A Framework for Dependability Analysis of Software Systems with Trusted Bases

by

Eunsuk Kang

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2010

Author .....
Department of Electrical Engineering and Computer Science
January 4, 2010

Certified by .....
Daniel N. Jackson
Professor
Thesis Supervisor

Accepted by .....
Terry P. Orlando
Chairman, Department Committee on Graduate Students

A Framework for Dependability Analysis of Software Systems with Trusted Bases

by

Eunsuk Kang

Submitted to the Department of Electrical Engineering and Computer Science
on January 4, 2010, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

A new approach is suggested for arguing that a software system is dependable. The key idea is to structure the system so that highly critical requirements are localized in small subsets of the system called *trusted bases*. In most systems, the satisfaction of a requirement relies on assumptions about the environment, in addition to the behavior of software. Therefore, establishing a trusted base for a critical property must be carried out as early as the requirements phase.

This thesis proposes a new framework to support this activity. A notation is used to construct a dependability argument that explains how the system satisfies critical requirements. The framework provides a set of analysis techniques for checking the soundness of an argument, identifying the members of a trusted base, and illustrating the impact of failures of trusted components. The analysis offers suggestions for redesigning the system so that it becomes more reliable. The thesis demonstrates the effectiveness of this approach with a case study on electronic voting systems.

Thesis Supervisor: Daniel N. Jackson
Title: Professor

Acknowledgments

I feel incredibly fortunate to have been under Daniel's guidance ever since I arrived at MIT. Without his insights, constant encouragements, humor, and tolerance for my frequent ramblings, I would never made headway in this research. He has inspired me to always look at the big picture. Among many other things, I am also thankful to him for picking on fonts and color schemes on my talk slides; his attention to subtle details such as these have also made me a better researcher and communicator.

I would like to thank my former advisers—Mark Aagaard, Joanne Atlee, and Nancy Day—for all their guidance and help during my undergraduate years. While working with them, I become interested in formal methods and gained my first exposure to academic research. Without them, I would not be where I am today.

My mentor at Microsoft Research, Ethan Jackson, has taught me so many things over the short period of three months in the summer. His appreciation of the right balance between theory and application is a skill that I will strive to perfect throughout the rest of my career.

I owe big thanks to my colleagues in the Software Design Group. Derek has been like a big brother to all of us newcomers in the group; without his enthusiasm and sage advice, I would probably fallen off the cliff numerous times. I would also like to thank Emina, with whom I shared the office, for sharing her wisdom and keeping me from steering off course; Rob, for making me think about the things that really matter; Greg, for teaching me about program analysis; Felix, for his amazing work on Alloy; and Carlos, for showing me the power of randomization. I also have to thank the members of the younger generation—Joe, Aleks, Rishabh, Jean, and Kuat, Zev, and Christian—for bringing new energy and enthusiasm to the group, patiently sitting through all my rants and rave, and above of all, making the Sofa Lounge a fun, enjoyable place to work (or slack off). Special thanks goes to Vijay Ganesh for his challenging discussions that have kept me oriented on different sides of the story.

I am grateful to all my friends at MIT and back home in Canada. Despite numerous years in school, my sanity still remains intact, thanks to their companionship and encouragements.

Finally, I dedicate this thesis to my parents and my wonderful sister. Words simply cannot express my gratitude to them.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

A traditional formulation of an argument for requirements satisfaction takes the shape

$$D, S \vdash R \tag{1.1}$$

where D represents a set of assumptions about domains, S the specifications of machines that implement software, and R the set of system requirements. That is, if all domain assumptions hold and machines meet their specifications, then the satisfaction of desired requirements must follow.

A system cannot be considered dependable based solely on this argument. Merely showing that the system satisfies its requirements under positive scenarios does not necessarily elicit a high level of confidence. In a system where safety and security are paramount concerns, it is unacceptable for a single glitch in a machine or an unexpected behavior of the environment to lead to a catastrophic failure. For example, one may argue that in an electronic voting system, the correctness of the total tally for each candidate is achieved when all participating members (voters, poll workers, and election authority) behave as expected, and the voting software meets its specification. But it would be rather ill-advised to jump to a conclusion that this system is dependable; a malicious poll worker, for instance, or a security exploit in the voting machine can compromise the integrity of the election [24].

In [14], Jackson proposed a new approach to arguing for the dependability of a software system. One of the key steps in this approach is to structure the system so that a critical requirement is localized to a small subset of the system's components that can be readily shown to be reliable. In the voting example, it may be desirable to design the system so that a security flaw in software cannot influence the outcome of the election [34]. On the other hand, it seems reasonable to place trust in the hands of the election authority; even if it decides to carry out a malicious act, it may be possible to detect this by available mechanisms (e.g. third-party auditing).

The idea of localizing critical properties for dependability is not new. In the security literature, the *trusted computing base* (TCB) is a well-known notion that describes the set of all software and hardware components in a system that are critical

to its security [26]. Rushby proposed an analogous version of the TCB for safety properties, called a safety kernel [35], where critical properties are enforced by a small core of the system.

However, existing approaches to establishing a TCB focus on the design of machines, past the requirements stage, well after the interfaces of the machines with the environment have already been determined. But this may be too late in the development. The satisfaction of a requirement relies on assumptions about the environment, in addition to the behavior of software. Therefore, decisions regarding the membership of a TCB (i.e. which components should be included in it) must involve the discussion of domain properties as well.

Furthermore, for a complex system, it is far from trivial to identify a TCB that is sufficient (i.e. it does not incorrectly exclude a critical component) and minimal (i.e. it does not contain a component that is not critical to the requirement). The first criterion ensures the soundness of an argument that a critical requirement needs to rely only on the TCB. The second criterion is just as important, because an unnecessarily large TCB may incur extra costs on testing and verification. However, in existing approaches, a justification for the scope of a TCB is often stated informally and thus, is not amenable to rigorous analysis.

We argue that in order to ensure the dependability of a system, establishing trusted bases for critical requirements should be carried out as a key activity during the requirements analysis phase. In our work, we use the term *trusted base*, instead of *trusted computing base*, since the set of critical components may also include non-computing entities in the environment, such as a human operator or a mechanical device.

In this thesis, we describe a model-based framework to support this activity. We envision that a requirement analyst will carry out the following tasks using our framework:

1. Construct an initial argument for the satisfaction of a critical requirement, which we call a *dependability argument*.

2. Apply rigorous analysis to check the consistency of the argument and identify a trusted base for the requirement.

3. Examine various scenarios in which certain members of the trusted base behave unexpectedly. This step helps the analyst determine the likelihood and impact of the failures of trusted components.

4. Attempt to justify, for each component in the trusted base, why the component can be reliably trusted. A justification may involve discharging assumptions about the environment with the knowledge of a domain expert, or showing that software can be built with a high degree of confidence to satisfy its specification.

5. If the trusted base contains components that cannot be trusted, then redesign the system so that a new trusted base no longer relies on these components.

This process may be repeated until the analyst arrives at a design with arguably reliable trusted bases for critical requirements.

1.2 Contributions

The contributions of this thesis include the following:

- A modeling notation, based on the Problem Frames approach [18], for constructing dependability arguments, and its formalization in Alloy [13] (Chapter 2).
- An analysis framework, which exploits the power of the Alloy Analyzer, for:
  1. checking the consistency of a dependability argument (Section 3.1).
  2. identifying and evaluating a trusted base for a requirement (Section 3.2).
  3. systematically generating failure scenarios using meta-objects (Section 3.3).
- A new model of dependence, called a property-part diagram, for showing the structure of a dependability argument and evaluating the modularity of design (Chapter 4).

In addition, we demonstrate our approach with a case study of electronic voting systems (Chapter 5). We describe models of two different voting systems: (1) an optical scan system, which is widely used in the U.S., and (2) the *Scantegrity* system [2], which is an extension to the optical scan system that provides *end-to-end verifiability*. Researchers have studied existing optical scan systems and identified various security flaws, many of which involve the optical scanner itself [5, 11, 23]. Scantegrity is intended to yield a strong confidence that the failure of an unreliable component, such as the scanner, does not compromise the outcome of the election. But dependability of a system does not grow out of vacuum; Scantegrity still relies on the correctness of its own trusted base. We show how our framework can be used to identify components in the trusted base, and provide a strong argument for why Scantegrity is more dependable than an optical scan system.

Chapter 2

Notation

2.1 Example: An Optical Scan Voting System

We begin the discussion of our framework with a model of an electronic voting system (Figure 2-1). The model is based on the descriptions of optical scan voting systems in [11, 23].

Our notation builds on the Problem Frames Approach [18], and from it inherits its basic modeling constructs: *phenomena*, *domains*, *machines*, and *requirements*. We chose the Problem Frames Approach as the basis for our framework, as opposed to other modeling notations, because (1) it focuses on interactions between machines and domains, and (2) it represents requirements explicitly as language constructs that act directly on domains. These two properties allow us to trace requirements through the system and highlight components that belong to a trusted base.

Figure 2-1 is a variant of Jackson's *program diagram* [18], which illustrates how a machine (a lined box) interacts with one or more domains (boxes) in the world to satisfy a requirement (a dotted oval). Jackson advocates that a top-level problem (i.e. a system-level requirement) be decomposed into smaller subproblems, each of which is solved in separation by building a machine. A typical problem diagram corresponds to one subproblem and contains exactly one requirement, one machine, and one or more domains. The subproblems are then composed again to show how the system as a whole satisfies the top-level requirement. The process of decomposition and recomposition is a research question that has been tackled elsewhere [15, 27]. Here, we assume that the analyst has already completed this step before constructing a satisfaction argument. Therefore, our diagram differs from a typical problem diagram in that we allow multiple requirements and machines in a single diagram.

A standard use case scenario in an optical scan system is as follows. A voter enters a polling station, with an intent to vote for a candidate. First, the voter walks to the check-in desk, which is staffed by poll workers, and receives exactly one ballot. Then, the voter walks to one of the available booths and casts his or her vote by filling the oval at the appropriate position on the ballot; from this point, we consider the ballot to be *a marked ballot*. Next, the voter walks to an optical scanner and inserts the marked ballot into the machine; the machine records the position of the filled oval

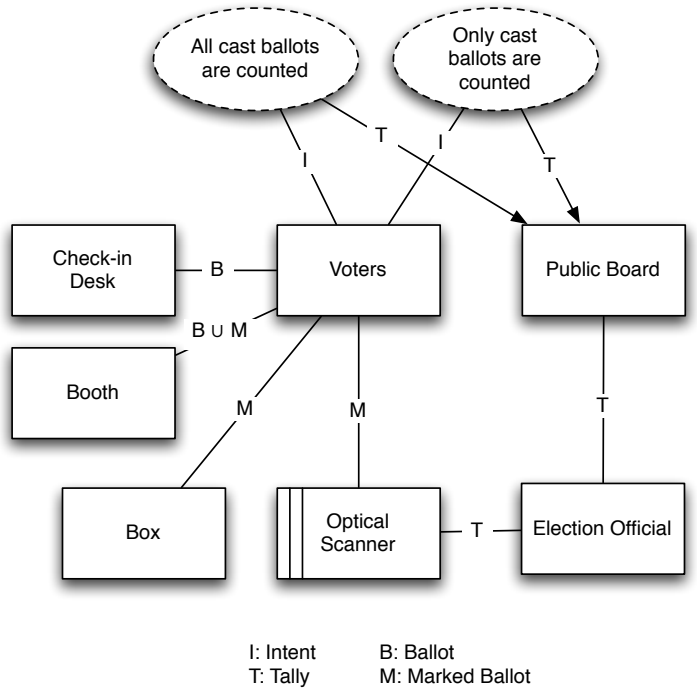


Figure 2-1: A problem diagram for an optical scan voting system

and returns the marked ballot. In the final step, the voter takes the marked ballot and inserts it into a box, which will contain physical ballots that can be used for a recount in case of a dispute.

After the poll has been closed, the optical scanner computes the final tally for each candidate and stores the result, along with the set of all ballot records, into a physical device, such as a flash drive. An election official retrieves the outcome of the election from the scanner and makes it available to the public.

The system must satisfy two critical *integrity* requirements[1]. First, it must ensure that every vote that is cast by the voter is actually included in the final tally. Secondly, only those ballots that were actually cast should be included in the final tally (i.e. no spurious votes). These two requirements together imply that the outcome of the election reflects exactly the intents of the voters.

2.2 Basic Constructs

In the following sections, we provide a formalization of our notation in Alloy [13], a first-order relational logic with transitive closure. We chose Alloy as an underlying formalism for two reasons. First, it is essentially pure logic, without any built-in idioms (compared to state-machine idioms in B [1] or VDM [20]), and its flexibility

[1]Every voting system must also ensure *privacy* of the voters; that is, it should not be possible to trace a marked ballot to a particular voter. For our purpose, we omit the discussion of the privacy requirement.

allows an unconventional structuring of system components and their specifications, which we exploit heavily. Second, its analysis engine, the Alloy Analyzer, allows automated consistency checking of specifications as well as discovery of unsatisfiable cores [43], which are important ingredients of our analysis framework. Third, the analyzer also has a built-in capability to visualize system snapshots, which is immensely helpful for understanding interactions among different components in the system.

Each system consists of domains, machines, and requirements:

```
abstract sig Domain {}
abstract sig Machine {}
abstract sig Requirement {}
```

A domain represents a portion of the environment, and has associated with it a collection of *phenomena*. Consider the following fragment of Alloy, which shows the *Voters* domain and all of its related phenomena:

```
one sig Voters extends Domain { // the Voters domain
  members : set Voter
}
sig Voter { // an individual voter
  intent : lone Candidate
}
sig Candidate {}
sig Tally { // number of votes for each candidate
  counts : Candidate -> one Int
}
sig Ballot {
  givenTo : Voter // voter that possesses this ballot
}
sig MBallot { // marked ballot
  ballot : Ballot,
  choice : lone Candidate,
  markedBy : Voter // voter that completed this ballot
}
fact AtMostOneMarkedBallot {
  ballot in MBallot lone -> lone Ballot
}
```

There are two categories of phenomena: *individuals* and *relations*. An individual may refer to a distinct entity in the physical world (e.g. a voter, a candidate, or a piece of ballot) or an abstract value (e.g. a positive integer tally). We represent a group of individuals as an unary relation, which we can define in Alloy using a signature. For example, the signature *Voter* introduces a set of phenomena that correspond to the voters who participate in the election. Other individuals in the above fragment of the voting system model include *Candidate*, *Tally*, *Ballot*, and *MBallot*.

A relation is a set of associates among two or more individuals. For example, the intent of a voter is a relation that associates the voter with the candidate that he or she intends to vote for. In Alloy, we represent multi-arity relations as fields of a

signature. For instance, the field *intent* of the signature *Voter* is a binary relation from *Voter* to *Candidate*. Other relational phenomena in the above fragment include *counts*, *givenTo*, *ballot*, *choice*, and *markedBy*.

Each ballot physically belongs to exactly one voter (*givenTo*), and becomes a marked ballot (*MBallot*) when the voter (*markedBy*) fills in his or her choice. The fact *AtMostOneMarkedBallot* ensures that each marked ballot corresponds to a unique physical ballot. We consider this to be a reasonable assumption, since a single marked ballot cannot physically originate from more than one piece of empty ballots (at least, in this universe). Conversely, marking a single ballot most likely cannot result in multiple, distinct pieces of marked ballots.

A machine represents the solution to the problem; it contains software that will be designed and implemented to satisfy the customer's requirements. It interacts with one or more domains by *observing* or *constraining* them, or both. We say that a machine *observes* a domain if the former queries the latter for a piece of information, but does not change it. We say that a machine *constrains* a domain if the former acts on the latter to change its state. Consider the following fragment of the Alloy model.

```
one sig OpticalScanner extends Machine {
  voters : Voters, // interacting domain
  records : set BallotImage, // internal phenomenon
  tally : Tally // shared phenomenon
}
sig BallotImage { // an electronic record for a ballot
  choice : lone Candidate,
  mballot : MBallot
}
one sig ElectionOfficial extends Domain {
  scanner : OpticalScanner,
  pboard : PublicBoard
}
one sig PublicBoard extends Domain {
  tally : Tally
}
```

The *OpticalScanner* machine observes the *Voters* domain by reading each voter's marked ballot and recording the choice into an electronic ballot image; the scanner does not constrain the voters. Ballot images are an example of an *internal phenomenon* because they are not visible to an external machine or domain (similarly to Parnas's information hiding [30]).

After all ballots have been cast, the scanner computes a tally based on the set of ballot images. The tally is an example of a *shared phenomenon*, because it is visible to another machine or domain—in this case, the election official.

A domain may also observe or constraint another domain or machine. For example, *ElectionOfficial* observes *OpticalScanner* by reading off the tally in the machine, and constrains *PublicBoard* by posting the tally onto the board and thereby changing the latter's state.

A requirement is a problem that must be solved by constructing one or more

machines. Because the requirement is expressed usually in terms of phenomena that appear in domains, not machines, one of the analyst's tasks is to derive the specification of the machine from the requirement and domain descriptions; this is an orthogonal issue that has been addressed in a previous work [39]. Like a machine, a requirement may either observe or constrain a domain.

```
one sig AllCastTallied extends Requirement {
  voters : Voters, // observes
  result : PublicBoard // constrains
}
one sig OnlyCastTallied extends Requirement {
  voters : Voters,
  result : PublicBoard
}
```

Each of the two requirements in the voting system observes the voters' intents, and constrains the number of votes that each candidate should receive.

2.3 Constraints

We have so far discussed the basic constructs of a model—domains, machines, and requirements—and relationships among them. Associated with each one of these constructs is a *constraint*. Three categories of constraints appear in a model. A domain is associated with a set of assumptions that describe its expected behaviors or properties. A machine is assigned a specification that it must fulfill. A requirement itself is a constraint that describes the customer's expectations on the states of the domains that it references. In this section, we describe how we specify these constraints in Alloy.

We first introduce three new signatures: *Constraint*, and its two subsets, *Satisfied* and *Unsatisfied*. We also add an Alloy *fact* to ensure that *Satisfied* and *Unsatisfied* form a complete, disjoint partition of *Constraint*. Through the subtyping relationships in Alloy, every domain, requirement, and requirement becomes an instance of either *Satisfied* or *Unsatisfied*:

```
abstract Constraint {}
sig Satisfied in Constraint {}
sig Unsatisfied in Constraint {}

fact {
  no (Satisfied & Unsatisfied) // disjointness
  Constraint = Satisfied + Unsatisfied // complete partition
}

abstract sig Domain extends Constraint {}
abstract sig Machine extends Constraint {}
abstract sig Requirement extends Constraint {}
```

A membership to these subset signatures indicates whether the member satisfies its constraint. For example, a domain belongs to *Satisfied* if and only if all of its assumptions hold; if it behaves unexpectedly for some reason (e.g. a malicious voter) and fails to satisfy one or more of its assumptions, it belongs to *Unsatisfied*. A *Satisfied* machine represents a piece of software that is implemented correctly[2]. Similarly, a requirement holds true if and only if it is a member of *Satisfied*. These two subset signatures are examples of *meta-objects*, because they are used to characterize or manipulate the *base-objects* (i.e. domains, machines, and requirements).

In Alloy, a *signature fact*, associated with a signature `T`, is a constraint that must hold true of every object of `T`. We use signature facts to assign constraints to a domain. For example, one desirable assumption about the voters is that they cast their ballots exactly as intended. We state the Voters domain to be satisfying if and only if every voter in the domain behaves as desired:

```
one sig Voters {
  members : set Voter
}{ // signature fact
  this in Satisfied iff
  all v : members | (markedBy.v).choice = v.intent
}
```

Again, we use signature facts to assign a specification to a machine. The specification of the optical scanner is twofold: (1) given a marked ballot by a voter, it must store the choice on the ballot into a ballot image, and (2) it must compute the total tally for each candidate:

```
one sig OpticalScanner extends Machine {
  voters : Voters,
  records : set BallotImage,
  tally : Tally
}{
    this in Satisfied
  iff
    (all mb : markedBy.(voters.members) |
      some rec : records |
        rec.choice = mb.choice &&
        rec.mballot = mb)
    &&
    (all c : Candidate |
      tally.counts[c] = #(records & choice.c))
}
```

The responsibility of the election official is to observe the tally inside the scanner

[2]A machine M that is implemented correctly does not necessarily meet its specification S, if it uses another machine N to satisfy S; one must show that N meets its own specification as well. This is called *contingent use*, which we describe in more detail in Section 4.3. On the other hand, if M is not implemented correctly, then clearly it violates its specification.

and directly post it to the public board. This behavior, again, can be expressed as a domain constraint:

```
one sig ElectionOfficial extends Domain {
  scanner : OpticalScanner,
  pboard : PublicBoard
}{
  this in Satisfied iff
  scanner.tally = pboard.tally
}
one sig PublicBoard extends Domain {
  tally : Tally
}{
  this in Satisfied
}
```

Note that *PublicBoard* has no associated constraints. As a simplification, we assume that this is a singleton domain whose sole purpose is to encapsulate the phenomenon *tally*.

Finally, we specify the requirement *AllCastTallied* to be satisfied when the number of votes that a candidate receives is at least the number of voters with an intent to vote for that candidate. Similarly, *OnlyCastTallied* is satisfied when a candidate's tally is at most the number of intentional voters.

```
one sig AllCastTallied extends Requirement {
  voters : Voters,
  result : PublicBoard
}{
  this in Satisfied iff
  all c : Candidate | result.tally.counts[c] >= #(intent.c)
}
one sig OnlyCastTallied extends Requirement {
  voters : Voters,
  result : PublicBoard
}{
  this in Satisfied iff
  all c : Candidate | result.tally.counts[c] <= #(intent.c)
}
```

As we shall see in Section 3.3, making the constraints explicit as objects, and classifying them as *Satisfied* or *Unsatisfied*, allow us to conveniently generate failures and examine their consequences on the requirements. This style of modeling, where objects are mixed with meta-objects, may seem somewhat unorthodox. However, our approach is not new. Hoare, in his 1984 paper [10], attaches to every process in a program a Boolean flag called *stable*, which indicates whether the process is behaving properly. By doing so, he effectively classifies all processes into two groups (stable and unstable processes), similarly to the way we partition components into *Satisfied* and *Unsatisfied*.

2.4 Well-Formedness Conditions

Not every Alloy model is a valid instance of specifications in our notation. We state two conditions for the well-formedness of a model:

- There should be *no dangling component* in the model. In other words, every domain must be connected to another domain through some shared phenomena, or be observed or constrained by a machine. Each machine must act on at least one domain. Furthermore, every requirement must reference at least one domain in the model.

- A constraint in a component can only mention phenomena that belong to or are shared with the component. For example, the constraint in the *ElectionOfficial* domain cannot reference an object of type *BallotImage* of the optical scanner, since the phenomenon is not shared between the two components.

  There are two exceptions to this rule. The first exception are *global phenomena*, which are phenomena that are globally visible to all components. Examples of such phenomena include abstract concepts like integers, and the names of candidates that are running in the election. Secondly, a constraint can refer to any phenomena that are *reachable* through another phenomenon that belongs to or is shared with its component. For example, the domain *Voters* contain phenomena of type *Voter*; a constraint in this domain can mention an object of type *MBallot*, since *Voter* is reachable through the relation *markedBy* from *MBallot*.

With a simple augmentation (e.g. an annotation on phenomena to indicate their sharing), it should be straightforward to perform automatic syntactic checks to ensure that a model satisfies these conditions.

2.5 Summary of the Chapter

In this chapter, we introduced the main ingredients of our modeling notation, as follows:

- The basic building blocks of our notation are *phenomena*, *domains*, *machines*, and *requirements*.

- Two types of phenomena exist: *individuals* and *relations*.

- Domains and machines interact with each other through *shared* phenomena. Other phenomena are *internal* to a domain or machine, and not visible to external components.

- A machine or domain A *observes* another domain or machine B by querying the phenomena that it shares with B, or *constrains* B by modifying the phenomena. A requirement *references* one or more domains that it is concerned with.

- Each domain, machine, and requirement is associated with a constraint that it must satisfy. A domain is associated with a set of assumptions about its behavior or properties; a machine with a specification that it must fulfill; and a requirement with a description of the desired state of the world. Each component is classified into either *Satisfied* or *Unsatisfied*, depending on whether or not it satisfies its constraint.

We also provided a formalization of our notation in Alloy. In the following chapter, we discuss how we build on this formalization to carry out different types of dependability analysis on a model.

Chapter 3

Analysis

The Alloy Analyzer provides three different kinds of analysis: (1) property checking, which attempts to generate a counterexample that shows the violation of a property, (2) simulation, which generates valid instances of system configurations, and (3) unsatisfiable core extraction, which identifies key portions of the model that contribute to the satisfaction of a property. In this section, we describe how each one of these analysis techniques can aid the analyst in assessing the dependability of a system and establishing a trusted base.

3.1 Checking the Consistency of a Satisfaction Argument

An argument for the satisfaction of a requirement R is in the form:

$$D, S \vdash R \tag{3.1}$$

In Alloy, a generic satisfaction argument for *all* requirements can be formulated as an assertion:

```
assert {
  Domain in Satisfied && Machine in Satisfied => Requirement in Satisfied
}
```

That is, if every domain (i.e. voters, the check-in desk, the election official, etc.) behaves as expected, and every machine (i.e. the optical scanner) is correctly implemented, then every requirement should hold. This argument can be specialized for a particular requirement (*AllCastTallied*, for example):

```
assert {
  Domain in Satisfied && Machine in Satisfied => AllCastTallied in Satisfied
}
```

The assertion can be checked automatically by the Alloy Analyzer, which attempts to find a counterexample, if any, that invalidates the assertion. A counterexample

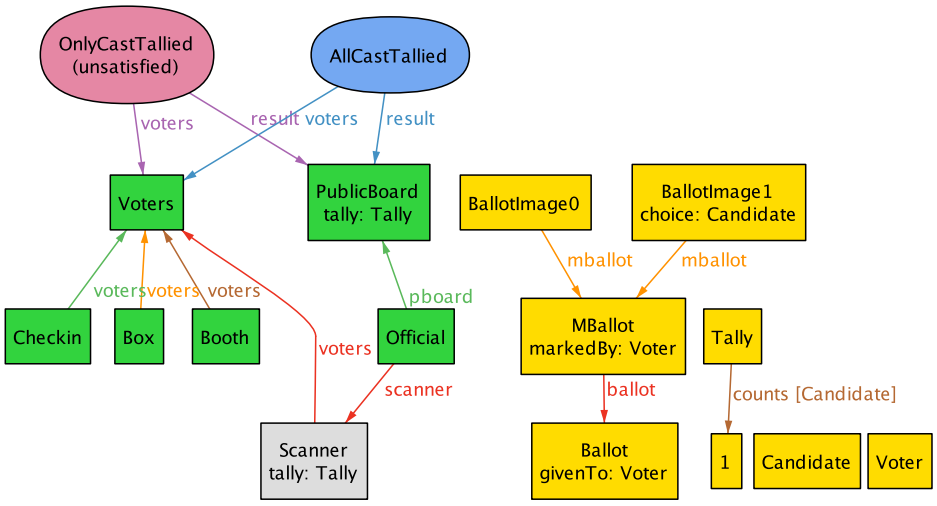


Figure 3-1: A counterexample to *OnlyCastTallied*

suggests that one or more domain assumptions or machine specifications need to be strengthened, or that the requirement is perhaps too strongly stated. The Alloy Analyzer reports no counterexamples for this assertion up to a bound.

The requirement *OnlyCastTallied* can also be checked using a similar formulation:

```
assert {
  Domain in Satisfied && Machine in Satisfied => OnlyCastTallied in Satisfied
}
```

This time, the Alloy Analyzer returns a counterexample (Fig. 3-1). It represents a scenario in which the total tally for the only running candidate is greater than the number of voters who intended to vote for the candidate (which happens to be zero, in this scenario). A close look at the counterexample shows that the scanner holds more ballot images than the number of ballots cast! It turns out that the first part of the specification for the scanner is too weak; it allows multiple records to be created for each marked ballot. As a fix, modifying the specification with a stronger quantifier *one* ensures that the scanner contains exactly one electronic record per marked ballot:

```
sig BallotImage {
  choice : lone Candidate,
  mballot : MBallot
}
one sig OpticalScanner extends Machine {
  voters : Voters,
  records : set BallotImage,
  ...
}{
    this in Satisfied
  iff
    (all mb : markedBy.(voters.members) |
```

```
      let rec = (records <: mballot).mb |
        one rec &&
        rec.choice = mb.choice)
  ...
}
```

When the Alloy Analyzer is asked to re-check the assertion, it no longer reports any counterexample.

3.2 Establishing Trusted Bases

A *satisfaction base* for a requirement is any set of domains and machines that together satisfy the requirement. Formally, let $\mathcal{R}$ be the set of all requirements, and $\mathcal{A}$ be the powerset of domains and machines. Then, we define a function $SB : \mathcal{R} \rightarrow \mathcal{P}(\mathcal{A})$ such that for every $R \in \mathcal{R}$ and $A \in \mathcal{A}$, A is a member of $SB(R)$ if and only if the following formula holds:

$$(\forall a : A \cdot a \in Satisfied) \Rightarrow R \in Satisfied$$

That is, A is a member of $SB(R)$ if and only if the domains and machines in A together establish R; we say that A is a satisfaction base for R. Then, SB is a function that associates each requirement with the set of all satisfaction bases for the requirement. Note that the set of all domains and machines in the system is a trivial satisfaction base for every valid requirement.

We say that a satisfaction base $S \in SB(R)$ for a requirement R is *minimal* if no proper subset of S is a satisfaction base for R. In other words, if any element in a minimal satisfaction base fails to satisfy its constraint, then the requirement may no longer hold.

A *trusted base* for a requirement is the minimal satisfaction base that is responsible for establishing the requirement. Formally, let us define a function $TB : \mathcal{R} \rightarrow \mathcal{A}$ such that for every $R \in \mathcal{R}$, $TB(R) \in SB(R)$ and $TB(R)$ is minimal. Note that the system may contain multiple minimal satisfaction bases for R, and thus, multiple potential candidates for $TB(R)$. In this case, it is up to the analyst to choose one that he or she evaluates to be the most trustworthy, and establish it as *the* trusted base for the requirement. We discuss a set of criteria for selecting one out of possible trusted bases in Section 3.2.2.

The task of the requirements analyst in establishing a trusted base is twofold: identification (Section 3.2.1) and evaluation (Section 3.2.3).

3.2.1 Identification of a Trusted Base

The analyst must first study the initial structure of the system and correctly identify a trusted base $TB(R)$ for each critical requirement R. A simple, naive method to find the trusted base exists. Begin by considering the set of all domains and machines, which forms a satisfaction base for R. Then, enumerate every proper subset of the base until a minimal satisfaction base is found. Although this method is sound, it

is also inefficient, since there are potentially an exponential number of subsets to explore.

We reduce the identification of a trusted base to the problem of finding *unsatisfiable cores* (unsat core), which has been studied extensively in the field of constraint satisfaction problems. Let S be a set of logical constraints that when conjoined together form an unsatisfiable formula. Then, an unsat core of S is a subset $T \subseteq S$ that is itself unsatisfiable. An unsat core T is *minimal* if removing any of the constraints in T results in the remainder of the core becoming satisfiable.

The Alloy Analyzer is a constraint solver; given a set of constraints, it attempts to satisfy the formula that corresponds to their conjunction. In particular, when asked to check an assertion R, the analyzer conjoins the system specification S with $\neg R$. Any solution to $S \wedge \neg R$ is a counterexample that violates the the assertion R.

The Alloy Analyzer is also equipped with an efficient algorithm for finding a minimal unsat core of a formula [43]. If $S \wedge \neg R$ turns out to be unsatisfiable, the analyzer returns a minimal unsat core. The core contains a constraint set $T \subseteq S$, which corresponds to a particular group G of domain assumptions and machine specifications in the system. By definition, if any constraint is removed from T, then the remainder of the core, $T' \wedge \neg P$, becomes satisfiable. In other words, if any domain or machine in G fails to satisfy its constraint, then the analyzer will find a counterexample to $S \wedge \neg R$. Then, it follows that G is a minimal satisfaction base for the requirement R. The analyst then may designate G as the trusted base for R.

Consider the optical scan voting system. When the analyst asks the Alloy Analyzer to check the assertion

```
assert {
  Domain in Satisfied && Machine in Satisfied => AllCastTallied in Satisfied
}
```

it finds no counterexample, and reports a minimal unsat core, which highlights the constraints for the following domains and machine:

```
{Voters, ElectionOfficial, OpticalScanner}
```

Given this information, we can define an Alloy function that associates a requirement with the set of domains and machines that form its trusted base:

```
fun TB : Requirement -> set (Domain + Machine) {
  AllCastTallied -> {Voters + ElectionOfficial + OpticalScanner}
}
```

Now, the analyst may rephrase the satisfaction argument for *AllCastTallied* and check it using the Alloy Analyzer:

```
assert {
  TB(AllCastTallied) in Satisfied => AllCastTallied in Satisfied
}
```

That is, regardless of how non-trusted domains or machines behave, the requirement should hold as long as the trusted components satisfy their constraints.

But $TB(AllCastTallied)$ should look suspicious, because it does not include components that a domain expert would expect to find. According to this trusted base, the requirement holds if each voter casts a ballot as intended, the optical scanner correctly records the ballots and computes the final tally, and the election official accurately reports the result. But it seems reasonable to expect that the requirement should also depend on the following constraints that are imposed by the check-in desk as well as the voting booth:

```
one sig Checkin extends Domain {
  voters : Voters,
}{
    this in Satisfied
  iff
    // each voter is given only a single ballot
    all v : voters.members |
      one givenTo.v
}
one sig Booth extends Domain {
  voters : Voters
}{
    this in Satisfied
  iff
    // the ballot is marked as the voter goes through the booth
    all v : voters.members |
        some givenTo.v
      =>
        some mb : MBallot |
          mb.markedBy = v and mb.ballot in givenTo.v
}
```

For example, if a voter is wrongfully denied a ballot at the check-in or allowed to secretly sneak out a ballot without marking it at the booth (to be handed off to another voter), then this may lead to situations in which *AllCastTallied* no longer holds.

A close look at the *Voters* domain reveals that its assumption is too strongly stated as it is:

```
one sig Voters {
  members : set Voter
}{ // signature fact
  this in Satisfied iff
  all v : members | (markedBy.v).choice = v.intent
}
```

Namely, this constraint forces the existence of a marked ballot for every voter, essentially rendering the constraints on *Checkin* and *Booth* redundant. This constraint can be weakened to reflect a more realistic assumption:

```
one sig Voters {
```

```
  members : set Voter
}{ // signature fact
  this in Satisfied iff
  all v : members, mb : MBallot |
    mb.markedBy = v => mb.choice = v.intent
}
```

The modified constraint says that, *if* a ballot is marked by the voter, then the choice on the ballot must match the intent of the voter. As a result of this weakening, showing *AllCastTallied* now requires the domain assumptions on *Checkin* (i.e. the voter picks up exactly one ballot from the check-in desk) and *Booth* (i.e. the voter marks the ballot at the booth). When the user asks the Alloy Analyzer to check *AllCastTallied* again, it reports an unsat core that can be used to derive the following trusted base:

```
TB(AllCastTallied) =
  {Voters, Checkin, Booth, ElectionOfficial, OpticalScanner}
```

This example illustrates the risk of *over-constraining* domain assumptions due to a specification error. The analyst would have overlooked the two domains *Checkin* and *Booth* as being inconsequential to the requirement. This is potentially dangerous, since it means that less attention will be paid to discharging the assumptions about these domains, even when they are, in fact, critical parts of the system.

The Alloy Analyzer reports no counterexample for *OnlyCastTallied*, and computes an unsat core that corresponds to the following trusted base, which turns out to be the same as the one for *AllCastTallied*:

```
TB(OnlyCastTallied) =
  {Voters, Checkin, Booth, ElectionOfficial, OpticalScanner}
```

Our notion of a trusted base differs in two ways from the notion of a *trusted computing base* (TCB) in the security literature, where each system contains exactly *one* TCB that is responsible for ensuring all critical properties:

1. In our approach, different requirements may have different trusted bases. For example, in an online shopping system such as Amazon, the requirement that the catalog displays the correct prices of items is likely to have a different trusted base than the one for the requirement about secure credit card handling.

2. Each requirement may have multiple trusted bases. For example, a mission-critical system may be equipped with a redundancy feature (e.g. N-version programming [3]) that ensures that a safety requirement holds, even if some of its main components fail to operate normally. In such a system, the requirement may be fulfilled by multiple sets of components, some of which may be trusted more than the others. We explain this in more detail in the following section.

3.2.2 Criteria for Selection of a Trusted Base

What happens when the system contains more than one minimal satisfaction base for a requirement (i.e. multiple potential candidates for the trusted base)? This implies that there are different ways in which the particular requirement can be fulfilled by the system. For instance, systems with fault-tolerance mechanisms often exhibit this characteristic. Such a system is designed to withstand the failure of a critical component and avoid catastrophes; this is often achieved using redundant components that "kick in" when some component fails.

As a simple example, consider a system with a requirement R, which is fulfilled, in the normal operation mode, by two machine components, C_1 and C_2. When the analyst checks an assertion

```
assert {
  Domain in Satisfied && Machine in Satisfied => R in Satisfied
}
```

the Alloy Analyzer returns an unsat core that represents a minimal satisfaction base $SB_1 = \{C_1, C_2\}$.

Let us also assume that the system is equipped with backup components $\{B_1, B_2\}$ as replacements for $\{C_1, C_2\}$. In addition, an arbiter A controls the backups to gracefully take over the operation when either one of $\{C_1, C_2\}$ fails. The analyst allows one of $\{C_1, C_2\}$ to behave arbitrarily (C_1, in this case), and checks whether the requirement is still satisfied:

```
assert {
  Domain in Satisfied && (Machine - C1) in Satisfied => R in Satisfied
}
```

The Alloy Analyzer reports that R still holds, and returns a minimal satisfaction base $SB_2 = \{A, B_1, B_2\}$.

So which one of SB_1 and SB_2 qualifies as the trusted base for R? In this particular scenario, a reasonable argument would be that SB_2 should be selected as the trusted base, since by their designation as backup components, a failure in SB_2 has a greater implication on the satisfaction of R than a failure in SB_1.

In general, give multiple minimal satisfaction bases, there may no definitive method to determine which one should be chosen as the trusted base. It is up to the analyst to make an informed decision after examining individual components in the bases. This is a research question that we plan to explore further in future. For now, we briefly discuss a set of criteria that the analyst may use to construct a "weighting" function that ranks the bases in terms of their desirability:

- **Size of the base (i.e. number of components)**: Larger the size of the base is, less desirable it is, since more effort must be spent on ensuring that every component in the trusted base is reliable.

- **Complexity of components**: Machine with complex specifications, such as an electronic voting machine, are difficult to check or verify for correctness.

Similarly, domains with a large number of assumptions complicate the task of the domain expert, and should be excluded from the trusted base, if possible.

- **Inherent characteristics of components**: Certain types of machines or domains may be considered, by nature, to be less reliable than others. For example, software is generally considered to be more complex and less reliable than hardware or mechanical parts. Similarly, in certain applications (such as train control systems), some experts argue that human operators are unreliable, and should be replaced with a computerized component.

Note that these criteria are only *soft* guidelines, and should be taken with caution. The insights of the analyst, and the particular application domain that the system belongs to, are factors that carry far more weight in deciding which components should be trusted with ensuring the dependability of the system.

3.2.3 Evaluation of a Trusted Base

Having identified a trusted base, the requirements analyst must show that the existing base is a reliable one by producing informal arguments to justify that each domain or machine in the base can be trusted with satisfying its assumption or specification. A domain expert examines each domain assumption to ensure that it is a reasonable (neither too weak nor too strong) characterization of the reality. For each machine specification, a software architect considers given design constraints (e.g. available hardware, platforms, costs, development time, etc) and decides whether the machine can be implemented to a high degree of confidence in its correctness. In particular, if the machine is liable for a highly critical requirement, then the architect must devise a validation plan, which may consists of testing, verification, and auditing techniques, to demonstrate the conformance of the machine against its specification. If no such viable plan exists, the specification may be too ambitious to achieve; the analyst should then consider the machine for an exclusion from the trusted base.

Let us consider the voting example again. We previously defined the trusted bases for the requirements *AllCastTallied* and *OnlyCastTallied* as follows:

TB(AllCastTallied) = TB(OnlyCastTallied) =
{Voters, Checkin, Booth, ElectionOfficial, OpticalScanner}

Evaluating the reliability of the trusted bases involves examining the assumptions and specifications of individual domains and machines in the bases:

- **Voters**: Each voter is assumed to cast the ballot exactly as intended. If the voter is coerced into voting for a particular candidate, then the integrity of the election may come under attack.

- **Check-in desk**: Poll workers at the checkin desk must be trusted to hand out exactly one ballot to each voter. The violation of this assumption may lead to ballot stuffing. The domain expert may argue that this is a reasonable assumption to make, since the desk is usually located in an open, visible area, and a large-scale attempt at mischief would likely be detected.

- **Booth**: Each ballot, given to a voter at the checkin desk, is marked as it passes through the voting booth. The voter must be prevented from walking out of the poll center with an incomplete ballot, which may then be handed off to another voter. By staffing the area around the booth with poll workers, it should be possible to enforce this constraint.

- **Election official**: The election official must be trusted to report the result to the central election board exactly as it is tallied by the optical scanner. The fidelity of this assumption necessarily stands on a social ground; the voters must trust that the officials will not act in favor of a particular candidate or a political party.

- **Optical scanner**: A software architect (one from a third-party contractor, such as Diebold, for example) must demonstrate that it is feasible to construct a highly reliable optical scanner that stores the voters' choices and computes total tallies. But it turns out this task is far from trivial. Various studies showed that existing optical scanners are vulnerable to a wide range of security attacks [5, 11, 23]. Unfortunately, poor software engineering is often employed in the development process of such a system. Further compounding the problem is the fact that software in most optical scanners is proprietary, and so it is not subject to public inspection, which could help uncover bugs and security flaws.

A number of prominent security experts have argued that due to the complex nature of software, it would be extremely difficult to achieve a high level of confidence in a purely software-based voting mechanism, even using the most advanced quality assurance techniques. They have proposed the notion of *software independence* [34], which states that the integrity of the election should not hinge on the correctness of a software component, such as the optical scanner. In Chapter 5, we describe a different kind of voting systems that achieves this property.

3.3 Generating Failure Scenarios

Having established a trusted base, the analyst might ask: What happens when a domain or a machine in this base behaves unexpectedly? Examining various failure scenarios is useful for understanding system vulnerabilities, and can provide suggestions for adding extra security or safety mechanisms. But generating these manually is often tedious and error-prone.

We describe how the analyst can use our framework to systematically generate failure scenarios. The idea is simple: The analyst uses the meta-objects *Satisfied* and *Unsatisfied* to force one or more components into undesirable states (i.e. behaving badly), and observes the consequence on the system requirements.

In the simplest case, the analyst can execute a *run* command in the Alloy Analyzer to generate instances of the system configurations in which *some* requirement does not hold:

```
run {
```

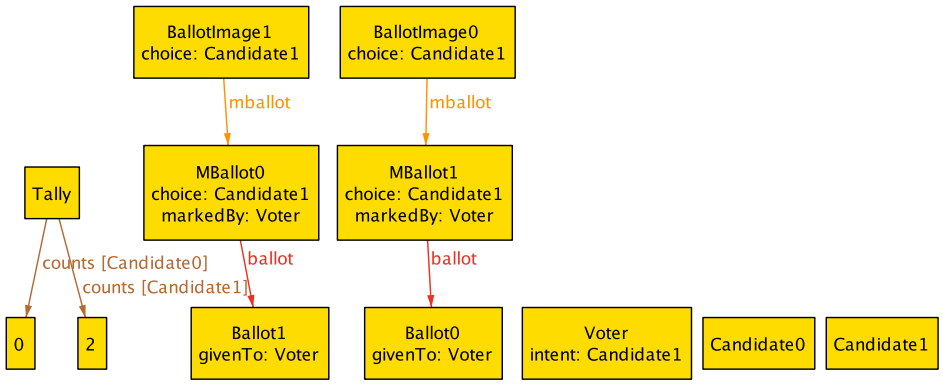


Figure 3-2: A scenario for the failure of *Checkin*

```
  some (Requirement & Unsatisfied)
}
```

By leaving the memberships of domains and machines unspecified, this command can generate a scenario with any combination of component failures. But this may not be the most systematic use of the meta-objects. Let us suppose that the analyst wishes to examine single-component failures—in particular, the failing component should be a domain. To accomplish this, we first define a function that returns the set of all unsatisfying domains in the world:

```
fun badDomains : set Domain {
  Domain & Unsatisfied
}
```

Then, the following *run* command generates a single-domain failure for a particular requirement (*OnlyCastTallied*, in this case):

```
run {
  one badDomains
  Machine in Satisfied
  OnlyCastTallied in Unsatisfied
}
```

The analyst now wants even more control, asking for a scenario in which only the domain *Checkin* fails to satisfy its constraint:

```
run {
  badDomains = Checkin
  Machine in Satisfied
  OnlyCastTallied in Unsatisfied
}
```

Running the command generates the scenario in Figure 3-2, where a voter is given two ballots at the check-in desk, thus giving an extra vote to a candidate. A similar

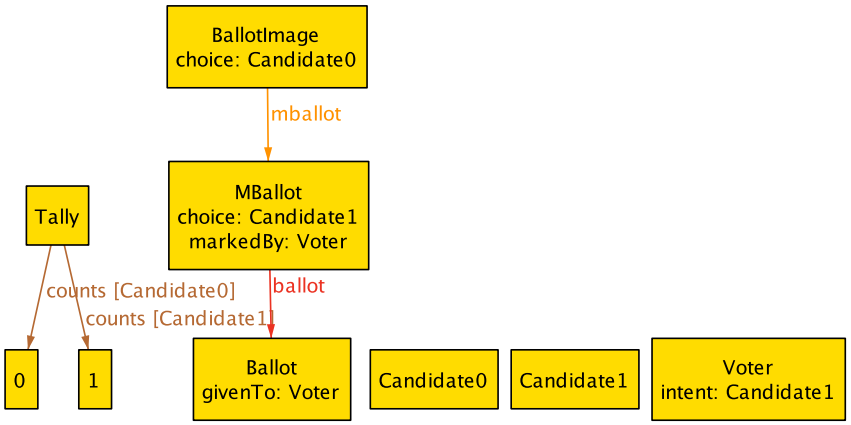


Figure 3-3: A scenario for the latent failure of *OpticalScanner*

style of *run* commands can be used also to generate failures that involve multiple components.

Just as interesting are scenarios in which the requirements *do* hold, but one or more components have failed to satisfy their constraints. For instance, the following command generates such a scenario in which some machines violate their specifications (in this case, the optical scanner), but the requirements still happen to be true:

```
fun badMachines : set Domain {
  Machine & Unsatisfied
}
run {
  some badMachines
  Domain in Satisfied
  Requirement in Satisfied
}
```

In the scenario shown in Figure 3-3, the optical scanner is clearly violating its specification, since it is incorrectly recording the voter's choice (*Candidate0* instead of *Candidate1*) into the ballot image. But the final tally turns out to be correct anyway, because the part that computes the tally is broken as well. If the machine is left in its current state, then it will very likely cause more visible failures in another election.

These are examples of *latent* failures. They require just as much of the analyst's attention, because they have the potential to lead to catastrophic failures of the system.

3.4 Summary of the Chapter

In this chapter, we described three different types of analysis within the context of the Alloy Analyzer:

- **Consistency checking**: An argument $D, M \vdash R$ is checked for its consistency

to ensure that if all domains behave as expected, and all machines are correctly implemented, then the desired requirement holds true.

- **Detection of a trusted base**: A technique for an unsat core detection is used to extract a minimal subset of domains and machines that establish the requirement. If multiple such bases exist, then the analyst designates one of them as the *trusted base* for the requirement.

- **Failure generation**: One or more of domains or machines are allowed to behave unexpectedly, and the analyst examines the consequences of their failures on the overall system.

We also proposed that the analyst must justify the reliability of a trusted base by discharging domain assumptions and providing concrete evidence that machines can be implemented to satisfy their specifications with a high degree of confidence. We showed how our approach may be used to argue that an optical scan voting system cannot be considered dependable, due to security vulnerabilities in a typical optical scanner. We describe a different kind of voting system—one whose integrity does not rely solely on the correctness of software—in Chapter 5.

Chapter 4

Property-Part Diagram

4.1 Introduction

A *property-part* diagram [16] is a notation for expressing dependences between parts of the system, and a predecessor to our current work in this thesis. Also inspired by the Problem Frames approach, it assigns properties to each part of the system, and shows how the parts together satisfy a top-level requirement. However, unlike traditional dependence diagrams, a relationship is expressed not between parts, but from a property P to a combination of parts and other properties that support P. Our goal back then was the same as now: to develop a notation to articulate a dependability argument, and identify links between a critical requirement and all parts that are responsible for its satisfaction.

As its name may suggest, the property-part notation consists of two basic constructs: *properties* and *parts*. Figure 4-1 shows a property-part diagram for the optical scan voting system. A part, drawn as a box in the diagram, is like a machine or a domain; it may represent a software or hardware component, or a group of phenomena in the environment. A property, drawn as an oval, may refer to a specification of a machine, an assumption about the environment, or a requirement to be fulfilled. In other words, properties are equivalent to constraints in our current notation.

A property P is connected to a part M if M must be implemented (in case of a machine) or assumed (a domain) to satisfy P. For example, the optical scanner is required to implement two pieces of functionality—recording ballots and computing a tally—and has incoming edges from the two ovals (nodes D and E in Figure 4-1) that represent these properties. Similarly, the analyst must discharge the assumption that the election official can be trusted to report the outcome of the election exactly as it is computed by the scanner (C).

A property P is connected to another property Q if the satisfaction of P depends in part on the satisfaction of Q. For example, the scanner may not compute a correct tally if it fails to accurately record the ballots into electronic images; hence, the diagram shows an edge between these two properties (B and D).

The *exposure* of a property P is the set of all parts that are reachable from P through the edges. These parts are responsible for establishing P; if any of the parts

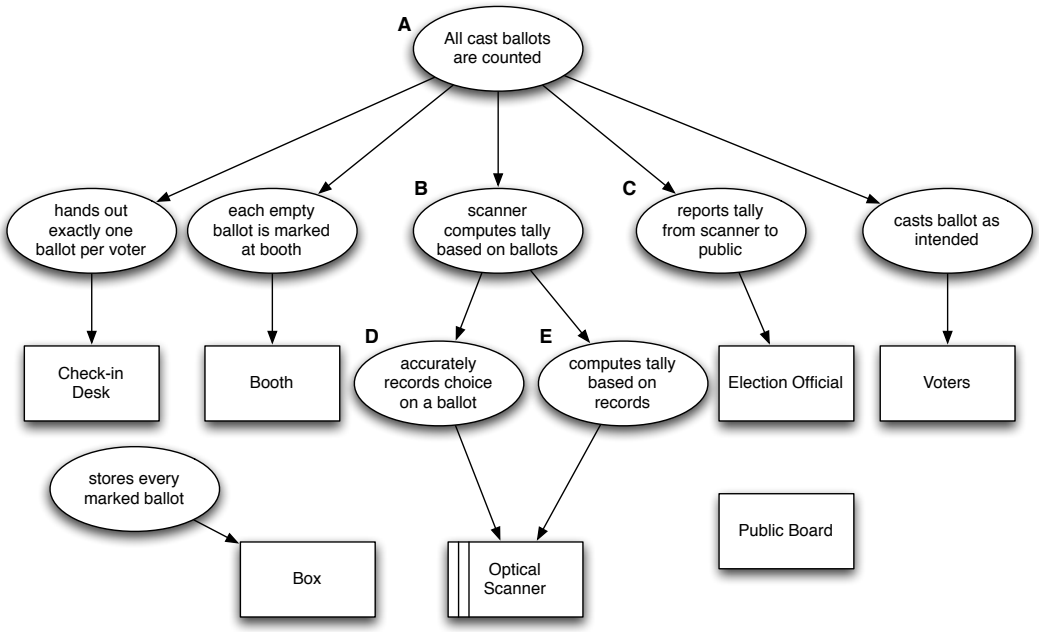


Figure 4-1: A property-part diagram for an optical scan voting system

fails to satisfy its property, then P may no longer hold. For example, the exposure of the requirement that all cast ballots are included in the final tally (A) amounts to the following set of parts:

$$\{CheckinDesk, Booth, OpticalScanner, ElectionOfficial, Voters\}$$

Note that this set is equal to the trusted base for the requirement, which we discussed in Section 3.2.1. The parts Box and $PublicBoard$ do not belong to this set, because their properties do not affect the requirement. However, they may belong to the exposure of another property; for instance, if we were to model a requirement for election recount, then Box would now belong to the exposure of this requirement, because it must securely store all physical ballots for an accurate recount.

The *argument* for a property P includes the set of all parts *and* properties that are reachable from P. It is an elaboration of a typical satisfaction argument (i.e. $D, S \vdash R$), since it shows not only those components that are critical to the property, but also illustrates how the property is satisfied by the components. In the voting example, the argument for "all cast ballots are counted" is rooted at node A. In this case, the argument turns out to be the entire graph, which implies that a large portion of the system is responsible for satisfying the requirement. One of the analyst's tasks is to examine the argument and attempt to reduce its size through the process of redesign.

4.2 Comparison with a Problem Diagram

For comparison, Figure 4-2 shows the optical voting system model expressed in the problem diagram and property-part diagram. We point out the following differences between the two notations:

- In the problem diagram, a property (or a requirement) is connected to only those domains that it references, whereas in the property-part diagram, the property is connected to every immediate member of its trusted base. The extent of the trusted base can be readily observed in the property-part diagram through simple graph traversal. This information is not immediately visible in the problem diagram, and would require computing the unsat core of the requirement. However, this does not render the unsat core useless, since in order to construct the property-part diagram in the first place, the trusted base would need to be identified anyway.

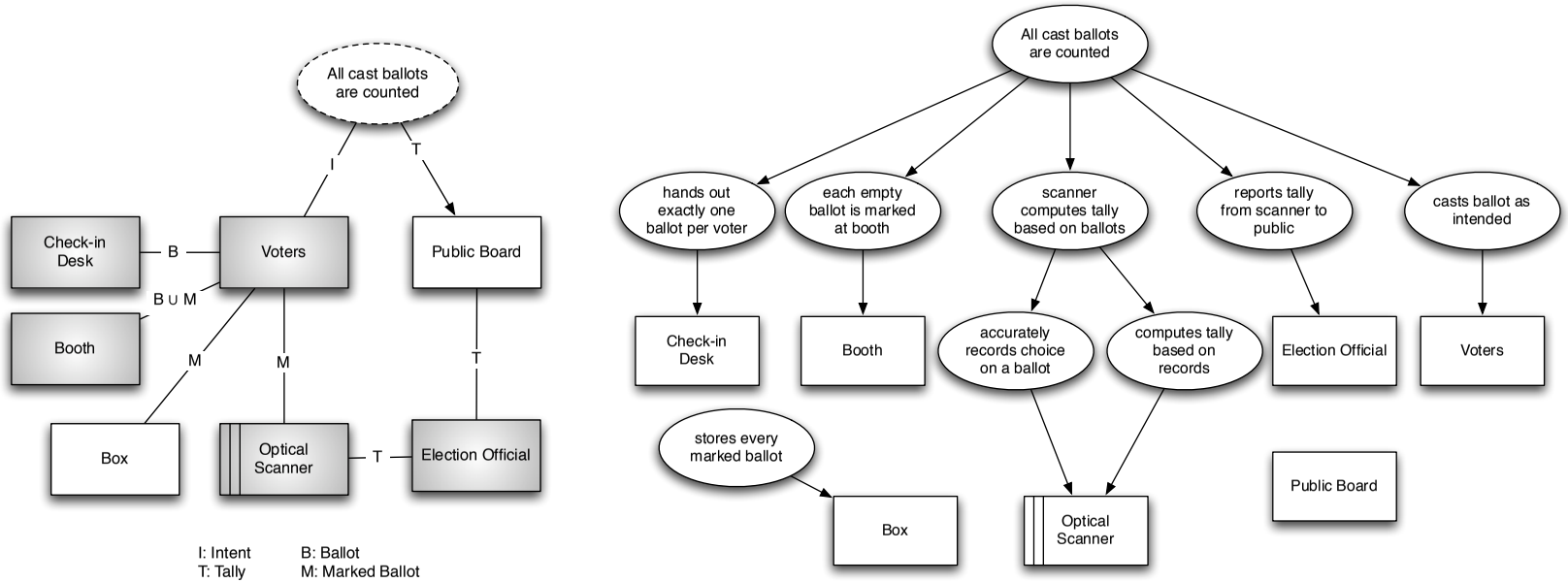
- The property-part diagram deliberately hides direct links between the parts. Therefore, inter-component relationships, such phenomena sharing, or *observes-or-constrains* relations, are shown only in the problem diagram. But again, if necessary, we can infer this information from property-to-part or property-to-property relationships in the property-part diagram.

- The property-part diagram makes explicit not only the requirements as nodes, but the properties (or constraints) of individual components as well. This allows a more fine-grained tracing of properties. For example, let us assume that the customer wants the voting system to satisfy an additional requirement for auditing the election process; as a part of this requirement, the scanner needs to support a functionality to print all electronic ballot records for manual examination. In the property-part diagram, an additional property node would be attached to the scanner to represent the extended specification. However, the problem diagram would simply show the machine as belonging to the trusted bases for both the correctness and audit requirements, even though different properties of the scanner are responsible for them.

In summary, we believe the two types of diagrams are complementary to each other. A problem diagram emphasizes the *structure of the problem context* and highlights relationships between components. On the other hand, a property-part diagram focuses the *structure of dependences* among properties and parts; it deliberately downplays direct relationships between parts, so that they do not interfere with the articulation of the dependence structure.

The analyst would find both types of information useful, and with simple analysis, either one of them could be generated from the other. We envision that in a typical scenario, the analyst would take advantage of these two notations in the following fashion:

1. Begin by laying out the structure of the problem context (i.e. connections between requirements, domains, and parts) in a problem diagram, and systemat-



(a) Problem diagram

(b) Property-part diagram

Figure 4-2: Models of the optical scan system in the two types of diagrams

ically derive domain assumptions and machine specifications, using techniques such as requirement progression [39].

2. Construct a satisfaction argument for a critical requirement, check its consistency, and find the trusted base for the requirement, using the analysis techniques that we described in Section 3.

3. Generate a property-part diagram based on the information in the trusted base, and further examine the dependences between the requirement and different parts of the system.

In the following section, we outline a technique for generating a property-part diagram from a model in our notation.

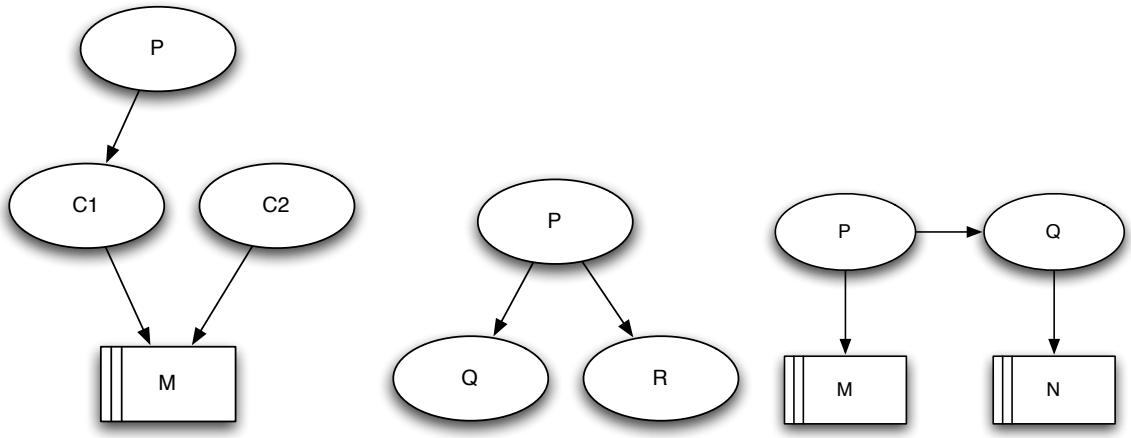
4.3 Generation of a Property-Part Diagram

Our framework does not currently support an automatic generation of a property-part diagram from a model of a dependability argument. Based on the recognition that the trusted base of a requirement is equal to its exposure, a simple model transformation technique should suffice for this task. But this is not without technical obstacles, and we discuss some of them here:

First of all, the way we currently structure constraints in a model allows only for detection of a rather coarse-grained trusted base. As we briefly pointed out in the previous section, when a machine or a domain is assigned multiple constraints, only some of them may be required for the satisfaction of a requirement. But with the current structuring, the unsat core facility does not recognize this fact. For example, consider the following simple model, where the machine M has two constraints associated with it—$C1$ and $C2$. Let us also assume that $C1$, by itself, logically implies the formula P, rendering $C2$ redundant:

```
one sig M extends Machine {
  ...
}{
  this in Satisfied iff C1 && C2
}
one sig R extends Requirement {
  ...
}{
  this in Satisfied iff P
}
assert {
  M in Satisfied => R in Satisfied
}
```

In the Alloy Analyzer, an unsat core includes a list of *top-level* conjuncts in a model. So, the unsat core found after checking the assertion in the above model unhelpfully highlights the entire formula



(a) partial dependence (b) property decomposition (c) contingent use

Figure 4-3: Patterns of dependences in a property-part diagram

```
this in Satisfied iff C1 && C2
```

There is a simple solution to this problem. For the purpose of generating a property-part diagram, we can break the formula into three smaller top-level conjuncts, without changing the behavior of the machine:

```
one sig M extends Machine {
  ...
}{
  this in Satisfied => C1 // formula (1)
  this in Satisfied => C2 // formula (2)
  C1 && C2 => this in Satisfied // formula (3)
}
```

Now, the unsat core highlights only the formula (1)

```
this in Satisfied => C1
```

and we can use this information to infer a dependence edge from the property P to $C1$ in the property-part diagram, as shown in Figure 4-3a. The unsat core does not highlight the formula (3), because the assertion requires only one direction of the double implication formula; that is, if the machine is satisfied, then its constraints must hold true.

Another issue is that our notation currently does not support ways to encode some of the dependence patterns that arise in a property diagram; namely, (1) property decomposition, and (2) contingent use.

Property decomposition, shown in Figure 4-3b, represents a claim that a property P is implied by the conjunction of properties Q and R. In our notation, we represent requirements as referencing domains, but we make no explicit connections among requirements, domain assumptions, and machine specifications. Therefore, it

is unlikely that we will be able to automatically infer dependence structures such as property decomposition from a model, without some guidance from the analyst. As a partial solution, we could introduce the notion of dependencies among requirements by introducing a new signature *CompRequirement* as follows:

```
sig CompRequirement extends Constraint {
  dependences : set Requirement
}{
  this in Satisfied iff dependences in Satisfied
}
```

Contingent use is a dependence structure in which a property P depends on a part M and another property Q, which, in turn, depends on a part N that is used by M (Figure 4-3c). Again, our notation currently does not encode information about contingent uses, and would need to be extended if we were to generate a property-part diagram with such structures. This is trickier to do, but one solution may be to augment each machine with a field *uses*, which represent the set of all components that are used by that machine:

```
sig Machine extends Constraint {
  uses : Machine + Domain
}
```

Recall our definition of a membership in $Satisfied$ for machines: We say that a machine M is in $Satisfied$ if and only if it is *implemented correctly*, not necessarily when it satisfies its specification P. In fact, P is satisfied when M is implemented correctly, *and* every one of its *uses* components satisfies its own specification Q. Then, we can express the contingent use pattern in Figure 4-3c by augmenting the left hand side of the double implication formula with $uses \subseteq Satisfied$, as follows:

```
sig M extends Machine {
  ...
}{
  uses = N
  this in Satisfied && uses in Satisfied iff P
}
sig N extends Machine {
  ...
}{
  this in Satisfied iff Q
}
```

In future, we plan to examine the relationship between our notation and a property-part diagram in more detail, and devise a technique for automatically generating a property-part diagram from a model.

Chapter 5

Case Study: Scantegrity

Scantegrity [2] is a voting system intended to provide a *end-to-end* correctness guarantee. It is deployed as an add-on on top of an existing electronic voting system, and does not require modification of the optical scanner. We describe a model of the Scantegrity system, and argue that its design achieves greater dependability than the optical scan system, by establishing more reliable trusted bases for critical requirements.

5.1 Basic Mechanism of Scantegrity

Figure 5-1 represents a problem diagram for the Scantegrity system. It shows two new domains (*WebBoard* and *Auditor*) and two new machines (*Publisher* and *Tabulator*). The roles of each of these domains and machines is explained in the following paragraphs, along with their relationships with the existing components in the optical scan system. We first describe a set of phenomena that Scantegrity introduces into the world:

```
sig Code {}
sig SBallot extends Ballot { // enhanced Scantegrity ballot
  codes : Candidate -> one Code
}
sig Receipt {
  mballot : MBallot,
  code : lone Code,
  keptBy : Voter
}
fact CodesUniquePerBallot {
  all sb : SBallot |
    all disj : c1, c2 : Candidate | sb.codes[c1] != sb.codes[c2]
}
fact AtMostOneReceipt {
  mballot in Receipt lone -> lone MBallot
}
```

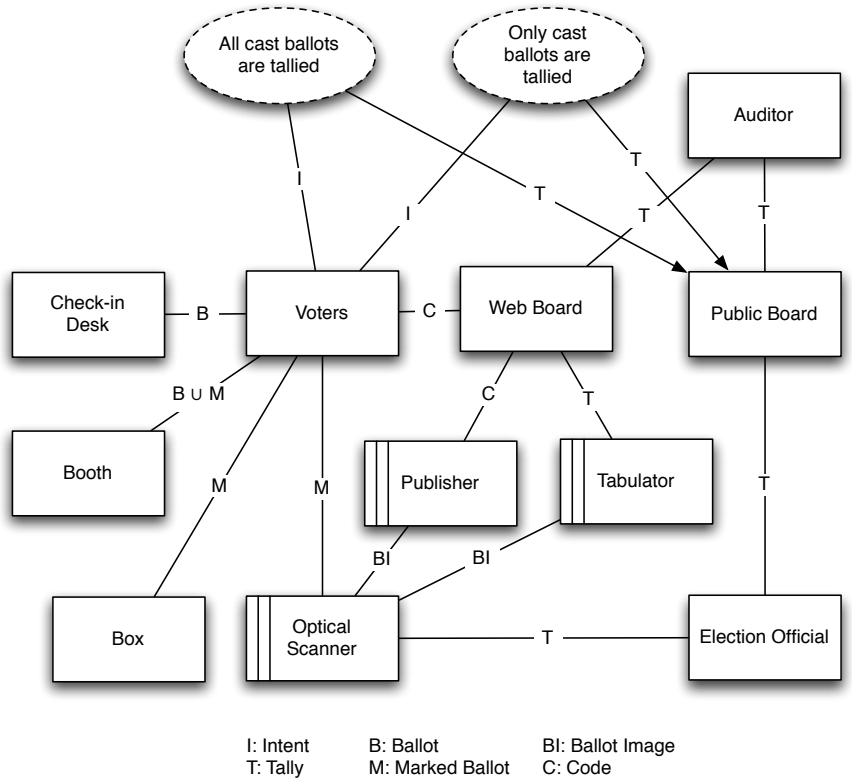


Figure 5-1: A problem diagram for the Scantegrity system

The key idea behind Scantegrity is a ballot *receipt*, which allows a voter to check that his or her vote is included in the final tally. At the check-in desk, each voter is given an enhanced ballot (*SBallot*) that contains a hidden mapping from each candidate to a *code*. Codes on each ballot are unique among themselves, and assigned randomly to candidates. The voter's act of marking a candidate position on the ballot reveals the code that has been assigned to the candidate. The voter is given an option to write this code on a perforated corner of the ballot. After scanning the marked ballot through the optical scanner, the voter may tear off the corner and keep it as a receipt (expressed by the relation *keptBy*).

The following fragment of Alloy shows the new domains and machines, along with their associated constraints:

```
one sig WebBoard extends Domain {
  codes : MBallot -> one Code,
  tally : Tally
}
one sig Publisher extends Machine {
  scanner : Scanner,
  wboard : WebBoard
}{
    this in Satisfied
  iff
```

```
    // publishes the confirmation code for every ballot image
    (all rec : scanner.records |
      let mb = rec.mballot |
        wboard.codes[mb] = mb.ballot.codes[rec.choice])
    &&
    // posts the codes only for the cast ballots
    (all postedBallot : wboard.codes.Code |
      some scanner.records & mballot.postedBallot)
}
one sig Tabulator extends Machine {
  scanner : Scanner,
  wboard : WebBoard,
  // mapping from a code on a ballot to a candidate
  switchboard : SBallot -> Code -> Candidate
}{
    this in Satisfied
  iff
    // a ballot contains correct mappings from codes to candidates
    (all sballot : SBallot |
      sballot.codes = ~(switchboard[sballot]))
    &&
    // independent tallying based on ballot images & codes
    (all c : Candidate |
      let matches =
        { rec : scanner.records |
          let sballot = rec.mballot.ballot |
              switchboard[sballot][sballot.codes[rec.choice]] = c } |
        wboard.tally.counts[c] = #matches)
    &&
    // ensure that there aren't any duplicate records from the scanner
    // e.g. Two ballot images for the same physical ballot
    (mballot.ballot in scanner.records lone -> lone SBallot)
}
```

After the election is over, the optical scanner computes the tally (as in the previous system), and an election official posts the result to a public board. In addition, all of the electronic records inside the optical scanner are transferred to another location and further processed for two purposes. First, a computer (the machine *Publisher*) examines each ballot image and posts the code that corresponds to the marked candidate on the ballot to a publicly available board on the Internet. A voter who decided to kept his or her receipt may check the code on the receipt against the posted one on the web board. If there is any discrepancy, the voter can alert the election authority for a possible tampering of the ballots.

Secondly, a piece of tabulator software (the machine *Translator*), separate from the tabulation component inside the optical scanner, computes the total tally by using a special data structure called a *switchboard*. The switchboard contains a mapping from every (ballot, code) pair to the corresponding candidate; the tabulator takes the

set of the ballot images from the scanner, and derives a vote for a candidate from each marked code. The tally computed from the tabulator is compared against the one from the optical scanner for any discrepancy (which would suggest a tampering of the ballots or an error inside the scanner). The switchboard is publicly available, so third-party auditors can write their own tabulator in order to verify the result.

5.2 Trusted Bases in Scantegrity

The critical requirements for the Scantegrity system remain the same as before: *AllCastTallied* and *OnlyCastTallied*. In fact, since Scantegrity is simply an add-on, it should not affect the correctness of the existing components in the optical scan system. That is, if all domains and machines satisfy their constraints, then these two requirements must hold as well. Not surprisingly, when the Alloy Analyzer checks the satisfaction arguments, it reports no counterexamples, and returns the same trusted bases as before:

```
TB(AllCastTallied) =
   {Voters, Checkin, Booth, ElectionOfficial, OpticalScanner}
TB(OnlyCastTallied) =
   {Voters, Checkin, Booth, ElectionOfficial, OpticalScanner}
```

However, our goal is to take the optical scanner out of the trusted bases. Before we do this, we first describe how Scantegrity achieves this. Recall that the specification of the scanner is twofold: (1) recording a marked ballot into an electronic image, and (2) computing the total tally based on the set of ballot images; the violation of either one of the two can lead to an incorrect election result. Scantegrity provides a mechanism to detect a failure of the scanner in achieving (1) or (2). If Scantegrity detects no such failure, then it ensures that the tally produced by the scanner is indeed correct.

But the mechanism in Scantegrity is no use without participation by the voters as well as third-party auditors in ensuring the election integrity. In particular, each voter is encouraged to check the code on the receipt against the published one on the web board. If the scanner fails to faithfully carry out (1), then one or more voter will discover that the two codes do not match.

```
one sig Voters {
  members : set Voter,
  wboard : WebBoard
}{
    this in Satisfied
  iff
    // every voter casts his/her ballot as intended
    (all v : members | (markedBy.v).choice = v.intent)
    &&
    // every voter checks the receipt
    (all v : members, receipt : keptBy.v |
      wboard.codes[receipt.mballot] = receipt.code)
```

```
}
```

In addition, a third-party auditor can compare the result on the public board (computed by the scanner and posted by the election official) with the one on the web (independently computed by *Tabulator*). If the scanner violates (2), then the two results will not match, prompting the election authority into an investigation.

```
one sig Auditor {
  pboard : PublicBoard,
  wboard : WebBoard
}{
    this in Satisfied
  iff
    // Auditor checks whether two independent tallies match
    all c : Candidate |
      pboard.tally.counts[c] = wboard.tally.counts[c]
}
```

Assuming that the voters and the auditor fulfill their responsibilities, we modify the original satisfaction argument to allow the scanner to behave in any fashion:

```
assert {
  Domain in Satisfied && (Machine − Scanner) in Satisfied =>
  AllCastTallied in Satisfied
}
```

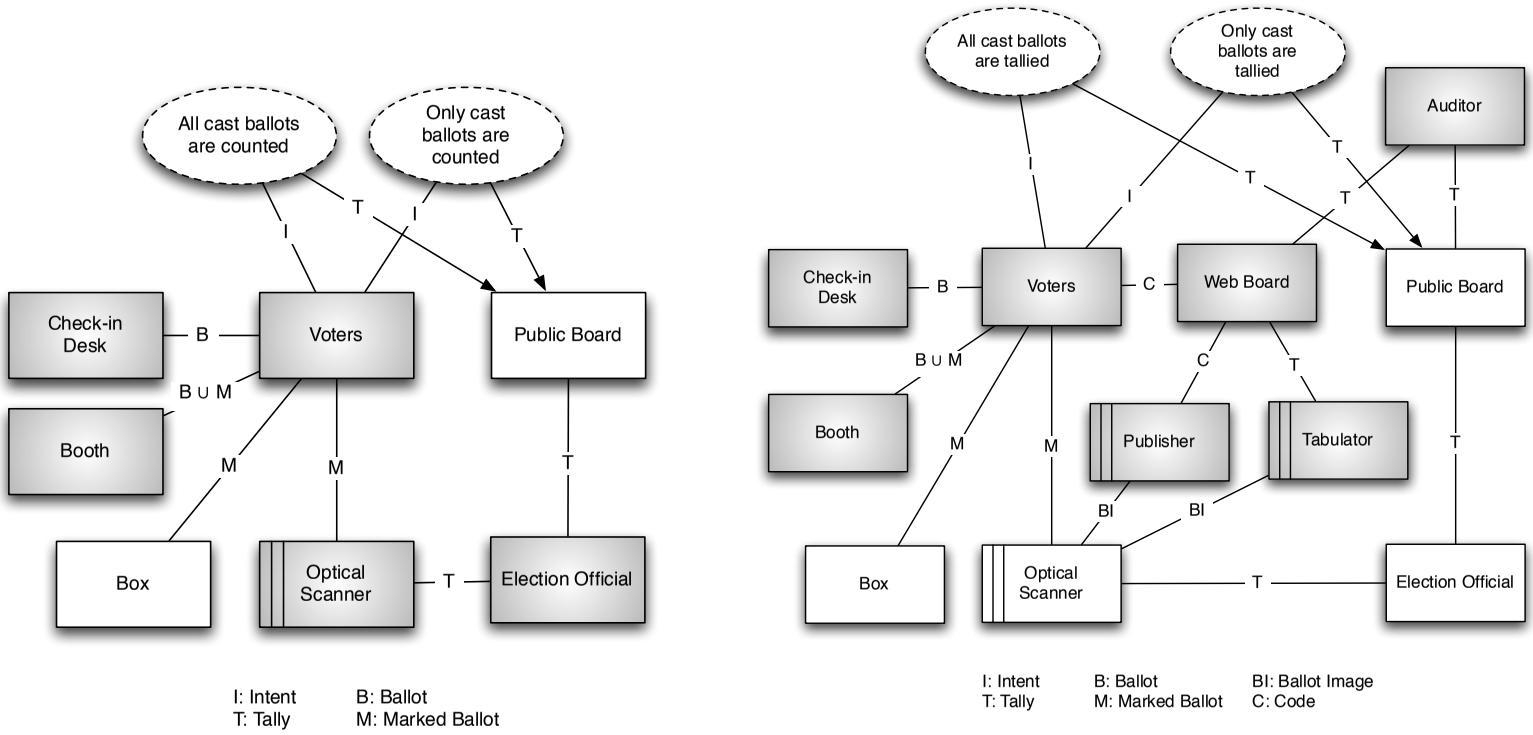
The Alloy Analyzer reports that the argument holds, and derives the following trusted base:

```
TB(AllCastTallied) =
   {Voters, Auditor, Checkin, Booth, Publisher, Tabulator}
```

Note that the requirement no longer hinges on the correct behaviors of the scanner and the election official. Instead it now depends on the publisher, which must correctly posts the codes for the voters to check on the web, and the tabulator, which must correctly compute an independent tally (to be compared by the auditor against the official tally). As previously mentioned, the voters and the auditor must each do their parts to detect any mishap in the election.

5.3 Comparison of Scantegrity and the Optical Scan System

Figure 5-2 highlights the members of the trusted bases (shaded components) in the two voting systems. A comparison between the two raises several interesting points. Foremost of all, the Scantegrity system is considerably more complex than the optical scan system. But this is no surprise. A system where dependability is a top priority requirement (e.g. the voting systems) incurs more investment in safety or security mechanisms than would normally be the case for a non-critical software system (e.g.



(a) optical scan

(b) Scantegrity

Figure 5-2: Trusted bases in the two electronic voting systems

a vending machine or a word processor). But this does not imply that complexity should necessarily be predominant throughout the system. Rather, the designer must strive to keep critical components as simple as possible so that they are easier to inspect for correctness.

For more detailed comparison, the property-part diagrams in Figure 5-3 illustrate the dependence structures for the requirement *AllCastTallied* in the two systems. A superficial attempt at comparison may count the number of parts and property nodes in the argument for the requirement, and conclude that the optical scanner is more reliable, because it has a smaller trusted base. But this, by itself, does not form a convincing dependability case for the optical scan system. In fact, the designers of the Scantegrity would argue the opposite way, using informal justifications (discussed in [2]) for why components in the trusted base can be reasonably trusted, while the others should be considered untrustworthy:

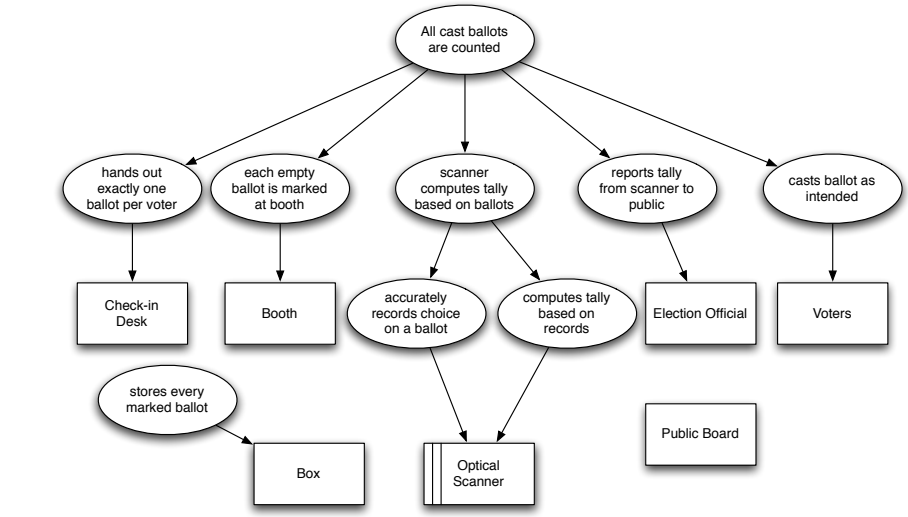
- **Voters**: We assume that voters are good citizens with an interest in maintaining the integrity of the election. Of course, in reality, this assumption would be too strong to make for every voter. In fact, Scantegrity assumes that only a subset of the voters will check their receipts on the web; as a result, it provides only a probabilistic guarantee (but nevertheless of a high degree) of the correctness. For simplicity, we deliberately omitted modeling of the probabilistic aspects of the voting system.

- **Auditor**: Scantegrity assumes that the third-party auditor is a politically neutral party with the sole purpose of observing the election process for any sign of fraud or error. In principle, anyone can act as an auditor, and there are often multiple auditing parties for each election, so even if some of them decide to conspire with a candidate, this should not undermine the outcome of the audit.

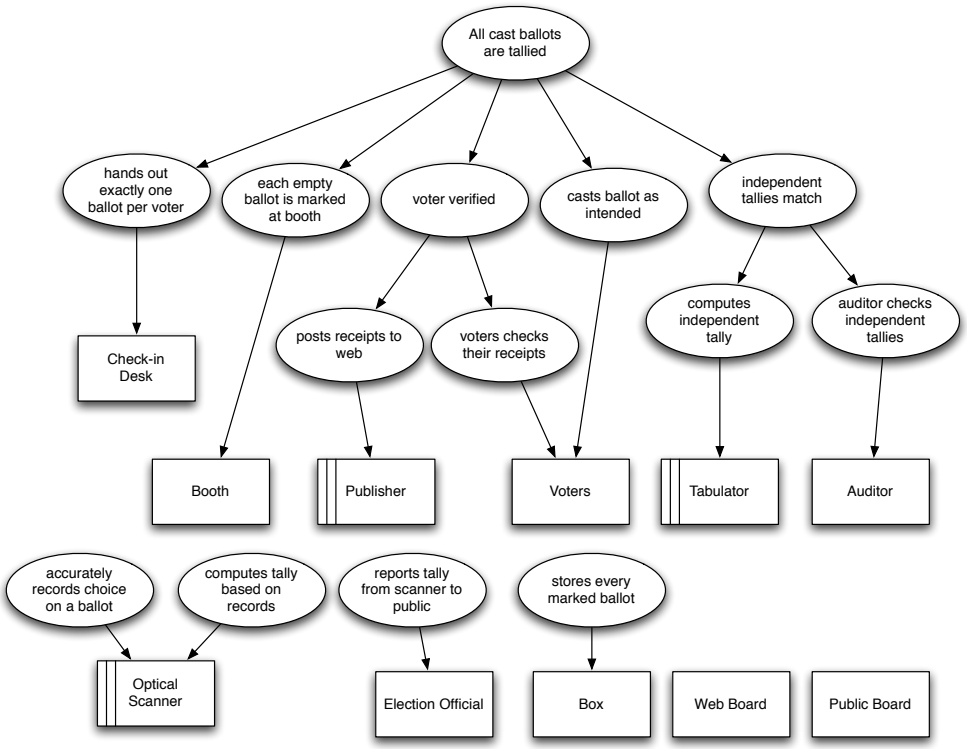
- **Publisher**: The machine carries out a much simpler task than the optical scanner does, and so it may be amenable to rigorous testing and verification techniques. In addition, the publishing software is open source, and can be inspected by the public for bugs and security flaws (unlike the software in the scanner, which is often proprietary).

- **Tabulator**: The switchboard is publicly available for deriving votes from a set of ballot images. Thus, anyone can, and is encouraged to, write their own tabulator to compute an independent tally. This open nature of the independent tallying process makes it highly unlikely that a malicious party can sabotage this component.

- **Check-in desk and voting booth**: Each of these two locations is staffed by multiple poll workers. Although it is possible that a malicious poll worker may intentionally hand out an extra ballot, or allow a voter to sneak out an unmarked ballot, it would be extremely difficult to carry out such a scam at a large scale, without ever being noticed.



(a) optical scan



(b) Scantegrity

Figure 5-3: Property-part diagrams for the two electronic voting systems, illustrating the dependence structures for *AllCastTallied*

- **Optical scanner**: As demonstrated by multiple studies [5, 11, 23], modern optical scanners are known to be susceptible to a variety of security attacks that may compromise the integrity of the election. It is unlikely that these machines will achieve flawless security in the near future, and therefore, they should not be trusted as the sole producer of the election outcome.

- **Election official**: Election officials are generally given greater responsibilities than poll workers, and form a smaller, more exclusive group. Thus, a malicious act by an official is likely to cause more damage, and also more difficult to detect. By allowing voters and auditors to verify the election outcome themselves, Scantegrity eliminates the need to necessarily trust the officials.

Consider the property-diagrams in Figure 5-3 again. Note that the optical scanner, which is considered to be the least reliable part of the voting system, is no longer reachable from the top-level requirement. This is exactly the goal of end-to-end verifiable voting systems such as Scantegrity—to remove unreliable components from the chain of trust.

It is important to recognize that many of the arguments for the reliability of a trusted component in Scantegrity are based on assumptions that are not proven, but socially accepted. The reader might rightfully object that the dependability of Scantegrity, after all, stands on shaky ground that may never be rigorously justified. But this is true of nearly all software-intensive systems; every one of them makes assumptions about its environment, involving physical laws (which can be falsified) and human behaviors (occasionally erratic and unexpected).

Chapter 6

Related Work

6.1 Formal Models of Requirements Specification

Our work is not the first attempt at a formalization of the Problem Frames notation [18]. In his PhD thesis [38], Seater provided a formalization of problem diagrams in Alloy. He uses predicates to describe requirements, domain assumptions, and machine specifications. Our approach further builds on his work; we explicitly represent system components as signatures, and attach their associated constraints as signature facts. By doing so, we clarify relationships among these components, and are also able to better leverage the visualization facility of the Alloy Analyzer.

Seater also introduced the idea of *requirement progression* [39], a technique for systematically deriving a machine specification from a requirement. Before being able to reason about trusted bases, the analyst must first determine the specifications of the machines to be built. Therefore, we believe that requirement progression should be carried out as an early key step in the system development.

Gunter et al. proposed a formal model for requirements specification [7], which was another major source of inspiration for our work. In this approach, they describe a system as consisting of five different types of artifacts: domain knowledge (similar to our domain assumptions), requirements, machine specifications, programs (which implement the specifications), and programming platforms (on which the programs execute); we much simplified this classification by abstracting away programs and platforms as being encompassed by machines. Like ours, their approach designates each phenomenon as being shared between a machine and a domain, or internal to one of them. They also provide a formalization of requirements satisfaction arguments. One major addition in our notation, missing from their discussion, is the structuring of the environment into domains, which allows us to talk about relationships among them, and consider some of them as being untrustworthy or less reliable than others.

6.2 Module Dependence Diagram

In his seminal paper [31], Parnas introduced what is now widely known as the *module dependence diagram* (MDD) as a simple mechanism to evaluate modularity in design.

A key idea in this notation is the *uses* relation as the basic notion of dependence. It denotes a module A as using another module B if a successful completion of A depends on the correctness of a service that B provides.

Our work on the property-part diagram, which eventually led to this thesis, was originally inspired by the enormous merits as well as some fundamental limitations of MDDs:

- The uses relation necessarily results in a hierarchical structure of dependences between components, since by definition, it always denotes one module as providing service to another. But this notion is inadequate for expressing certain kinds of dependence structures. A key observation is that some components do not depend on one another, but *collaborate* to satisfy a higher goal or property.

  For example, in a real-time operating system, a scheduler enables periodic execution of a process by inserting it into a job queue and invoking the process' function at appropriate times. A typical MDD would indicate the scheduler as being dependent on the process, and yet it is not clear whether process is in the service of the scheduler. An equally valid argument would be that the scheduler is the one providing a service, since the process depends on itself being periodically scheduled. But a more sensible description would be to say that the schedule collaborates with various processes to ensure that the OS provides critical services to its client within a time deadline.

- The uses relation treats all dependences as having an equal weight. If a module A depends on B as well as C in an MDD, then it implies that a breakage in B or C may compromise the correctness of A. But sometimes this does not reveal the essential characteristics of a design. Perhaps A uses B for relatively mundane tasks, but together with C, it performs a highly critical task, so the dependence of A on C carries a far greater weight.

  For example, a typical user application, such as a web browser or a word process, relies on many different services inside an OS, such as networking, the file system, memory management, and process scheduling. Some of these services (the latter two) are more critical to the security and integrity of the system than others (the former two). Therefore, it would be sensible to pay more attention to dependences on these critical components, and examine the consequences of their failures. In fact, one of the primary motivations for the concept of a microkernel originates from a similar observation that critical components should be isolated to a small part of the system [42].

We have attempted to overcome these limitations by providing an explicit representation of properties in a property-part diagram. Collaboration can be expressed by assigning properties to each part, and then showing how these properties together imply a top-level property. The diagram also allows fine-grained tracing of properties throughout the system, and so the analyst can easily identify parts that belong to the exposure (i.e. trusted base) of a critical property.

In prior work [12], Jackson introduced an enhanced version of the MDD. In this approach, specifications are like ports on a module A, and a relation maps an incoming port (which represents a specification that is required by A and provided by another module B) to an outgoing port (a specification that A provides to some other module C). This approach overcomes some of the limitations with the uses relation. However, it requires each specification to be bound to a specific module, and cannot adequately express system-level properties that are achieved by a collaboration of multiple modules.

6.3 Goal-Oriented Notations

Goal-oriented approaches [4, 22, 45] take on the insight that goals are essential components of the system, and that they must be an explicit part of any requirements engineering framework.

Of these, KAOS [4] is most related to our approach. In its current version [33], the KAOS framework integrates four different kinds of models: a *goal model*, a *responsibility model*, an *object model*, and an *operation model*:

- A goal model takes the form of an AND-OR graph, and represents each top-level requirement as a *goal*. The goal is reduced into subgoals until they are small enough to be assigned to *agents*, which are like parts in the property-part diagram. The model illustrates the structure of a satisfaction argument for the requirement.

- A responsibility model assigns each of the primitive subgoals to agents. Combined with the goal model, it shows how these agents collaborate together to satisfy a higher goal, similarly to the property-part diagram.

- An operation model describes dynamic behaviors of these agents, and operations that they perform to satisfy their goals.

- An UML object model shows relationships among the agents and entities in the environment.

In addition, KAOS provides a formalization of goals and agents' responsibilities, along with the structure of an argument for the satisfaction of the goals.

The property-part diagram bears a resemblance to the goal model (which shows how system-level goals depend on smaller subgoals) and the responsibility model (which connects these subgoals to components in the system). But there are also key differences between the two approaches, mainly in the way properties are assigned to components (agents or parts). In KAOS, decomposition of a goal is completed when the analyst deems that the subgoals are primitive enough to be assigned to agents. For example, in an elevator system (described in [33]), a top-level requirement that "the system is protected against fire" is decomposed until one of the subgoals becomes "moving elevator is stopped next floor in case of fire signal", which is then assigned to the agent *ElevatorController*.

In our approach, the analyst would draw a problem diagram to represent this subgoal as a requirement that references two domains—the elevator and the floors; a machine *ElevatorController* would be shown as acting on the elevator by sending appropriate signals. Then, the analyst would derive a specification for *ElevatorController*, which would be along the lines of " in case of fire signal, the controller sends a *StopNext* signal to the elevator". This specification is more desirable than the one assigned to the agent in the KAOS approach, since it is described in terms of phenomena at the interface between the machine and the environment.

In general, KAOS emphasizes a top-down approach, highlighting decomposition of goals and the resulting structures of satisfaction arguments. In comparison, our approach puts greater emphasis on the structuring of the problem context for individual requirements, showing relationships between components through shared phenomena, and how the properties of domains and machines are formed as a result of this structuring. It seems to us that the two approaches are complementary to each other. In fact, the property decomposition pattern (Figure 4-3b, Section 4) in the property-part diagram was in part inspired by KAOS. We plan to explore possible synergies with KAOS in future as we investigate the integration of our modeling notation with the property-part diagram.

6.4 Trusted Bases

In security, a *trusted computing base* (TCB) refers to the set of all software and hardware components that are critical to the system security [26]. Establishing a TCB is widely regarded as a key step in the design of a secure system. However, outside the security community, the notion of a TCB has had relatively little impact on system development methodology.

Leveson and her colleagues were amongst the firsts to propose that a system be designed for safety from the outset, rather than relying on testing and inspection [28]; here, they also coined the term *safety kernel*. Rushby further elaborated on this idea, and provided a formalization of how a safety kernel may be used to enforce the safety of the overall system, regardless how untrusted components behave [35]. Wika and Knight further discussed the safety kernel within the context of two systems—a medical device and a nuclear reactor controller—but without providing much insight into how to establish or identify a safety kernel. Scheneider formally characterized the types of security properties that are enforceable by a reference monitor, and argued that his approach is a generalization of a safety kernel [37]. In general, it seems to us that many of the existing techniques for building TCBs, such as sandboxing and capability-based mechanisms [29], should be applicable to a wider variety of dependable systems, especially ones that are safety-critical.

Saltzer et al. introduced the notion of the *end-to-end principle* [36], which suggests that a system should be designed to ensure the correctness of critical operations at its end points, instead of relying on low-level components. Although initially discussed in the context of distributed systems, this principle has since become widely influential in other fields, such as security. For example, the idea of end-to-end verifiability,

which is a key guiding principle in the Scantegrity design, was inspired by this prior work.

6.5 Other Approaches to Dependability

Assurance based development (ABD) [6] is a methodology that integrates an *assurance case* with the development of a dependable system. An assurance case is an argument that the system satisfies dependability goals, and takes its form in the Goal Structuring Notation (GSN) [22]. The analyst states a safety or security requirement (e.g. "detect a failure in the optical scanner") as the top-level goal in a GSN structure (which is similar to a goal model in KAOS), and further decomposes it into smaller subgoals. However, instead of laying out the entire goal decomposition tree from the outset, ABD promotes that the assurance case be incrementally constructed top-down in parallel with the system development. Each design decision is made with some dependability goal in mind, and this decision, in turn, drives how the goal is decomposed into subgoals.

Each leaf of the decomposition tree is discharged with a piece of evidence that shows how the goal will be achieved; this may include testing plans, verification strategies, domain knowledge, etc. Consequently, the structure of an assurance argument mirrors the structure of a *process*. In contrast, the structure of a dependability argument in the property-part diagram is derived from the structure of the *overall system* in the problem diagram. As a result, the property-part diagram is likely to be more suitable for reasoning about dependences between components and the success of localizing critical properties. But this is not to say that the process is not important. As we argued with the voting examples, any dependability case must include strong pieces of evidence for why components can be trusted to behave as expected.

Problem Oriented Software Engineering (POSE) [8], which extends the Problem Frames approach, is a method for systematically deriving machine architectures from a given problem. The underlying framework is based on sequent calculus, and provides a set of derivation rules that can be used to transform a model into a more detailed solution, while ensuring that the the system as a whole satisfies the desired requirements. The POSE approach puts more emphasis on the structure of the *solution space* (i.e. design of machines), whereas we are currently focused on the dependability analysis within the *problem space*. Our long-term goal is to explore how the outcome of the analysis in the problem space drives the design of the system, so it seems that POSE should be examined further as a complementary approach to our work.

6.6 Failure Analysis

Many of the existing model-based techniques in failure analysis—including but not limited to fault trees, attack graph generation [41], and obstacle analysis [44]–are *goal-directed*. That is, the analyst describes an undesirable state of the system (sometimes called an anti-requirement or an anti-goal), and applies techniques to generate

malicious or erroneous behaviors of the components that lead to the failure. In comparison, our approach is *component-directed*; we systematically allow one or more components in the system to behave unexpectedly and observe the consequences on the system. In this sense, our approach bears a resemblance to fault injection, although the latter term usually refers to introducing faults into an implementation during the testing phase.

One distinguished aspect of our approach is the use of the meta-objects *Satisfied* and *Unsatisfied*, allowing us to "toggle" between normal and erroneous states of a component. Combined with a constraint solver such as the Alloy Analyzer, this provides a flexible and convenient mechanism for generating different kinds of failure scenarios—for example, one that involves multiple components, or latent failures that may go undetected. Furthermore, the visualizer can take advantage of these meta-objects, and highlight only the components that exhibit failures.

Chapter 7

Discussion

7.1 Benefits of the Structuring Mechanism

The Problem Frames approach forms a strong basis for our notation, with the structuring of a system into three distinct layers—domains, machines and requirements. But reasoning about trusted bases, in principle, does not require such a structuring. An alternative approach to building a dependability argument, using a notation such as Alloy, could simply consist of declarations of phenomena and a set of predicates that describe their constraints and relationships. The analyst would use an assertion to check that the conjunction of the predicates implies the satisfaction of a critical requirement, and examine an unsatisfiable core to identify a trusted base, which would consist of some subset of the predicates. Then, what do we gain by imposing this extra structure on the model?

- A key step in designing a system involves a division of responsibilities among different parts of the system [25]. Grouping constraints into domains and machines clarifies this division, and facilitates the allocation of resources for discharging domain assumptions and machine specifications. The structuring also allows the analyst to talk about certain parts of the system as being more trustworthy or reliable than others.

- The structuring makes relationships among domains, machines, and requirements explicit. It is well known (but sometimes neglected) that a requirement directly interacts only with domains, and a machine must be built to observe or constrain these domains in such a way that results in the satisfaction of the requirement; this clarification is one of the hallmarks of the Problem Frames approach. Furthermore, the structuring enables tracing of information flow throughout the system (similarly to data-flow analysis). For example, in the optical scan system, with simple analysis of *observes-or-constrains* relationships, it can be inferred that the information originating from the *Voters* domain flows through the path

  <Voters, OpticalScanner, ElectionOfficial, PublicBoard>

because *OpticalScanner* observes *Voters*, and *ElectionOfficial* observes *OpticalScanner* and constrains *PublicBoard*.

7.2 Modeling Style

Alloy is essentially a pure logic, and its flexibility supports various modeling idioms. For example, the analyst could describe a system using a dynamic behavioral model, with an explicit representation of a state, showing how the system achieves a requirement through a sequence of operations. A very different style of a model is also possible—one that describes static relationships among components, which together satisfy an invariant that ultimately implies the desired requirement.

A static model tends to be more succinct, and easier to reason about. It also suffers less from the risk of an implementation bias, because it expresses only *what* the relationships among components are, not *how* they are achieved. However, certain properties or relationships are tricky to express in a static model (e.g. a property that depends on a strict ordering events), and dynamic modeling may be preferable in these situations. In general, reasoning about a system often involves a mix of the two modeling styles.

The debate of "dynamic versus static" is largely an orthogonal issue to our work. However, through our experience with the notation so far, we have discovered that the static style is surprisingly effective and natural for modeling constrains in domains, machines, and requirements. Our voting system models follow this approach; constraints are written to declare static relationships between pairs of components, and no dynamic behaviors are explicitly described. For example, the following constraint says that there is a ballot image for each marked ballot produced by a voter; what events occur to produce this effect are not described.

```
sig BallotImage {
  choice : lone Candidate,
  mballot : MBallot
}
one sig OpticalScanner extends Machine {
  voters : Voters,
  records : set BallotImage,
  tally : Tally
}{
  ...
  (all mb : markedBy.(voters.members) |
    let rec = (records <: mballot).mb |
      one rec &&
      rec.choice = mb.choice)
  ...
}
```

Alternatively, the analyst could take a more bottom-up approach, beginning with a set of basic phenomena (e.g. voters, candidates, ballot images, etc) and writing

predicates to describe operations that these entities perform on each other. For example, we could declare a signature *OpticalScanner* to represent the state of the machine, and write a predicate *insert* to show how the state changes from s to s' (note that *OpticalScanner* is no longer directly related to the *Voters* domain):

```
sig OpticalScanner {
  records : set BallotImage,
  tally : Tally
}
pred insert[v : Voter, s, s' : OpticalScanner] {
  s'.tally = s.tally
  let markedBallot = markedBy.voter,
      rec = {i : BallotImage | i.choice = mb.choice && i.mballot = mb} |
        one rec &&
        s'.records = s.records + rec
}
```

We posit, pending further validation, that our structuring mechanism is conducive to the former, static style of modeling over the latter, dynamic one. In hindsight, this is not surprising. An approach that involves a top-down structuring of the system—beginning with a layout of requirements, domains and machines—is likely to encourage the analyst to reason about the relationships between them as invariants over their shared phenomena. It is worth noting that many of the system models written in Alloy (including the author's model of a flash file system [21]) follow the dynamic style of modeling. An interesting experiment would be to apply our structuring mechanism to some of these models, and observe whether they have compelling static counterparts.

7.3 Modeling with Meta-Objects

The flexibility of the *signature* mechanism [17] in Alloy allows reification of meta-level concepts, which are made available in the model for logical manipulation. In our notation, the set *Constraint* and all of its subtypes—*Requirement*, *Machine*, *Domain*, *Satisfied*, and *Unsatisfied*–are examples of *meta-objects*, because (1) they are used to describe or manipulate *base-objects* (e.g. *Voters*, *Booth*, *Scanner*, etc.), and (2) they are language constructs that can be directly referenced in a model. Benefits of modeling with meta-objects include the following:

1. The analyst can define logical relationships between meta-objects and base-objects, and exploit these relationships to generate interesting system configurations. In our framework, we used *Satisfied* and *Unsatisfied* to manipulate the status of components, and generate failure scenarios (Section 3.3).

2. Meta-objects can be exploited inside the Alloy visualizer to emphasize aspects of the system that would otherwise be implicit. For example, the analyst can customize the visualizer to distinguish the members of the set *Unsatisfied* from

other components, thereby highlighting problematic parts of the system. Similarly, the analyst can define a function to compute the set of all machines and domains that are reachable from a requirement. The function is available to the visualizer, so the analyst can take advantage of this to hide all components that are irrelevant to a particular requirement of concern.

3. The analyst may use meta-objects to define templates of generic properties that can be reused across different models. For example, the assertion

```
Domain in Satisfied && Machine in Satisfied => Requirement in Satisfied
```

is a generic form of a satisfaction argument that can be applied to any model that is written in our notation.

A similar idea can be applied to derive the flow of information throughout the system. In one possible formulation, we could augment requirements, domains, and machines with generic relations *constrains* and *observes* as follows:

```
abstract sig Constraint {
  observes : set Constraint,
  constrains : set Constraint
}
```

The field *observes* is a binary relation that maps a component C to the set S of all components that C observes; information flows from each member of S to C. Similarly, the expression *C.constrains* returns the set T of all components that C acts on; information flows from C to every member of T. For example, in the optical scan voting system, the *ElectionOfficial* domain observes *OpticalScanner* and constrains *PublicBoard*, so the two relations for this domain would be defined as follows:

```
sig ElectionOfficial {
  scanner : OpticalScanner,
  pboard : PublicBoard
}{
  observes = scanner // observes
  constrains = pboard // constrains
}
```

Then, by taking the transitive closure of these relations, we can define a function $flows$ that traces the flow of information throughout the system:

```
fun flows : Constraint -> Constraint {
  ^(~observes + constrains)
}
```

For example, the expression $C.flows$ would return the set of all components to which any information that originates from C flows.

Note that this style of modeling does not require the built-in meta features in Alloy, although they could be useful for simplifying the derivation of meta-level expressions.

7.4 Limitations

The analyst must keep in mind that, as Jackson points out [19], any attempt to model the real world using a formalism will result in imperfect descriptions. Our approach also faces the same type of limitations. First of all, we made a number of simplifications in order to keep the modeling task tractable. For example, we assumed that every voter casts the ballot exactly as intended; but in reality, some small fraction of voters make mistakes while filling in their ballots. We also deliberately omitted intricate details of the optical scanner (e.g. its various hardware components); this means that the scanner may actually be susceptible to a wider variety of security attacks than our models indicate. We could have also easily forgotten to include certain phenomena as a part of the voting system models.

This highlights the importance of a domain expert's role in discharging domain assumptions. It may never be possible to model the real world to perfection, but the domain expert must ensure that the descriptions are reasonably accurate, because dependability arguments rely on their fidelity.

Chapter 8

Conclusion

8.1 Summary of the Proposed Approach

In this thesis, we have proposed a new approach to analyzing the dependability of a system. The key idea in this approach is the notion of a trusted base, which refers to the set of all components that are responsible for fulfilling a critical requirement. We have argued the following points:

- In order to show that a system is dependable, the analyst must identify the trusted bases for all critical requirements, and ensure that every component in the trusted bases can be trusted with a high degree of confidence. The justifications for the trust should form an integral part of a dependability case.
- The system includes not only machine components like software or hardware, but also parts of the environment, such as human operators and physical entities. Therefore, identification and evaluation of a trusted base must take into behaviors and properties of the environment.
- If a trusted base contains components that are considered unreliable, the analyst must attempt to redesign the system so that those components no longer belong to the new trusted base.

We summarize our approach in the following list of steps, which the analyst may refer to as a"recipe for dependability":

1. Identify the list of critical requirements that the system must fulfill.
2. Construct a problem diagram to layout the structure of the problem space, with requirements, domains that they reference, and machines that must be implemented to satisfy these requirements. Using techniques such as requirement progression [39], derive domain assumptions and machine specifications.
3. Formally specify the system using our modeling framework, and check the consistency of satisfaction arguments for the requirements (i.e. $D, M \vdash R$) in the Alloy Analyzer (Section 3.1). In this step, the analyst may discover that some

of the domain assumptions or machine specifications need to be strengthened to ensure that the requirements hold.

4. Using the unsat core facility in the Alloy Analyzer, identify a trusted base for each requirement (Section 3.2.1). When there are multiple potential candidates for a trusted base, then the analyst selects one that is deemed to be the most trustworthy, based on a weighting function that evaluates the reliability of the bases. (Section 3.2.2).

5. Examine various scenarios in which some of the components in the trusted base fail to behave as expected (Section 3.3), and estimate the likelihood and impact of the failures. This step provides the analyst with information that may be factored into the decision to keep or exclude a component from the trusted base.

6. For each component in the trusted base, attempt to come up with a justification for why that component can be reliably trusted (Section 3.2.3). Every justification must be supported with a strong piece of evidence, such as testing plans, verification strategies, and credible domain knowledge. The analyst may also generate a property-part diagram based on the information in the trust base, and examine the the exposure of the requirement throughout the system (Chapter 4).

7. If the trusted base contains components that cannot be reliably trusted, then the analyst must attempt to redesign the system. This may involve re-structuring the system as to exclude those components from the trusted base, or adding extra fault-tolerance mechanisms to ensure that the requirement holds even in the presence of component failures.

8. Repeat Steps 2-6 until the analyst is able to produce a dependability case that shows that the system fulfills all of the critical requirements.

Note that our approach is, by no means, a "silver bullet" that guarantees that the resulting system will be completely reliable. However, we believe that our work is a step towards a cost-effective and systematic method for building systems that users can depend on with confidence.

8.2 Future Work

Our approach is useful for identifying trusted bases and suggesting that the system be redesigned, if necessary, for greater dependability. But it is far from clear how to carry out the task of redesign. We could consider Scantegrity as an improved design of the optical scan system, but how does one actually arrive at a system such as Scantegrity? Much work remains to be done on extracting reusable design knowledge from examples of successful (and unsuccessful) software engineering projects, and developing systematic approaches for applying the knowledge to a new design task.

Pattern-based approaches, such as Hanmer's patterns for fault-tolerance systems [9], may be helpful here.

A model written in our notation represents a system at the highest level of design. In the next step of the development, the machines in the model would eventually need to be mapped into design descriptions at the architectural level. An interesting research challenge is whether existing design knowledge (e.g. architectural styles [40]) can be leveraged to structure the machines. In previous work [32], researchers have looked at decomposition of a machine into a set of architectural components (such as pipes and filters) using the Problem Frames notation. Their work is a promising approach that should be further explored.

In reality, requirements change frequently throughout the course of the development. An interesting question to ask is how the structure of the system changes due to new or modified requirements. Note that it is relatively easier (but still by no means trivial) to modify the specification of a machine than to alter the behavior or property of a domain. In some cases, it may be nearly impossible to do the latter. In general, it would be interesting to study the types of changes that machines or domains undergo in response to a change in a requirement, and classify them into different categories for later reuse.

Appendix A

Complete Alloy Models of the Electronic Voting Systems

```
module voting/notation

/*
 * notation.als
 *
 * Basic constructs of the notation
 */

open util/boolean

// constrains
abstract sig Constraint {}
sig Satisfied in Constraint {}
sig Unsatisfied in Constraint {}

fact {
  no (Satisfied & Unsatisfied)  // disjointness
  Constraint = Satisfied + Unsatisfied  // complete partition
}

// domains
abstract sig Domain extends Constraint {}

// machines
abstract sig Machine extends Constraint {}

// requirements
abstract sig Requirement extends Constraint {}
```

```
module voting/common
```

```
/*
 * common.als
 *
 * Phenonema and domains that are common to both
 * the optical scan system and Scantegrity
 */

open voting/notation

/*
 * phenonema
 */

sig Voter { // an individual voter
  intent : lone Candidate
}

sig Candidate {}

sig Tally {
  counts : Candidate -> one Int,
}{
  all c : Candidate.counts | c >= 0
}

// blank ballot prior to being marked
abstract sig Ballot {
  givenTo : Voter   // voter that this ballot is given to
}

// marked ballot
abstract sig MBallot {
  ballot : Ballot,
  choice : lone Candidate,
  markedBy : Voter  // voter that has completed this ballot
}

// ballot image inside a scanner
sig BallotImage {
  choice : lone Candidate,
  mballot : MBallot
}

// An assumption
// Each completed ballot corresponds to a unique physical ballot
fact AtMostOneMarkedBallot {
  ballot in MBallot lone -> lone Ballot
```

```
}

/*
 * domains
 */

// where the tally gets posted
one sig PublicBoard extends Domain {
  tally : Tally
}{
  this in Satisfied
}
```

```
module voting/optical

/*
 * optical.als
 *
 * Model of the optical scan voting system
 */

open voting/common

/*
 * domains
 */

one sig Voters extends Domain {
  members : set Voter
}{
    this in Satisfied
  iff
    // voter marks the ballot as intended
    all v : members, mballot : MBallot |
        mballot.markedBy = v
      implies
        mballot.choice = v.intent
}

one sig Checkin extends Domain {
  voters : Voters
}{
    this in Satisfied
  iff
    // each voter is given exactly one ballot
    all v : Voters.members |
      one givenTo.v
}
```

```
one sig Booth extends Domain {
  voters : Voters
}{
    this in Satisfied
  iff
    // the ballot doesn't go missing as the voter goes through the booth
    all v : Voters.members |
        some givenTo.v
      implies
        some cb : MBallot | cb.markedBy = v && cb.ballot in givenTo.v
}

one sig Official extends Domain {
  scanner : Scanner,
  pboard : PublicBoard
}{
    this in Satisfied
  iff
    // offical correctly records the tally
    scanner.tally = pboard.tally
}

one sig Box extends Domain {
  voters : Voters,
  mballots : set MBallot
}{
    this in Satisfied
  iff
    // each paper ballot from the voters is stored in the box
    all v : Voters.members |
      markedBy.v in mballots
}

/*
 * machine
 */

one sig Scanner extends Machine {
  voters : Voters,
  records : set BallotImage,
  tally : Tally,
}{
    this in Satisfied
  iff
    (
    (all cb : markedBy.(Voters.members) |
```

```
      let rec = (records <: mballot).cb |
        one rec &&   // exaclty one soft record per completed ballot
        rec.choice = cb.choice) && // and correctly records the voters choice

        // results are correctly tabulated
        (all c : Candidate |
      tally.counts[c] = #(records & choice.c))
    )
}

/*
 * requirements
 */

// for simplification, we assume that all voters belong to the Voters domain
fact VotersDomainComplete {
  Voters.members = Voter
}

// each candidate receives all votes that were intended for him/her
one sig AllCastTallied extends Requirement {
  voters : Voters,
  result : PublicBoard
}{
    this in Satisfied
  iff
    all c : Candidate |
      result.tally.counts[c] >= #(intent.c)
}

// each candidate receives only the votes that were intended for him/her
one sig OnlyCastTallied extends Requirement {
  voters : Voters,
  result : PublicBoard
}{
    this in Satisfied
  iff
    all c : Candidate |
      result.tally.counts[c] <= #(intent.c)
}

/*
 * trusted bases
 */

fun TB : Requirement -> set (Domain + Machine) {
  AllCastTallied -> {Voters + Checkin + Booth + Official + Scanner} +
```

```
  OnlyCastTallied -> {Voters + Checkin + Booth + Official + Scanner}
}

// If all domains & machines in trusted bases are behaving correctly,
// then the correctness req. holds true
check AllCastTalliedReq {
    TB[AllCastTallied] in Satisfied
  implies
    AllCastTallied in Satisfied
} for 5

check OnlyCastTalliedReq {
    TB[OnlyCastTallied] in Satisfied
  implies
    OnlyCastTallied in Satisfied
} for 5
```

```
module voting/scantegrity

/*
 * scantegrity.als
 *
 * Scantegrity
 * An end-to-end verifiable voting system
 *
 * Based on the paper
 * "Scantegrity II: End-to-End Verifiability for Optical Scan Election Systems
 *   using Invisible Ink Confirmation Codes"
 * by David Chaum et al.
 * in 2008 USENIX Electronic Voting Workshop
 */

open voting/common

/*
 * phenonema specific to Scantegrity
 */

// confirmation code
sig Code {}

// each candidate is linked to a confirmation code
sig SBallot extends Ballot {
  codes : Candidate -> one Code
}{
  // all codes are unique
  all disj c1, c2 : Candidate |
    codes[c1] != codes[c2]
```

```
}

// receipt that the voter takes
// may optionally write down the confirmation code revealed on the ballot
sig Receipt {
  mballot : MBallot,
  code : lone Code,
  keptBy : Voter
}

// An assumption
// Each receipt corresponds to a unique physical ballot
fact AtMostOneReceipt {
  Receipt <: mballot in Receipt lone -> lone MBallot
}

/*
 * domains
 */

one sig Voters extends Domain {
  wboard : WebBoard,
  members : set Voter
}{
    this in Satisfied
  iff
    (
    // voter marks the ballot as intended
    (all v : members, mballot : MBallot |
        mballot.markedBy = v
      implies
        mballot.choice = v.intent) &&
        // all voters verify the confirmation codes on their ballots to
        // match the published codes on the web board
    (all v : members, receipt : keptBy.v |
      wboard.codes[receipt.mballot] = receipt.code)
    )
}

one sig Checkin extends Domain {
  voters : Voters,
}{
    this in Satisfied
  iff
    // each voter is given exaclty one ballot
    all v : Voters.members |
      one givenTo.v
```

```
}

one sig Booth extends Domain {
  voters : Voters
}{
    this in Satisfied
  iff
    // The ballot doesn't go missing as the voter goes through the booth
    // and the voter keeps the receipt
    all v : Voters.members, sballot : Ballot |
        sballot in givenTo.v
      implies
        some cb : MBallot |
          cb.markedBy = v &&
          cb.ballot in sballot &&
          some r : Receipt |
            r.mballot = cb &&
            r.keptBy = v &&
            r.code = cb.ballot.codes[cb.choice]
}

// web site where the voter can check the confirmation code on their receipt
// It also publishes the tally
one sig WebBoard extends Domain {
  codes : MBallot -> one Code,
  tally : Tally
}{
  this in Satisfied
}

one sig Official extends Domain {
  scanner : Scanner,
  result : PublicBoard
}{
    this in Satisfied
  iff
    // offical correctly records the tally
    scanner.tally = result.tally
}

one sig Box extends Domain {
  voters : Voters,
  mballots : set MBallot
}{
    this in Satisfied
  iff
    // each paper ballot from the voters is stored in the box
```

```
    all v : Voters.members |
      markedBy.v in mballots
}

one sig Auditor extends Domain {
  pboard : PublicBoard,
  wboard : WebBoard
}{
    this in Satisfied
  iff
    // auditor checks to make sure that the independent tallies match
    talliesMatch[pboard.tally, wboard.tally]
}

pred talliesMatch[t1, t2 : Tally] {
  all c : Candidate |
    t1.counts[c] = t2.counts[c]
}

/*
 * machines
 */

one sig Publisher extends Machine {
  scanner : Scanner,
  wboard : WebBoard
}{
    this in Satisfied
  iff
    (
    // translates the scanned records and publishes confirmation codes
    // for all cast ballots
    (all rec : scanner.records |
      wboard.codes[rec.mballot] = rec.mballot.ballot.codes[rec.choice]) &&

    // too weak; need the following additional constraint
    (all postedBallot : wboard.codes.Code |
      some scanner.records & mballot.postedBallot)
    )
}

one sig Tabulator extends Machine {
  scanner : Scanner,
  wboard : WebBoard,
  // mapping from a code on a ballot to a candidate
  // In reality, for secrecy, this link is hidden using cryptographic protocols
  switchboard : SBallot -> Code -> Candidate
```

```
}{
    this in Satisfied
  iff
    (
    // ballots are properly created with correct mappings from
    // candidates to codes
    (all sballot : SBallot |
      sballot.codes = ~(switchboard[sballot])) &&

    // independent tallying step based on the set of cast ballots
    // and confirmation codes
    (all c : Candidate |
      let matches =
      { rec : scanner.records |
        let sballot = rec.mballot.ballot |
          switchboard[sballot][sballot.codes[rec.choice]] = c } |
          wboard.tally.counts[c] = #matches) &&

    // Check whether there are any suprious records in the scanner
    // e.g. Two ballot images for the same physical ballot
    ((BallotImage <: mballot).ballot in scanner.records lone -> lone SBallot)
    )
}

one sig Scanner extends Machine {
  voters : Voters,
  records : set BallotImage,
  tally : Tally,
}{
    this in Satisfied
  iff
    (
    // each voter's ballot is correctly recorded inside the scanner
    (all cb : markedBy.(Voters.members) |
      let rec = (records <: mballot).cb |
        one rec &&   // exaclty one soft record per completed ballot
        rec.choice = cb.choice) && // and correctly records the voters choice

    // results are correctly tabulated
    (all c : Candidate |
      tally.counts[c] = #(records & choice.c))
    )
}

/*
 * requirements
 */
```

```
fact VotersDomainComplete {
  // for simplification, we assume that all voters belong to the Voters domain
  Voters.members = Voter
}

// each candidate receives all votes that were intended for him/her
one sig AllCastTallied extends Requirement {
  voters : Voters,
  result : PublicBoard
}{
    this in Satisfied
  iff
    all c : Candidate |
      result.tally.counts[c] >= #(intent.c)
}

// each candidate receives only the votes that were intended for him/her
one sig OnlyCastTallied extends Requirement {
  voters : Voters,
  result : PublicBoard
}{
    this in Satisfied
  iff
    all c : Candidate |
      result.tally.counts[c] <= #(intent.c)
}

/*
 * trusted bases
 */

fun TB : Requirement -> set (Domain + Machine) {
  AllCastTallied ->
    {Voters + Auditor + Checkin + Booth + Tabulator + Publisher} +
  OnlyCastTallied ->
    {Voters + Auditor + Checkin + Booth + Tabulator + Publisher}
}

// If all domains & machines in trusted bases are behaving correctly,
// then the correctness req. holds true
check AllBallotsTalliedReq {
    TB[AllCastTallied] in Satisfied
  implies
    AllCastTallied in Satisfied
} for 5
```

```
check OnlyBallotsTalliedReq {
    TB[OnlyCastTallied] in Satisfied
  implies
    OnlyCastTallied in Satisfied
} for 5
```

Bibliography

[1] Jean-Raymond Abrial. *The B-book: Assigning programs to meanings*. Prentice Hall, 1996.

[2] David Chaum, Richard Carback, Jeremy Clark, Aleksander Essex, Stefan Popoveniuc, Ronald L. Rivest, Peter Y. A. Ryan, Emily Shen, and Alan T. Sherman. Scantegrity ii: End-to-end verifiability for optical scan election systems using invisible ink confirmation codes. In *EVT*, 2008.

[3] Liming Chen and Algirdas Avizienis. N-version programming: a fault-tolerance approach to reliability of software operation. In *Proc. 8th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-8)*, pages 3–9. Citeseer, 1978.

[4] Anne Dardenne, Axel van Lamsweerde, and Stephen Fickas. Goal-directed requirements acquisition. *Sci. Comput. Program.*, 20(1-2):3–50, 1993.

[5] Seda Davtyan, Sotiris Kentros, Aggelos Kiayias, Laurent D. Michel, Nicolas C. Nicolaou, Alexander Russell, Andrew See, Narasimha Shashidhar, and Alexander A. Shvartsman. Taking total control of voting systems: firmware manipulations on an optical scan voting terminal. In *SAC*, pages 2049–2053, 2009.

[6] Patrick J. Graydon, John C. Knight, and Elisabeth A. Strunk. Assurance based development of critical systems. In *DSN*, pages 347–357, 2007.

[7] Carl A. Gunter, Elsa L. Gunter, Michael Jackson, and Pamela Zave. A reference model for requirements and specifications-extended abstract. In *ICRE*, page 189, 2000.

[8] Jon G. Hall, Lucia Rapanotti, and Michael Jackson. Problem oriented software engineering: Solving the package router control problem. *IEEE Trans. Software Eng.*, 34(2):226–241, 2008.

[9] Robert S. Hanmer. *Patterns for fault tolerant software*. John Wiley and Sons, 2007.

[10] C. A. R. Hoare. Programs are Predicates. *Royal Society of London Philosophical Transactions Series A*, 312:475–488, October 1984.

[11] Harri Hursti. Critical security issues with Diebold optical scan design. *Black Box Voting Project, July*, 4, 2005.

[12] Daniel Jackson. Module dependences in software design. In *RISSEF*, pages 198–203, 2002.

[13] Daniel Jackson. *Software Abstractions: Logic, language, and analysis.* MIT Press, 2006.

[14] Daniel Jackson. A direct path to dependable software. *Commun. ACM*, 52(4):78–88, 2009.

[15] Daniel Jackson and Michael Jackson. Separating concerns in requirements analysis: An example. In *RODIN Book*, pages 210–225, 2006.

[16] Daniel Jackson and Eunsuk Kang. Property-Part Diagrams: A Dependence Notation for Software Systems. In *ICSE '09 Workshop: A Tribute to Michael Jackson*, 2009.

[17] Daniel Jackson, Ilya Shlyakhter, and Manu Sridharan. A micromodularity mechanism. In *ESEC / SIGSOFT FSE*, pages 62–73, 2001.

[18] Michael Jackson. *Problem Frames: Analyzing and structuring software development problems.* Addison-Wesley, 2000.

[19] Michael Jackson. What can we expect from program verification? *IEEE Computer*, 39(10):65–71, 2006.

[20] Cliff B. Jones. *Systematic software development using VDM.* Prentice Hall, 1990.

[21] Eunsuk Kang and Daniel Jackson. Formal modeling and analysis of a flash filesystem in alloy. In *ABZ*, pages 294–308, 2008.

[22] Tim Kelly and Rob Weaver. The Goal Structuring Notation–A Safety Argument Notation. In *Proc. of Dependable Systems and Networks 2004 Workshop on Assurance Cases*, 2004.

[23] Aggelos Kiayias, Laurent D. Michel, Alexander Russell, and Alexander A. Shvartsman. Security assessment of the diebold optical scan voting terminal. Technical report, University of Connecticut Voting Technology Research Center, October 2006.

[24] Tadayoshi Kohno, Adam Stubblefield, Aviel D. Rubin, and Dan S. Wallach. Analysis of an electronic voting system. In *IEEE Symposium on Security and Privacy*, pages 27–, 2004.

[25] Butler W. Lampson. Hints for computer system design. *IEEE Software*, 1(1):11–28, 1984.

[26] Butler W. Lampson, Martin Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: Theory and practice. *ACM Trans. Comput. Syst.*, 10(4):265–310, 1992.

[27] Robin C. Laney, Leonor Barroca, Michael Jackson, and Bashar Nuseibeh. Composing requirements using problem frames. In *RE*, pages 122–131, 2004.

[28] N. G. Leveson, T. J. Shimeall, J. L. Stolzy, and J. C. Thomas. Design for safe software. In *American Institute of Aeronautics and Astronautics, Aerospace Sciences Meeting, 21 st, Reno, NV*, 1983.

[29] Henry M. Levy. *Capability-based computer systems*. Digital Press, 1984.

[30] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[31] David Lorge Parnas. Designing software for ease of extension and contraction. In *Proceedings of the 3rd international conference on Software engineering*, pages 264–277. IEEE, 1978.

[32] Lucia Rapanotti, Jon G. Hall, Michael Jackson, and Bashar Nuseibeh. Architecture-driven problem decomposition. In *RE*, pages 80–89, 2004.

[33] Respect-IT. *A KAOS Tutorial*, 2007. Retrieved from "www.objectiver.com/fileadmin/download/documents/KaosTutorial.pdf" on Dec 27, 2009.

[34] Ron L. Rivest and John P. Wack. On the notion of "software independence" in voting systems. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 366(1881):3759, 2008.

[35] John Rushby. Kernels for safety. *Safe and Secure Computing Systems*, pages 210–220, 1989.

[36] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-to-end arguments in system design. *ACM Trans. Comput. Syst.*, 2(4):277–288, 1984.

[37] Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.

[38] Robert Seater. *Building dependability arguments for software intensive systems*. PhD thesis, Massachusetts Institute of Technology, 2009.

[39] Robert Seater and Daniel Jackson. Requirement progression in problem frames applied to a proton therapy system. In *RE*, pages 166–175, 2006.

[40] Mary Shaw and David Garlan. *Software architecture: perspectives on an emerging discipline*. Prentice Hall, 1996.

[41] Oleg Sheyner, Joshua W. Haines, Somesh Jha, Richard Lippmann, and Jeannette M. Wing. Automated generation and analysis of attack graphs. In *IEEE Symposium on Security and Privacy*, pages 273–284, 2002.

[42] Andrew S. Tanenbaum, Jorrit N. Herder, and Herbert Bos. Can we make operating systems reliable and secure? *IEEE Computer*, 39(5):44–51, 2006.

[43] Emina Torlak, Felix Sheng-Ho Chang, and Daniel Jackson. Finding minimal unsatisfiable cores of declarative specifications. In *FM*, pages 326–341, 2008.

[44] Axel van Lamsweerde and Emmanuel Letier. Handling obstacles in goal-oriented requirements engineering. *IEEE Trans. Software Eng.*, 26(10):978–1005, 2000.

[45] Eric S. K. Yu. Towards modeling and reasoning support for early-phase requirements engineering. In *RE*, pages 226–235, 1997.