

Naval Research Laboratory

WASHINGTON, DC 20375-5320

NRL/MR/5542--95-7768

External COMSEC Adaptor Software Engineering Methodology

Andrew Moore
Eather Chapman
David Kim
Eric Klinker
David Mihelcic
Charles Payne, Jr.
Maria Voreh

*Center for High Assurance Computer Systems
Information Technology Division*

Kenneth Hayman

*Defence Science & Technology Organisation
Salisbury, South Australia*

August 31, 1995

Approved for public release; distribution unlimited.

ABSTRACT

The External COMSEC Adaptor (ECA) is a device responsible for providing cryptographic protection of information based on rules that (possibly coarsely) define the sensitivity of that information. The ECA is trusted to satisfy a set of critical requirements that support data confidentiality in the network in which it is embedded. Ensuring that the ECA is worthy of this trust requires defining its critical requirements precisely and constructing a strong argument that its implementation satisfies these requirements. This paper describes a software engineering methodology that uses formal methods for specifying and verifying the most critical requirements of the ECA and uses testing and simulation for verifying the overall functional requirements of the ECA. The methodology integrates the formal specifications and proofs with structured software documentation to clarify the relationship between the refinement of ECA functionality and the argument that the ECA meets its critical requirements. This methodology was used successfully to build the ECA using the KG84A to satisfy its cryptographic requirements.

CONTENTS

1. INTRODUCTION.....	1
2. BACKGROUND.....	2
3. THE ECA.....	3
4. ELEMENTS OF THE METHODOLOGY.....	4
4.1. SOFTWARE DOCUMENTATION.....	5
4.2. REQUIREMENTS VALIDATION.....	6
4.3. SOFTWARE VERIFICATION.....	7
4.4. SOFTWARE MAINTENANCE.....	8
5. STRUCTURE OF THE METHODOLOGY.....	9
5.1. METHODOLOGY OVERVIEW.....	10
5.2. ELICITING SYSTEM REQUIREMENTS.....	12
5.3. DESIGNING THE SYSTEM.....	13
5.4. DESIGNING THE SOFTWARE.....	14
5.5. IMPLEMENTING THE SOFTWARE.....	15
6. CONCLUSIONS.....	15
REFERENCES.....	16

External COMSEC Adaptor Software Engineering Methodology

1. Introduction

A system is considered trustworthy if there is an acceptably high probability that it satisfies all its critical requirements. A critical requirement of a system is any requirement that, if not satisfied, can result in the system behaving catastrophically. The External COMSEC Adaptor (ECA) is a device responsible for providing cryptographic protection of information based on rules that (possibly coarsely) determine the sensitivity of that information.¹ The ECA is intended for use in networks that have critical requirements for data confidentiality, i.e., requirements that users of the network shall not gain access to information for which they are not authorized. Although the network may have other critical requirements, e.g., integrity or availability, the ECA's primary role is to ensure the confidentiality of information. Ensuring the ECA's trustworthiness requires defining its confidentiality requirements precisely and collecting a body of evidence (called the assurance argument) that the implementation satisfies these requirements. Unfortunately, conventional methods of software engineering rarely provide adequate assurance that an implementation satisfies its critical requirements.

This paper describes a software engineering methodology that addresses the inadequacies of conventional methods for building high assurance software. The methodology is based on the use of structured software engineering/documentation techniques and formal proof methods. The structured techniques help to assure that we are building the software correctly, i.e., that the software meets its overall functional requirements; the formal proof methods provide greater assurance that the software meets its critical requirements. This methodology was used successfully to build the ECA using the KG84A to satisfy its cryptographic requirements. This device is also known as the ECA-KG84A.

The rest of this paper is organized as follows. Section 2 motivates the definition of the methodology by describing shortcomings of conventional software engineering methods and identifying techniques for alleviating these problems. Section 3 describes characteristics of networks appropriate for embedding the ECA and the primary requirements of the ECA in that context. Section 4 introduces the primary tools and techniques used in our methodology and their intended role in the software development and maintenance process. Section 5 describes how these tools and techniques are combined to form the basis for a rigorous method for developing the ECA's software.² Finally, Section 6 notes the effect that schedule and manpower constraints of the ECA development had on the definition of the methodology.

¹ Henceforth, we refer to the networked environment in which the ECA is embedded as "the network." A "user" of the network is anyone able to gain access to the communication services provided by the network, in either an intended and acceptable manner or in an unintended or malicious manner.

² Methods for analyzing the implications of the assurance provided by the ECA in the context of a particular network are beyond the scope of this paper.

2. Background

While the methodology described in this paper was initially motivated by the characteristics of the ECA, these characteristics are shared by a much broader class of systems referred to as *reactive*. Reactive systems differ from transformational systems in that they are, to a large extent, continually driven by external and internal stimuli. Typical examples of reactive systems include communication networks, computer operating systems, missile and avionics systems, voice transmission systems, and the man-machine interface of many kinds of ordinary software [1]. Evidence indicates that conventional software engineering methods are inadequate for developing reactive systems that must satisfy complex safety- or security-critical requirements. Relying solely on testing to achieve the level of assurance often mandated by critical applications requires an infeasible amount of time to execute the huge number test cases involved [2]. Since many digital systems behave in an event-driven, non-continuous manner, testing behavior at a subset of the system's input space, and interpolating its behavior at other points, is an unjustifiable approach [3]. Furthermore, the accuracy of software reliability estimates using software fault-tolerant design methods that rely on the independence of program errors are losing some credibility as the assumptions on which the statistical models are based are (empirically) invalidated [4-7].

For systems that require only moderate levels of assurance, relying solely on testing to achieve the assurance required is more effective but not without problems. Testing that software meets its security requirements - e.g., that it never enters an unsecure state - is notoriously difficult. Part of this difficulty stems from the fact that demonstrating security typically involves showing that the software never does something it should not, rather than showing that the software has some particular capability. Further, the security risk associated with a system is not directly proportional to the number of errors in the design or implementation of that system; a single flaw in a system may compromise all or most of the data that system is responsible for protecting. These problems would be diminished if rigorous methods were available for determining the security requirements of individual components necessary to support a system level security policy; such a method would allow testing components before integration, when the input space is much smaller. Methods that do exist either provide relatively low assurance or are only in the early stages of theoretical development having little or no tool support. The immature state of requirements decomposition methods often forces security testing to be delayed until system integration when flaws are more difficult to uncover and costly to repair.³

The development of high assurance reactive systems requires a more incremental and thorough approach to verification. The construction of the argument that the software satisfies its critical requirements needs to begin during system requirements definition and continue through the levels of software design to the final implementation. It should be possible, at any stage of software development, to convince oneself that the software as currently specified satisfies - or, at least, is consistent with - the

³ Simulation, i.e., the capability to test specifications, ameliorates this situation somewhat.

security requirements of the system as a whole. Such an approach combined with mathematical (formal) methods of specification and proof allows the development of a rigorous argument that the software satisfies its critical properties. Formal logic allows reasoning about the entire input space (and all reachable states), not just those covered in the test suite.

The use of formal methods of specification and proof for constructing an assurance argument has certain limitations that must be acknowledged in any rational development approach. They are based on the mathematical modeling of a system and its requirements and the proof that one level of specification conforms to another.⁴ If assumptions made in the model of a system are not valid for the ultimate implementation, proofs performed may not be applicable to the system as implemented. Furthermore, most formal methods are labor intensive due to the lack of production-quality tool support, making their use practical only for the most critical requirements of application systems.

Fortunately, the areas in which formal specification and proof techniques are weak are largely the areas in which testing is strong. Testing code provides information about its execution rather than about a (possibly flawed) model of its execution. Once appropriate test suites and a test environment are identified, testing and the analysis of test results can take place with relatively little human intervention. These points suggest that a rational high assurance development approach use formal specification and proof to complement, rather than to replace, the testing process. As recommended in [3], "the most sensible and pragmatic approach is surely to improve the effectiveness of each [i.e., testing and formal proof] as much as possible, and to maximize the extent to which each supports and compensates for weaknesses of the other." The methodology we have applied to the ECA balances the use of formal methods for specifying and verifying the most critical requirements of the ECA with the use of testing and simulation for verifying the overall functional requirements of the ECA.

3. The ECA

The primary means of enforcing data confidentiality in the network in which the ECA is embedded is through cryptography. Information is protected by its encryption using a secret key. Confidentiality is achieved by partitioning the network into a domain for processing sensitive plaintext data, the Red Domain, and a domain for processing encrypted sensitive data and plaintext non-sensitive data, the Black Domain. Users residing in the Red Domain are trusted to protect the information they process to a degree appropriate for the security classification of the data. Users residing in the Black Domain are assumed to be malicious and are not cleared for the sensitive data. The network is responsible for ensuring the separation of the Red Domain and the Black Domain and for ensuring that all communications between users over the network are permitted by a pre-defined connection plan.

⁴ An implementation is simply viewed as a specification that can be executed by a machine.

As depicted in Figure 1, the ECA provides the mechanism through which Red/Black separation for the network is achieved. The ECA must use an NSA-endorsed encryption device as the basis for protecting information it processes. Since the ECA is required to operate in a networked environment, it must be able to bypass certain network control and routing information around the encryption device. As long as the control and routing information is not sensitive, its bypass does not violate the security policy. However, the bypass may be used to pass sensitive information, either accidentally or maliciously, to the Black Domain thus violating the security policy. The ECA's responsibility is to limit the potential for bypassing sensitive information around the encryption function to the greatest degree practicable.

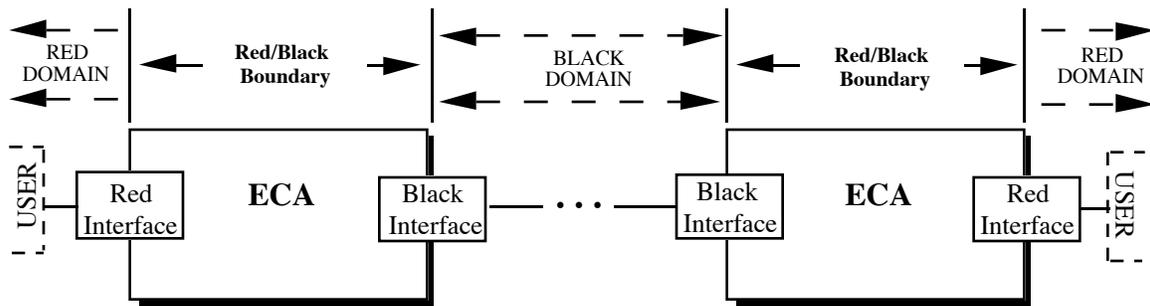


Figure 1: The ECA in a Networked Environment

4. Elements of the Methodology

A quality software engineering methodology has to answer at least the following four questions:

- How do you validate that the software requirements accurately reflect the expectations of the users? (Did we build the right product?)
- How do you verify that the software implementation meets the requirements specified? (Did we build the product correctly?)
- How do you maintain the software both while it is being built and after it is released? (How do we update the product while maintaining the validation and verification?)
- How do you document the software design, implementation and assurance argument? (What product did we build?)

The above four questions relate to the issues of software documentation, requirements validation, software verification, and software maintenance, respectively.

The methodology applied to the ECA relies on a number of existing tools and techniques that help system developers deal with these four issues in a coherent and rigorous manner. Among these are

- the Statemate Computer-Aided Software Engineering (CASE) tool [10-12] developed at i-Logix, Inc.,
- the Communicating Sequential Processes (CSP) specification language and proof theory [13-15] originally developed by C.A.R. Hoare at Oxford University,
- the mEVES verification environment [16-21] for proving properties about sequential programs developed by ORA, Canada,
- the Concurrent Versions System (CVS) [22,23] for managing multiple revisions of text developed by Prisma, Inc., and
- the Software Cost Reduction (SCR) Software Development and Documentation Methodology [8,9] originally developed at NRL.

The rest of this section presents a brief overview of these tools/techniques in the context of the four questions and related issues posed at the beginning of this section. We treat software documentation first, since documentation is a critical activity in all development phases, and then treat validation, verification and maintenance, respectively.

4.1. Software Documentation

The SCR Documentation Methodology provides the structure for documenting the requirements, design, implementation and assurance argument for the ECA software. Specifications and proofs developed using Statemate, CSP, and mEves are incorporated into this document structure to strengthen the assurance argument throughout the design and implementation process. While the SCR methodology is described fully elsewhere, we summarize its essential characteristics relevant to the ECA application.

The SCR Methodology suggests the following documentation:

- Computer System Requirements Document [24]
- System Design Document [24]
- Software Requirements Document [24]
- Software Module Guide Document [25]
- Module Interface Documents [26]
- Module Implementation Documents [27]

The purpose of the Computer System Requirements Document is to "describe the requirements of a computer system in terms of a set of environment variables of concern to the system's user and define the physical interpretation of mathematical variables representing the environmental state." [28] This document should specify constraints on the implementation by specifying the external behavior only; it should specify what the system is required to do without specifying how to do it. The document should be structured so that it can be used as a reference tool throughout the development process and so that it is easy

to change. Finally, the requirements document provides a place to record forethought about the life cycle of the system, such as likely future changes that may have to be accommodated. [24]

The purpose of the System Design Document is to "identify the computers within the computer system and define their communication with the environment." [28] This document describes the top-level decomposition of the system into physical components and the flow of control between the components. Decisions are made and recorded as to whether a component will be implemented in hardware only or in a combination of hardware and software.

The requirements for those components that are, at least partially, implemented in software are recorded in the Software Requirements Document. These can be derived solely from the Computer System Requirements and the System Design Documents. A system built according to the System Design Document and the requirements defined in the Software Requirements Document must be sufficient to satisfy the requirements defined in the Computer System Requirements Document.

The purpose of the Software Module Guide is to decompose the software task into distinct and relatively small modules based on the criteria of information hiding [29,30]. The designers must decide what characteristics of the system are most likely to change in future updates of the system. Each characteristic is then hidden in some module, i.e., the characteristic is not visible at the module's interface. The hierarchical structure of the module decomposition helps future designers and maintainers quickly find the modules of the software affected by a change. If the change was considered to be a likely modification during earlier phases of development, modifications to the existing software will be confined to a minimal number of modules.

The Module Interface Document "treats each module identified in the Software Module Guide as a 'black-box' and specifies its interface. It identifies all module access programs which share hidden data structure and describes the behavior of the module in externally observable terms." [28] Such external-only behavioral specification allows the implementors of different modules to work independently once all module interfaces are defined. This increases productivity since many modules may be implemented simultaneously.

Finally, the Module Implementation Document explains the design and implementation for each module, the actual code, and the verification that the code meets the specification defined at the module interface. Explanations should be appropriate for both the reviewers of the implementation and the future maintainers of the final product.

4.2. Requirements Validation

Unlike other aspects of software development, requirements validation is a necessarily informal process. It involves determining whether a system that meets the requirements defined in the Computer System Requirements Document is actually the system desired. This can be decided only after extensive

inspection by individuals qualified to make decisions about the validity and completeness of the requirements stated; such inspections are an important part of the SCR Methodology [31].

To complement static inspections of the requirements, it may be possible to develop a model of the system requirements that can be executed to determine whether the requirements accurately represent the desired system. The Statemate CASE tool, based on the formal theory of statecharts [1], allows modeling the behavior of systems and the graphical execution of these models to determine their validity. Statemate can be used to view the model from three different viewpoints: the behavioral view, the functional view, and the physical view. Statemate supports a graphical language for each of these views: statecharts represent the behavioral view, activity charts represent the functional view, and module charts represent the physical view. The flexibility with which Statemate can view the system under development is useful throughout system requirements specification and design.

The functional and behavioral views of Statemate combined with the ability to execute these views helps people determine the validity of a set of requirements. By executing a system model of requirements, they can more easily determine whether the requirements are appropriate. If not, their experience should help them "debug" the statement of requirements. This process should not be viewed as a replacement for thorough inspections but as an aid in the inspection process. Qualified personnel are needed for the inspection no matter how it is performed.

4.3. Software Verification

Software verification is the process of determining whether the product of a given phase of software development meets the requirements established during the previous phase, e.g., showing that a software design specification meets the software requirements. There are potentially many levels of software verification that must be performed during the development life-cycle. Verification can proceed dynamically, through simulation of software designs or testing of their implementations, or statically, through design/implementation inspections or through formal proofs that implementations satisfy their specifications.

The conventional means of software verification is through simulation and testing. As discussed previously, Statemate can be used to simulate system and software designs. The graphical execution of specifications is valuable in determining whether a given design meets the requirements specified for that design. The hierarchical refinement capability of Statemate helps assure that successive refinements to a design conform to the higher level specifications.

Testing can proceed on individual programs or during integration using black and/or white-box techniques. The SCR methodology eases the unit testing process by forcing the division of the software into small, independent modules that have precisely defined interfaces. The definition of correctness for the access programs of the module is explicitly and completely defined in the module's interface specification. Integration testing is made easier due to the precise computer/software requirements specification. This not

only provides a strict definition of correct behavior, but also a means to quickly generate complete and systematic tests.

Code inspections are the conventional means of performing static analysis of software designs and implementations. The SCR Methodology promotes rigorous inspections throughout the software development life-cycle [31]. An even more rigorous analysis of designs and code can be performed using formal mathematical proof of an implementation's conformance to a specification. Such an approach requires the mathematical definition of the semantics for a language and a proof theory for specifying and proving properties about the constructs of the language. The CSP language has these characteristics.

CSP allows the description of systems composed of networks of communicating processes. A CSP process communicates with its environment through named communication channels. Olderog and Hoare [13] describe a family of increasingly sophisticated models for CSP; less sophisticated members of the family enable specification and proof of a subset of the properties that the more sophisticated members enable. We chose a particular model of CSP, called the Trace Model, due to its comparative simplicity and its ability to prove safety properties⁵ of networks of processes. The Trace Model maps a process to an alphabet and a set of traces. The alphabet of a process specifies all communication events, i.e., channel-value pairs, in which it is permitted to engage. A trace of a process is an observation of its execution. It consists of a finite sequence of all communication events in which the process has engaged at some moment in time. Properties specified about systems described in CSP take the form of restrictions on the traces that a process representing the system may engage. If the set of traces associated with the process actually conform to these restrictions, the system is said to satisfy the properties.

The CSP description, specification and proof techniques described above are primarily targeted for higher level system design and verification. The mEves Verification System provides facilities uniquely constructed to prove properties about actual computer programs that are written in the mVerdi language [17,18]. mEves does much of the work required to generate the conditions under which the program satisfies the specified properties. mEves's automated support for proving that these conditions hold eases the difficult formal verification process. ORA's subsequent version of mEves, called Eves, and its associated language Verdi [21] were not available in time for the ECA development.

4.4. Software Maintenance

The SCR Methodology promotes the family style of system development to reduce the cost of system development and maintenance, training, and documentation [32]. The family approach forces designers to consider the entire family of systems before building any member. Software is structured to

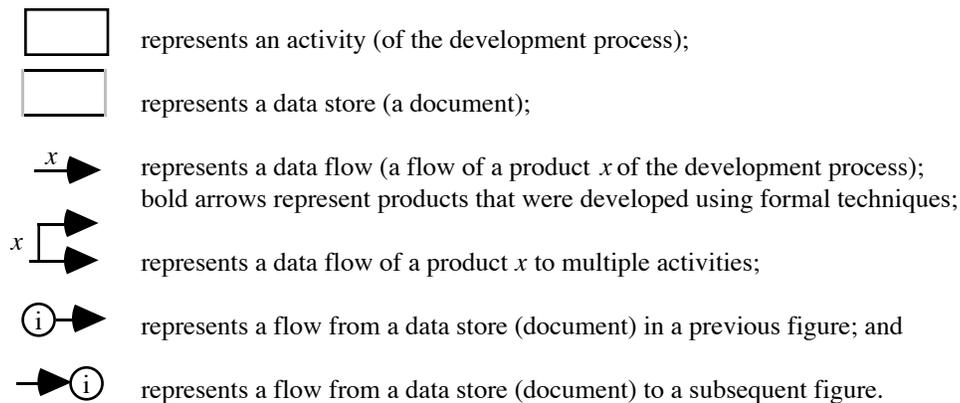
⁵ Safety for concurrent processes corresponds to partial correctness for sequential programs. Intuitively, safety properties specify that some undesirable state is never entered whereas liveness properties specify that some state is eventually entered. Note that this use of the term safety is different from our use of the term safety-critical; safety-critical refers to human safety.

hide in separate modules those aspects of the members that may differ. Different family members can be constructed by modifying the appropriate modules, rather than changing the overall program structure. Provided that the family was defined appropriately, future changes to the system by its maintainers will likely result in implementing a different family member. This organization permits differences among the members of the family while taking advantage of their common aspects.

Although the SCR Methodology promotes ease of change of the software, the methodology has no specific provisions for performing the bookkeeping tasks necessary for consistently managing successive modifications/refinements of the software design and implementation. For this purpose we use the Concurrent Versions System (CVS) [22,23]. CVS supports storage and retrieval of multiple versions of text, maintenance of a complete history of changes, resolution of access conflicts among multiple development team members, and system release and configuration control. By including the text "\$Header\$" at the beginning of a file, CVS automatically prepends information specifying the file name, revision number, creation time, author and configuration status.

5. Structure of the Methodology

The previous section introduced the individual tools and techniques that we are using and described the role they can play in developing high quality software. This section describes how we use these tools and techniques to develop ECA software that conforms to its derived security requirements with high assurance. We begin by presenting an overview of the methodology, shown in Figure 2, and a description of the documentation produced by the development process. A more detailed description of the methodology is presented in subsequent sections as a sequence of actions that must be performed; each document that is produced is the result of some subsequence of these actions. Figures 3 through 6 outline the actions performed and documents produced. These figures use constructors similar to those used in Statemate activity charts:



We italicize words/phrases in the textual description of each figure that correspond with products (including documents) of the development process, i.e., names associated with data stores or data flows in the figure.

5.1. Methodology Overview

The common use of physical separation of processing domains to promote security of cryptographic devices (and, increasingly, for multi-level secure systems in general, e.g., [33]) suggests the need to deal explicitly with concurrent and distributed systems within the methodology. Conventional techniques for building secure systems are often based on simple finite state machine models of the system's security policy, such as the Bell and LaPadula Model [34]. Such models typically rely on the determination of a global system state - if a property can be proved of the initial state of the system and in all states reachable from the initial state, the system is said to satisfy that property. For concurrent and distributed systems, the maintenance of a global state that is up-to-date with respect to all components may be difficult if not impossible. If the global state does not accurately reflect the states of the system components - for example, due to race conditions - an intuitively insecure operation may be permitted. The methodology applied to the ECA explicitly supports the specification and analysis of concurrent and distributed systems.

Figure 2 illustrates our process for constructing the assurance argument for the ECA's software. The SCR methodology provides a framework for expressing the assurance argument. The primary components of SCR are listed along the left side of the figure. Along the right side are the specification languages and tools that contributed to the implementation and verification of the ECA. The result of the integration of SCR and the specification languages/tools is the ECA's assurance argument illustrated in the center of the figure.

A variety of formal and informal techniques are used to allow reasoning across four semantic domains. The network security policy is expressed in English, the critical requirements model was specified and refined in CSP, the ECA's components were specified and verified in mVerdi and then implemented in Ada using the Verdex Ada Development System (VADS). Slanted arrows indicate a refinement of a specification to a more detailed specification or implementation; vertical arrows indicate a translation of a specification from one semantic domain to another at a comparable abstraction level. Dashed arrows indicate an informal refinement/translation; solid arrows indicate a formal one. The increase in width of the argument from top to bottom reflects additional detail specified at the lower levels.

The following sections partition the development process into four parts and describe in more detail the application of formal techniques during each part. The ordering of the parts does not necessarily imply the chronological ordering of the tasks between parts during the development process. To simplify the figures, only the major activities and major flows between activities in the development process are illustrated.⁶ In general, techniques such as Statemate and SCR are used to document the overall function of

⁶ We make no attempt to depict the iteration of activities needed when problems are uncovered during later stages of refinement.

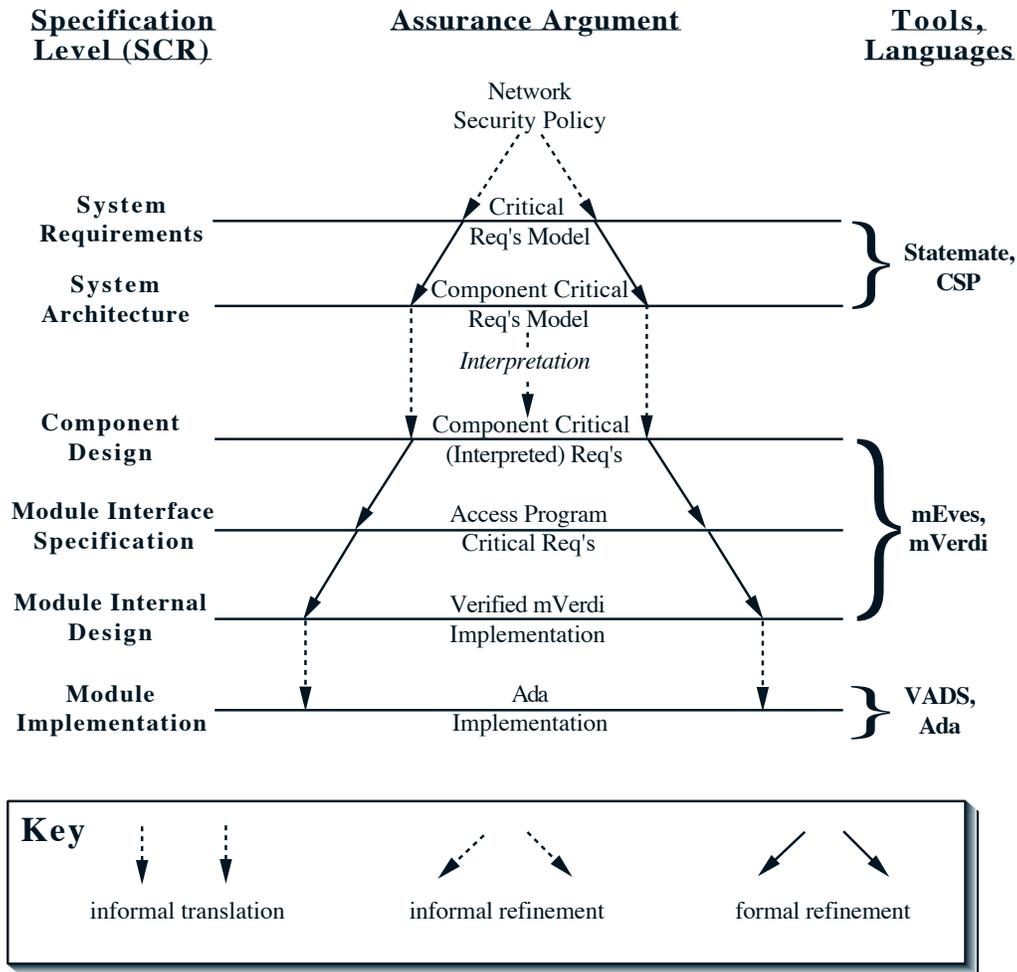


Figure 2: Overview of the Methodology

the system, whereas techniques such as CSP and mEves are used to construct the formal assurance argument. The intent of the formal assurance argument is to show how the implementation satisfies critical requirements identified during the system requirements definition activity.

The primary documents produced as the result of applying this methodology follow:

- System Requirements Document⁷
- System Design Document
- Module Guide Document
- Critical Requirements Model Document
- Critical Requirements Decomposition Document
- Critical Requirements Interpretation Document
- Module Interface Document
- Module Implementation Document

⁷ The ECA software requirements differ only slightly from the system requirements, so we did not construct a distinct software requirements document.

Documents along the far left, which correspond to the first three levels of the SCR specification in Figure 2, describe the refinement of the overall function of the system along the lines required by the SCR methodology. Documents along the far right, which correspond to the first three levels of the assurance argument refinement in Figure 2, describe the refinement of the formal assurance argument. Documents in the middle, which correspond to the last three levels in Figure 2, describe the refinement of the software integrated with the refinement of the argument that the software conforms to its critical requirements. The required content of each document is outlined in the following subsections.

5.2. Eliciting System Requirements

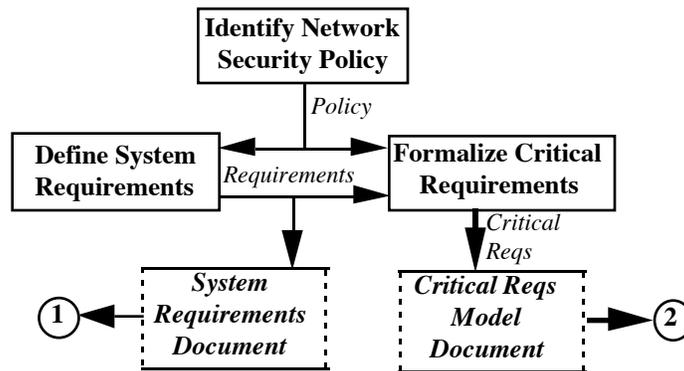


Figure 3: Requirements Documentation

As discussed in the introduction of this paper, the security *policy* of the network motivates the directions taken in the development of the ECA. The system *requirements* for the ECA are identified through extensive discussion with the administrators/developers of the networked environment and the eventual users of the system. By considering the security policy in the context of the ECA's requirements, a model of the *critical requirements* of the ECA is defined and documented in the ECA *Critical Requirements Model Document*. This model is formalized in the Trace Model of the CSP language as a series of restrictions on the traces in which the ECA can engage. A goal of the modeling process is to describe the security requirements in an abstract setting, providing a framework from which to describe the more specific security requirements of each of a set of systems. The ECA *System Requirements Document* describes the requirements in a format similar to that described by the SCR Methodology [24].

5.3. Designing the System

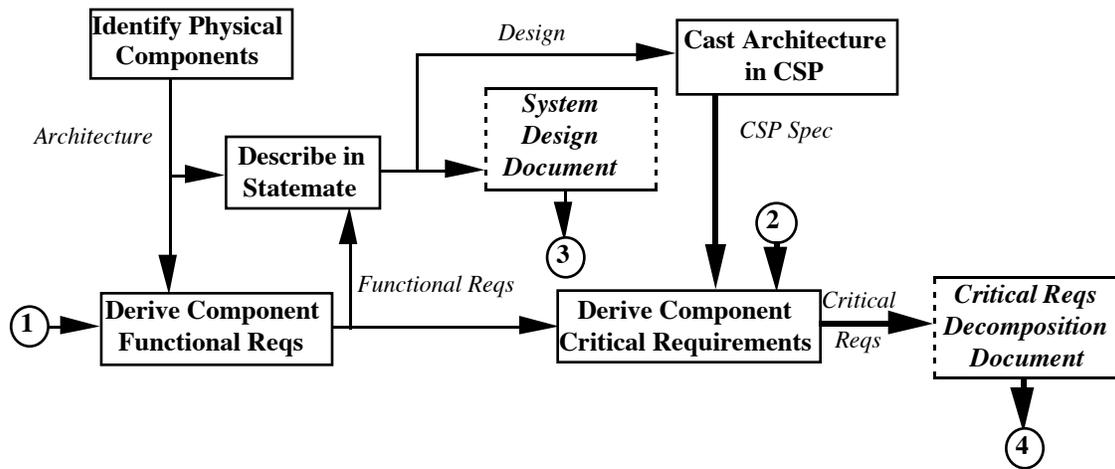


Figure 4: System Decomposition Documentation

The system design process involves the physical decomposition of the system into sequential components of the *architecture*. The communication between the components, i.e., their interfaces, is specified precisely and completely. Statemate is used to *design* the ECA's top-level component architecture and communication flow control among the components identified. The Statemate specification, recorded in the *System Design Document*, is executed to determine its conformance with derived component *functional requirements* and to assess the viability of the implementation. In order to argue formally that the design meets the critical properties specified in the Critical Requirements Model Document, the architecture described in Statemate is translated to a *CSP specification*.

Formal assurance that the system design meets its critical requirements is gained by decomposing the critical system requirements into *critical requirements* on the individual components using the Trace Model of CSP. We have defined an iterative approach [35] to performing such decompositions. An extension to the CSP notation, involving process composition with hidden internal structure, promotes hierarchical system design and decomposition. The approach described reduces the problem of proving that the system meets its critical requirements to the simpler, albeit non-trivial, problem of proving that each component meets its derived requirements. The ECA *Critical Requirements Decomposition Document* describes the CSP architecture, the derived critical properties for each component identified, and the proof that these properties, when taken together, are sufficient to satisfy the requirements of the Critical Requirements Model.

5.4. Designing the Software

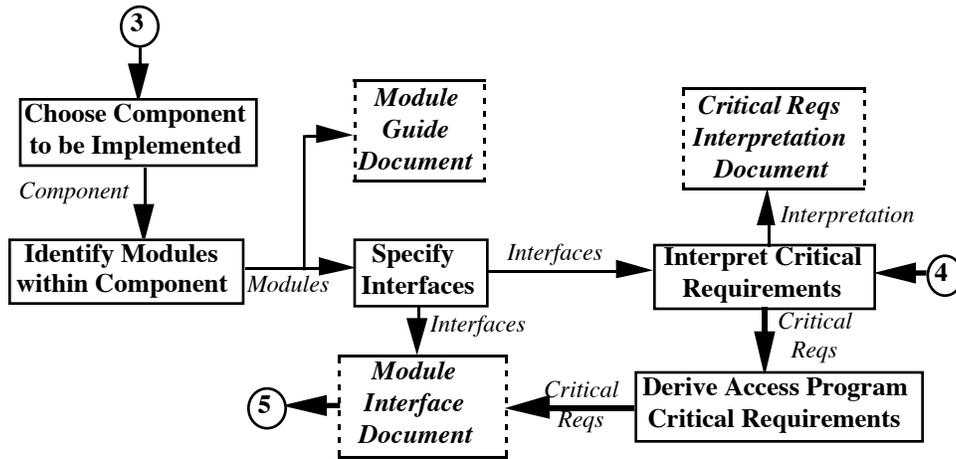


Figure 5: Software Design Documentation

The first step of the software design for a given *component* is the *module* decomposition. Likely modifications to the ECA, i.e., differences among ECA family members, are considered so that the decomposition hides the appropriate characteristics of the ECA. This step is crucial for the future maintainability of the ECA. The result is documented in the ECA *Module Guide Document*.

Although the method of deciding how to partition the software is primarily informal, the definition of the *interfaces* for the modules identified proceeds in both a formal and an informal manner. The module interfaces are specified in the conventional SCR-style [26] with an additional section defining the critical requirements of the individual access programs of each interface in mVerdi, the language of the mEves system. Before the access program critical requirements can be determined, however, the *interpretation* of the abstractions on which the critical requirements model and decomposition is based needs to be specified. The interpretation involves (1) mapping the primitives of the model/decomposition onto the elements of the implementation that implement those primitives and (2) validating that this mapping satisfies the internal assumptions of the modeling/decomposition process. The results of the interpretation are described in the ECA *Critical Requirements Interpretation Document*.

The primary goal of the interpretation is to derive the concrete *critical requirements* for the ECA software in terms of the primitives of the ECA module interfaces. The interpretation requires mapping a trace specification in the CSP world to pre- and post-conditions on individual programs. This is accomplished by defining a Get and a Put routine that reflect the semantics of the CSP input and output operators, respectively. An mVerdi program that communicates using these routines builds up a trace of its execution, which is recorded in a specification-only variable. This permits us to derive *critical requirements* of individual access programs defined on each module's interface. The results of the above

process (including the SCR module interface specifications and the access program critical requirements) are documented in the *ECA Module Interface Document*.

5.5. Implementing the Software

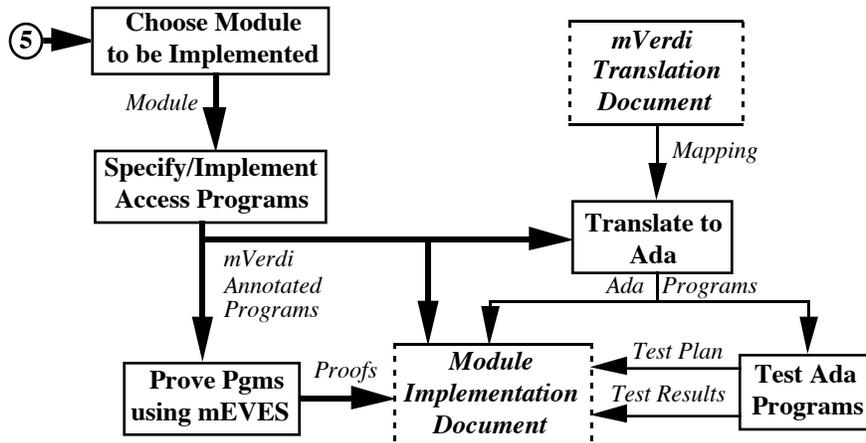


Figure 6: Software Implementation Documentation

Once the software design is established, each *module* is implemented to satisfy its interface specification as an independent entity. Rather than coding directly in Ada, each access program is first specified and implemented as an *mVerdi annotated program* and then formally proven to meet its critical requirements using mEves. A standard for *mapping* mVerdi programs to Ada helps ensure the consistency and correctness of the translation for the ECA software throughout the implementation process. Once the *Ada programs* have been generated, a *test plan* is developed that includes the types of tests to be performed, e.g., white- or black-box, and the test suite to be run. For each access program, the *ECA Module Implementation Document* comprises the model of implementation, the *mVerdi programs* and *proofs* with logical commentary, the *Ada programs* (including comments justifying differences between the Ada and the mVerdi code), and the *test results*.

6. Conclusions

Developing a comprehensive and completely formal assurance argument was not possible for the ECA, given the schedule and manpower constraints of the development effort. These constraints required us to accept informal translations between the major specification and programming languages used, e.g., translations from CSP trace specifications to mVerdi pre-/post-condition specifications and from mVerdi code to Ada code. In addition, the ECA development constraints did not permit investigating approaches to address rigorously differences between the data abstractions used in the ECA critical requirements model and those used in the ECA implementation. Since CSP supports only procedural refinement, the methodology, as defined, precludes rigorously verifying properties of the application-specific software, firmware and

hardware that reside below the Trace Model abstraction, e.g., implementations of calls to send and receive messages over particular channels.

We report on lessons learned from applying this methodology to the development of the ECA in [36]. Future work involves analyzing the ECA development and certification effort with the goal of improving the methodology for future application.

References

- [1] Harel, D., "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming* 8, Elsevier North-Holland, pp. 231-274, 1987.
- [2] Butler, R.W., G.B. Finelli, "The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software," *IEEE Transactions on Software Engineering*, Vol. 19, No. 1, Jan. 1993.
- [3] Rushby, J., "Formal Specification and Verification for Critical Systems: Tools, Achievements, and Prospects," *Electric Power Research Institute Technical Report TR-100294*, January 1992.
- [4] Scott, R.K., J.W. Gault, D.F. McAllister, "Fault-Tolerant Software Reliability Modeling," *IEEE Transactions on Software Engineering*, pp. 1491-1501, Dec. 1985.
- [5] Eckhardt, D.E., L.D. Lee, "A Theoretical Basis for the Analysis of Multiversion Software Subject to Coincident Errors," *IEEE Transactions on Software Engineering*, pp. 1511-1517, Dec. 1985.
- [6] Knight, J.C., N.G. Leveson, "An Experimental Evaluation of the Assumptions of Independence in Multiversion Programming," *IEEE Transactions on Software Engineering*, Vol SE-12, pp. 96-109, Jan. 1986.
- [7] Shimeall, T.J., N.G. Leveson, "An Empirical Comparison of Software Fault-Tolerance and Fault elimination," *IEEE Transactions on Software Engineering*, pp. 173-183, Feb. 1991.
- [8] Labaw, B., et.al., "The SCR Methodology: Collected Papers," *NRL Internal Report*.
- [9] Clements, P.C., "Software Cost Reduction through Disciplined Design," *Naval Research Laboratory Review*, July 1985.
- [10] Harel, D., A. Pnueli, "On the Development of Reactive Systems," *NATO ASI Series*, Vol. 13 (K.R. Apt, ed.), Springer-Verlag, New York, 1985, pp. 477-498.
- [11] Harel, D., A. Pnueli, J. Schmidt, R. Sherman, "On the Formal Semantics of Statecharts," *Proc. 2nd IEEE Symp. on Logic in Computer Science*, Ithaca, NY, 1987, pp. 54-64.
- [12] Harel, D., H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, M. Trakhtenbrot, "STATEMATE: A Working Environment for the Development of Complex Reactive Systems," *IEEE Transactions on Software Engineering*, Vol. 16, No. 4, April 1990.
- [13] Olderog, E.R., C.A.R. Hoare, "Specification Oriented Semantics for Communicating Sequential Processes," *ACTA Informatica*, Vol 23, pp. 9-66, 1986.
- [14] Hoare, C.A.R., "Communicating Sequential Processes," Englewood Cliffs, NJ: Prentice-Hall International, 1985.
- [15] Chen, Z.C., C.A.R. Hoare, "Partial Correctness of Communicating Sequential Processes," *Proceedings International Conference on Distributed Computing*, Paris, France, Apr. 1981.

- [16] Craigen, D., et.al., "mEves: A Tool for Verifying Software," Proceedings 10th International Conference on Software Engineering, Singapore, Apr. 1988, also published as ORA, Inc. Technical Report CP-87-5402-26, Oct. 1988.
- [17] Craigen, D., "A Description of mVerdi," ORA, Inc. Technical Report CP-87-5420-02, Nov. 1987.
- [18] Craigen, D., M. Saaltink "An mVerdi User's Guide," ORA, Inc. Technical Report TR-87-5420-12, Nov. 1987.
- [19] Pase, B., M. Saaltink, S. Kromodimoeljo, "mEves User's Manual," ORA, Inc. Technical Report TR-87-5420-14, Nov. 1987.
- [20] Craigen, D., "An Application of the m-EVES Verification System," ORA, Inc. Technical Report CP-88-5402-29, Feb. 1988.
- [21] Craigen, D., "Reference Manual for the Language Verdi," ORA, Inc. Technical Report TR-90-5429-09, Feb. 1990.
- [22] Tichy, Walter F., "Design, Implementation, and Evaluation of a Revision Control System," in Proceedings of the 6th International Conference on Software Engineering, IEEE, Tokyo, Sept. 1982.
- [23] Berliner, Brian, "CVS II: Parallelizing Software Development," Prisma, Inc. Technical Report, 5465 Mark Dabling Blvd., Colorado Springs, CO 80918, (also distributed as part of the CVS program sources available via anonymous FTP from "prisma.com").
- [24] van Schouwen, A.J., "The A-7 Requirements Model: Re-examination for Real-Time Systems and an Application to Monitoring Systems," Technical Report 90-276, TRIO, Queen's University, Kingston, Ontario, May 1990.
- [25] Parnas, D.L., P.C. Clements, D.M. Weiss, "The Modular Structure of Complex Systems," Proceedings, Seventh International Conference on Software Engineering, Mar. 1984.
- [26] Clements, P.C., A. Parker, D.L. Parnas, J. Shore, K. Britton, "A Standard Organization for Specifying Abstract Interfaces, NRL Report 8815, Jun. 1984.
- [27] Faulk, S., B. Labaw, D. Parnas, "SCR Module Implementation Documentation Guidelines," NRL Technical Memorandum 7590-072:SF:FL:DP, Apr. 1983.
- [28] Wang, Y., "Formal and Abstract Software Module Specifications - A Survey," Queen's University Technical Report 91-307, ISSN 0836-0227, May 1991.
- [29] Parnas, D.L., P.C. Clements, D.M. Weiss, "Enhancing Reusability with Information Hiding," Proceedings, Workshop on Reusability in Programming, Sep. 1983.
- [30] Clements, P.C., "Using Information-Hiding as a Design Discipline: Techniques and Lessons," Proceedings, Structured Development Forum VIII, Seattle, Washington, Aug. 1986.
- [31] Parnas, D.L., D.M. Weiss, "Active Design Reviews: Principles and Practices," NRL Report 8927, Nov. 1985.
- [32] Parnas, D.L., "On the Design and Development of Program Families," IEEE Transactions on Software Engineering, Vol. SE-2, No. 1, Mar. 1976.
- [33] Froscher, J.N., C. Meadows, "Achieving a Trusted Database Management System Using Parallelism," in Database Security, II: Status and Prospects, C.E. Landwehr editor, Elsevier North-Holland, 1989.

- [34] Bell, D., L. LaPadula, "Secure Computer Systems: Mathematical Foundations and Model," MITRE Report MTR 2547, v2, Nov. 1973.
- [35] Moore, A.P., "The Specification and Verified Decomposition of System Requirements Using CSP," IEEE Transactions on Software Engineering, Vol. 16, No. 9, Sep. 1990.
- [36] Payne, C.N., A.P. Moore, D.M. Mihelcic, "An Experience Modeling Critical Requirements," Ninth Annual Conference on Computer Assurance, June 1994.