is used in this step.

= To "verify eligible", the following need to be done in any order (including simultaneously), verify voter is a resident of the correct precint and confirm voter's age is at least 18.

E If *Voter Ineligible Exception*, then complete the step "register one voter".

Verify Voter Is A Resident Of The Correct Precint

○ The resources *voter* and *agent* are used in this step.

→ To "verify voter is a resident of the correct precint", request utility bill or other proof of residence.

E If *Voter Ineligible Exception*, then rethrow the exception *Voter Ineligible Exception*.

Request Utility Bill Or Other Proof Of Residence

○ The resources *voter* and *agent* are used in this step.

→ To "request utility bill or other proof of residence", present proof of residence and then verify proof of residence.

E If *Voter Ineligible Exception*, then rethrow the exception *Voter Ineligible Exception*.

Present Proof Of Residence

↑ Successful completion of the step "present proof of residence" should yield the *proofOfResidence*.

○ The resource *agent* is used in this step.

Verify Proof Of Residence

↓ The *proofOfResidence* is required to "verify proof of residence".

○ The resource *agent* is used in this step.

E If *Voter Ineligible Exception*, then rethrow the exception *Voter Ineligible Exception*.

Confirm Voter's Age Is At Least 18

○ The resource *agent* is used in this step.

E If *Voter Ineligible Exception*, then rethrow the exception *Voter Ineligible Exception*.

Complete The Registration

↕ The *votingRoll* is required to "complete the registration" and may be modified during this step.

● Before starting "complete the registration", the resource *agent* must be acquired.

Prepare For And Conduct Election At Precinct

↕ The *coverSheet*, *votingRoll*, and *repository* are required to "prepare for and conduct election at precinct" and may be modified during this step.

● Before starting "prepare for and conduct election at precinct", the resource *agent* must be acquired.

→ To "prepare for and conduct election at precinct", the following need to be done in the listed order

- pre-polling checks
- authenticate and vote
  - This step must be done at least once.
  - The cardinality of this step is controlled by the following parameter: agent.
- add unused ballots to repository
- count all ballots in teams of three and reconcile with ballot cover sheet

Pre-Polling Checks

↓ The *votingRoll* and *repository* are required to "pre-polling checks".

↑ Successful completion of the step "pre-polling checks" should yield the *coverSheet*.

● Before starting "pre-polling checks", the resource *agent* must be acquired.

= To "pre-polling checks", the following need to be done in any order (including simultaneously),

- check voting machines
  - This step must be done at least once.
  - The cardinality of this step is controlled by the following parameter: agent.
- check if voting rolls are present
- check if supplies are present
- check if ballots are present

E If *Faulty Voting Machine Exception*, then handle faulty voting machine and then complete "prepare for and conduct election at precinct".

Check Voting Machines

↓ The *voteRepository* is required to "check voting machines".

● Before starting "check voting machines", the resource *agent* must be acquired.

→ To "check voting machines", assure totals are initialized to zero.

E If *Faulty Voting Machine Exception*, then rethrow the exception *Faulty Voting Machine Exception*.

Assure Totals Are Initialized To Zero

↓ The *voteRepository* is required to "assure totals are initialized to zero".

● Before starting "assure totals are initialized to zero", the resource *agent* must be acquired.

E If *Faulty Voting Machine Exception*, then rethrow the exception *Faulty Voting Machine Exception*.

Check If Voting Rolls Are Present

↓ The *votingRoll* is required to "check if voting rolls are present".

● Before starting "check if voting rolls are present", the resource *agent* must be acquired.

Check If Supplies Are Present

● Before starting "check if supplies are present", the resource *agent* must be acquired.

Check If Ballots Are Present

↓ The *voteRepository* is required to "check if ballots are present".

↑ Successful completion of the step "check if ballots are present" should yield the *coverSheet*.

● Before starting "check if ballots are present", the resource *agent* must be acquired.

Authenticate And Vote

↕ The *votingRoll* and *repository* are required to "authenticate and vote" and may be modified during this step.

● Before starting "authenticate and vote", the resource *agent* must be acquired.

→ To "authenticate and vote", the following need to be done in the listed order

- perform pre-vote authentication
- check off voter as voted
- issue correct ballot type
- record voter preference

Perform Pre-Vote Authentication

↕ The *votingRoll* and *voterName* are required to "perform pre-vote authentication" and may be modified during this step.

● Before starting "perform pre-vote authentication", the resource *agent* must be acquired.

○ The resource *voter* is used in this step.

→ To "perform pre-vote authentication", the following need to be done in the listed order

- state name
- confirm voter's residence and name against voting roll
- verify voter has not voted
- sign voting roll

E If *Voter Not Registered Exception*, then handle voter not on voting roll exception and then complete "authenticate and vote".

E If *Voter Already Checked Off Exception*, then handle voter already checked off exception and then complete "authenticate and vote".

State Name

↑ Successful completion of the step "state name" should yield the *voterName*.

○ The resource *agent* is used in this step.

Confirm Voter's Residence And Name Against Voting Roll

↓ The *votingRoll* and *voterName* are required to "confirm voter's residence and name against voting roll".

○ The resource *agent* is used in this step.

E If *Voter Not Registered Exception*, then rethrow the exception *Voter Not Registered Exception*.

Verify Voter Has Not Voted

↓ The *votingRoll* and *voterName* are required to "verify voter has not voted".

○ The resource *agent* is used in this step.

E If *Voter Already Checked Off Exception*, then rethrow the exception *Voter Already Checked Off Exception*.

Sign Voting Roll

↕ The *votingRoll* is required to "sign voting roll" and may be modified during this step.

○ The resource *agent* is used in this step.

Check Off Voter As Voted

↓ The *voterName* is required to "check off voter as voted".

↕ The *votingRoll* is required to "check off voter as voted" and may be modified during this step.

● Before starting "check off voter as voted", the resource *agent* must be acquired.

Issue Correct Ballot Type

↑ Successful completion of the step "issue correct ballot type" should yield the *ballot*.

● Before starting "issue correct ballot type", the resource *agent* must be acquired.

⇸ To "issue correct ballot type", the following should be tried, in the listed order until one succeeds, issue regular ballot or issue provisional ballot.

Issue Regular Ballot

↑ Successful completion of the step "issue regular ballot" should yield the *regularBallot*.

○ The resource *agent* is used in this step.

E If *Voter Ineligible For Regular Ballot Exception*, then continue with the next step.

Issue Provisional Ballot

↑ Successful completion of the step "issue provisional ballot" should yield the *provisionalBallot*.

○ The resource *agent* is used in this step.

Record Voter Preference

↓ The *voterPreference* is required to "record voter preference".

↕ The *repository* is required to "record voter preference" and may be modified during this step.

○ The resource *agent* is used in this step.

✪ To "record voter preference", one of the following should be chosen to perform: record voter preference, elb 0 or record voter preference, elb 1.

Record Voter Preference, Elb 0

↓ The *voterPreference* is required to "record voter preference, elb 0".

↕ The *repository* is required to "record voter preference, elb 0" and may be modified during this step.

○ The resource *agent* is used in this step.

→ To "record voter preference, elb 0", fill out paper ballot and then submit ballot.

Fill Out Paper Ballot

↕ The *voterPreference* is required to "fill out paper ballot" and may be modified during this step.

○ The resource *agent* is used in this step.

E If *Voter Spoiled Ballot Exception*, then handle spoiled paper ballot and then continue with the next step.

Submit Ballot

↓ The *voterPreference* is required to "submit ballot".

↕ The *repository* is required to "submit ballot" and may be modified during this step.

● Before starting "submit ballot", the resource *agent* must be acquired.

◇ To "submit ballot", one of the following should be chosen to perform: submit ballot, elb 0, submit ballot, elb 1, or submit ballot, elb 2.

Submit Ballot, Elb 0

↓ The *voterPreference* is required to "submit ballot, elb 0".

↕ The *repository* is required to "submit ballot, elb 0" and may be modified during this step.

● Before starting "submit ballot, elb 0", the resource *agent* must be acquired.

→ To "submit ballot, elb 0", tamper with ballot and then record ballot.

Tamper With Ballot

↕ The *voterPreference* is required to "tamper with ballot" and may be modified during this step.

Record Ballot

↓ The *voterPreference* is required to "record ballot".

↕ The *repository* is required to "record ballot" and may be modified during this step.

Submit Ballot, Elb 1

↓ The *voterPreference* is required to "submit ballot, elb 1".

↕ The *repository* is required to "submit ballot, elb 1" and may be modified during this step.

● Before starting "submit ballot, elb 1", the resource *agent* must be acquired.

→ To "submit ballot, elb 1", record ballot.

Submit Ballot, Elb 2

↓ The *voterPreference* is required to "submit ballot, elb 2".

↕ The *repository* is required to "submit ballot, elb 2" and may be modified during this step.

● Before starting "submit ballot, elb 2", the resource *agent* must be acquired.

→ To "submit ballot, elb 2", fake ballot and then record one ballot.

Fake Ballot

↑ Successful completion of the step "fake ballot" should yield the *fakedVoterPreference*.

Record One Ballot

↓ The *fakedVoterPreference* and *voterPreference* are required to "record one ballot".

↕ The *repository* is required to "record one ballot" and may be modified during this step.

○ To "record one ballot", one of the following should be chosen to perform: record voter ballot or record election official ballot.

Record Voter Ballot

↓ The *voterPreference* is required to "record voter ballot".

↕ The *repository* is required to "record voter ballot" and may be modified during this step.

Record Election Official Ballot

↓ The *fakedVoterPreference* is required to "record election official ballot".

↕ The *repository* is required to "record election official ballot" and may be modified during this step.

Handle Spoiled Paper Ballot

↕ The *voterPreference* is required to "handle spoiled paper ballot" and may be modified during this step.

⇸ To "handle spoiled paper ballot", the following should be tried, in the listed order until one succeeds, let voter have another paper ballot or let voter have third paper ballot.

Let Voter Have Another Paper Ballot

↑ Successful completion of the step "let voter have another paper ballot" should yield the *newBallot*.

● Before starting "let voter have another paper ballot", the resources *voter* and *agent* must be acquired.

→ To "let voter have another paper ballot", issue correct ballot type and then fill out paper ballot.

E If *Voter Spoiled Ballot Exception*, then continue with the next step.

Let Voter Have Third Paper Ballot

↕ The *voterPreference* is required to "let voter have third paper ballot" and may be modified during this step.

→ To "let voter have third paper ballot", let voter have another paper ballot.

Record Voter Preference, Elb 1

↓ The *voterPreference* is required to "record voter preference, elb 1".

↕ The *repository* is required to "record voter preference, elb 1" and may be modified during this step.

○ The resource *agent* is used in this step.

→ To "record voter preference, elb 1", fill out electronic ballot and then submit e-ballot.

Fill Out Electronic Ballot

↕ The *voterPreference* is required to "fill out electronic ballot" and may be modified during this step.

○ The resource *agent* is used in this step.

→ To "fill out electronic ballot", make selections and then confirm selections.

E If *Voter Spoiled Ballot Exception*, then handle spoiled electronic ballot and then continue with the next step.

E If *Wrong Candidate Selected*, then handle wrong candidate selected and then complete "record voter preference, elb 1".

Make Selections

↑ Successful completion of the step "make selections" should yield the *voterPreference*.

Confirm Selections

↕ The *voterPreference* is required to "confirm selections" and may be modified during this step.

E If *Voter Spoiled Ballot Exception*, then rethrow the exception *Voter Spoiled Ballot Exception*.

E If *Wrong Candidate Selected*, then rethrow the exception *Wrong Candidate Selected*.

Submit E-Ballot

↓ The *voterPreference* is required to "submit e-ballot".

↕ The *repository* is required to "submit e-ballot" and may be modified during this step.

⊙ To "submit e-ballot", one of the following should be chosen to perform:

- submit e-ballot, elb 0
- submit e-ballot, elb 1
- submit e-ballot, elb 2
- submit e-ballot, elb 3

E If *Faulty Voting Machine Exception*, then handle faulty voting machine exception and then complete "record voter preference, elb 1".

Submit E-Ballot, Elb 0

↓ The *voterPreference* is required to "submit e-ballot, elb 0".

↕ The *repository* is required to "submit e-ballot, elb 0" and may be modified during this step.

● Before starting "submit e-ballot, elb 0", the resource *agent* must be acquired.

→ To "submit e-ballot, elb 0", commit to repository, then issue unique id, and finally print receipt.

E If *Faulty Voting Machine Exception*, then rethrow the exception *Faulty Voting Machine Exception*.

Commit To Repository

↓ The *voterPreference* is required to "commit to repository".

↕ The *repository* is required to "commit to repository" and may be modified during this step.

E If *Faulty Voting Machine Exception*, then rethrow the exception *Faulty Voting Machine Exception*.

Issue Unique Id

↓ The *voterPreference* is required to "issue unique id".

Print Receipt

↓ The *voterPreference* is required to "print receipt".

Submit E-Ballot, Elb 1

↓ The *voterPreference* is required to "submit e-ballot, elb 1".

↕ The *repository* is required to "submit e-ballot, elb 1" and may be modified during this step.

→ To "submit e-ballot, elb 1", commit to repository and then print receipt.

E If *Faulty Voting Machine Exception*, then rethrow the exception *Faulty Voting Machine Exception*.

Submit E-Ballot, Elb 2

↓ The *voterPreference* is required to "submit e-ballot, elb 2".

↕ The *repository* is required to "submit e-ballot, elb 2" and may be modified during this step.

● Before starting "submit e-ballot, elb 2", the resource *agent* must be acquired.

→ To "submit e-ballot, elb 2", commit to repository.

E If *Faulty Voting Machine Exception*, then rethrow the exception *Faulty Voting Machine Exception*.

Submit E-Ballot, Elb 3

↓ The *voterPreference* is required to "submit e-ballot, elb 3".

↕ The *repository* is required to "submit e-ballot, elb 3" and may be modified during this step.

→ To "submit e-ballot, elb 3", fake e-ballot, then record one e-ballot, and finally print receipt.

Fake E-Ballot

↑ Successful completion of the step "fake e-ballot" should yield the *fakedVoterPreference*.

Record One E-Ballot

↓ The *fakedVoterPreference* and *voterPreference* are required to "record one e-ballot".

↕ The *repository* is required to "record one e-ballot" and may be modified during this step.

○ To "record one e-ballot", one of the following should be chosen to perform: record voter e-ballot or record DRE e-ballot.

Record Voter E-Ballot

↓ The *voterPreference* is required to "record voter e-ballot".

↕ The *repository* is required to "record voter e-ballot" and may be modified during this step.

Record DRE E-Ballot

↓ The *fakedVoterPreference* is required to "record DRE e-ballot".

↕ The *repository* is required to "record DRE e-ballot" and may be modified during this step.

Handle Faulty Voting Machine Exception

→ To "handle faulty voting machine exception", let voter have another electronic ballot and then let voter vote at another machine.

Let Voter Have Another Electronic Ballot

↑ Successful completion of the step "let voter have another electronic ballot" should yield the *newBallot*.

● Before starting "let voter have another electronic ballot", the resources *voter* and *agent* must be acquired.

→ To "let voter have another electronic ballot", issue correct ballot type and then fill out electronic ballot.

E If *Voter Spoiled Ballot Exception*, then rethrow the exception *Voter Spoiled Ballot Exception*.

Let Voter Vote At Another Machine

↓ The *voterPreference* is required to "let voter vote at another machine".

→ To "let voter vote at another machine", record voter preference.

Handle Spoiled Electronic Ballot

↕ The *voterPreference* is required to "handle spoiled electronic ballot" and may be modified during this step.

↛ To "handle spoiled electronic ballot", the following should be tried, in the listed order until one succeeds, let voter have another electronic ballot or let voter have third ballot.

Let Voter Have Third Ballot

↕ The *voterPreference* is required to "let voter have third ballot" and may be modified during this step.

→ To "let voter have third ballot", let voter have another electronic ballot.

Handle Wrong Candidate Selected

↓ The *voterPreference* is required to "handle wrong candidate selected".

↕ The *repository* is required to "handle wrong candidate selected" and may be modified during this step.

○ The resource *voter* is used in this step.

→ To "handle wrong candidate selected", record voter preference, elb 1.

Handle Voter Already Checked Off Exception

↕ The *repository* and *ballot* are required to "handle voter already checked off exception" and may be modified during this step.

● Before starting "handle voter already checked off exception", the resource *agent* must be acquired.

○ The resource *voter* is used in this step.

→ To "handle voter already checked off exception", let voter vote with provisional ballot.

Let Voter Vote With Provisional Ballot

↕ The *repository* and *ballot* are required to "let voter vote with provisional ballot" and may be modified during this step.

○ The resources *voter* and *agent* are used in this step.

→ To "let voter vote with provisional ballot", issue provisional ballot and then record voter preference.

Handle Voter Not On Voting Roll Exception

↕ The *repository* and *ballot* are required to "handle voter not on voting roll exception" and may be modified during this step.

● Before starting "handle voter not on voting roll exception", the resource *agent* must be acquired.

○ The resource *voter* is used in this step.

→ To "handle voter not on voting roll exception", let voter vote with provisional ballot.

Add Unused Ballots To Repository

↕ The *coverSheet* and *repository* are required to "add unused ballots to repository" and may be modified during this step.

Count All Ballots In Teams Of Three And Reconcile With Ballot Cover Sheet

↓ The *voteRepository* is required to "count all ballots in teams of three and reconcile with ballot cover sheet".

↕ The *coverSheet* is required to "count all ballots in teams of three and reconcile with ballot cover sheet" and may be modified during this step.

Handle Faulty Voting Machine

● Before starting "handle faulty voting machine", the resource *agent* must be acquired.

Count Votes

↓ The *coverSheet*, *repository*, and *roster* are required to "count votes".

↕ The *totalTallies* is required to "count votes" and may be modified during this step.

● Before starting "count votes", the resource *agent* must be acquired.

→ To "count votes", count votes from all precincts.

E If *Vote Count Inconsistent Exception*, then recount votes and then complete "conduct election".

Count Votes From All Precincts

↓ The *coverSheet*, *repository*, and *roster* are required to "count votes from all precincts".

↕ The *totalTallies* is required to "count votes from all precincts" and may be modified during this step.

○ The resource *agent* is used in this step.

→ To "count votes from all precincts", perform ballot and vote count (This step must be done at least once.) (The cardinality of this step is controlled by the following parameter: agent.) , then perform random audit, and finally report vote totals to Secretary of State.

E If *Vote Count Inconsistent Exception*, then rethrow the exception *Vote Count Inconsistent Exception*.

Perform Ballot And Vote Count

↓ The *coverSheet*, *repository*, and *roster* are required to "perform ballot and vote count".

↑ Successful completion of the step "perform ballot and vote count" should yield the *tallies* and *totalTallies*.

● Before starting "perform ballot and vote count", the resource *agent* must be acquired.

→ To "perform ballot and vote count", the following need to be done in the listed order

- perform reconciliations
- scan votes
- confirm tallies match
- add vote count to vote total

Perform Reconciliations

↓ The *coverSheet*, *repository*, and *roster* are required to "perform reconciliations ".

→ To "perform reconciliations ", reconcile voting roll and cover sheet and then reconcile total ballots and counted ballots.

Reconcile Voting Roll And Cover Sheet

↓ The *coverSheet* and *roster* are required to "reconcile voting roll and cover sheet".

● Before starting "reconcile voting roll and cover sheet", the resource *agent* must be acquired.

Reconcile Total Ballots And Counted Ballots

↓ The *coverSheet* and *repository* are required to "reconcile total ballots and counted ballots".

● Before starting "reconcile total ballots and counted ballots", the resource *agent* must be acquired.

Scan Votes

↓ The *repository* is required to "scan votes".

↑ Successful completion of the step "scan votes" should yield the *tallies*.

● Before starting "scan votes", the resource *agent* must be acquired.

Confirm Tallies Match

↓ The *coverSheet* is required to "confirm tallies match".

↕ The *tallies* is required to "confirm tallies match" and may be modified during this step.

● Before starting "confirm tallies match", the resource *agent* must be acquired.

E If *Vote Count Inconsistent Exception*, then handle discrepancy and then continue with the next step.

Add Vote Count To Vote Total

↓ The *tallies* is required to "add vote count to vote total".

↕ The *totalTallies* is required to "add vote count to vote total" and may be modified during this step.

● Before starting "add vote count to vote total", the resource *agent* must be acquired.

Handle Discrepancy

↓ The *repository* is required to "handle discrepancy".

↕ The *tallies* is required to "handle discrepancy" and may be modified during this step.

● Before starting "handle discrepancy", the resource *agent* must be acquired.

○ To "handle discrepancy", one of the following should be chosen to perform: rescan or manually count votes.

Rescan

↓ The *repository* is required to "rescan".

↑ Successful completion of the step "rescan" should yield the *tallies*.

→ To "rescan", scan votes and then override software.

Override Software

● Before starting "override software", the resource *agent* must be acquired.

Manually Count Votes

↓ The *voteRepository* is required to "manually count votes".

↕ The *tallies* is required to "manually count votes" and may be modified during this step.

→ To "manually count votes", read out voter preference, then confirm voter preference is read correctly, and finally tally votes.

Read Out Voter Preference

↓ The *voterPreference* is required to "read out voter preference".

Confirm Voter Preference Is Read Correctly

↓ The *voterPreference* is required to "confirm voter preference is read correctly".

Tally Votes

↕ The *tallies* is required to "tally votes" and may be modified during this step.

= To "tally votes", the following need to be done in any order (including simultaneously), increment and announce appropriate tally and increment and announce appropriate tally.

E If *Vote Count Inconsistent Exception*, then restart the step "manually count votes".

Increment And Announce Appropriate Tally

↕ The *tallies* is required to "increment and announce appropriate tally" and may be modified during this step.

○ The resource *agent:team* is used in this step.

E If *Vote Count Inconsistent Exception*, then rethrow the exception *Vote Count Inconsistent Exception*.

Perform Random Audit

↓ The *repository* is required to "perform random audit".

↕ The *tallies* and *auditTallies* are required to "perform random audit" and may be modified during this step.

→ To "perform random audit", select precincts for 1% mandatory manual audit, then manually count votes (The cardinality of this step is controlled by the following parameter: voteRepository.) , and finally confirm audit tallies are consistent.

E If *Vote Count Inconsistent Exception*, then rethrow the exception *Vote Count Inconsistent Exception*.

Select Precincts For 1% Mandatory Manual Audit

↕ The *tallies* is required to "select precincts for 1% mandatory manual audit" and may be modified during this step.

Confirm Audit Tallies Are Consistent

↓ The *tallies* and *auditTallies* are required to "confirm audit tallies are consistent".

E If *Vote Count Inconsistent Exception*, then rethrow the exception *Vote Count Inconsistent Exception*.

Report Vote Totals To Secretary Of State

↓ The *tallies* is required to "report vote totals to Secretary of State".

● Before starting "report vote totals to Secretary of State", the resource *agent* must be acquired.

Recount Votes

↓ The *repository* and *originalTallies* are required to "recount votes".

● Before starting "recount votes", the resource *agent* must be acquired.

→ To "recount votes", recount selected ballots and then report vote totals to Secretary of State.

Recount Selected Ballots

↓ The *repository* and *originalTallies* are required to "recount selected ballots".

↕ The *recountedVoteTotals* is required to "recount selected ballots" and may be modified during this step.

● Before starting "recount selected ballots", the resource *agent* must be acquired.

→ To "recount selected ballots", count paper ballots in teams of three and then check sum consistency.

Count Paper Ballots In Teams Of Three

↓ The *repository* is required to "count paper ballots in teams of three".

↑ Successful completion of the step "count paper ballots in teams of three" should yield the *recountedVoteTotals*.

→ To "count paper ballots in teams of three", manually count votes.

E If *Recount Discrepancy Exception*, then handle recount discrepancy and then continue with the next step.

▲

Check Sum Consistency

↓ The *originalTallies* and *recountedVoteTotals* are required to "check sum consistency".

○ The resource *agent* is used in this step.

→ To "check sum consistency", confirm sum matches total.

E If *Recount Discrepancy Exception*, then handle recount discrepancy and then continue with the next step.

Confirm Sum Matches Total

↓ The *originalTallies* and *recountedVoteTotals* are required to "confirm sum matches total".

○ The resource *agent* is used in this step.

E If *Recount Discrepancy Exception*, then rethrow the exception *Recount Discrepancy Exception*.

Handle Recount Discrepancy

○ The resource *agent* is used in this step.

APPENDIX D

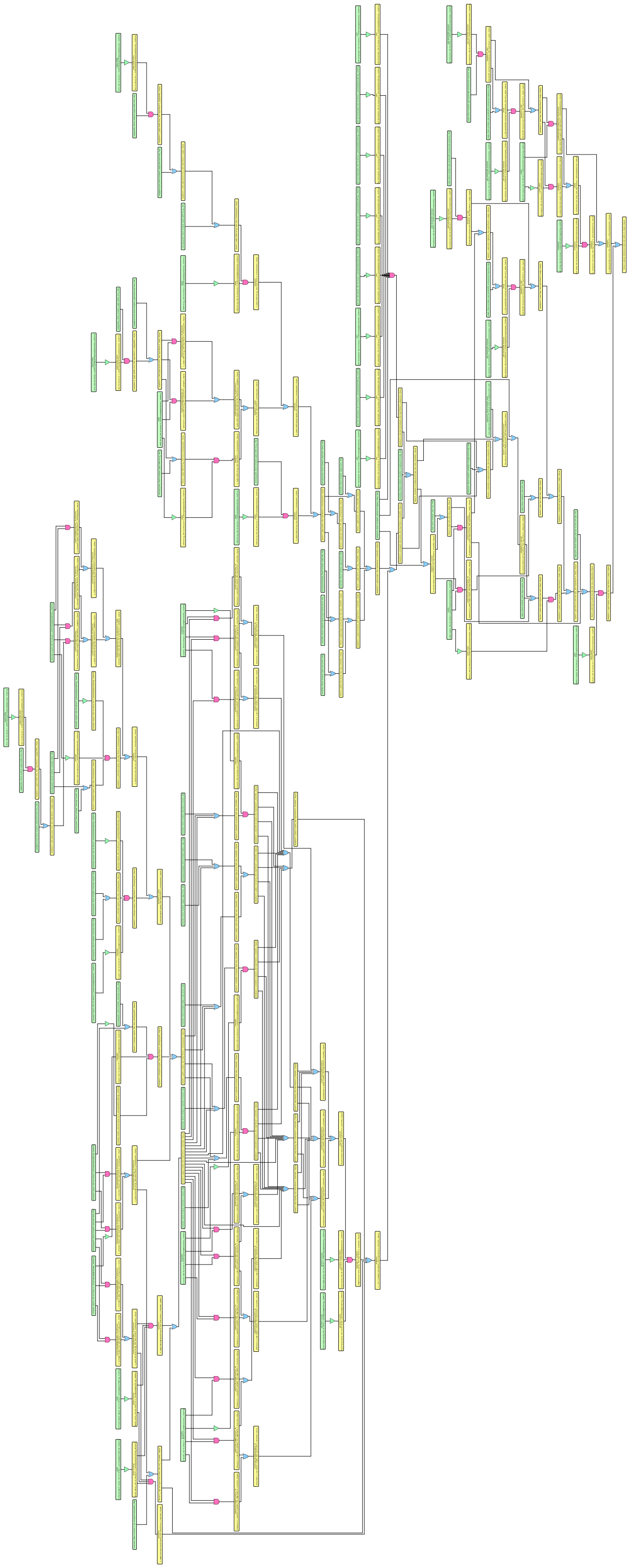
EVALUATION ARTIFACTS FOR THE SPLC CASE STUDY EXTENDED PROCESS FAMILY

For each set of experiments performed and presented in Chapter 5, we present the resulting artifacts here for completeness.

D.1 Fault Tree Analysis for the Extended SPLC Case Study

Fault tree analysis (FTA) was performed on the extended SPLC process family including all abstractions presented in Appendix C above. The hazard under consideration is the step results are posted on Yolo County Elections Office website receiving the wrong voteCount artifact (of type VoteCount.java, specified as a JavaBean using the eponymous artifact mode in Visual-JIL) as input. This experiment was performed on a 3GHz dual-core Intel Core i7 processor with 16GB of physical RAM. It was run under an Eclipse virtual machine (VM) with 4,096MB–16,384MB memory allowance, and the Little-JIL Analysis Toolset translator was also given a 4,096MB–16,384MB memory allowance.

The fault tree is included below, followed by the corresponding Minimal Cut Sets (MCSs).



MCS 1) {

Step “add vote count to vote total” produces wrong “totalTallies”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 2-1) {

Exception “VoteCountInconsistentException” is thrown by step “confirm tallies match”

Step “check if ballots are present” produces wrong “coverSheet”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 2-2) {

Exception “VoteCountInconsistentException” is not thrown by step “confirm tallies match”

Step “check if ballots are present” produces wrong “coverSheet”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 3-1) {

Exception “VoteCountInconsistentException” is thrown by step “confirm tallies match”

Step “add unused ballots to repository” produces wrong “repository”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 3-2) {

Exception "VoteCountInconsistentException" is not thrown by step "confirm tallies match"

Step "add unused ballots to repository" produces wrong "repository"

Exception "VoteCountInconsistentException" is not thrown by step "perform random audit"

}

MCS 4-1) {

Exception "VoteCountInconsistentException" is thrown by step "confirm tallies match"

Step "record voter ballot" produces wrong "repository"

Exception "VoteCountInconsistentException" is not thrown by step "perform random audit"

}

MCS 4-2) {

Exception "VoteCountInconsistentException" is not thrown by step "confirm tallies match"

Step "record voter ballot" produces wrong "repository"

Exception "VoteCountInconsistentException" is not thrown by step "perform random audit"

}

MCS 5-1) {

Exception "VoteCountInconsistentException" is thrown by step "confirm tallies match"

Exception "VoterSpoiledBallotException" is not thrown by step "fill out paper ballot"

Step "fill out paper ballot" produces wrong "voterPreference"

Exception "VoteCountInconsistentException" is not thrown by step "perform random audit"

}

MCS 5-2) {

Exception “VoteCountInconsistentException” is not thrown by step “confirm tallies match”

Exception “VoterSpoiledBallotException” is not thrown by step “fill out paper ballot”

Step “fill out paper ballot” produces wrong “voterPreference”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 6-1) {

Exception “VoteCountInconsistentException” is thrown by step “confirm tallies match”

Step “fill out paper ballot” produces wrong “voterPreference”

Exception “VoterSpoiledBallotException” is not thrown by step “fill out paper ballot”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 6-2) {

Exception “VoteCountInconsistentException” is not thrown by step “confirm tallies match”

Step “fill out paper ballot” produces wrong “voterPreference”

Exception “VoterSpoiledBallotException” is not thrown by step “fill out paper ballot”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 7-1) {

Exception “VoteCountInconsistentException” is thrown by step “confirm tallies match”

Exception “VoterIneligibleForRegularBallotException” is not thrown by step “issue regular ballot”

Step “issue regular ballot” produces wrong “regularBallot”

Exception “VoterSpoiledBallotException” is not thrown by step “fill out paper ballot”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 7-2) {

Exception “VoteCountInconsistentException” is thrown by step “confirm tallies match”

Exception “VoterIneligibleForRegularBallotException” is not thrown by step “issue regular ballot”

Step “issue regular ballot” produces wrong “regularBallot”

Exception “VoterSpoiledBallotException” is thrown by step “fill out paper ballot”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 7-3) {

Exception “VoteCountInconsistentException” is not thrown by step “confirm tallies match”

Exception “VoterIneligibleForRegularBallotException” is not thrown by step “issue regular ballot”

Step “issue regular ballot” produces wrong “regularBallot”

Exception “VoterSpoiledBallotException” is not thrown by step “fill out paper ballot”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 7-4) {

Exception “VoteCountInconsistentException” is not thrown by step “confirm tallies match”

Exception “VoterIneligibleForRegularBallotException” is not thrown by step “issue regular ballot”

Step “issue regular ballot” produces wrong “regularBallot”

Exception “VoterSpoiledBallotException” is thrown by step “fill out paper ballot”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 8-1) {

Exception “VoteCountInconsistentException” is thrown by step “confirm tallies match”

Step “issue provisional ballot” produces wrong “provisionalBallot”

Exception “VoterSpoiledBallotException” is not thrown by step “fill out paper ballot”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 8-2) {

Exception “VoteCountInconsistentException” is thrown by step “confirm tallies match”

Step “issue provisional ballot” produces wrong “provisionalBallot”

Exception “VoterSpoiledBallotException” is thrown by step “fill out paper ballot”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 8-3) {

Exception “VoteCountInconsistentException” is not thrown by step “confirm tallies match”

Step “issue provisional ballot” produces wrong “provisionalBallot”

Exception “VoterSpoiledBallotException” is not thrown by step “fill out paper ballot”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 8-4) {

Exception “VoteCountInconsistentException” is not thrown by step “confirm tallies match”

Step “issue provisional ballot” produces wrong “provisionalBallot”

Exception “VoterSpoiledBallotException” is thrown by step “fill out paper ballot”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 9-1) {

Exception “VoteCountInconsistentException” is thrown by step “confirm tallies match”

Exception “VoterSpoiledBallotException” is not thrown by step “fill out paper ballot”

Step “issue regular ballot” produces wrong “regularBallot”

Exception “VoterIneligibleForRegularBallotException” is not thrown by step “issue regular ballot”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 9-2) {

Exception “VoteCountInconsistentException” is not thrown by step “confirm tallies match”

Exception “VoterSpoiledBallotException” is not thrown by step “fill out paper ballot”

Step “issue regular ballot” produces wrong “regularBallot”

Exception “VoterIneligibleForRegularBallotException” is not thrown by step “issue regular ballot”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 10-1) {

Exception “VoteCountInconsistentException” is thrown by step “confirm tallies match”

Exception “VoterSpoiledBallotException” is not thrown by step “fill out paper ballot”

Step “issue provisional ballot” produces wrong “provisionalBallot”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 10-2) {

Exception “VoteCountInconsistentException” is not thrown by step “confirm tallies match”

Exception “VoterSpoiledBallotException” is not thrown by step “fill out paper ballot”

Step “issue provisional ballot” produces wrong “provisionalBallot”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 11-1) {

Exception “VoteCountInconsistentException” is thrown by step “confirm tallies match”

Exception “VoterSpoiledBallotException” is not thrown by step “fill out paper ballot”

Step “fill out paper ballot” produces wrong “voterPreference”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 11-2) {

Exception “VoteCountInconsistentException” is not thrown by step “confirm tallies match”

Exception “VoterSpoiledBallotException” is not thrown by step “fill out paper ballot”

Step “fill out paper ballot” produces wrong “voterPreference”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 12-1) {

Exception “VoteCountInconsistentException” is thrown by step “confirm tallies match”

Step “record election official ballot” produces wrong “repository”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 12-2) {

Exception “VoteCountInconsistentException” is not thrown by step “confirm tallies match”

Step “record election official ballot” produces wrong “repository”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 13-1) {

Exception “VoteCountInconsistentException” is thrown by step “confirm tallies match”

Step “fake ballot” produces wrong “fakedVoterPreference”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 13-2) {

Exception “VoteCountInconsistentException” is not thrown by step “confirm tallies match”

Step “fake ballot” produces wrong “fakedVoterPreference”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 14-1) {

Exception “VoteCountInconsistentException” is thrown by step “confirm tallies match”

Step “record ballot” produces wrong “repository”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 14-2) {

Exception “VoteCountInconsistentException” is not thrown by step “confirm tallies match”

Step “record ballot” produces wrong “repository”

Exception “VoteCountInconsistentException” is not thrown by step “perform random

audit”

}

MCS 15-1) {

Exception “VoteCountInconsistentException” is thrown by step “confirm tallies match”

Step “record ballot” produces wrong “repository”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 15-2) {

Exception “VoteCountInconsistentException” is not thrown by step “confirm tallies match”

Step “record ballot” produces wrong “repository”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 16-1) {

Exception “VoteCountInconsistentException” is thrown by step “confirm tallies match”

Step “tamper with ballot” produces wrong “voterPreference”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 16-2) {

Exception “VoteCountInconsistentException” is not thrown by step “confirm tallies match”

Step “tamper with ballot” produces wrong “voterPreference”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 17-1) {

Exception “VoteCountInconsistentException” is thrown by step “confirm tallies match”

Step “commit to repository” produces wrong “repository”

Exception “FaultyVotingMachineException” is not thrown by step “commit to repository”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 17-2) {

Exception “VoteCountInconsistentException” is not thrown by step “confirm tallies match”

Step “commit to repository” produces wrong “repository”

Exception “FaultyVotingMachineException” is not thrown by step “commit to repository”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 18-1) {

Exception “VoteCountInconsistentException” is thrown by step “confirm tallies match”

Exception “FaultyVotingMachineException” is not thrown by step “commit to repository”

Step “commit to repository” produces wrong “repository”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 18-2) {

Exception “VoteCountInconsistentException” is not thrown by step “confirm tallies match”

Exception “FaultyVotingMachineException” is not thrown by step “commit to repository”

Step “commit to repository” produces wrong “repository”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 19-1) {

Exception “VoteCountInconsistentException” is thrown by step “confirm tallies match”

Step “fake e-ballot” produces wrong “fakedVoterPreference”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 19-2) {

Exception “VoteCountInconsistentException” is not thrown by step “confirm tallies match”

Step “fake e-ballot” produces wrong “fakedVoterPreference”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 20-1) {

Exception “VoteCountInconsistentException” is thrown by step “confirm tallies match”

Step “record DRE e-ballot” produces wrong “repository”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 20-2) {

Exception “VoteCountInconsistentException” is not thrown by step “confirm tallies match”

Step “record DRE e-ballot” produces wrong “repository”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 21-1) {

Exception “VoteCountInconsistentException” is thrown by step “confirm tallies match”

Step “make selections” produces wrong “voterPreference”

Exception “VoterSpoiledBallotException” is thrown by step “confirm selections”

Exception “WrongCandidateSelected” is not thrown by step “handle spoiled electronic ballot”

Exception “WrongCandidateSelected” is not thrown by step “handle spoiled electronic ballot”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 21-2) {

Exception “VoteCountInconsistentException” is not thrown by step “confirm tallies match”

Step “make selections” produces wrong “voterPreference”

Exception “VoterSpoiledBallotException” is thrown by step “confirm selections”

Exception “WrongCandidateSelected” is not thrown by step “handle spoiled electronic ballot”

Exception “WrongCandidateSelected” is not thrown by step “handle spoiled electronic ballot”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 22-1) {

Exception “VoteCountInconsistentException” is thrown by step “confirm tallies match”

Step “record voter preference, elb 1” produces wrong “repository”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 22-2) {

Exception “VoteCountInconsistentException” is not thrown by step “confirm tallies match”

Step “record voter preference, elb 1” produces wrong “repository”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 23-1) {

Exception “VoteCountInconsistentException” is thrown by step “confirm tallies match”

Step “make selections” produces wrong “voterPreference”

Exception “WrongCandidateSelected” is thrown by step “confirm selections”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 23-2) {

Exception “VoteCountInconsistentException” is not thrown by step “confirm tallies match”

Step “make selections” produces wrong “voterPreference”

Exception “WrongCandidateSelected” is thrown by step “confirm selections”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 24-1) {

Exception “VoteCountInconsistentException” is thrown by step “confirm tallies match”

Exception “VoterSpoiledBallotException” is not thrown by step “confirm selections”

Step “make selections” produces wrong “voterPreference”

Exception “WrongCandidateSelected” is not thrown by step “confirm selections”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 24-2) {

Exception “VoteCountInconsistentException” is not thrown by step “confirm tallies match”

Exception “VoterSpoiledBallotException” is not thrown by step “confirm selections”

Step “make selections” produces wrong “voterPreference”

Exception “WrongCandidateSelected” is not thrown by step “confirm selections”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 25-1) {

Exception “VoteCountInconsistentException” is thrown by step “confirm tallies match”

Exception “VoterSpoiledBallotException” is not thrown by step “confirm selections”

Step “confirm selections” produces wrong “voterPreference”

Exception “WrongCandidateSelected” is not thrown by step “confirm selections”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 25-2) {

Exception “VoteCountInconsistentException” is not thrown by step “confirm tallies match”

Exception “VoterSpoiledBallotException” is not thrown by step “confirm selections”

Step “confirm selections” produces wrong “voterPreference”

Exception “WrongCandidateSelected” is not thrown by step “confirm selections”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 26-1) {

Exception “VoteCountInconsistentException” is thrown by step “confirm tallies match”

Step “make selections” produces wrong “voterPreference”

Exception “VoterSpoiledBallotException” is not thrown by step “confirm selections”

Exception “WrongCandidateSelected” is not thrown by step “confirm selections”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 26-2) {

Exception “VoteCountInconsistentException” is not thrown by step “confirm tallies match”

Step “make selections” produces wrong “voterPreference”

Exception “VoterSpoiledBallotException” is not thrown by step “confirm selections”

Exception “WrongCandidateSelected” is not thrown by step “confirm selections”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 27-1) {

Exception “VoteCountInconsistentException” is thrown by step “confirm tallies match”

Step “confirm selections” produces wrong “voterPreference”

Exception “VoterSpoiledBallotException” is not thrown by step “confirm selections”

Exception “WrongCandidateSelected” is not thrown by step “confirm selections”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 27-2) {

Exception “VoteCountInconsistentException” is not thrown by step “confirm tallies match”

Step “confirm selections” produces wrong “voterPreference”

Exception “VoterSpoiledBallotException” is not thrown by step “confirm selections”

Exception “WrongCandidateSelected” is not thrown by step “confirm selections”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 28-1) {

Exception “VoteCountInconsistentException” is thrown by step “confirm tallies match”

Exception “VoterSpoiledBallotException” is thrown by step “confirm selections”

Step “make selections” produces wrong “voterPreference”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 28-2) {

Exception “VoteCountInconsistentException” is not thrown by step “confirm tallies match”

Exception “VoterSpoiledBallotException” is thrown by step “confirm selections”

Step “make selections” produces wrong “voterPreference”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 29-1) {

Exception “VoteCountInconsistentException” is thrown by step “confirm tallies match”

Exception “VoterSpoiledBallotException” is thrown by step “confirm selections”

Step “issue regular ballot” produces wrong “regularBallot”

Exception “VoterIneligibleForRegularBallotException” is not thrown by step “issue regular ballot”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 29-2) {

Exception “VoteCountInconsistentException” is not thrown by step “confirm tallies match”

Exception “VoterSpoiledBallotException” is thrown by step “confirm selections”

Step “issue regular ballot” produces wrong “regularBallot”

Exception “VoterIneligibleForRegularBallotException” is not thrown by step “issue regular ballot”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 30-1) {

Exception “VoteCountInconsistentException” is thrown by step “confirm tallies match”

Exception “VoterSpoiledBallotException” is thrown by step “confirm selections”

Step “issue provisional ballot” produces wrong “provisionalBallot”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 30-2) {

Exception “VoteCountInconsistentException” is not thrown by step “confirm tallies match”

Exception “VoterSpoiledBallotException” is thrown by step “confirm selections”

Step “issue provisional ballot” produces wrong “provisionalBallot”

Exception “VoteCountInconsistentException” is not thrown by step “perform random

audit”

}

MCS 31-1) {

Exception “VoteCountInconsistentException” is thrown by step “confirm tallies match”

Exception “WrongCandidateSelected” is not thrown by step “confirm selections”

Step “make selections” produces wrong “voterPreference”

Exception “VoterSpoiledBallotException” is not thrown by step “confirm selections”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 31-2) {

Exception “VoteCountInconsistentException” is not thrown by step “confirm tallies match”

Exception “WrongCandidateSelected” is not thrown by step “confirm selections”

Step “make selections” produces wrong “voterPreference”

Exception “VoterSpoiledBallotException” is not thrown by step “confirm selections”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 32-1) {

Exception “VoteCountInconsistentException” is thrown by step “confirm tallies match”

Exception “WrongCandidateSelected” is not thrown by step “confirm selections”

Step “confirm selections” produces wrong “voterPreference”

Exception “VoterSpoiledBallotException” is not thrown by step “confirm selections”

Exception “VoteCountInconsistentException” is not thrown by step “perform random

audit”

}

MCS 32-2) {

Exception “VoteCountInconsistentException” is not thrown by step “confirm tallies match”

Exception “WrongCandidateSelected” is not thrown by step “confirm selections”

Step “confirm selections” produces wrong “voterPreference”

Exception “VoterSpoiledBallotException” is not thrown by step “confirm selections”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 33-1) {

Exception “VoteCountInconsistentException” is thrown by step “confirm tallies match”

Step “record voter e-ballot” produces wrong “repository”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 33-2) {

Exception “VoteCountInconsistentException” is not thrown by step “confirm tallies match”

Step “record voter e-ballot” produces wrong “repository”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 34-1) {

Exception “VoteCountInconsistentException” is thrown by step “confirm tallies match”

Step “commit to repository” produces wrong “repository”

Exception “FaultyVotingMachineException” is not thrown by step “commit to repository”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 34-2) {

Exception “VoteCountInconsistentException” is not thrown by step “confirm tallies match”

Step “commit to repository” produces wrong “repository”

Exception “FaultyVotingMachineException” is not thrown by step “commit to repository”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 35-1) {

Exception “VoteCountInconsistentException” is thrown by step “confirm tallies match”

Step “scan votes” produces wrong “tallies”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 35-2) {

Exception “VoteCountInconsistentException” is not thrown by step “confirm tallies match”

Step “scan votes” produces wrong “tallies”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 36) {

Step “increment and announce appropriate tally” produces wrong “tallies”

Exception “VoteCountInconsistentException” is not thrown by step “increment and announce appropriate tally”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 37) {

Step “increment and announce appropriate tally” produces wrong “tallies”

Exception “VoteCountInconsistentException” is not thrown by step “increment and announce appropriate tally”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 38) {

Step “scan votes” produces wrong “tallies”

Exception “VoteCountInconsistentException” is not thrown by step “perform random audit”

}

MCS 39) {

Exception "VoteCountInconsistentException" is not thrown by step "confirm tallies match"

Step "confirm tallies match" produces wrong "tallies"

Exception "VoteCountInconsistentException" is not thrown by step "perform random audit"

}

MCS 40) {

Exception "VoteCountInconsistentException" is not thrown by step "confirm tallies match"

Step "add unused ballots to repository" produces wrong "coverSheet"

Exception "VoteCountInconsistentException" is not thrown by step "perform random audit"

}

MCS 41) {

Exception "VoteCountInconsistentException" is not thrown by step "confirm tallies match"

Step "count all ballots in teams of three and reconcile with ballot cover sheet" produces wrong "coverSheet"

Exception "VoteCountInconsistentException" is not thrown by step "perform random audit"

}

MCS 42) {

Step "increment and announce appropriate tally" produces wrong "tallies"

Exception "VoteCountInconsistentException" is not thrown by step "increment and announce appropriate tally"

Exception "RecountDiscrepancyException" is thrown by step "confirm new totals match"

}

MCS 43) {

Exception “VoteCountInconsistentException” is not thrown by step “increment and announce appropriate tally”

Step “increment and announce appropriate tally” produces wrong “tallies”

Exception “RecountDiscrepancyException” is thrown by step “confirm new totals match”

}

MCS 44-1) {

Step “increment and announce appropriate tally” produces wrong “tallies”

Exception “VoteCountInconsistentException” is not thrown by step “increment and announce appropriate tally”

Exception “RecountDiscrepancyException” is thrown by step “confirm new totals match”

Exception “RecountDiscrepancyException” is not thrown by step “check sum consistency”

}

MCS 44-2) {

Exception “RecountDiscrepancyException” is not thrown by step “confirm new totals match”

Step “increment and announce appropriate tally” produces wrong “tallies”

Exception “VoteCountInconsistentException” is not thrown by step “increment and announce appropriate tally”

Exception “RecountDiscrepancyException” is not thrown by step “check sum consistency”

}

MCS 45-1) {

Exception “VoteCountInconsistentException” is not thrown by step “increment and announce appropriate tally”

Step “increment and announce appropriate tally” produces wrong “tallies”

Exception “RecountDiscrepancyException” is thrown by step “confirm new totals match”

Exception “RecountDiscrepancyException” is not thrown by step “check sum consistency”

}

MCS 45-2) {

Exception “RecountDiscrepancyException” is not thrown by step “confirm new totals match”

Exception “VoteCountInconsistentException” is not thrown by step “increment and announce appropriate tally”

Step “increment and announce appropriate tally” produces wrong “tallies”

Exception “RecountDiscrepancyException” is not thrown by step “check sum consistency”

}

D.2 Finite-State Verification Results for the Extended SPLC Case Study

The extended SPLC case study was found to violate the property presented in 5.11, given the bindings in Table D.1.

Property Event	Step Name
voter marks ballot	fill out paper ballot
voter marks ballot	fill out electronic ballot
insider marks ballot	fake ballot
insider marks ballot	tamper with ballot
insider marks ballot	fake e-ballot
count ballots	scan votes

Table D.1. Bindings between steps (all steps are specified to be in state COMPLETED) in the Little-JIL process family definition for the extended SPLC case study and events in the property alphabet for the property presented in Figure 5.11.

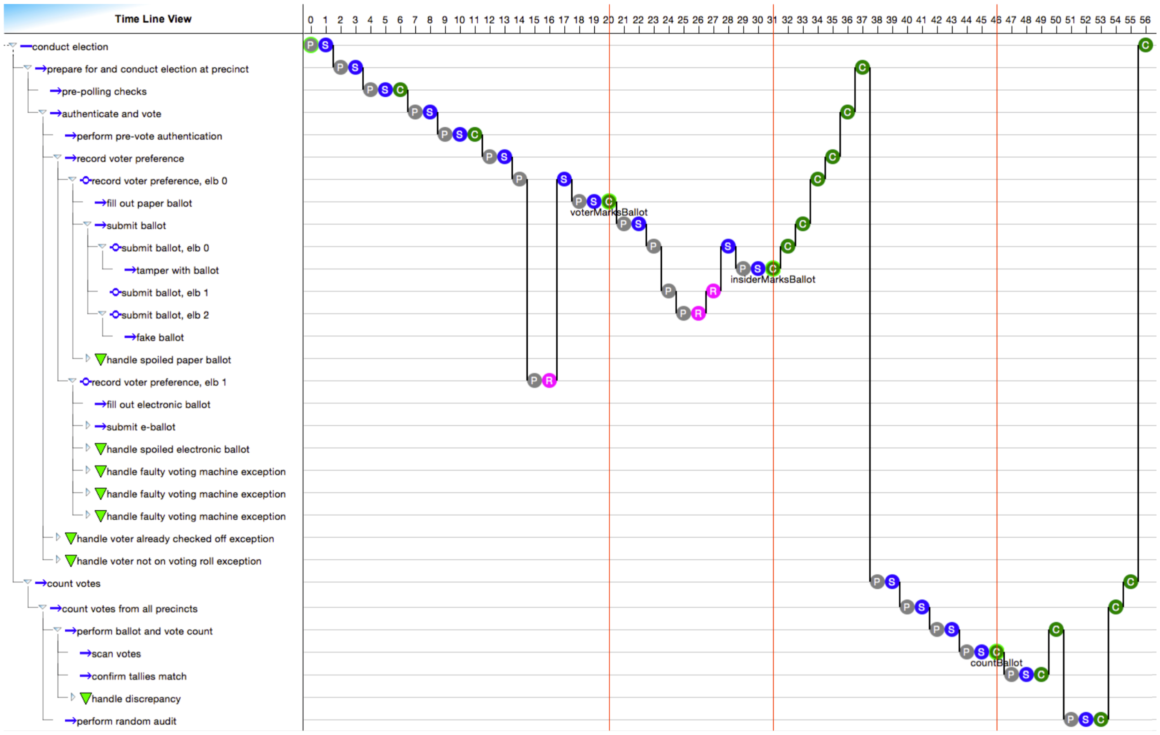


Figure D.1. The extended SPLC case study process family violated the property under consideration. The counterexample trace from FLAVERS is included above.

APPENDIX E

THE VOTE REMOTELY PROCESS FAMILY

The vote remotely process family used for the scalability experiments presented in Chapter 5 is presented in its entirety here as an HTML narration. This is the complete family including all elaborations discussed, and is presented as HTML in order to present its artifact and resource specifications in addition to the coordination diagrams presented earlier. The table of contents including an indented outline of the entire process follows.

Table Of Contents

- submit ballot

- → mark and return ballot, elb 2
  - → print then vote
    - print ballot at home
    - vote on ballot
  - ○ return ballot
    - → mail ballot back
    - → fax ballot back
      - sign release of confidentiality
      - fax ballot
- → mark and return ballot, elb 3
  - vote on ballot
  - → fax ballot back

- ○ collect ballots
  - → collect ballots, elb 0
    - → process internet ballot that was faxed in
      - get voter info from cover sheet and retrieve ballot
      - fax back acknowledgment
      - validate ballot
      - duplicate ballot on ballot stock
      - → process ballot
        - read precinct barcode on ballot
        - place ballot in batch
    - → scan ballots
      - run ballots through scanner
      - keep track of how many ballots from precinct X are in batch Y
      - ⊖ rescan ballots' precinct codes
  - → collect ballots, elb 1
    - → process envelope
      - add envelope to batch
      - → receive and validate envelope
        - → check envelope
          - compare signature on envelope with signature on file
          - place red line through signature
          - add envelope to repository in oder of precinct number
          - ⊖ handle missing signature exception
            - contact putative voter
            - sign envelope

- ⛔ handle signature mismatch exception
  - compare signatures
  - contact putative voter
- open envelope and retrieve ballot
- → process ballot
- → scan ballots
- → collect ballots, elb 2
  - → process email ballot that was mailed in
    - add envelope to batch
    - open envelope and retrieve ballot
    - validate ballot
    - duplicate ballot on ballot stock
    - → process ballot
  - → scan ballots
- → collect ballots, elb 3
  - → process internet ballot
    - → retrieve BoD ballot
      - login to DemocracyLive
      - print ballot
    - validate ballot
    - duplicate ballot on ballot stock
  - → scan internet ballots
    - run ballots through scanner

- → count ballots
  - vote tallies are read off of memory cards from scanners
  - results are posted on Yolo County Elections Office website

Additionally, all the steps within the process family are defined along with their corresponding artifact and resource specifications, and any exceptional control flow in detail below.

Vote Remotely

→ To "vote remotely", the following need to be done in the listed order

- prepare ballots and send them to voters
- mark and return ballot
  - This step must be done at least once.
  - The cardinality of this step is controlled by the following parameter: .
- collect ballots
- count ballots

Prepare Ballots And Send Them To Voters

↕ The *votingRoll* and *ballotCollection* are required to "prepare ballots and send them to voters" and may be modified during this step.

→ To "prepare ballots and send them to voters", acquire list of voters and then distribute ballots.

Acquire List Of Voters

↑ Successful completion of the step "acquire list of voters" should yield the *votingRoll*.

● Before starting "acquire list of voters", the resource *agent* must be acquired.

Distribute Ballots

↓ The *votingRoll* is required to "distribute ballots".

↕ The *ballotCollection* is required to "distribute ballots" and may be modified during this step.

⌖ To "distribute ballots", one of the following should be chosen to perform:

- distribute ballots, elb 0
- distribute ballots, elb 1
- distribute ballots, elb 2
- distribute ballots, elb 3

Distribute Ballots, Elb 0

↓ The *votingRoll* is required to "distribute ballots, elb 0".

↕ The *ballotCollection* is required to "distribute ballots, elb 0" and may be modified during this step.

→ To "distribute ballots, elb 0", prepare individual ballot (This step must be done at least once.) (The cardinality of this step is controlled by the following parameter: voter.) and then fax ballots to voters.

Prepare Individual Ballot

↓ The *voterRegistrationList* is required to "prepare individual ballot".

↕ The *ballotCollection* is required to "prepare individual ballot" and may be modified during this step.

● Before starting "prepare individual ballot", the resource *voter* must be acquired.

→ To "prepare individual ballot", determine precinct for voter and then choose ballot with correct precinct code.

Determine Precinct For Voter

↓ The *voterRegistrationList* is required to "determine precinct for voter".

○ The resources *observer:team*, *agent:team*, and *voter* are used in this step.

Choose Ballot With Correct Precinct Code

↑ Successful completion of the step "choose ballot with correct precinct code" should yield the *ballot*.

↕ The *ballotCollection* is required to "choose ballot with correct precinct code" and may be modified during this step.

○ The resources *observer:team*, *agent:team*, and *voter* are used in this step.

Fax Ballots To Voters

↓ The *votingRoll* and *ballotCollection* are required to "fax ballots to voters".

○ The resources *observer:team* and *agent:team* are used in this step.

Distribute Ballots, Elb 1

↓ The *votingRoll* is required to "distribute ballots, elb 1".

↕ The *ballotCollection* is required to "distribute ballots, elb 1" and may be modified during this step.

→ To "distribute ballots, elb 1", prepare DemocracyLive ballots (This step must be done at least once.) (The cardinality of this step is controlled by the following parameter: voter.) and then email ballot access to voters .

Prepare DemocracyLive Ballots

↓ The *votingRoll* is required to "prepare DemocracyLive ballots".

↕ The *ballotCollection* is required to "prepare DemocracyLive ballots" and may be modified during this step.

● Before starting "prepare DemocracyLive ballots", the resource *voter* must be acquired.

→ To "prepare DemocracyLive ballots", look up races for voter and then generate ballot.

Look Up Races For Voter

↕ The *votingRoll* is required to "look up races for voter" and may be modified during this step.

○ The resources *observer:team* and *agent:team* are used in this step.

Generate Ballot

↑ Successful completion of the step "generate ballot" should yield the *ballot*.

↕ The *ballotCollection* is required to "generate ballot" and may be modified during this step.

○ The resources *observer:team* and *agent:team* are used in this step.

Email Ballot Access To Voters

↓ The *votingRoll* and *ballotCollection* are required to "email ballot access to voters ".

Distribute Ballots, Elb 2

↓ The *votingRoll* is required to "distribute ballots, elb 2".

↕ The *ballotCollection* is required to "distribute ballots, elb 2" and may be modified during this step.

→ To "distribute ballots, elb 2", prepare individual ballot (This step must be done at least once.) (The cardinality of this step is controlled by the following parameter: voter.) and then email ballots to voters.

Email Ballots To Voters

↓ The *votingRoll* and *ballotCollection* are required to "email ballots to voters".

Distribute Ballots, Elb 3

↓ The *votingRoll* is required to "distribute ballots, elb 3".

↕ The *ballotCollection* is required to "distribute ballots, elb 3" and may be modified during this step.

→ To "distribute ballots, elb 3", prepare individual VBM ballot (This step must be done at least once.) (The cardinality of this step is controlled by the following parameter: voter.) , then hand over ballots to post office, and finally deliver ballots to voters.

Prepare Individual VBM Ballot

↓ The *voterRegistrationList* is required to "prepare individual VBM ballot".

↕ The *ballotCollection* is required to "prepare individual VBM ballot" and may be modified during this step.

● Before starting "prepare individual VBM ballot", the resource *voter* must be acquired.

→ To "prepare individual VBM ballot", the following need to be done in the listed order

- determine precinct for voter
- mark envelope with AVID number
- choose ballot with correct precinct code
- place ballot in addressed envelope
- add envolope to mail batch

Mark Envelope With AVID Number

↑ Successful completion of the step "mark envelope with AVID number" should yield the *envelope*.

○ The resources *observer:team*, *agent:team*, and *voter* are used in this step.

Place Ballot In Addressed Envelope

↓ The *ballot* is required to "place ballot in addressed envelope".

↕ The *envelope* is required to "place ballot in addressed envelope" and may be modified during this step.

○ The resources *observer:team* and *agent:team* are used in this step.

Add Envolope To Mail Batch

↓ The *envelope* is required to "add envolope to mail batch".

↕ The *ballotCollection* is required to "add envolope to mail batch" and may be modified during this step.

○ The resources *observer:team* and *agent:team* are used in this step.

Hand Over Ballots To Post Office

↓ The *voterRegistrationList* is required to "hand over ballots to post office".

↕ The *ballotCollection* is required to "hand over ballots to post office" and may be modified during this step.

Deliver Ballots To Voters

↕ The *ballotCollection* is required to "deliver ballots to voters" and may be modified during this step.

Mark And Return Ballot

↕ The *ballot* is required to "mark and return ballot" and may be modified during this step.

⚙ To "mark and return ballot", one of the following should be chosen to perform:

- mark and return ballot, elb 0
- mark and return ballot, elb 1
- mark and return ballot, elb 2
- mark and return ballot, elb 3

Mark And Return Ballot, Elb 0

↕ The *ballot* is required to "mark and return ballot, elb 0" and may be modified during this step.

● Before starting "mark and return ballot, elb 0", the resource *agent* must be acquired.

→ To "mark and return ballot, elb 0", vote on ballot and then mail ballot back.

Vote On Ballot

↕ The *ballot* is required to "vote on ballot" and may be modified during this step.

○ The resource *agent* is used in this step.

Mail Ballot Back

↕ The *ballot* is required to "mail ballot back" and may be modified during this step.

○ The resource *agent* is used in this step.

→ To "mail ballot back", sign envelope, then insert ballot, and finally seal and mail envelope.

Sign Envelope

↕ The *envelope* is required to "sign envelope" and may be modified during this step.

○ The resource *agent* is used in this step.

Insert Ballot

↕ The *ballot* and *envelope* are required to "insert ballot" and may be modified during this step.

Seal And Mail Envelope

↓ The *envelope* is required to "seal and mail envelope".

Mark And Return Ballot, Elb 1

↕ The *ballot* is required to "mark and return ballot, elb 1" and may be modified during this step.

● Before starting "mark and return ballot, elb 1", the resource *agent* must be acquired.

→ To "mark and return ballot, elb 1", authenticate on DemocracyLive, then vote on ballot, and finally submit ballot.

Authenticate On DemocracyLive

↑ Successful completion of the step "authenticate on DemocracyLive" should yield the *ballot*.

○ The resource *voter* is used in this step.

Submit Ballot

↓ The *ballot* is required to "submit ballot".

Mark And Return Ballot, Elb 2

↕ The *ballot* is required to "mark and return ballot, elb 2" and may be modified during this step.

● Before starting "mark and return ballot, elb 2", the resource *agent* must be acquired.

→ To "mark and return ballot, elb 2", print then vote and then return ballot.

Print Then Vote

↕ The *ballot* is required to "print then vote" and may be modified during this step.

○ The resource *agent* is used in this step.

→ To "print then vote", print ballot at home and then vote on ballot.

Print Ballot At Home

↓ The *eBallot* is required to "print ballot at home".

↑ Successful completion of the step "print ballot at home" should yield the *paperBallot*.

● Before starting "print ballot at home", the resource *printer* must be acquired.

○ The resource *agent* is used in this step.

Return Ballot

↕ The *ballot* is required to "return ballot" and may be modified during this step.

○ The resource *agent* is used in this step.

⌖ To "return ballot", one of the following should be chosen to perform: mail ballot back or fax ballot back.

Fax Ballot Back

↕ The *ballot* is required to "fax ballot back" and may be modified during this step.

→ To "fax ballot back", sign release of confidentiality and then fax ballot.

Sign Release Of Confidentiality

↑ Successful completion of the step "sign release of confidentiality" should yield the *release*.

Fax Ballot

↓ The *release* is required to "fax ballot".

↕ The *ballot* is required to "fax ballot" and may be modified during this step.

Mark And Return Ballot, Elb 3

↕ The *ballot* is required to "mark and return ballot, elb 3" and may be modified during this step.

● Before starting "mark and return ballot, elb 3", the resource *agent* must be acquired.

→ To "mark and return ballot, elb 3", vote on ballot and then fax ballot back.

Collect Ballots

↓ The *votingRoll* is required to "collect ballots".

↑ Successful completion of the step "collect ballots" should yield the *ballotCount*.

↕ The *ballotCollection* is required to "collect ballots" and may be modified during this step.

⭘ To "collect ballots", one of the following should be chosen to perform:

- collect ballots, elb 0
- collect ballots, elb 1
- collect ballots, elb 2
- collect ballots, elb 3

Collect Ballots, Elb 0

↓ The *votingRoll* is required to "collect ballots, elb 0".

↑ Successful completion of the step "collect ballots, elb 0" should yield the *ballotCount*.

↕ The *ballotCollection* is required to "collect ballots, elb 0" and may be modified during this step.

→ To "collect ballots, elb 0", process internet ballot that was faxed in (This step must be done at least once.) (The cardinality of this step is controlled by the following parameter: voter.) and then scan ballots.

Process Internet Ballot That Was Faxed In

↓ The *votingRoll* is required to "process internet ballot that was faxed in".

↕ The *ballotCollection* is required to "process internet ballot that was faxed in" and may be modified during this step.

● Before starting "process internet ballot that was faxed in", the resource *voter* must be acquired.

→ To "process internet ballot that was faxed in", the following need to be done in the listed order

- get voter info from cover sheet and retrieve ballot
- fax back acknowledgment
- validate ballot
- duplicate ballot on ballot stock
- process ballot

Get Voter Info From Cover Sheet And Retrieve Ballot

↑ Successful completion of the step "get voter info from cover sheet and retrieve ballot" should yield the *ballot* and *release*.

↕ The *ballotCollection* is required to "get voter info from cover sheet and retrieve ballot" and may be modified during this step.

E If *Missing Ballot Exception*, then complete the step "process internet ballot that was faxed in".

Fax Back Acknowledgment

↓ The *votingRoll* and *release* are required to "fax back acknowledgment".

Validate Ballot

↓ The *votingRoll* and *ballot* are required to "validate ballot".

E If *Provisional Ballot Exception*, then complete the step "process internet ballot that was faxed in".

Duplicate Ballot On Ballot Stock

↓ The *eBallot* is required to "duplicate ballot on ballot stock".

↑ Successful completion of the step "duplicate ballot on ballot stock" should yield the *paperBallot*.

● Before starting "duplicate ballot on ballot stock", the resource *scanner* must be acquired.

Process Ballot

↕ The *ballotCollection* and *ballot* are required to "process ballot" and may be modified during this step.

○ The resources *observer:team* and *agent:team* are used in this step.

→ To "process ballot", read precinct barcode on ballot and then place ballot in batch.

Read Precinct Barcode On Ballot

↕ The *ballot* is required to "read precinct barcode on ballot" and may be modified during this step.

Place Ballot In Batch

↓ The *ballot* is required to "place ballot in batch".

↕ The *ballotCollection* is required to "place ballot in batch" and may be modified during this step.

Scan Ballots

↓ The *ballotCollection* is required to "scan ballots".

↑ Successful completion of the step "scan ballots" should yield the *ballotCount*.

→ To "scan ballots", run ballots through scanner and then keep track of how many ballots from precinct X are in batch Y.

Run Ballots Through Scanner

↓ The *ballotCollection* is required to "run ballots through scanner".

↑ Successful completion of the step "run ballots through scanner" should yield the *ballotCount*.

Keep Track Of How Many Ballots From Precinct X Are In Batch Y

↓ The *ballotCollection* is required to "keep track of how many ballots from precinct X are in batch Y".

↕ The *ballotCount* is required to "keep track of how many ballots from precinct X are in batch Y" and may be modified during this step.

● Before starting "keep track of how many ballots from precinct X are in batch Y", the resource *agent* must be acquired.

E If *Ballots Placed In Wrong Precinct Batch Exception*, then rescan ballots' precinct codes and then continue with the next step.

Rescan Ballots' Precinct Codes

↕ The *ballotCollection* is required to "rescan ballots' precinct codes" and may be modified during this step.

Collect Ballots, Elb 1

↓ The *votingRoll* is required to "collect ballots, elb 1".

↑ Successful completion of the step "collect ballots, elb 1" should yield the *ballotCount*.

↕ The *ballotCollection* is required to "collect ballots, elb 1" and may be modified during this step.

→ To "collect ballots, elb 1", process envelope (This step must be done at least once.) (The cardinality of this step is controlled by the following parameter: voter.) and then scan ballots.

Process Envelope

↓ The *voterRegistrationList* is required to "process envelope".

↕ The *ballotCollection* is required to "process envelope" and may be modified during this step.

● Before starting "process envelope", the resource *voter* must be acquired.

→ To "process envelope", add envelope to batch, then receive and validate envelope, and finally process ballot.

Add Envelope To Batch

↑ Successful completion of the step "add envelope to batch" should yield the *envelope*.

↕ The *ballotCollection* is required to "add envelope to batch" and may be modified during this step.

Receive And Validate Envelope

↓ The *voterRegistrationList* is required to "receive and validate envelope".

↑ Successful completion of the step "receive and validate envelope" should yield the *ballot*.

↕ The *ballotCollection* and *envelope* are required to "receive and validate envelope" and may be modified during this step.

→ To "receive and validate envelope", check envelope and then open envelope and retrieve ballot.

Check Envelope

↓ The *voterRegistrationList* is required to "check envelope".

↕ The *ballotCollection* and *envelope* are required to "check envelope" and may be modified during this step.

→ To "check envelope", compare signature on envelope with signature on file, then place red line through signature, and finally add envelope to repository in oder of precinct number.

Compare Signature On Envelope With Signature On File

↓ The *voterRegistrationList* is required to "compare signature on envelope with signature on file".

↕ The *envelope* is required to "compare signature on envelope with signature on file" and may be modified during this step.

E If *Missing Signature Exception*, then handle missing signature exception and then continue with the next step.

E If *Signature Mismatch Exception*, then handle signature mismatch exception and then continue with the next step.

Place Red Line Through Signature

↕ The *envelope* is required to "place red line through signature" and may be modified during this step.

Add Envelope To Repository In Oder Of Precinct Number

↓ The *envelope* is required to "add envelope to repository in oder of precinct number".

↕ The *ballotCollection* is required to "add envelope to repository in oder of precinct number" and may be modified during this step.

Handle Missing Signature Exception

↕ The *envelope* is required to "handle missing signature exception" and may be modified during this step.

● Before starting "handle missing signature exception", the resources *voter* and *agent* must be acquired.

→ To "handle missing signature exception", contact putative voter and then sign envelope.

Contact Putative Voter

○ The resources *voter* and *agent* are used in this step.

E If *Voter Cannot Be Reached Exception*, then complete the step "handle missing signature exception".

Handle Signature Mismatch Exception

↓ The *voterRegistrationList* is required to "handle signature mismatch exception".

↕ The *envelope* is required to "handle signature mismatch exception" and may be modified during this step.

● Before starting "handle signature mismatch exception", the resources *voter* and *agent* must be acquired.

⇹ To "handle signature mismatch exception", the following should be tried, in the listed order until one succeeds, compare signatures or contact putative voter.

Compare Signatures

↓ The *voterRegistrationList* is required to "compare signatures".

↕ The *envelope* is required to "compare signatures" and may be modified during this step.

○ The resource *agent* is used in this step.

E If *Signature Mismatch Exception*, then continue with the next step.

Open Envelope And Retrieve Ballot

↑ Successful completion of the step "open envelope and retrieve ballot" should yield the *ballot*.

↕ The *ballotCollection* and *envelope* are required to "open envelope and retrieve ballot" and may be modified during this step.

E If *Missing Ballot Exception*, then complete the step "receive and validate envelope".

Collect Ballots, Elb 2

↓ The *votingRoll* is required to "collect ballots, elb 2".

↑ Successful completion of the step "collect ballots, elb 2" should yield the *ballotCount*.

↕ The *ballotCollection* is required to "collect ballots, elb 2" and may be modified during this step.

→ To "collect ballots, elb 2", process email ballot that was mailed in (This step must be done at least once.) (The cardinality of this step is controlled by the following parameter: voter.) and then scan ballots.

Process Email Ballot That Was Mailed In

↓ The *votingRoll* is required to "process email ballot that was mailed in".

↕ The *ballotCollection* is required to "process email ballot that was mailed in" and may be modified during this step.

● Before starting "process email ballot that was mailed in", the resource *voter* must be acquired.

→ To "process email ballot that was mailed in", the following need to be done in the listed order

- add envelope to batch
- open envelope and retrieve ballot
- validate ballot
- duplicate ballot on ballot stock
- process ballot

Collect Ballots, Elb 3

↓ The *votingRoll* is required to "collect ballots, elb 3".

↑ Successful completion of the step "collect ballots, elb 3" should yield the *ballotCount*.

↕ The *ballotCollection* is required to "collect ballots, elb 3" and may be modified during this step.

→ To "collect ballots, elb 3", process internet ballot (This step must be done at least once.) (The cardinality of this step is controlled by the following parameter: voter.) and then scan internet ballots.

Process Internet Ballot

↓ The *votingRoll* is required to "process internet ballot".

↕ The *ballotCollection* is required to "process internet ballot" and may be modified during this step.

● Before starting "process internet ballot", the resource *voter* must be acquired.

→ To "process internet ballot", retrieve BoD ballot, then validate ballot, and finally duplicate ballot on ballot stock.

Retrieve BoD Ballot

↑ Successful completion of the step "retrieve BoD ballot" should yield the *ballot*.

↕ The *ballotCollection* is required to "retrieve BoD ballot" and may be modified during this step.

→ To "retrieve BoD ballot", login to DemocracyLive and then print ballot.

Login To DemocracyLive

↑ Successful completion of the step "login to DemocracyLive" should yield the *ballot*.

Print Ballot

↕ The *ballotCollection* and *ballot* are required to "print ballot" and may be modified during this step.

Scan Internet Ballots

↓ The *ballotCollection* is required to "scan internet ballots".

↑ Successful completion of the step "scan internet ballots" should yield the *ballotCount*.

→ To "scan internet ballots", run ballots through scanner.

Count Ballots

↓ The *ballotCount* and *ballotCollection* are required to "count ballots".

→ To "count ballots", vote tallies are read off of memory cards from scanners and then results are posted on Yolo County Elections Office website .

Vote Tallies Are Read Off Of Memory Cards From Scanners

↓ The *ballotCollection* and *ballotCount* are required to "vote tallies are read off of memory cards from scanners".

↑ Successful completion of the step "vote tallies are read off of memory cards from scanners" should yield the *voteCount*.

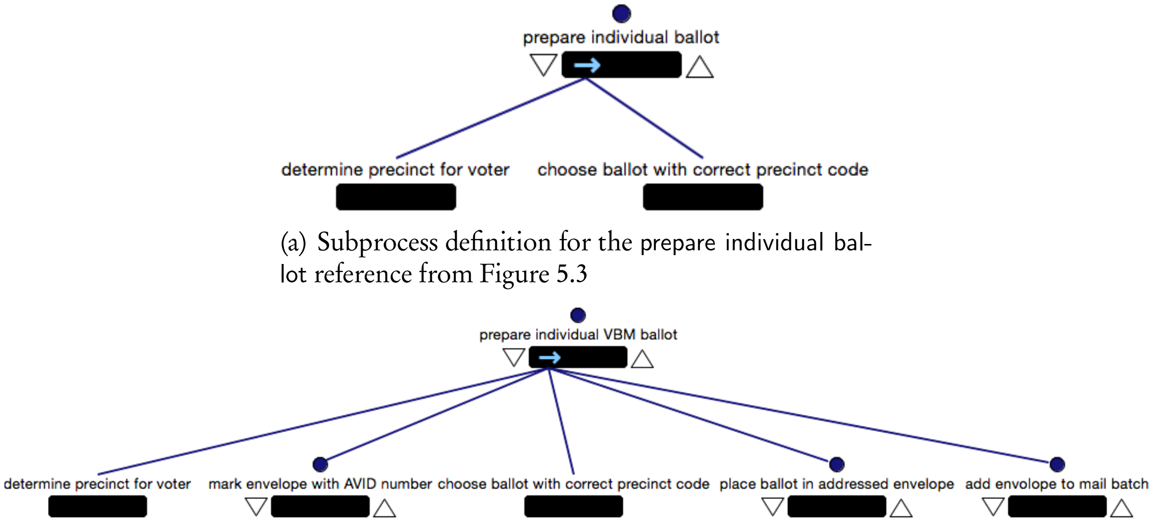
○ The resources *observer:team* and *agent:team* are used in this step.

Results Are Posted On Yolo County Elections Office Website

↓ The *voteCount* is required to "results are posted on Yolo County Elections Office website ".

○ The resources *observer:team* and *agent:team* are used in this step.

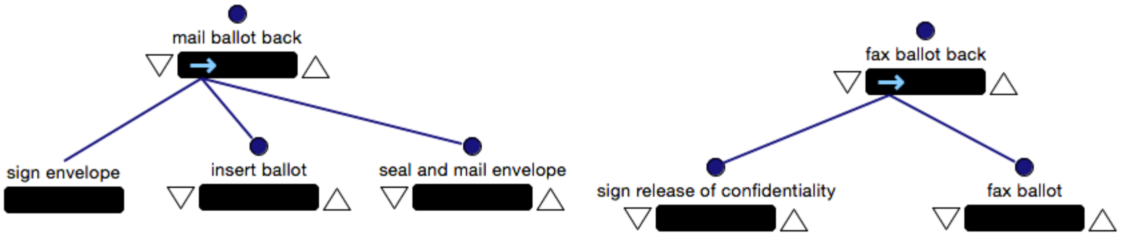
Lastly, coordination diagrams for intermediate steps not presented in Chapter 5 are included below.


prepare individual ballot
determine precinct for voter
choose ballot with correct precinct code
prepare individual VBM ballot
determine precinct for voter
mark envelope with AVID number
choose ballot with correct precinct code
place ballot in addressed envelope
add envelope to mail batch

(a) Subprocess definition for the prepare individual ballot reference from Figure 5.3

(b) Subprocess definition for the prepare individual VBM ballot reference from Figure 5.3

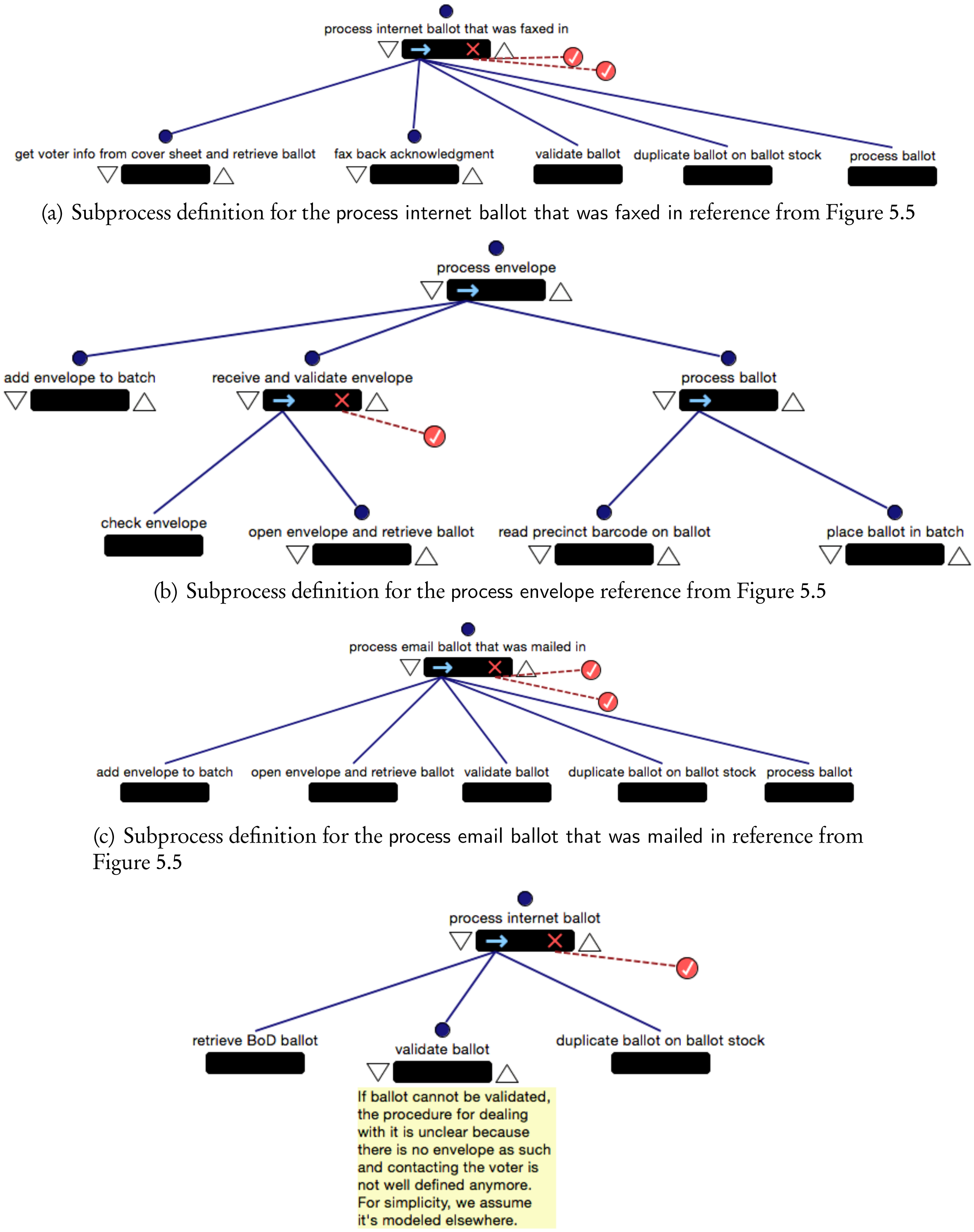
Figure E.1. Complete specifications of the subprocesses for ballot preparation from Figure 5.3 in Chapter 5.



(a) Subprocess definition for the mail ballot back references from Figure 5.4

(b) Subprocess definition for the fax ballot back references from Figure 5.4

Figure E.2. Complete specifications of the subprocesses for ballot transmission from Figure 5.4 in Chapter 5.


process internet ballot that was faxed in
get voter info from cover sheet and retrieve ballot
fax back acknowledgment
validate ballot
duplicate ballot on ballot stock
process ballot
process envelope
add envelope to batch
receive and validate envelope
process ballot
check envelope
open envelope and retrieve ballot
read precinct barcode on ballot
place ballot in batch
process email ballot that was mailed in
add envelope to batch
open envelope and retrieve ballot
validate ballot
duplicate ballot on ballot stock
process ballot
process internet ballot
retrieve BoD ballot
validate ballot
duplicate ballot on ballot stock

If ballot cannot be validated, the procedure for dealing with it is unclear because there is no envelope as such and contacting the voter is not well defined anymore. For simplicity, we assume it's modeled elsewhere.

(a) Subprocess definition for the process internet ballot that was faxed in reference from Figure 5.5

(b) Subprocess definition for the process envelope reference from Figure 5.5

(c) Subprocess definition for the process email ballot that was mailed in reference from Figure 5.5

(d) Subprocess definition for the process internet ballot reference from Figure 5.5

Figure E.3. Complete specifications of the subprocesses for ballot processing from Figure 5.5 in Chapter 5.

APPENDIX F

EVALUATION ARTIFACTS FOR THE VOTE REMOTELY PROCESS FAMILY

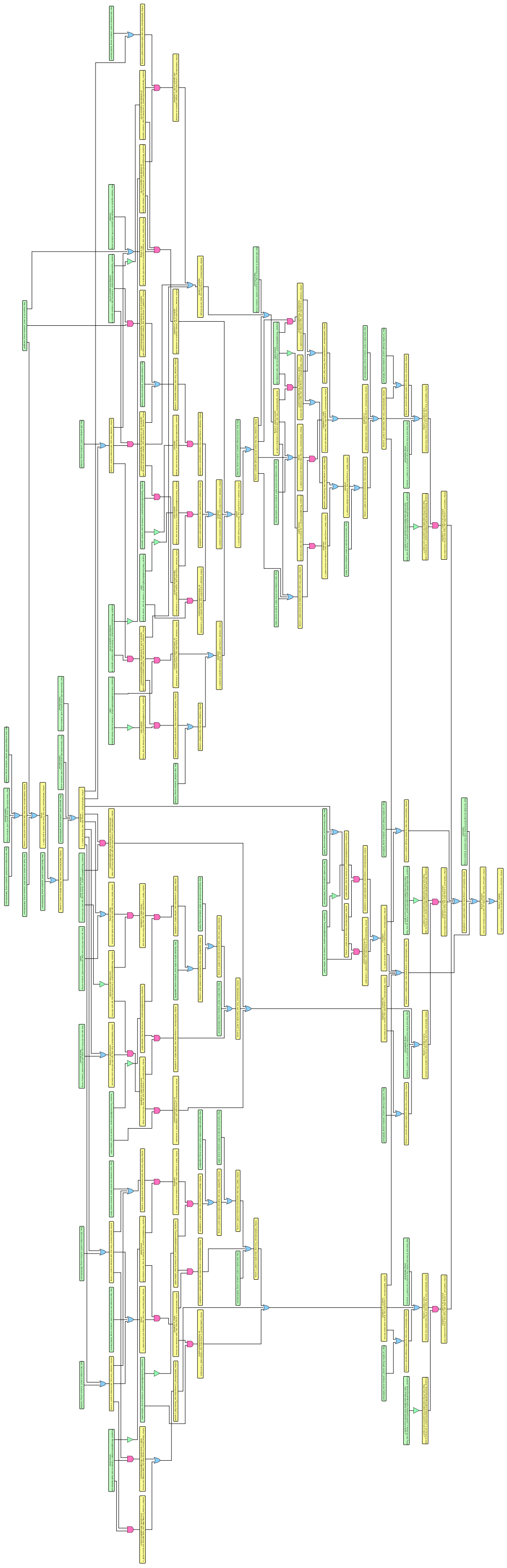
For each set of experiments performed and presented in Chapter 5, we present the resulting artifacts here for completeness.

F.1 Fault Tree Analysis for the NIST Case Study

Fault tree analysis (FTA) was performed on the NIST process family including all abstractions presented above, and then on each process instance describing a single voting modality. Therefore, five sets of results were obtained: one for the vote remotely process family, including all modalities; and one each for vote-by-mail, or VBM, vote-by-fax, vote-by-email, and vote-by-ballot-on-demand. The hazard under consideration is the step results are posted on Yolo County Elections Office website receiving the wrong voteCount artifact (of type VoteCount.java, specified as a JavaBean using the eponymous artifact mode in Visual-JIL) as input. Each set of experiments was performed on a 3GHz dual-core Intel Core i7 processor with 16GB of physical RAM. All experiments were run under an Eclipse virtual machine (VM) with 4,096MB–16,384MB memory allowance. For analysis results, the Little-JIL Analysis Toolset translator was also given a 4,096MB–16,384MB memory allowance.

F.1.1 Fault tree analysis results for the vote remotely process family

The fault tree for the vote remotely process family is included below, followed by the corresponding Minimal Cut Sets (MCSs).



```
MCS 1-1) {
Step “login to DemocracyLive” produces wrong “ballot”
Exception “ProvisionalBallotException” is thrown by step “validate ballot”
}

MCS 1-2) {
Exception “ProvisionalBallotException” is not thrown by step “validate ballot”
Step “login to DemocracyLive” produces wrong “ballot”
}

MCS 2-1) {
Step “print ballot” produces wrong “ballotCollection”
Exception “ProvisionalBallotException” is thrown by step “validate ballot”
}

MCS 2-2) {
Exception “ProvisionalBallotException” is not thrown by step “validate ballot”
Step “print ballot” produces wrong “ballotCollection”
}

MCS 3-1) {
Step “deliver ballots to voters” produces wrong “ballotCollection”
Exception “ProvisionalBallotException” is thrown by step “validate ballot”
}

MCS 3-2) {
Exception “ProvisionalBallotException” is not thrown by step “validate ballot”
```

Step “deliver ballots to voters” produces wrong “ballotCollection”
}

MCS 4-1) {
Step “acquire list of voters” produces wrong “votingRoll”
Exception “ProvisionalBallotException” is thrown by step “validate ballot”
}

MCS 4-2) {
Exception “ProvisionalBallotException” is not thrown by step “validate ballot”
Step “acquire list of voters” produces wrong “votingRoll”
}

MCS 5-1) {
Step “mark envelope with AVID number” produces wrong “envelope”
Exception “ProvisionalBallotException” is thrown by step “validate ballot”
}

MCS 5-2) {
Exception “ProvisionalBallotException” is not thrown by step “validate ballot”
Step “mark envelope with AVID number” produces wrong “envelope”
}

MCS 6-1) {
Step “add envelope to mail batch” produces wrong “ballotCollection”
Exception “ProvisionalBallotException” is thrown by step “validate ballot”
}

MCS 6-2) {

Exception “ProvisionalBallotException” is not thrown by step “validate ballot”

Step “add envelope to mail batch” produces wrong “ballotCollection”

}

MCS 7-1) {

Step “choose ballot with correct precinct code” produces wrong “ballotCollection”

Exception “ProvisionalBallotException” is thrown by step “validate ballot”

}

MCS 7-2) {

Exception “ProvisionalBallotException” is not thrown by step “validate ballot”

Step “choose ballot with correct precinct code” produces wrong “ballotCollection”

}

MCS 8-1) {

Step “hand over ballots to post office” produces wrong “ballotCollection”

Exception “ProvisionalBallotException” is thrown by step “validate ballot”

}

MCS 8-2) {

Exception “ProvisionalBallotException” is not thrown by step “validate ballot”

Step “hand over ballots to post office” produces wrong “ballotCollection”

}

MCS 9-1) {

```
Step “choose ballot with correct precinct code” produces wrong “ballotCollection”
Exception “ProvisionalBallotException” is thrown by step “validate ballot”
}

MCS 9-2) {
Exception “ProvisionalBallotException” is not thrown by step “validate ballot”
Step “choose ballot with correct precinct code” produces wrong “ballotCollection”
}

MCS 10-1) {
Step “generate ballot” produces wrong “ballotCollection”
Exception “ProvisionalBallotException” is thrown by step “validate ballot”
}

MCS 10-2) {
Exception “ProvisionalBallotException” is not thrown by step “validate ballot”
Step “generate ballot” produces wrong “ballotCollection”
}

MCS 11-1) {
Step “choose ballot with correct precinct code” produces wrong “ballotCollection”
Exception “ProvisionalBallotException” is thrown by step “validate ballot”
}

MCS 11-2) {
Exception “ProvisionalBallotException” is not thrown by step “validate ballot”
Step “choose ballot with correct precinct code” produces wrong “ballotCollection”
```

```
}

MCS 12) {
Step “run ballots through scanner” produces wrong “ballotCount”
}

MCS 13) {
Exception “BallotsPlacedInWrongPrecinctBatchException” is not thrown by step “keep track of how many ballots from precinct X are in batch Y”
Step “keep track of how many ballots from precinct X are in batch Y” produces wrong “ballotCount”
}

MCS 14) {
Exception “BallotsPlacedInWrongPrecinctBatchException” is not thrown by step “keep track of how many ballots from precinct X are in batch Y”
Step “run ballots through scanner” produces wrong “ballotCount”
}

MCS 15) {
Exception “BallotsPlacedInWrongPrecinctBatchException” is not thrown by step “keep track of how many ballots from precinct X are in batch Y”
Step “keep track of how many ballots from precinct X are in batch Y” produces wrong “ballotCount”
}

MCS 16) {
```

Exception “BallotsPlacedInWrongPrecinctBatchException” is not thrown by step “keep track of how many ballots from precinct X are in batch Y”

Step “run ballots through scanner” produces wrong “ballotCount”

}

MCS 17) {

Step “run ballots through scanner” produces wrong “ballotCount”

Exception “BallotsPlacedInWrongPrecinctBatchException” is not thrown by step “keep track of how many ballots from precinct X are in batch Y”

}

MCS 18) {

Step “keep track of how many ballots from precinct X are in batch Y” produces wrong “ballotCount”

Exception “BallotsPlacedInWrongPrecinctBatchException” is not thrown by step “keep track of how many ballots from precinct X are in batch Y”

}

MCS 19-1) {

Exception “MissingBallotException” is not thrown by step “open envelope and retrieve ballot”

Step “compare signatures” produces wrong “envelope”

Exception “SignatureMismatchException” is not thrown by step “compare signatures”

}

MCS 19-2) {

Exception “MissingBallotException” is thrown by step “open envelope and retrieve bal-

lot”

Step “compare signatures” produces wrong “envelope”

Exception “SignatureMismatchException” is not thrown by step “compare signatures”

}

MCS 20-1) {

Exception “MissingBallotException” is not thrown by step “open envelope and retrieve ballot”

Step “sign envelope” produces wrong “envelope”

}

MCS 20-2) {

Exception “MissingBallotException” is thrown by step “open envelope and retrieve ballot”

Step “sign envelope” produces wrong “envelope”

}

MCS 21-1) {

Exception “MissingBallotException” is not thrown by step “open envelope and retrieve ballot”

Exception “MissingSignatureException” is not thrown by step “compare signature on envelope with signature on file”

Exception “SignatureMismatchException” is not thrown by step “compare signature on envelope with signature on file”

Step “add envelope to batch” produces wrong “envelope”

}

MCS 21-2) {

Exception “MissingBallotException” is thrown by step “open envelope and retrieve ballot”

Exception “MissingSignatureException” is not thrown by step “compare signature on envelope with signature on file”

Exception “SignatureMismatchException” is not thrown by step “compare signature on envelope with signature on file”

Step “add envelope to batch” produces wrong “envelope”

}

MCS 22-1) {

Exception “MissingBallotException” is not thrown by step “open envelope and retrieve ballot”

Exception “MissingSignatureException” is not thrown by step “compare signature on envelope with signature on file”

Exception “SignatureMismatchException” is not thrown by step “compare signature on envelope with signature on file”

Step “compare signature on envelope with signature on file” produces wrong “envelope”

}

MCS 22-2) {

Exception “MissingBallotException” is thrown by step “open envelope and retrieve ballot”

Exception “MissingSignatureException” is not thrown by step “compare signature on envelope with signature on file”

Exception “SignatureMismatchException” is not thrown by step “compare signature on envelope with signature on file”

Step “compare signature on envelope with signature on file” produces wrong “envelope”

}

MCS 23-1) {

Exception “MissingBallotException” is not thrown by step “open envelope and retrieve ballot”

Step “place red line through signature” produces wrong “envelope”

}

MCS 23-2) {

Exception “MissingBallotException” is thrown by step “open envelope and retrieve ballot”

Step “place red line through signature” produces wrong “envelope”

}

MCS 24-1) {

Exception “MissingBallotException” is not thrown by step “open envelope and retrieve ballot”

Exception “MissingSignatureException” is thrown by step “compare signature on envelope with signature on file”

Step “add envelope to batch” produces wrong “envelope”

}

MCS 24-2) {

Exception “MissingBallotException” is thrown by step “open envelope and retrieve ballot”

Exception “MissingSignatureException” is thrown by step “compare signature on envelope with signature on file"

Step "add envelope to batch" produces wrong "envelope"

}

MCS 25-1) {

Exception "MissingBallotException" is not thrown by step "open envelope and retrieve ballot"

Exception "MissingSignatureException" is not thrown by step "compare signature on envelope with signature on file"

Step "add envelope to batch" produces wrong "ballotCollection"

Exception "SignatureMismatchException" is not thrown by step "compare signature on envelope with signature on file"

}

MCS 25-2) {

Exception "MissingBallotException" is thrown by step "open envelope and retrieve ballot"

Exception "MissingSignatureException" is not thrown by step "compare signature on envelope with signature on file"

Step "add envelope to batch" produces wrong "ballotCollection"

Exception "SignatureMismatchException" is not thrown by step "compare signature on envelope with signature on file"

}

MCS 26-1) {

Exception "MissingBallotException" is not thrown by step "open envelope and retrieve ballot"

Exception “SignatureMismatchException” is thrown by step “compare signature on envelope with signature on file”

Step “add envelope to batch” produces wrong “envelope”

}

MCS 26-2) {

Exception “MissingBallotException” is thrown by step “open envelope and retrieve ballot”

Exception “SignatureMismatchException” is thrown by step “compare signature on envelope with signature on file”

Step “add envelope to batch” produces wrong “envelope”

}

MCS 27-1) {

Exception “MissingBallotException” is not thrown by step “open envelope and retrieve ballot”

Step “add envelope to repository in oder of precinct number” produces wrong “ballotCollection”

}

MCS 27-2) {

Exception “MissingBallotException” is thrown by step “open envelope and retrieve ballot”

Step “add envelope to repository in oder of precinct number” produces wrong “ballotCollection”

}

MCS 28) {

Exception “MissingBallotException” is not thrown by step “open envelope and retrieve ballot”

Step “open envelope and retrieve ballot” produces wrong “ballotCollection”

}

MCS 29) {

Step “place ballot in batch” produces wrong “ballotCollection”

}

MCS 30) {

Exception “MissingBallotException” is not thrown by step “open envelope and retrieve ballot”

Step “open envelope and retrieve ballot” produces wrong “ballot”

}

MCS 31) {

Step “read precinct barcode on ballot” produces wrong “ballot”

}

MCS 32-1) {

Step “get voter info from cover sheet and retrieve ballot” produces wrong “ballotCollection”

Exception “MissingBallotException” is not thrown by step “get voter info from cover sheet and retrieve ballot”

Exception “ProvisionalBallotException” is thrown by step “validate ballot”

}

MCS 32-2) {

Step “get voter info from cover sheet and retrieve ballot” produces wrong “ballotCollection”

Exception “MissingBallotException” is not thrown by step “get voter info from cover sheet and retrieve ballot”

Exception “ProvisionalBallotException” is not thrown by step “validate ballot”

}

MCS 33) {

Step “place ballot in batch” produces wrong “ballotCollection”

}

MCS 34) {

Step “read precinct barcode on ballot” produces wrong “ballot”

}

MCS 35) {

Step “duplicate ballot on ballot stock” produces wrong “paperBallot”

}

MCS 36) {

Step “get voter info from cover sheet and retrieve ballot” produces wrong “ballot”

Exception “MissingBallotException” is not thrown by step “get voter info from cover sheet and retrieve ballot”

Exception “ProvisionalBallotException” is not thrown by step “validate ballot”

}

MCS 37-1) {

Exception “ProvisionalBallotException” is thrown by step “validate ballot”

Step “open envelope and retrieve ballot” produces wrong “ballotCollection”

Exception “MissingBallotException” is not thrown by step “open envelope and retrieve ballot”

}

MCS 37-2) {

Step “open envelope and retrieve ballot” produces wrong “ballotCollection”

Exception “MissingBallotException” is not thrown by step “open envelope and retrieve ballot”

Exception “ProvisionalBallotException” is not thrown by step “validate ballot”

}

MCS 38-1) {

Exception “ProvisionalBallotException” is thrown by step “validate ballot”

Step “add envelope to batch” produces wrong “ballotCollection”

Exception “MissingBallotException” is not thrown by step “open envelope and retrieve ballot”

}

MCS 38-2) {

Exception “MissingBallotException” is not thrown by step “open envelope and retrieve ballot”

Step “add envelope to batch” produces wrong “ballotCollection”

Exception “ProvisionalBallotException” is not thrown by step “validate ballot”

}

MCS 38-3) {

Exception “MissingBallotException” is thrown by step “open envelope and retrieve ballot”

Step “add envelope to batch” produces wrong “ballotCollection”

}

MCS 39-1) {

Exception “ProvisionalBallotException” is thrown by step “validate ballot”

Step “add envelope to batch” produces wrong “envelope”

Exception “MissingBallotException” is not thrown by step “open envelope and retrieve ballot”

}

MCS 39-2) {

Exception “MissingBallotException” is not thrown by step “open envelope and retrieve ballot”

Step “add envelope to batch” produces wrong “envelope”

Exception “ProvisionalBallotException” is not thrown by step “validate ballot”

}

MCS 39-3) {

Step “add envelope to batch” produces wrong “envelope”

Exception “MissingBallotException” is thrown by step “open envelope and retrieve ballot”

}

MCS 40) {

Step “read precinct barcode on ballot” produces wrong “ballot”

}

MCS 41) {

Step “duplicate ballot on ballot stock” produces wrong “paperBallot”

}

MCS 42) {

Exception “MissingBallotException” is not thrown by step “open envelope and retrieve ballot”

Step “open envelope and retrieve ballot” produces wrong “ballot”

Exception “ProvisionalBallotException” is not thrown by step “validate ballot”

}

MCS 43) {

Step “place ballot in batch” produces wrong “ballotCollection”

}

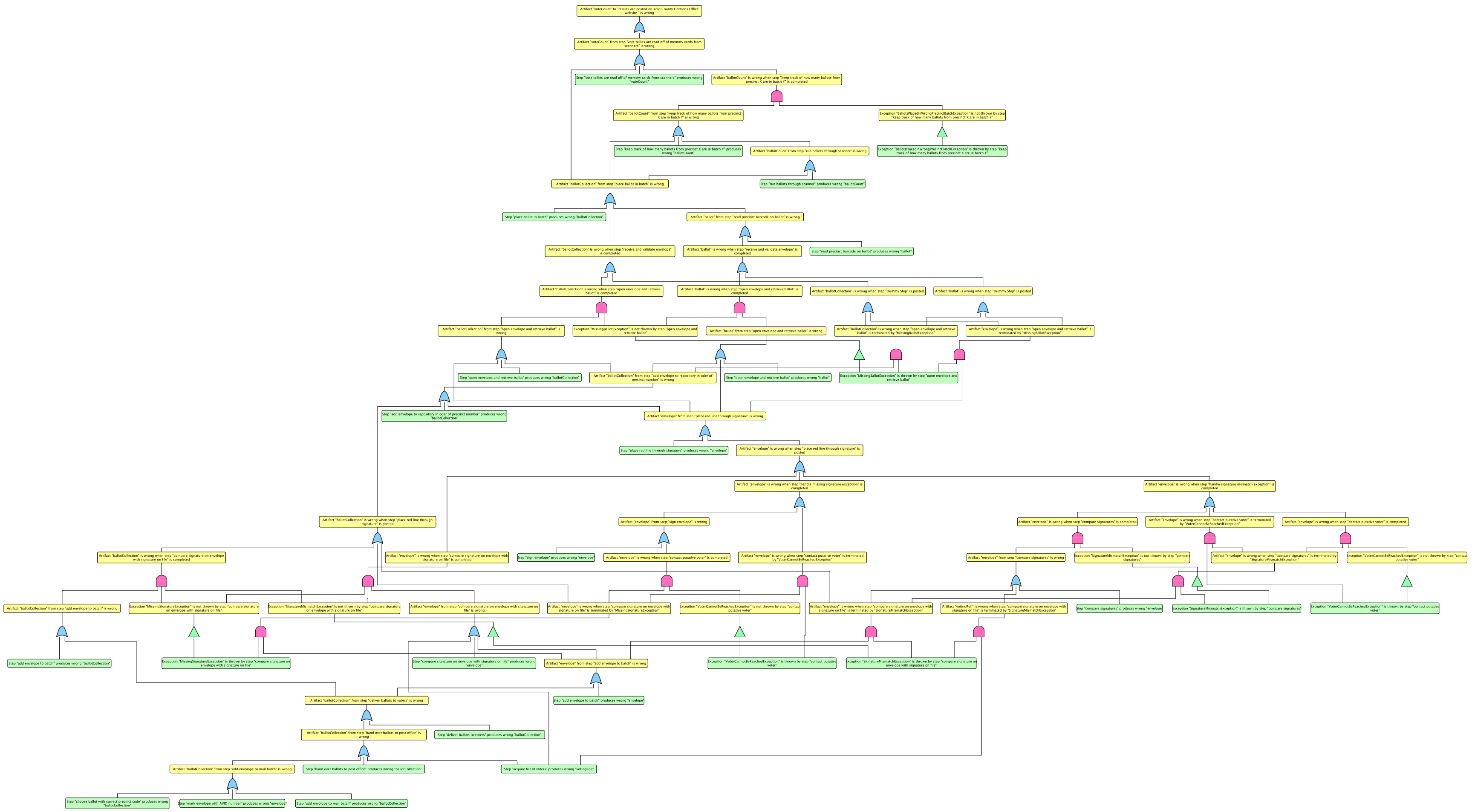
MCS 44) {

Step “vote tallies are read off of memory cards from scanners” produces wrong “vote-Count”

}

F.1.2 Fault tree analysis results for the vote-by-mail process

The fault tree for the vote-by-mail process is included below, followed by the corresponding MCSs.


Artifact "voteCount" to "results are posted on Yolo County Elections Office website " is wrong
Artifact "voteCount" from step "vote tallies are read off of memory cards from scanners" is wrong
Step "vote tallies are read off of memory cards from scanners" produces wrong "voteCount"
Artifact "ballotCount" is wrong when step "keep track of how many ballots from precinct X are in batch Y" is completed
Artifact "ballotCount" from step "keep track of how many ballots from precinct X are in batch Y" is wrong
Exception "BallotsPlacedInWrongPrecinctBatchException" is not thrown by step "keep track of how many ballots from precinct X are in batch Y"
Step "keep track of how many ballots from precinct X are in batch Y" produces wrong "ballotCount"
Artifact "ballotCount" from step "run ballots through scanner" is wrong
Exception "BallotsPlacedInWrongPrecinctBatchException" is thrown by step "keep track of how many ballots from precinct X are in batch Y"
Artifact "ballotCollection" from step "place ballot in batch" is wrong
Step "run ballots through scanner" produces wrong "ballotCount"
Step "place ballot in batch" produces wrong "ballotCollection"
Artifact "ballot" from step "read precinct barcode on ballot" is wrong
Artifact "ballotCollection" is wrong when step "receive and validate envelope" is completed
Artifact "ballot" is wrong when step "receive and validate envelope" is completed
Step "read precinct barcode on ballot" produces wrong "ballot"
Artifact "ballotCollection" is wrong when step "open envelope and retrieve ballot" is completed
Artifact "ballot" is wrong when step "open envelope and retrieve ballot" is completed
Artifact "ballotCollection" is wrong when step "Dummy Step" is posted
Artifact "ballot" is wrong when step "Dummy Step" is posted
Artifact "ballotCollection" from step "open envelope and retrieve ballot" is wrong
Exception "MissingBallotException" is not thrown by step "open envelope and retrieve ballot"
Artifact "ballot" from step "open envelope and retrieve ballot" is wrong
Artifact "ballotCollection" is wrong when step "open envelope and retrieve ballot" is terminated by "MissingBallotException"
Artifact "envelope" is wrong when step "open envelope and retrieve ballot" is terminated by "MissingBallotException"
Step "open envelope and retrieve ballot" produces wrong "ballotCollection"
Artifact "ballotCollection" from step "add envelope to repository in oder of precinct number" is wrong
Step "open envelope and retrieve ballot" produces wrong "ballot"
Exception "MissingBallotException" is thrown by step "open envelope and retrieve ballot"
Step "add envelope to repository in oder of precinct number" produces wrong "ballotCollection"
Artifact "envelope" from step "place red line through signature" is wrong
Step "place red line through signature" produces wrong "envelope"
Artifact "envelope" is wrong when step "place red line through signature" is posted
Artifact "envelope" is wrong when step "handle missing signature exception" is completed
Artifact "envelope" is wrong when step "handle signature mismatch exception" is completed
Artifact "ballotCollection" is wrong when step "place red line through signature" is posted
Artifact "envelope" from step "sign envelope" is wrong
Artifact "envelope" is wrong when step "compare signatures" is completed
Artifact "envelope" is wrong when step "contact putative voter" is terminated by "VoterCannotBeReachedException"
Artifact "envelope" is wrong when step "contact putative voter" is completed
Artifact "ballotCollection" is wrong when step "compare signature on envelope with signature on file" is completed
Artifact "envelope" is wrong when step "compare signature on envelope with signature on file" is completed
Step "sign envelope" produces wrong "envelope"
Artifact "envelope" is wrong when step "contact putative voter" is completed
Artifact "envelope" is wrong when step "contact putative voter" is terminated by "VoterCannotBeReachedException"
Artifact "envelope" from step "compare signatures" is wrong
Exception "SignatureMismatchException" is not thrown by step "compare signatures"
Artifact "envelope" is wrong when step "compare signatures" is terminated by "SignatureMismatchException"
Exception "VoterCannotBeReachedException" is not thrown by step "contact putative voter"
Artifact "ballotCollection" from step "add envelope to batch" is wrong
Exception "MissingSignatureException" is not thrown by step "compare signature on envelope with signature on file"
Exception "SignatureMismatchException" is not thrown by step "compare signature on envelope with signature on file"
Artifact "envelope" from step "compare signature on envelope with signature on file" is wrong
Artifact "envelope" is wrong when step "compare signature on envelope with signature on file" is terminated by "MissingSignatureException"
Exception "VoterCannotBeReachedException" is not thrown by step "contact putative voter"
Artifact "envelope" is wrong when step "compare signature on envelope with signature on file" is terminated by "SignatureMismatchException"
Artifact "votingRoll" is wrong when step "compare signature on envelope with signature on file" is terminated by "SignatureMismatchException"
Step "compare signatures" produces wrong "envelope"
Exception "SignatureMismatchException" is thrown by step "compare signatures"
Exception "VoterCannotBeReachedException" is thrown by step "contact putative voter"
Step "add envelope to batch" produces wrong "ballotCollection"
Exception "MissingSignatureException" is thrown by step "compare signature on envelope with signature on file"
Step "compare signature on envelope with signature on file" produces wrong "envelope"
Artifact "envelope" from step "add envelope to batch" is wrong
Exception "VoterCannotBeReachedException" is thrown by step "contact putative voter"
Exception "SignatureMismatchException" is thrown by step "compare signature on envelope with signature on file"
Artifact "ballotCollection" from step "deliver ballots to voters" is wrong
Step "add envelope to batch" produces wrong "envelope"
Artifact "ballotCollection" from step "hand over ballots to post office" is wrong
Step "deliver ballots to voters" produces wrong "ballotCollection"
Artifact "ballotCollection" from step "add envelope to mail batch" is wrong
Step "hand over ballots to post office" produces wrong "ballotCollection"
Step "acquire list of voters" produces wrong "votingRoll"
Step "choose ballot with correct precinct code" produces wrong "ballotCollection"
Step "mark envelope with AVID number" produces wrong "envelope"
Step "add envelope to mail batch" produces wrong "ballotCollection"

MCS 1) {

Step “run ballots through scanner” produces wrong “ballotCount”

Exception “BallotsPlacedInWrongPrecinctBatchException” is not thrown by step “keep track of how many ballots from precinct X are in batch Y”

}

MCS 2) {

Step “keep track of how many ballots from precinct X are in batch Y” produces wrong “ballotCount”

Exception “BallotsPlacedInWrongPrecinctBatchException” is not thrown by step “keep track of how many ballots from precinct X are in batch Y”

}

MCS 3) {

Step “place ballot in batch” produces wrong “ballotCollection”

}

MCS 4) {

Step “read precinct barcode on ballot” produces wrong “ballot”

}

MCS 5-1) {

Exception “MissingBallotException” is thrown by step “open envelope and retrieve ballot”

Step “sign envelope” produces wrong “envelope”

}

MCS 5-2) {

Step “sign envelope” produces wrong “envelope”

Exception “MissingBallotException” is not thrown by step “open envelope and retrieve ballot”

}

MCS 6-1) {

Exception “MissingBallotException” is thrown by step “open envelope and retrieve ballot”

Exception “MissingSignatureException” is not thrown by step “compare signature on envelope with signature on file”

Exception “SignatureMismatchException” is not thrown by step “compare signature on envelope with signature on file”

Step “acquire list of voters” produces wrong “votingRoll”

}

MCS 6-2) {

Exception “MissingSignatureException” is not thrown by step “compare signature on envelope with signature on file”

Exception “SignatureMismatchException” is not thrown by step “compare signature on envelope with signature on file”

Step “acquire list of voters” produces wrong “votingRoll”

Exception “MissingBallotException” is not thrown by step “open envelope and retrieve ballot”

}

MCS 7-1) {

Exception “MissingBallotException” is thrown by step “open envelope and retrieve ballot”

Exception “MissingSignatureException” is not thrown by step “compare signature on envelope with signature on file”

Exception “SignatureMismatchException” is not thrown by step “compare signature on envelope with signature on file”

Step “compare signature on envelope with signature on file” produces wrong “envelope”

}

MCS 7-2) {

Exception “MissingSignatureException” is not thrown by step “compare signature on envelope with signature on file”

Exception “SignatureMismatchException” is not thrown by step “compare signature on envelope with signature on file”

Step “compare signature on envelope with signature on file” produces wrong “envelope”

Exception “MissingBallotException” is not thrown by step “open envelope and retrieve ballot”

}

MCS 8-1) {

Exception “MissingBallotException” is thrown by step “open envelope and retrieve ballot”

Exception “MissingSignatureException” is not thrown by step “compare signature on envelope with signature on file”

Exception “SignatureMismatchException” is not thrown by step “compare signature on envelope with signature on file”

Step “add envelope to batch” produces wrong “envelope”

}

MCS 8-2) {

Exception “MissingSignatureException” is not thrown by step “compare signature on envelope with signature on file”

Exception “SignatureMismatchException” is not thrown by step “compare signature on envelope with signature on file”

Step “add envelope to batch” produces wrong “envelope”

Exception “MissingBallotException” is not thrown by step “open envelope and retrieve ballot”

}

MCS 9-1) {

Exception “MissingBallotException” is thrown by step “open envelope and retrieve ballot”

Exception “MissingSignatureException” is not thrown by step “compare signature on envelope with signature on file”

Exception “SignatureMismatchException” is not thrown by step “compare signature on envelope with signature on file”

Step “deliver ballots to voters” produces wrong “ballotCollection”

}

MCS 9-2) {

Exception “MissingSignatureException” is not thrown by step “compare signature on envelope with signature on file”

Exception “SignatureMismatchException” is not thrown by step “compare signature on envelope with signature on file”

Step “deliver ballots to voters” produces wrong “ballotCollection”

Exception “MissingBallotException” is not thrown by step “open envelope and retrieve ballot”

}

MCS 10-1) {

Exception “MissingBallotException” is thrown by step “open envelope and retrieve ballot”

Exception “MissingSignatureException” is not thrown by step “compare signature on envelope with signature on file”

Exception “SignatureMismatchException” is not thrown by step “compare signature on envelope with signature on file”

Step “hand over ballots to post office” produces wrong “ballotCollection”

}

MCS 10-2) {

Exception “MissingSignatureException” is not thrown by step “compare signature on envelope with signature on file”

Exception “SignatureMismatchException” is not thrown by step “compare signature on envelope with signature on file”

Step “hand over ballots to post office” produces wrong “ballotCollection”

Exception “MissingBallotException” is not thrown by step “open envelope and retrieve ballot”

}

MCS 11-1) {

Exception “MissingBallotException” is thrown by step “open envelope and retrieve bal-

lot"

Exception "MissingSignatureException" is not thrown by step "compare signature on envelope with signature on file"

Exception "SignatureMismatchException" is not thrown by step "compare signature on envelope with signature on file"

Step "add envelope to mail batch" produces wrong "ballotCollection"

}

MCS 11-2) {

Exception "MissingSignatureException" is not thrown by step "compare signature on envelope with signature on file"

Exception "SignatureMismatchException" is not thrown by step "compare signature on envelope with signature on file"

Step "add envelope to mail batch" produces wrong "ballotCollection"

Exception "MissingBallotException" is not thrown by step "open envelope and retrieve ballot"

}

MCS 12-1) {

Exception "MissingBallotException" is thrown by step "open envelope and retrieve ballot"

Exception "MissingSignatureException" is not thrown by step "compare signature on envelope with signature on file"

Exception "SignatureMismatchException" is not thrown by step "compare signature on envelope with signature on file"

Step "choose ballot with correct precinct code" produces wrong "ballotCollection"

}

MCS 12-2) {

Exception “MissingSignatureException” is not thrown by step “compare signature on envelope with signature on file”

Exception “SignatureMismatchException” is not thrown by step “compare signature on envelope with signature on file”

Step “choose ballot with correct precinct code” produces wrong “ballotCollection”

Exception “MissingBallotException” is not thrown by step “open envelope and retrieve ballot”

}

MCS 13-1) {

Exception “MissingBallotException” is thrown by step “open envelope and retrieve ballot”

Exception “MissingSignatureException” is not thrown by step “compare signature on envelope with signature on file”

Exception “SignatureMismatchException” is not thrown by step “compare signature on envelope with signature on file”

Step “mark envelope with AVID number” produces wrong “envelope”

}

MCS 13-2) {

Exception “MissingSignatureException” is not thrown by step “compare signature on envelope with signature on file”

Exception “SignatureMismatchException” is not thrown by step “compare signature on envelope with signature on file”

Step “mark envelope with AVID number” produces wrong “envelope”

Exception “MissingBallotException” is not thrown by step “open envelope and retrieve ballot”

}

MCS 14-1) {

Exception “MissingBallotException” is thrown by step “open envelope and retrieve ballot”

Exception “SignatureMismatchException” is not thrown by step “compare signatures”

Step “compare signatures” produces wrong “envelope”

}

MCS 14-2) {

Exception “SignatureMismatchException” is not thrown by step “compare signatures”

Step “compare signatures” produces wrong “envelope”

Exception “MissingBallotException” is not thrown by step “open envelope and retrieve ballot”

}

MCS 15-1) {

Exception “MissingBallotException” is thrown by step “open envelope and retrieve ballot”

Step “place red line through signature” produces wrong “envelope”

}

MCS 15-2) {

Step “place red line through signature” produces wrong “envelope”

Exception “MissingBallotException” is not thrown by step “open envelope and retrieve ballot"

}

MCS 16-1) {

Exception "MissingBallotException" is thrown by step "open envelope and retrieve ballot"

Step "add envelope to repository in oder of precinct number" produces wrong "ballotCollection"

}

MCS 16-2) {

Step "add envelope to repository in oder of precinct number" produces wrong "ballotCollection"

Exception "MissingBallotException" is not thrown by step "open envelope and retrieve ballot"

}

MCS 17-1) {

Exception "MissingBallotException" is thrown by step "open envelope and retrieve ballot"

Exception "MissingSignatureException" is not thrown by step "compare signature on envelope with signature on file"

Exception "SignatureMismatchException" is not thrown by step "compare signature on envelope with signature on file"

Step "add envelope to batch" produces wrong "ballotCollection"

}

MCS 17-2) {

Exception “MissingSignatureException” is not thrown by step “compare signature on envelope with signature on file”

Exception “SignatureMismatchException” is not thrown by step “compare signature on envelope with signature on file”

Step “add envelope to batch” produces wrong “ballotCollection”

Exception “MissingBallotException” is not thrown by step “open envelope and retrieve ballot”

}

MCS 18-1) {

Exception “MissingBallotException” is thrown by step “open envelope and retrieve ballot”

Exception “MissingSignatureException” is thrown by step “compare signature on envelope with signature on file”

Step “add envelope to batch” produces wrong “envelope”

}

MCS 18-2) {

Exception “MissingSignatureException” is thrown by step “compare signature on envelope with signature on file”

Step “add envelope to batch” produces wrong “envelope”

Exception “MissingBallotException” is not thrown by step “open envelope and retrieve ballot”

}

MCS 19-1) {

Exception “MissingBallotException” is thrown by step “open envelope and retrieve ballot”

Exception “MissingSignatureException” is thrown by step “compare signature on envelope with signature on file”

Step “deliver ballots to voters” produces wrong “ballotCollection”

}

MCS 19-2) {

Exception “MissingSignatureException” is thrown by step “compare signature on envelope with signature on file”

Step “deliver ballots to voters” produces wrong “ballotCollection”

Exception “MissingBallotException” is not thrown by step “open envelope and retrieve ballot”

}

MCS 20-1) {

Exception “MissingBallotException” is thrown by step “open envelope and retrieve ballot”

Exception “MissingSignatureException” is thrown by step “compare signature on envelope with signature on file”

Step “hand over ballots to post office” produces wrong “ballotCollection”

}

MCS 20-2) {

Exception “MissingSignatureException” is thrown by step “compare signature on envelope with signature on file”

Step “hand over ballots to post office” produces wrong “ballotCollection”

Exception “MissingBallotException” is not thrown by step “open envelope and retrieve ballot”

}

MCS 21-1) {

Exception “MissingBallotException” is thrown by step “open envelope and retrieve ballot”

Exception “MissingSignatureException” is thrown by step “compare signature on envelope with signature on file”

Step “acquire list of voters” produces wrong “votingRoll”

}

MCS 21-2) {

Exception “MissingSignatureException” is thrown by step “compare signature on envelope with signature on file”

Step “acquire list of voters” produces wrong “votingRoll”

Exception “MissingBallotException” is not thrown by step “open envelope and retrieve ballot”

}

MCS 22-1) {

Exception “MissingBallotException” is thrown by step “open envelope and retrieve ballot”

Exception “MissingSignatureException” is thrown by step “compare signature on envelope with signature on file”

Step “add envelope to mail batch” produces wrong “ballotCollection”

}

MCS 22-2) {

Exception “MissingSignatureException” is thrown by step “compare signature on envelope with signature on file”

Step “add envelope to mail batch” produces wrong “ballotCollection”

Exception “MissingBallotException” is not thrown by step “open envelope and retrieve ballot”

}

MCS 23-1) {

Exception “MissingBallotException” is thrown by step “open envelope and retrieve ballot”

Exception “MissingSignatureException” is thrown by step “compare signature on envelope with signature on file”

Step “choose ballot with correct precinct code” produces wrong “ballotCollection”

}

MCS 23-2) {

Exception “MissingSignatureException” is thrown by step “compare signature on envelope with signature on file”

Step “choose ballot with correct precinct code” produces wrong “ballotCollection”

Exception “MissingBallotException” is not thrown by step “open envelope and retrieve ballot”

}

MCS 24-1) {

Exception “MissingBallotException” is thrown by step “open envelope and retrieve bal-

lot"

Exception "MissingSignatureException" is thrown by step "compare signature on envelope with signature on file"

Step "mark envelope with AVID number" produces wrong "envelope"

}

MCS 24-2) {

Exception "MissingSignatureException" is thrown by step "compare signature on envelope with signature on file"

Step "mark envelope with AVID number" produces wrong "envelope"

Exception "MissingBallotException" is not thrown by step "open envelope and retrieve ballot"

}

MCS 25-1) {

Exception "MissingBallotException" is thrown by step "open envelope and retrieve ballot"

Step "add envelope to batch" produces wrong "envelope"

Exception "SignatureMismatchException" is thrown by step "compare signature on envelope with signature on file"

}

MCS 25-2) {

Step "add envelope to batch" produces wrong "envelope"

Exception "SignatureMismatchException" is thrown by step "compare signature on envelope with signature on file"

Exception "MissingBallotException" is not thrown by step "open envelope and retrieve ballot”

}

MCS 26-1) {

Exception “MissingBallotException” is thrown by step “open envelope and retrieve ballot”

Step “deliver ballots to voters” produces wrong “ballotCollection”

Exception “SignatureMismatchException” is thrown by step “compare signature on envelope with signature on file”

}

MCS 26-2) {

Step “deliver ballots to voters” produces wrong “ballotCollection”

Exception “SignatureMismatchException” is thrown by step “compare signature on envelope with signature on file”

Exception “MissingBallotException” is not thrown by step “open envelope and retrieve ballot”

}

MCS 27-1) {

Exception “MissingBallotException” is thrown by step “open envelope and retrieve ballot”

Step “hand over ballots to post office” produces wrong “ballotCollection”

Exception “SignatureMismatchException” is thrown by step “compare signature on envelope with signature on file”

}

MCS 27-2) {

Step “hand over ballots to post office” produces wrong “ballotCollection”

Exception “SignatureMismatchException” is thrown by step “compare signature on envelope with signature on file”

Exception “MissingBallotException” is not thrown by step “open envelope and retrieve ballot”

}

MCS 28-1) {

Exception “MissingBallotException” is thrown by step “open envelope and retrieve ballot”

Step “acquire list of voters” produces wrong “votingRoll”

Exception “SignatureMismatchException” is thrown by step “compare signature on envelope with signature on file”

}

MCS 28-2) {

Step “acquire list of voters” produces wrong “votingRoll”

Exception “SignatureMismatchException” is thrown by step “compare signature on envelope with signature on file”

Exception “MissingBallotException” is not thrown by step “open envelope and retrieve ballot”

}

MCS 29-1) {

Exception “MissingBallotException” is thrown by step “open envelope and retrieve ballot”

Step “add envelope to mail batch” produces wrong “ballotCollection”

Exception “SignatureMismatchException” is thrown by step “compare signature on envelope with signature on file”

}

MCS 29-2) {

Step “add envelope to mail batch” produces wrong “ballotCollection”

Exception “SignatureMismatchException” is thrown by step “compare signature on envelope with signature on file”

Exception “MissingBallotException” is not thrown by step “open envelope and retrieve ballot”

}

MCS 30-1) {

Exception “MissingBallotException” is thrown by step “open envelope and retrieve ballot”

Step “choose ballot with correct precinct code” produces wrong “ballotCollection”

Exception “SignatureMismatchException” is thrown by step “compare signature on envelope with signature on file”

}

MCS 30-2) {

Step “choose ballot with correct precinct code” produces wrong “ballotCollection”

Exception “SignatureMismatchException” is thrown by step “compare signature on envelope with signature on file”

Exception “MissingBallotException” is not thrown by step “open envelope and retrieve ballot”

}

MCS 31-1) {

Exception “MissingBallotException” is thrown by step “open envelope and retrieve ballot”

Step “mark envelope with AVID number” produces wrong “envelope”

Exception “SignatureMismatchException” is thrown by step “compare signature on envelope with signature on file”

}

MCS 31-2) {

Step “mark envelope with AVID number” produces wrong “envelope”

Exception “SignatureMismatchException” is thrown by step “compare signature on envelope with signature on file”

Exception “MissingBallotException” is not thrown by step “open envelope and retrieve ballot”

}

MCS 32) {

Step “open envelope and retrieve ballot” produces wrong “ballot”

Exception “MissingBallotException” is not thrown by step “open envelope and retrieve ballot”

}

MCS 33) {

Step “open envelope and retrieve ballot” produces wrong “ballotCollection”

Exception “MissingBallotException” is not thrown by step “open envelope and retrieve ballot"

}

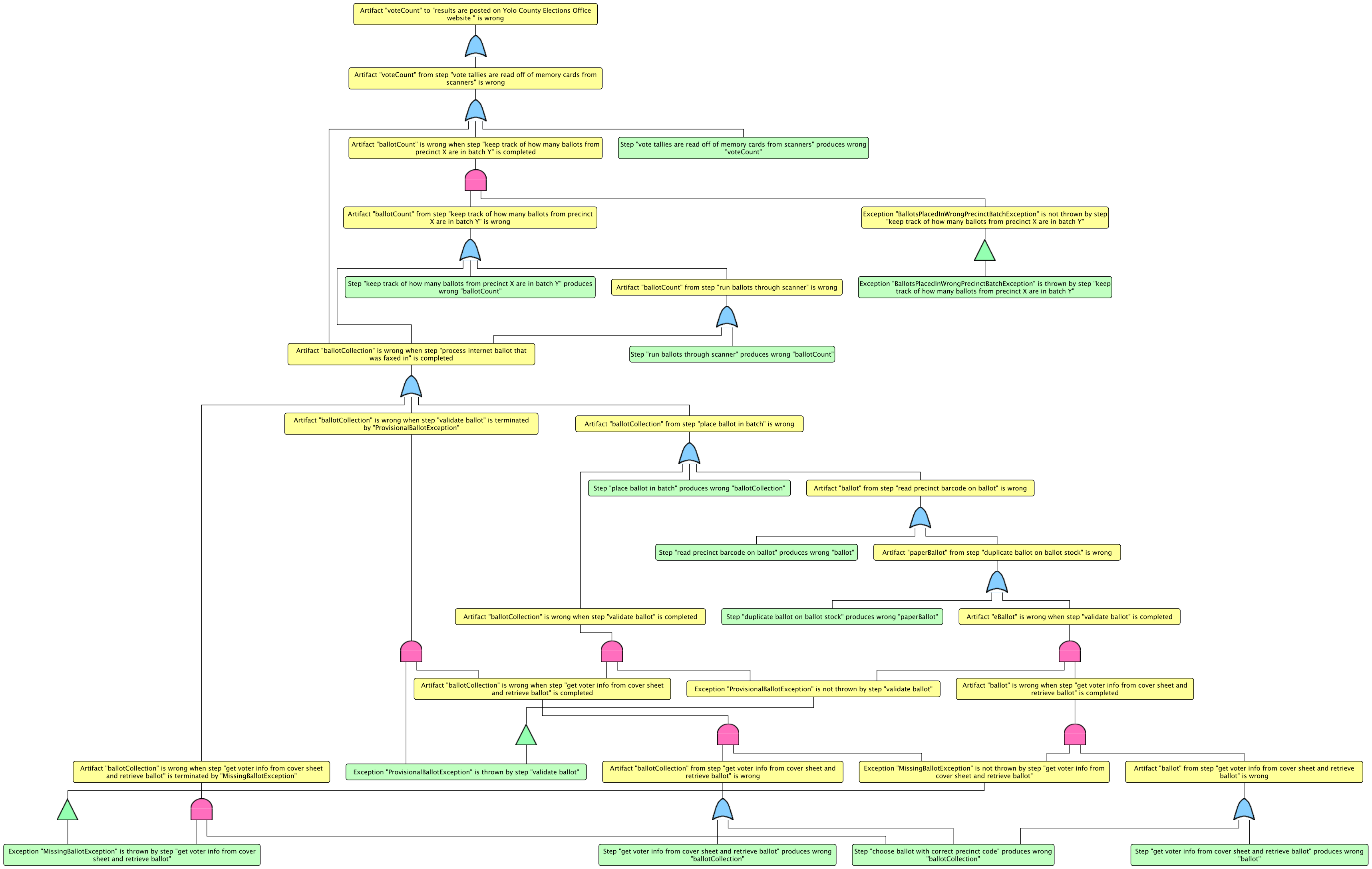
MCS 34) {

Step "vote tallies are read off of memory cards from scanners" produces wrong "vote-Count"

}

F.1.3 Fault tree analysis results for the vote-by-fax process

The fault tree for the vote-by-fax is included below, followed by the corresponding MCSs.


Artifact "voteCount" to "results are posted on Yolo County Elections Office website " is wrong
Artifact "voteCount" from step "vote tallies are read off of memory cards from scanners" is wrong
Artifact "ballotCount" is wrong when step "keep track of how many ballots from precinct X are in batch Y" is completed
Step "vote tallies are read off of memory cards from scanners" produces wrong "voteCount"
Artifact "ballotCount" from step "keep track of how many ballots from precinct X are in batch Y" is wrong
Exception "BallotsPlacedInWrongPrecinctBatchException" is not thrown by step "keep track of how many ballots from precinct X are in batch Y"
Step "keep track of how many ballots from precinct X are in batch Y" produces wrong "ballotCount"
Artifact "ballotCount" from step "run ballots through scanner" is wrong
Exception "BallotsPlacedInWrongPrecinctBatchException" is thrown by step "keep track of how many ballots from precinct X are in batch Y"
Artifact "ballotCollection" is wrong when step "process internet ballot that was faxed in" is completed
Step "run ballots through scanner" produces wrong "ballotCount"
Artifact "ballotCollection" is wrong when step "validate ballot" is terminated by "ProvisionalBallotException"
Artifact "ballotCollection" from step "place ballot in batch" is wrong
Step "place ballot in batch" produces wrong "ballotCollection"
Artifact "ballot" from step "read precinct barcode on ballot" is wrong
Step "read precinct barcode on ballot" produces wrong "ballot"
Artifact "paperBallot" from step "duplicate ballot on ballot stock" is wrong
Artifact "ballotCollection" is wrong when step "validate ballot" is completed
Step "duplicate ballot on ballot stock" produces wrong "paperBallot"
Artifact "eBallot" is wrong when step "validate ballot" is completed
Artifact "ballotCollection" is wrong when step "get voter info from cover sheet and retrieve ballot" is completed
Exception "ProvisionalBallotException" is not thrown by step "validate ballot"
Artifact "ballot" is wrong when step "get voter info from cover sheet and retrieve ballot" is completed
Artifact "ballotCollection" is wrong when step "get voter info from cover sheet and retrieve ballot" is terminated by "MissingBallotException"
Exception "ProvisionalBallotException" is thrown by step "validate ballot"
Artifact "ballotCollection" from step "get voter info from cover sheet and retrieve ballot" is wrong
Exception "MissingBallotException" is not thrown by step "get voter info from cover sheet and retrieve ballot"
Artifact "ballot" from step "get voter info from cover sheet and retrieve ballot" is wrong
Exception "MissingBallotException" is thrown by step "get voter info from cover sheet and retrieve ballot"
Step "get voter info from cover sheet and retrieve ballot" produces wrong "ballotCollection"
Step "choose ballot with correct precinct code" produces wrong "ballotCollection"
Step "get voter info from cover sheet and retrieve ballot" produces wrong "ballot"

MCS 1-1) {

Exception “MissingBallotException” is thrown by step “get voter info from cover sheet and retrieve ballot”

Step “choose ballot with correct precinct code” produces wrong “ballotCollection”

}

MCS 1-2) {

Exception “ProvisionalBallotException” is thrown by step “validate ballot”

Exception “MissingBallotException” is not thrown by step “get voter info from cover sheet and retrieve ballot”

Step “choose ballot with correct precinct code” produces wrong “ballotCollection”

}

MCS 1-3) {

Step “choose ballot with correct precinct code” produces wrong “ballotCollection”

Exception “MissingBallotException” is not thrown by step “get voter info from cover sheet and retrieve ballot”

Exception “ProvisionalBallotException” is not thrown by step “validate ballot”

}

MCS 2-1) {

Exception “ProvisionalBallotException” is thrown by step “validate ballot”

Exception “MissingBallotException” is not thrown by step “get voter info from cover sheet and retrieve ballot”

Step “get voter info from cover sheet and retrieve ballot” produces wrong “ballotCollection”

}

MCS 2-2) {

Exception “MissingBallotException” is not thrown by step “get voter info from cover sheet and retrieve ballot”

Step “get voter info from cover sheet and retrieve ballot” produces wrong “ballotCollection”

Exception “ProvisionalBallotException” is not thrown by step “validate ballot”

}

MCS 3) {

Step “read precinct barcode on ballot” produces wrong “ballot”

}

MCS 4) {

Step “duplicate ballot on ballot stock” produces wrong “paperBallot”

}

MCS 5) {

Step “get voter info from cover sheet and retrieve ballot” produces wrong “ballot”

Exception “MissingBallotException” is not thrown by step “get voter info from cover sheet and retrieve ballot”

Exception “ProvisionalBallotException” is not thrown by step “validate ballot”

}

MCS 6) {

Step “place ballot in batch” produces wrong “ballotCollection”

}

MCS 7) {

Step "vote tallies are read off of memory cards from scanners" produces wrong "voteCount"

}

MCS 8) {

Step "run ballots through scanner" produces wrong "ballotCount"

Exception "BallotsPlacedInWrongPrecinctBatchException" is not thrown by step "keep track of how many ballots from precinct X are in batch Y"

}

MCS 9) {

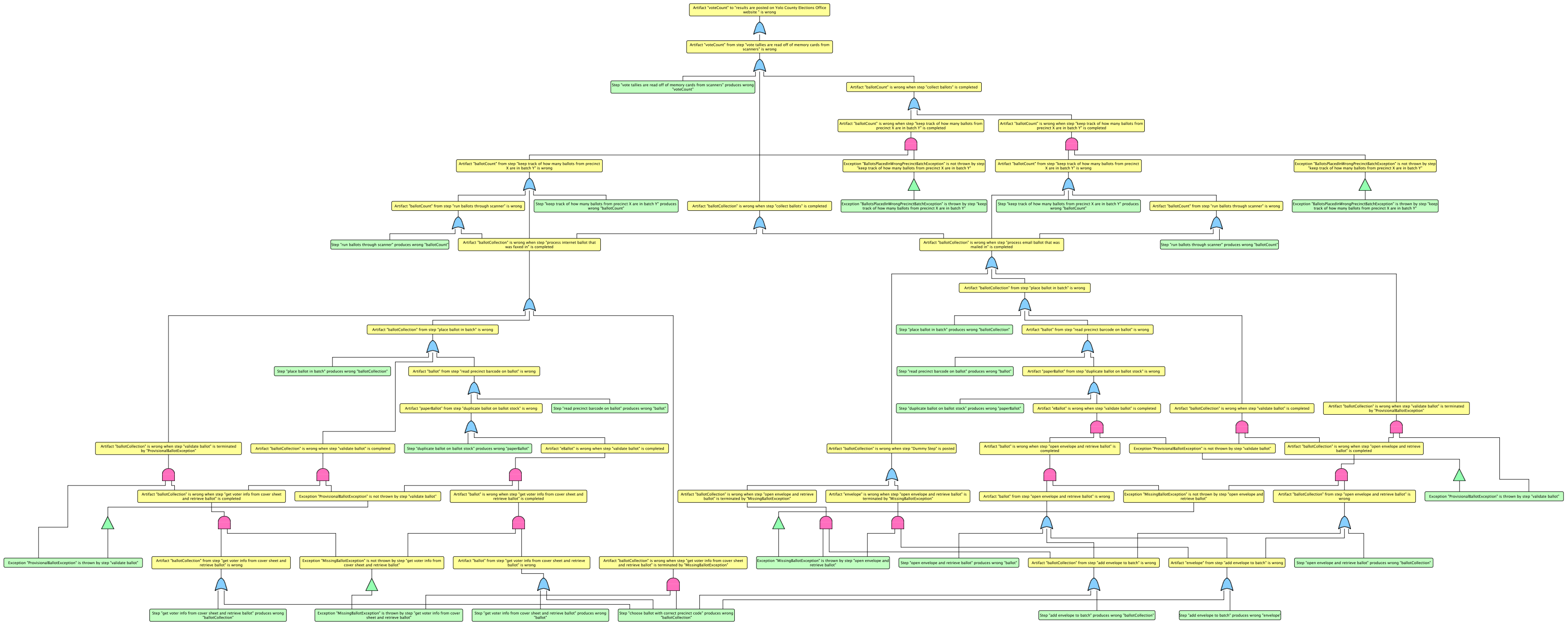
Step "keep track of how many ballots from precinct X are in batch Y" produces wrong "ballotCount"

Exception "BallotsPlacedInWrongPrecinctBatchException" is not thrown by step "keep track of how many ballots from precinct X are in batch Y"

}

F.1.4 Fault tree analysis results for the vote-by-email process

The fault tree for the vote-by-email is included below, followed by the corresponding MCSs.


Artifact "voteCount" to "results are posted on Yolo County Elections Office website " is wrong
Artifact "voteCount" from step "vote tallies are read off of memory cards from scanners" is wrong
Step "vote tallies are read off of memory cards from scanners" produces wrong "voteCount"
Artifact "ballotCount" is wrong when step "collect ballots" is completed
Artifact "ballotCount" is wrong when step "keep track of how many ballots from precinct X are in batch Y" is completed
Artifact "ballotCount" is wrong when step "keep track of how many ballots from precinct X are in batch Y" is completed
Artifact "ballotCount" from step "keep track of how many ballots from precinct X are in batch Y" is wrong
Exception "BallotsPlacedInWrongPrecinctBatchException" is not thrown by step "keep track of how many ballots from precinct X are in batch Y"
Artifact "ballotCount" from step "keep track of how many ballots from precinct X are in batch Y" is wrong
Exception "BallotsPlacedInWrongPrecinctBatchException" is not thrown by step "keep track of how many ballots from precinct X are in batch Y"
Artifact "ballotCount" from step "run ballots through scanner" is wrong
Step "keep track of how many ballots from precinct X are in batch Y" produces wrong "ballotCount"
Artifact "ballotCollection" is wrong when step "collect ballots" is completed
Exception "BallotsPlacedInWrongPrecinctBatchException" is thrown by step "keep track of how many ballots from precinct X are in batch Y"
Step "keep track of how many ballots from precinct X are in batch Y" produces wrong "ballotCount"
Artifact "ballotCount" from step "run ballots through scanner" is wrong
Exception "BallotsPlacedInWrongPrecinctBatchException" is thrown by step "keep track of how many ballots from precinct X are in batch Y"
Step "run ballots through scanner" produces wrong "ballotCount"
Artifact "ballotCollection" is wrong when step "process internet ballot that was faxed in" is completed
Artifact "ballotCollection" is wrong when step "process email ballot that was mailed in" is completed
Step "run ballots through scanner" produces wrong "ballotCount"
Artifact "ballotCollection" from step "place ballot in batch" is wrong
Artifact "ballotCollection" from step "place ballot in batch" is wrong
Step "place ballot in batch" produces wrong "ballotCollection"
Artifact "ballot" from step "read precinct barcode on ballot" is wrong
Step "place ballot in batch" produces wrong "ballotCollection"
Artifact "ballot" from step "read precinct barcode on ballot" is wrong
Step "read precinct barcode on ballot" produces wrong "ballot"
Artifact "paperBallot" from step "duplicate ballot on ballot stock" is wrong
Artifact "paperBallot" from step "duplicate ballot on ballot stock" is wrong
Step "read precinct barcode on ballot" produces wrong "ballot"
Step "duplicate ballot on ballot stock" produces wrong "paperBallot"
Artifact "eBallot" is wrong when step "validate ballot" is completed
Artifact "ballotCollection" is wrong when step "validate ballot" is completed
Artifact "ballotCollection" is wrong when step "validate ballot" is terminated by "ProvisionalBallotException"
Artifact "ballotCollection" is wrong when step "validate ballot" is terminated by "ProvisionalBallotException"
Artifact "ballotCollection" is wrong when step "validate ballot" is completed
Step "duplicate ballot on ballot stock" produces wrong "paperBallot"
Artifact "eBallot" is wrong when step "validate ballot" is completed
Artifact "ballotCollection" is wrong when step "Dummy Step" is posted
Artifact "ballot" is wrong when step "open envelope and retrieve ballot" is completed
Exception "ProvisionalBallotException" is not thrown by step "validate ballot"
Artifact "ballotCollection" is wrong when step "open envelope and retrieve ballot" is completed
Artifact "ballotCollection" is wrong when step "get voter info from cover sheet and retrieve ballot" is completed
Exception "ProvisionalBallotException" is not thrown by step "validate ballot"
Artifact "ballot" is wrong when step "get voter info from cover sheet and retrieve ballot" is completed
Artifact "ballotCollection" is wrong when step "open envelope and retrieve ballot" is terminated by "MissingBallotException"
Artifact "envelope" is wrong when step "open envelope and retrieve ballot" is terminated by "MissingBallotException"
Artifact "ballot" from step "open envelope and retrieve ballot" is wrong
Exception "MissingBallotException" is not thrown by step "open envelope and retrieve ballot"
Artifact "ballotCollection" from step "open envelope and retrieve ballot" is wrong
Exception "ProvisionalBallotException" is thrown by step "validate ballot"
Exception "ProvisionalBallotException" is thrown by step "validate ballot"
Artifact "ballotCollection" from step "get voter info from cover sheet and retrieve ballot" is wrong
Exception "MissingBallotException" is not thrown by step "get voter info from cover sheet and retrieve ballot"
Artifact "ballot" from step "get voter info from cover sheet and retrieve ballot" is wrong
Artifact "ballotCollection" is wrong when step "get voter info from cover sheet and retrieve ballot" is terminated by "MissingBallotException"
Exception "MissingBallotException" is thrown by step "open envelope and retrieve ballot"
Step "open envelope and retrieve ballot" produces wrong "ballot"
Artifact "ballotCollection" from step "add envelope to batch" is wrong
Artifact "envelope" from step "add envelope to batch" is wrong
Step "open envelope and retrieve ballot" produces wrong "ballotCollection"
Step "get voter info from cover sheet and retrieve ballot" produces wrong "ballotCollection"
Exception "MissingBallotException" is thrown by step "get voter info from cover sheet and retrieve ballot"
Step "get voter info from cover sheet and retrieve ballot" produces wrong "ballot"
Step "choose ballot with correct precinct code" produces wrong "ballotCollection"
Step "add envelope to batch" produces wrong "ballotCollection"
Step "add envelope to batch" produces wrong "envelope"

MCS 1) {

Exception “BallotsPlacedInWrongPrecinctBatchException” is not thrown by step “keep track of how many ballots from precinct X are in batch Y”

Step “run ballots through scanner” produces wrong “ballotCount”

}

MCS 2) {

Exception “BallotsPlacedInWrongPrecinctBatchException” is not thrown by step “keep track of how many ballots from precinct X are in batch Y”

Step “keep track of how many ballots from precinct X are in batch Y” produces wrong “ballotCount”

}

MCS 3) {

Step “run ballots through scanner” produces wrong “ballotCount”

Exception “BallotsPlacedInWrongPrecinctBatchException” is not thrown by step “keep track of how many ballots from precinct X are in batch Y”

}

MCS 4) {

Step “keep track of how many ballots from precinct X are in batch Y” produces wrong “ballotCount”

Exception “BallotsPlacedInWrongPrecinctBatchException” is not thrown by step “keep track of how many ballots from precinct X are in batch Y”

}

MCS 5) {

Step “vote tallies are read off of memory cards from scanners” produces wrong “voteCount”

}

MCS 6-1) {

Exception “MissingBallotException” is thrown by step “open envelope and retrieve ballot”

Step “choose ballot with correct precinct code” produces wrong “ballotCollection”

}

MCS 6-2) {

Exception “ProvisionalBallotException” is not thrown by step “validate ballot”

Exception “MissingBallotException” is not thrown by step “open envelope and retrieve ballot”

Step “choose ballot with correct precinct code” produces wrong “ballotCollection”

}

MCS 6-3) {

Exception “ProvisionalBallotException” is thrown by step “validate ballot”

Exception “MissingBallotException” is not thrown by step “open envelope and retrieve ballot”

Step “choose ballot with correct precinct code” produces wrong “ballotCollection”

}

MCS 7-1) {

Exception “MissingBallotException” is thrown by step “open envelope and retrieve ballot”

Step “add envelope to batch” produces wrong “ballotCollection”

}

MCS 7-2) {

Exception “ProvisionalBallotException” is not thrown by step “validate ballot”

Exception “MissingBallotException” is not thrown by step “open envelope and retrieve ballot”

Step “add envelope to batch” produces wrong “ballotCollection”

}

MCS 7-3) {

Exception “ProvisionalBallotException” is thrown by step “validate ballot”

Exception “MissingBallotException” is not thrown by step “open envelope and retrieve ballot”

Step “add envelope to batch” produces wrong “ballotCollection”

}

MCS 8-1) {

Exception “MissingBallotException” is thrown by step “open envelope and retrieve ballot”

Step “add envelope to batch” produces wrong “envelope”

}

MCS 8-2) {

Exception “ProvisionalBallotException” is not thrown by step “validate ballot”

Exception “MissingBallotException” is not thrown by step “open envelope and retrieve ballot”

Step “add envelope to batch” produces wrong “envelope”

}

MCS 8-3) {

Exception “ProvisionalBallotException” is thrown by step “validate ballot”

Exception “MissingBallotException” is not thrown by step “open envelope and retrieve ballot”

Step “add envelope to batch” produces wrong “envelope”

}

MCS 9) {

Step “duplicate ballot on ballot stock” produces wrong “paperBallot”

}

MCS 10) {

Exception “ProvisionalBallotException” is not thrown by step “validate ballot”

Exception “MissingBallotException” is not thrown by step “open envelope and retrieve ballot”

Step “open envelope and retrieve ballot” produces wrong “ballot”

}

MCS 11) {

Step “read precinct barcode on ballot” produces wrong “ballot”

}

MCS 12-1) {

Exception “ProvisionalBallotException” is not thrown by step “validate ballot”

Exception “MissingBallotException” is not thrown by step “open envelope and retrieve ballot”

Step “open envelope and retrieve ballot” produces wrong “ballotCollection”

}

MCS 12-2) {

Exception “ProvisionalBallotException” is thrown by step “validate ballot”

Exception “MissingBallotException” is not thrown by step “open envelope and retrieve ballot”

Step “open envelope and retrieve ballot” produces wrong “ballotCollection”

}

MCS 13) {

Step “place ballot in batch” produces wrong “ballotCollection”

}

MCS 14-1) {

Exception “MissingBallotException” is not thrown by step “get voter info from cover sheet and retrieve ballot”

Step “get voter info from cover sheet and retrieve ballot” produces wrong “ballotCollection”

Exception “ProvisionalBallotException” is thrown by step “validate ballot”

}

MCS 14-2) {

Exception “MissingBallotException” is not thrown by step “get voter info from cover sheet and retrieve ballot”

Step “get voter info from cover sheet and retrieve ballot” produces wrong “ballotCollection”

Exception “ProvisionalBallotException” is not thrown by step “validate ballot”

}

MCS 15) {

Step “place ballot in batch” produces wrong “ballotCollection”

}

MCS 16) {

Step “read precinct barcode on ballot” produces wrong “ballot”

}

MCS 17) {

Exception “ProvisionalBallotException” is not thrown by step “validate ballot”

Exception “MissingBallotException” is not thrown by step “get voter info from cover sheet and retrieve ballot”

Step “get voter info from cover sheet and retrieve ballot” produces wrong “ballot”

}

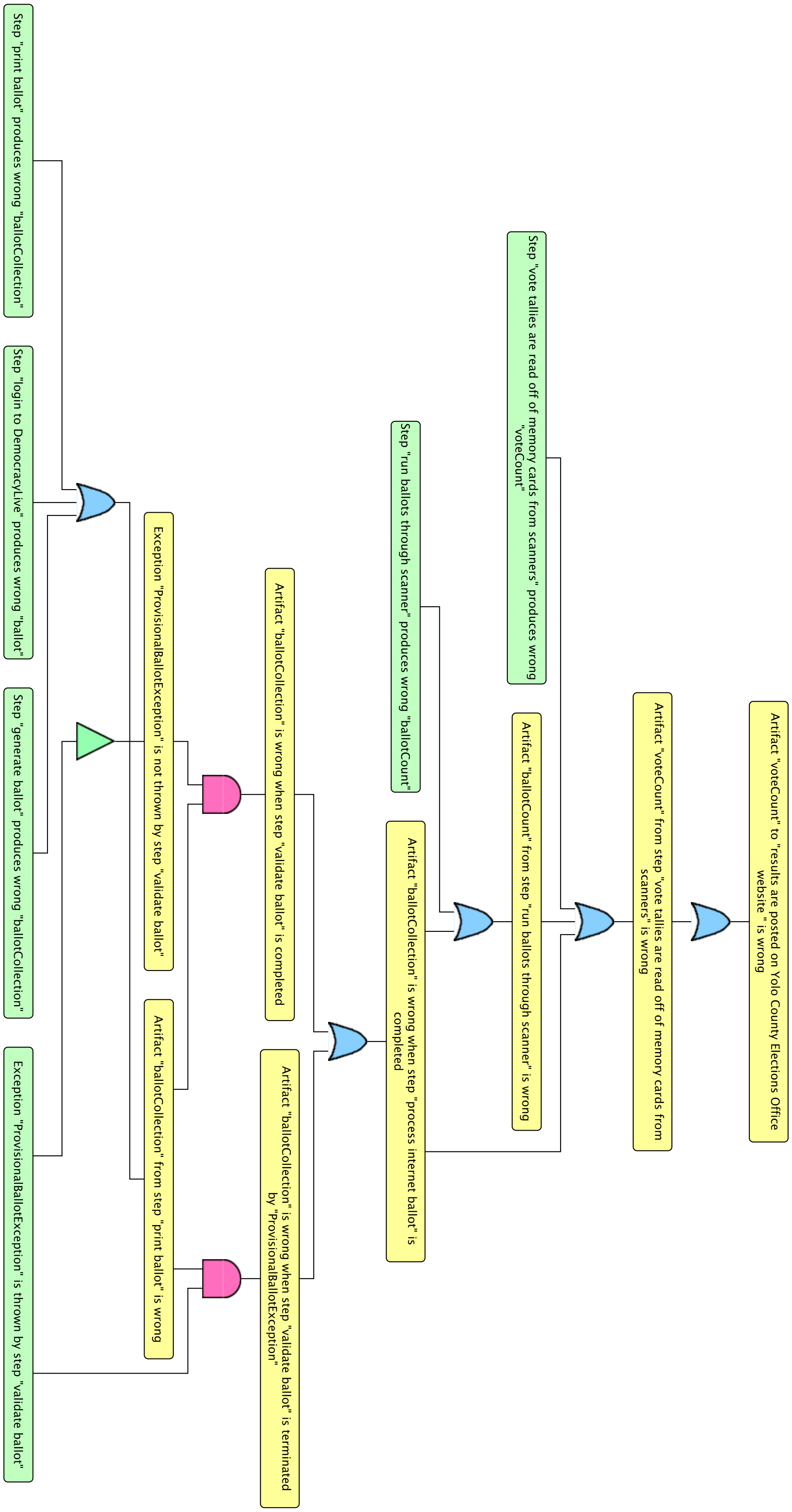
MCS 18) {

Step “duplicate ballot on ballot stock” produces wrong “paperBallot”

}

F.1.5 Fault tree analysis results for the vote-by-ballot-on-demand process

The fault tree for the vote-by-ballot-on-demand is included below, followed by the corresponding MCSs.


Artifact "voteCount" to "results are posted on Yolo County Elections Office website " is wrong
Artifact "voteCount" from step "vote tallies are read off of memory cards from scanners" is wrong
Step "vote tallies are read off of memory cards from scanners" produces wrong "voteCount"
Artifact "ballotCount" from step "run ballots through scanner" is wrong
Step "run ballots through scanner" produces wrong "ballotCount"
Artifact "ballotCollection" is wrong when step "process internet ballot" is completed
Artifact "ballotCollection" is wrong when step "validate ballot" is completed
Artifact "ballotCollection" is wrong when step "validate ballot" is terminated by "ProvisionalBallotException"
Exception "ProvisionalBallotException" is not thrown by step "validate ballot"
Artifact "ballotCollection" from step "print ballot" is wrong
Step "print ballot" produces wrong "ballotCollection"
Step "login to DemocracyLive" produces wrong "ballot"
Step "generate ballot" produces wrong "ballotCollection"
Exception "ProvisionalBallotException" is thrown by step "validate ballot"

MCS 1-1) {

Step “generate ballot” produces wrong “ballotCollection”

Exception “ProvisionalBallotException” is thrown by step “validate ballot”

}

MCS 1-2) {

Step “generate ballot” produces wrong “ballotCollection”

Exception “ProvisionalBallotException” is not thrown by step “validate ballot”

}

MCS 2-1) {

Step “print ballot” produces wrong “ballotCollection”

Exception “ProvisionalBallotException” is thrown by step “validate ballot”

}

MCS 2-2) {

Step “print ballot” produces wrong “ballotCollection”

Exception “ProvisionalBallotException” is not thrown by step “validate ballot”

}

MCS 3-1) {

Step “login to DemocracyLive” produces wrong “ballot”

Exception “ProvisionalBallotException” is thrown by step “validate ballot”

}

MCS 3-2) {

Step “login to DemocracyLive” produces wrong “ballot”

Exception “ProvisionalBallotException” is not thrown by step “validate ballot”

}

MCS 4) {

Step “vote tallies are read off of memory cards from scanners” produces wrong “voteCount”

}

MCS 5) {

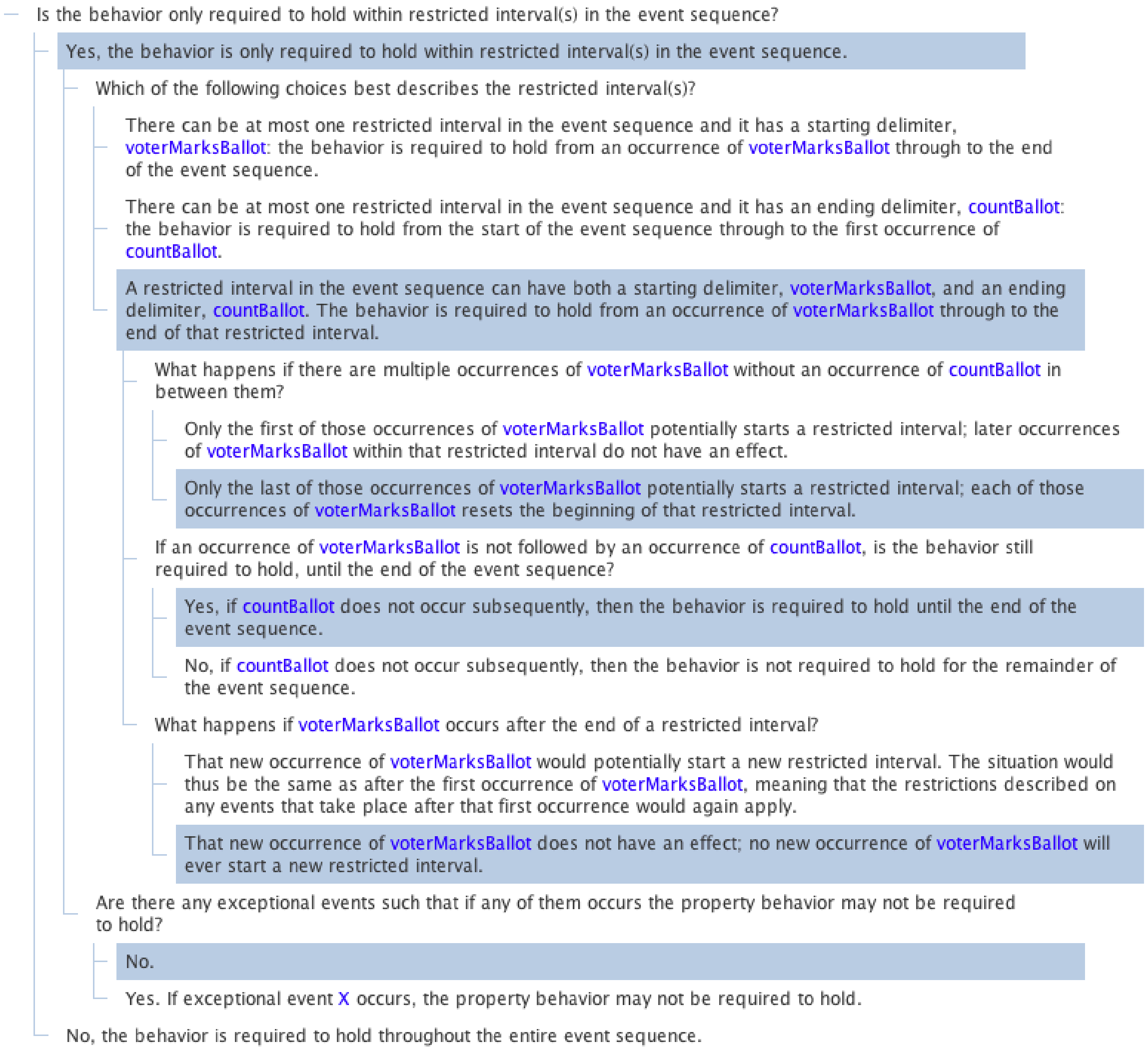
Step “run ballots through scanner” produces wrong “ballotCount”

}

F.2 Finite-State Verification Results for NIST Case Study

All the artifacts from the NIST case study were verified against a single property, encoding that after a voter votes, an insider (e.g., corrupt poll worker) should not be able to mark that ballot until it has been counted. This is the property used for the extended SPLC case study as well and it is based on the property originally presented in [67]. The full property specification is presented in Chapter 5, and we present the question trees used to aid in the scope and behavior definitions in Figures F.1 and F.2, respectively, for completeness.

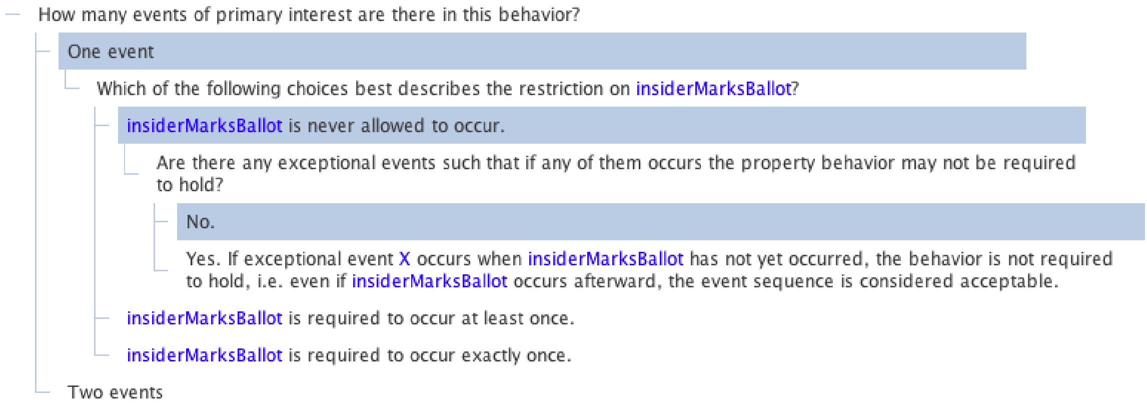
Seven sets of results were obtained, including the vote remotely process family consisting of all four variants, each of the four variants as a process instance, and two additional families resulting from removing all fax-based variants from the ALL family, and removing all fax-based and all BoD-based variants from the ALL family, as explained in Chapter 5. Each artifact is presented below.



- Is the behavior only required to hold within restricted interval(s) in the event sequence?
  - Yes, the behavior is only required to hold within restricted interval(s) in the event sequence.
    - Which of the following choices best describes the restricted interval(s)?
      - There can be at most one restricted interval in the event sequence and it has a starting delimiter, voterMarksBallot: the behavior is required to hold from an occurrence of voterMarksBallot through to the end of the event sequence.
      - There can be at most one restricted interval in the event sequence and it has an ending delimiter, countBallot: the behavior is required to hold from the start of the event sequence through to the first occurrence of countBallot.
      - A restricted interval in the event sequence can have both a starting delimiter, voterMarksBallot, and an ending delimiter, countBallot. The behavior is required to hold from an occurrence of voterMarksBallot through to the end of that restricted interval.
        - What happens if there are multiple occurrences of voterMarksBallot without an occurrence of countBallot in between them?
          - Only the first of those occurrences of voterMarksBallot potentially starts a restricted interval; later occurrences of voterMarksBallot within that restricted interval do not have an effect.
          - Only the last of those occurrences of voterMarksBallot potentially starts a restricted interval; each of those occurrences of voterMarksBallot resets the beginning of that restricted interval.
        - If an occurrence of voterMarksBallot is not followed by an occurrence of countBallot, is the behavior still required to hold, until the end of the event sequence?
          - Yes, if countBallot does not occur subsequently, then the behavior is required to hold until the end of the event sequence.
          - No, if countBallot does not occur subsequently, then the behavior is not required to hold for the remainder of the event sequence.
        - What happens if voterMarksBallot occurs after the end of a restricted interval?
          - That new occurrence of voterMarksBallot would potentially start a new restricted interval. The situation would thus be the same as after the first occurrence of voterMarksBallot, meaning that the restrictions described on any events that take place after that first occurrence would again apply.
          - That new occurrence of voterMarksBallot does not have an effect; no new occurrence of voterMarksBallot will ever start a new restricted interval.
    - Are there any exceptional events such that if any of them occurs the property behavior may not be required to hold?
      - No.
      - Yes. If exceptional event X occurs, the property behavior may not be required to hold.
  - No, the behavior is required to hold throughout the entire event sequence.

Figure F.1. Scope question tree for the property "After a *voter marks ballot* event, no *insider marks ballot* event can occur until *count ballots* occurs."

The vote-by-mail process satisfied the property.



- How many events of primary interest are there in this behavior?
  - One event
    - Which of the following choices best describes the restriction on insiderMarksBallot?
      - insiderMarksBallot is never allowed to occur.
        - Are there any exceptional events such that if any of them occurs the property behavior may not be required to hold?
          - No.
          - Yes. If exceptional event X occurs when insiderMarksBallot has not yet occurred, the behavior is not required to hold, i.e. even if insiderMarksBallot occurs afterward, the event sequence is considered acceptable.
      - insiderMarksBallot is required to occur at least once.
      - insiderMarksBallot is required to occur exactly once.
  - Two events

Figure F.2. Behavior question tree for the property "After a *voter marks ballot* event, no *insider marks ballot* event can occur until *count ballots* occurs."

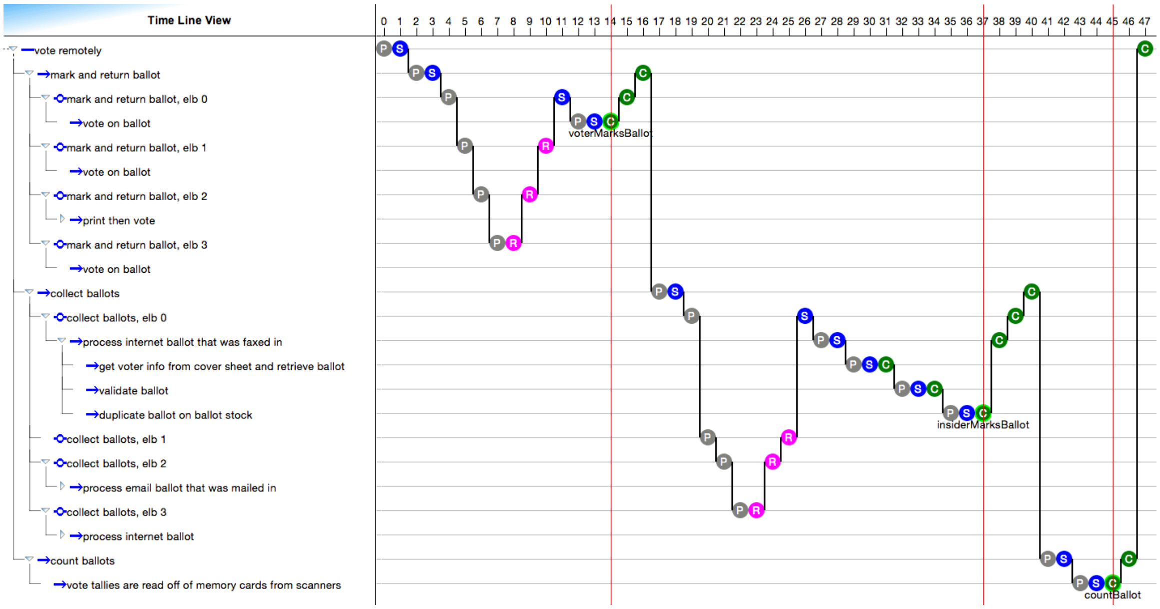


Figure F.3. The vote remotely process family including all four variants violated the property under consideration. The counterexample trace from FLAVERS is included above.

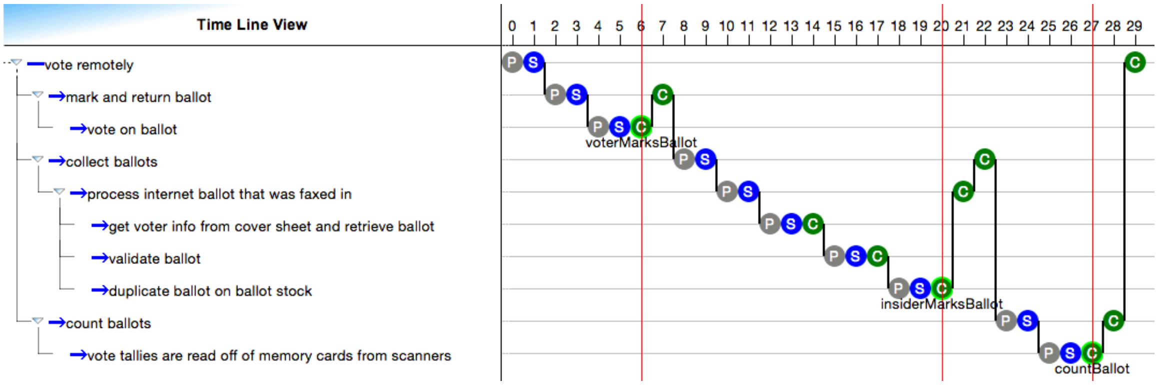


Figure F.4. The vote-by-fax process violated the property under consideration. The counterexample trace from FLAVERS is included above.

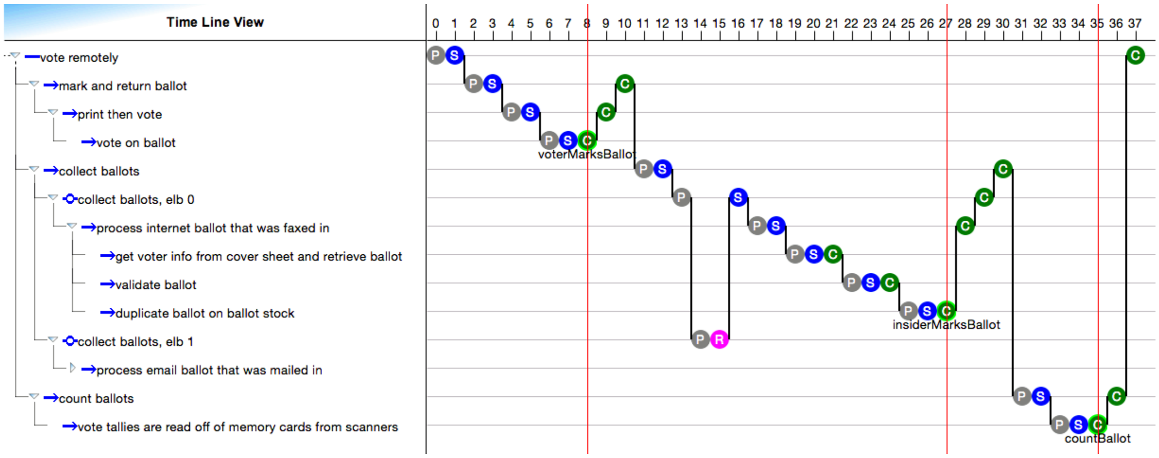


Figure F.5. The vote-by-email process violated the property under consideration. The counterexample trace from FLAVERS is included above.

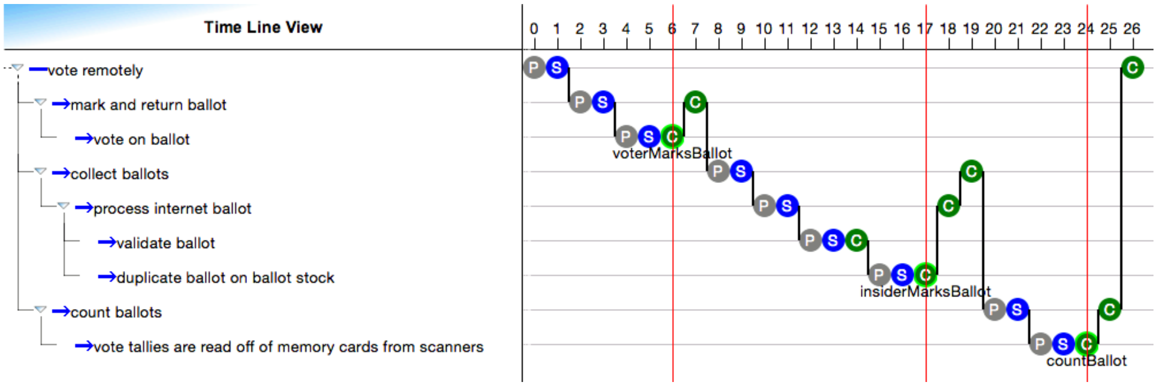


Figure F.6. The vote-by-ballot-on-demand process violated the property under consideration. The counterexample trace from FLAVERS is included above.

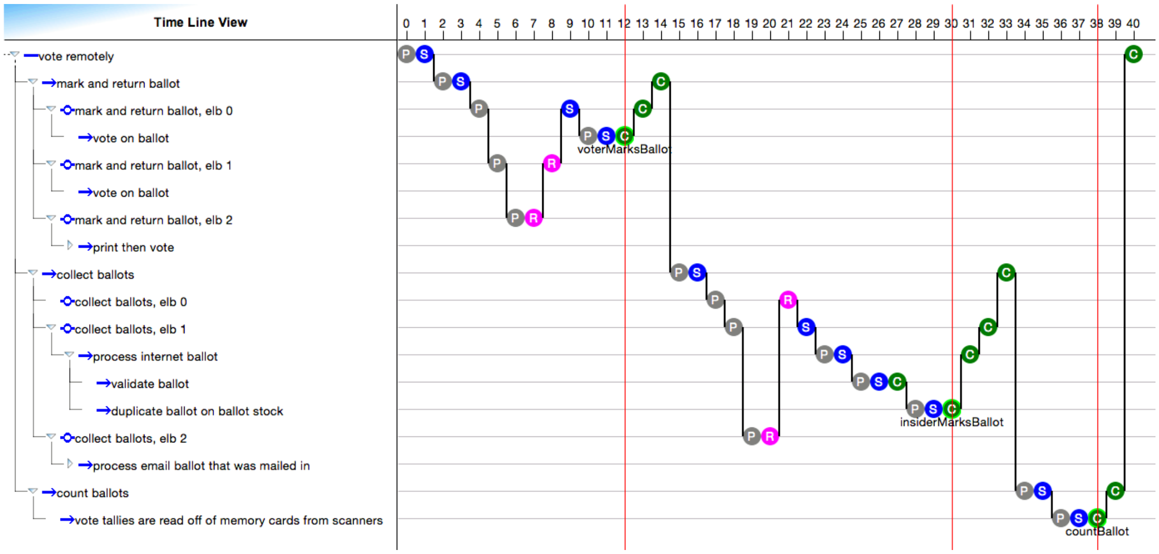


Figure F.7. The ALL-Fax process family including the three variants apart from vote-by-fax (vote-by-mail, vote-by-email, and vote-by-ballot-on-demand) violated the property under consideration. The counterexample trace from FLAVERS is included above.

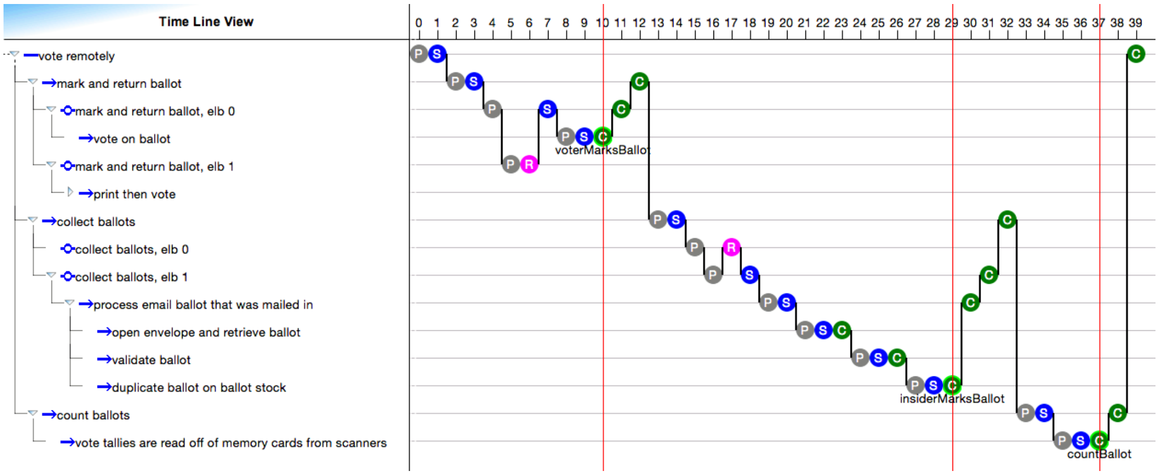


Figure F.8. The ALL-Fax-BoD process family including the two variants apart from vote-by-fax and vote-by-ballot-on-demand (i.e., vote-by-mail and vote-by-email) violated the property under consideration. The counterexample trace from FLAVERS is included above.

BIBLIOGRAPHY

[1] Apache Tapestry. http://tapestry.apache.org/.

[2] Google Web Toolkit. http://code.google.com/webtoolkit/.

[3] XML Path Language (XPath). http://www.w3.org/TR/xpath/.

[4] Anastasopoulos, M., and Muthig, D. An evaluation of aspect-oriented programming as a product line implementation technology. *Software Reuse: Methods, Techniques and Tools* (2004), 141–156.

[5] Apel, S., Leich, T., and Saake, G. Aspectual feature modules. *IEEE Trans. Softw. Eng. 34*, 2 (2008), 162–180.

[6] Apel, Sven, Janda, Florian, Trujillo, Salvador, and Kästner, Christian. Model superimposition in software product lines. In *Theory and Practice of Model Transformations* (2009), vol. 5563 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp. 4–19.

[7] Armbrust, Ove, Katahira, Masafumi, Miyamoto, Yuko, Munch, Jurgen, Nakao, Haruka, and Ocampo, Alexis. Scoping software process lines. *Softw. Process: Improvement Practice 14*, 3 (2009), 181–197.

[8] Asirelli, Patrizia, ter Beek, MauriceH., Fantechi, Alessandro, and Gnesi, Stefania. A model-checking tool for families of services. In *Formal Techniques for Distributed Systems*, vol. 6722 of *LNCS*. Springer Berlin Heidelberg, 2011, pp. 44–58.

[9] Atkinson, C., Bayer, J., and Muthig, D. Component-based product line development: The KobrA approach. In *SPLC: Proc. 1st Int. Softw. Prod. Line Conf.* (2000), pp. 289–309.

[10] Bachmann, Felix, and Bass, Len. Managing variability in software architectures. In *SSR '01: Proc. Symp. Softw. Reusability* (2001), pp. 126–132.

[11] Batory, D. Feature models, grammars, and propositional formulas. *Software Product Lines* (2005), 7–20.

[12] Batory, D., and O'Malley, S. The design and implementation of hierarchical software systems with reusable components. *ACM Trans. Softw. Eng. Methodol. 1*, 4 (1992), 355–398.

[13] Batory, Don. Feature-oriented programming and the AHEAD tool suite. In *ICSE '04: Proc. 26th Int. Conf. Softw. Eng.* (2004), pp. 702–703.

[14] Beuche, D. Modeling and building software product lines with pure:: variants. In *SPLC '08: Proc. 12th Int. Softw. Prod. Line Conf.* (2008), IEEE, p. 358.

[15] Boehm, Barry, and Belz, Frank. Experiences with the spiral model as a process model generator. In *Proceedings of the 5th international software process workshop on Experience with software process models* (1990), IEEE Computer Society, pp. 43–45.

[16] Brooke, Phillip J., and Paige, Richard F. Fault trees for security system design and analysis. *Computers & Security 22*, 3 (April 2003), 256–264.

[17] Cass, A.G., Sutton Jr, S.M., and Osterweil, L.J. Formalizing rework in software processes. *Software Process Technology* (2003), 16–31.

[18] Chan, Kit Yan, Kwong, CK, and Wong, TC. Modelling customer satisfaction for product development using genetic programming. *Journal of Engineering Design 22*, 1 (2011), 55–68.

[19] Chen, Bin. *Improving Processes Using Static Analysis Techniques*. PhD thesis, University of Massachusetts Amherst, 2010.

[20] Classen, Andreas, Cordy, Maxime, Heymans, Patrick, Legay, Axel, and Schobbens, Pierre-Yves. Model checking software product lines with SNIP. *International Journal on Software Tools and Technology Transfer 14*, 5 (2012), 589–612.

[21] Classen, Andreas, Heymans, Patrick, Schobbens, Pierre-Yves, Legay, Axel, and Raskin, Jean-François. Model checking lots of systems: efficient verification of temporal properties in software product lines. In *ICSE '10: Proc. 32nd Int. Conf. Softw. Eng.* (2010), pp. 335–344.

[22] Clements, P., and Northrop, L. *Software Product Lines–Practices and Patterns*. Addison-Wesley Prof., 2001.

[23] Conboy, Heather M., Avrunin, George S., and Clarke, Lori A. Process-based derivation of requirements for medical devices. In *Proceedings of the 1st ACM International Health Informatics Symposium* (New York, NY, USA, 2010), IHI '10, ACM, pp. 656–665.

[24] Czarnecki, K., and Eisenecker, U.W. Components and generative programming. In *ESEC '99/FSE-7: Proc. 7th Euro. Softw. Eng. Conf./7th ACM SIGSOFT Int. Symp. Found. Softw. Eng.* (1999), pp. 2–19.

[25] Czarnecki, K., and Eisenecker, U.W. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[26] Czarnecki, K., Helsen, S., and Eisenecker, U. Staged configuration using feature models. *Software Product Lines* (2004), 162–164.

[27] Czarnecki, K., and Pietroszek, K. Verifying feature-based model templates against well-formedness OCL constraints. In *Proceedings of the 5th international Conference on Generative Programming and Component Engineering* (2006), ACM, pp. 211–220.

[28] Czarnecki, Krzysztof, Antkiewicz, Michal, Kim, Chang Hwan Peter, Lau, Sean, and Pietroszek, Krzysztof. Model-driven software product lines. In *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 2005), OOPSLA '05, ACM, pp. 126–127.

[29] da Mota Silveira Neto, Paulo Anselmo, Carmo Machado, Ivan do, McGregor, John D, De Almeida, Eduardo Santana, and de Lemos Meira, Silvio Romero. A systematic mapping study of software product lines testing. *Information and Software Technology 53*, 5 (2011), 407–423.

[30] Dhungana, D., and Groher, I. Genetics as a role model for software variability management. In *Software Engineering - Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on* (May 2009), pp. 239–242.

[31] Dwyer, Matthew B., Clarke, Lori A., Cobleigh, Jamieson M., and Naumovich, Gleb. Flow analysis for verifying properties of concurrent software systems. *ACM Trans. Softw. Eng. Methodol. 13*, 4 (2004), 359–430.

[32] Ericson, Clifton A. Fault tree analysis—a history. In *Proceedings of the 17th International System Safety Conference* (1999), pp. 1–9.

[33] Gacek, C., and Anastasopoules, M. Implementing product line variabilities. In *Proc. 2001 Symp. Softw. Reusability* (2001), pp. 109–117.

[34] Gomaa, Hassan. *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Addison-Wesley, 2004.

[35] Gomaa, Hassan, Kerschberg, Larry, and Farrukh, Ghulam A. Domain modeling of software process models. *Proceedings Sixth IEEE International Conference on Engineering of Complex Computer Systems* (2000), 50–60.

[36] Goues, Claire Le, Nguyen, ThanhVu, Forrest, Stephanie, and Weimer, Westley. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering 38*, 1 (2012), 54–72.

[37] Gruler, Alexander, Leucker, Martin, and Scheidemann, Kathrin. Modeling and model checking software product lines. In *Formal Methods for Open Object-Based Distributed Systems*, vol. 5051 of *LNCS*. Springer Berlin Heidelberg, 2008, pp. 113–131.

[38] Hallerbach, A., Bauer, T., and Reichert, M. Capturing variability in business process models: the Provop approach. *Journal of Software Maintenance and Evolution: Research and Practice - Best papers from the BPM 2008 Workshops* (2010).

[39] Harman, Mark, Jia, Yue, and Langdon, WilliamB. Babel pidgin: Sbse can grow and graft entirely new functionality into a real world system. In *Search-Based Software Engineering*, Claire Le Goues and Shin Yoo, Eds., vol. 8636 of *Lecture Notes in Computer Science*. Springer International Publishing, 2014, pp. 247–252.

[40] Heuer, André, Budnik, Christof J., Konrad, Sascha, Lauenroth, Kim, and Pohl, Klaus. Formal definition of syntax and semantics for documenting variability in activity diagrams. In *SPLC '10: Proc. 14th Int. Softw. Prod. Line Conf.* (2010), pp. 62–76.

[41] Jazayeri, M., and Linden, F. Van Der. *Software Architecture for Product Families: Principles and Practice*. Addison-Wesley, 2000.

[42] Kang, K.C. et al. Feature-oriented domain analysis (FODA) feasibility study. Tech. rep., Software Engineering Institute, Carnegie Mellon University, 1990.

[43] Kang, Kyo C., Kim, Sajoong, Lee, Jaejoon, Kim, Kijoo, Shin, Euiseob, and Huh, Moonhang. FORM: A feature-oriented reuse method with domain specific reference architectures. *Annals Softw. Eng. 5*, 1 (1998), 143–168.

[44] Kästner, C., and Apel, S. Type-checking software product lines-a formal approach. In *ASE '08: Proc. 23rd Int. Conf. Automated Softw. Eng.* (2008), pp. 258–267.

[45] Kästner, C., Thüm, T., Saake, G., Feigenspan, J., Leich, T., Wielgorz, F., , and Apel, S. FeatureIDE: a tool framework for feature-oriented software development. In *ICSE '09: Proc. 31st Int. Conf. Softw. Eng.* (2009), pp. 611–614.

[46] Kästner, Christian, Apel, Sven, and Kuhlemann, Martin. Granularity in software product lines. In *ICSE '08: Proc. 30th Int. Conf. Softw. Eng.* (2008), pp. 311–320.

[47] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J., and Irwin, J. Aspect-oriented programming. In *ECOOP '97: Proc. 11th Euro. Conf. Object-Oriented Program.* (1997), pp. 220–242.

[48] Knauber, S.J.a.P. Synergy between component-based and generative approaches. In *ESEC '99/FSE-7: Proc. 7th Euro. Softw. Eng. Conf./7th ACM SIGSOFT Int. Symp. Found. Softw. Eng.* (1999), pp. 2–19.

[49] Lauenroth, K., Pohl, K., and Toehning, S. Model checking of domain artifacts in product line engineering. In *2009 IEEE/ACM International Conference on Automated Software Engineering* (2009), IEEE, pp. 269–280.

[50] Lee, Jihyun, Kang, Sungwon, and Lee, Danhyung. A survey on software product line testing. In *Proceedings of the 16th International Software Product Line Conference-Volume 1* (2012), ACM, pp. 31–40.

[51] Linsbauer, Lukas, Lopez-Herrejon, Roberto Erick, and Egyed, Alexander. Feature model synthesis with genetic programming. In *Search-Based Software Engineering*. Springer International Publishing, 2014, pp. 153–167.

[52] Liu, J., Basu, S., and Lutz, R.R. Compositional model checking of software product lines using variation point obligations. *Automated Software Engineering 18*, 1 (2011), 39–76.

[53] Lopez-Herrejon, Roberto Erick, Galindo, José A, Benavides, David, Segura, Sergio, and Egyed, Alexander. Reverse engineering feature models with evolutionary algorithms: An exploratory study. In *Search Based Software Engineering*. Springer, 2012, pp. 168–182.

[54] Martínez-Ruiz, T., García, F., and Piattini, M. Towards a SPEM v2. 0 Extension to Define Process Lines Variability Mechanisms. *Software Engineering Research, Management and Applications* (2008), 115–130.

[55] Martínez-Ruiz, Tomás, García, Félix, Piattini, Mario, and De Lucas-Consuegra, Francisco. Process variability management in global software development: a case study. In *ICSSP: Int. Conf. Softw. Sys. Process* (2013), pp. 46–55.

[56] Martínez-Ruiz, Tomás, Münch, Jürgen, García, Félix, and Piattini, Mario. Requirements and constructors for tailoring software processes: a systematic literature review. *J. Softw. Quality 20*, 1 (2012), 229–260.

[57] Mohalik, Swarup, Ramesh, S., Millo, Jean-Vivien, Krishna, Shankara Narayanan, and Narwane, Ganesh Khandu. Tracing spls precisely and efficiently. In *SPLC '12: Proceedings of the 16th International Conference on Software Engineering* (2012), pp. 186–195.

[58] Pohl, K., Böckle, G., and Van Der Linden, F. *Software product line engineering: foundations, principles, and techniques*. Springer-Verlag New York Inc, 2005.

[59] Pohl, Klaus, and Metzger, Andreas. Variability management in software product line engineering. In *ICSE '06: Proc. 28th Int. Conf. Softw. Eng.* (2006), pp. 1049–1050.

[60] Raunak, Mohammad S., and Osterweil, Leon J. *Resource Management in Complex, Dynamic Environments*. PhD thesis, University of Massachusetts Amherst, 2010.

[61] Ribó, Josep M., and Franch, Xavier. A precedence-based approach for proactive control in software process modelling, 2002.

[62] Schmid, Klaus, and van der Linden, Frank. Introducing and optimizing software product lines using the FEF. In *SPLC '09: Proc. 13th Int. Softw. Prod. Line Conf.* (2009), pp. 311–311.

[63] Schneider, Fred B. Beyond hacking: An SOS! Keynote at the 32nd International Conference on Software Engineering (ICSE), May 2010.

[64] Schneier, Bruce. Modeling security threats. *Dr. Dobb's Journal 22*, 12 (December 1999), 4–6.

[65] Schobbens, Pierre-Yves, Heymans, Patrick, and Trigaux, Jean-Christophe. Feature diagrams: A survey and a formal semantics. In *IEEE Int. Conf. Requir. Eng.* (2006), pp. 139–148.

[66] Simidchieva, B. I., Engle, S.J., Clifford, M., Jones, A. Clay, Peisert, S., Bishop, M., Clarke, L. A., and Osterweil, L. J. Modeling and analyzing faults to improve election process robustness. In *EVT/WOTE '10: Proc. 2010 Electronic Voting Tech. Workshop/Workshop Trustworthy Elections* (2010).

[67] Simidchieva, B. I., and Osterweil, Leon J. Generation, composition, and verification of families of human-intensive systems. In *Proceedings of SPLC '14: the 18th International Software Product Line Conference* (2014).

[68] Simidchieva, B.I., and Osterweil, L.J. Characterizing process variation: NIER track. In *ICSE '11: 33rd Int. Conf. Softw. Eng.* (2011), pp. 836–839.

[69] Simidchieva, Borislava I., Clarke, Lori A., and Osterweil, Leon J. Representing process variation with a process family. In *Software Process Dynamics and Agility: Proceedings of the International Conference on Software Process* (2007), Qing Wang, Dietmar Pfahl, and David M. Raffo, Eds., vol. 4470 of *LNCS*, Springer, pp. 109–120.

[70] Simidchieva, Borislava I., and Osterweil, Leon J. Categorizing and modeling variation in families of systems: a position paper. In *ESCA '10: Proc. 4th Euro. Conf. Softw. Arch.* (2010), pp. 316–323.

[71] Simidchieva, Borislava I., Osterweil, Leon J., and Wise, A. Structural considerations in defining executable process models. In *Proceedings of the International Conference on Software Process* (2009), Qing Wang, Ray Madachy, and Dietmar Pfahl, Eds., vol. 5543 of *LNCS*, Springer, pp. 366–376.

[72] Sinnema, Marco, Deelstra, Sybren, Nijhuis, Jos, and Bosch, Jan. COVAMOF: a framework for modeling variability in software product families. In *Software Product Lines* (2004), vol. 3154 of *LNCS*, pp. 25–27.

[73] Smaragdakis, Yannis, and Batory, Don. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM Trans. Softw. Eng. Methodol. 11*, 2 (2002), 215–255.

[74] Smith, Rachel L., Avrunin, George S., Clarke, Lori A., and Osterweil, Leon J. PROPEL: an approach supporting property elucidation. In *ICSE '02: Proc. 24th Int. Conf. Softw. Eng.* (2002), pp. 11–21.

[75] Svahnberg, Mikael, van Gurp, Jilles, and Bosch, Jan. A taxonomy of variability realization techniques. *Software: Practice and Experience 35*, 8 (July 2005), 705–754.

[76] Thaker, S., Batory, D., Kitchin, D., and Cook, W. Safe composition of product lines. In *Proc. 6th Int. Conf. Generative Program. Comp. Eng.* (2007), pp. 95–104.

[77] Trujillo, Salvador, Batory, Don, and Diaz, Oscar. Feature oriented model driven development: A case study for portlets. In *ICSE '07: Proc. 29th Int. Conf. Softw. Eng.* (2007), pp. 44–53.

[78] van Der Aalst, W. M. P., Pesic, M., and Schonenberg, H. Declarative workflows: Balancing between flexibility and support. *Computer Science-Research and Development 23*, 2 (2009), 99–113.

[79] van der Aalst, Wil M. P., Dumas, Marlon, Gottschalk, Florian, ter Hofstede, Arthur H. M., La Rosa, Marcello, and Mendling, Jan. Preserving correctness during business process model configuration. *Formal Aspects of Computing* (April 2009), 1–25.

[80] van der Aalst, W.M.P., Dumas, M., Gottschalk, F., ter Hofstede, A.H.M., La Rosa, M., and Mendling, J. Correctness-Preserving Configuration of Business Process Models. In *Fundamental approaches to software engineering: 11th international conference, FASE 2008, held as part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008: proceedings* (2008), Springer-Verlag New York Inc, p. 46.

[81] van Ommering, Rob. Building product populations with software components. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering* (Los Alamitos, CA, USA, 2002), IEEE Computer Society, pp. 255–265.

[82] van Ommering, Rob, van der Linden, Frank, Kramer, Jeff, and Magee, Jeff. The Koala component model for consumer electronics software. *IEEE Computer 33*, 3 (2000), 78–85.

[83] Washizaki, Hironori. Building Software Process Line Architectures from Bottom Up. In *Product-Focused Softw. Process Improvement (PROFES)* (2006), pp. 415–421.

[84] Weiss, David M., and Lai, Chi Tau Robert. *Software product-line engineering: a family-based software development process*. Addison-Wesley, 1999.

[85] White, Stephen A. Business process modeling notation (bpmn). Tech. Rep. formal/2009-01-03, Business Process Management Initiative (BPMI), January 2009.

[86] Zhu, Liming, Osterweil, Leon J., Staples, Mark, Kannengiesser, Udo, and Simidchieva, Borislava I. Desiderata for languages to be used in the definition of reference business processes. *International Journal of Software and Informatics 1*, 1 (December 2007), 37–65.