Towards an Engineering Discipline of Computational Security

Ali Mili, Alex Vinokurov
College of Computer Science
New Jersey Institute of Technology
Newark NJ 07102-1982 mili@cis.njit.edu

Lamia Labed Jilani
Institut Superieur de Gestion
Bardo 2000 Tunisia
Lamia.Labed@isg.rnu.tn

Frederick T. Sheldon
U.S. DOE Oak Ridge National Lab
PO Box 2008, MS 6085, 1 Bethel Valley Rd
Oak Ridge TN 37831-6085 sheldonft@ornl.gov

Rahma Ben Ayed
ENIT, University of Tunis
Belvedere, 1002 Tunisia
rahma.benayed@enit.rnu.tn

September 13, 2006

Abstract

George Boole ushered the era of modern logic by arguing that logical reasoning does not fall in the realm of philosophy, as it was considered up to his time, but in the realm of mathematics. As such, logical propositions and logical arguments are modeled using algebraic structures. Likewise, we submit that security attributes must be modeled as formal mathematical propositions that are subject to mathematical analysis. In this paper, we approach this problem by attempting to model security attributes in a refinement-like framework that has traditionally been used to represent reliability and safety claims.

Keywords

Computable security attributes, survivability, integrity, dependability, reliability, safety, security, verification, testing, fault tolerance.

1 Modeling Security as a Dependability Attribute

Even though logically, system reliability is driven exclusively by the existence and possible manifestation of faults, empirical observations regularly show a very weak correlation between faults and reliability. In [7], Mills and Dyer discuss an example where they find a variance of 1 to 50 in the impact of faults on reliability; i.e. some faults cause system failure 50 times more often than others; while their experiment highlights a variance of 1 to 50, we have no doubt that actual variance is in fact unbounded. Also, they find that they can remove 60 percent of a system's faults and improve its reliability by only ... 3 percent.In a study of IBM software products, Adams [1] finds that many faults in the system are only likely to cause failure after hundreds of thousands of months of product usage.

We argue that the same may be true for security: vulnerabilities in a system may have widely varying impacts on system security. In fairness, the variance may be wider for reliability than for security, because in malicious security violations high impact vulnerabilities may be more attractive targets than lower impact vulnerabilities, but wide variances are still quite plausible. Wide variances, to the extent that they are borne out, have broad impacts on security management:

- In practice, security ought not be defined as the absence of vulnerabilities, no more than reliability is defined by the absence of faults.
- In practice, security ought not be measured or quantified by the number of vulnerabilities, just as it is widely agreed (as highlighted by Adams' [1] and Mills' [7] work) that faults per KLOC is an inappropriate measure of reliability.
- Security cannot be improved by focusing on vulnerabilities, as we have no way to tell whether a given vulnerability has low (1) or high (50) impact on security. Rather, security should be managed by pursuing a policy that leads us to the highest impact vulnerabilities first (a similar approach to usage pattern testing [4, 7, 9]).

In light of these observations, we argue in favor of modeling security in a way that reflects its visible, measurable, observable attributes, rather than its hypothesized causes. To this effect, we introduce the outline of a *Logic for System Security*, which represents / captures security properties in terms of its observable attributes. This logic is defined in terms of the following features:

- A notation for *security specification*, which details how to capture security requirements of a system.

- A formula for *security certification*, which formulates the condition under which a system (represented by its security abstraction) meets a given set of security requirements (represented by security specifications).

Note that in order to quantify reliability as the mean time to *failure*, we must define what it means to *fail*, which in turn requires that we define *specification* and *correctness*. Likewise, defining and quantifying security requires that we define the concepts of security specification and security certification. In this paper, we discuss broad premises that characterize our approach, and present tentative notations and formulas for the proposed logic for system security.

In section 2 we briefly review our results in using a refinement calculus to compose verification claims and decompose verification goals in terms of reliability and safety. In section 3 we discuss a generalized representation of reliability, safety and security claims that stems from our refinement based model. In section 4 we discuss formal definitions of some security attributes, and explore how these definitions lend these attributes to be integrated into our refinement-based model. Finally, we briefly summarize and assess our findings in section 6.

2 Background: Genesis of our Approach

In this section we will briefly present the main contributions of [6], then we discuss how and why we propose to extend this work, thereby laying the groundwork for our subsequent developments. For the sake of readability, we will keep the discussion of this section (and most of the paper, in fact) fairly non-technical, referring interested readers to bibliographic sources for details; though we may sometimes present mathematical formulas, to fix the reader's ideas, we do not consider that understanding the details of these formulas is required to follow our discussions. Also, while the formulas we present refer to a relation-based refinement calculus, *we submit that most of our claims hold for most specification/ refinement models*. Indeed, it is possible to define the concept of refinement in any specification model, and to build our arguments from the ground up using the model-specific refinement ordering.

2.1 Refinement Calculi

Without significant loss of generality, we use homogeneous relations (i.e. relations from some set S to itself) to represent functional specifications and program functions. Among constant relations on some space S we consider the *universal* relation ($S \times S$), that we denote by L; the *identity* relation ($(s,s)|s \in S$), that we denote by I, and the *empty* relation ($\{\}$), that we denote by ϕ. We denote the relational product by mere concatenation, i.e. RR' as the product of R by R', and we use the *hat* symbol ($\widehat{R}$) to represent relational inversion. A relation R is said to be *total* if and only if $I \subseteq R\widehat{R}$; and a relation is said to be *deterministic* if and only if $\widehat{R}R \subseteq I$.

We introduce the *refinement* ordering between relations (interpreted as specifications) as follows: R *refines* R' if and only if

$$RL \cap R'L \cap (R \cup R') = R'.$$

We denote this property by $R \sqsupseteq R'$ or $R' \sqsubseteq R$ and we admit that this is a partial ordering (i.e. it is reflexive, antisymmetric and transitive). Intuitively, R refines R' if and only if R captures all the requirements information of R'. Also, a program P is correct with respect to a specification R if and only if the program's function refines R. To further convey the meaning of the refinement ordering, we note that R refines R' if and only if any program that is correct with respect to R is (a fortiori) correct with respect to R'.

In addition to its ordering properties, the refinement relation also has lattice-like properties [2]. Two specifications R and R' are said to be *consistent* if they can be refined simultaneously; in relational terms, this is written as:

$$RL \cap R'L = (R \cap R')L.$$

The reason why consistency is important for the purpose of lattice properties is that only consistent relations admit a *join* (least upper bound). If R and R' are consistent then they admit a join with respect to the refinement ordering, which is denoted by $R \sqcup R'$ and defined by

$$R \sqcup R' = \overline{RL} \cap R' \cup \overline{R'L} \cap R \cup R \cap R'.$$

The join captures all the requirements information in R and all the requirements information in R' (upper bound) and nothing more (least). Whereas the join is conditional, the meet is not: any two relations R and R' have a meet (greatest lower bound), which is denoted by $R \sqcap R'$ and defined by:

$$R \sqcap R' = RL \cap R'L \cap (R \cup R').$$

The meet represents all the requirements information that is captured simultaneously by R and R'.

2.2 Composing Dependability Claims

In this section we use the lattice-like structure of the refinement ordering to discuss how to compose dependability claims. Specifically, we consider a system that we have statically verified for some correctness criterion, that we have tested against some functional oracle using some test data, and that we have made fault tolerant by appropriate assertions and recovery routines, the question we wish to ask is: How can we add up the individual claims that each measure allows us? What claims do these measures, combined together, allow us to make? How do we know whether the measures we have taken (testing, proving, fault tolerance) are complementing each other, or whether they are testing the same aspects over and over again?

To answer these questions, we have (in [6]) proposed a common refinement based model, in which we cast all three families of methods (static verification, testing, and fault tolerance); we have shown that all three methods can be interpreted as establishing that the system being analyzed refines some specification, that depends on the method and the method's parameters. Using the join operator, we can then compose eclectic measures, stemming from different methods, into a single claim. Specifically, we discuss below, briefly, how we interpret all three families of methods by means of refinement. We let P be the program that we are interested in.

- *Static Verification*. If we prove by static analysis that P is correct with respect to some specification V, we represent this claim by:
$$P \sqsupseteq V.$$
- *Testing*. We assume that we have tested program P on test data D using oracle Ω, and that all tests have been executed successfully (if not, we redefine D), we claim that this allows us to write:
$$P \sqsupseteq_D \backslash\Omega,$$
where ${}_D\backslash\Omega$ represents the (pre) restriction of Ω to D. Details can be found in [6].
- *Fault Tolerance*. If we test some condition C at run-time, and whenever the condition does not hold we invoke a recovery routine W, then we can claim that:
$$P \sqsupseteq C \sqcap W,$$
where we take the liberty to use the same symbol (C) to represent the condition and the relation that represents it, and to use the same symbol (W) to represent the recovery routine and the relation that represents it. Because we do not know for each execution whether C holds or not, we do not know whether we can claim $P \sqsupseteq C$ (if C holds) or $P \sqsupseteq W$ (if C does not hold). Since we are assured that P refines at least one of them at each execution, we know that it refines their meet.

From static analysis, we infer: $P \sqsupseteq V$. From (certification) testing, we infer: $P \sqsupseteq T$, where $T =_D \backslash\Omega$. From fault tolerance, we infer: $P \sqsupseteq F$, where $F = C \sqcap W$. From lattice theory, we infer:
$$P \sqsupseteq (V \sqcup T \sqcup F).$$

2.3 Decomposing Dependability Goals

A more interesting application of the lattice of refinement involves decomposing a complex dependability goal into simpler sub-goals. Imagine that we must prove that some product P refines a complex specification R, and imagine that R is structured as the join of several simpler sub-specifications, say R_1, R_2, ... R_k; we had shown in [2] that the join offers a natural mechanism to structure complex specifications as aggregates of simpler specifications. Lattice properties provide that in order to prove $P \sqsupseteq R$, it suffices to prove $P \sqsupseteq R_i$ for all i.

We consider the question: which method (among static verification, testing, fault tolerance) is best adapted for each sub-specification R_i. This question is discussed in some detail and illustrated in [6]. We summarize it briefly here:

- *Static Verification*. Ideal candidates for static verification are relations that are reflexive and transitive. Indeed, static verification usually revolves around inductive arguments (of loops, recursive calls); the reflexivity of the specification makes the basis of induction trivial, and the transitivity of the specification makes the induction step trivial. What makes static verification very difficult in general is the need to invent or guess invariant assertions and intermediate assertions; when the specification at hand is reflexive and transitive, it can be used as a sufficient (i.e. sufficiently strong) assertion throughout the program. Verifying programs against reflexive transitive specifications is so straightforward, it can actually be readily automated.
- *Testing*. Ideal specifications for testing are relations that can be coded reliably, as we do not want a faulty oracle to mislead the whole testing process. Because testing is done off-line (in the sense: not during the normal operation of the system), execution

efficiency is not a major consideration (by contrast with executable assertions), but reliability of the oracle is.

- *Fault Tolerance*. Ideal specifications for fault tolerance are unary relations, i.e. relations that refer to the current state but not to past states (for example, in a sorting program, checking that the current array is sorted is a unary property, while checking that the current array is a permutation of the initial array is a binary property). What makes fault tolerance techniques inefficient is the need for saving past states (memory overhead) and for checking the correctness of current states with respect to past states (CPU overhead). With unary specifications, we are spared both of these overheads.

In [6] we show an example of application where the overall verification effort of a program with respect to a compound specification is significantly smaller than the effort of applying any one of the methods.

2.4 Extensions of the Model

In this paper we extend out previous work in three orthogonal directions:

- First, by replacing the logical claims of the original model with probabilistic claims, on the grounds that all methods, even formal methods, produce claims with associated degrees of (un)certainty, and with associated implicit conditions, which probability theory is equipped to capture and reason about.

- Second, by expanding the model to include, not only reliability claims, but also claims dealing with safety and security, on the grounds that these claims are interdependent, and that from the user's standpoint it does not matter whether the failure of a system is due to faulty design or to malicious activity.

- Third, by integrating *failure cost* into the equation, on the grounds that a complex specification typically has many components, whose failures carry widely varying costs, that we must account for in a differential manner.

This paper does not produce results in the sense of solutions that are analyzed, validated and deployed; rather, it offers motivated ideas and proposals, that serve as a launching pad for further research.

3 A Unified Representation

In this section we critique the model presented in the previous section, then propose a generalization that addresses some of its shortcomings.

3.1 The Need for Generalization

In order to motivate the need for generalizing the model presented in the previous section, we briefly discuss why it is inadequate, as it stands.

- Most dependability measures are best modeled as probabilistic claims rather than firm logical claims.

- Most claims are contingent upon implicit conditions. For example, testing is contingent upon the condition that the testing environment is a faithful simulation of the operating environment (or, more precisely, that it is at least as harsh as the operating environment). Also, static verification is contingent upon the condition that the verification rules used in the static proof are borne out by the compiler and the operating environment. Also, fault tolerance is contingent upon the condition that the assertion-checking code and the recovery code are free of faults.

- Many claims may lend themselves to more than one interpretation. For example, if we test P against oracle Ω using test data D, we can interpret this in one of two ways: either that P refines ${}_D\backslash\Omega$ with probability 1.0 (subject to the hypothesis discussed above, that the testing environment subsumes the operating environment); or that P refines Ω (not restricted to D this time), subject to the subsumption hypothesis, and to the hypothesis that D is a faithful representative of the program's domain (i.e. P fails on D if and only if it fails on the whole domain), with some probability p less than 1.0. While the logic, refinement based, model discussed in section 2 represents only the first interpretation, the probabilistic model can represent both. In addition, we will see how the proposed model allows us to keep both interpretations, and makes use of them both (which is only fair, since they are both plausible interpretations).

- If we admit the premise that dependability claims are probabilistic, we must now consider *failure costs*. It is not enough to know that P refines R_i with some probability p_i, over some period of operational time; we must also know what costs we will incur in the case (probability $(1-p_i)$) that P fails to refine R_i during that time.

- While the refinement ordering proves to be adequate for representing reliability claims and safety claims, as we will discuss subsequently, it is not adequate for representing security claims. We wish to generalize the form that specifications can take, and consequently also generalize the concept of refinement to capture security properties.

3.2 A Generalized Model

We submit the premise that dependability methods can be characterized by the following features:

- **Property**. This feature represents the property that we want to establish about P: In section 2 we were interested exclusively in refinement, but it is possible to imagine other properties, such as performance (with respect to performance requirements), security (with respect to security requirements), recoverability preservation, etc.
- **Reference**. This feature represents the reference with respect to which we are claiming the property cited above. This can be a functional specification (if the property is correctness, or recoverability preservation), an operational specification (if the property is a performance property), or a security specification (if the property is a security property), etc.
- **Assumption**. This is the condition assumed by the verification method; all verification methods are typically based on a set of (often) implicit assumptions, and are valid only to the extent that these assumptions hold. We propose to make these assumptions explicit, so that we can reason about them.
- **Certainty**. This feature represents the probability with which we find that the property holds about P with respect to the reference, conditional upon the Assumption. The same dependability measure (e.g. testing P with respect to some oracle using some test data, proving a refinement property with respect to some specification, etc) can be interpreted in more than one way, possibly with different probabilities.
- **Failure Cost**. Safety and security requirements are usually associated with costs, which quantify the amount of loss that results from failing to meet them. Safety costs may include loss or endangerment of human life, financial loss, endangerment of a mission, etc. Security costs may include disclosure of classified information, loss of availability, exposure of personal information, etc. The purpose of this feature is to quantify this cost factor, and associate it explicitly with the failure that has caused it.
- **Verification Cost**. Verification costs complement the information provided by failure costs, by quantifying how much it costs to avoid failure, or reduce the probability of failure. Together these two functions help manage risks and risk mitigation.

To reflect this characterization, we represent dependability claims as follows:

$$\Pi(P \sqsupseteq R|A) = p,$$

where P is the product, $\sqsupseteq$ is the property we claim about it, R is the specification against which we are making the claim, A is the assumption under which we are making the claim, and p is the probability with which we are making the claim. We further add two cost functions:

- *Failure Cost*: This function (which we denote by ϕ) maps a property (say, $\sqsupseteq$) and a reference (say, some specification R) into a cost value (quantified in financial terms, or in terms of human lives at risk, etc). Hence

  $$\phi(\sqsupseteq, R)$$

  represents the cost that we expect to incur whenever a candidate system P fails to satisfy property $\sqsupseteq$ with respect to R.
- *Verification Cost*: This function (which we denote by ν) maps a property (say, $\sqsupseteq$), a reference (say, R), an assumption (say, A) and a method (say, M), to a cost value (expressed in Person Months). Hence

  $$\nu(\sqsupseteq, R, A, M)$$

  represents the cost of applying method M to prove that P satisfies property $\sqsupseteq$ with respect to R under the assumption A.

Although this model appears on the face of it to deal only with claims that pertain to the whole system P, we can in fact use it to represent verification steps taken on components of P [10]. We use an illustrative example: We let P be the composition of two components, say P_1 and P_2, and we assume that we have used some method to establish the following claim:

$$\Pi(P_1 \sqsupseteq R_1|A) = p.$$

We submit that this can be written as a property of P (rather than merely a property of P_1) if we add an assumption about P_2. Hence, for example, we can infer a claim of the form

$$\Pi(P \sqsupseteq (R_1 R_2)|A \wedge P_2 \sqsupseteq R_2) = p',$$

for some reference R_2 and some probability p'. We submit that this model enables us to collect every piece of information that we can derive from dependability measures, so that all the verification effort that is expended on P can be exploited (to support queries, as we will discuss in section 5).

3.3 Implications of the Model

The first implication of this probabilistic model is that verification claims are no longer additive, in the sense that we discussed in section 2. While in the logic, refinement-based model we could sum up all our claims in a single refinement property, in the new probabilistic model it is generally not possible to do so. Nor is it desirable, in fact, as the result would probably be so complex as to be of little use. What we advocate instead is to use an inference system where all the collected claims can be stored, and subsequently used to answer queries about the dependability of the system. This will be illustrated in section 5 through a simple example.

The second implication of this model is that it allows us to introduce a measure of dependability that integrates cost information. When we say that a system P has a given MTTF, it is with respect to some implicit specification, say R. It is also with respect to some implicit understanding of failure cost, i.e. how much we stand to lose if our system fails to satisfy R. If we consider that R is an aggregate of several sub-specifications, say R_1, R_2, ... R_k, it is conceivable that the components R_1, R_2, ... R_k have different failure costs associated with them; for example, failing to refine R_1 will cost significantly more than failing to refine R_k, but the MTTF does not reflect this, as it considers both as failures to refine R. We introduce the concept of *Mean Failure Cost* (MFC), which combines terms of the form

$$\Pi(\overline{P \sqsupseteq R_i}) \times \phi(\sqsupseteq, R_i)$$

where the term $\Pi(\overline{P \sqsupseteq R_i})$ represents the probability that P fails to refine R_i and $\phi(\sqsupseteq, R_i)$ represents the cost that we incur when it does.

4 Modeling Security

In order to integrate security into the refinement model discussed above, and take advantage of its capability in terms of composing claims and decomposing goals, we must formulate security properties in refinement-like terms. In [8] Nicol et al. discuss a number of dimensions of security, including: *data confidentiality, data integrity, authentication, survivability, non-repudiation,* etc. In the context of this paper, we focus our attention on *survivability*, and readily acknowledge a loss of generality; other dimensions of security are under investigation. Survivability is defined in [3] as the capability of a system to fulfill its mission in a timely manner, in the presence of attacks, failures, or accidents [8]. We discuss in turn how to represent security (survivability) requirements, and how to represent the claim that a system meets these security requirements. In the sequel we discuss in turn two aspects of security: *survivability* and *integrity*. The modeling of other aspects is under investigation.

4.1 Modeling Survivability

4.1.1 Specifying Survivability Requirements

We note that there are two aspects to survivability: the ability to deliver some services, and the ability to deliver these services in a timely manner; to accommodate these, we formula security requirements by means of two relations, one for each aspect. Using a relational specification model presented in [2] we propose to formulate functional requirements as follows:

- An input space, that we denote with X; this set contains all possible inputs that may be submitted to the system, be they legitimate or illegitimate (part of an attack/ intrusion).
- Using space X, we define space H, which represents the set of sequences of elements of X; we refer to H as the set of *input histories* of the specification. An element h of H represents an input history of the form

$$..h_N.h_{N-1}...h_3.h_2.h_1.h_0,$$

  where h_0 represents the current input, h_1 represents the previous input, h_2 represents the input before that, etc.
- An output space Y, which represents all possible outputs of the system in question.
- A relation ϕ from H to Y that specifies for each input history h (which may include intrusion/ attack actions) which possible outputs may be considered correct (or at least acceptable). Note that ϕ is not necessarily deterministic, hence there may be more than one output for a given input history. Note also that this relation may be different from relation R which specifies the normal functional requirements of the system: while R represents the desired functional properties that we expect from the system, ϕ represents the minimal functional properties we

must have even if we are under attack; hence while it is possible to let $\phi = R$, it is also possible (perhaps even typical) to let there be a wide gap between them.

As for representing timeliness requirements, we propose the following model:

- The same input space X, and history space H.
- A relation from H to the set of positive real numbers, which represents for each input history h the maximum response time we tolerate for this input sequence, even in the presence of attacks. We denote this relation by ω.

In the sequel, we discuss under what condition do we consider that a system S satisfies the security requirements specified by the pair (ϕ, ω).

4.1.2 Certifying Survivability Properties

Given a survivability requirements specification of the form (ϕ, ω), we want to discuss under what condition we consider that a program S that takes inputs in X and produces outputs in Y can be considered to satisfy these survivability requirements. Space limitations preclude us from a detailed modeling of attacks/ intrusions, hence we will, for the purposes of this paper, use the following notations:

- Given a legitimate input history h, we denote by $v(h)$ an input history obtained from h by inserting an arbitrary intrusion sequence (i.e. sequence of actions that represent an intrusion into the system).
- Given an input history h (that may include intrusion actions) we denote by $\theta(S, h)$ the response time of S on input history h.

Using these notations, we introduce the following definition.

Definition 1 *A system S is said to be* secure *with respect to specification (ϕ, ω) if and only if*

1. *For all legitimate input history h,*

$$(h, S(h)) \in \phi \Rightarrow (v(h), S(v(h))) \in \phi.$$

2. *For all legitimate input history h,*

$$\theta(S, h) < \omega(h) \Rightarrow \theta(S, v(h)) < \omega(h).$$

The first clause of this definition can be interpreted as follows: if system S behaves correctly with respect to ϕ in the absence of an intrusion, then it behaves correctly with respect to ϕ in the presence of an intrusion. Note the conditional nature of this clause: we are not saying that S has to satisfy ϕ at all times, as that is a reliability condition; nor are we saying that S has to satisfy ϕ in the presence of an intrusion, as we do not know whether it satisfies in the absence of an intrusion (surely we do not expect the intrusion to improve the behavior of the system —all we hope for is that it does not degrade it). Rather we are saying that if S satisfies ϕ in the absence of an intrusion, then it satisfies it in the presence of an intrusion.

The second clause articulates a similar argument, pertaining to the response time: if the response time of S was within the boundaries set by ω in the absence of an intrusion, then it remains within those bounds in the presence of an intrusion.

At the risk of overloading the refinement symbol ($\sqsupseteq$), we resolve to use it to represent the property that a system P is secure (according to the definition above) with respect to a survivability specification (ϕ, ω). The form of the specification, when it is explicit, resolves the ambiguity. Hence we write

$$P \sqsupseteq (\phi, \omega)$$

to mean that P is secure with respect to (ϕ, ω).

4.1.3 Integrating Survivability

The definition that we propose here is focused entirely on effects rather than causes, and gives meaning to the concept of *survivability failure*. Using this concept, we can now quantify security by adding terms of the form

$$\Pi(\overline{P \sqsupseteq R}) \times \phi(\sqsupseteq, R)$$

to the mean failure cost, producing a function that quantifies the expected failure cost, without distinction on whether the failure is due to a design fault (reliability, safety) or a an intrusion (security). In [12] Stevens et al. present measures of security in terms of MTTD (D: vulnerability discovery) and MTTE (E: exploitation of discovered vulnerability). By contrast with our (re) definition, these definitions are focused on causes (rather than effect); in fairness, Stevens et al. propose them as intruder models rather than security models. The difference between our effect-based measure and Stevens' cause-based measure is that a vulnerability may be discovered without leading to an intrusion, and an intrusion may be launched without leading to a security failure in the sense of our definition.

4.2 Modeling Integrity

4.2.1 Specifying Integrity

A requirement for integrity refers, generally, to a system's state, and stipulates limits within which the system state may vary as the system is running. The system model that we take in this section represents the system by its state space (which we refer to as Σ), and operations that interact with the outside world, possibly affecting and being affected by the internal state space. We assume that the set of operations (which we refer to as Ω) includes an initialization operation, which we call *init*.

Given this model, we submit that an integrity requirement is specified by providing a subset Σ' of Σ. This subset characterizes all the internal states that we consider to meet the integrity requirement.

4.2.2 Certifying Integrity

We consider a system S defined by a state space Ω and a set of operations Ω. And we consider an integrity specification defined by Ω'.

Definition 2 *We say that system S meets the integrity specification Σ' if and only if the following conditions are met:*

1. *Operation init satisfies the following condition:*
$$\forall \sigma \in \Sigma : init(\sigma) \notin \Sigma' \Rightarrow Alarm(\sigma).$$
2. *All other operations w in W satisfy the following condition:*
$$\forall \sigma \in \Sigma : \sigma \in \Sigma' \wedge \omega(\sigma) \notin \Sigma' \Rightarrow Alarm(\sigma).$$

This definition can be interpreted as follows: According to Schneider et al [11], the condition of integrity is met if and only if integrity constraints are not violated in an undetected manner. If the condition were simply, *are not violated*, then our definition would stipulate that init maps the system state inside Σ' and subsequent operations (ω) keep it inductively in Σ'. Because Schneider et. al. allow for violation of the condition, provided the violations are detected, our definition merely specifies that if either the basis of induction or the induction step does not hold, an exception (which we represent by $Alarm(\sigma)$) is raised.

5 Illustration: Dependability Queries

In the previous section we discussed how we can represent dependability claims in a unified model; in this section, we briefly discuss how to deploy claims represented in this manner to support queries. We will first discuss, in broad terms, some inference rules; then we show a sample example of illustration.

5.1 Inference Rules

We envision a database in which we accumulate all the verification claims that we obtain, from various methods, applied to various components (though typically the whole system), against various specifications (functional specifications, security specifications, etc), reflecting various properties (correctness, security, recoverability preservation, etc). Queries are submitted to this database and inference rules allow us to determine how to answer the query in light of available claims. We classify inference rules into a number of categories:

- *Probability Rules*. This category includes all the rules that stem from probability theory, including especially identities that pertain to conditional probability.
- *Refinement Rules*. This category includes all the rules that stem from the partial order structure of the refinement ordering. For example, if R refines R', then we know, by transitivity of the refinement ordering, that
$$\Pi(P \sqsupseteq R|A) \leq \Pi(P \sqsupseteq R'|A).$$
- *Lattice Rules*. This category includes all the rules that stem from the lattice structure of the refinement ordering. For Example, if R_1 and R_2 are specifications that admit a join, we have
$$\Pi(P \sqsupseteq (R_1 \sqcup R_2)|A) \geq \Pi(P \sqsupseteq R_1) \times \Pi(P \sqsupseteq R_2).$$
- *Conversion Rules*. This category includes the rules that reflect relationships between the various properties that we wish to claim (correctness, recoverability preservation, security, etc). Examples of relations that we capture by these rules include: the fact that if P is correct with respect to R, it is recoverability-preserving with respect to R; the fact that the security of P is contingent upon the correctness of the components that enforce its security policies; etc.

5.2 A Tool Prototype

We have developed a *very* sketchy prototype of a tool that stores claims and supports queries. In its current form, the prototype includes only *probability rules*, hence has very limited capability. Nevertheless, it allows us to discuss our vision of its function and its operation. The first screen of the prototype offers the following options:

- *Record a Reliability/ Safety Claim*. Clicking on this tab prepares the system for receiving details about a dependability claim (reliability, safety, etc) with respect to a functional specification. Given that such claims have the general form:

  $$\Pi(P \sqsupseteq R|A) = p,$$

  the system prompts the user to fill in fields for the property ($\sqsupseteq$), the reference (R), the assumption (A), and the probability (p).

- *Record a Security Claim*. Clicking on this tab presents an entry screen that prompts the user for a security specification (two fields: a functional requirement and an operational requirement 4.1.1), a field for an assumption, and a field for a probability. There is no need for a *property* field, since the property is predetermined by the choice of tabs.

- *Record Cost Information*. As we recall, there are two kinds of cost information that we want to record: failure cost, and verification cost. Depending on the user's selection, the system presents a spreadsheet with four columns (Property, Reference, Cost, Unit —for failure cost), or six columns (Property, Reference, Method, Assumption, Cost, Unit —for verification cost). This information is stored in tabular form to subsequently answer queries on failure costs or verification costs.

- *Record Domain Knowledge*. Because dependability claims are formulated using domain-specific notations, a body of domain-specific knowledge is required to highlight relevant properties and relationships, and to enable the inference mechanism to process queries. This domain knowledge is recorded by selecting the appropriate tab on the system.

- *Queries*. Clicking on the tab titled *Submit Query* prompts the user to select from a list of query format. The only format that is currently implemented is titled *Validity of a Claim*, and its purpose is to check the validity of a claim formulated as

  $$\Pi(P \sqsupseteq R|A) \geq p,$$

  for some property $\sqsupseteq$, reference (Specification) R, Assumption A, and probability p. Notice that we do not have equality, but inequality; this feature can be used if we have taken a number of dependability measures and wish to check whether they are sufficient to allow us to claim that P refines R with a greater certainty than a threshold probability p.

To answer a query, the system composes a theorem that has the query as goal clause, and uses recorded dependability claims and domain knowledge as hypotheses. The theorem prover we have selected for this purpose is *Otter* [5].

5.3 A Sample Demo

To illustrate the operation of the tool, we take a simple example. We will present, in turn, the dependability claims that we submit to this system, then the domain knowledge, and finally the query; this example is totally contrived and intends only to illustrate what we mean by composing diverse dependability claims. Also, even though the model that we envision has inference capabilities that are based on many types of rules (probabilistic identities, refinement rules, lattice identities, relations between various refinement properties, etc), in this demo we only deploy probabilistic rules.

For the purposes of this example, we summarily introduce the following notations, pertaining to a fictitious nuclear power plant:

- *Specifications*. We consider a specification, which we call *SafeOp*, which represents the requirement that the operation of the reactor is safe. We also (naively) assume that this requirement can be decomposed into two sub-requirements, whose specifications, *CoreTemp* and *ExtRad*, represent requirements for safe core temperatures and safe external radiation levels.

- *Assumptions*. We assume (artificially) that the claims we make about refining specifications *CoreTemp* and *ExtRad* are contingent upon a combination of conditions that involve two predicates: *FireWall*, which represents the property that the system's firewall is operating correctly; and *ITDetection*, which represents the property that the system's Insider Threat Detection is working properly.

Using these notations, we illustrate the deployment of the tool by briefly presenting the security claims, the domain knowledge, then the query that we submit to it.

- *Claims*. Using the system's GUI screens, we enter the following claims, where P represents the reactor's control system:

  $$\Pi(P \sqsupseteq CoreTemp|FireWall) = 0.98.$$

  $$\Pi(P \sqsupseteq CoreTemp|(\neg FireWall \wedge ITDetection))$$

$$= 0.95.$$

$$\Pi(P \sqsupseteq CoreTemp|(\neg FireWall \wedge \neg ITDetection)) = 0.93.$$

$$\Pi(P \sqsupseteq ExtRad|FireWall) = 0.95.$$

$$\Pi(P \sqsupseteq ExtRad|\neg FireWall) = 0.90.$$

- *Domain Knowledge*. We submit the following domain knowledge under the form of predicates, where *indep(p,q)* means that events p and q are independent; one could questions whether some of the claims of independence are well-founded, but we make these assumptions for the sake of simplicity.

$$indep(FireWall, ITDetection).$$

$$indep(P \sqsupseteq CoreTemp, P \sqsupseteq ExtRad).$$

$$P \sqsupseteq SafeOp \Leftrightarrow (P \sqsupseteq CoreTemp \wedge P \sqsupseteq ExtRad).$$

- *Query*. We submit the query whether the following claim

  $$\Pi(P \sqsupseteq SafeOp|A) \geq 0.90,$$

  is valid, where A is the assumption that the probability of *FireWall* is 0.90 and the probability of *IT-Detection* is 0.80.

The system generates a theorem and submits it to Otter; then it analyzes the output file to determine if a proof was produced. The claim is deemed to be valid.

6 Conclusion

In this paper we have considered past work that attempts to compose eclectic verification claims and decompose aggregate verification goals, and have attempted to extend it. We have attempted to extend it by encompassing more dimensions of dependability, acknowledging the probabilistic nature of claims and goals, integrating failure and verification costs, and highlighting relationships between diverse dimensions of dependability. Also, we have defined two aspects of security, namely survivability and integrity, and explored how these definitions allow us to integrate security concerns within the broader refinement based model of dependability. We envision to continue this work by modeling other aspects of security, and by exploring how a computational approach, based on observed security attributes (rather than hypothesized causes) may lead in practice to better security management. A prototype that we have developed to illustrate our approach shows how we can store various dependability claims, stemming presumably from distinct validation efforts, into a database, then query the database to see if the available claims warrant specific properties we are interested in.

References

[1] E.N. Adams. Optimizing preventive service of software products. *IBM Journal of Research and Development*, 28(1):2–14, 1984.

[2] N. Boudriga, F. Elloumi, and A. Mili. The lattice of specifications: Applications to a specification methodology. *Formal Aspects of Computing*, 4:544–571, 1992.

[3] R.J. Ellison, D.A. Fisher, R.C. Linger, H.F. Lipson, T. Longstaff, and N.R. Mead. Survivable network systems: An emerging discipline. Technical Report CMU/SEI-97-TR-013, CMU Software Engineering Institute, november 1997.

[4] R.C. Linger. Cleanroom process model. *IEEE Software*, 11(2):50–58, 1994.

[5] William McCune. Otter 3.3 reference manual. Technical Report Technical Memorandum No 263, Argonne National Laboratory, August 2003.

[6] A. Mili, B. Cukic, T. Xia, and R. Ben Ayed. Combining fault avoidance, fault removal and fault tolerance: An integrated model. In *Proceedings, 14th IEEE International Conference on Automated Software Engineering*, pages 137–146, Cocoa Beach, FL, October 1999. IEEE Computer Society.

[7] H.D. Mills and M. Dyer et al. Cleanroom software engineering. *IEEE Software*, 4(5):19–25, 1987.

[8] David M. Nicol, William H. Sanders, and Kishor S. Trivedi. Model based evaluation: From dependability to security. *IEEE Transactions on Dependable Computing*, 1(1):48–65, 2004.

[9] S.J. Prowell, C.J. Trammell, R.C. Linger, and J.H. Poore. *Cleanroom Software Engineering: Technology and Process*. SEI Series in Software Engineering. Addison Wesley, 1999.

[10] YanSong Ren, Rick Buskens, and Oscar Gonzales. Dependable initialization of large scale distributed software. Technical report, AT& T Bell Labs, December 2004.

[11] F.B. Schneider, editor. *Trust in Cyberspace*. National Academy Press, 1998.

[12] Fabrice Stevens, Tod Courtney, Sankalp Singh, Adnan Agbaria, John F Meyer, William H Sanders, and Partha Pal. Model based validation of an intrusion tolerant information system. In *Proceedings, SRDS*, pages 184–194, 2004.