Arguing Security: A Framework for Analyzing Security Requirements

Charles B. Haley BA, MS

A thesis submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science

Department of Computer Science
Faculty of Mathematics and Computer Science
The Open University

March 2007

Abstract

When considering the security of a system, the analyst must simultaneously work with two types of properties: those that can be shown to be true, and those that must be argued as being true. The first consists of properties that can be demonstrated conclusively, such as the type of encryption in use or the existence of an authentication scheme. The second consists of things that cannot be so demonstrated but must be considered true for a system to be secure, such as the trustworthiness of a public key infrastructure or the willingness of people to keep their passwords secure. The choices represented by the second case are called trust assumptions, and the analyst should supply arguments explaining why the trust assumptions are valid.

This thesis presents three novel contributions: a framework for security requirements elicitation and analysis, based upon the construction of a context for the system; an explicit place and role for trust assumptions in security requirements; and structured satisfaction arguments to validate that a system can satisfy the security requirements. The system context is described using a problem-centered notation, then is validated against the security requirements through construction of a satisfaction argument. The satisfaction argument is in two parts: a formal argument that the system can meet its security requirements, and structured informal arguments supporting the assumptions exposed during argument construction. If one cannot construct a convincing argument, designers are asked to provide design information to resolve the problems and another pass is made through the framework to verify that the proposed solution satisfies the requirements. Alternatively, stakeholders are asked to modify the goals for the system so that the problems can be resolved or avoided. The contributions are evaluated by using the framework to do a security requirements analysis within an air traffic control technology evaluation project.

Author's Declaration

Much of the material in this thesis appears in the following papers & books.

- Haley, C. B., Laney, R. C., Moffett, J. D., & Nuseibeh, B. (2003). "Using Trust Assumptions in Security Requirements Engineering," in *The Second Internal iTrust Workshop On Trust Management In Dynamic Open Systems*. Imperial College, London, UK, 15-17 Sep.
- Haley, C. B., Laney, R. C., Moffett, J. D., & Nuseibeh, B. (2004). "Picking Battles: The Impact of Trust Assumptions on the Elaboration of Security Requirements," in *Proceedings of the Second International Conference on Trust Management (iTrust'04)*, vol. 2995. St Anne's College, Oxford, UK: Lecture Notes in Computer Science (Springer-Verlag), 29 Mar - 1 Apr, pp. 347-354.
- Haley, C. B., Laney, R. C., & Nuseibeh, B. (2004). "Deriving Security Requirements from Crosscutting Threat Descriptions," in *Proceedings of the Third International Conference on Aspect-Oriented Software Development (AOSD'04)*. Lancaster, UK: ACM Press, 22-26 Mar, pp. 112-121.
- Moffett, J. D., Haley, C. B., & Nuseibeh, B. (2004). "Core Security Requirements Artefacts," Department of Computing, The Open University, Milton Keynes, UK, Technical Report 2004/23, June.
- Haley, C. B., Laney, R. C., Moffett, J. D., & Nuseibeh, B. (2004). "The Effect of Trust Assumptions on the Elaboration of Security Requirements," in *Proceedings of the 12th International Requirements Engineering Conference (RE'04)*. Kyoto, Japan: IEEE Computer Society Press, 6-10 Sep, pp. 102-111.
- Haley, C. B., Moffett, J. D., Laney, R., & Nuseibeh, B. (2005). "Arguing Security: Validating Security Requirements Using Structured Argumentation," in *Proceedings of the Third Symposium on Requirements Engineering for Information Security (SREIS'05) held in conjunction with the 13th International Requirements Engineering Conference (RE'05)*. Paris, France, 29 Aug.
- Haley, C. B., Laney, R. C., Moffett, J. D., & Nuseibeh, B. (2006). "Using Trust Assumptions with Security Requirements," Requirements Engineering Journal, vol. 11 no. 2 (April), pp. 138-151.
- Haley, C. B., Laney, R. C., Moffett, J. D., & Nuseibeh, B. (2006). "Arguing Satisfaction of Security Requirements," in *Integrating Security and Software Engineering: Advances and Future Vision*, H. Mouratidis & P. Giorgini, Eds.: Idea Group.
- Haley, C. B., Moffett, J. D., Laney, R., & Nuseibeh, B. (2006). "A Framework for Security Requirements Engineering," in Proceedings of the 2006 Software Engineering for Secure Systems Workshop (SESS'06), co-located with the 28th International Conference on Software Engineering (ICSE'06). Shanghai, China, 20-21 May.

All of the work presented in this thesis describes original contributions of the author, with two exceptions:

- The work on representing security requirements as constraints, described in Chapter 5, was done in collaboration with Jonathan D. Moffett.
- The industrial experience, presented in Chapter 7, by necessity (and perhaps by definition) involved other industrial project members in the construction and evaluation of the contexts, goals, and arguments.

Acknowledgements

I wish to thank my supervisors Bashar Nuseibeh and Robin Laney for their invaluable guidance, reviews, and encouragement to finish this thesis; Jonathan Moffett for his generous support, ideas, and very pertinent (and sometimes pointed) criticisms; and Michael Jackson for his support and for occasionally encouraging me to rejoin the real world.

A special thanks goes to my family: to my wife and colleague Debra Haley for thoughtful discussion, criticism, editorial corrections, and especially for her continuous support and understanding during this process; to my sons David and Steven for their encouragement, their pride in my taking on this challenge, and for finding such humor in "Dad, do your homework!"; and (although they will never know their impact) to my cats Lizzie & Tasha for sitting on my lap, shoulders, or keyboard when I wanted to write, forcing me to think instead.

The financial support of the Leverhulme Trust is gratefully acknowledged, as is the support of the ELeGI EU project number IST-002205.

Finally, I acknowledge the support of The Department of Computing of The Open University. Without the department giving me time and encouragement, this thesis would not have been completed.

Table of Contents

Table of Figures

Chapter 1. Introduction

Over the last few years, reports of software security failures have become commonplace. Statistics from the Software Engineering Institute's CERT Coordination Center, a center of internet security expertise, show that the number of reported application vulnerabilities rose from 171 in 1995 to 5,990 in 2005 (CERT, 2006). The sources of problems are diverse. One source is programming errors; in 2003, one internet worm named Blaster, exploiting a flaw in Microsoft's Windows operating system, reportedly infected approximately 500,000 computers (Gallagher, 2003). "Estimates are that it [Blaster] cost approximately $1.3 billion to correct and in lost productivity" (Ibid). Another source is not looking at security requirements of the complete system. For example, CardSystems Solutions exposed details of some 40 million credit cards by storing unneeded transaction history data where hackers could get to it (Dash, 2005); this visible storage was part of their system but not part of their security planning. The resulting loss has not been disclosed, but is known to be in excess of several millions of dollars (Federal Trade Commission, 2006). These two examples strongly suggest that improving software-based system security would have a significant financial impact.

This thesis addresses the second source of security problems: the failure to consider security requirements of the complete system, or said another way, the failure to obtain *adequate security requirements* for a system. By adequate security requirements, we mean requirements that if respected, lead to a system's security goals being satisfied. Adequate general requirements have been shown to have a very positive impact on the success of projects: for examples see the Standish Group's Chaos reports (Standish Group, 1995, 1999, 2001), and the introduction to Mead et al. (Mead, Hough, & Stehney, 2005). Although the empirical evidence is not yet unequivocal, there is evidence that adequate security requirements will have as positive an impact on system security as adequate general requirements have on system success

(Mead, Hough et al., 2005). The CLASP process (Comprehensive Lightweight Application Security Process), co-authored by John Viega (Viega & McGraw, 2002; Viega, 2005a, b), emphasizes the importance of security requirements, saying that one should "[e]nsure that security requirements have the same level of 'citizenship' as all other 'must haves.'" (Secure Software Inc., 2006)

Before continuing further, we must agree on what is meant by *system*. In this thesis, the word system includes the software, and in addition the people who use the software and all the bits and pieces around the software (computers, printers, etc.). We are dealing with requirements, and this definition of system is consistent with common usage in requirements engineering. For example, Zave and Jackson say that "we use 'system' only to refer to a general artifact that might have both manual and automatic components, such as an 'airline reservation system'." (1997). Van Lamsweerde uses the word similarly: "The target system is not just a piece of software, but also comprises the environment that will surround it; […]." (2000). Going a bit further back, Swartout & Balzer include the pipes & bins in the system when describing their package router example (1982). In summary, we can say that requirements engineering is charged with providing detailed & relevant information about the requirements that a system is to satisfy. Our usage of system is consistent with this.

We claim that the adequacy of security requirements can be evaluated using three criteria. The first criterion is *clarity*: one must have a clear understanding of what the security requirements mean, and their effects within the system context in which they apply. The second is *incorporation of assumptions about behavior*: security requirements must take assumptions about the behavior of objects found in the system into consideration. The third is *satisfaction*: one must be able to determine whether the security requirements satisfy the security goals, and whether the system can satisfy the requirements. We propose three contributions to assist a requirements engineer with developing security requirements that satisfy these criteria. The first is a security requirements framework, incorporating system context and providing a practical definition of security requirements. The second is an explicit place and role for assumptions, concentrating on their role in security requirements satisfaction arguments. The third is the use

of formal and informal structured arguments to validate that a system can satisfy its security requirements. The second and third contributions are incorporated into the first, our security requirements framework, facilitating an understanding of eliciting, validating, and verifying security requirements and other artifacts.

We explore these three criteria in Sections 1.1 – 1.3, following. The contributions are further discussed in Section 1.4 of this introduction and, of course, throughout the remainder of this thesis.

1.1 Criterion One – Clarity of Security Requirements

Security needs arise when stakeholders establish that some objects involved in a system, be they tangible (e.g., cash) or intangible (e.g., information), have value. Such objects are termed *assets* (ISO/IEC, 1999c), and the stakeholders naturally wish to protect themselves from any harm that might come from abuse of these assets. Security goals express this desire, describing the involved asset(s) and the harm to be prevented. The usual approach is to treat these security goals as non-functional requirements. The question to answer is whether this approach results in clear security requirements that respond to the needs of the system.

1.1.1 Security Requirements as Non-Functional Requirements

Security requirements have traditionally been considered to be non-functional quality requirements ((Chung, Nixon, Yu, & Mylopoulos, 2000; Devanbu & Stubblebine, 2000; Firesmith, 2004; Glintz, 2005) and many others), meaning that like other kinds of quality requirements (e.g., performance, usability, cost to run), they do not have simple *yes/no* satisfaction criteria. Instead, one must somehow determine whether a quality requirement has been *satisficed* (satisfied well enough) (Mylopoulos, Chung, & Nixon, 1992). This is difficult in general, and security requirements present some additional challenges. First, once one descends from the very general and obvious statements (e.g., 'the system should be secure'), instead of talking about what is to happen, people tend to think about and express security requirements in terms of things that are to be prevented. Verifying that something is prevented can be likened to

proving a negative; it is very difficult, if not impossible, to show that there are no counter-examples. Second, for security requirements, the tolerances on 'satisfied enough' are much smaller, usually approaching zero; stakeholders want criteria for security requirements to be very close to yes/no. Third, the amount of time and money that stakeholders might be willing to dedicate to satisfying a security requirement can depend on the *risk* and *impact* of a security failure; one cannot justify a large expense to protect something of low direct or indirect value. One must be able to connect specific development & operational expense to the requirements being satisfied, in order to determine cost/benefit information.

Expressing security requirements in a positive sense, similar to functional requirements, would reduce the difficulties described above. Functional requirements describe what is to happen, not what is not to happen, helping the implementers understand what they are to do. Tolerances are (in theory) simpler; functional requirements have binary satisfaction criteria, either the function happens or it does not, and they can have test criteria to determine what 'the function happens' means. The cost of making something happen is easier to measure than the cost of making something not happen, facilitating cost/benefit analysis. Expressing security requirements in the positive sense (what is to happen) would bring similar benefits.

1.1.2 Security Requirements & Context

System context can have a profound effect on both security goals and security requirements. As said earlier, in this thesis the word *system* represents more than the software. We include the environment the software runs within: the people who will use, maintain, and depend on the system; the physical environment the system is to exist within; the operating environment the software runs within; and any systems, computer-based and otherwise, already in place. Security requirements can vary, depending on the context. To illustrate, consider some software intended for use by an executive on his or her desktop computer. The software may or may not have any intrinsic need for security; a spreadsheet program would be a good example. Even though the spreadsheet program may have no intrinsic security goals associated with it, the information the executive manipulates may be confidential, creating a *maintain confidentiality*

security goal for the system, where the system comprises the computer, the office, the spreadsheet program, the executive, and the confidential data. The security goal arises because of how the spreadsheet is used, which is a property of the context within which the program resides. When the system components {computer, office, spreadsheet program, executive} are considered alone, no confidentiality security goal arises. The goal arises only when {confidential data} is added.

Continuing the example, one might consider satisfying the confidentiality goal by adding a security requirement that the system architecture include a locking office door, something completely divorced from the software. Alternatively, one might require that the spreadsheet program should satisfy the goal, perhaps by addition of authentication and encryption. However, these solutions would be inadequate if the executive is in an office that is not soundproofed, and either a) the executive uses a program that reads the information aloud, permitting an attacker to listen without being seen, or b) if the attacker can hear and decode the keystrokes typed on the executive's keyboard (Zhuang, Zhou, & Tygar, 2005). The example shows that properties of the system context that are frequently not considered can have a profound effect on the security of the system.

1.2 Criterion Two – Incorporation of Assumptions about Behavior

When considering system behavior, the requirements engineer must determine which parts of the world are part of the problem, and therefore to be included in the analysis. An extreme view is that every atom in the universe is part of every problem, and therefore an analysis must consider everything made of atoms. As this is clearly impractical, the analyst must choose a subset of *domains* (real-world elements) that s/he considers pertinent (Jackson, 1995, 2001). In so choosing, the analyst defines the system context; it consists of those domains having properties considered relevant to the problem.

When considering security, one factor influencing an analyst's choice about whether or not a domain is relevant is the analyst's set of *trust assumptions* (Viega, Kohno, & Potter, 2001; Viega & McGraw, 2002). Trust assumptions are explicit or implicit choices to trust a domain

will behave as expected. These assumptions can have a significant impact on the security of a system. For example, most analysts implicitly assume that the compiler is not a security risk, and it would never occur to them to include the compiler in the security analysis. In his 1983 Turing award acceptance lecture, Ken Thompson (1984) demonstrated that this assumption might not be justified by showing how a compiler could be a Trojan horse, introducing trapdoors into applications. Viega et al. (2001) claim that "application providers often assume that their code will execute in a non-hostile environment", and then show how this assumption leads to security breaches. Their example shows 'secrets' hidden in code, where the secrets easily can be exposed through examination of the executable file. The Thompson and Viega examples illustrate how the requirements engineer's implicit trust of some domains in the environment can introduce unknown amounts of risk into the system. Viega et al. went as far as to say that "without recognizing all the entities and their trust relationships in a software system during the requirements phase of a project, that project is doomed from the start." (2001)

The voice-reading spreadsheet program example in section 1.1.2 further illustrates the point. The analyst easily could tacitly, and erroneously, consider that the spreadsheet program did not present a security risk, assuming that the office did not leak information, by not considering its use in an office without soundproofed walls. Like context, trust assumptions can have a significant impact on security requirements.

1.3 Criterion Three – Satisfaction of Security Requirements

If one goes to the trouble to produce security requirements for a system, it is reasonable to ask whether the system can satisfy the requirements. The more rigorous the process used to establish satisfaction, the more confidence one can have that the system will be secure. The strongest process is a proof. A weaker alternative to a proof is an argument. A high-quality argument engenders confidence that the requirements will be satisfied. The weaker the argument, the more faith one must have that the result will, in the end, be acceptable.

No analysis of security requirement satisfaction can include every possible domain that could be a part of the system. Every proof or argument will include trust assumptions, at minimum

that the domains not considered will do no harm, and establishment of satisfaction depends upon the validity of these trust assumptions. Rigorous proofs of validity of trust assumptions are hard to come by, because malice and accident must be taken into account. Instead of proving that a trust assumption *is* valid, one instead produces arguments that the trust assumption *should be considered* valid. The argument must be sufficiently convincing, using properties of the system and domains as evidence.

Trust assumption validity arguments are, in effect, sub-arguments of the proof or argument that security requirements are satisfied, and their quality directly affects the validity of the containing argument. The sub-arguments should be an explicit part of establishing satisfaction of security requirements.

1.4 Contributions

As indicated above, this thesis presents three novel contributions aimed at assisting a requirements engineer with developing adequate security requirements:

1. A security requirements framework, incorporating system context, and providing a practical definition of security requirements that have clear yes/no satisfaction criteria. The framework also provides a scaffold for the next two contributions.
2. Further elaboration of trust assumptions, concentrating on their role in security requirements satisfaction arguments.
3. The use of formal and informal structured arguments to validate that a system can satisfy its security requirements.

1.5 Novelty of the Contributions

The three contributions build upon existing work. The discussion of the meaning of *system* on page 14 showed that context is important in requirements engineering. Others assert that security requirements analysis must be placed in a system context, or the analysis will not be complete; see, for example, (Devanbu & Stubblebine, 2000) and (Firesmith, 2003a). Our contribution is a systematic incorporation of context into a framework for security requirements

engineering, and then using the context to discover trust assumptions and to develop the satisfaction arguments.

Trust assumptions are mentioned by name in Viega in (Viega, Kohno et al., 2001; Viega & McGraw, 2002), and are alluded to as simple assumptions in other work (e.g., (Firesmith, 2003a; van Lamsweerde, 2004)). Our contribution is making explicit their role both in determining the size of the context and in security requirement satisfaction arguments.

Satisfaction arguments have appeared in the literature in several guises. For example, *correctness arguments* appear in (Jackson, 2001) and (Hall, Rapanotti, & Jackson, 2005), *satisfaction arguments* in (Attwood, Kelly, & McDermid, 2004), (Hammond, Rawlings, & Hall, 2001), and (Hull, Jackson, & Dick, 2002: pgs 143-158), *adequacy arguments* in (Jackson, 2006), and *safety arguments* in (Kelly, 1999). Our contribution is the extension of these arguments for security, proposing two additional factors that should be considered: trust assumptions within a system context. We further propose that representing security satisfaction arguments by a combination of formal and structured informal arguments leads to significant benefits. The formal arguments provide the yes/no criteria, assuring that the requirements are satisfied, assuming that the trust assumptions are valid. The informal arguments, using a jurisprudence-like style of argumentation, show why the trust assumptions are acceptable. The informal arguments are not proofs, but instead are sufficiently convincing in their context.

1.6 Research Methodology

Our research was piloted by the three classic steps: a) identify gaps through examination of the literature, b) propose ways to fill (some of) the gaps, then c) validate that the gaps are indeed filled. The first two steps presented no particular difficulty beyond the inherent intellectual challenge. The third step was more problematic.

Three options to validate the contributions presented in this thesis were considered:

- Testing the contributions by replaying an existing case study.
- Testing the contributions using constructed examples.

- Testing the contributions in a live industrial project.

We discuss each of these options in turn below.

1.6.1 Option 1 – Validation by Replaying an Existing Case Study

This form of validation would use as its baseline an existing published case study. The project would be run again using the contributions presented in this thesis, and the results compared. A case study to be used in this fashion must meet the following preconditions:

1. The project must have identified security as a success factor.
2. The documentation in the case study for the requirements phase must be at a level sufficient to understand the goals of the project.
3. There must be sufficient information in the case study to permit construction of the system context, to permit use of the contributions of this thesis, and to compare the results.

Despite extensive searches of both the literature and information on the web, we were unable to find a case study that met these requirements. We hypothesize that such case studies are not available because commercial entities are very unwilling to advertise their security failures, and because considering security before system design (e.g., during requirements analysis) is new.

1.6.2 Option 2 – Validation using Constructed Examples

This form of validation requires one to construct a problem where security plays a role, then work through the example problem to show how the contributions presented in this thesis help identify security requirements. The only precondition is that the example shows how the contributions presented in this thesis are used. We use this validation method in this thesis.

Validating using constructed examples has the following strengths:

1. The examples can be constructed to best illustrate the contributions.
2. The contributions can be described in a tutorial fashion.
3. The examples can be perturbed, if needed, to show alternative results.

Using a constructed example has one significant disadvantage: the way the example is constructed may mask problems with the contributions that real examples would make evident.

To help minimize this risk and to provide us with a sanity check on our work, we have published the contributions in several peer-reviewed venues (see the next section). The criticisms received have helped enormously with filling in gaps in the contributions.

1.6.3 Option 3 – Validation by Testing on a Live Project

To validate by testing on a live project, one would in effect do the security requirements work twice, once in the way that the project had intended, and once using the contributions presented in this thesis. The results would then be compared. Successful use of this validation method has several preconditions:

1. The project stakeholders must have identified security as a success factor for the project.
2. The project schedule (start & finish) must be compatible with the PhD research timeline.
3. The complexity of the project must be compatible with the resources available.
4. The project must be willing to dedicate resources sufficient to use the framework presented in Chapter 5 and, in particular, the satisfaction arguments presented in Chapter 6.

No industrial partner available to us had a project that met all the above conditions. In fact, no project met conditions 2 and 4.

Given that no project met all the conditions, we decided to try to validate the contributions in a project that met some of the conditions, using a 'trial' approach as opposed to an experiment that compared the two outcomes. We assumed a consultancy-like role in one project that met conditions 1 and 3, and we were able to show that the contributions could be of value during the design phase; these results are presented in Chapter 7. Unfortunately, the project's duration prevented us from following the project to its completion, so we do not know what use the project made of the information we developed.

1.7 Publication History of Contributions

Much of the material in this thesis has been published, primarily in international venues, with increasing levels of maturity. The publications were peer-reviewed, with the exception of three technical reports.

Trust assumptions, (Chapter 4) are described in (Haley, Laney, Moffett, & Nuseibeh, 2003, 2004a, b, 2006a), and used in (Haley, Moffett, Laney, & Nuseibeh, 2005; Haley, Laney, Moffett, & Nuseibeh, 2006b).

The security requirements framework (Chapter 5) was first described in (Moffett & Nuseibeh, 2003) (a technical report), substantially elaborated in (Moffett, Haley, & Nuseibeh, 2004) (a technical report), and further refined in (Haley, Moffett, Laney, & Nuseibeh, 2006). Synopses of the ideas in Chapter 5 have appeared in (Haley, Laney et al., 2003, 2004a, b; Haley, Laney, & Nuseibeh, 2004c; Haley, Moffett et al., 2005; Haley, Laney et al., 2006a). Threat descriptions were introduced in (Haley, Laney et al., 2004c).

Our work on security satisfaction arguments (Chapter 6) was first published in (Haley, Laney, & Nuseibeh, 2005) (a technical report), and substantially modified and elaborated in (Haley, Moffett et al., 2005; Haley, Laney et al., 2006b).

We note that although the technical report (Moffett, Haley et al., 2004) was not peer reviewed, it has had an impact, as evidenced by being cited by (at least) He (He, 2005), Mead, Hough et al. (2005), and Redwine (2006).

1.8 Structure of this Thesis

This thesis comprises eight chapters. The first is this introduction. Chapter 2 provides background information relied upon in the following chapters. Chapter 3 presents related work, expanding upon the discussion in this chapter. Next, the three contributions are discussed, beginning in Chapter 4 with trust assumptions. Chapter 5 introduces our security requirements framework, describing the framework using a constructed example. A major part of the framework, our security satisfaction arguments, is presented in Chapter 6. Chapter 7 provides the industrial example described earlier, and Chapter 8 finishes with discussion, future work, and concluding remarks.

Chapter 2. Background

Problem frames (Jackson, 2001) are used in this thesis to describe system context for security requirements, and to describe phenomena used in behavior specifications. This section presents some background information on problem frames, along with a discussion of requirements and specifications in a problem frames context.

In addition, this chapter justifies the definitions of some terms used in this thesis, and provides some background material on parallel elaboration of requirements and architecture.

2.1 Problem Frames

Problem frames are used during problem analysis, providing mechanisms for describing the domains in a problem. When using problem frames, the analyst decomposes larger problems into a collection of smaller ones. These subproblems are later recomposed, providing the solution for the original problem.

In a problem frames universe, a requirements engineer describes problems by describing the interaction of *domains* that exist in the *world*. The problem frames notation captures domains in a problem along with the interconnections between them. For example, assume that the requirements elicitation process for a box that protects documents from fire produces the requirement *open the fireproof box when a door-open button is pushed*. The elicitation process tells us that the stakeholders want a system consisting of (at least) a box, a door, and a button.

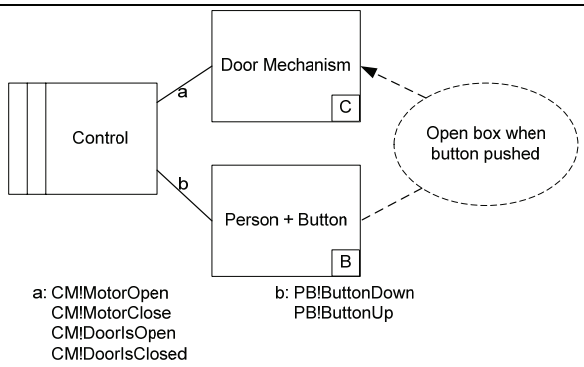

Figure 2-1 – A basic Problem Frames diagram

Figure 2-1 illustrates one set of domains that could satisfy the requirement: a basic automatic door system with three domains, two of which are *given* and one of which is to be *designed*. The first given domain, `Door Mechanism`, is the box's door mechanism domain, capable of opening and shutting the box's door. The second given domain, `Person + Button`, is the one requesting that the door be opened; for convenience this domain includes both the button to be pushed and the human pushing the button. The third domain, `Control`, is a designed domain, indicated by the two vertical lines in the box. It is the *machine*, the domain that will bridge the gap between the other two domains in order to fulfill the requirement that the door open when the button is pushed. The oval presents the requirement that the machine is to satisfy.

In problem frames, every domain has *interfaces*, which are defined by the *phenomena* visible to other domains. Phenomena (e.g., events and signals) are visible: they can be observed. The problem frames notation shows the phenomena shared between two domains on the line between the domains by labeling the line (the 'a' and 'b' in Figure 2-1). The label refers to a set of phenomena on the interface. Phenomena are *controlled* by one of the domains on the interface; the controlling domain is indicated by an abbreviation in front of an exclamation mark. For example, in Figure 2-1, the interface between the Person + Button domain and the Control Machine is labeled 'b'. There are two phenomena on the interface, ButtonDown and ButtonUp, both controlled by the Person + Button domain as indicated by 'PB!'. The Control Machine controls the Boolean phenomena MotorOpen and MotorClose (turn on and off the

motor) on the interface between the machine and the Door Mechanism. The Door Mechanism controls the phenomena DoorIsOpen and DoorIsClosed.

One can think of domains as a set or as a class (a type). When a set, instances of the domain in the running system are members of the set, but might not all be the same type. This way of thinking is attractive in that it permits an object to be a member of multiple sets, which clearly happens in reality (a 'person' can be both a 'user' and an 'administrator'). Unfortunately, sometimes we want to talk about a class of objects (objects with particular attributes & properties), not one of a set (e.g., a particular authenticated user). To solve this problem, one can think of a domain as a class, where all the attributes & properties of interest are defined by the class. When the system is realized (instantiated), objects that are instances of the domain classes interact with each other, and these instances can be named. The downside of choosing the class point of view is that objects can be instances of more than one class, creating something like multiple inheritance. In this thesis, we use both schemes, as appropriate to the situation.

Requirements are optative. They describe *desired* behavior (phenomena: inputs, outputs, and states visible at their interfaces) instead of *existing* behavior (Jackson, 2001). Descriptions of the behavior of designed domains are both optative and indicative. A description is optative in the description where the domain is being designed, but indicative when the designed domain is placed into a system. Descriptions of the behavior of given domains are indicative; they describe an "objective truth" about the behavior of the domain.

2.1.1 Requirements and Specifications

According to Zave and Jackson (1997), a *requirement* is an optative description of what the system is to do. Requirements describe a *desired effect* in terms of phenomena visible in the world. Jackson (2001) describes a requirement as "the effects in the problem domain that […] the machine is to guarantee."

Again referring to Zave & Jackson (1997), *specifications* are about phenomena across all the domains in a problem. The specification of an individual domain is a description of the behavior of the domain in terms of the interplay of its phenomena, indicative and optative, visible at its

interface. The term *interplay* incorporates the notions of sequencing: stimulus, response, and causality. The specification of a system is the collection of domain specifications that together permit the fulfillment of the requirement(s).

The distinction between requirement and specification is an important one. A requirement does not describe how a system is to be implemented, but instead describes what is desired by the stakeholders in terms of phenomena visible at certain domains in the real world. It is the specification that describes how, in terms of the phenomena of all the domains in the system, the requirement is fulfilled. For example, the requirement "given a temperature input in Fahrenheit, the system shall display that temperature in Celsius" is describing some input phenomena on one domain (probably a keyboard) and some output phenomena of another domain: the display; these are the requirement phenomena. The rest of the phenomena in the system exist to make the system produce its output requirement phenomena, given its input requirement phenomena.

Correctness arguments use this correspondence between requirements and specifications. To show that a system correctly satisfies the requirement, one must show that the interplay of specification phenomena causes the requirements phenomena to occur at correct points. If the phenomena are described formally, then the correctness argument can be a proof. If the phenomena are described informally, the correctness argument is equally informal.

It is worth noting that Jackson has recently moved from *correctness arguments* to *adequacy arguments* (2006), which are very similar to the satisfaction arguments described in this thesis. When asked about this shift[1], Jackson explained that when the "real world" is involved, it is not possible to describe *all* possible behavior, especially in the face of failure, and therefore one cannot *prove* correctness. One instead ensures that an adequate number of cases have been considered, and argues why that set of cases is adequate.

[1] Personal communication between the author and Michael Jackson.

2.1.2 Indicative vs. Optative Descriptions

Indicative domain properties are normally expected to be known and constant; the same stimulus in the same context produces the same response. This is what Jackson meant by "objective truth" (2001). Optative domain properties are those one wants; they do not exist yet. Unfortunately, when reasoning about security one should put aside the convenient "indicative properties" concept and assume that all domain properties are optative, because one way an attacker can succeed is by perturbing behavior of domains thought indicative. Consider the pushbutton in the domain shown Figure 2-1; when the button is pushed, the circuit connected to the button is closed. This would seem to be an indicative property. Now put some confidential information in the box, and then consider the same button from the point of view of an attacker. The attacker might cut the wire, connect an alternate or second button to the wire, or put a circuit in the middle that analyzes the context of the button push and either passes it on or does not. The property can no longer be considered objectively true. It has become optative: what one *wants* to be true, or alternatively what *should* be true.

Security requirements are optative, describing characteristics of the system that the requirements engineer desires to be true. The lesson learned from the above discussion is that, unlike functional requirements, security requirements should assume that indicative domain properties are optative, because a goal of an attacker might be to change the behavior of some indicative phenomena. A successful attack means one of two things: that phenomena exist that were not described in the problem, or that behavior (the specification, or interplay of phenomena) assumed to be indicative (to be true), is not.

2.2 Definitions

Software engineering, security requirements, and security engineering have vocabularies that share many terms. Unfortunately, the terms do not always have the same meanings. To help avoid confusion, this section presents how some of the terms are used in this thesis.

2.2.1 Security and Safety

As said in the introduction, this thesis is about security requirements. One question that frequently arises is whether we consider safety when considering security. There is a very close relationship between security and safety requirements. Both deal with system failures that lead to harm. Both deal with analysis of the context to look for evidence about how failures might occur. Both add requirements to reduce the possibility of, or to mitigate the effects of, these failures. We did not wish to consider safety in this thesis, and therefore we needed to find a way to define, or scope, our efforts so that they did not include safety, but equally so that safety would fit in a structure that also includes our contributions. In other words, we needed to find compatible definitions of safety and security.

Some authors say the difference between security and safety is *intention* (e.g., (Firesmith, 2003a; Jonsson, 1998; Leveson, 1986)), and we use this definition. Safety concerns harm caused by accident, while security concerns harm caused intentionally by an attacker. Failures of security can easily lead to safety concerns; consider placing a bomb on an airliner. Equally, failures of safety can lead to security concerns; consider an accident involving a truck carrying unencrypted backup tapes.

The use of intention as a discriminator is not universally agreed. For example, (Avizienis, Laprie, Randell, & Landwehr, 2004) defines security as "the absence of unauthorized access to, or handling of, system state." The paper discusses the role of intention, but does not give it any particular emphasis. This differs from one of its predecessors, which recognizes that security is dominated by intentionally provoked faults (Laprie, 1992). The SafSec methodology (Lautieri, Cooper, & Jackson, 2005) combines safety and security, without introducing intention. The ITSEC defines security as "the combination of confidentiality, integrity and availability [of information]" (Senior Officials Group - Information Systems Security, 1991: pg 115), a much more restricted view of security that does not include intention.

There are several definitions for safety in the standards, and these definitions do not help disambiguate the terms. For example, Avizienis et al. define safety as the "absence of

catastrophic consequences on the user(s) and the environment" (Avizienis, Laprie et al., 2004), a definition that could easily include security-related items such as integrity. IEC 61508-4 defines safety as "freedom from unacceptable risk" (CENELEC, 2002: pg 11), a definition that certainly includes security. Both of these definitions are made clearer by including intention as a differentiator.

Although we recognize that using intention as the differentiator between safety and security is sometimes uncomfortable, we feel that the distinction being made between intention and accident is helpful. It assists with setting bounds on both the context and the mitigations. Consider the possibility of failure of some component in an aircraft, potentially causing the aircraft to crash. Under our definition, this is a safety problem and therefore not considered in our analysis. However, if the component could be provoked to fail, then we have a security problem: preventing the (intentional) actions that could provoke the failure.

2.2.2 Asset, Threat & Vulnerability

These three words are used throughout the security literature, but not always with the same meaning. In this thesis, we use the definitions used by Chivers & Fletcher in (2005) (quoted here):

- Asset: a resource of value to an organization (e.g., hardware, software, data, people).
- Threat: a potential harm that could occur to an asset.
- Vulnerability: a weakness in a system that allows an attack to realize a threat.

These definitions are consistent with those found in (NIST, 1995) and (Mead, Hough et al., 2005).

These definitions are rather different from some proposed elsewhere. One definition has threats confounded with attacks and/or attackers. For example, Firesmith in (2004) defines a threat as "a general condition, situation, or state (typically corresponding to the motivation of potential attackers) that may result in one or more related attacks." ISO 15408, an information security standard, does not define the word 'threat', but it does characterize a threat by "A threat

shall be described in terms of an identified threat agent, the attack, and the asset that is the subject of the attack." (ISO/IEC, 1999a: pg 45).

For another definition, consider the definition in (Breu & Innerhofer–Oberperfler, 2005): "A Threat is defined as any event that can result in the violation of a Security Requirement." Here, threats are defined in terms of security requirements, as opposed to defining security requirements in terms of threats. We do not use this definition because it leaves unsaid what is used to determine security requirements.

The definitions we use have the advantage of clearly separating the concepts of asset, threat, attack, and vulnerability. One need not bring attackers and vulnerabilities into the discussion when considering what harm can follow some abuse of an asset. One can, but need not, speculate about the motivations of an attacker when considering whether a particular vulnerability would permit realization of some threat.

The definitions are discussed further in Chapter 5.

2.2.3 Validation & Verification

As said in Chapter 1, this thesis is about obtaining adequate security requirements. Adequate requirements are testable, in that sufficient criteria are provided to establish that a system satisfies the requirements (e.g., the "fit criteria" in (Robertson & Robertson, 1999)). Establishing satisfaction of requirements is usually considered part of *validation* and *verification*, and therefore one prerequisite to accomplishing our goal is agreement on the definitions of the two terms; what validation and verification are. The purpose of this section is to present and justify the definitions we use for the terms.

It is commonly held in the requirements engineering community that one should be able to determine whether a set of requirements accurately represent the goals/desires of the stakeholders, and to determine whether a delivered system satisfies the requirements. For example, one can determine whether the requirement *When the user enters a temperature in Fahrenheit, the system shall display that temperature in Celsius* is correct by asking the

stakeholders if this is indeed what they want the system to do. The constructed system is checked against the requirement by entering Fahrenheit values and seeing if the correct Celsius values are displayed. This process is an example of validation and verification, but it does not make clear which step is validation and which step is verification.

Unfortunately, the literature does not provide a clear definition of the terms validation and verification. Boehm in (1984) informally defines validation as asking "are we building the right product", and verification as asking "are we building the product right", and Easterbrook in (1996) expands on these definitions, saying that "validation is concerned with checking that the system will meet the customer's actual needs, while verification is concerned with whether the system is well engineered". The IEEE Standard for Software Verification and Validation (IEEE, 1998) defines the words in a way that permits them to be used almost interchangeably, but the notes on the definitions give more precision, saying that validation is "the process of examining a product to determine conformity with user needs" and verification is "the process of examining the result of a given activity to determine conformity with the stated requirement for that activity" (pg 71). Caughlin, writing about simulation, defines the terms similarly to these notes (2000). Pemberton & Sommerville, in a paper about testing, use (and justify) similar definitions (1997).

Another example of the use the terms validation and verification is found in (Soudah, Pilch, Doebling, & Nitta, 2004). Quoting the relevant paragraph: "V&V is the multi-disciplinary process of demonstrating credibility in simulation results. Credibility is built by collecting evidence that a) the numerical model is being solved correctly and b) the simulation model adequately represents the appropriate physics. The former activity is called Verification and requires intimate knowledge of the mathematical model representing the physics, the numerical approximation derived from that model, software quality engineering (SQE) practices, and numerical error estimation methods. The latter, termed Validation is accomplished by comparing simulation output with experimental data and quantifying the uncertainties in both. Broad knowledge of modeling and experimentation, augmented with a deep understanding of statistical methods, are necessary for Validation." Here, verification is the process of ensuring

that the software correctly implements the mathematical model of the world, and validation is the process of ensuring that the model accurately represents reality. In this definition, the model is clearly an intermediate artifact between the goals (and possibly requirements) and the software. One *validates* that the model satisfies the stakeholders' goal: that the model represent reality. One *verifies* that the model is satisfied by the software: that the software accurately implements the model. This usage is consistent with the Easterbrook quote in the above paragraph.

Our definition of validation and verification is compatible with the examples in the above paragraphs. We first assume the existence of a hierarchy of activities; *requirements* are inferior to *goals* but superior to *delivered software*. Given this assumption, we define validation and verification by determining which direction in the hierarchy one is looking. We define verification as *determining whether an activity in question is satisfied by the result of a hierarchically inferior activity*. We define validation as *determining whether an activity in question satisfies a hierarchically superior activity*.

To clarify, assume that some project has stakeholders (S) and a three-level hierarchy of constructed artifacts. Moving from outermost to innermost, level 1 is documented goals (G), level 2 is documented requirements (R), and level 3 is the delivered system (D). Verification is done by looking down the artifacts hierarchy: for example by checking whether R *is satisfied by* D. Validation is done by looking *up* the artifacts hierarchy: for example by checking whether R *satisfies* G. Our definitions are consistent with the above discussion, and have the added advantage (for us) of fitting into a hierarchy within which artifacts are constructed.

Note that because of the imprecision in the way artifacts are described, one must be careful about assuming that validation and verification are transitive. Verifying that (S is satisfied by G) and that (G is satisfied by R) does not necessarily verify that (S is satisfied by R). For example, assume that the stakeholders want a system to support a manufacturing process that uses gold for some reason. Gold is determined to be an asset, and the security goal `Protect gold from theft` is added to the system. The goal does not include information about *why* there is gold in the system. Without that information, the goal could be operationalized by the requirement `Gold`

`shall be buried in very deep holes`. This requirement satisfies the goal, and the goal satisfies the stakeholder. However, the requirement would almost certainly not satisfy the stakeholder, because the requirement does not satisfy the business goal.

2.2.4 Functional and Non-Functional Requirements

Requirements are often separated into two categories: functional requirements that describe what a system is to do, and non-functional requirements that describe some characteristic or quality the system is to have[2] ((Chung, Nixon et al., 2000) and many others). Functional requirements describe what a system *does*: they describe visible effects in the world that the system lives in. Jackson describes requirements as the interplay of phenomena that one wishes to be visible at interfaces of a particular set of domains (real world objects) implicated in a system (2001). One domain, the machine, orchestrates the communication between domains so that the interplay of phenomena is exhibited. The requirements are validated by checking with the stakeholders that the interplay produces what they wish. The requirements are verified by checking that the interplay takes place as specified.

Non-functional requirements ((Chung, Nixon et al., 2000; van Lamsweerde, 2001) & many others), cover such areas as performance, stability, ease-of-use, and (traditionally) security. These requirements, also called quality requirements (e.g., (Firesmith, 2003b)), generally do not have clear criteria for determining if they have been satisfied; there is no clear mapping from the requirement to effects in the world (van Lamsweerde, 2001; Mylopoulos, Chung et al., 1992). Validation and verification are problematic for quality requirements because yes/no validation and verification measurement criteria are hard to come by. Because of this difficulty, one must decide if a requirement has been *satisficed* (Mylopoulos, Chung et al., 1992), or satisfied well enough, which opens the question of what *well enough* means. For example, it may be easy to produce a requirement that states clearly what the stakeholders desire (for validation) but is unclear about what the system is to do (for verification), or vice versa. Consider the requirement/goal *the system shall be easy to use*. This goal is easy to validate; the users can

[2] NFR's can also be requirements on the development process, but we leave that aside in this discussion.

quickly say "Yes, I want that." However, the goal is very difficult to verify, because it says nothing about what ease of use *is* in the context of the system, or how one can know whether the system has achieved it. The goal could be changed to be *the system shall conform to the Common User Access standards*. It is much easier to verify that this goal is satisfied by the system, but most users would have difficulty confirming that a system conforming to that requirement would in fact be easy to use. Somehow, an original validatable goal must be translated into visible and measurable behavior in the context of the system. Only then can one verify that the system indeed has the required behavior.

2.3 Parallel Elaboration of Requirements & Architecture

The *Twin Peaks* model (Nuseibeh, 2001a, b) shows that the elaboration of requirements and architecture should proceed in parallel, each influencing the other. The model proposes a partial development methodology wherein *requirements* and *architecture* (where architecture includes *implementation*) are simultaneously elaborated and verified against each other, bound together by the *specification process*. The model extends the spiral method (Boehm, 1988) by making elaboration of requirements an explicit part of the spiral. The benefits derived from the model include earlier understanding of the problem(s) being solved because architectural constraints are discovered earlier, rapid turn-around, inherent recognition and incorporation of project management concerns such as IKIWISI (I'll Know It When I See It), easier incorporation of reusable components such as COTS (Commercial Off-The-Shelf) products, and rapid change in requirements and technology (Boehm, 2000).

Figure 1 (originally in (Nuseibeh, 2001a), copied from (Haley & Nuseibeh, 2003)) illustrates how a project might move from idea to implementation while using Twin Peaks. The peaks represent the requirements and architecture artifacts. The further one moves down a peak, the more detail is present and the more complete the artifact is. The spiral line represents the specification process, which is itself not an artifact but the simultaneous application of various and distinct methods to elaborate requirements and implementation.

The need for iteration between requirements and architecture is doubly true in the context of security requirements, because as was shown in the spreadsheet example in Chapter 1, security is a systems-level problem. One cannot accurately determine the security requirements without the context of the system, and the architecture of the system is part of its context. To illustrate the idea, consider a trivial functional requirement *business proposals shall be stored electronically using a format defined by the customer*. In addition, assume the existence of the general security goal *business proposals are to be treated as company-confidential information*. Without knowing the domains involved in the problem, how does one know how to keep the information confidential? One can postulate the existence of computers used to write and store the proposals, but cannot go much further. The designer could choose to put the machines in a

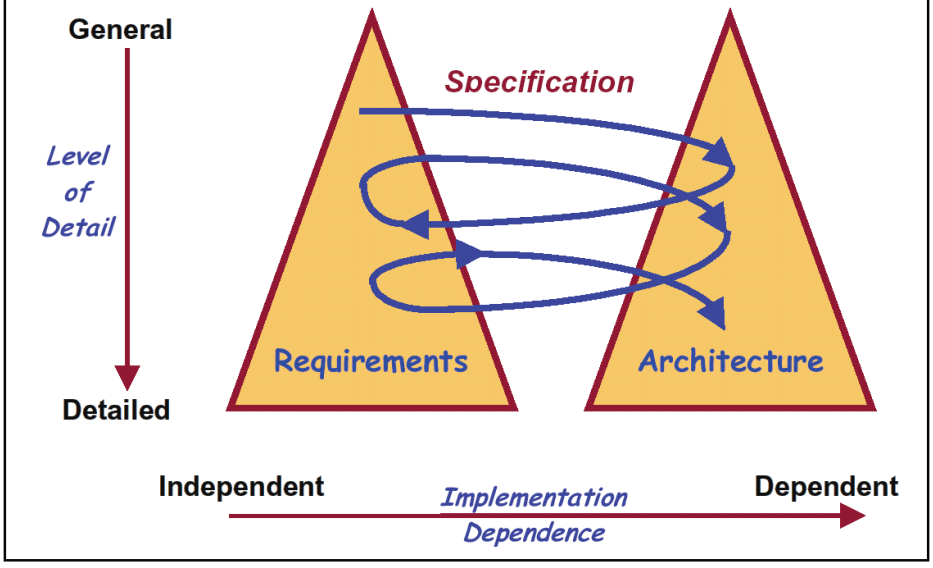

Figure 2-2 – Twin Peaks

locked room, in which case the *room key* becomes a phenomenon in the problem and the security requirements must describe the constraints on obtaining and using a key. Alternatively, the designer might specify a client/server architecture in which the client machines are publicly accessible. In this case, the client machine domain can be physically accessed by anyone and the proposals are potentially visible where the client and server domains connect (the network). The security requirements must describe constraints on who can use the client machine and on who can see the information where the domains connect.

When the requirements engineer attempts to build a correctness/satisfaction/adequacy argument for some requirement, it could be that an acceptable argument cannot be constructed because there is not enough information available from the context. The requirements engineer would then request the designers to intervene, making (or applying already made) design decisions appropriate for the level of information available, changing the context by changing and adding phenomena and possibly domains. The requirements engineer starts again with the new context, attempting to construct acceptable arguments. This iteration continues until an acceptable argument is made.

It is highly likely that applying a security requirement to a problem will alter the problem, possibly by changing phenomena, adding or removing domains in the existing problem, or both. For example, the specification to fulfill a security requirement *information shared between the client and server domain must not be accessible to anyone* must be evaluated in terms of visible phenomena. The designer must assure either that information shared between the domains is not visible outside the problem or that seeing what passes between the domains does not reveal the information. Either way, the physical properties of the connection need to be described.

Chapter 3. Related Work

Hindsight permits an examination of the literature using three lenses derived from the criteria listed in the Sections 1.1 – 1.3 of Chapter 1 of this thesis. Our first lens is the second criterion, assumptions about behavior: examining how context and assumptions are made explicit[3]. Our second lens is the first criterion, clarity: how security requirements are defined and their meanings understood. Our last lens is the third criterion, satisfaction: using design rationale and argument capture for security requirements verification.

3.1 Context & Assumptions

This section examines the literature using the first lens, asking how the system context and assumptions are made explicit.

For a security breach to occur, an attacker must make use of some entry point into a system to get to the assets. Given that the attacker is in the real world, the entry point must also be a real-world *domain* in the system (in the large, not just software). If no entry points exist that an attacker can use, the system cannot be exploited. (Nor, probably, can it be used, but that is another problem).

It is axiomatic that when an analyst constructs a context for a system, assumptions will be made about the behavior of domains in that context. Being a bit silly but making the point, even though the analyst believes that he or she is considering all the worst-case scenarios where all defenses are inexplicably breached, the analyst almost certainly (and implicitly) assumes that the computer running the software is not hostile. One must have a special mind set, such as the one described in *Programming Satan's Computer* (Anderson & Needham, 1995), before

[3] The second criterion is treated first to avoid some forward references.

everything would be assumed to be under the control of an enemy. On the other hand, the analyst might make implicit assumptions that are more problematic, such as "employees are always honest". In addition to exploring context, this section examines the literature by asking whether a representation for context should facilitate explicit capture of *trust assumptions*; assumptions the analyst makes about trusting the stated behavior of domains.

The section begins with a detailed look at context and assumptions in *i** and its derivatives, then moves on to KAOS and SeDAn. It ends with a brief examination of other security requirements work.

3.1.1 The i Framework*

The *i** framework (Yu, 1997; Yu & Liu, 2001) takes an 'actor, intention, goal' approach, where security and trust relationships within the model are modeled as "softgoals": goals that have no quantitative measure for satisfaction. The *i** framework incorporates the NFR framework, including related security work (Chung, 1993; Chung, Nixon et al., 2000). Liu et al. extended the framework to better support security and privacy by modeling the attacker as a malicious stakeholder (Liu, Yu, & Mylopoulos, 2003). Countermeasures, which are themselves (soft)goals, are added to thwart the attacker.

The Liu et al. work focuses on the *attacker* as the primary point of analysis (Liu, Yu et al., 2003). One finds vulnerabilities by asking what an attacker might wish to gain while playing some role, and then looking for ways that the attacker might achieve the wish. As *i** is focused on the actor, it is difficult to explore side effects of an actor's actions in the real world. For example, one cannot easily model implicit connections between actors formed because of an actor's actions, such as leaving one's password on a post-it note or the effects of a laptop being stolen.

*i** can be used to demonstrate the need for certain trust assumptions, specifically those that restrict which agents are permitted to play particular roles, and those that exclude the possibility of an agent exhibiting undesired behavior. There is, however, no convenient way to insert these trust assumptions into the model (beyond text annotations) without expanding the scope of the

analysis. For example, consider one of the countermeasures proposed in (Liu, Yu et al., 2003): "User Authentication Mechanism". In the diagram, the countermeasure is a leaf task. The actors and mechanisms that support, provide, and rescind authentication credentials are not mentioned, but are clearly being trusted by the analyst to be correct. To make these trust assumptions explicit in the model, one must add the actors who administer authentication, a process that is by necessity highly recursive.

The Tropos project uses the *i** framework, adding wider lifecycle coverage. Tropos focuses on connecting agent-oriented architecture and development with *i**, extending the *i** model to describe the details of the agents' behaviors (Castro, Kolp, & Mylopoulos, 2001). A formal specification language was added in (Fuxman, Pistore, Mylopoulos, & Traverso, 2001). Security, represented as constraints on the interactions between two agents, was later added (Gani, Manson, Giorgini, & Mouratidis, 2003; Mouratidis, Giorgini, & Manson, 2003), extending the specification language to express these constraints and agent interaction dependencies. Architectural styles beyond agent-orientation are also discussed (Mouratidis, Giorgini et al., 2003). Trust and trust delegation were added (Giorgini, Massacci, Mylopoulos, & Zannone, 2004; Giorgini, Massacci, Mylopoulos, & Zannone, 2005), along with appropriate extensions to the specification language.

Although Tropos has significantly enhanced *i**'s ability to represent security constraints and dependencies, and in particular confidentiality dependencies, it does not extend *i**'s ability to represent trust assumptions made by the analyst about the real world. The authorization example described above also applies to Tropos; one finds authorization constraints and sub-goals in an early Tropos security paper (Mouratidis, Giorgini et al., 2003), but one cannot explicitly indicate that administration of the authorization information is trusted, beyond extending the goal structure to include analysis of credential administration. One reasonable position is that some trust assumptions are implicit in the definitions and conditions of the formal modeling language (as can be said for KAOS below).

Other work has extended *i** in related directions. Gans et al. add distrust and inter-agent communication ("speech acts") (Gans, Jarke, Kethers et al., 2001). Actors in the system decide

dynamically to trust or to distrust each other. Yu and Cysneiros have looked at privacy (2002), exploring how privacy requirements fit into an *i** model. Both papers are concerned with analyzing trust relations between actors/agents in the running system, as opposed to capturing the requirements engineer's assumptions.

Because *i** and its derivatives do not model the real-world components of the system as it will eventually be built, certain classes of trust assumptions are difficult to make explicit. The best examples relate to unexpected connections between domains, such as information on paper passing through a mailroom, people hearing through walls, and security of backup media. Other examples can be found when looking at interactions between components on a system that are not *i** actors.

The conclusion one reaches is that although *i** works well for early requirements analysis and for actor/agent-based system analysis, it does not sufficiently represent general context and trust assumptions for security requirements analysis.

3.1.2 KAOS

KAOS (Dardenne, van Lamsweerde, & Fickas, 1993; van Lamsweerde, 2001), a goal-oriented requirements engineering method, uses *obstacles* to analyze security and safety (van Lamsweerde & Letier, 2000). An obstacle to some goal "is a condition whose satisfaction may prevent the goal from being achieved" (van Lamsweerde, 2004). A recent addition is anti-goals, a refinement of obstacles, to discover and close vulnerabilities (van Lamsweerde, Brohez, De Landtsheer, & Janssens, 2003; van Lamsweerde, 2004). One begins with a goal model for some system; the goal model includes a domain model expressed using temporal logic. Security goals for objects in the domain are enumerated using a catalog of general goals (e.g., confidentiality, integrity, etc.). One inverts these goals to express the goals of some attacker (anti-goals), and then looks for vulnerabilities in the original domain model that permit the anti-goals to be realized.

As in *i**, there are ways in KAOS to find and express some kinds of trust assumptions. One could argue that some *expectations*, which are terminal goals under the responsibility of non-

software agents ((van Lamsweerde, 2004), referred to as *assumptions* in (van Lamsweerde, 2001)), are expressions of trust assumptions, as the analyst is choosing to stop analysis at that point. Domain-specific axioms might also fall into the category of trust assumptions. For example, the *authorized* predicate described in (van Lamsweerde, 2004) is clearly depending upon knowing if an agent is an owner, a proxy, or a manager, but there is no expression of how it is known or managed, or what domain behavior permits it to be known.

Using KAOS, one expresses security goals in terms of the vulnerability to be addressed. As noted by van Lamsweerde, not all vulnerabilities must be eliminated, but instead may be mitigated or ignored (van Lamsweerde, 2004). The choice varies with the context of the vulnerability – the level of harm being risked and the probability that the harm will occur. One creates goals that express the choices made for a particular vulnerability. Focusing on the vulnerability as opposed to the asset to be protected loses information explaining the provenance of the goal (the context of the vulnerability). Goal refinement further distances the goal from its source. This distance creates difficulty when considering whether the cost of satisfying a security goal in a particular context is justified by the risk presented by the vulnerability in that context.

KAOS does not express context in terms of real-world domains, but instead expresses it in terms of goals, objects related to the goals, and actions needed to achieve the goals. Objects are not necessarily physical domains. Behavior is expressed in terms of logical pre- and post-conditions on objects. The actual recognized and emitted stimuli (phenomena) that permit the post-conditions to be satisfied are not recorded. As such, KAOS is a level removed from the real world. As the attacker is firmly placed in the real world, there is a mismatch between what the attacker manipulates and what KAOS models.

3.1.3 SeDAn

The SeDAn (Security Design Analysis) method (Chivers & Fletcher, 2005), developed concurrently with (and independently of) the work described in this thesis, has many similarities with our work, incorporating an explicit notion of context and a definition of security

requirements as constraints. A SeDAn context is an information flow graph, mapping the flow of information assets from their source through a network of services running on a grid. The definition of 'services' includes the users of the services. Using an attack model, one looks for paths in the graph that attackers can exploit. Such "paths of attack" may be within a service (e.g., internal users), where services are connected (administrative or organizational boundaries), or where information is exposed to the external world. The goal is to show whether a path exists from an attacker to the information asset. Constraints (security requirements) are placed on services to block such paths.

A SeDAn context is focused on information assets and software services running on a grid, using a UML description of the interconnection of services. The boundary of the system (it's 'edge') is the user interface, and analysis is limited to vulnerabilities within this boundary. For example, bribing a user is not considered directly, because the user is outside the system boundary (Chivers & Fletcher, 2005: pg 878).

Using SeDAn, an asset and threat analysis is done to determine the risk (including both likelihood and impact) that a path of attack can be utilized. One makes a table of the assets and the security concerns that affect the assets, noting the impact of the violation of the concern. Next, one determines which assets might be available through a given path of attack and the likelihood that the path of attack can be utilized in the undesired way. Impact and likelihood are combined, resulting in a quantified value for risk.

For risks considered serious enough, constraints are added that, when satisfied, will cause the path to be sufficiently blocked. The constraints are on deployment, system topology, behavior of a service, and external access. They act "as requirements for implementers … " (Ibid: pg 882). SeDAn does not contain an explicit satisfaction argument for how the constraint sufficiently mitigates the risk, but does contain model checking to check that the system will satisfy the constraint.

Chivers and Fletcher acknowledge that satisfying constraints can cause additional functionality to be added and that this new functionality "may include new assets and services,

and these in turn may have confidentiality or integrity concerns" (Chivers & Fletcher, 2005: pg 887). However, they say neither how this iteration is structured within the process, nor how traceability back to the security concern that provoked the addition of functionality is maintained.

3.1.4 Other Work

He and Antón (2003) concentrate on privacy, working on mechanisms to assist trusting of privacy policies, for example on web sites. They propose a context-based access model. Context is determined using "purpose" (why is information being accessed), "conditions" (what conditions must be satisfied before access can be granted), and "obligations" (what actions must be taken before access can be granted). Their framework, like *i**, describes properties desired at run-time, not the requirements engineer's assumptions about the domains forming the solution.

Security requirements have been added to SCR (Heitmeyer, 2001) and to the WinWin framework (In & Boehm, 2001). As with *i** and KAOS, one can locate some trust assumptions in both SCR and WinWin by looking for where the analyst stopped. The implicit decision to limit the context almost certainly has some number of trust assumptions behind it.

Several people have described techniques that assist with reasoning about security by postulating the existence of an attacker who attempts to exploit the system in a way that will cause harm. Sindre and Opdahl introduced the idea of *mis*use and *mis*actors into use cases to identify potential security flaws in a system (2000). Their work concentrated on simplicity, using the diagrams as a communications tool and saying that "misuse diagrams must only be seen as a support for eliciting threats". Alexander extended the relations over those presented by Sindre et al., adding mitigation and restriction (2002a; 2002b). McDermott et al. described abuse cases, concentrating on exploring the details of an exploit and documenting the route and expertise needed to be successful (McDermott & Fox, 1999; McDermott, 2001). All of these techniques employ use cases, an actor/action model, so they have many of the same representation problems as *i**. In these cases, an analyst's trust assumptions are implicit in the diagrams, and not made explicit. One can argue that the very choices of which cases to analyze

constitute trust assumptions, because the analyst is choosing which interactions are important. It is worth noting that although these techniques do not capture trust assumptions in some explicit way, they would be quite useful for testing validity of the satisfaction arguments built using trust assumptions.

Srivatanakul et al. combined use cases with risk analysis techniques taken from safety (Srivatanakul, Clark, & Polack, 2004), specifically HAZOP (Kletz, 1999; McDermid, Nicholson, Pumfrey, & Fenelon, 1995). They extended the abuse and misuse case work discussed above ((Alexander, 2002a, b; McDermott & Fox, 1999; McDermott, 2001; Sindre & Opdahl, 2000)) by adding HAZOP 'guideword'-driven analysis of use cases to find potential abuses. One uses the guidewords to find *deviations* for the elements in a use case (e.g., actors, associations, event flow, pre- and post-conditions). These deviations represent potential violations of "security properties" of a system. If a security property is (potentially) violated, the deviation represents a (potential) successful attack. One locates the vulnerabilities that were exploited, then takes appropriate steps to close or mitigate the vulnerabilities. The method, like misuse cases, abuse cases, and abuse frames (Lin, Nuseibeh, Ince et al., 2003), takes what might be considered a bottom-up approach; the methods locate vulnerabilities that lead to security requirements that, if satisfied, will ensure the closure of the vulnerabilities. If no vulnerabilities are found, then the satisfaction argument has been bolstered. Finally, the technique employs use cases and therefore has many of the same representation problems as *i**.

Some of the work in the aspect-oriented requirements engineering (AORE) community is related to identification of security requirements. Yu et al. proposed an extension to *i** to model softgoals, including security softgoals, as aspects (Yu, Leite, & Mylopoulos, 2004). Rashid et al. propose that ideas from aspect-oriented software development can be used when mapping non-functional requirements onto functional requirements (Rashid, Sawyer, Moreira, & Araújo, 2002; Rashid, Moreira, & Araújo, 2003). They start by identifying the non-functional requirements (NFRs) that affect more than one functional requirement, determine what the effect of the overlap is, then model the composition of the requirements. In their work, security is treated identically to other NFRs. Context and assumptions are not taken into account.

Brito and Moreira (2004) propose that non-functional requirements from an NFR catalog (Chung, Nixon et al., 2000), be integrated with functional requirements using a composition process. The composition process connects security goals with functional requirements and permits supplying a priority for satisfaction arguments, but does not aid with the construction of these arguments. None of this work incorporates capture of the assumptions made by a requirements engineer when specifying a system.

3.2 Expressing Security Requirements

This section examines the literature through our second lens: how security requirements are defined and their meanings understood. We look from the point of view that to be most useful, security requirements should have the following characteristics of functional requirements: they should be unambiguous, verifiable, and free of conflicts. In addition, given general security goals, there should be a clear pathway to finding security requirements.

If the security requirements cannot be verified (recall that verification looks down the hierarchy, in this case to establish that requirements are satisfied by the system), then one cannot know if the system is appropriately secure. This problem often arises when security requirements are expressed in ambiguous or overly general terms, such as “the system must be secure” and “only authorized users can use the system.” Verifying the system against these requirements requires one to guess at their meaning. The developers must somehow determine what ‘secure’ means, who is a ‘user’, what users are authorized to do, and when they are authorized to do it. What is needed is a way of expressing security requirements that avoids these problems.

The need for avoiding and resolving conflict in security goals and requirements can be illustrated by considering two stakeholder groups in a health care system: physicians and regulatory compliance officers. Physicians have a duty of care; they are morally and in some cases legally obliged to provide adequate care. They will demand from a system what they consider sufficient functionality and privilege needed to carry out their duties. We see from Anderson’s report (1996) that one privilege physicians frequently expect is to be able to discuss

a case with some other physician. However, privacy regulations require patient consent before disclosure of information report (Anderson, 1996; Mouratidis, Giorgini et al., 2003). Regulatory compliance officers are charged with ensuring the respecting of privacy regulations, meaning that physicians must not share information with other physicians until the patient gives his or her consent. We find here a conflict of duties (duty of care vs. duty of compliance) that will affect who has which privileges in a system. However, the system's requirements in this instance must be free of conflicts, because if not, the implementers may resolve the conflicts in potentially inconsistent and incorrect ways. The conflict between rival stakeholders must be resolved by the production of an agreed set of security requirements.

3.2.1 Security Requirements as Security Functions

It is common to express security requirements by describing the security mechanisms to be used. For example, ISO 15408 (ISO/IEC, 1999a, b, c), a security specification that is the ISO version of the Common Criteria (Common Criteria Sponsoring Organizations, 2006a, b, c), provides examples of security requirements of the form (somewhat paraphrased) "The […] Security Function (TSF) shall explicitly deny access of subjects to objects based on the [rules …]" (ISO/IEC, 1999b: pg 48), where "rules" appear to be a mechanism. Regarding encryption, one finds "The TSF shall distribute cryptographic keys in accordance with a [specified cryptographic key distribution method] that meets the following: [list of standards]" (Ibid: pg 39). Again, a mechanism is being described. In addition, both examples say what the function is to do, not the purpose it is to accomplish.

The NIST Computer Security Handbook states that "These [security] requirements can be expressed as technical features (e.g., access controls), assurances (e.g., background checks for system developers), or operational practices (e.g., awareness and training)" (NIST, 1995: pg 80), in effect defining security requirements in terms of functions and practices. Other security guides imply that recommendations such as "Acquire Firewall Hardware and Software" (e.g., (Allen, 2001)) are requirements.

Defining requirements in terms of function leaves out key information: *what* objects need protecting and, more importantly, *why* the objects need protecting. Although both the ISO and NIST documents say that the underlying reasons why objects are to be protected come from the functionality of the system, they provide little guidance on how to connect the functionality to the security needs. Instead of describing *when* and *why* objects are to be protected, they describe what mechanisms *are* to be used to protect the objects.

It should be noted that the ISO and NIST guides are excellent sources of state-of-the-practice security mechanisms. The requirements engineer would be well advised to consider the functions described in these documents as excellent pointers to areas in a system that could participate in security satisfaction arguments (see Chapter 6).

3.2.2 Security Requirements as Non-functional Requirements

Devanbu & Stubblebine (2000) remark that security requirements are a kind of non-functional requirement. Kotonya and Sommerville (1998), when discussing non-functional requirements, in which they include security, define them as "restrictions or constraints" on system services. Similar definitions can be found in other textbooks. Rushby (2001) appears to take a similar view, stating "security requirements mostly concern what must not happen". Using the Tropos methodology, Mouratidis, Giorgini et al. state that "security constraints define the system's security requirements" (2003). Chung explicitly defines information security requirements as non-functional requirements (Chung, 1993; Chung, Nixon et al., 2000).

Firesmith defines security requirements as "a quality requirement that specifies a required amount of security […] in terms of a system-specific criterion and a minimum level […] that is necessary to meet one or more security policies." (Firesmith, 2003a, 2004). This appears to be a form of constraint, an impression reinforced by an example he provides: "The [application] shall protect the buyer-related data […] it transmits from corruption […] due to unsophisticated attack [when] […] Buyer Buys Item at Direct Sale [to a level of] 99.99%."

The problem with these definitions is their lack of specificity and guidance for the designers. What "system services" are being constrained? What effect will the constraint have on the

functionality of the system? How can any eventual chosen constraint be validated to ensure that it accurately reflects the stakeholders' wishes? Referring to Firesmith's example, what is an "unsophisticated attack?" What does the measure "99.99%" mean? It could mean that if 10,000 attacks are known, the developers can ignore one. Alternatively, it could be a way of saying "all" without actually saying it.

One major problem with percentage-style quantification of security requirements is the binary nature of the majority of security attacks; in most cases, an attack works or it does not. If an attack does not work the first time, it probably will not work the second time unless the parameters of the attack or the system state are changed. Anti-intrusion measures such as account lockout help ensure that attacks following a failed attack attempt will have a lower probability of success. On the other hand, if the attack works once (the system is penetrated), then the attack will likely continue to work until the vulnerability is removed. Successful attacks can (usually) be repeated as often as the attacker wishes, and even shared amongst attackers[4]. In high-threat situations, a successful attack will almost certainly occur (Redwine, 2006: pg 80). It is difficult to know what the percentage quantification means in these cases.

Although we agree with defining security requirements as constraints, we argue that two precisions are necessary: a more precise definition and representation of constraints, and a way of moving from business goals to constraints.

3.2.3 Security Requirements from Privacy & Trust

Some researchers look at security from the point of view that if an agent can trust that information it 'owns' is kept private, then security goals will be met. De Landtsheer proposes modeling which properties that agents, authorized or not, can know (De Landtsheer & van Lamsweerde, 2005). The Tropos project (Giorgini, Massacci et al., 2005 and several others) takes a similar view, but extended to include agents' intentions and explicit trust delegation.

[4] One example is "script kiddie" attacks, where an experienced attacker produces toolkits for inexperienced attackers to use.

Breaux Vail, and Antón (2006) extract privacy rights and obligation information from "policy documents" to assist with development of security requirements.

These approaches work well in contexts and problems dominated by privacy concerns. They are less effective when considering vulnerabilities in a system context. They are also less effective in applications where privacy (c.f. confidentiality) is not the dominant concern. The example in Chapter 7 is one such case. Air traffic control is dominated by integrity and availability concerns; high confidence is needed that airplanes are where they say they are.

3.2.4 Other Portrayals of Security Requirements

Many authors implicitly assume that security requirements are identical to high-level security goals. Tettero et al are explicit about this, defining security requirements as the confidentiality, integrity, and availability of the entity for which protection is needed (Tettero, Out, Franken, & Schot, 1997). While this is a clear definition, in some cases it may not result in precise enough requirements. In the above example, both doctors and the administrators would probably agree on the importance of confidentiality, integrity, and availability of the clinical information, but they would disagree on the concrete security requirements that express those goals. The requirements need to be more explicit about *who* can do *what*, *when*.

Some authors identify security requirements with security policies. Devanbu & Stubblebine (2000) define a security requirement as "a manifestation of a high-level organizational policy into the detailed requirements of a specific system. [… We] loosely (ab)use the term 'security policy' […] to refer to both 'policy' and 'requirement'". Anderson (2001) is less direct; he states that a security policy is "a document that expresses […] what […] protection mechanisms are to achieve" and that "the process of developing a security policy […] is the process of requirements engineering". Redwine (2006) reports that the "software system security policy is part of software system requirements placing constraints on system behavior". The difficulty with security policies is their chameleon-like meaning. As the discussion above shows, the term can be used for anything from a high-level aspiration to an implementation. Therefore, without

accompanying detailed explanation, it is not satisfactory to define security requirements as security policies.

Lee et al. (Lee, Lee, & Lee, 2002) point out the importance of considering security requirements in the development life cycle, but do not define them. ISO/IEC 15408 (ISO/IEC, 1999c) does not define security requirements in its glossary. However, in one place, they are depicted as being at a higher level than functional requirements, but in another as "security requirements, such as authorization credentials and the IT implementation itself", which appears to be at too low a level. Heitmeyer (2001) shows how the SCR method can be used to specify and analyze security properties, without giving the criteria for distinguishing them from other system properties.

A number of papers have focused on security requirements by describing how they may be violated. For example, McDermott & Fox (1999), followed independently by Sindre & Opdahl (2000) and elaborated by Alexander (2003), describe abuse and misuse cases, extending the use case paradigm to undesired behavior. Liu, Yu & Mylopoulos (2003) describe a method of analyzing possible illicit use of a system, but omit the important initial step of identifying the security requirements of the system before attempting to identify their violations. One could argue that Chivers and Fletcher (2005) are in this camp with SeDAn, as they focus on attackers and the paths they might take into a system. One consequence of these approaches is that they indicate what a system is not to do in specific situations, but not in the general case. General security requirements must be inferred from the list of undesirable situations.

Van Lamsweerde (2004) describes a process by which security goals are made precise and refined until reaching security requirements; see section 3.1.2 for more detail. Antón & Earp (2001) use the GBRAM method to operationalize security goals for the generation of security policies and requirements, but do not define security requirements.

Mead et al. in the SQUARE methodology (2005) describe security requirements as being at the system level or the software level. They do not define what requirements are, beyond saying that “Requirements are concerned with what the system should do”. They also introduce the

notion of "architectural constraints" that specify "how it should be done," leaving open how one distinguishes between a constraint that a system use an existing authentication system and a requirement that the system support authentication in a given context. Our framework fits well within SQUARE, providing a pathway from goals to requirements, and making the requirements (or constraints) implied by the context clear.

3.3 Use of Design Rationale and Argument Capture for Verification

Our third lens, the use of design rationale and argument capture for security requirements verification, is used in this section.

The work presented in this thesis is related to, and builds upon, research on design rationale and argument capture, on safety requirements analysis, and more generally on ideas behind problem domain analysis.

3.3.1 Design Rationale

Design rationale is principally concerned with capturing how one arrived at a decision, alternate decisions, or the parameters that went into making the decision (Lee & Lai, 1991). For example, Buckingham Shum focuses on how rationale (argument) is visualized, especially in collaborative environments (2003). Potts and Bruns (1988), and later Burge and Brown (2004) discuss capturing how decisions were made, which decisions were rejected, and the reasons behind these actions. Mylopoulos et al. (Mylopoulos, Borgida, Jarke, & Koubarakis, 1990) present a way to represent formally knowledge that was captured in some way, without focusing on the outcome of any decisions. Ramesh and Dhar (1992) describe a system for "capturing history in the upstream part of the life cycle." Fischer, Lemke et al. (1996) suggest that the explicit process of argumentation can itself feed into and benefit design. Finkelstein and Fuks (1989) suggest that the development of specifications by multiple stakeholders who hold disparate views may be achieved through an explicit dialogue that captures speech acts, such as assertions, questions, denials, challenges, etc. The representation of the dialogue is then a rationale for the specifications constructed. The common element in all of the above work is the

capture over time of the thoughts and reasons behind decisions. Whether the decisions satisfy the needs is not the primary question.

When analyzing security requirements, the ultimate goal is to *convince a reader* that the security requirements can be satisfied, and that nothing is omitted that could result in the requirements not being satisfied. The process used is relevant only as it relates to completeness. Optimality is not part of the argument. Of course, we make no claim that it is useless to have the history that led to the final arguments; such a history will certainly be useful if the arguments fail to convince, or if the situation changes.

3.3.2 Safety Cases

Kelly (1999) argues that "a safety case should communicate a clear, comprehensive and defensible argument that a system is acceptably safe to operate in a particular context." He goes on to show the importance of the distinction between *argument* and *evidence*. An argument calls upon appropriate evidence to convince a reader that the argument holds.

Attwood and Kelly (2004) use the same principles, taking the position that argument forms a bridge between requirements and specification, permitting capture of sufficient information to realize rich traceability. Combining the two ideas, argument for safety cases and using arguments for traceability, Kelly's quote presented above is paraphrased as "a security satisfaction argument should communicate a clear, comprehensive, and defensible argument that a system is secure enough to operate in its context."

The techniques proposed by Kelly are not directly applicable to security without modification, primarily because the techniques are focused around objective evidence, component failure, and accident; rather than subjective reasoning, subversion, and malicious intent.

3.3.3 Problem Domain Analysis

Zave and Jackson in (1997), and Jackson in (2001), argue that one should construct a *correctness argument* for a system, where the argument is based on known and desired

properties of the *domains* involved in the *problem*. To quote Jackson, "Your [correctness] argument must convince yourself and your customer that your proposed machine will ensure that the requirement is satisfied in the problem domain." This position is the same as Kelly's, with the proviso that Kelly's arguments focus equally on all domains, with no special emphasis on the machine.

Correctness arguments apply to security requirements, with a significant distinction. It is very difficult to talk about correctness when discussing security. One can convince the reader that the proposed system meets the needs, but it is far more difficult to prove that the system is correct. The distinction between convince and prove (or show) is important. It is not possible to prove the negative – that violations of security goals do not exist – but one can be convincing that sufficient outcomes have been addressed.

3.4 Chapter Summary

The review of the literature shows that our three criteria for adequate security requirements are not yet adequately satisfied by existing work. To reiterate, our criteria are:

1. *Clarity*: one must have a clear understanding of what security requirements mean, and their effects within the system context in which they apply.
2. *Incorporation of assumptions about behavior*: security requirements must take into consideration an analyst's implicit or explicit decisions to trust behavior of objects found in the system.
3. *Satisfaction*: one must be able to determine whether the security requirements satisfy the security goals, and if the system can satisfy the requirements.

Our contributions flow directly from a desire to satisfy all the criteria. To satisfy the first criterion, we propose a framework for security requirements engineering explicitly incorporating system context. To satisfy the second, we propose the use of trust assumptions in security requirements. To satisfy the third, we propose combined formal/informal security requirements satisfaction arguments.

Chapter 4. Trust Assumptions

Recall that a system comprises not only software, but also all the diverse constituents needed to achieve its purpose. For example, a computing system clearly includes the computers, but also incorporates real-world elements such as the people who will use, maintain, and depend on the system; the physical and logical environment within which the system will exist; and any systems already in place. When operating in a systems context, the requirements engineer must determine which real-world elements are to be included in the analysis. The analyst must define the *context* within which requirements analysis takes place by selecting the *domains* (the aforementioned real-world constituents) that are considered pertinent (Jackson, 1995, 2001). In doing so, the analyst reduces the size of the context to those domains relevant to the problem.

As explained in Chapter 1, one factor influencing an analyst's choice about whether or not a domain is relevant to a system's security, and therefore to be included in the context, is the analyst's set of *trust assumptions*. Trust assumptions are explicit or implicit choices not to challenge some described characteristics of domains, and can have a significant impact on the security of a system. To repeat the example from Chapter 1, most analysts implicitly assume that the compiler is not a security risk; it would not occur to them to include it in the analysis. Thompson demonstrated that this assumption is not necessarily justified by showing how a compiler could introduce trapdoors into applications (1984). Thompson's example and the other in Chapter 1 illustrate how the requirements engineer's implicit trust of some domains in the environment can introduce unknown amounts of risk into the system.

Although these examples demonstrate the need to capture and analyze trust assumptions, little exploration has been done on how to find, represent, and quantify them; and then to analyze their effect on the system under discussion. We correct this omission by first providing a better understanding of what trust assumptions are, and then by making explicit their place

within satisfaction arguments. This chapter provides the former, a better understanding, by examining trust assumptions as independent artifacts used in very informal argumentation. Chapter 6 presents the latter, putting them into the context of satisfaction arguments.

4.1 Definition of Trust Assumptions

We define a trust assumption as a choice made by a requirements engineer to depend upon a domain having certain properties, in order to satisfy a security requirement. The requirements engineer *trusts* the *assumption* to be true. These assumed properties act as *domain restrictions;* they restrict the domain in some way that contributes to the satisfaction of the security requirement.

4.1.1 Purpose of Trust Assumptions

The requirements engineer is responsible for constructing an argument that security requirements are satisfied – the *satisfaction argument*. In most cases, the satisfaction argument cannot be made without depending on domain properties that cannot be verified with the information available in the context. The requirements engineer has a choice: either add a trust assumption that asserts that the properties are valid, or expand the scope as necessary to verify the properties, which is a highly recursive process. By choosing to add a trust assumption, the requirements engineer ends the recursion and explicitly limits the scope of the analysis.

To illustrate making the choice to expand the scope or adding a trust assumption, assume the existence of a security requirement stipulating that the computers operate for at least eight hours in the event of a power failure (an availability requirement). The requirements engineer, working with the designers and the stakeholders, can satisfy this requirement by adding backup generators to the system. Appropriate phenomena would be added so that the machine can detect the power loss, control the generators, detect going beyond eight hours, etc. In most situations, the requirements engineer can trust the manufacturer of the generators to supply equipment that does not intentionally permit an attacker to take control of the generators and prevent them from operating (a denial of service attack). The analyst trusts the behavior of the

generators, and adds a trust assumption to that effect. By adding the trust assumption, the requirements engineer does not need to include the manufacturer of the generators in the analysis. The analyst uses the trust assumption to limit the scope of the analysis.

As explained above, trust assumptions contribute to the satisfaction of security requirements. There is not necessarily a one-to-one correspondence between a trust assumption and the security requirements satisfied. Several trust assumptions may be necessary to satisfy a security requirement (an *and* decomposition), any one of several trust assumptions may be sufficient to satisfy a security requirement (an *or* decomposition), or some combination of the two. In addition, one trust assumption may play a role in satisfying multiple security requirements.

4.1.2 The 'Trust' in Trust Assumptions

We must first define what we mean by *trust* in trust assumptions. We use a variant of the definition of trust proposed by Grandison & Sloman (2003): "[Trust] is the quantified belief by a trustor with respect to the competence, honesty, security and dependability of a trustee within a specified context". In our case, the *requirements engineer* trusts some domain to participate 'competently and dependably' in the satisfaction of a security requirement in the context of the problem.

In the Grandison & Sloman definition, the quantification of trust represents the level of confidence that the trust assumption is valid. Said another way, the quantification represents the risk that including the trust assumption, and thereby limiting the scope of analysis, may not be justified. In this thesis, the quantification is binary; the trust assumption is thought to be valid, or it is not.

The Thompson (1984) example in the introduction gives us an example of a trust assumption. An analyst's (probably implicit) trust of the compiler vendor not to include trapdoor generators in the compiler may be misplaced. If the compiler has been compromised, then some number of vulnerabilities may exist, such as the existence of a universal password, denial-of-service traps, or information leaks. Successful attacks using these vulnerabilities will have some impact on the organization: they will cause harm. The organization must decide whether the risk presented by

the vulnerabilities that might come into existence if the trust assumption is not valid is sufficient to justify the time and expense of the expansion of the analysis required to verify the compiler.

The risk presented by a trust assumption is not the same as the risk associated with a vulnerability that might exist if the trust assumption is not valid. The risk presented by a trust assumption measures how likely it is that the vulnerability might exist if the trust assumption is invalid. The risk associated with a vulnerability measures the likelihood that the vulnerability can be successfully exploited, along with the impact of a successful exploit. As the example in the previous paragraph shows, the two measures are independent. If a compiler has been compromised to modify the password checker of the `login` program (the case described by Thompson), the trust assumption is invalid and the risk of the existence of a vulnerability is high. However, if the `login` program is not used in a system, then the risk presented by the vulnerability is nil, regardless of the validity of a trust assumption stating that the compiler vendor can be trusted.

A discussion of formal risk analysis is outside the scope of this thesis, and will not be further discussed.

4.1.3 Representation of Trust Assumptions

A trust assumption consists of the following information:

- Identification of the dependent domain. The trust assumption restricts this domain.
- Effect of the trust assumption. The trust assumption a) restricts instances of the domain to be instantiations of some class or members of some set, b) restricts phenomena on the interfaces of the domain, or c) some combination of the two. Note that phenomena restrictions can be an assertion that some phenomena will not appear on the interface, or will only occur in a specific sequence/interchange.
- Narrative description of the restriction(s). If the trust assumption restricts the instances of a domain type, then describe the attributes of instances of the domain type before and after application of the restriction (in effect, a description of the subtype). If the trust assumption

restricts phenomena, then describe the restriction and its effect on the valid interplay of phenomena. At this point, when discussing the validity and effect of the restrictions in this section, the analyst should take the position that the trust assumption is valid.

- Preconditions. Some trust assumptions may be considered valid only if some other conditions are true. Some examples might be the earlier application of some other trust assumption to the dependent domain and/or the existence of domains not otherwise included in the analysis.
- Justification for the inclusion of the trust assumption. This is not a justification of the restrictions, but is instead an informal discussion of why the trust assumption should be considered valid. If there are risks associated with the trust assumption, they should be listed and discussed.
- List of security requirements (the constraints) that this trust assumption satisfies partially or completely. A trust assumption participates in satisfaction of a security requirement is by appearing in a satisfaction argument for that requirement.

4.1.4 Trust Assumptions as Domain Restrictions

Trust assumptions either restrict instances of a domain to some subtype, restrict the phenomena that a domain can produce, or both. To illustrate restricting instances of a domain's type, consider a company's door security system. By restricting entrance to people who pass the system's test (whatever that is), the system in effect changes the type of the domain from *People* to *Employees*. To illustrate restricting phenomena, consider the output of the balance enquiry function of an ATM. The analyst might assume that the ATM displays the information for the account indicated by the card, not some other account. The trust assumption is that no defects exist that would cause the ATM to display information for some other account.

4.2 Worked Example

The Secure Electronic Transaction (SET) Specifications (Secure Electronic Transaction LLC, 1997a, b, c) describe a set of mechanisms intended to provide an acceptable level of security for on-line purchasing. This worked example looks at incorporating the SET specifications into software to support cardholder-side payment authorization. There is one functional requirement (in the problem frames sense): Complete the Purchase. This example considers one asset, *Customer Account Information (CAI)*, and one derived security goal *Purchases shall be authorized*. Several trust assumptions are derived during the analysis.

To derive the trust assumptions, we first determine what actions might cause harm, then negate these actions to express the security requirements (the constraints). (Describing threats is described further in Chapter 5.) Two such action/harms are used in this example: *exposure of cardholder account information could lead to financial loss* (from the confidentiality concern), and *unauthorized use of cardholder credentials could lead to financial loss* (from the integrity concern). We next add security requirements (constraints) to the requirements:

```
SR1: only authorized individuals may use the cardholder credentials.
```

and

```
SR2: only authorized users may see the CAI
```

The trust assumptions needed to satisfy the security requirements will be described in a later section.

4.2.1 SET Overview

SET describes a series of operations between players in an electronic purchase transaction using a credit card. In SET, a *cardholder* requests a cryptographic certificate from a *certificate authority* (CA). The CA verifies that the cardholder has a credit card account with an *issuer*, and then supplies a certificate. The cardholder can subsequently use the certificate to make purchases from a *merchant*. The merchant uses a *payment gateway* to pass the transaction to the *acquirer* (the merchant's bank) for collection. The acquirer normally operates the payment

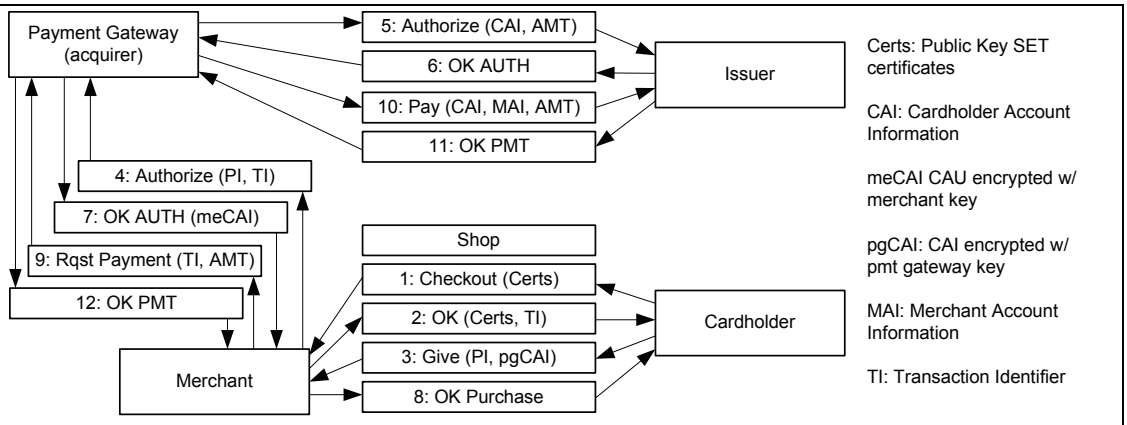

Figure 4-1 – Simplified SET processing flows

gateway. Figure 4-1 presents a simplified version of the SET "processing flows" (terminology from (Secure Electronic Transaction LLC, 1997c)), showing the players and the messages they interchange. The arrows represent the direction of the flow of a message. The numbers in the boxes indicate sequence. Several SET messages and fields that do not have a direct bearing on this discussion have been omitted from the diagram, in particular the obtaining of certificates and private keys, and the initial verification of cardholder information. In addition, the diagram shows the merchant using the CAI, which although optional in SET is the technique that the SET specifications claims will be the most often used. (Secure Electronic Transaction LLC, 1997b: pg 14)

4.2.2 SET-Identified Security Assumptions

The SET specifications make the following security-related assumptions about the SET environment relevant to this worked example. They are relevant because they point us at vulnerabilities considered by the writers of the SET specifications.

- SA1: The cardholder ensures that no one else has access to his/her private key. (Secure Electronic Transaction LLC, 1997c: pg 16) In particular, SET software vendors shall "ensure that the certificate and related information is stored in a way to prevent unauthorized access." (Ibid: pg 46)

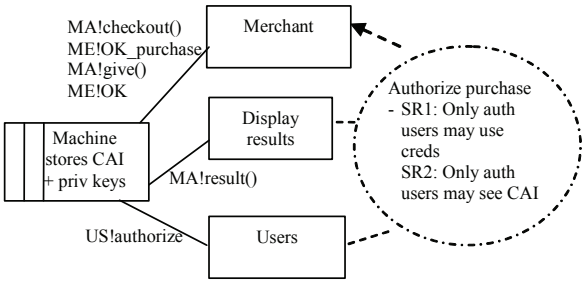

Figure 4-2 – Purchase problem

- SA2: Cardholder, merchant, and payment gateway machines are free of viruses and trojan horses, and are not susceptible to being hacked. (Secure Electronic Transaction LLC, 1997c: pg 11)
- SA3: Programming methods and the cryptographic system, and in particular, the random number generators, are of the highest quality. (Ibid: pg 16)
- SA4: The merchant's system stores account information in an encrypted form, and if possible off-line or behind a firewall. (Secure Electronic Transaction LLC, 1997b: pg 39)

4.2.3 The Initial Problem Diagram

There is only one requirement in this worked example and therefore only one problem diagram. The context does not include the shopping process, but instead focuses on the point where a purchase is completed. Figure 4-2 shows a first-cut problem diagram, built by considering the SET processing flows.

Recall from the discussion at the beginning of this section that there are two security requirements to be satisfied: SR1: *only authorized individuals may use the cardholder credentials*, and SR2: *only authorized users may see cardholder account information (CAI)*. CAI is made visible by the CAI phenomena in the problem diagram, and the asset *cardholder credentials* is stored in the machine. Our goal is to generate an informal argument that these security requirements are satisfied.

By tracing the CAI through the problem diagram, one sees that it must reside in unknown form within the Machine domain. According to the SET specification, the CAI must be encrypted between the machine and the merchant. There is nothing in the problem description (problem diagram) that indicates that only the user or the merchant can see the CAI. One can say the same thing about cardholder credentials. We can say nothing about whether SR1 or SR2 are satisfied. We use these observations, the requirements SR1 and SR2, and the security assumptions SA1-SA4 to make the following trust assumptions[5]:

- TA1-1 – satisfaction of SR1: As the credentials are stored on the machine, and as there is no apparent way to limit who can access these credentials, SA1 forces us to assume that the domain *Users* in the problem contains only individuals authorized to use the credentials.
- TA1-2 – satisfaction of SR1: The CAI and credentials are not visible outside the machine. (SA2)
- TA1-3 – satisfaction of SR1: The generated symmetric encryption keys are cryptographically secure. (SA3)
- TA1-4 – satisfaction of SR1 and SR2: The merchant cannot know the cardholder's private key, and therefore cannot see the CAI as it passes through to the payment gateway.

The first trust assumption TA1-1, that the domain *Users* contains only authorized individuals, is clearly risky, making the argument that SR1 is satisfied very problematic. There is no information available in the context to justify the claim. The analyst should change the problem to eliminate the trust assumption and reduce the risk. A similar statement must be made about TA1-2, because nothing can be found in the context that allows the engineer to claim that the storage is secure. If the information can be read without supplying some credentials that are not stored on the machine, then the existence of viruses, spyware, and other programs/users make the trust assumption's claim ludicrous. Vulnerabilities permitting realization of the threats still

[5] The labeling TA1-n instead of TAn is used because we will make a second set of trust assumptions TA2-n later.

exist, and appropriate domains and phenomena must be added to close the vulnerabilities and satisfy the requirement.

Verifying TA1-3 is probably not necessary, assuming that the cryptographic software comes from a company that the requirements engineer believes has verified its applications. If the engineer is uncomfortable with this belief, then a domain representing the encryption software company must be added to the problem, and then analyzed appropriately.

TA1-4 serves to limit the scope of the analysis, stating that nothing on the other side of the merchant can expose CAI to the merchant. Unfortunately, the SET processing flows diagram (Figure 4-1, step 7) shows that the payment gateway can give the CAI back to the merchant. The trust assumption is invalid and must be removed.

Because TA1-1 was rejected, a *passphrase* has been added to verify that the user is authorized. The passphrase is used to encrypt the CAI and certificate storage. Use of the passphrase and encryption protects the CAI against both viruses and other users of the machine. Spyware that can capture the entry of the passphrase is still a problem, one that is not further discussed in this thesis. Because the rejection of TA1-1 & TA1-4 caused the system to be modified, we do not look further at TA1-2.

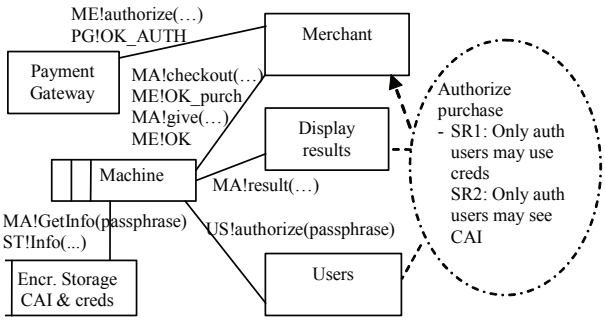

Figure 4-3 – Purchase problem (second try)

Figure 4-3 presents the modified problem. The context has been expanded to include the payment gateway.

Thinking about the satisfaction argument using the new problem diagram exposes the need for the following trust assumptions:

- TA2-1 – satisfaction of SR1 and SR2: Users will not expose the passphrase, ensuring that only authorized individuals use the credentials (SR1) and that authorized individuals may see the CAI (SR2)
- TA2-2 – satisfaction of SR2: The merchant implements the SET recommendations and securely stores the CAI. There is no practical way to bypass this security, regardless of storage medium (operational, backup, etc.)
- TA2-3 – satisfaction of SR2: The merchant's employees authorized to see the CAI will not reveal it.
- TA2-4 – satisfaction of SR2: The CAI never appears in the clear on the merchant's internal LAN – Local Area Network.
- The same trust assumptions that apply to the merchant also apply to the payment gateway.

Figure 4-4 presents the solution along with the four trust assumptions. To reduce the complexity of the diagram, we do not show the phenomena or the trust assumptions applied to the payment gateway. The trust assumptions are represented diagrammatically by an arc from the dependent domain to an oval containing a short summary of the depended-upon properties.

The risk presented by TA2-1, that the passphrase will not remain confidential, may or may not be acceptable. Personal experience indicates that it was not acceptable to at least one bank. When BNP (Banque Nationale de Paris) announced its SET implementation, the bank sent a smartcard reader to each customer who agreed to use SET. The user was required to know the passphrase, to insert the appropriate smartcard into the reader, and to know the PIN for the card. Learning the passphrase was not sufficient. One needed a second phrase (the PIN) and physical possession of the card.

The remaining trust assumptions are problematic. There is no practical way for a requirements engineer to examine every merchant and payment gateway company, so the assumptions must be accepted at face value.

The trust assumptions required to fulfill the security requirement might provoke a debate about whether a customer-side product based SET is worth constructing. Given that the CAI can be stored on the merchant's machine, the difference between a SET solution and the ubiquitous solution based on SSL (secure sockets layer) is not large. Using SET, it is more difficult for a merchant to change an order, but a dishonest merchant would have no problem creating new non-SET orders charged to the customer. Dishonest merchants and employees could sell the

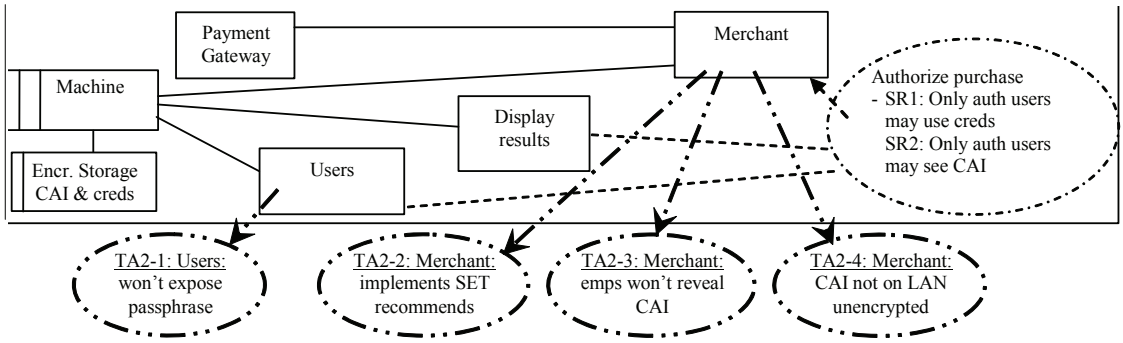

Figure 4-4 – Purchase problem (third try)

account information. Hackers could steal it. The requirements engineer can do nothing to mitigate the problems exposed by these trust assumptions. The stakeholders must decide whether the risks are acceptable. It is interesting to note that SET has been largely abandoned.

4.3 Chapter Summary

This chapter introduced one of our contributions: trust assumptions. We have provided an approach for using trust assumptions when reasoning about the satisfaction of security requirements. The approach uses the strong distinction between system requirements and machine specifications found in problem frames, permitting the requirements engineer to choose how to conform to the requirements. The trust assumptions embedded in the solution inform requirements engineers, better enabling them to choose between alternate ways of satisfying the functional requirements while ensuring that vulnerabilities are removed or not created. Finally, trust assumptions provide a foundation for making informal satisfaction arguments about the security of a proposed system.

The informal arguments presented in this chapter suffer from three flaws. The arguments have a very informal structure, and are not amenable to analysis. Justification of trust assumptions can introduce other trust assumptions, and this is not accounted for. Finally, there is no systematic exploration of the linkages between the argument and the trust assumptions.

Our proposed security requirements framework, described in the next two chapters, addresses these problems by:

- better defining security requirements and relating them to security goals.
- placing security requirements in a framework that explicitly permits iteration and requirements replacement.
- adding a formal security satisfaction argument that incorporates trust assumptions into the premises.
- adding a recursive informal satisfaction argument that permits one to argue the validity of trust assumptions, perhaps by (recursively) creating other trust assumptions.

Chapter 5. A Security Requirements Framework

The literature review in Chapter 3 and the discussion of trust assumptions in Chapter 4 exposed several security requirements problem areas:

- Multiple definitions of security requirements
- Inconsistent and difficult to understand satisfaction criteria for security requirements
- No structure for verifying that a system can satisfy the security requirements
- No explicit inclusion of the analyst's trust assumptions
- A general lack of a clear pathway for deriving security requirements from business goals.

We propose a security requirements framework to address these problems[6], facilitating an understanding of the elicitation, validation, and verification of security requirements and other artifacts by integrating the concepts of the two disciplines of requirements engineering and security engineering.

The framework takes two concepts from requirements engineering: the concept of business goals that are operationalized into functional requirements while applying appropriate constraints, and the concept of satisfaction (or adequacy) arguments. From security engineering, the framework concept of assets, together with threats of harm to those assets. In our framework:

- Security goals and security requirements aim to protect assets from harm.

[6] The framework was first described in (Moffett & Nuseibeh, 2003) and substantially elaborated in (Moffett, Haley et al., 2004)

- Primary security goals are operationalized into primary security requirements, which take the form of constraints on the functional requirements sufficient to protect the assets from identified harms. Primary security requirements are, consequently, preventative.
- Feasibility, tradeoff, and conflict analyses (Redwine, 2006: pg 81) may lead to the addition of secondary security goals, which will (eventually) manifest themselves as additional functional and/or secondary security requirements. Secondary security goals and requirements may call for detective or preventative measures, a possibility discussed further below.
- Security satisfaction arguments show that the system can respect the security requirements.

The framework assists with understanding the place of security requirements within the development of an individual application, along with the relationships between the security requirements and other artifacts produced during development.

5.1 Framework vs. Process

This thesis proposes a framework within which development processes might function. One might think of the framework as a set of ordered milestones, indicating by when certain artifacts are to have been produced. The framework says that one should produce X and Y, and that one must produce X before one can produce Y, but it says nothing about *how* one produces X or Y. The *how* would be a *process*: a set of steps that if followed should allow one to make the transition from X to Y. The distinction is important because most organizations have a process they follow, so imposing one would be difficult. However, a process can be fitted into this framework if the process produces visible functional requirements, and if the user of the process will produce context and problem diagrams along with the other artifacts already produced by the process.

One difficulty with describing a framework is that many steps or outputs are abstract or unspecified. To overcome this difficulty, in this thesis we instantiate the framework using a

combination of Goal-Oriented Requirements Engineering (van Lamsweerde, 2001) and Problem Frames (Jackson, 2001), describing it in terms of a set of activities.

5.2 Definition of Security Goals

Security goals are derived from the business goals of the system (Allen, 2001). Some number of actors, operations, and objects will be required to satisfy the business goals. To paraphrase somewhat the introduction to this thesis, security goals arise when stakeholders establish that they wish to avoid *harm* to some objects in the context of the system, be they tangible (e.g., cash) or intangible (e.g., information), that have direct or indirect value. Objects valued in either way are called *assets*, and the stakeholders naturally wish to protect themselves from any harm that might come from abusing these assets.

Harm may not be to the asset itself (direct harm), but instead may be a consequence of some misuse or abuse of the asset (indirect harm). Examples of indirect harm include damage to reputation caused by exposure of flawed hiring policies, loss of contracts caused by exposure of pricing or costing information, or loss of trade secrets through the theft of some newly designed widget. In other words, one is not necessarily protecting assets *from* harm, but is instead protecting *against* harm caused by abuse of assets. Consider the case where the asset is confidential information, such as the design for an unreleased product. Abusing the information by making it public does not harm the information, but future revenue of the company could be adversely affected. Now consider the case of the destruction of a building. One harm is direct: the cost of replacing the building. However, other harms are possible, such as (again) the loss of future revenue caused by the inability to do business. In this case, there are multiple harms, each with diverse risk and impact, which might require different protections.

One set of security goals describe conditions that must be avoided in order to keep the level of harm to an acceptable level. For example, tangible assets might be destroyed, stolen, or modified; the harm is the loss of the asset itself (direct harm). Information assets might be destroyed, revealed, or modified; the harm could be the loss of the asset (direct harm) or the consequences of exposing the asset (indirect harm).

The security community has enumerated some general security concerns, labeling them with the letters **C**, **I**, **A**, and more recently a second **A** ((Pfleeger & Pfleeger, 2002) and other security textbooks):

- **C**onfidentiality: ensure that an asset is visible only to actors authorized to see it. Confidentiality is larger than 'prevent read access to a file'. For example, it includes controlling visibility of a data stream on a network, and of papers on someone's desk.
- **I**ntegrity: ensure that the asset is not corrupted. Integrity is larger than 'prevent write access to a file', for example including ensuring that transactions that should not occur indeed do not, that the contents of backup media are not changed, that incorrect entries in a paper-based accounting system are not made, and data streams are not modified between their endpoints.
- **A**vailability: ensure that the asset is readily accessible to agents that need it, when they need it. A counterexample is preventing a company from doing business by denying it access to something important, such as access to its computer systems or its offices.
- **A**uthentication: ensure that the identity of the asset or actor is known. A common example is the simple login. More complicated examples include mutual authentication (e.g., exchange of cryptography keys), and intellectual property rights management.

By connecting these general concerns to the assets implicated in a system, and then postulating *actions* that would violate these concerns (that would be an abuse of the asset), one can construct extended descriptions of possible threats to assets. These *threat descriptions* (Haley, Laney et al., 2004c) are phrases of the form *performing action X on/to/with asset Y could cause harm Z*. Threat descriptions permit a form of asset-centered threat modeling, and are represented by a three-element tuple: the asset, the action that will exploit the asset, and the subsequent harm. Threat descriptions are generated by enumerating the assets involved in the system, then for each asset, listing the *actions* that exploit the asset to cause direct or indirect harm. The action is derived from the security concern; it does not name a specific vulnerability or attack path. For example, one can imagine *erasing* (an action related to the integrity concern)

the *customer records* (the asset) of a company to cause *loss of revenue* (the harm). A set of security goals is found by negating the threat descriptions, which in goal-oriented requirements engineering terms makes them into *prevent* (or *avoid*) goals.

Another set of security goals can be found by combining management control principles and application business goals. Management control principles include common security principles such as *least privilege* and *separation of duties* (NIST, 1995: pg 109). Application business goals will determine the applicability of management control principles to the system, for example by defining those privileges that are needed for the application, and excluding those that are not. An organization may already have done the analysis and published policies that apply to assets in a system. The security goal is a statement that the policies and/or principles be applied where appropriate in the system.

Note that legitimate stakeholders may have conflicting security goals. The set of relevant security goals may be mutually inconsistent, and inconsistencies will need to be resolved during the goal analysis process before a set of consistent requirements can be obtained.

Looking at the goals of attackers could be useful when determining security goals for the system, for example when enumerating assets or quantifying harm, but we do not consider them a part, even negated, of the set of security goals. The goals of the system owner and other legitimate stakeholders are not directly related to the goals of attackers, because security is not a zero sum game like football. In football, the goals won by an attacker are exactly the goals lost by the defender. Security is different; there is no exact equivalence between the losses incurred by the asset owner and the gains of the attacker. To see this, look at two examples:

- Robert Morris unleashed the Internet Worm, causing millions of dollars of damage, apparently as an experiment without serious malicious intent (Spafford, 1989). The positive value to the attacker was much less than the loss incurred by the attacked sites.
- Many virus writers today are prepared to expend huge effort in writing a still more ingenious virus, which may cause little damage (screen message "You've got a Virus").

> Generally, there is no simple relationship between the gains of a virus writer and the losses incurred by those who are attacked.

The consequences of security not being a zero sum game are twofold: The first is that the evaluation of possible harm to an asset can generally be carried out without reference to particular attackers; one needs only to determine that harm can be incurred. The second is that the goals of attackers cannot be solely used to arrive at the security goals of a defender to prevent harm; further consideration is necessary to determine whether and what harm is incurred if the attacker satisfies his or her goals.

5.3 Definition of Security Requirements

We define security requirements as constraints on the functions of the system, where these constraints operationalize one or more security goals.

Security requirements operationalize the security goals as follows:

- They are constraints on the system's functional requirements, rather than themselves being functional requirements.
- They express the system's security goals in operational terms, precise enough to be given to a designer/architect. Security requirements, like functional requirements, are prescriptive, providing a *specification* (behavior in terms of phenomena – see Chapter 2 Section 2.1) to achieve the desired effect.

The fact that security requirements are constraints on functional requirements rather than separate functional requirements is important for validation of the functional requirements. Validating a set of functional requirements in the face of constraints is easier than validating requirements consisting of the original functional requirements and the additional functional requirements added for security. In the first case, one need check only that after the functions are constrained, they still do what they originally were intended to do. In the second case, the system designer decides how the requirements interact and how the interactions are realized. Only after design is complete can one check to see if functionality has changed beyond

acceptability. Adding constraints to particular functional requirements (ones where the assets in question are implicated) keeps interaction analysis a part of requirements engineering.

5.4 From Security Goals to Security Requirements

We propose an iterative hierarchy of security goals and security requirements. The first iteration produces *primary* goals and requirements that are derived from the business goals and functional requirements. These goals and requirements are *primary* in the sense that if the resulting system respects the primary security requirements, then the system will satisfy the primary security goals.

Further iterations produce *secondary* security goals and requirements. They are added for one or both of the following reasons: 1) to enable construction of an acceptable *satisfaction argument* for the satisfaction of primary security requirements, or 2) to permit an acceptable *feasible realization* of the primary security requirements. Satisfaction arguments are discussed later in this chapter and more fully in Chapter 6.

The term *feasible realization* takes into consideration technical feasibility, cost/benefit plus risk, and stakeholder tradeoffs (Redwine, 2006). It may be that there is no practical way to respect a constraint and thereby prevent the harm; destroying a computer room with an atomic explosion comes to mind. Perhaps stakeholders do not agree on the goals or requirements. Risk analysis may indicate that the cost of respecting a security requirement is excessive, in which case the analyst may decide to detect violation after the fact, and then both recover from and repair the breach. Availability requirements are a good example - many such requirements do not *prevent* loss of availability, but instead imply a recovery capability. For example, secondary security goals would be added to the system to require that backups be taken and to manage these backups. Analysis of the secondary security goals may lead to the addition of secondary security requirements. This is, of course, a recursive process.

Secondary security goals and security requirements are not secondary in terms of importance, but are instead secondary because they exist to enable satisfaction, to an acceptable level, of the primary and hierarchically superior secondary security requirements.

Secondary security goals can provoke the modification of existing functional requirements, or the addition of new functional requirements. This will occur when satisfaction of the secondary security goal requires addition of new management capabilities (e.g., management of authentication mechanisms), alteration of system-level workflows, or addition of new assets that the system must accommodate in some way.

It is very important to note that secondary security goals and requirements supersede the primary security requirements, and can change the context and behavior of the system. For example, choosing to use attack detection instead of prevention implies that the primary security requirement will not be completely satisfied, as the attack will not be prevented. The choice means that the secondary goals and associated security requirements are considered *suitably equivalent* to the primary security requirements; they cover and replace (but do not delete) them. The decision to use detection instead of prevention could also change the behavior of the system because of the addition of domains and phenomena to facilitate detection.

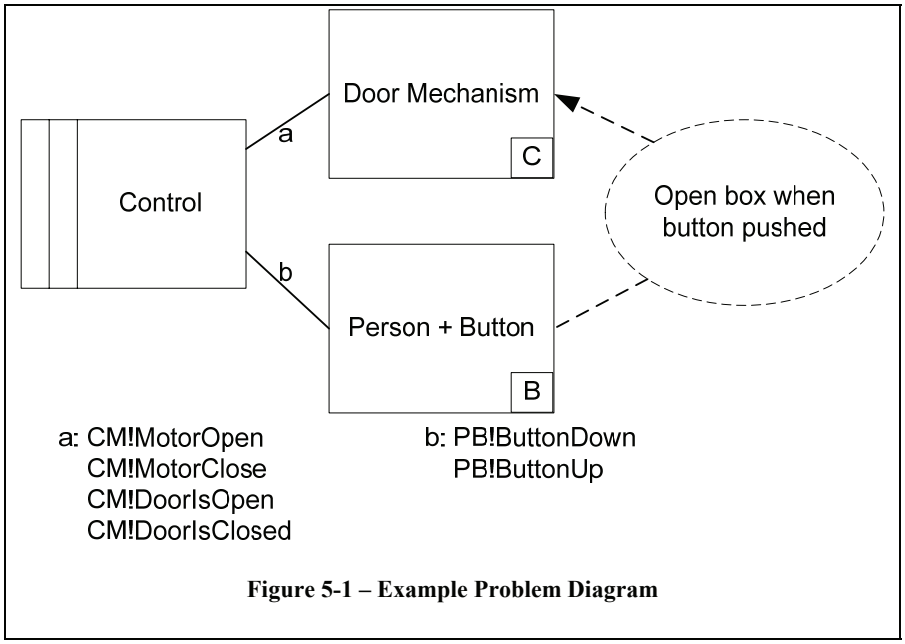

Figure 5-1 – Example Problem Diagram

5.5 Security Requirements and Context

We reiterate that security requirements are applied in the system context, which is larger than the software. A security requirement can affect many parts of the system, some completely outside the software to be constructed. A variant of Jackson's problem frame diagrams (Jackson, 2001) is used to represent the system context. To refresh the reader's memory, the sample problem diagram from Chapter 2, Section 2.1 is reproduced here as Figure 5-1. The boxes are domains. Lines connecting the boxes represent interfaces, which are labeled with lower-case letters. The phenomena on the interfaces are listed as a set with the appropriate label. The domain controlling a given phenomenon is indicated using a letter or letters in front of an exclamation mark ('!').

We chose problem frames to represent the context because problem frames permit us to incorporate behavior specification of real-world domains at a level of formality ranging from none to very. The behavior specification is necessary for constructing the satisfaction arguments we propose. We are not claiming that use of problem frames is necessary. We are not, however, aware of a better substitute.

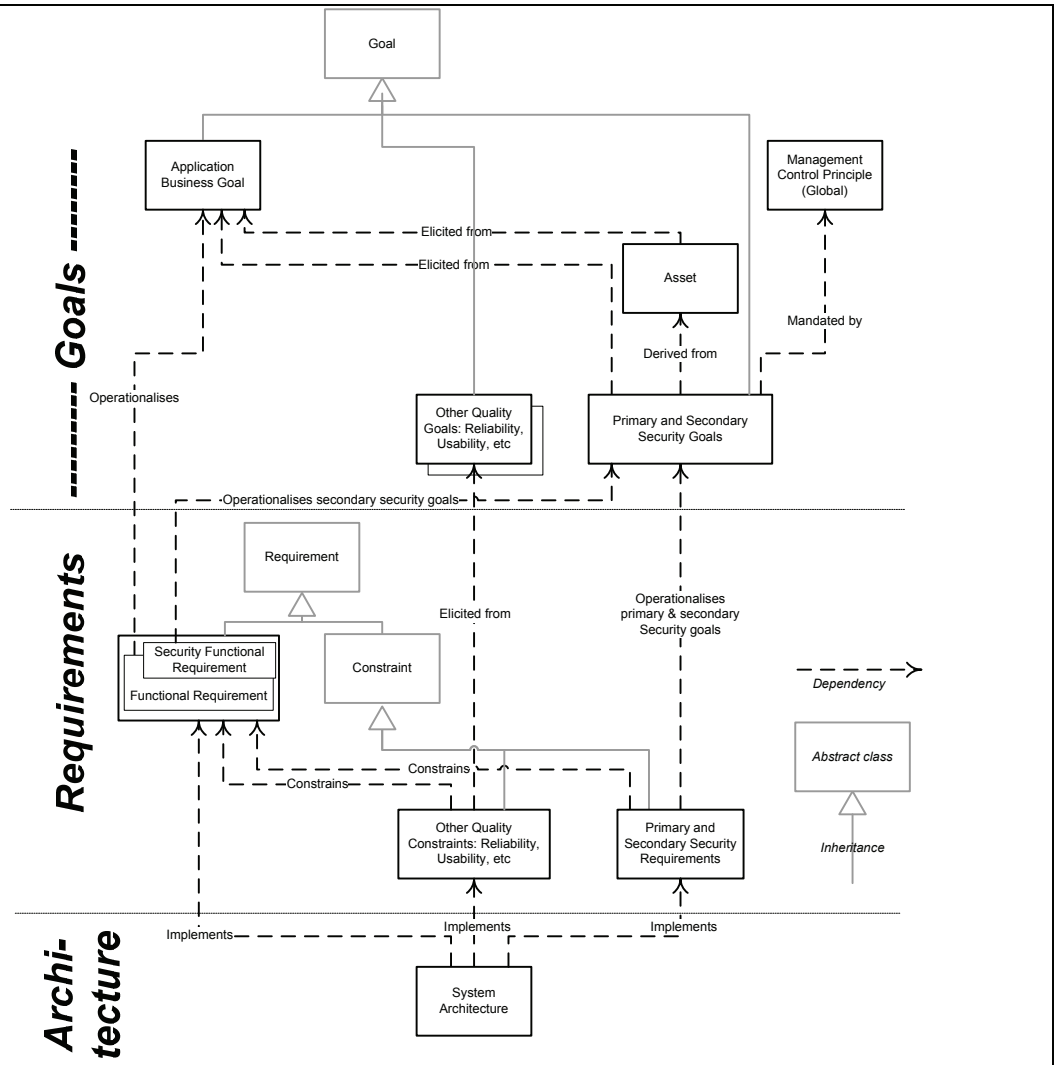

Figure 5-2 – Security Requirements Core Artifacts (Class diagram)

5.6 Development Artifacts and Dependencies

All system development processes have recognizable stages that produce artifacts.

5.6.1 Core Artifacts

Core artifacts are successively closer representations of a working system. They are ordered in the abstraction hierarchy shown in Figure 5-2, progressing from the most abstract to the final concrete working system. At early stages, core artifacts are typically documents or prototypes.

The final core artifact is the working system itself, consisting of a combination of physical and software items.

Two sets of core artifacts are of most interest to this thesis. On the mainstream requirements engineering side, one finds descriptions of goals, requirements, and the system (in the large) context & architecture. On the security engineering side, one finds assets and control principles.

5.6.2 Support Artifacts

Support artifacts are artifacts that help to develop, analyze, or justify the design of a core artifact. They may include formal analysis, informal argument, calculation, example or counter-example, etc. They are by-products of processes, whose aim is to help produce verified and valid core artifacts.

5.6.3 Dependencies between Artifacts

There are dependencies in the artifact hierarchy. For example, an operationalized requirement is dependent upon a higher-level goal from which it has been derived, because alteration of the goal may cause alteration of the requirement. We call this kind of dependency *hierarchical dependency*.

There is also a reverse kind of dependency: *feasibility*. If it proves impossible to implement a system that sufficiently satisfies a requirements specification, then this will force a change in the goals or requirements. The higher-level artifact is dependent on the feasibility of the artifacts below it in the hierarchy.

These dependency relationships have an important implication for the structure of development processes. If an artifact is dependent upon the implementation of another artifact for its feasibility, then if the implementation is not feasible, there must be an iteration path in the process back to the ancestor from its descendant.

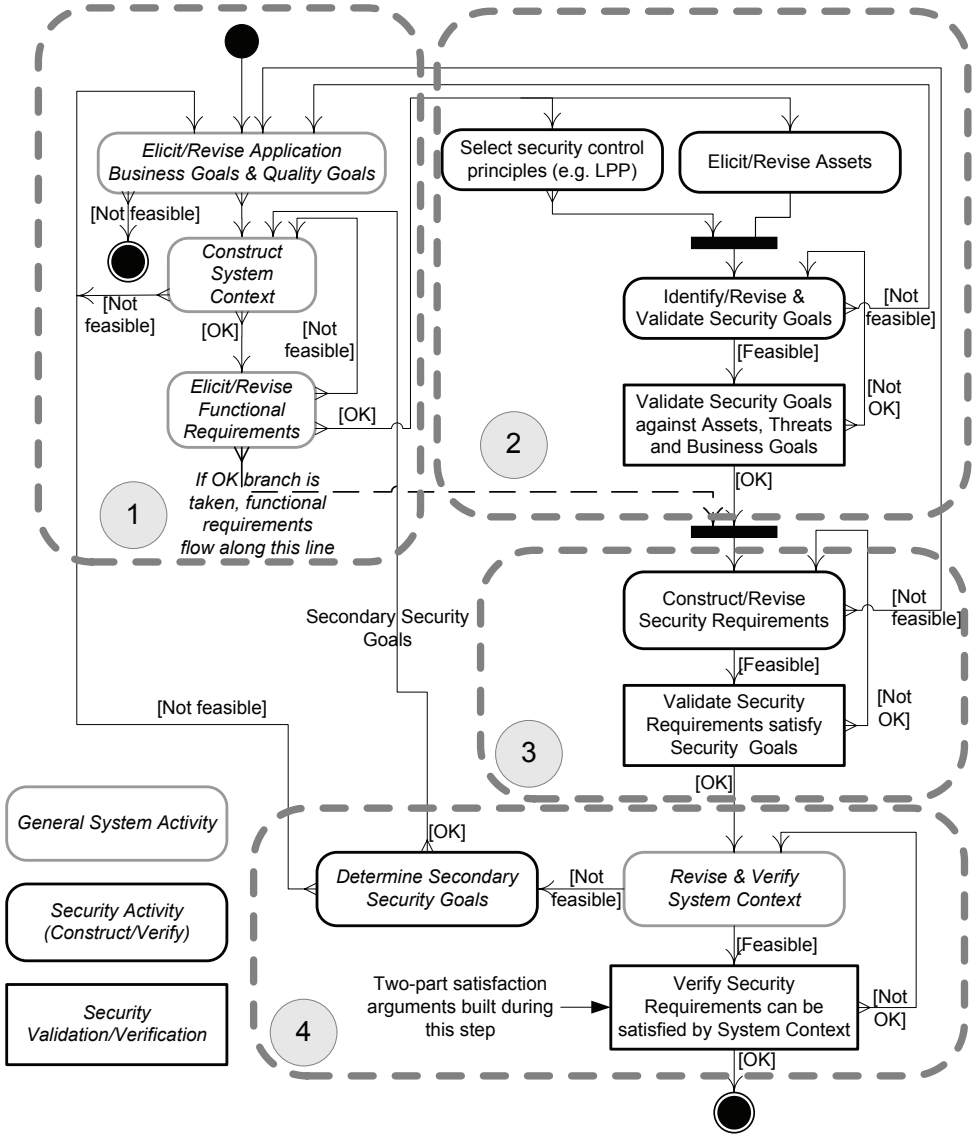

Figure 5-3 – Security Requirements Process Overview (Activity Diagram)

5.7 Framework Overview

Figure 5-3 shows an ordered set of activities for moving from business goals to security requirement satisfaction arguments. Boxes in the figure represent activities that produce artifacts. Typically, a box in the figure has two exits, one for success, and one for failure. Failure can be one of two kinds. The first failure kind is that it is not feasible to create a

consistent set of the artifacts called for by that activity. The second kind is that validation of the artifacts against a higher level – such as validation that security requirements satisfy security goals – shows that they fail to meet their aims. For example, one might be unable to construct a set of validated functional requirements from the business goals. Alternatively, one might fail to construct adequate security requirement satisfaction arguments. Iteration may cascade upwards if the problem cannot be resolved at the preceding step.

There are four general stages in the activity diagram. Although one could describe these stages in terms of the artifacts produced, along with the ordering between them, it is clearer to describe them in terms of what is the goal of the activities in each stage. The activities are:

- Stage 1: identify functional requirements
- Stage 2: identify/revise security goals
- Stage 3: identify/revise security requirements
- Stage 4: verify that the security requirements can be satisfied by the system, by constructing satisfaction arguments.

Each stage is discussed in more detail below.

5.7.1 Stage 1: Identify Functional Requirements

The only requirement the framework places upon the development process is that the engineer produce problem diagrams as described in Chapter 2 Section 2.1. How the requirements engineer gets to this point is open.

5.7.2 Stage 2: Identify/Revise Security Goals

There are three general steps required to identify the security goals: identify candidate assets, select the management principles to apply, and then determine the security goals. The result is a set of security goals, which are validated by ensuring that the business goals remain satisfied.

The first iteration through this stage results in the generation of primary security goals. Subsequent iterations result in secondary security goals, either because of asset analysis or because they were passed up from a temporally previous step, which are traceable, perhaps through multiple levels and through security requirements, to the original, primary, security goal(s).

5.7.2.1 Identify Candidate Assets

The goal of this step is to find all the objects in the system context that might have value, direct or indirect. In general, assets consist of all the information objects stored in or accessed by the system and any tangible objects such as the computers themselves. An object has direct value when the potential harm described in a threat is to the object itself. An object has indirect value if realizing a threat involving that asset causes harm somewhere else, such as to revenue, to costs, or to reputation. An object can have both direct and indirect value; when money is taken from a bank, the bank both loses the money and has its reputation harmed.

One potential asset might contain, or enclose, other potential assets. A good example is a database that contains individual information assets. Another example is backup media, which can contain any number of information assets.

5.7.2.2 Select Management Principles

The functions that the system is to provide must be compared to the management principles that the organization wishes to apply. These principles might include (not intended to be an exhaustive list):

- separation of duties (NIST, 1995: pg 109) – dividing roles and responsibilities to ensure that no one person has sufficient privilege to both start and complete important transactions.
- least privilege (Ibid) – ensuring that a person has only what is required to do his or her job, in both privilege to *know* and privilege to *do*.

- audit trails (ISO/IEC, 1999b: pg 181; NIST, 1995: pg 213) – recording information about events of potential security interest, such as who did what, when.
- Chinese wall (Brewer & Nash, 1989) –not permitting operations if a potential conflict of interest exists, such as an analyst giving advice to both company A and competitors of company A.

The sector the system is being designed for may have standard management principles, such as no outside network connections, or no removable media capabilities on any computer. In addition, the organization might have already done a harm/risk analysis and developed organization-wide security policies for asset types. Which global policies to apply within the system under consideration must be identified and fed into the next step.

5.7.2.3 Determine/Revise Security Goals

When developing security goals, one should determine whether a harm analysis must be done for the assets. If the analysis has been done elsewhere (e.g., organization-wide policies) and if the assets are covered by the policies, then a list of security goals is generated by applying the management principles to the assets and business goals of the system. The result is a set of *achieve* goals with forms similar to "achieve Separation of Duties when paying invoices" or "audit all uses of account information."

If the analysis done elsewhere is not considered sufficient, one should do a harm analysis. In general, harm results from the violation of one or more of the security concerns described in Section 5.2. For information assets, these concerns are *confidentiality*, *integrity*, and *availability*. The concerns are similar for tangible assets: *exposure*, *modification*, and *deprivation* (theft or destruction). These concerns are used to enumerate the threat descriptions. One asks questions of the form "what harm could come from violating the [insert concern here] of [insert asset here]?" Answers to these questions are threat descriptions, which are represented as tuples of the form `{action, asset, harm}`.

Threats may have a time element, stating that the harm will occur only if the violation occurs before or after some point, or within some interval. For example, a company's earnings report is

confidential (and therefore valuable) only up to the moment it is made public. The time element is important when looking for and countering vulnerabilities, as it gives an indication of how severe a given vulnerability is and what measures are appropriate for countering the vulnerability.

It is worth noting again that an object might not have any value in itself, but instead is valued by the harm caused indirectly to something else. For example, information about the amount of money paid to redecorate the company president's office has no intrinsic value, but may be highly valued because exposing the figure could damage the reputation of the company. In other words, when evaluating threats (how assets are associated with harms), one must look for direct and indirect effects.

5.7.3 Stage 3: Identify/Revise Security Requirements

Recall that we define security requirements as constraints on functional requirements that are needed to satisfy applicable security goals. To determine the constraints, we must determine which security goals apply to which functional requirements, which means we must know which assets are implicated in fulfilling a particular functional requirement. We use Jackson's problem diagrams (Jackson, 2001) for this purpose; these diagrams describe the *system context*. We do not attempt to identify a particular problem class, but instead describe domains, their interconnections, shared phenomena, and requirements into a system problem diagram.

A simple example of a functional requirement decorated with such a constraint is:

```
The system shall provide Personnel Information only to members of Human
Resources Dept.
```

The constraint ("`only to …`") is attached to the function ("`provide Personnel Information`"); it makes sense only in the context of the function. One might also impose temporal constraints:

```
The system shall provide Personnel Information only during normal office
hours
```

and complex constraints on traces, for example the Chinese Wall Security Policy, (Brewer & Nash, 1989):

```
The system shall provide Personal Information only to any person who has
not previously accessed information about a person in a different
subsidiary.
```

Availability requirements might need to express constraints on response time:

```
The system shall provide Personnel Information within 1 hour for 99% of
requests.
```

Note that this availability requirement differs only in magnitude from a Response Time quality goal, which might use the same format to require a sub-second response time.

Once a set of security requirements has been developed, one must validate that the security requirements satisfy the security goals. This would be done using satisfaction arguments appropriate to the level of formality used to describe the goals. Given that goals are often written in plain text, the arguments could have a form similar to our inner arguments (see Section 6.1.2). How these arguments are expressed is left open to the designer of the process to be used, and not defined within our framework.

In the same fashion as security goals, the first iteration through this stage results in primary security requirements. Subsequent iterations generate secondary security requirements.

5.7.4 Stage 4: Verify Security Requirements against System Context

It is important to verify that the security requirements are satisfied by the system as described by the context. We propose the use of formal and structured informal argumentation for this verification step: to convince a reader that a system can satisfy the security requirements laid upon it. These arguments, called *satisfaction arguments* and discussed more completely in Chapter 6, are in two parts. The first part, the *outer argument*, consists of a formal argument to prove a system can satisfy its security requirements, drawing upon claims about the domains in a system, and assuming the claims are accepted. The second part, the *inner argument*, consists of structured informal arguments, supporting the claims made in the formal argument about the system's behavior and characteristics. Building on our understanding of security requirements, the satisfaction arguments assist with identifying security-relevant system properties, and determining how inconsistent and implausible assumptions about them affect the security of a system.

Stage 4 begins by verifying that the functional requirements, as constrained by the security requirements, remain satisfied by the system as described by the system context. Once the functional requirements are shown to be satisfied, the security requirements themselves are verified by construction of the two-part argument described above. If it proves impossible to construct valid arguments, the context is revisited. If necessary, secondary security goals are added to correct problems, and the context is revisited. If this turns out to be infeasible, it is necessary to return to the beginning, revisiting the business goals. See the next section for additional details.

5.8 Iteration

One reason that an analyst may fail to construct a convincing satisfaction argument is that there is not enough information available to justify the claims (trust assumptions) made. For example, to justify a claim that users are authenticated, there must be some phenomena exchanged between the user and the rest of the system. The choice of phenomena and behavior is a design decision that may have a significant impact on the system architecture and context. For example, it is possible that architectural choices are imposed that extend the context to include all of IT management. For these reasons, the framework assumes that the process includes *Twin Peaks* requirements/design iterations (see Chapter 2 Section 2.3), asking the designers to add more detail into the system context so that claims can be justified. These iterations move from stage four to stage one, and from there back through the activities.

The details added during a requirements/design iteration may require new functions to be added to the system, thus generating new functional requirements. Continuing the authentication example from above, assume the designers choose a retinal-scanning authentication technique. The designers add domains and phenomena to the context to describe how authentication takes place from the point of view of the user (in problem space). However, one cannot necessarily stop at the addition of phenomena. The authentication system must be managed. New assets have been added to the system, for example the retina description information. New domains have been added: for example the administrators and the retinal scanners. New goals have been

added to the system: assure that the functional additions serve their purpose. These additions could easily have an impact on system security, precipitating the addition of new security goals, or changing existing ones.

When the requirements engineer or designer alters the context, they (might) add secondary security goals to the system to ensure that the preconditions or consequences of the alterations become part of the requirements for the system. A new requirements and asset analysis must be performed. Continuing the authentication example, a goal similar to `manage authentication database` would be added in stage 4. The process would then restart in stage 1 with a reanalysis of the context and functional requirements, to understand the consequences of the new goal. New assets (e.g., the authentication data) would be found in stage 2, and then new security goals to protect the assets and new security requirements to constrain functional operations wherever the new asset appears would be added.

Another possibility is that the requirements/design iteration will establish that there is no feasible way to satisfy the security requirement(s). In this case, the designers and the stakeholders must come to an agreement on some acceptable alternative, such as a weaker constraint, attack detection, and/or attack recovery. They would add appropriate secondary security goals to the system, probably resulting in new secondary security requirements. The resulting secondary security goals and requirements cover the ones that were not feasible. As the new secondary goals and requirements are considered suitably equivalent to the originals, satisfying the new ones is considered to satisfy the originals.

Clearly the ‘secondariness’ of any goals added must be remembered. If the hierarchically superior (‘more primary’) security requirement is changed, then the secondary security goals may need changing. For example, if authentication became unnecessary, then the `manage authentication database` goal should be removed, with the consequential removal of derived functional requirements and assets.

Finally, it is possible that no feasible way to satisfy a security requirement exists, and no agreement can be reached on alternatives. In this case, one must return to the original business

and quality goals of the application, modifying the initial conditions to change the assets implicated in or the security goals of the system. Alternatively, one might decide not to build the system.

5.9 Worked Example

We use an example of a Personnel Information display system to illustrate the framework. The example begins in this chapter, working through stages 1 through 3, and then continues with stage 4 in Chapter 6. We begin by stating the business goals for a simple system. Next, we present the functional requirements, and then derive the system security requirements by applying the organization's security goals to the functional requirements.

Where appropriate, we omit from the discussion processes that are not part of the framework.

5.9.1 Stage 1: Identify Functional Requirements

We begin this example assuming that the work in this stage has already been carried out. The assumption is that the business goals have been elicited and that there is only one goal:

```
BG1: Provision of people's personnel information to them.
```

We further assume that the stakeholders agree that there is one functional requirement:

```
FR1: On request from a Person (instance of People), the system shall
provide HR data (persData) for a specified payroll number (persNumber) to
that Person.
```

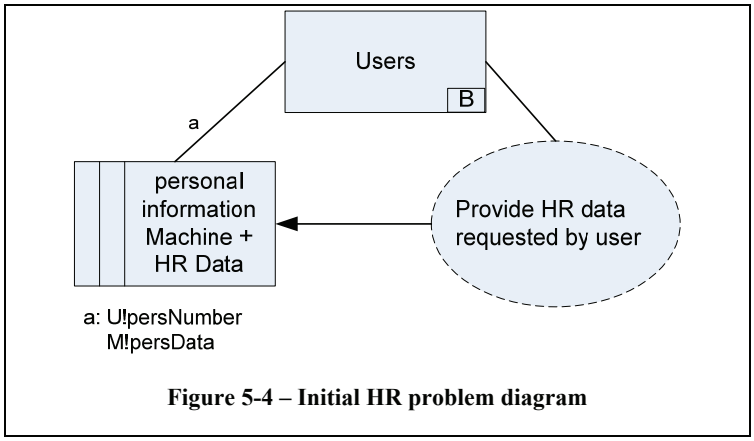

Figure 5-4 – Initial HR problem diagram

Figure 5-4 shows the problem diagram for the requirement and context. There are two phenomena of interest. The first, `U!persNumber`, is the user's request for personnel information. The second, `M!persData`, is the information returned by the request.

5.9.2 Stage 2: Identify/Revise Security Goals

Discussion with the stakeholders shows that, ignoring physical assets such as the computers and the buildings, there is only one asset implicated in the system: `persData`, an information asset.

We now list the threat descriptions (*`action`* `on` *`asset`* `to cause` *`harm`*). Actions that might cause harm that can be done to `persData` are exposure (loss of confidentiality), alteration (loss of integrity), and denial of service (loss of availability), resulting in various harms. Some possible threat descriptions are:

```
Confidentiality threat descriptions:
     {unauthorized exposure, persData, expense of privacy violation lawsuit)}
     {unauthorized exposure, persData, expense of discontented employee}
Integrity threat descriptions:
     {unauthorized alteration, persData, expense of salary underpay lawsuit}
     {unauthorized alteration, persData, expense of excess salary}
     {unauthorized alteration, persData, expense of information restore}
Availability threat descriptions:
     {~available, persData, expense of late salary payments}
     {~available, persData, expense of discontented employees}
     {~available, persData, expense of unfiled government-mandated reports}
```

The system owner considers all of these threat descriptions to represent significant risk, and therefore wishes to avoid them.

The confidentiality threat descriptions give rise to the security goal

`SG1: prevent unauthorized exposure of PersData`

Likewise, the next integrity threat descriptions give rise to the goal

`SG2: prevent unauthorized alteration of PersData,`

The availability threat descriptions give rise to

`SG3: prevent denial of access to PersData by authorized persons.`

5.9.3 Stage 3: Identify/Revise Security Requirements

The next step is to derive security requirements from the combination of business goals and security goals. Recall that security requirements constrain the function called for by a functional requirement that operationalizes a business goal. Investigating the word *authorized* in SG1, the requirements engineer determines that an individual is permitted to see only his or her own data. Furthermore, assume that the data for a person contains certain statistical information such as the difference of that person's salary from the mean salary of the department. It is possible that this information will expose other employees' salary because of lack of sufficient statistical aggregation (e.g. a small department), and therefore the system must not display this information to the employees. This complexity leads the stakeholders to agree that personnel information is to be interpreted by a trained HR staff member, and not be exposed directly to the employee. From these choices and by applying SG1 to FR1, one derives the security requirement (constraint) SR1: [FR1] only to HR staff. An informal argument that this requirement satisfies the security goal is: confidentiality of personnel data implies that people in general cannot be allowed access to this information, but HR staff can be relied upon to maintain its confidentiality. Therefore, a constraint that permits HR staff, but nobody else, to access it will satisfy the security goal.

SG2 is less obvious. There is no functional requirement that permits modification of PersData, so one might assume that there is no functional requirement for SG2 to constrain. However, FR1 does *display* information, and clearly one wishes that the information displayed be an exact analog of the stored information. From this wish, one can formulate the security requirement SR2: [FR1] only if the displayed information is a correct representation of stored information.

Applying SG3 to FR1, the (somewhat arbitrary) security requirement SR3 is derived: SR3: [FR1] within 60 minutes of its request.

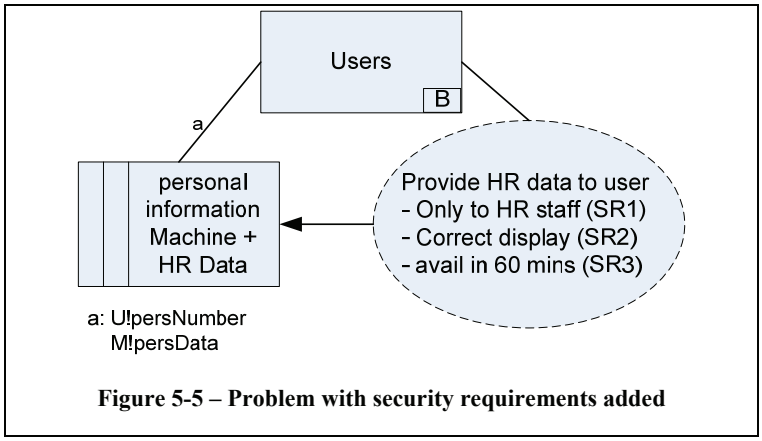

Figure 5-5 – Problem with security requirements added

Figure 5-5 shows the problem diagram, modified to show the constraints.

At this point, one should validate that the original business goals are still adequately satisfied in the face of the security requirements. It is possible that a security requirement is so constraining that the system no longer meets its business goals. For example, SR1 could have been the constraint `[FR1] only to the Chief Information Officer`. This constraint is arguably so severe that the system would not meet its business goals.

5.10 Chapter Summary

This chapter presented the second of our contributions, our security requirements framework. The framework incorporates a practical definition of security requirements that have clear yes/no satisfaction criteria. It also makes the role of system context explicit.

The next chapter completes our explanation of our framework, focusing on Stage 4 – verification that security requirements can be satisfied by the system. We describe our third contribution, security satisfaction arguments, then complete the remainder of our example.

Chapter 6. Security Requirement Satisfaction Arguments

As was said in Chapter 5, it is important to verify that the security requirements are satisfied by the system as described by the context. We use *satisfaction arguments* for this purpose. Chapter 4 introduced informal satisfaction arguments based around trust assumptions. This chapter extends those ideas, proposing the use of formal and structured informal argumentation for this verification step: to convince a reader that a system can satisfy the security requirements laid upon it. These satisfaction arguments are in two parts. The first part, the *outer argument*, consists of a formal argument to prove a system can satisfy its security requirements, drawing upon claims about a system, and assuming the claims are accepted. The second part, the *inner argument*, consists of structured informal arguments to support the claims made in the first argument about system behavior and characteristics. Building on our understanding of security requirements, the two-step satisfaction arguments assist with determining security-relevant system properties, and how inconsistent and implausible assumptions about them affect the security of a system.

6.1 Trust Assumptions & Arguments

A security satisfaction argument must satisfy two goals: 1) given a collection of domain properties and trust assumptions, to show that a system can be secure, and 2) have a uniform structure for the satisfaction argument so that the effects of trust assumptions are made more explicit. We satisfy these goals by splitting the satisfaction argument into two parts: a *formal outer argument* that is first constructed, and informal structured *inner arguments* that are constructed next to support the outer argument. If acceptable inner arguments to support the outer argument cannot be constructed, then one must reject the outer argument.

Chapter 4 presented *trust assumptions*, which are claims about the behavior or the type of domains included in the system, where the claims are made in order to satisfy a security requirement. These claims represent an analyst's trust that domains behave as described. Trust assumptions are in the end the analyst's opinion, and therefore assumed to be true. At some point, the inner arguments must stop, depending on these unsupported assumptions. We are now able to define what trust assumptions are in our framework: unsupported statements about the behavior of the system, made in order to create a convincing inner argument.

6.1.1 The Outer Argument

The formal outer argument uses claims about the behavior of the system (interplay of phenomena) to demonstrate that a security requirement (a constraint) is satisfied. The formal argument is expressed using some logic chosen by the requirements engineer, where the premises are formed from claims about domain properties and behavior, and the conclusion is the satisfaction of the security requirement. For simplicity, we use propositional logic in this chapter, resulting in the outer argument being a proof of the form:

```
(domain behavior premises) ⊢ (security requirement(s))
```

6.1.2 The Inner Arguments

Inner arguments are informal arguments made to support the claims used in the outer argument. This thesis proposes a form inspired by the work of Toulmin (1958), one of the earliest advocates and developers of a structure for informal human reasoning and argumentation. We chose Toulmin-style arguments for what might be considered an engineering reason: they are well suited for our purpose because other than requiring that an argument have a conclusion, they impose restrictions on neither what can be argued nor the logical system to which the argument must conform. Toulmin arguments facilitate the capture of:

- relationships between domain properties – the premises in the formal argument.
- the trust assumptions that either are, or eventually support, these premises.
- reasons why the argument may not be valid.

Toulmin et al. (Toulmin, Rieke, & Janik, 1979) describe arguments as consisting of:

1. *Claims*, the end point of the argument – what one wishes to convince the world of.
2. *Grounds*, providing any underlying support for the argument, such as evidence, facts, common knowledge, etc.
3. *Warrants*, connecting and establishing relevancy between the grounds and the claims. A warrant explains how the grounds are related to the claim, not the validity of the grounds themselves.
4. *Backing*, establishing that the warrants are themselves trustworthy. These are, in effect, grounds for believing the warrants.
5. *Modal qualifiers*, establishing within the context of the argument the reliability or strength of the connections between warrants, grounds, and claims. Modal qualifiers permit the introduction of rebutting circumstances.
6. *Rebuttals*, describing what might invalidate any of the grounds, warrants, or backing, thus invalidating the support for the claim.

Toulmin proposed a diagram for arguments that indicates how the parts fit together (Toulmin, 1958), shown in Figure 6-1. The lines in the figure show 'movement' of the argument from left (grounds) to right (claims). Intersections show where parts of the argument support or detract from the main line. Warrants support using grounds to justify a claim, but rebuttals weaken the argument.

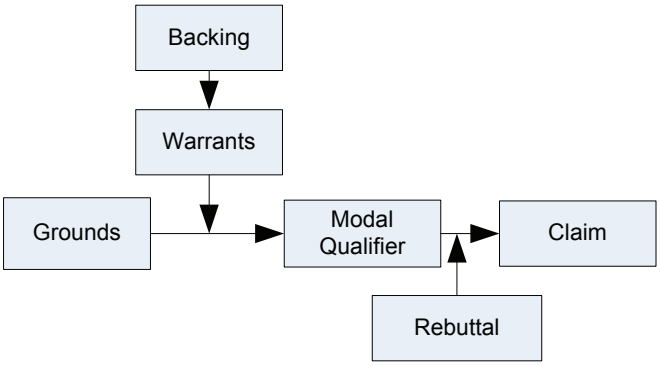

Figure 6-1 – Generic Toulmin-form argument

The items in an argument are summarized by Toulmin et al. (1979) as follows: "The *claims* involved in real-life arguments are, accordingly, *well founded* only if sufficient *grounds* of an appropriate and relevant kind can be offered in their support. These grounds must be connected to the claims by reliable, applicable, *warrants*, which are capable in turn of being justified by appeal to sufficient *backing* of the relevant kind. And the entire structure of argument put together out of these elements must be capable of being recognized as having this or that kind and degree of certainty or probability as being dependent for its reliability on the absence of certain particular extraordinary, exceptional, or otherwise *rebutting* circumstances."

Newman & Marshall (1991) show that the Toulmin form suffers because the fundamental recursive nature of the argument is obscured. One may need to argue the grounds, thereby making them claims; we found this in Chapter 4 when looking at trust assumptions. One may need to argue the warrants; this is the reason for the existence of the backing, but it is not clear how the backing differs from grounds in a normal argument. Newman and Marshall propose several extensions of Toulmin arguments, such as "argument chains" (claims become grounds), "argument hierarchies" (claims become warrants), "confluence arguments" (the 'and'ing of multiple arguments), and "connections by rebuttal" (rebuttals in sub-arguments).

Although these different extensions serve different purposes in an argument, we claim that a single structure can accommodate all of them. To that end, we propose a unifying scheme that makes the recursive properties of arguments and the relationships between grounds, warrants, and claims explicit, while keeping the basic connections between the components that Toulmin proposed. In our scheme, each of the components of a Toulmin-form argument is either a proposition (unargued) or a sub-argument (argued). We also include logical connectives in order to accommodate "confluence arguments".

This scheme is realized using a simple language to represent the structure of the extended Toulmin arguments. The language captures the essence of Toulmin arguments while generalizing recursion and sub-arguments. A textual language was chosen because a) textual

```
argument                    : optional_assignments claim '.'
                            | argument optional_assignments claim '.' ;

optional_assignments        : LET assignments ';'
                            | // empty ;

assignments                 : assignment
                            | assignments ',' assignment ;

assignment                  : IDENTIFIER '=' atom ;

claim                       : optional_grounds proposition optional_rebuttals;

optional_rebuttals          : REBUTTED BY rebuttals_list
                            | // empty     ;

rebuttals_list              : rebuttal
                            | rebuttals_list ',' rebuttal ;

rebuttal                    : proposition
                            | proposition MITIGATED BY proposition
                            | proposition MITIGATED BY '(' claim ')' ;

optional_grounds            : GIVEN GROUNDS grounds_expr optional_warrant THUS CLAIM
                            | // empty ;

optional_warrant            : WARRANTED BY grounds_expr
                            | // empty     ;

grounds_expr                : grounds_factor
                            | grounds_expr AND grounds_factor  ;

grounds_factor              : grounds_term
                            | grounds_factor OR grounds_term    ;

grounds_term                : grounds
                            | NOT grounds ;

grounds                     : proposition | '(' claim ')' ;

proposition                 : IDENTIFIER ':' atom
                            | IDENTIFIER
                            | atom  ;

atom                        : STRING       ;
```

Figure 6-2 – Language Grammar

utterances[7] are easier to manipulate automatically than tree diagrams, b) argument graphs are easily generated from the parser's abstract syntax tree, and c) a 'compiler' can assist in dynamic browsing of arguments. The syntax of the language is formally defined by the LR(1) grammar (Aho, Sethi, & Ullman, 1986) shown in Figure 6-2. We will show how the language is used in the worked example.

Our extensions have the side effect of making the role of trust assumptions within the argument explicit. Recall that in our argument language, a component of an argument is either a

[7] Utterance: a stream of symbols that, when processed by a lexical analyzer, becomes a stream of lexemes that is processed by a parser to determine if the utterance is valid, as determined by the syntactic (and possibly semantic) rules of the language.

proposition or a sub-argument. In other words, some components are leaf nodes (propositions), and others are interior nodes. Leaf nodes are trust assumptions.

Applying the notion of leaf nodes to our two-part argument structure, trust assumptions are:

- Premises found in an outer argument that do not appear as a claim on an inner argument. Such premises are, in effect, unsupported claims about domain behavior, consisting of an inner argument that consists only of a claim.
- Grounds, warrants, etc., that are found in an inner argument but do not appear as a claim in some other inner argument.

This definition of trust assumptions fits well with both the discussion in Chapter 4 and the extended recursive Toulmin argumentation described in this section.

6.2 Worked Example

The example of a Personnel Information display system began in Chapter 5 Section 5.9 is continued here to illustrate the outer and inner arguments. The work in stages 1 through 3 was done in Chapter 5, providing us with primary security requirements. We construct the satisfaction arguments in this chapter. Given the system security requirements, there are design decisions to be made about where to locate the security functionality and the approach to be used, and we provide one example of this.

Reviewing the information in Chapter 5, recall that there was one business goal

```
BG1: Provision of people's personnel information to them.
```

Initial requirements were elicited and there was only one functional requirement:

```
FR1: On request from a Person (instance of People), the system shall
display personnel information (PersData) for a specified payroll number
(Payroll#) to that Person.
```

Three security goals were identified:

```
SG1: prevent unauthorized exposure of PersData
SG2: prevent unauthorized alteration of PersData,
SG3: prevent denial of access to PersData by authorized persons.
```

Applying SG1, SG2, and SG3 to FR1 resulted in three security requirements:

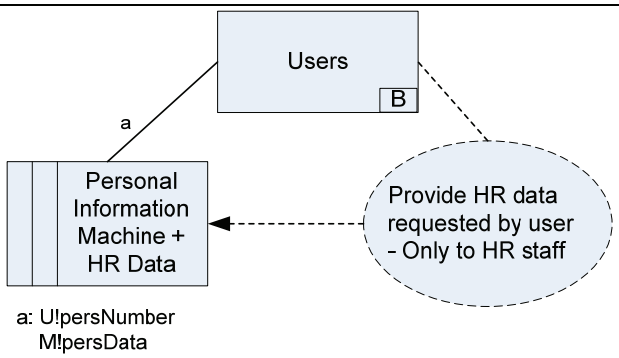

Figure 6-3 - Problem diagram for the HR data retrieval application

SR1: Personnel information must be provided only to HR staff.

SR2: displayed information must be a correct representation of stored information.

SR3: Personnel information must be provided to HR staff within 60 minutes of its request.

Although three security requirements were derived, considering one of them is sufficient to explore our ideas. Working through the others would be needlessly repetitive. We choose SR1.

Figure 6-3 shows the initial problem diagram for this application. There are two phenomena of interest. The first, U!persNumber, is the user's request for personnel information. The second, M!persData, is the information returned by the request.

6.2.1 Constructing Satisfaction Arguments

Our goal is to construct a convincing satisfaction argument that a system can satisfy its security requirements. The reader may note the use of the word "can", instead of the word "will". We use the phrase "can satisfy" because one cannot know if the eventual implementation will respect the specifications. Nor can one know if the system will introduce unintended vulnerabilities, which will manifest themselves as phenomena not described in the behavioral specification but visible in the world; buffer overflows are a prime example.

We begin by constructing an outer argument that proves the claim: HR data is provided only to HR staff.

6.2.1.1 The Outer Argument

Starting with the HR problem shown in Figure 6-3, we first attempt to construct a proof that M!persData occurs only when U!persNumber is input by a member of HR staff, or more formally that M!persData ⊢ (User ∈ HR).

There are two domains in the problem: the domain 'Users' and the 'machine' (which contains the data). To construct the argument, the behavior of the system is first described more formally. We chose a notation based on the causal logic described in (Moffett, Hall, Coombes, & McDermid, 1996) because a) phenomena in our context diagrams are normally events, handled well by a causal logic, b) 'a causes b' is well understood in requirements engineering, and c) causal logic introduces temporal properties without introducing the complexity of temporal modal logic.

Three important points must be made about our behavior specifications:

1. A statement A shall cause D can be expressed as the propositional implication A → D. The emission of phenomenon A always results in the emission of phenomenon D. We recognize that such an expression assumes that the temporal properties of shall cause are not significant, and this assumption is either a trust assumption or must be explicitly investigated in the inner argument.
2. The behavior specification is assumed to be complete, in that if the behavior specification consists of exactly A shall cause D, then no phenomenon other than A can cause D, and D cannot occur spontaneously. In other words, the mutual implication A ↔ D is true.
3. Extending #2 above, if the behavior specification consists of some set of expressions A shall cause D, B shall cause D, and C shall cause D, then no phenomenon other than A, B, and C can cause D. Expressed as a mutual implication, this is (A | B | C) ↔ D.

The behavior of the domains in Figure 6-3, expressed in our chosen notation in terms of the phenomena, is:

```
U!persNum shall cause M!persData
```

A major problem is immediately exposed. Given what is seen in the behavior description, there is no way to connect the system's behavior to the security requirement, because the type of the

domain ‘Users’ is too general. It apparently includes all humans, regardless of whether or not they are HR staff members, or even employees. The formal argument cannot be constructed. A *requirements/design* iteration is required; the system designers must be asked for help.

There are (at least) three design choices:

1. Introduce a physical restriction, e.g., a guard, to change the type of the domain from ‘Users’ to ‘HR staff’. Doing so would permit construction of the following outer argument (proof):

```
H           symbol     defined as (User, member of HR because of the guard).
I           symbol     defined as the occurrence of phenomenon U!persNum
D           symbol     defined as the occurrence of phenomenon M!persData
1. I ↔ D    premise    from the behavioral specification
2. I → H    premise    if input entered, then user ∈ HR (because of guard)
3. D        premise    assume personal information is displayed
4. D → I               split implication from #2
5. I                   detach 3, 4
6. H        conclusion detach 2, 5
```

2. Introduce phenomena into the system permitting authentication and authorization, thereby changing the type of the domain from ‘Users’ to ‘HR staff’.
3. Introduce a trust assumption (TA) asserting that the type of the domain is ‘HR staff’, even though no information is available to support the assertion.

To make the example more interesting, we choose option 2, which requires a *requirements/design* iteration. When asked, the designers chose to use an existing password-based authentication mechanism. The following secondary security goal is added:

SSG1: Users are to be authenticated as HR staff

The requirements engineer returns to the box *Construct System Context* in stage one of the activity diagram (see Figure 5-3 in Chapter 5, on page 82). The appropriate domains and phenomena are added to the context. Passing through the remainder of stage one and then stages 2 and 3 provides us with the following:

- Functional requirements to manage the authentication system must be considered in stage 1. However, as the authentication system already exists, no new functional requirements need be added.

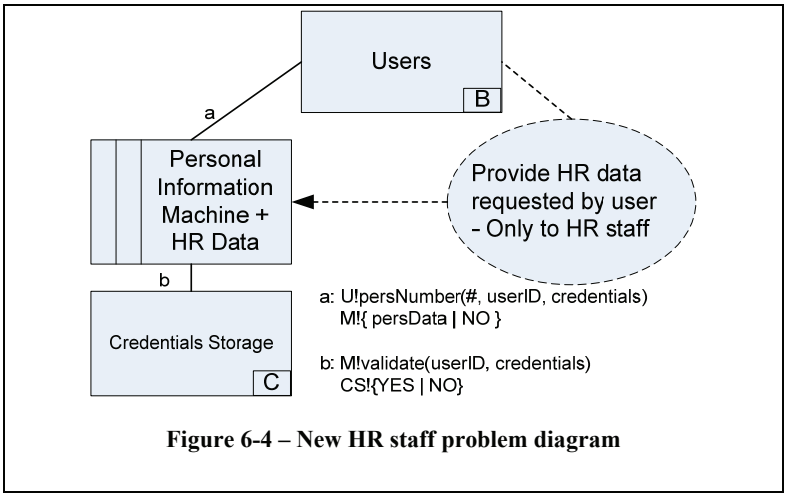

Figure 6-4 – New HR staff problem diagram

- The information in the authentication system is an asset. However, the same comment as above still applies: no new goals need be added because the system already exists.
- No secondary security requirements need be added, because SSG1 did not cause any new assets or other secondary security goals to come into existence.

Figure 6-4 shows the resulting problem diagram that will be used in this second iteration of stage 4 of the activity diagram. The diagram shows that the user is to supply some sort of credentials along with the request for information. These credentials are passed to the existing external authentication and authorization engine, which uses the internal predicate `isValid()` to determine if the credentials are for a member of human resources and then answer yes or no. If the answer is yes, then the machine provides the data, otherwise the request is refused. The corresponding behavior specification is:

```
1. U!persNumber(#, userID, credentials)
       shall cause M!validate(UserID, credentials)
2. M!validate(userID, credentials) and isValid(userID, credentials)
       shall cause CS!YES
3. M!validate(userID, credentials) and not isValid(userID, credentials)
       shall cause CS!NO
4. CS!YES shall cause M!persData
5. CS!NO shall cause M!NO
```

One can now construct the satisfaction argument for the reformulated problem. One begins with the outer argument, first defining the symbols to be used. These symbols are shown in the following table.

Symbol	Derived from (see Figure 6-4)
I : InputRequest	U!persNumber(#, userID, credentials)
V: CredsPresentedForValidation	M!validate(userID, credentials)
Y: ReplyYes	CS!YES
D: DisplayInfo	M!persData
C: CredsAreValid	isValid(userID, credentials)
H: MemberOfHR	Conclusion: user is member of HR

We derive the following predicate logic premises from the behavioral specification. These premises are the grounds used in the formal argument and, if necessary, will be supported by informal arguments.

Name	Premise	Description
P1	I → V	Behavior specification statement #1
P2	C → H	Definition of isValid: if credentials are valid then user is a member of HR.
P3	Y → (C & V)	Behavioral descriptions #2 and #3: a Yes happens only if credentials are presented for validation, then validated. Derived from the mutual implication (C & V) ↔ Y that converts to ((C & V) → Y) & (Y → (C & V))
P4	D → Y	Behavior descriptions #4 and #5: display happens only if the answer from behavior descriptions #2 and #3 was CS!YES

As the requirement is that information be displayed only to a member of HR, D is included as a premise and H as the conclusion. Thus, one wants to show:

```
(P1, P2, P3, P4, D ⊢ H).
```

A proof is shown in Figure 6-5.

1	I → V	(Premise P1)
2	C → H	(Premise P2)
3	Y → C & V	(Premise P3)
4	D → Y	(Premise P4)
5	D	(Premise)
6	Y	(Detach (→ elimination), 4, 5)
7	C & V	(Detach, 3, 6)
8	V	(Split (& elimination), 7)
9	C	(Split (& elimination), 7)
10	H	(Detach, 2, 9)
11	D → H	(Conclusion, 5 leads to 10)

Figure 6-5 – Proof that the security argument is satisfied

6.2.1.2 The Inner Arguments

Each of the rules used in the outer argument should be examined critically. We choose premises P1, P3, & P4 for initial consideration. These premises are probably not controversial, because one can say that they are part of the specification of the system to be implemented. The arguments thus consist of one trust assumption, as shown in the following utterance in our argument language:

```
let G1 = "system will be correctly implemented";
given grounds G1 thus claim P1.
given grounds G1 thus claim P3.
given grounds G1 thus claim P4.
```

Premise P2 is more complex. This premise is making the claim that instances of the domain 'Users' are limited to be instances of the subtype 'HR members', because only HR members have valid credentials. We show an argument for this claim below. This argument incorporates three trust assumptions: G2, G3, and G4.

```
given grounds
  G2: "Valid credentials are given only to HR members"
warranted by
(
 given grounds
  G3: "Credentials are given in person"
 warranted by
  G4: "Credential administrators are honest & reliable"
 thus claim
  C1: "Credential administration is correct"
)
thus claim
 P2: "HR credentials provided --> HR member"
rebutted by
 R1: "HR member is dishonest",
 R2: "social engineering attack succeeds",
 R3: "person keeps credentials when changing depts" .
```

The three rebuttals in the argument require some treatment. Recall that rebuttals express conditions under which the argument does not hold. If the rebuttals remain in the argument, they create implicit trust assumptions saying that the conditions expressed in the rebuttals will not

occur, which may be acceptable. Alternatively, one could construct an argument against a rebuttal. If the stakeholder is unwilling to accept the rebuttals, then the system must somehow be changed to mitigate them. We examine a mitigation of R1 in the next section.

6.2.2 Removing Rebuttals by Adding Secondary Security Goals

At times, the most straightforward way to remove a rebuttal might be to add functionality to a system, which is done by adding secondary security goals, then passing back through the activities to see if new functional requirements are added, as well as new assets and security requirements. This process would permit adding new grounds or warrants to mitigate the conditions that permit the rebuttal.

As an example, consider a dishonest HR member selling credentials (an instance of R1). One could mitigate this risk by increasing the probability that an unusual use of the employee's credentials would be detected, thus raising the probability that the misuse would be detected. This is new functionality.

As already noted, the framework permits addition of new functionality by adding secondary security goals and then satisfying these goals. In this example, the secondary security goal to add is

SSG2: ensure that HR members do not sell credentials.

After adding this goal, a *requirements/design* iteration is required to add sufficient design information to the context to be able to satisfy this security goal. We pass back to stage 1 in our activity diagram (see Figure 5-3 in Chapter 5, on page 82) and pass to the step *elicit/revise functional requirements*. In this example, one might add two functional requirements to the system in order to satisfy SSG2:

- FR2: all uses of HR credentials shall be logged
- FR3: any use of HR credentials from a location outside the HR department shall be immediately signaled by email to the HR director.

As the context does not contain the phenomena required to satisfy these functional requirements, the context is revisited and appropriate phenomena added.

After passing through stages two and three of the activity diagram, these functional requirements would then be used in stage 4 as grounds in an argument against the rebuttal R1:

```
given grounds
 G5: "uses of HR creds are logged (see FR2)"
  and
 G6: "uses of HR creds from outside are emailed to HR director (see FR3)"
warranted by
 G7: "these actions increase the probability of detecting improper cred use"
  and
 G8: "the employee does not want to get caught"
thus claim
 C2: "HR members will not sell their credentials".
```

C2 is added as a mitigating proposition to the rebuttal in argument 1.

```
R1: "HR member is dishonest" mitigated by C2
```

The passing through of stages 2 and 3 of the activity diagram needs further discussion. In our framework, one must ask if the new functional requirements `FR2` and `FR3` give rise to new assets and therefore new security goals (stage 2), and whether any existing or new security goals that are applied to functional requirements gives rise to new security requirements (stage 3). In the current example, at least one new asset has been created: the access log. One could argue that the HR director's email has become an asset, or has at least changed character. Analysis of these assets would produce threat descriptions (one threat description produced might be `{alter, log data, inability to verify honesty}`), which would lead to new secondary security goals in stage 2, which would lead to new secondary security requirements in stage 3, which would lead to additional satisfaction arguments. The process continues until an acceptable set of satisfactory arguments is constructed.

6.3 Chapter Summary

This chapter described our third contribution, the structured formal and informal argumentation to verify that a system can satisfy its security requirements by being *sufficiently convincing* that the system can satisfy the security requirements laid upon it. The formal

argument is used to prove a system can satisfy its security requirements, drawing upon claims about a system's behavior. The informal arguments are used to support the claims made in the first argument about system behavior and characteristics. These two part satisfaction arguments provide assurance by combining formal proof with evidence-based argumentation. They assist with determining security-relevant system properties, and inconsistent or implausible assumptions about them.

Chapter 7. Evaluation

We applied our framework in the "CRISTAL UK" project (Watson, 2006), a research initiative, managed by NATS (formerly National Air Traffic Services) for the EUROCONTROL CASCADE Programme. Although safety issues raised by potential use of the new technology are well understood and are being fully considered by the project, potential changes in security requirements are less well understood. Therefore, our goals were to gain experience with the application of our framework to validate its utility, and to discover security requirements in our chosen problem domain.

The experience was very revealing. For the project, we exposed assumptions and potential security problems that may need to be considered; determining precisely what actions to take is a future task for the project and will be based on a risk assessment. As for the framework, our systematic argumentation exposed hidden assumptions about system behavior that led to potential security problems. However, we also exposed problems with our framework: constructing and understanding the formal arguments, representation of the informal arguments, and determining the size and content of the system context used for analysis.

This chapter is structured as follows. It begins in Section 7.1, with a detailed overview of the project and the technology. Section 7.2 presents the analysis. Section 7.3 discusses lessons learned, and Section 7.4 concludes.

7.1 Project Overview

The "CRISTAL UK" project (Watson, 2006) is a research initiative, managed by NATS for the EUROCONTROL CASCADE Programme in collaboration with Raytheon Systems Limited, SITA and QinetiQ.

The project is charged with "determining the role of 'passive surveillance' in NATS future surveillance system[s]" (Watson, 2006). It is investigating the potential role of passive surveillance technologies in air traffic control areas where radar is used currently, such as in and around the airspace at busy airports.

In the context of this project, passive surveillance means using information broadcast by aircraft, without any active request or interrogation, to derive surveillance information about the aircraft, such as its position. This is opposed to active surveillance, which uses transmissions from a ground system (e.g., radar) to determine the location of an aircraft or to generate a response from it.

The members of the team have different roles in the project. NATS is responsible for the CRISTAL UK project and its deliverables. The Open University is not responsible for any deliverables in the project, but instead has a limited advisory role. Nonetheless, we hope that the requirements the project developed and, more importantly, the arguments, rebuttals, and mitigations that our analysis generated, will find their place in the project's delivered analysis.

7.1.1 Background – Air Traffic Control

Air Traffic Control is responsible for the safe and efficient movement of aircraft through a given airspace. Unfortunately, 'safe' and 'efficient' are at odds with each other. An empty airspace is a safe one – no loss of life or property due to problems with aircraft is possible – but it is also a very inefficient one. One increases efficiency by adding aircraft into the airspace, which increases risk that an accident (or an intentional act leading to loss) will occur. Air traffic controllers try to keep the risk low by maintaining safe horizontal and vertical distances (separation) between aircraft. To do so, air traffic controllers must know the identity and position of aircraft with a high degree of accuracy, integrity, and assurance.

7.1.2 Separation

The most important job of an air traffic controller is to maintain a safe separation between aircraft while ensuring that the aircraft get to where they want to go. The minimum separation

between aircraft at a given time is dependent on many factors, including the speed of aircraft, surveillance accuracy, the ability to communicate with aircraft and between controllers, the redundancy of surveillance systems, and the ability to spot and rectify mistakes.

Most of the factors are strongly influenced by how often the controller is told where an aircraft actually is, as opposed to where it is supposed to be. The more often positions are reported, the more accurate the controller's picture of the airspace is, assuming that the position reports are correct. The controller determines aircrafts' positions using active and passive surveillance.

7.1.3 Active versus Passive Surveillance

Active surveillance describes a process to determine the position of aircraft independently of where the aircraft thinks it is. There are two systems in use: primary radar and secondary radar. Primary radar operates by broadcasting directional pulses and listening for pulses reflected off aircraft. This system is independent because no help is required from the aircraft to be detected by the radar. Primary radar can only provide the position of the aircraft. Secondary radar operates by using highly directional transmissions of enquiries. Aircraft are expected to respond to the query in a fixed time. The position of the aircraft is determined from the position of the antenna and the time required to hear a response from an aircraft. The response can (and does) contain information, such as the aircraft's identity and its altitude. Where primary radar is considered independent, secondary radar can be considered to be 'cooperative' surveillance.

As secondary radar depends upon the aircraft responding to an enquiry, it will not detect aircraft that do not respond. Typically, primary and secondary radar antennae are installed together on the same rotating mount and used together to complement one another. If the primary radar detects something that is not responding to secondary radar enquiries, the air traffic controller can take appropriate action.

Passive surveillance consists of equipment that listens for transmissions from aircraft, then computes the position using that transmission; the surveillance system makes no request of the aircraft for transmission. There are two general techniques in use:

- The aircraft broadcasts its identity and position information. The surveillance system uses the information as is.
- The surveillance system uses a network of multiple receivers and multilateration (intersection of the hyperboloids described by the difference in arrival time of the transmission at each receiver) to determine the position of the transmitter.

The first technique is known as ADS-B (Automatic Dependent Surveillance – Broadcast). It uses satellite navigation technology on board the aircraft to determine where the aircraft is, and then broadcasts that position to other users without the need for any pilot input or radar interrogation. This technique depends upon the aircraft knowing its accurate position. An aircraft that either maliciously or through equipment failure reports an incorrect position will be misplaced; the only sanity check available is to check if a position report makes sense (is credible). Receiving credible but erroneous information is a key problem to be addressed.

While ADS-B can be used by ground users as a replacement for traditional surveillance techniques like radar, it is also seen as an enabling technology for new methods of air traffic control. The broadcast of surveillance data that can be received by all users, including other aircraft, may permit tasks normally undertaken by a controller to be delegated to the pilot. These ideas are encompassed in the concept of Airborne Separation Assistance Systems (ASAS) (Cervo, 2005).

The second technique has similar characteristics to secondary radar; the computation of the position depends solely upon the timing of receipt of signals.

Neither secondary radar nor one of the passive surveillance techniques can detect aircraft that are not co-operating.

7.1.4 Increasing Use of Passive Surveillance

The use of passive surveillance has become more attractive to Air Traffic Control Service Providers (ANSPs) in recent years because aircraft are increasingly being equipped with suitable avionics. In addition to the perceived operational benefits of these technologies, there

are potentially significant cost savings in procurement and through-life maintenance costs of these technologies over traditional surveillance means.

According to EUROCONTROL, increased use of passive surveillance should bring the following benefits (list quoted from (Rekkas, 2005)):

- Reduced ground infrastructure cost, resulting in a lower cost base and higher Efficiency.
- Reduced controller and pilot workload, and thus increased productivity achieved by the introduction of automated support, the reduction of voice communications workload and the automation of routine aircrew and controller tasks. This will lead to Capacity and Safety benefits.
- Increased flexibility, achieved by the provision of a new communications medium that aircrew and controllers can use in combination with existing voice communications. This is expected to lead to Efficiency and Safety benefits.
- Improved pilot and controller situational awareness and monitoring, achieved by an increase in the availability and quality of the information (e.g., from aircraft systems). This will lead to Capacity, Efficiency and Safety benefits.
- More balanced distribution of tasks among pilots and controllers achieved through an improved task distribution in ATC sectors and the delegation of tasks from the controller to the pilot. This will lead to Capacity, Efficiency and Safety benefits.
- More balanced distribution of workload between different ATC sectors achieved through the introduction of new procedures supported by automation that will enable the transfer of some tasks to adjacent sectors.

The US Federal Aviation Authority has a very similar list (Federal Aviation Administration, 2003). The open question, and the reason for the existence of many projects including CRISTAL UK, is whether these benefits can be obtained with adequate safety and security.

7.1.5 Using ADS-B to Achieve the Benefits

In order to obtain the majority of the benefits of passive surveillance, there must be aircraft-based equipment available that reports the required information about the aircraft. The ADS-B standard and complying equipment will meet this need.

ADS-B-equipped aircraft broadcast information approximately once per second. These transmissions include information about the position and status of the aircraft. The information is broadcast in various messages that include airborne position, surface position, aircraft identification and type, airborne velocity, and aircraft operational status messages (CASA, 2004). This information is collected by ADS-B receivers and then passed to air traffic control processing systems to be displayed to the controller, either on existing displays (preferred) or on some new display. The information broadcast by an ADS-B system is derived both from the avionic systems in the aircraft (e.g., air speed, barometric altitude, aircraft status) and from satellite navigation equipment (e.g., surface position, geometric altitude, and ground speed). ADS-B messages are not 'signed' in any fashion; one cannot verify that a message actually comes from the aircraft identified in the contents of the message.

7.2 The Security Requirements Analysis

The project asks whether ADS-B position reports can (or should) be considered to be a primary position source. We analyzed the security implications of this position using our framework by stepping through the activities in Figure 5-3 in Chapter 5, on page 82. The sections below are numbered using *iteration.stage*, where 'stage' comes from Figure 5-3. For example, the second stage of the first iteration will be numbered 1.2.

7.2.1 The First Iteration

During this first iteration, we established the context for the system, the functional requirements, and the primary security goals & requirements.

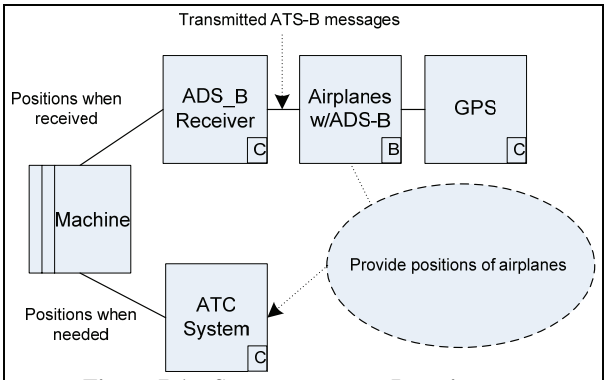

Figure 7-1 – System context – Iteration one

Step 1.1 – Identify Functional Requirements.

In this stage of the activity diagram, we identified the business goal(s) of the system under analysis, described the context, and identified the functional requirement(s). This task was dramatically simplified because working ADS-B equipment was supplied by project partners and the initial business goal was given. That business goal was:

`BG1: Provide safe and efficient air traffic management.`

Given the above goal and project's remit, the functional requirement can be summarized by:

`FR1: Provide positions of aircraft.`

The only task remaining was to determine the context, which is shown in Figure 7-1.

Step 1.2 – Identify Security Goals.

This step was charged with determining the assets involved with the system, the harms that the assets can suffer (directly or indirectly), and finally the security goals to avoid those harms.

The direct assets found from the context are GPS receivers and signals, aircraft, positions of the aircraft (broadcast), ground receivers, and the ATC system (including the controllers). The indirect assets are the contents of the aircraft (e.g., passengers), items around the ATC area (e.g., buildings, infrastructure, potentially the airport), and the aircraft owner's business (e.g., reputation, profitability, etc.).

Using this list of assets, we can (with the help of the project's domain experts) determine the harms involved in the system, and then the threat descriptions expressed as *violation of general security goal* on *asset* can cause *harm*. The threat descriptions are:

General goal: confidentiality:

```
T1: {publicizing, airplanes' position, facilitating attack in air}
T2: {publicizing, airplanes' position, loss of trade secrets}
```

The stakeholders made the decision that threats T1 & T2 are outside of the project's remit.

General goal: integrity

```
T3: {~correct, airplanes' position, lost property due to collision or
    crash}
T4: {~correct, airplanes' position, lost revenue due to increased
    separation}
T5: {~correct, airplanes' position, lost revenue due to lost confidence}
```

General goal: availability

```
T6: {~available, airplanes' position, lost property due to
    collision/crash}
T7: {~available, airplanes' position, lost revenue due to increased
    separation}
T8: {~available, airplanes' position, lost revenue due to lost
    confidence}
```

The security goals are determined by *avoiding* the action in the threat descriptions. Given these threat descriptions, the security goals are:

```
SG1: Have correct positions (avoids T3, T4, and T5)
SG2: Report positions on a timely basis (avoids T6, T7, T8)
```

Step 1.3 – Identify Security Requirements.

In this step, we determined the constraints to place on the functional requirement `FR1: Provide positions of airplanes`. We did this by composing the security goals and the functional requirement, resulting in a constrained functional requirement.

The composition produces two security requirements (constraints). The first is

```
SR1 [FR1: Provide positions of aircraft]: positions shall be accurate.
```

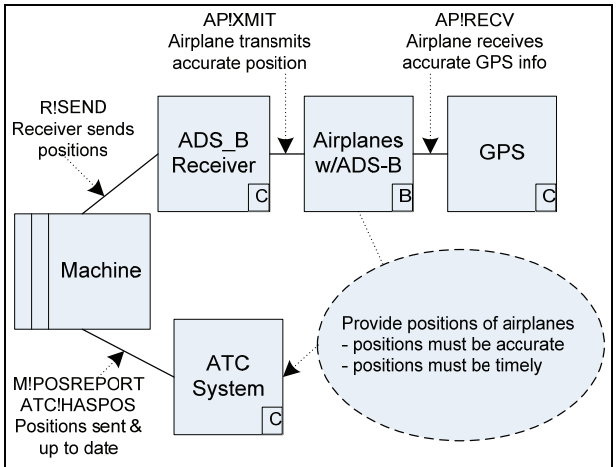

Figure 7-2 - Context with constrained requirement

The NATS requirement for accuracy is that the aircraft be within 300 meters of its reported position when the position is received. However, ADS-B can potentially improve on that by an order of magnitude, and the consequences of this must be studied. SR1 operationalizes SG1.

The second constraint is

`SR2 [FR1: Provide positions of airplanes]: positions shall be timely.`

The NATS requirement for timeliness is that a new position be received within 4 to 6 seconds of the last position report, or of the aircraft entering controlled airspace. SR2 operationalizes SG2. Figure 7-2 shows the context with the constraints.

An informal satisfaction argument that `SR1` and `SR2` satisfy `SG1` and `SG2` is as follows: the goal `SG1` is satisfied because accurate positions are available when needed (`SR1` and `SR2`); and `SG2` is satisfied directly by `SR2`.

Step 1.4 – Satisfaction Arguments

We began by constructing the formal outer argument. The steps are 1) annotate the context with the phenomena exchanged between domains, 2) develop a behavioral specification for the system in terms of the phenomena, and then 3) use the phenomena and behavioral specification in a proof that if they are complete, the system can satisfy the security requirements.

The Phenomena

Figure 7-2 shows the phenomena exchanged within the system and used in the behavior specification. The naming convention is "sending domain!message". The phenomena are:

`AP!RECV:` The airplane receives GPS broadcasts.

`AP!XMIT:` The airplane transmits its position.

`R!SEND:` The receiver sends the received position to the machine.

`M!POSREPORT:` The machine sends the position to the ATC system.

`ATC!HASPOS:` The ATC confirms that it has the aircraft's position.

The Behavior Specification

The behavioral specification is built using the variant of the causal logic described in Chapter 6 Section 6.2.1. For this project's ATC system, the behavioral specification is:

```
AP!RECV shall cause AP!XMIT
AP!XMIT shall cause R!SEND
R!SEND shall cause M!POSREPORT
M!POSREPORT shall cause ATC!HASPOS
```

We recognized that reception of GPS signals by the aircraft will not actually cause the aircraft to transmit position reports, but instead enables them. We chose to accept this slight misstatement instead of adding a clock to the context and changing to a temporal logic. As a consequence, `AP!RECV shall cause AP!XMIT` embeds the assumption that it repeats often enough to satisfy the NATS requirement. We also assumed that each processing step in the system will complete in an appropriate amount of time, again to avoid changing to a temporal logic.

The Outer (Formal) Argument

There was now enough information to construct the outer argument, a proof that the system can respect the security requirements. We want to prove that

$$\texttt{AP!RECV} \vdash \texttt{ATC!HASPOS}$$

If we can prove this, then we have proved that the system can satisfy both `SR1` (accuracy) and `SR2` (timeliness), given the following assumptions: 1) the context is correct and the implementation introduces no conflicting behavior, and 2) the temporal properties assumed

1.	AP!RECV → AP!XMIT	(premise)
2.	AP!XMIT → R!SEND	(premise)
3.	R!SEND → M!POSREPORT	(premise)
4.	M!POSREPORT → ATC!HASPOS	(premise)
5.	AP!RECV	(assumption)
6.	AP!XMIT	(Detach, 1, 5)
7.	R!SEND	(Detach, 2, 6)
8.	M!POSREPORT	(Detach, 3, 7)
9.	ATC!HASPOS	(Detach, 4, 8)

Figure 7-3 - The outer argument (proof)

above are not significant. Some of these assumptions will be challenged when we build the inner arguments.

A proof is shown in Figure 7-3.

The Inner Arguments

The premises and assumptions of the outer argument comprise a set of assumptions that must hold for the system to be secure. The purpose of the inner arguments is to challenge these assumptions in order to establish whether they hold in the real world. In our case, steps 1 through 5 in Figure 7-3 are the assumptions to be challenged.

As explained in Chapter 6 Section 6.1.2, we chose to represent arguments in our framework in a text form because this form handles complex grounds-to-claim graphs and recursion in the arguments more naturally. The argument for the initial premise AP!RECV → AP!XMIT in this form is:

given grounds
Received GPS positions are accurate (AP!RECV & assumptions)
warranted by
Calculations are accurate (assumption)
thus claim
Airplanes transmit accurate positions (AP!XMIT)
[rebutted by ...]

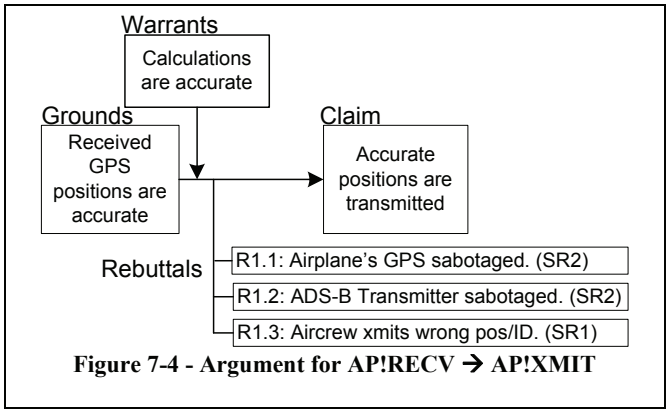

Figure 7-4 - Argument for AP!RECV → AP!XMIT

One of our first lessons learned was that although it is easy to understand the text representation of an argument when the argument is simple, understanding by project members became more difficult as the arguments become more complex. As such, we changed to a modified form of the argument diagrams Toulmin proposed. Figure 7-4 shows the argument in this form, along with the newly added rebuttals. The text in parentheses (e.g., SR2) is the security requirement that is violated if the rebuttal is true. Figure 7-5 through Figure 7-8 show the arguments for premises 2 through 4 (numbers of the lines in the proof), and for the assumption (line 5).

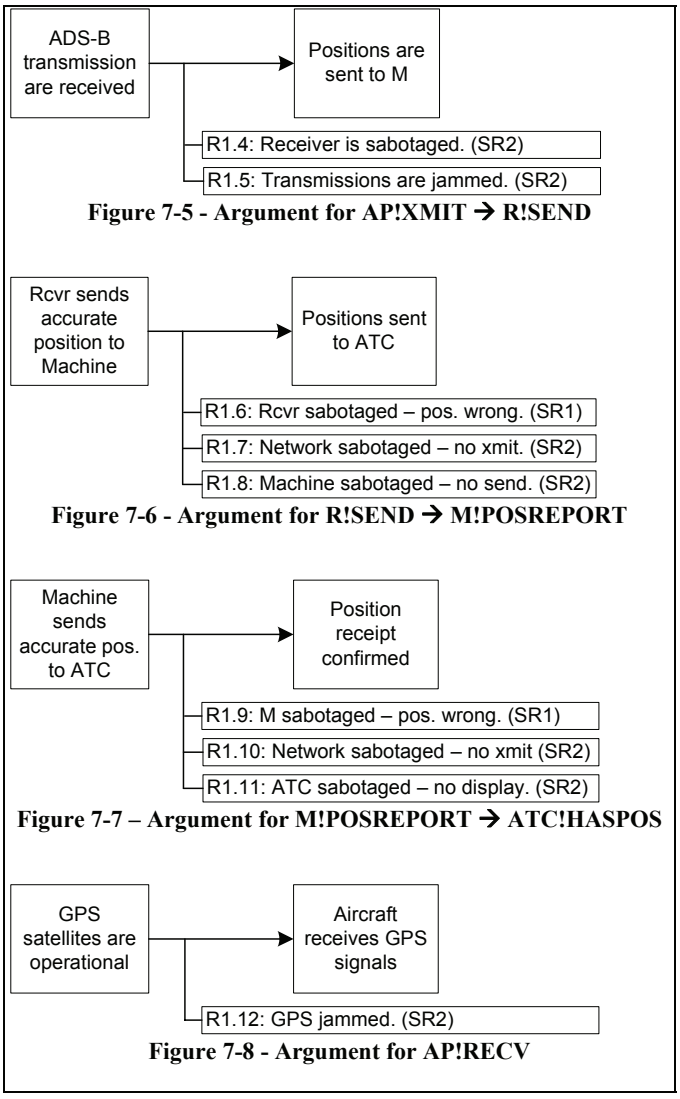

Figure 7-5 - Argument for AP!XMIT → R!SEND

Figure 7-6 - Argument for R!SEND → M!POSREPORT

Figure 7-7 – Argument for M!POSREPORT → ATC!HASPOS

Figure 7-8 - Argument for AP!RECV

There are 12 rebuttals in the arguments. These rebuttals fall into three general categories: sabotage where equipment is sabotaged to break it (R1.1, R1.2, R1.4, and R1.6 through R1.11), externally caused denial of service (R1.5 and R1.12), and the intentional transmission of incorrect data (R1.3). Each of these rebuttals should be evaluated to determine whether it should be

mitigated, and if so how. If a rebuttal is to be mitigated, then iteration is required. The project assumed that R1.3 presented an unacceptable risk of terrorism; aircraft believed to be following some track X but really going somewhere else could do a great deal of damage.

Note that rebuttals that are safety concerns are not considered here. For example, the equivalent of 'jamming' can be caused by natural phenomena such as multipath and electrical interference. We consider these to be naturally occurring behavior, and therefore to be considered during a safety analysis.

7.2.2 The Second Iteration

In order to mitigate R1.3, we needed to find a way to know that the position an aircraft transmits is the true position of the aircraft. We were less concerned with detecting that an aircraft transmitting a correct position is using the wrong identity.

Multilateration can be used to determine the position of a transmitter, computing the position by measuring the difference in a transmission's arrival time at multiple receivers. We chose this approach, and changed the context appropriately. The new context is shown in Figure 7-9.

Stepping through the framework, we see that we do not have any new functional requirements (we put aside administration of the multilateration system). We do have new assets, the multilateration computers, but they did not add any new security goals in the context of this project. As such, our security requirements did not change.

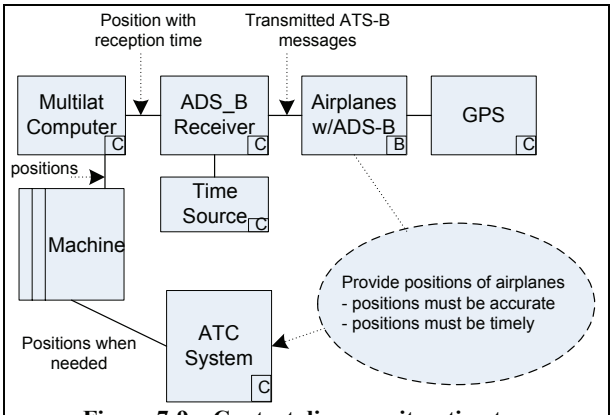

Figure 7-9 – Context diagram, iteration two

The behavior specification does have a significant change. We must describe the behavior of the new component in the context. The behavior specification is now:

```
AP!RECV shall cause AP!XMIT
AP!XMIT shall cause R!SEND
R!SEND shall cause MC!SEND
MC!XMIT shall cause M!POSREPORT
M!POSREPORT shall cause ATC!HASPOS
```

We now have a new premise in our proof, corresponding to the new component of the behavior specification.

We learned another lesson at this point. It was easier to describe the effects of the iteration using a graphical 'sub-argument' technique, rather than expressing the arguments again. This technique applies the mitigation directly to the rebuttal in the argument developed during the first iteration. We used that technique here. Figure 7-10 shows the resulting argument and mitigation. The figure also shows the next set of rebuttals, described in the next paragraph.

The first rebuttal (`R2.1.1`) challenges the assumption that the transmitter is actually in the airplane it says it is in, or is even in an airplane. One could have a small airplane accompanying a large one. The small plane broadcasts the position, which would permit the large airplane to divert. Alternatively, one could have a series of transmitters in cars, pretending to be the airplane. The second rebuttal (`R2.1.2`) challenges the assumption that there is a transmitter where multilateration says it is. It is possible to use multiple transmitters and vary the timing to

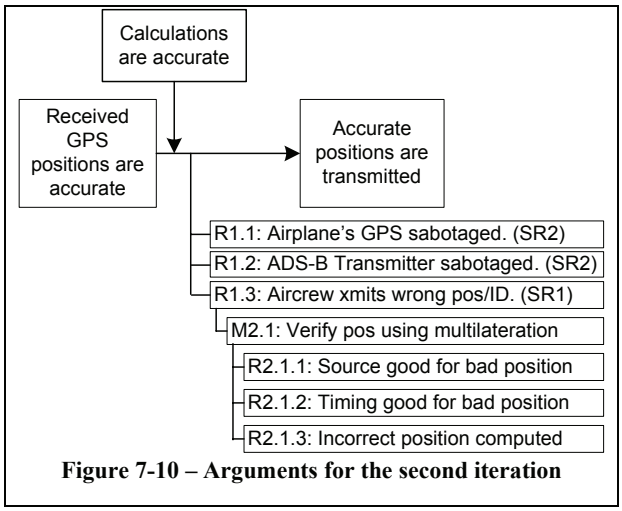

Figure 7-10 – Arguments for the second iteration

create ‘virtual transmitters’ at any position (Capkun & Hubaux, 2004). The third rebuttal (R2.1.3) challenges the assumption that the clocks in the receivers are synchronized. It is possible to perturb the clock at particular receivers, which would cause the position calculation to be offset. More detail on this rebuttal would require looking at specific multilateration time synchronization solutions.

7.2.3 The Third Iteration

A third iteration would be required to deal with rebuttals R2.1.*, assuming that the risks are determined significant, which at first glance they appear to be. For example, primary radar mitigates all of them, because it gives a reliable indication that something really is at the position reported, and that there is not something elsewhere. R2.1.2 could possibly be mitigated by using antennae that provide an approximation of the ‘angle off of horizontal’ of a transmission. R2.1.3 can be mitigated by use of a secure clock synchronization technology.

7.3 Lessons Learned

This experience taught us several things about using our framework in a real project setting.

The outer (formal) arguments were difficult to construct and explain. One problem was the nature of the proof. The outer argument proves that *if the assumptions are valid*, *if the behavior specification is correct*, and *if there are no other behaviors*, then the system can be secure. It does not prove that a system *will be* secure. Given these distinctions, some people did not see their utility and wanted to skip directly to the inner arguments. However, in our framework it is the outer arguments that provide the assumptions that the inner arguments test, so skipping this step was not appropriate. We need to find a better way to motivate, capture, and represent the outer arguments.

It is worth noting that the need to test the assumptions flowing from the outer arguments (the premises) did not present a problem. People seemed to enjoy constructing the inner arguments.

The project members were happier using a graphical representation of the inner arguments, even though the representation had less expressive power than text representation. This, plus the desire to bypass the outer arguments, led to us using the rebuttal, mitigation, rebuttal graphical argument form. Unfortunately, there are many arguments that would not be easy to express completely in this form, such as when a mitigation requires a warrant or covers several rebuttals. Tool support for converting between the text and graphical forms and for graphically rendering summary arguments would be very helpful.

Domain knowledge is certainly required, but can sometimes lead people not to question assumptions. We found that it was easy for domain experts implicitly to assume that something behaves in manner X because that is how it has always done. We found that having domain non-experts in a project helped; it seemed that someone from outside was more likely to ask "why is that?" at odd times. It should be noted that once the questions were asked, we had no problem having lively and productive discussions.

Security problems expand the system context in unexpected ways. For example, the buildings in a city are (usually) not considered part of an ATC problem until considering whether someone will decide to fly into one. Neither are the GPS satellite signals, until GPS jammers are considered. The challenge we faced was to expand the context as much as necessary, but no more so than that.

Iteration is required, especially when considering mitigations. However, iteration requires careful management to ensure that interactions are detected. The choice to represent mitigations in the context of their rebuttals led naturally to considering them one at a time, when in fact they should be considered together as part of a complete analysis. For example, it makes sense to consider all the jamming scenarios together (e.g., ADS-B jamming, clock sync jamming, GPS jamming), instead of considering them independently.

7.4 Conclusions

We had two goals for the project: to gain experience with the application of our framework to validate its utility, and to discover security requirements in our chosen problem domain. As we used the framework to produce security requirements, rebuttals, and mitigations that had not previously been considered, we consider that we succeeded with both goals.

Two particular future work items deserve mention. The project showed the need for more tool support for representing outer arguments, and we are adding this task to our near-term future work list. The effort also showed the need for work on better enabling construction and understanding of the outer (formal) arguments by people who do not normally use formality, which is a longer-term research question.

Chapter 8. Discussion & Future Work

We presented three contributions in this thesis. Recapitulating from Chapter 1, the first is a *security requirements framework* incorporating a coherent definition of what security requirements are and an explicit recognition of the importance of context: the world within which the system and the potential attackers exist. The second is *trust assumptions*, making their role in security requirements explicit. The third is two-part *satisfaction arguments* for validating whether the system can satisfy the security requirements, incorporating a formal part to establish what premises are key for security, and an informal part to challenge the premises and the trust assumptions that support them. These contributions work together to support security requirements engineering where a) asset and security goal analysis are done in the business context of the system, b) the effects of security requirements on the functional requirements are understood, c) design constraints are taken into account, and d) the satisfaction of security requirements is established through the use of arguments. The usefulness of the contributions has been validated through constructed examples, an industrial case study, and peer review.

Of course, questions and challenges have been raised during the research, and more work remains to be done. Some challenges are described in Section 8.1. Section 8.2 discusses future work, and this thesis concludes with Section 8.3.

8.1 Questions & Challenges

Several challenging questions were raised during our research.

8.1.1 Problem vs. Solution Space

A reasonable objection to the framework described in this thesis is that one is designing the system in order to determine its requirements. To some extent, this is true; the details of the system and its domains are being refined iteratively.

However, although it is true that the system and system context are being determined, the software design is not. What is being specified are the inputs and outputs (the phenomena) that the software will see and produce. By iterating between requirements and design, the environment (or context) that the software lives within is being refined to include additional domains that need to exist, and additional phenomena required to make use of these domains.

8.1.2 Traceability of Secondary Security Functional Requirements

Adding functionality to support security requirements creates a traceability problem. This issue was raised during the discussion of SeDAn in Chapter 3 Section 3.1.3. Chapter 6 provided two examples where this sort of functionality was added: addition of credential verification to permit the outer argument to be constructed, and addition of monitoring and logging functionality to support removal of the dishonest employee rebuttal. Chapter 7 provided another, the addition of multilateration. Although potentially one could trace back through the recursion in the process to connect the functions and the security requirement they support, it would be best if these functions remained strongly connected because the need for these functions could change or disappear if the security requirement changes. Currently, no mechanisms for maintaining such traceability are provided in the framework, beyond tracing mitigations to their rebuttals. Such mechanisms would be part of any eventual tool support.

8.1.3 Representing all Security Requirements as Constraints

Representing some security requirements as constraints can feel awkward. For example, consider the following security management principle

`Encryption shall be of the highest quality available`.

Adding the above as a constraint on every functional requirement could be problematic, because many of the functions do not have any obvious relationship to encryption. The constraint would appear as a restriction on who is allowed to view/access the information being encrypted, but this is one level removed from the principle.

Our position is that if a general security principle similar to the one presented above is to have any effect on the behavior of the system, it will appear either as a security requirement (a direct constraint) or as a trust assumption in an argument. For example, if the context includes a wireless LAN and there is an access constraint on the function, then the outer argument (proof) must include a premise stating that information on the LAN cannot be viewed/altered by unauthorized parties or, more probably, is viewable only by authorized parties. This premise could be supported by an inner argument referring to the quality of the encryption, thereby indirectly constraining the problem to satisfy the encryption goal.

8.1.4 Representing Required Behavior as Constraints

In many cases, constraints describe what a system *must* do, as opposed to what a system *must not* do. Although these cases are indeed constraints in the sense that they limit the choices available to the requirements engineer and architects, the terminology feels backwards to users.

8.1.5 Consistency of Trust Assumptions

One trust assumption should be consistent with (should not conflict with) another trust assumption. Given that by definition trust assumptions are not argued (if they are, they become claims), there is no mechanism in place to help assure this consistency.

A similar problem exists with respect to arguments. Nothing in the framework verifies that two arguments are consistent with each other, or that one argument depends on some trust assumption *T*, and some other argument depends on *not T*.

Resolving these issues involves solving some difficult issues. See Section 8.2.1 for more detail.

8.1.6 Trust Assumptions - Creation of Obligations

Trust assumptions create what might be thought of as obligations on the domains to which the assumptions are attached. The domains are expected to perform as trusted, or to 'competently, honestly, and dependably' conform to the trust assumption. One can say that domains are expected to *discharge* these obligations. This implies a stronger connection between domains and trust assumptions in inner arguments than currently exists in the framework.

On the other hand, the idea that domains have obligations might lead to high-level (requirements) aspects. If multiple domains must discharge the same obligation, then there is crosscutting. This idea needs further exploring.

8.1.7 Risk Analysis

The framework as described in this thesis assumes a binary level of confidence in trust assumptions, leading to a binary level of confidence in arguments that use the trust assumptions. The framework, and especially the arguments, should incorporate non binary-valued risk analysis. There are three principle points in the framework where finer-grained risk should be considered.

- Trust assumptions: trust assumptions should carry a level of confidence that the trust assumption will hold true.
- Threat analysis: one should have an idea of the impact and likelihood of the realization of a threat, in order to ascertain whether mitigating the threat is worth the cost.

- Arguments: The levels of confidence of trust assumptions used in an argument should aggregate somehow, resulting in a level of confidence for the argument. The level of confidence in an argument should take into account the implicit trust assumption that all rebuttals have been considered. This level of confidence must next be converted to the likelihood that a vulnerability exists that permits a threat to be realized.

8.1.8 Satisfaction Arguments – Constructing Outer Arguments

One issue in our framework is that the outer arguments (the formal proofs) are constructed in an ad hoc manner. This creates a barrier to general acceptance of the framework. As we noted in Chapter 7 Section 7.2, the outer arguments are difficult to construct and explain. More research is needed on proof construction aids, perhaps built directly from a behavior and phenomena specifications. We should also explore the issues and challenges of using a temporal logic for behavior specification, so that we could use some of the verification tools available for these logics.

8.1.9 Satisfaction Arguments – Constructing Inner Arguments

One question that arises is "how does the analyst find rebuttals, grounds, and warrants?" Unfortunately, we have no recipe, but a method inspired by the how/why questions used in goal-oriented requirements engineering methods such as KAOS ((van Lamsweerde, 2001) and many others) suggests itself. Given a claim, the analyst asks 'why is this claim true?' and 'what happens if it is not true?' The analyst first chooses which claim is being argued, and then uses the 'why' question to gather the grounds that are pertinent to the claim along with the warrants that connect the grounds to the claim. The argument is then constructed.

The analyst next asks the question "what can prevent this claim from being true?" The answers are the initial rebuttals. Some of these rebuttals will be challenges of the grounds or warrants; these create the need for sub-arguments where the challenged item is a claim. In other cases, the rebuttal will not be addressed, thereby creating an implicit trust assumption stating

that the event(s) described in the rebuttal are not to be considered. A third possibility is to add new grounds to the argument that remove the conditions assumed by the rebuttal.

Referring again to Chapter 7 Section 7.2, the enthusiasm showed by people while constructing the arguments arguably mitigates the lack of a recipe. People enjoyed looking for ways to break assumptions. Although there is little evidence beyond impression and anecdote, it may be that the competitive aspect of finding rebuttals is a strength of our framework.

8.1.10 Other Satisfaction Arguments in the Framework

This thesis proposes satisfaction arguments for verifying that the security requirements can be satisfied by the system. There are two other security-related satisfaction arguments that could fit in the framework. The first is that the goals are complete and consistent; if all the goals are satisfied, then no harm can come through abuse of assets. The second is that the security requirements are complete and consistent; the set of security requirements has satisfied the set of security goals. One could also imagine an argument that the asset analysis is complete. This thesis does not address these other arguments.

8.2 Future Work

This section presents ideas for future research suggested by the work described in this thesis

8.2.1 The Inner Argument

One question that begs for attention is whether, and if so how, to formalize the inner arguments. If inner arguments are formalized, one can imagine tool support to validate the arguments, or perhaps even to generate proofs for them. However, before we continue we must determine what kinds of formal arguments are appropriate in our context.

Fetzer, discussing the shortcomings of formal verification (1988), describes two different forms of argument: inductive and deductive. He characterizes them as follows (Ibid: pg 1051):

"The features that distinguish (good) deductive arguments are the following:

(a) they are demonstrative, i.e., if their premises were true, their conclusions could not be false (without contradiction);

(b) they are non-ampliative, i.e., there is no information or content in their conclusions that is not already contained in their premises; and,

(c) they are additive, i.e., the addition of further information in the form of additional premises can neither strengthen nor weaken these arguments, which are already maximally strong."

He next says that inductive arguments are non-demonstrative, ampliative, and non-additive. "Inductive arguments are meant to be knowledge-expanding, while deductive arguments are meant to be truth-preserving." In other words, deductive arguments prove something about the world, and inductive arguments draw inferences about the world. Using these distinctions, he argues that deductive arguments can be used on algorithms, but not on programs.

Fetzer's distinction between algorithms and programs (1988) is very relevant to security. Algorithms are intellectual entities, and therefore can be verified using deduction because the messiness of the world is excluded from the model. Programs run in messy environments, what Fetzer calls *causal environments*, and therefore formal verification using deduction is dubious because the model does not include all possible behavior. He argues that inductive arguments do work in the causal environments because inductive arguments permit the conclusion to be false even if the premises are all true[8]. Inductive argumentation provides a structure, but does not place constraints on the world. This conclusion is significant when thinking about security, because one strategy used by attackers is to violate some assumption about the world, causing the system to do something outside what is intended. Consider using a deontic logic (McNamara, 2006) for the inner arguments, modeling arguments based on permission and

[8] Having a false conclusion in the face of true premises can happen if an ampliative step becomes invalid through addition of another true premise. Consider the following example: 300 people queried said A, therefore most people say A. This argument can be contradicted by adding the premise 'no other people in the world will say A', which does not contradict the first premise but does make the conclusion false.

obligation. The difficulty is that permission and obligation in the real world are fuzzy. What is permission, exactly? How is permission granted, when, and to whom? Are individuals who are indistinguishable by the system (e.g., use identical credentials) the same individuals with the same permissions? There is nothing in the world that forces an individual to fulfill an obligation, so what does 'obligation' in the model mean? Similar points can be made about epistemic logic (Hendricks & Symons, 2006), modeling belief and knowledge. For example, what does 'knows' mean in the face of overhearing a dinner conversation or discovery of secrets through insufficient statistical aggregation (see Section 5.9.3)? We conclude from Fetzer's reasoning that a formalization of our inner arguments must be a formalization of the argument, and not a formalization of the world itself.

Our conclusion is further strengthened by Gödel's incompleteness theorem, which states loosely that "any consistent formal system must be incomplete" (MacKenzie, 2001: pg 90). Gödel showed that in such a system S, there will be some theorem A that is known to be true but cannot be proved true. The existence of A can permit paradox. In other words, the world is larger than the world described by the logical system. The real world is the largest of all.

Research and more experience are required to determine how to formalize an argumentation system for the inner arguments. Argumentation systems being developed by the AI community for use in law (e.g., (Bench-Capon & Prakken, 2005; Bench-Capon & Staniford, 1995; Gordon, 1993)) could be useful. The work in truth management systems (e.g., (de Kleer, 1986) and follow-ons) could also be useful. We have been asked whether our arguments would constitute due diligence in the same way that a safety argument does. This question has both technical and legal implications, both of which we are interested in exploring. Although the details of the arguments themselves are not directly relevant, the framework used in Bandara et al.'s work on security argumentation for firewalls should be further investigated (Bandara, Kakas, Lupu, & Russo, 2006).

Hunter's work on argument representation and consistency is very germane (e.g., Hunter, 2004, 2005). In particular, the techniques for tolerating inconsistency of the knowledge base, and the incorporation of belief and relevance into a formal structure, are very interesting.

We also wish to explore tools for representing our Toulmin arguments, or something close to them. Tools like Compendium (Compendium Institute, 2005) and Araucaria (Reed, 2005) hold promise, albeit for different reasons. Compendium is designed to capture arguments, while Araucaria is designed to represent and check their syntactic consistency.

8.2.2 Other Future Work

Correcting or improving upon the questions & challenges raised in Section 8.1 is one source of future work. For example, the issue of consistency of trust assumptions and arguments could be addressed by using a representation for trust assumptions in which the vocabulary and semantics are specified. The difficulty will be expressiveness and, of course, the issues raised in Section 8.2.1. In addition, tool support for tracing the use of trust assumptions and propagating confidence would be helpful.

The aspect-oriented requirements engineering (AORE) area (e.g., Rashid, Sawyer et al., 2002; Rashid, Moreira et al., 2003) offers many possibilities to investigate. One area to look at is whether security requirements (constraints) are usefully mapped into design aspects, which should be possible if there is traceability from the functional requirements into the design. Another would be to examine whether trust assumptions exhibit aspect-like crosscutting properties, and if so whether these properties could be used for cross-system risk analysis.

We want to investigate incorporating a risk analysis framework such as CORAS (2005) into our security requirements framework. Doing so would help capture rationale for why certain secondary requirements can be considered suitably equivalent to the original primary requirements. In addition, CORAS has tool support that should be useful.

One area that should prove fruitful is connecting our security requirements with one of the security-related UML variants. Doing so should help propagate some benefits of design model checking up into the functional and asset analysis stages. Equally useful, the asset analysis in the framework should help inform the development of the model. Some preliminary work has been done related to integrating the framework with UMLSec (Jürjens, 2005); the possibilities seem promising.

We want to develop aids for constructing the outer arguments, but it is not at all obvious how best to accomplish this. One idea we want to pursue includes developing a model for the behavior specifications that would permit checking the validity of the behavior, an idea related to the incorporation of UML possibility described earlier. In this case, trust assumptions would become assertions in the model. Other ideas that may or may not lead somewhere include proof templates, tools that guide construction of a proof by asking questions about behavior, and exploring the derivation of the outer (formal) arguments using a pseudo natural language.

Some other future work opportunities are:

- Tool support for managing the artifacts generated while using the framework, and in particular the traceability between them.
- Tools that can convert between the more powerful text representation and the more intuitive graphical representation.
- Incorporation of trust assumptions and argumentation into other requirements frameworks, for example *i** & KAOS.
- Further use of the framework in industrial settings.

8.3 Conclusion

This thesis has presented our three contributions, and has shown how these contributions work together to improve capture and analysis of security requirements. To reiterate, the contributions are a security requirements framework, trust assumptions, and two-part satisfaction arguments. When using the three contributions during security requirements capture and analysis, a context is defined, the effects of security requirements within that context are understood, design constraints are taken into account, and the satisfaction of security requirements is established. The usefulness of these contributions has been validated.

Our research is not unusual, in that it has provoked more questions and has suggested opportunities to extend our work. The extensions outlined in this chapter present significant challenges, which we look forward to addressing.

References

Aho, A. V., Sethi, R., & Ullman, J. D. (1986). *Compilers - Principles, Techniques, and Tools*: Addison Wesley.

Alexander, I. (2002a). "Modelling the Interplay of Conflicting Goals with Use and Misuse Cases," in *Proceedings of the 8th International Workshop on Requirements Engineering: Foundation for Software Quality (REFSQ'02)*. Essen, Germany, 9-10 Sep, pp. 145-152.

Alexander, I. (2002b). "Initial Industrial Experience of Misuse Cases in Trade-Off Analysis," in *Proceedings of the IEEE Joint International Conference on Requirements Engineering (RE'02)*. Essen, Germany, pp. 61-68.

Alexander, I. (2003). "Misuse Cases in Systems Engineering," *Computing and Control Engineering Journal*, vol. 14 no. 1 (Feb), pp. 40-45.

Allen, J. H. (2001). "CERT System and Network Security Practices," in *Proceedings of the Fifth National Colloquium for Information Systems Security Education (NCISSE'01)*. George Mason University, Fairfax, VA, USA, 22-24 May.

Anderson, R. & Needham, R. (1995). "Programming Satan's Computer," in *Lecture Notes in Computer Science*, vol. 1000, *Computer Science Today: Recent Trends and Developments*, van Leeuwen, J., Ed. Berlin / Heidelberg: Springer, pp. 426-440.

Anderson, R. (2001). *Security Engineering: A Guide to Building Dependable Distributed Systems*.

Anderson, R. J. (1996). "A Security Policy Model for Clinical Information Systems," in *Proceedings of the 1996 IEEE Symposium on Security and Privacy*. Oakland, CA, USA, 6-8 May, pp. 30-43.

Antón, A. I. & Earp, J. B. (2001). "Strategies for Developing Policies and Requirements for Secure E-Commerce Systems," in *E-Commerce Security and Privacy*, vol. 2, *Advances In Information Security*, Ghosh, A. K., Ed.: Kluwer Academic Publishers, Jan 15, pp. 29-46.

Attwood, K., Kelly, T., & McDermid, J. (2004). "The Use of Satisfaction Arguments for Traceability in Requirements Reuse for System Families: Position Paper," in *Proceedings of the International Workshop on Requirements Reuse in System Family Engineering, Eighth International Conference on Software Reuse*. Carlos III University of Madrid, Madrid, Spain, 5 Jul, pp. 18-21.

Avizienis, A., Laprie, J.-C., Randell, B., & Landwehr, C. (2004). "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1 no. 1 (Jan-Mar), pp. 11-33.

Bandara, A. K., Kakas, A., Lupu, E. C., & Russo, A. (2006). "Using Argumentation Logic for Firewall Policy Specification and Analysis," in *Proceedings of the 17th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM'06)*, vol. 4269, *Lecture Notes in Computer Science*, State, R., Meer, S. v. d., O'Sullivan, D., & Pfeifer, T., Eds. Dublin, Ireland: Springer Berlin/Heidelberg, 23-25 October, pp. 185-196.

Bench-Capon, T. & Prakken, H. (2005). "Argumentation," in *Information Technology & Lawyers: Advanced technology in the legal domain, from challenges to daily routine*, Lodder, A. R. & Oskamp, A., Eds.: Springer, pp. 69-90.

Bench-Capon, T. J. M. & Staniford, G. (1995). "PLAID: Proactive Legal Assistance," in *Proceedings of the 5th International Conference on Artificial Intelligence and Law*. College Park, MD, USA: ACM Press, pp. 81 - 88.

Boehm, B. (1984). "Verifying and Validating Software Requirements and Design Specifications," *IEEE Software*, vol. 1 no. 1 (Jan), pp. 75-88.

Boehm, B. (1988). "A Spiral Model of Software Development and Enhancement," *IEEE Computer*, vol. 21 no. 5 (May), pp. 61-72.

Boehm, B. (2000). "Requirements that Handle IKIWISI, COTS, and Rapid Change," *IEEE Computer*, vol. 33 no. 7 (Jul), pp. 99-102.

Breaux, T. D., Vail, M. W., & Antón, A. I. (2006). "Towards Regulatory Compliance: Extracting Rights and Obligations to Align Requirements with Regulations," in *Proceedings of the 14th IEEE International Requirements Engineering Conference (RE'06)*. Minneapolis, MN, USA: IEEE Computer Society, 11-15 Sep, pp. 46-55.

Breu, R. & Innerhofer–Oberperfler, F. (2005). "Model Based Business Driven IT Security Analysis," in *Proceedings of the Third Symposium on Requirements Engineering for Information Security (SREIS'05) held in conjunction with the 13th International Requirements Engineering Conference (RE'05)*. Paris France, 29 Aug.

Brewer, D. F. C. & Nash, M. J. (1989). "The Chinese Wall security policy," in *Proceedings of the 1989 IEEE Symposium on Security and Privacy*. Oakland, CA, USA: IEEE Computer Society Press, 1-3 May, pp. 206 - 214.

Brito, I. & Moreira, A. (2004). "Integrating the NFR framework in a RE model," presented at Early Aspects 2004: Aspect-Oriented Requirements Engineering and Architecture Design (AORE'04), with the Third International Conference on Aspect-Oriented Software Development (AOSD'04), Lancaster University, Lancaster, UK, 22 Mar.

Buckingham Shum, S. J. (2003). "The Roots of Computer Supported Argument Visualization," in *Visualizing Argumentation: Software Tools for Collaborative and Educational Sense-Making*, Kirschner, P. A., Buckingham Shum, S. J., & Carr, C. S., Eds. London UK: Springer-Verlag, pp. 3-24.

Burge, J. E. & Brown, D. C. (2004). "An Integrated Approach for Software Design Checking Using Design Rationale," in *Proceedings of the First International Conference on Design Computing and Cognition*, Gero, J. S., Ed. Cambridge, MA, USA: Kluwer Academic Press, 19-21 July, pp. 557-576.

Capkun, S. & Hubaux, J.-P. (2004). "Securing position and distance verification in wireless networks," Swiss Federal Institute of Technology Lausanne, Lausanne, Switzerland, Technical Report EPFL/IC/200443, May.

CASA (2004). "Australian Technical Standard Order: Airborne Stand-alone Extended Squitter, Automatic Dependent Surveillance - Broadcast (ADS-B), Transmit Only Equipment," Australian Civil Aviation Safety Authority, Standard ATSO-C1005, 21 Dec.

Castro, J., Kolp, M., & Mylopoulos, J. (2001). "A Requirements-Driven Development Methodology," in *Proceedings of The 13th Conference on Advanced Information Systems Engineering (CAiSE'01)*. Interlaken, Switzerland, 4-8 Jun, pp. 108-123.

Caughlin, D. (2000). "An Integrated Approach to Verification, Validation, and Accredition of Models and Simulations," in *Proceedings of the 2000 Winter Simulation Conference*, vol. 1. Colorado Springs, CO, USA, 10-13 Dec, pp. 872-881.

CENELEC (2002). "Functional safety of electrical/electronic/programmable electronic safety-related systems — Part 4: Definitions and abbreviations," European Committee for Electrotechnical Standardization, Brussels, European Standard IEC 61508-4, 15 March.

CERT (2006). "CERT/CC Statistics 1988-2005." Pittsburgh, PA USA: CERT CC, http://www.cert.org/stats/cert_stats.html.

Cervo, F. (2005). "Airborne Separation Assistance Systems." EUROCONTROL, Newsletter, http://www.eurocontrol.int/mil/public/standard_page/newsletter0605art2.html.

Chivers, H. & Fletcher, M. (2005). "Applying Security Design Analysis to a service-based system," *Software: Practice and Experience*, vol. 35 no. 9, pp. 873-897.

Chung, L. (1993). "Dealing with Security Requirements during the Development of Information Systems," in *Lecture Notes in Computer Science*, vol. 685, Rolland, C., Bodart, F., & Cauvet, C., Eds. Paris, France: Springer, 9-11 June, pp. 234 - 251.

Chung, L., Nixon, B., Yu, E., & Mylopoulos, J. (2000). *Non-Functional Requirements in Software Engineering*: Kluwer Academic Publishers.

Common Criteria Sponsoring Organizations (2006a). "Common Criteria for Information Technology Security Evaluation Part 1: Introduction and General Model, Version 3.1 Rev 1," National Institute of Standards and Technology CCMB-2006-09-001, Sept.

Common Criteria Sponsoring Organizations (2006b). "Common Criteria for Information Technology Security Evaluation Part 2: Security Functional Components, Version 3.1 Rev 1," National Institute of Standards and Technology CCMB-2006-09-002, Sept.

Common Criteria Sponsoring Organizations (2006c). "Common Criteria for Information Technology Security Evaluation Part 3: Security assurance components, Version 3.1 Rev 1," National Institute of Standards and Technology CCMB-2006-09-003, Sept.

Compendium Institute (2005). "Compendium." http://www.compendiuminstitute.org/.

CORAS (2005). "CORAS - A Platform for Risk Analysis of Security Critical Systems." http://www2.nr.no/coras/.

Dardenne, A., van Lamsweerde, A., & Fickas, S. (1993). "Goal-Directed Requirements Acquisition," *Science of Computer Programming (Elsevier)*, vol. 20 no. 1-2, pp. 3-50.

Dash, E. (2005). "Weakness in the Data Chain," *New York Times* (20 June).

De Landtsheer, R. & van Lamsweerde, A. (2005). "Reasoning about confidentiality at requirements engineering time," in *Proceedings of the 10th European Software Engineering Conference (ESEC-FSE'05) held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Lisbon, Portugal: ACM Press, 5-9 Sep, pp. 41 - 49.

Devanbu, P. & Stubblebine, S. (2000). "Software Engineering for Security: A Roadmap," in *The Future of Software Engineering*, Finkelstein, A., Ed.: ACM Press.

Easterbrook, S. (1996). "The Role of Independent V&V in Upstream Software Development Processes," in *Proceedings of the 2nd World Conference on Integrated Design and Process Technology (IDPT-96)*. Austin, TX, USA, 1-4 Dec.

Federal Aviation Administration (2003). "Roadmap for Performance-Based Navigation, Version 1.0," US Federal Aviation Administration, Washington, DC, USA, 22 July.

Federal Trade Commission (2006). "Matter of CardSystems Solutions Inc." Docket No.C-052 3148, 23 Feb 2006

Fetzer, J. H. (1988). "Program Verification: The Very Idea," *Communications of the ACM*, vol. 31 no. 9 (September), pp. 1048-1063.

Finkelstein, A. & Fuks, H. (1989). "Multiparty Specification," in *Proceedings of the 5th International Workshop on Software Specification and Design*. Pittsburgh, PA, USA, pp. 185-195.

Firesmith, D. G. (2003a). "Common Concepts Underlying Safety, Security, and Survivability Engineering," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, Technical Report CMU/SEI-2003-TN-033, Dec.

Firesmith, D. G. (2003b). "Using Quality Models to Engineer Quality Requirements," *Journal of Object Technology*, vol. 2 no. 5 (Sep-Oct), pp. 67-75.

Firesmith, D. G. (2004). "Specifying Reusable Security Requirements," *Journal of Object Technology*, vol. 3 no. 1 (Jan-Feb), pp. 61-75.

Fischer, G., Lemke, A. C., McCall, R., & Morch, A. (1996). "Making Argumentation Serve Design," in *Design Rationale Concepts, Techniques, and Use*, Moran, T. & Carroll, J., Eds.: Lawrence Erlbaum and Associates, pp. 267-293.

Fuxman, A., Pistore, M., Mylopoulos, J., & Traverso, P. (2001). "Model Checking Early Requirements Specifications in Tropos," in *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering*. Toronto, Ontario, Canada, pp. 174-181.

Gallagher, M. D. (2003). "Returning Health to the Telecom Sector While Opening Doors to Disruptive Technology (a speech)." U.S. Department of Commerce, National Telecommunications and Information Administration, http://www.ntia.doc.gov/ntiahome/speeches/2003/mdgccia_09242003.htm.

Gani, A., Manson, G., Giorgini, P., & Mouratidis, H. (2003). "Analysing Security Requirements of Information Systems using Tropos," in *Proceedings of the 5th International Conference on Enterprise Information Systems (ICEIS'03)*. Angers, France, 23-26 Apr.

Gans, G., Jarke, M., Kethers, S., Lakemeyer, G., Ellrich, L., Funken, C., et al. (2001). "Requirements Modeling for Organization Networks: A (Dis)Trust-Based Approach," in *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering (RE'01)*. Toronto, Canada: IEEE Computer Society Press, 27-31 Aug, pp. 154-165.

Giorgini, P., Massacci, F., Mylopoulos, J., & Zannone, N. (2004). "Requirements Engineering meets Trust Management: Model, Method, and Reasoning," in *Proceedings of the Second International Conference on Trust Management*. Oxford, UK: LNCS (Springer-Verlag), 29 Mar-1 Apr, pp. 176-190.

Giorgini, P., Massacci, F., Mylopoulos, J., & Zannone, N. (2005). "Modeling Security Requirements Through Ownership, Permission and Delegation," in *Proceedings of the 13th IEEE International Conference on Requirements Engineering (RE'05)*. Paris, France: IEEE Computer Society, 29 Aug - 2 Sep, pp. 167 - 176.

Glintz, M. (2005). "Rethinking the Notion of Non-Functional Requirements," in *Proceedings of the Third World Congress for Software Quality (3WCSQ'05)*, vol. II. Munich, Germany, 26-30 Sep, pp. 55-64.

Gordon, T. F. (1993). "The Pleadings Game: Formalizing Procedural Justice," in *Proceedings of the 4th International Conference on Artificial Intelligence and Law*. Amsterdam, The Netherlands: ACM Press, pp. 10-19.

Grandison, T. & Sloman, M. (2003). "Trust Management Tools for Internet Applications," in *Proceedings of the The First International Conference on Trust Management*, vol. 2692. Heraklion, Crete, Greece: Springer Verlag, 28-30 May.

Haley, C. B., Laney, R. C., Moffett, J. D., & Nuseibeh, B. (2003). "Using Trust Assumptions in Security Requirements Engineering," in *The Second Internal iTrust Workshop On Trust Management In Dynamic Open Systems*. Imperial College, London, UK, 15-17 Sep.

Haley, C. B. & Nuseibeh, B. (2003). "Going On-Line on a Shoestring: An Experiment in Concurrent Development of Requirements and Architecture," in *Proceedings of the SSGRR 2003w International Conference on Advances in Infrastructure for e-Business, e-Education, e-Science, e-Medicine, and Mobile Technologies on the Internet*. L'Aquila, Italy: Telecom Italia Learning Services, 2-11 Jan.

Haley, C. B., Laney, R. C., Moffett, J. D., & Nuseibeh, B. (2004a). "Picking Battles: The Impact of Trust Assumptions on the Elaboration of Security Requirements," in *Proceedings of the Second International Conference on Trust Management (iTrust'04)*, vol. 2995. St Anne's College, Oxford, UK: Lecture Notes in Computer Science (Springer-Verlag), 29 Mar - 1 Apr, pp. 347-354.

Haley, C. B., Laney, R. C., Moffett, J. D., & Nuseibeh, B. (2004b). "The Effect of Trust Assumptions on the Elaboration of Security Requirements," in *Proceedings of the 12th International Requirements Engineering Conference (RE'04)*. Kyoto, Japan: IEEE Computer Society Press, 6-10 Sep, pp. 102-111.

Haley, C. B., Laney, R. C., & Nuseibeh, B. (2004c). "Deriving Security Requirements from Crosscutting Threat Descriptions," in *Proceedings of the Third International Conference on Aspect-Oriented Software Development (AOSD'04)*. Lancaster, UK: ACM Press, 22-26 Mar, pp. 112-121.

Haley, C. B., Laney, R. C., & Nuseibeh, B. (2005). "Validating Security Requirements Using Structured Toulmin-Style Argumentation," Department of Computing, The Open University, Milton Keynes, UK, Technical Report 2005/04, 21 March.

Haley, C. B., Moffett, J. D., Laney, R., & Nuseibeh, B. (2005). "Arguing Security: Validating Security Requirements Using Structured Argumentation," in *Proceedings of the Third Symposium on Requirements Engineering for Information Security (SREIS'05), co-located with the 13th International Requirements Engineering Conference (RE'05)*. Paris, France, 29 Aug.

Haley, C. B., Laney, R. C., Moffett, J. D., & Nuseibeh, B. (2006a). "Using Trust Assumptions with Security Requirements," *Requirements Engineering Journal*, vol. 11 no. 2 (April), pp. 138-151.

Haley, C. B., Laney, R. C., Moffett, J. D., & Nuseibeh, B. (2006b). "Arguing Satisfaction of Security Requirements," in *Integrating Security and Software Engineering: Advances and Future Vision*, Mouratidis, H. & Giorgini, P., Eds.: Idea Group, pp. 16-43.

Haley, C. B., Moffett, J. D., Laney, R., & Nuseibeh, B. (2006). "A Framework for Security Requirements Engineering," in *Proceedings of the 2006 Software Engineering for Secure Systems Workshop (SESS'06), co-located with the 28th International Conference on Software Engineering (ICSE'06)*. Shanghai, China, 20-21 May, pp. 35-41.

Hall, J. G., Rapanotti, L., & Jackson, M. (2005). "Problem frame semantics for software development," *Software and Systems Modeling*, vol. 4 no. 2 (May), pp. 189-198.

Hammond, J., Rawlings, R., & Hall, A. (2001). "Will it work?," in *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering (RE'01)*, 27-31 Aug 2001, pp. 102-109.

He, Q. & Antón, A. I. (2003). "A Framework for Modeling Privacy Requirements in Role Engineering," in *Proceedings of the Ninth International Workshop on Requirements Engineering: Foundation for Software Quality*, *The 15th Conference on Advanced Information Systems Engineering (CAiSE'03)*. Klagenfurt/Velden, Austria, 16 Jun.

He, Q. (2005). "Requirements-Based Access Control Analysis and Policy Specification," PhD Dissertation, North Carolina State University,

Heitmeyer, C. L. (2001). "Applying 'Practical' Formal Methods to the Specification and Analysis of Security Properties," in *Proceedings of the International Workshop on Information Assurance in Computer Networks: Methods, Models, and Architectures for Network Computer Security (MMM ACNS 2001)*, vol. 2052. St. Petersburg, Russia: Springer-Verlag, Heidelberg, 21-23 May, pp. 84-89.

Hendricks, V. & Symons, J. (2006). "Epistemic Logic." Stanford University, http://plato.stanford.edu/entries/logic-epistemic/.

Hull, M. E. C., Jackson, K., & Dick, A. J. J. (2002). *Requirements Engineering*. UK: Springer Verlag.

Hunter, A. (2004). "Making Argumentation More Believable," in *Proceedings of the Nineteenth National Conference on Artificial Intelligence*. San Jose, CA, USA: The AAAI Press, pp. 269-274.

Hunter, A. (2005). "Presentation of Arguments and Counterarguments for Tentative Scientific Knowledge," in *Proceedings of the Second International Workshop on Argumentation in Multi-Agent Systems (ArgMAS'05)*, vol. 4049, *Lecture Notes in Computer Science*, Carbonell, J. G. & Siekmann, J., Eds. Utrecht, The Netherlands: Springer Berlin/Heidelberg, pp. 245-263.

IEEE (1998). "IEEE Standard for Software Verification and Validation," IEEE 1012-1998, 20 Jul.

In, H. & Boehm, B. W. (2001). "Using WinWin Quality Requirements Management Tools: A case study," *Annals of Software Engineering (Kluwer)*, vol. 11 no. 1 (Nov), pp. 141-174.

ISO/IEC (1999a). "Information Technology - Security Techniques - Evaluation Criteria for IT Security - Part 3: Security Assurance Requirements," ISO/IEC, Geneva, Switzerland, International Standard 15408-3, 1 Dec.

ISO/IEC (1999b). "Information Technology - Security Techniques - Evaluation Criteria for IT Security - Part 2: Security Functional Requirements," ISO/IEC, Geneva, Switzerland, International Standard 15408-2, 1 Dec.

ISO/IEC (1999c). "Information Technology - Security Techniques - Evaluation Criteria for IT Security - Part 1: Introduction and General Model," ISO/IEC, Geneva, Switzerland, International Standard 15408-1, 1 Dec.

Jackson, M. (1995). *Software Requirements and Specifications*: Addison Wesley.

Jackson, M. (2001). *Problem Frames*: Addison Wesley.

Jackson, M. (2006). "The Structure of Software Development Thought," in *Structure for Dependability: Computer-Based Systems from an Interdisciplinary Perspective*, Besnard, D., Gacek, C., & Jones, C., Eds.: Springer, pp. 228-253.

Jonsson, E. (1998). "An Integrated Framework for Security and Dependability," in *Proceedings of the 1998 Workshop on New Security Paradigms*. Charlottesville, VA, USA, 22-26 Sep, pp. 22-29.

Jürjens, J. (2005). *Secure Systems Development with UML*. Berlin & Heidelberg: Springer-Verlag.

Kelly, T. P. (1999). "Arguing Safety - A Systematic Approach to Safety Case Management," D.Phil Dissertation, University of York, York, UK.

de Kleer, J. (1986). "An assumption-based TMS," *Artificial Intelligence*, vol. 28 no. 2 (March), pp. 127-162.

Kletz, T. (1999). *Hazop and Hazan: Identifying and assessing process industry hazards*, Fourth ed. Rugby, UK: Institution of Chemical Engineers.

Kotonya, G. & Sommerville, I. (1998). *Requirements Engineering: Processes and Techniques*. United Kingdom: John Wiley and Sons.

van Lamsweerde, A. (2000). "Requirements Engineering in the Year 00: A Research Perspective," in *Proceedings of the 22nd International Conference on Software Engineering (ICSE'00)*: IEEE Computer Society Press, 4-11 Jun.

van Lamsweerde, A. & Letier, E. (2000). "Handling Obstacles in Goal-oriented Requirements Engineering," *Transactions on Software Engineering (IEEE)*, vol. 26 no. 10 (Oct), pp. 978-1005.

van Lamsweerde, A. (2001). "Goal-oriented Requirements Engineering: A Guided Tour," in *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering (RE'01)*. Toronto, Canada: IEEE Computer Society Press, 27-31 Aug, pp. 249-263.

van Lamsweerde, A., Brohez, S., De Landtsheer, R., & Janssens, D. (2003). "From System Goals to Intruder Anti-Goals: Attack Generation and Resolution for Security Requirements Engineering," in *Requirements for High Assurance Systems Workshop (RHAS'03)*, *Eleventh International Requirements Engineering Conference (RE'03)*. Monterey, CA, USA, 8 Sep.

van Lamsweerde, A. (2004). "Elaborating Security Requirements by Construction of Intentional Anti-Models," in *Proceedings of the 26th International Conference on Software Engineering (ICSE'04)*. Edinburgh, Scotland, 26-28 May, pp. 148-157.

Laprie, J.-C. (1992). "Dependability: A Unifying Concept for Reliable, Safe, Secure Computing," in *Proceedings of the IFIP 12th World Computer Congress on Algorithms, Software, Architecture - Information Processing*. Madrid, Spain: North-Holland, 7-11 Sept, pp. 585-593.

Lautieri, S., Cooper, D., & Jackson, D. (2005). "SafSec: Commonalities Between Safety and Security Assurance," in *Constituents of Modern System-safety Thinking: Proceedings of the Thirteenth Safety-critical Systems Symposium*, Redmill, F. & Anderson, T., Eds. Southampton: Springer, pp. 65-78.

Lee, J. & Lai, K.-Y. (1991). "What's in Design Rationale?," *Human-Computer Interaction*, vol. 6 no. 3-4, pp. 251-280.

Lee, Y., Lee, J., & Lee, Z. (2002). "Integrating Software Lifecycle Process Standards with Security Engineering," *Computers and Security*, vol. 21 no. 4, pp. 345-355.

Leveson, N. G. (1986). "Software Safety: Why, What, and How," *ACM Computing Surveys*, vol. 18 no. 2 (June), pp. 125-163.

Lin, L., Nuseibeh, B., Ince, D., Jackson, M., & Moffett, J. (2003). "Introducing Abuse Frames for Analyzing Security Requirements," in *Proceedings of the 11th IEEE International Requirements Engineering Conference (RE'03)*. Monterey, CA, USA, 8-12 Sep, pp. 371-372.

Liu, L., Yu, E., & Mylopoulos, J. (2003). "Security and Privacy Requirements Analysis Within a Social Setting," in *Proceedings of the 11th IEEE International Requirements Engineering Conference (RE'03)*. Monterey, CA, USA, 8-12 Sept, pp. 151-161.

MacKenzie, D. (2001). *Mechanizing Proof*. Cambridge, MA, USA & London, England: MIT Press.

McDermid, J. A., Nicholson, M., Pumfrey, D. J., & Fenelon, P. (1995). "Experience with the application of HAZOP to computer-based systems," in *Proceedings of the Tenth Annual Conference on Computer Assurance (COMPASS '95)*. Gaithersburg, MD, USA, 25-29 June, pp. 37-48.

McDermott, J. & Fox, C. (1999). "Using Abuse Case Models for Security Requirements Analysis," in *Proceedings of the 15th Computer Security Applications Conference (ACSAC'99)*. Phoenix, AZ, USA: IEEE Computer Society Press, 6-10 Dec, pp. 55-64.

McDermott, J. (2001). "Abuse-Case-Based Assurance Arguments," in *Proceedings of the 17th Computer Security Applications Conference (ACSAC'01)*. New Orleans, LA, USA: IEEE Computer Society Press, 10-14 Dec, pp. 366-374.

McNamara, P. (2006). "Deontic Logic." Stanford University, http://plato.stanford.edu/entries/logic-deontic/.

Mead, N. R., Hough, E. D., & Stehney, T. R., II (2005). "Security Quality Requirements Engineering (SQUARE) Methodology," CMU/SEI, Technical Report CMU/SEI-2005-TR-009, ESC-TR-2005-009, November.

Moffett, J. D., Hall, J. G., Coombes, A., & McDermid, J. A. (1996). "A Model for a Causal Logic for Requirements Engineering," *Requirements Engineering*, vol. 1 no. 1 (March), pp. 27-46.

Moffett, J. D. & Nuseibeh, B. (2003). "A Framework for Security Requirements Engineering," Department of Computer Science, University of York, York, UK, Technical Report YCS368, Aug.

Moffett, J. D., Haley, C. B., & Nuseibeh, B. (2004). "Core Security Requirements Artefacts," Department of Computing, The Open University, Milton Keynes, UK, Technical Report 2004/23, June.

Mouratidis, H., Giorgini, P., & Manson, G. (2003). "Integrating Security and Systems Engineering: Towards the Modelling of Secure Information Systems," in *Proceedings of the 15th Conference on Advanced Information Systems Engineering (CAiSE'03)*. Klagenfurt/Velden, Austria: Springer-Verlag, 16-20 Jun, pp. 63-78.

Mylopoulos, J., Borgida, A., Jarke, M., & Koubarakis, M. (1990). "Telos: Representing Knowledge About Information Systems," *ACM Transactions on Information Systems (TOIS)*, vol. 8 no. 4 (October), pp. 325 - 362.

Mylopoulos, J., Chung, L., & Nixon, B. (1992). "Representing and Using Nonfunctional Requirements: A Process-Oriented Approach," *Transactions on Software Engineering (IEEE)*, vol. 18 no. 6 (Jun), pp. 483-497.

Newman, S. E. & Marshall, C. C. (1991). "Pushing Toulmin Too Far: Learning From an Argument Representation Scheme," Xerox PARC, Palo Alto, CA, USA, Technical Report SSL-92-45.

NIST (1995). "An Introduction to Computer Security: The NIST Handbook," National Institute of Standards and Technology (NIST), Special Pub SP 800-12, Oct.

Nuseibeh, B. (2001a). "Weaving Together Requirements and Architectures," *IEEE Computer*, vol. 34 no. 3 (Mar), pp. 115-117.

Nuseibeh, B. (2001b). "Weaving the Software Development Process Between Requirements and Architecture," in *From Software Requirements to Architectures (STRAW'01)*, *23rd International Conference on Software Engineering (ICSE'01)*. Toronto, Ontario, Canada, 12-19 May.

Pemberton, D. & Sommerville, I. (1997). "VOCAL: A Framework for Test Identification and Deployment," *IEE Proceedings for Software Engineering*, vol. 144 no. 5 (Oct-Dec), pp. 249 -260.

Pfleeger, C. P. & Pfleeger, S. L. (2002). *Security in Computing*: Prentice Hall.

Potts, C. & Bruns, G. (1988). "Recording the Reasons for Design Decisions," in *Proceedings of the 10th International Conference on Software Engineering (ICSE'88)*. Singapore: IEEE Computer Society, pp. 418-427.

Ramesh, B. & Dhar, V. (1992). "Supporting Systems Development by Capturing Deliberations During Requirements Engineering," *Transactions on Software Engineering (IEEE)*, vol. 18 no. 6 (June), pp. 498-510.

Rashid, A., Sawyer, P., Moreira, A. M. D., & Araújo, J. (2002). "Early Aspects: A Model for Aspect-Oriented Requirements Engineering," in *Proceedings of the IEEE Joint International Conference on Requirements Engineering (RE'02)*. Essen, Germany, 9-13 Sep, pp. 199-202.

Rashid, A., Moreira, A. M. D., & Araújo, J. (2003). "Modularisation and Composition of Aspectual Requirements," in *Proceedings of the 2nd International Conference on Aspect-oriented Software Development (AOSD'03)*. Boston, MA, USA: ACM Press, 17-21 Mar, pp. 11-20.

Redwine, S. T., Jr. (Editor) (2006). "Software Assurance: A Guide to the Common Body of Knowledge to Produce, Acquire, and Sustain Secure Software," Department of Homeland Security, Version 1.05.245, 15 Aug.

Reed, C. (2005). "Araucaria." http://araucaria.computing.dundee.ac.uk/.

Rekkas, C. (2005). "Cascade Charter," EUROCONTROL Report, Version 1.0, 11 Apr.

Robertson, S. & Robertson, J. (1999). *Mastering the Requirements Process*, First ed: Addison-Wesley Professional.

Rushby, J. (2001). "Security Requirements Specifications: How and What?," in *Proceedings of the Symposium on Requirements Engineering for Information Security (SREIS)*. Indianapolis, IN, USA, 5-6 Mar.

Secure Electronic Transaction LLC (1997a). "SET Secure Electronic Transaction Specification Book 3: Formal Protocol Definition, Version 1.0," Purchase, NY, USA, 31 May.

Secure Electronic Transaction LLC (1997b). "SET Secure Electronic Transaction Specification Book 2: Programmer's Guide, Version 1.0," Purchase, NY, USA, 31 May.

Secure Electronic Transaction LLC (1997c). "SET Secure Electronic Transaction Specification Book 1: Business Description, Version 1.0," Purchase, NY, USA, 31 May.

Secure Software Inc. (2006). "CLASP: Comprehensive Lightweight Application Security Process," Secure Software Inc., McLean, VA, USA Version 2.0.

Senior Officials Group - Information Systems Security (1991). "Information Technology Security Evaluation Criteria (ITSEC)," Department of Trade and Industry, London, Version 1.2, June.

Sindre, G. & Opdahl, A. L. (2000). "Eliciting Security Requirements by Misuse Cases," in *Proceedings of the 37th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS-Pacific'00)*. Sydney, Australia, 20-23 Nov, pp. 120-131.

Soudah, J., Pilch, M., Doebling, S. W., & Nitta, C. (2004). "Verification & Validation: Credibility in Stockpile Modeling and Simulation," National Nuclear Security Administration, Fact Sheet NA/ASC-200FS-04-Rev.1.

Spafford, E. H. (1989). "The internet worm program: an analysis," *ACM SIGCOMM Computer Communication Review*, vol. 19 no. 1 (Jan), pp. 17-57.

Srivatanakul, T., Clark, J. A., & Polack, F. (2004). "Writing Effective Security Abuse Cases," Department of Computer Science, University of York, York, UK, Technical Report YCS-2004-375, 11 May.

Standish Group (1995). "The CHAOS Report," Research Report.

Standish Group (1999). "CHAOS: A Recipe for Success," Research Report.

Standish Group (2001). "EXTREME CHAOS," Research Report.

Swartout, W. & Balzar, R. (1982). "On the Inevitable Intertwining of Specification and Implementation," *Communications of the ACM*, vol. 25 no. 7 (Jul), pp. 438-440.

Tettero, O., Out, D. J., Franken, H. M., & Schot, J. (1997). "Information security embedded in the design of telematics systems," *Computers and Security*, vol. 16 no. 2, pp. 145-164.

Thompson, K. (1984). "Reflections on Trusting Trust," *Communications of the ACM*, vol. 27 no. 8 (Aug), pp. 761-763.

Toulmin, S. E. (1958). *The Uses of Argument*. Cambridge, UK: Cambridge University Press.

Toulmin, S. E., Rieke, R. D., & Janik, A. (1979). *An Introduction to Reasoning*. New York, NY, USA: Macmillan.

Viega, J., Kohno, T., & Potter, B. (2001). "Trust (and Mistrust) in Secure Applications," *Communications of the ACM*, vol. 44 no. 2 (Feb), pp. 31-36.

Viega, J. & McGraw, G. (2002). *Building Secure Software: How to Avoid Security Problems the Right Way*: Addison Wesley.

Viega, J. (2005a). "Building Security Requirements with CLASP," in *Proceedings of the 2005 workshop on Software Engineering for Secure Systems (SESS'05), held in conjuntion with the 27th International Conference on Software Engineering (ICSE'05)*. St Louis, MO, USA: ACM Press, 15-16 May, pp. 1-7.

Viega, J. (2005b). "Security in the Software Development Lifecycle: An introduction to CLASP, the Comprehensive Lightweight Application Security Process," Secure Software, Inc., McLean, Virginia, USA, White Paper.

Watson, M. (2006). "UK ADS-B in a Radar Environment." EUROCONTROL, Presentation slides, http://www.eurocontrol.int/cascade/gallery/content/public/documents/Presentations/Session%202%20-%20Trials%20and%20Implementations/Watson%20-%20UK%20ADS-B%20in%20a%20radar%20environment.pdf.

Yu, E. (1997). "Towards Modelling and Reasoning Support for Early-Phase Requirements Engineering," in *Proceedings of the Third IEEE International Symposium on Requirements Engineering (RE'97)*. Annapolis, MD, USA, 6-10 Jan, pp. 226-235.

Yu, E. & Liu, L. (2001). "Modelling Trust for System Design using the i* Strategic Actors Framework," in *Trust in Cyber-societies, Integrating the Human and Artificial Perspectives*, Falcone, R., Singh, M. P., & Tan, Y.-H., Eds.: Springer-Verlag Heidelberg, pp. 175-194.

Yu, E. & Cysneiros, L. M. (2002). "Designing for Privacy and Other Competing Requirements," in *Second Symposium on Requirements Engineering for Information Security (SREIS'02)*. Raleigh, NC, USA, 15-16 Oct.

Yu, Y., Leite, J. C. S. d. P., & Mylopoulos, J. (2004). "From Goals to Aspects: Discovering Aspects from Requirements Goal Models," in *Proceedings of the 12th International Requirements Engineering Conference (RE'04)*. Kyoto, Japan: IEEE Computer Society Press, 6-10 Sept, pp. 38-47.

Zave, P. & Jackson, M. (1997). "Four Dark Corners of Requirements Engineering," *Transactions on Software Engineering and Methodology (ACM)*, vol. 6 no. 1 (Jan), pp. 1-30.

Zhuang, L., Zhou, F., & Tygar, J. D. (2005). "Keyboard Acoustic Emanations Revisited," in *Proceedings of the 12th ACM Conference on Computer and Communications Security*. Alexandra, VA, USA, 7-11 Nov, pp. 373-382.