INTEGRATING SECURITY MODELING INTO EMBEDDED SYSTEM DESIGN

By

Matthew Eby

Thesis

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

in

Electrical Engineering

May, 2007

Nashville, Tennessee

Approved:

Dr. Gabor Karsai

Dr. Akos Ledeczi

To My Parents,
for their many years
of dedicated guidance.

ACKNOWLEDGEMENTS

This work was supported in part by TRUST (The Team for Research in Ubiquitous Secure Technology), which receives support from the National Science Foundation (NSF award number CCF-0424422).

I would like to thank my advisor, Dr. Gabor Karsai, for his support and advice which for which I am most indebted. To Dr. Sandeep Neema, Dr. Janos Sztipanovits, Dr. Akos Ledeczi and Dr. Yuan Xue for their guidance and patience, which has tested and stretched me both intellectually and as a person.

Thanks to all the grad students and staff at ISIS for their willingness to listen and help when I encountered problems in my research: Zoltan Molnar, Nagabhushan Mahadevan, Jan Werner, Janos Mathe, Daniel Balasubramanian, Ethan Jackson.

Every accomplishment I have achieved is due to those whose shoulders I stand on. My parents, Dave and Phyllis, have always been supportive and taught me what it means to be dedicated have a strong work ethic. My grandparents, Bob & Sue Hartmann and Jim & Freda Eby, have served as models of what can be achieved through a lifetime of dedication and service. Thanks to my brother, Drew, and my sisters, Michelle and Cara, who have been a large part of my life.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF ABBREVIATIONS

AADL – Architecture Analysis and Design Language

BLP – Bell-LaPadula

BON – Builder Object Network

COTS – Commercial off the Shelf

DAC – Discretionary Access Control

DSML – Domain Specific Modeling Language

GME – Generic Modeling Environment

MAC – Mandatory Access Control

MIC – Model Integrated Computing

MILS – Multiple Independent Levels of Security

MLS – Multi-Level Security

MOF – Meta Object Facility

OCL – Object Constraint Language

OMG – Object Management Group

RBAC – Role Based Access Control

SCADA – Supervisory Control and Data Acquisition

SMAL – Security Model Analysis Language

SMoLES – Simple Modeling Language for Embedded Systems

TCSEC – Trusted Computer System Evaluation Criteria

UML – Unified Modeling Language

CHAPTER I

INTRODUCTION

Embedded systems are a class of computer based systems that cover a broad range of applications. Embedded systems typically meet some or all of the following descriptions:

- designed to perform a dedicated task (not a general purpose computer)
- tight coupling to the physical environment using sensor and actuators
- hard or soft real-time requirements
- limited system resources (CPU, memory, power, etc)

There is an ever increasing concern about security threats as embedded systems are moving towards networked applications [3]. Model based approaches have proven to be effective techniques for embedded systems design [14]. However, existing modeling tools were not designed to meet the current and future security challenges of networked embedded systems. In this thesis, we propose a framework to incorporate security modeling into embedded system design. This thesis presents a security analysis tool that can easily integrate with existing tool chains to create co-design environments that concurrently addresses security, functionality and system architecture aspects of embedded systems.

Security Threats to Embedded Systems

Embedded systems play a crucial role in critical infrastructure, which is essential to national security, success and economic health [3], [4]. Under the Clinton

Administration, the Presidential Decision Directive 63 [1] outlined the importance of cybersecurity on critical infrastructure. In 2003, the Bush Administration followed suit and released *The National Strategy for the Physical Protection of Critical Infrastructure and Key Assets* [2]. Until recently, the computer systems that oversee and control critical infrastructure relied on "security by obscurity" [5]. These systems were one of a kind, proprietary designs that were secure because details of their design were not freely accessible. The recent trend is to build these systems out of commercial off the shelf (COTS) components such as Ethernet, Windows, PLCs and standard web services. This shift opens up these systems to the same sorts of threats that standard desktop PCs face. It is clear that there is increasing concern of the security threats on these kinds of embedded systems [5],[7]. Successful attacks have been reported on the US Power Grid [8] and the sewer system of Australia's Maroochy Shire Council [9]. Other incidents such as a worm infection [12] have affected the Davis-Besse Nuclear Power Plant and CSX Railroad Corp. [9]. To understand the importance of security issues in embedded system design, let's look at two examples of these security breaches.

In January 2003, the Davis-Besse Nuclear Power Plant [9] had been offline for almost a year for repair and maintenance when the Slammer worm brought down safety systems in the plant. The Safety Parameter Display System, which monitors safety factors such as core temperature and radiation sensors, was brought down for five hours and the Plant Process Computer, another monitoring system, was down for six hours. This attack was possible because there was a T1 line directly connecting the plant to an unsecured network of a contractor. This allowed the Slammer worm to bypass the plant's firewall and quickly bring down these safety systems. This breach was an undirected attack by a worm, yet still caused significant damage. What is even more concerning is a directed attack by an intelligent attacker with malicious intent.

One such example of an insider attack occurred at the Maroochy Shire Council waste water plant in Australia [9]. Vitek Boden had worked for a contractor that commissioned a SCADA system at the plant over a period of two years. After the contract was up, Boden positioned himself for a consultant position at the plant and was rejected. Using a stolen laptop computer and off the shelf wireless transmitter from his car he was able to control the entire system and release 264,000 gallons of sewage over 46 break in attempts.

To address such security threats, we need to evaluate how these systems are built and identify what can be done to prevent future threats. Next, we look at the current state of embedded software design process in regards to security design methods.

Current State of Embedded System Design

Model Integrated Computing (MIC) [13] is gaining wide recognition in the field of embedded software design. Models represent embedded software, its deployment platform and its interactions with the physical environment. Models facilitate formal analysis, verification, validation and generation of embedded systems [14]. Hence, this approach is superior to traditional manual software development process. Although, there is modeling tool support for analysis of functionality, performance, power consumption, safety, etc., currently available tools incorporate little if any support for security modeling. As a result, security is looked at only once the complete system has been built. At best, this approach of addressing security in the last stages of development is inefficient taking large amounts of effort to achieve only modest improvements in security. Engineers designing embedded systems usually do not have the experience to address security issues and in many cases are not even aware of the issues [17]. Fixing security vulnerabilities involves releasing patches, which can introduce new problems such as viruses [18] or security vulnerabilities [19]. Still, systems designed without security in mind are intrinsically insecure. Patches can fix specific security vulnerabilities, but do not address poor system architecture.

Many times vulnerabilities are only discovered once they have been exploited. To address unknown threats, systems can be isolated in private corporate networks using firewalls and intrusion detection systems. But such perimeter defenses, even if they are flawless, cannot protect against insider attacks [20]. In light of this situation, we advocate modeling environments that incorporate security into the early design phase of embedded systems.

Model based approaches have the advantage of executable specifications. Once the system is implemented the models can serve as documentation of the system architecture. This could have helped prevented the worm attack on the Davis-Besse plant. The T1 line through which the worm entered could have been flagged by model analysis tool as an unsecured entry point into the facility.

One of the few modeling languages with security extensions is UML. Currently available extensions to UML provide: access control [21] [22], fair exchange, assumptions about confidentiality and integrity [23]. There are existing tools that can guarantee some security properties using automated proof verifiers [23]. However, UML-based Model-driven Security is not sufficient for design and analysis of embedded systems. We strongly believe that embedded system design can benefit from Domain-

Specific Modeling Languages (DSML) as opposed to the one-size-fits-all approach of UML. For example, there is no concept of hardware in UML which makes it ill suited for embedded systems with their diverse hardware architectures. In many embedded applications system resources are scarce. Added overhead for security can have drastic effects on performance. An ideal embedded software development environment will allow the engineer at design time to analyze security and performance tradeoffs based on the hardware platform.

Problem Statement

MIC can meet the challenges of designing secure embedded systems. A key advantage of the model based approach is the abstraction of the application domain. This abstraction is facilitated through the use of DSMLs. A DSML provides a system designer a set of concepts that are specifically tailored for a certain application domain. In our case, the domain is networked embedded real-time systems, such as process control systems, automotive, avionics and robotics systems. A DSML with the proper level of abstraction hides the inconsequential details of a system while allowing the engineer to shift focus to more important aspects. There are many examples of DSMLs developed for embedded system design in different domains [MILAN [24], SMoLES [26], AADL [24]]. We propose an extension mechanism for DSMLs that adds security concepts similar to UML extensions [21]. By extending embedded system DSMLs, we can add tool support for security analysis, validation, verification and generation. These security tools will extend the large tool chains that already exist for embedded system design.

CHAPTER II

BACKGROUND AND LITERATURE SURVEY

MLS: Multi Level Security

Multi Level Security is an approach to enabling computer systems to handle and process data with differing levels of sensitivity. The Trusted Computer System Evaluation Criteria (TCSEC) [27] is a Department of Defense standard that sets the criteria for evaluating MLS systems. A MLS system implements some sort of access control mechanism that defines what users have access to which data. There are two types of access control defined by TCSEC: Discretionary Access Control (DAC) and Mandatory Access Control (MAC). DAC protects information on a need-to-know basis. Users are permitted access to objects based on their group identity and are able to pass this permission on to other users at their discretion. MAC protects information by assigning clearances to users which define what the sensitivity level of information they are allowed to access. The clearances signed to users under MAC are strictly enforced; permissions can not be delegated at a user's discretion. Two models for enforcing MAC are the Bell-LaPadula and Biba model which are discussed below.

Bell-LaPadula and Biba Security Models

The two traditional models for dealing with information flow in systems are the Bell-LaPadula model [28] and the Biba model [29]. Both of these models enforce an access control scheme that defines the rights of a subject to access information. The Bell-LaPadula security model deals with confidentiality of information in computer systems.

The Biba model deals with integrity of information in computer systems. Both models view a system as a set of subjects and objects. Subjects can be human users or software processes and objects can be data or files stored on the system. Subjects and objects are assigned a security level and a compartment which define what information a given subject is permitted to access. Below are the concepts that Bell-LaPadula originally defined and then were used by Biba:

$S = \{S_1, S_2, \cdots, S_n\}$	Set of all Subjects in system
$O = \{O_1, O_2, \cdots, O_m\}$	Set of all Objects in system
$C = \{C_1, C_2, \cdots, C_q\}$	Set of Security Levels (i.e. Classified, Secret, Top Secret)
$K = \{K_1, K_2, \cdots, K_q\}$	Set of Compartments (i.e. NATO, FBI, CIA)

Figure 1 Bell-LaPadula Concepts

A security designation is a pair (Security Level, Set of Compartments). Each subject in a system is assigned a security designation and each object is assigned a security designation. The set of all security levels is an ordered set that can be evaluated as an inequality (i.e. Top Secret > Secret). Compartments are an unordered set. Under Bell-LaPaduala a subject, S_1, is only allowed access to an object, O_1, if the security designation of S_1 dominates the security designation of O_1. Domination is defined as:

$$(C_1, K_1) \text{ dominates } (C_2, K_2)$$

If and only if

$$C1 \geq C2 \text{ and } K1 \supseteq K2$$

Bell-LaPadula defines two security properties, the Simple Security Property and the * (star) Property. When enforced, these two properties guarantee the confidentiality of objects in the system. These two properties define what rights a particular subject has to read or write an object.   The Simple Security Property states that a subject may read information from an object only if the subject's security designation dominates the object's security designation. The * Property states that a subject may write information to an object only if the object's security designation dominates the subject's security designation.

One of the main criticisms of Bell-LaPadula as a security model is that it only addresses confidentiality of information. In response to this weakness the Biba model was defined to address integrity of information. Confidentiality deals with the secrecy of information while integrity deals with which information is trustworthy. The Biba also enforces a Simple Security Property and a * Property. The Biba version of the Simple Security Property states that a subject may read information from an object only if the object's security designation dominates the subject's security designation. The Biba version of the * Property states that a subject may write information to an object only if the subjects's security designation dominates the objects's security designation.

RBAC: Role Based Access Control

RBAC is form of access control in computer based systems. RBAC is used as a method for efficiently managing the assignment of permissions between subjects and resources. Subjects (human users or processes) are assigned permissions which define what actions the subject is permitted to perform on a resource.

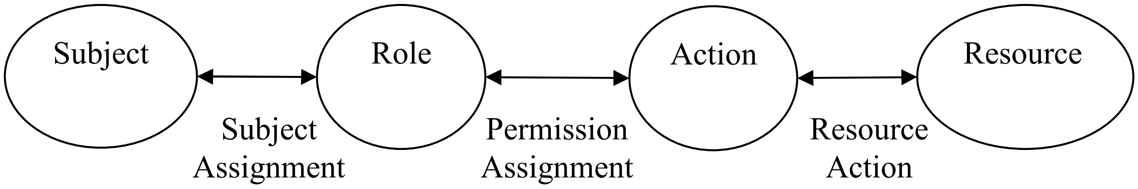


Figure 2 Role Based Access Control Diagram

Figure 2 shows the basic role based access control scheme. In a computer system there are resources such as a file or memory segment. For each resource there is a resource to action mapping that specifies what actions are possible on that given resource. Actions can be such things as read, write and execute. A subject is assigned one or more roles through the subject assignment mapping. Roles are given permissions to perform actions on resources through the permission mapping. This subject then inherits from its assigned roles any permissions that the roles have. A subject may want to perform an action on a certain resource. The advantage of RBAC is that the permission mapping does not directly map actions to subjects. Mapping actions directly to subjects results in state explosion. By using a role to indirectly map actions to subjects, state explosion is avoided. Often, it is feasible to use a small number of roles for a large number of subjects. This minimizes the number of mappings that need to be made when a new subject is created.

MILS: Multiple Independent Levels of Security

The TCSEC served as the official DoD standard for secure computer systems from 1983 until 2005 when it was replaced by the Common Criteria [30]. Under TCSEC systems are classified in one of four divisions: D, C, B, and A. Where the divisions have subdivisions that are labeled as follows: D, C1, C2, B1, B2, B3, and A1. These divisions were replaced by what Common Criteria defines as Evaluation Assurance Levels (EAL). There are seven EALs which are labeled as EAL1 through EAL7 with EAL7 being the highest assurance level. For full descriptions of the assurance levels refer to the Common Criteria [30] and TSCEC [27] specifications. The highest levels of assurance (A1 under TCSEC and EAL7 under Common Criteria) require formal verification of security properties to be certified. The cost of evaluating and certifying software to these assurance levels can be prohibitive. The MILS architecture aims to reduce the size and complexity of the code that must be evaluated at the higher assurance levels. The MILS approach used a partition kernel [10] as a means to isolate security critical code from non security critical code. By isolating the security critical code in a separate partition, the effort is only needed to verify the code in the security critical partition. Effort is not wasted on verifying non security critical code. The concept of a partition kernel was first introduced by John Rushby [10]. Partition kernels provide a mechanism to guarantee safety and security in computer based applications. In systems where both trusted and untrusted processes are running, measures need to be taken to guarantee that the untrusted processes do not comprise the trusted processes. A partitioning kernel guarantees that processes running in separate partitions are separated both spatially and temporally.

UMLsec

The Unified Modeling Language (UML) [11] is a widely used standard with many tools supporting the language. One of the key features of UML is that it is general and expressive enough to capture models in a wide variety of domains. However, this expressiveness also means that it is not tailored to any specific domain. To overcome this generality there is a standardized UML extension mechanism that allows users to specify extensions through use of stereotypes, tagged values and constraints. UML Tool developers must specify how UML will be used (i.e. naming of stereotypes) so that domain concepts can be expressed and their tools understand the models. This specification of a UML extension is referred to as a UML profile. UMLsec is a security verification framework built by Jan Jürjens [23] using these extension mechanisms. UMLsec enables secure system development by specifying security requirements of a system. These security requirements are concepts like secrecy, integrity, and authenticity. By using automated theorem provers, system diagrams are verified against the specified requirements.

Survey of Embedded System DSMLs

The MIC approach uses DSMLs to provide the system modeler with a layer of abstraction that is suited for their specific application domain. A DSML is a five tuple L = <C, A, S, M_S, M_C> where:

- C: the concrete syntax describes the notation for representing models (whether textual or visual)
- A: the abstract syntax describes the set of all valid models in the DSML
- S: the semantic domain describes the meaning of a model, usually in terms of a mathematical domain
- M_S: the semantic mapping maps abstract syntax to the semantic domain
- M_C: the parsing of concrete syntax is based on the abstract syntax

In MIC, DSMLs are defined by UML class diagram style metamodels where the concrete syntax of the metamodel defines the abstract syntax of the DSML. The metamodel consists of the UML-like class diagram along with OCL constraints on classes. This abstract syntax describes the set of all valid models in the DSML.

The Object Constraint Language (OCL) is a language for defining well-formedness rules for DSMLs. Often a class diagram is not expressive enough to define a modeling language. For example, a containment relationship in a class diagram specifies that one object can be contained by another. However, the containment relationship in and of itself is not sufficient in all cases. Perhaps there is a scenario when containment can only be allowed if a certain condition is met (i.e. a document can only be contained in a hard drive if that hard drive is larger than the document size). OCL allows rules to be defined that place conditions on associations defined in the class diagram.

The goal of the MIC approach is to provide the system modeler a modeling environment that is highly tailored to their application domain. This approach is preferable to a modeling environment that is targeted to a large variety of domains. This gain in generality is often at the sacrifice of usability. The advantage of using a DSML is that the modeling environment is tailored to a specific domain, allowing the modeling language to achieve the best balance between power and usability. There have been several DSMLs developed using the Generic Modeling Environment (GME) [14][15] that are targeted toward modeling, simulation, analysis, and generation of embedded systems. GME is a metaprogrammable tool which facilitates the graphical implementation of DSMLs through the use of metamodels. GME has a framework for tool developers to design interpreters for a DSML. Interpreters are executable code that have access to a model. Interpreters are capable of such activities as traversing models, modifying models, providing feedback to modelers or generating code. The embedded system DSMLs described in this thesis have been implemented in GME.

SMoLES: Simple Modeling Language for Embedded Systems

The Simple Modeling Language for Embedded Systems (SMoLES) was designed to have a concise syntax that allows constructing embedded systems from components [26]. The components are assumed to be concurrently executing objects that communicate and synchronize with each other. Furthermore, objects can perform blocking I/O operations in which they wait for the result, while other objects can execute. Communication between components means passing data from a source component to a destination component, which is then enabled to run in order to process the data. In addition to data triggering, periodic timers can also trigger the components. The language consists of components and assemblies. Components are the elementary building blocks, and contain input and output ports, which are used to receive/send data tokens from/to other components. Assemblies contain components, and describe how they are interconnected. Like components, assemblies can have their own input and output ports, and assemblies can contain other assemblies. Assemblies are organized into a hierarchy. The various components and assemblies in the hierarchy communicate with each other through the dataflows, as specified by the designer. SMoLES has a code generation utility that will interpret the model and output C++ code that will execute on top of a small custom dataflow kernel. Figure 3 shows a example model in the SMoLES DSML.

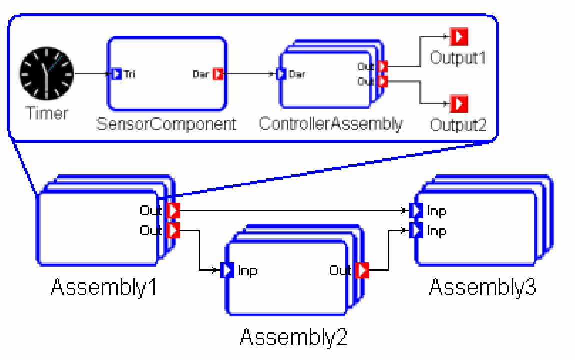


Figure 3 Example SMoLES Model

AADL: Architecture Analysis and Design Language

The Architecture Analysis and Design Language [25] is targeted toward avionics and automotive embedded applications, however it is suited for any embedded application with real-time requirements. The goal of AADL is to have a common language that can model both the software and hardware architecture of a system. This allows the system designer to analyze the interaction between software and hardware before the system is built. This approach of analyzing the system early in the development process greatly reduces project overhead including cost and time. Embedded systems are especially sensitive to complex interactions between hardware and software if they have real-time requirements. AADL provides the ability to assure such things as schedulability, power consumption, safety and fault tolerance.

AADL views a system in terms of system components, hardware components and software components. These components are defined with type and implementation component declarations. A component type declaration specifies the component interface. The interface of the component defines the externally observable attributes such as ports that are connection points with other components. A component implementation declaration specifies a component's internal structure (i.e. the subcomponents and how the subcomponents are interconnected). The system component as its name suggests is the top level component of an embedded system. The root level of an AADL model must be a system. Systems can contain other subsystems.

The top level software component is a process. A process component represents a protected address space whose boundaries are enforced at runtime. Processes can contain threads and thread groups. Process components are contained by system components. Thread components represent a schedulable unit that can execute concurrently. Thread groups are used for grouping data, threads and thread groups inside of processes. Data components represent data in source text and subprogram components represent a callable piece of source code. Both data and subprograms are contained by processes, threads and thread groups.

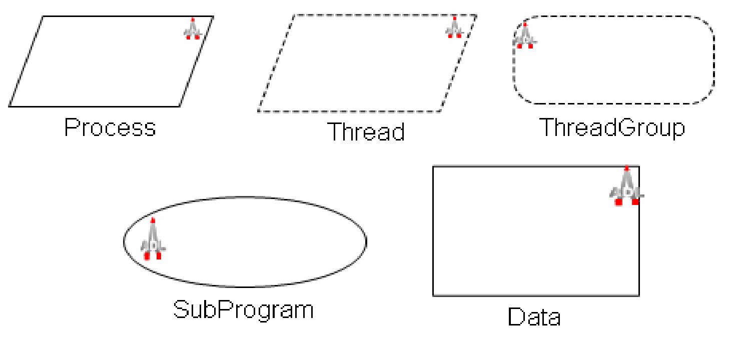


Figure 4 AADL Software Components

Figure 5 shows the types of hardware components available in AADL. The hardware topology can be modeled by defining memory, processors and devices interconnected by buses. All hardware components can be contained inside system components. Memory components can also be contained inside a processor to represent a cache or contained in another memory component.

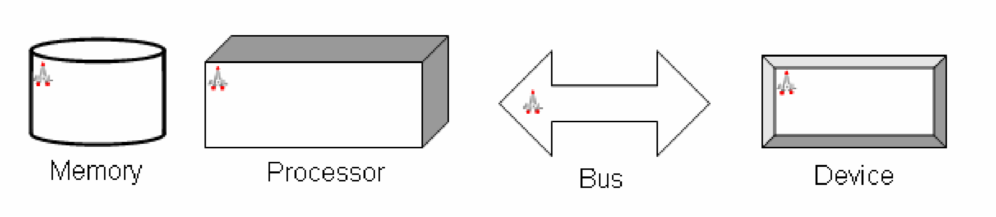


Figure 5 AADL Hardware Components

Figure 6 shows an example of an AADL model. Components have features that define how they interface with other components. In this particular example, Pilot_Display is a device and the other components are processes. Each of these components has input and output ports that accept incoming and outgoing connection from other system components.

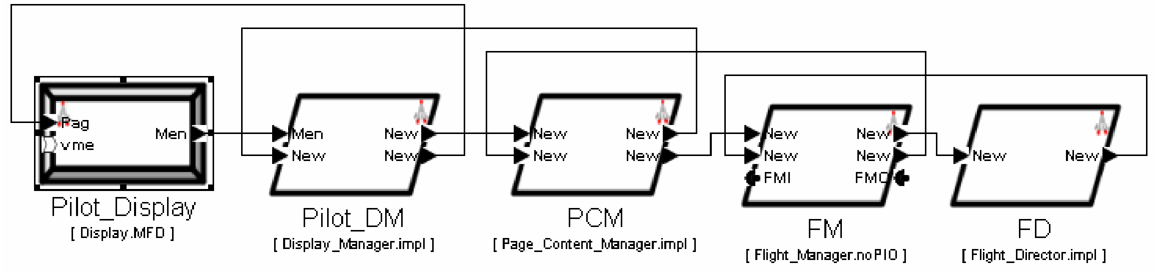


Figure 6 Example: AADL Model

AADL components also have properties. The component view of the system is a graphical representation of the system architecture. However, components also have textual properties that describe the given component. These properties describe the components and how they interact. Some examples of component properties are:

- clock speed for a processor component

- attribute classifying threads as periodic, aperiodic or sporadic
- worst case execution time of a subprogram
- maximum latency over a port connection

The system architecture along with the properties of components allows the system to be analyzed for such things as schedulability, power consumption, safety, etc.

Summary of Background and Literature Survey

Multilevel secure systems describe access control mechanisms (MAC and DAC) to protect access to sensitive information. MILS provides an architecture for reducing the effort required to verify MLS system to the higher assurance levels. UMLsec provides UML extensions that facilitate the automated verification of system security properties, such as those properties of MLS systems described in this chapter. However, the UMLsec extensions are not tailored to any specific domain. DSMLs provide modelers with concepts suited to their application domain. Both, AADL and SMoLES are examples of DSMLs that have been successfully used to design embedded systems. However, these DSMLs are not meant to capture concepts from the security domain. There needs to be security specific extensions to DSMLs able to capture security concepts such as those concepts in UMLsec. These security specific extensions can be integrated with languages such as AADL and SMoLES to capture and analyze the security properties of embedded systems.

CHAPTER III

METAMODEL COMPOSITION

The goal is to create security extensions to existing embedded system design languages that can be reusable. The mechanism that enables this reuse is metamodel composition. Metamodel composition can be described by the following equation:

$$DSML_A \circ DSML_B \Rightarrow DSML_C$$

This composition is specified by defining how two DSMLs (*$DSML_A$* and *$DSML_B$*) are joined together to form a common DSML (*$DSML_C$*). The resulting DSML will have properties of both the original DSMLs. If there are two different DSMLs that support the design of embedded systems, where models in the two DSMLs are able to capture properties related to different aspects of the system then it would be advantageous to compose these two languages.

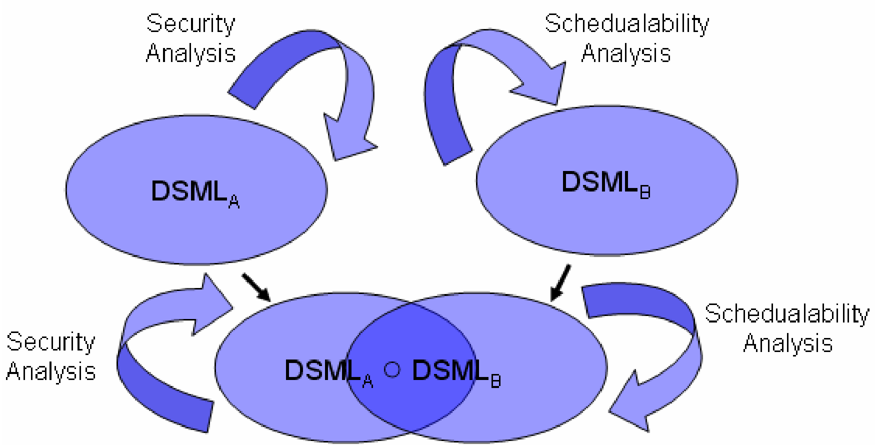


Figure 7 Metamodel Composition

Figure 7 shows an example of two composed DSMLs where models in *DSML$_A$* capture some security related properties of embedded systems and models in *DSML$_B$* capture some schedulability related properties of embedded systems. If a system designer wants to analyze a system for both security and schedulability he/she would have to create two systems models, one in *DSML$_A$* and one in *DSML$_B$*. For small, trivial examples this may be feasible, but for real world applications these systems are on a much larger scale, many times with of thousands of components. It is not feasible to have two separate models of such large systems. It would be difficult to ensure that both models are congruent with each other, since changes to security properties can have effects to schedulability and vice versa. The solution to this is to compose *DSML$_A$* and *DSML$_B$* to form a common DSML where a single model can capture both security and schedulability properties. This forms a codesign environment where a single model with multiple aspects can be analyzed. This proves to be useful since embedded system properties such as security, schedulability, power consumption, functionality, etc. are often tightly coupled with each other. By creating this codesign environment, there is a common interface where tradeoffs can be made between, as in this example, security and schedulability.

GME Mechanisms for Metamodel Composition

GME supports the composition [15] of metamodels such that two existing languages can be combined to form a new language with the properties of both the original languages. The user can import a read-only copy of a metamodel using the GME library function. The power of this is that users can create libraries of metamodels where changes in the original metamodel propagate to anywhere that the metamodel is used.

This library feature can be used to either compose or extend a metamodel. GME supports several operators that enable the composition of metamodels. The user can create proxies (or references) to classes in the library metamodel and then use one of these operators to specify how the metamodels are composed. The first operator is the class equivalence operator. This represents the full union between two class objects. The inheritance operator is a directional operator that defines a parent-child relationship between two classes. The child object has all the attributes and associations of the parent in addition to any refinements to the child object. For cases when finer grained inheritance operations are necessary there are two more specialized inheritance operators: Implementation Inheritance and Interface Inheritance. The Implementation Inheritance operator defines a parent-child relationship where the child inherits all the attributes of the parent class and all the containment relationships where the parent acts as the container. The Interface Inheritance operator defines a parent-child relationship where the child does not inherit any of the attributes and inherits all the association attributes except those containment associations where the parent acts as the container. Figure 8 shows these composition operators [15].

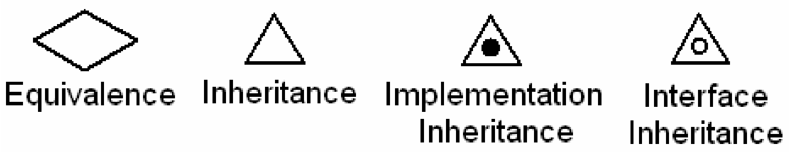


Figure 8 GME Metamodel Composition Operators

Described above are the mechanisms available in GME that are used to compose metamodels. The following chapters will look at two example DSMLs that will be composed. A DSML that captures security properties is composed with a DSML for embedded system design. More in depth information about metamodel composition using GME can be found in [16].

CHAPTER IV

SECURITY MODEL ANALYSIS LANGUAGE

This chapter will demonstrate a process for integrating security analysis into existing tool chains to create a security co-design environment. The approach taken is to create a single common DSML that is used to capture and analyze security properties of systems, and only those. The advantage of this approach is that the effort needed develop the security analysis tool is only spent once. Then this tool can be incorporated into existing embedded systems languages with minimal effort. By defining mappings from an embedded system DSML onto the security analysis DSML, it is possible to analyze the security properties of the embedded system model. Figure 9 illustrates the process of defining mappings from one or more DSMLs onto a language supporting security analysis and feeding the analysis results back to the DSML.

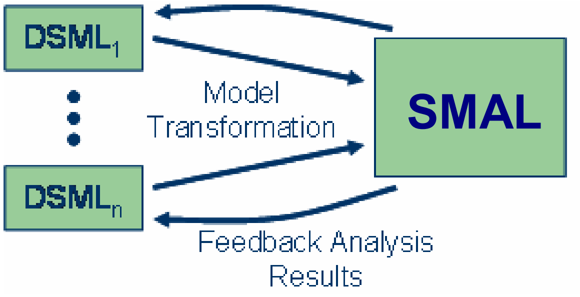


Figure 9 Mappings from DSMLs to SMAL enable security analysis of the DSMLs

Security Model Analysis Language (SMAL), enables a user to model and analyze security related properties of embedded systems. (Note that while SMAL is technically a DSML, from this point out the term DSML will only be used in reference to a language for embedded systems design to which we wish to add security analysis capabilities.) The purpose of this analysis tool is to identify points in the system model that violate certain security requirements and provide useful feedback to the modeler. SMAL allows such violations to be identified and remedied at design time before they can be exploited. Currently, SMAL supports two types of analyses: information flow analysis and threat model analysis. These model analyses and the SMAL metamodel are detailed in the following sections.

Information Flow Analysis

To analyze the Bell-LaPadula and Biba models, SMAL views a system as a set of partitions, a set of data objects contained in each partition and the dataflows inside and across the partitions. Figure 10 shows a screen shot of a SMAL model with *Partitions*, *InformationFlows* and *Data* models.

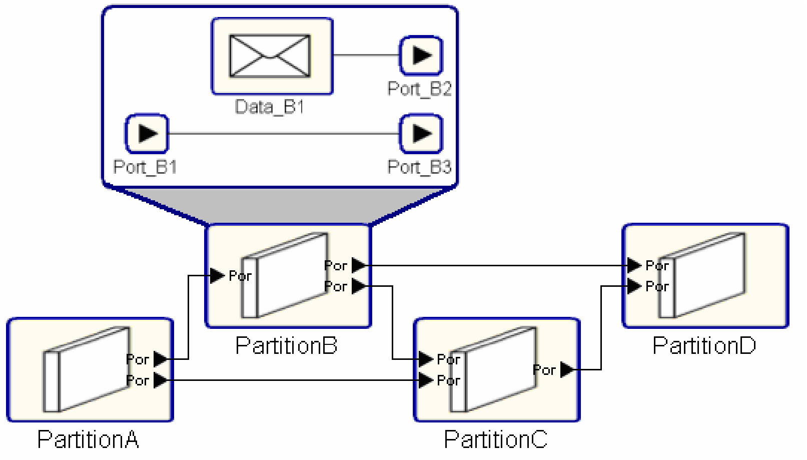


Figure 10 Partitions and dataflows in SMAL

Dataflows are represented as connections between input and output ports on a partition. In SMAL, *Partitions* are the subjects and are assigned a *SecurityLevel* and *Compartment* attributes. A data object inherits the *SecurityLevel* and *Compartment* classification of its containing partition. SMAL allows the *SecurityLevel* to be an integer value and the *Compartment* to be a string value. Our analysis tool treats each data object as the root node in a tree search algorithm. The tool will traverse the dataflow paths originating from a data object and verify that each partition through which that data object flows has a *SecurityLevel* and *Compartment* that permit that partition to access the data object. Bell-LaPadula does not allow information to flow to a lower *SecurityLevel,* while Biba does not allow information to flow to a higher *SecurityLevel*. When composed, these two models only allow information to flow between partitions with the same *SecurityLevel*. Applying both models is too restrictive in a system where the designer does not need to restrict access to all data objects. There may be some data objects that have a confidentiality requirement but no integrity requirement and vice versa. To provide a less restrictive model, data objects in SMAL are assigned two Boolean attributes, *confidentiality* and *integrity*. The flow of every data object is evaluated based on the settings of these attributes. When *confidentiality* is true, the Bell-LaPadula model is enforced and when *integrity* is true, the Biba model is enforced on the flow of that data object between partitions.

Threat Model Analysis

In a distributed system, partitions may reside on multiple nodes and data is transferred between these nodes over some communication channel. The information flow analysis addresses the movement of data explicitly defined in the system model but does not address man-in-the-middle attacks on the physical channel. To prevent such attack, the communication channel must be encrypted. Adversary modeling in SMAL enables the analysis tool to identify vulnerable channels and determine which encryption algorithms can be used to protect data being transmitted on that channel. Figure 11 illustrates the adversary model. In each *System* there is an *EncryptionAlgorithms* library that contains the set of all encryption algorithms that can be used to encrypt a channel. Each *System* also contains a set of *Adversary* models that define which encryption algorithms are vulnerable in the context of that adversary. Each *Adversary* contains a set of *Susceptibility* references. Each reference refers to an *EncryptionAlgorithm* that is defined in the *EncryptionAlgorithms* library. The *Susceptibility* reference represents that the *EncryptionAlgorithm* is susceptible to attack by the containing *Adversary*. This reference has an attribute, *MaxKeySize*, which means that the referenced *EncryptionAlgorithm* is susceptible to that adversary if the strength of its encryption is not greater than *MaxKeySize*. Together, the *EncryptionAlgorithm* library and *Adversary* models allow our analysis tool to determine which algorithms are safe to use to encrypt information flows. Each *InformationFlow* in SMAL has an attribute, *AnticipatedAdversary*, which identifies the *Adversary* model associated with that *InformationFlow*. Each *InformationFlow* in SMAL also has an *EncryptionAlgorithm* and *KeySize* attribute. For each *Information Flow* in the *System*, the analysis tool checks the *EncryptionAlgorithm* and *KeySize* attribute against the set of encryption algorithms that are susceptible to the adversary model specified by *AnticipatedAdversary*.

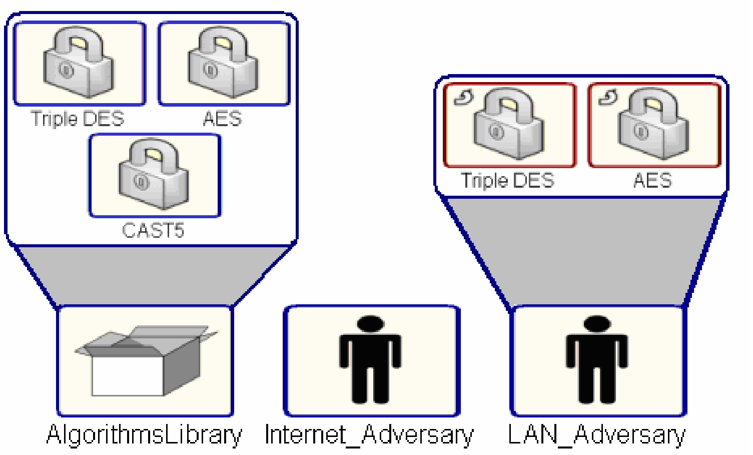


Figure 11 Encryption algorithms library and adversary models in SMAL

SMAL Metamodel

Figure 12 shows the metamodel of SMAL which is implemented in GME using the MetaGME paradigm.

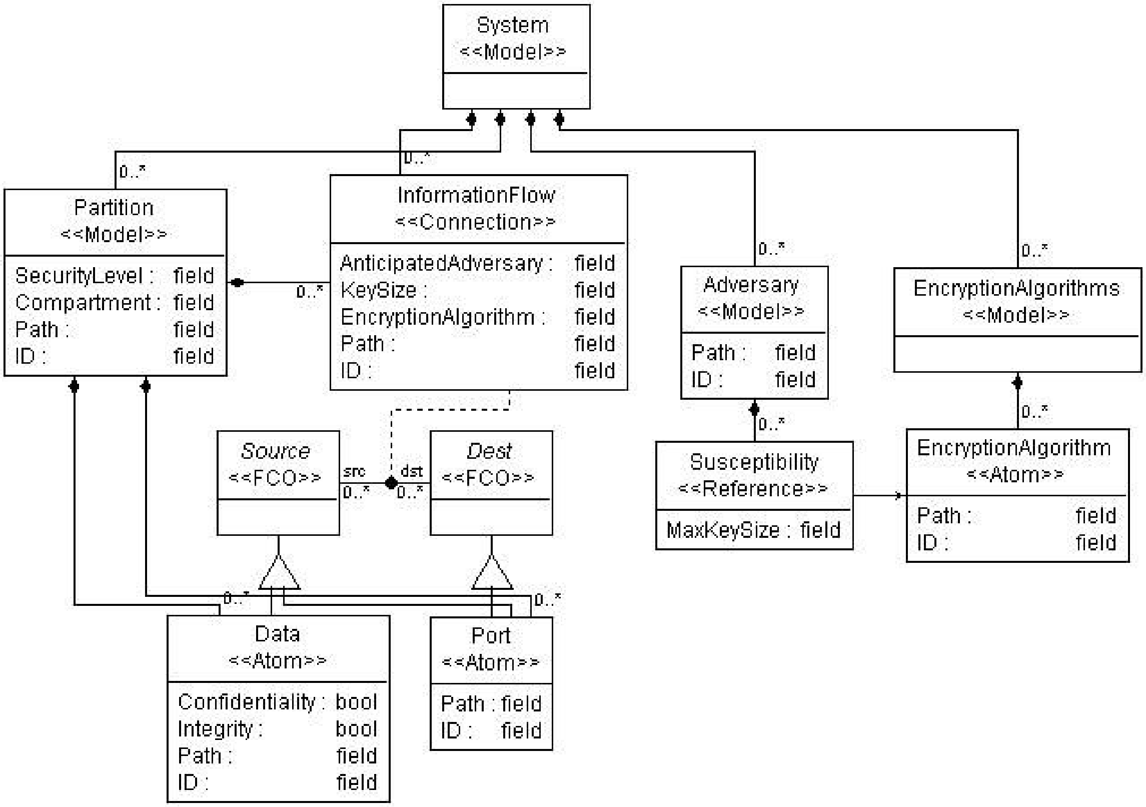


Figure 12 SMAL Metamodel

Short descriptions of the metamodel concepts used in SMAL are shown in Figure 13. For more in depth reading about these metamodeling concepts refer to [15].

<<FCO>>	*FCOs* are abstract classes that serve as a base class for other classes (model, atom, connection, set, reference) in a metamodel.
<<Model>>	By *model* we mean an object that represents something in the world. Models have internal structure and can contain other FCOs.
<<Atom>>	*Atoms* (or *atomic parts*) are simple modeling objects that do not have internal structure (i.e. they do not contain other objects), although they can have attributes. Atoms can be used to represent entities, which are indivisible, and exist in the context of their parent model.
<<Connection>>	*Connections* are associations between two objects in a model. They are visualized as a line connecting the two objects.
<<Reference>>	*References* are parts that are similar in concept to pointers found in various programming languages.

Figure 13 MetaGME Concepts

In SMAL, System models are contained in the root folder. Each System model can contain Partion models, InformationFlow connections, Adversary models and EncryptionAlgorithms models. Several of the classes in the SMAL metamodel have an

ID and a Path attribute field. These attributes support the transformation of models from an embedded system DSML to the SMAL paradigm. All objects in a GME model have an id number that uniquely identifies that object within the model and all objects have a path relative to the root folder. During the transformation process, this path and id information is lost. The id and path of an object in the SMAL model will not be the same id and path of that object in the corresponding DSML model. To preserve this information the path and unique id of each object in the DSML model are written to the attribute fields of the corresponding SMAL objects during the model transformation.

The portion of the SMAL metamodel that supports information flow analysis includes: System models, Partition models, InformationFlow connections, Data atoms and Port atoms. Partitions models have an integer attribute for SecurityLevel and a string attribute for Compartment. These attributes support the information flow analysis. The Partitions can contain Data atoms, Port atoms and InformationFlow connections. Data atoms have boolean attributes for Confidentiality and Integrity requirements. The InformationFlow connections can connect Data atoms to Port atoms or Port atoms to Port atoms. These InformationFlow connections can be between and across partitions.

To support the adversary analysis more classes are added to the metamodel. The additional portions of the SMAL metamodel needed to support adversary analysis include: Adversary models, EncryptionAlgorithms models, EncryptionAlgorithm atoms, and Susceptibility references. The System model can contain Adversary models and EncryptionAlgorithms models. The EncryptionAlgorithms models can contain EncryptionAlgorithm atoms. The Adversary models can contain Susceptibility references, which reference EncryptionAlgorithm atoms. The Susceptibility reference contains an integer attribute for MaxKeySize. InformationFlow connections have a string attribute for the AnticipatedAdversary and Algorithm as well as an integer attribute for KeySize.

Formalization of Analysis as OCL Constraints

Figure 14 shows the well-formedness rules for SMAL. There are three constraints: Biba, BLP and Compartment which call the functions BibaViolation, BLPviolation and CompartmentViolation respectively. These three constraints capture the essence of the information flow analysis. The other constraint (Adversary) and constraint functions (getMKS and SusceptibleAlgorithm) capture the essence of the adversary analysis which is discussed in the next section.

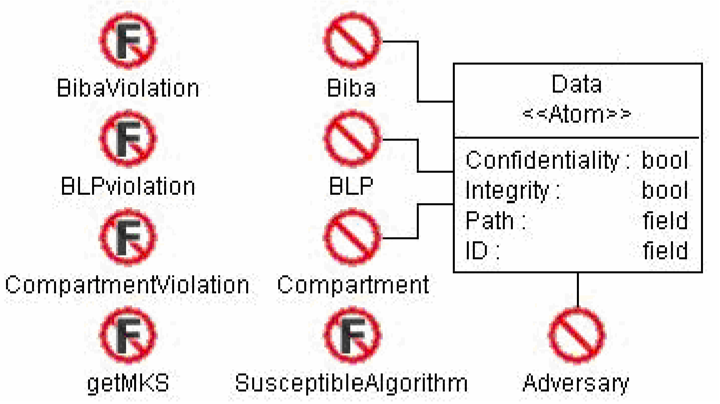


Figure 14 Constraints associated with Data in SMAL Metamodel

Figure 15 shows the OCL statements that formalize the information flow analysis and Figure 16 shows the OCL statements that formalize the adversary analysis. These have been written in a simplified form for readability purposes. The full version of these constraints can be found in Appendix A.

```
Constraint Biba
data.Integrity implies not data.BibaViolation()

Constraint Function BibaViolation
data.dstInformationFlow()->iterate(port; violation = false |
     if(data.parent().SecurityLevel < port.parent().SecurityLevel) then
          true
     else
          violation || port.BibaViolation()
     endif
)

Constraint BLP
data.Confidentiality implies not data.BLPviolation()

Constraint Function BLPviolation
data.dstInformationFlow()->iterate( port; violation = false |
     if(data.parent().SecurityLevel > port.parent().SecurityLevel) then
          true
     else
          violation || port.BLPviolation()
     endif
)

Constraint Compartment
data.Confidentiality || data.Integrity implies not
data.CompartmentViolation()

Constraint Function CompartmentViolation
data.dstInformationFlow()->iterate( port; violation = false |
     if(data.parent().Compartment <> port.parent().Compartment) then
          true
     else
          violation || port.CompartmentViolation()
     endif
)
```

Figure 15 OCL constraints for information flow analysis

```
Constraint Adversary
data.Confidentiality || data.Integrity implies
not data.SusceptibleAlgorithm()

Constraint Function SusceptibleAlgorithm
if (data.srcInformationFlow()->iterate(conn; violation = false |
     if( conn.AnticipatedAdversary <> "") then
            let MKS = conn.getMKS(conn.src()) in
            let KS = conn.KeySize in
            MKS >= KS
     else
          violation
     endif
)) then
          true
```

```
else
          data.dst().exists(t | t.SusceptibleAlgorithm())
endif

Constraint Function getMKS(source)
source.parent().parent().modelParts(Adversary)->iterate(a;
MKS = 0 |
     if (a.name = InformationFlow.AnticipatedAdversary) then
          a.referenceParts(Susceptibility).any(a | a.name =
          self.EncryptionAlgorithm).MaxKeySize
     else
          MKS
     endif
)
```

Figure 16 OCL constraints for adversary analysis

These constraints specified above are enforced on SMAL models using the OCL constraint checker built into GME. However, these rules have also been implemented in a C++ interpreter. This provides several advantages over OCL constraints, which were written for formalization purposes. For large models the interpreter will execute much faster than the OCL constraints can be evaluated. The interpreter is able to provide much more useful feedback messages to the user when there is an error in the model. The OCL constraints will only return that a particular data object is vulnerable, but do not identify where in the model this vulnerability occurs. The interpreter identifies the information flow that causes the vulnerability. For this reason, the interpreter can identify more vulnerabilities than the OCL constraints. The algorithm used in the OCL constraints is a tree search algorithm that stops executing and returns a warning when the first vulnerability is found. The algorithm used in the interpreter is a tree search algorithm that will traverse the entire tree and return a warning for each vulnerability. For example, if a data object has a confidentiality requirement that is violated at multiple points in the model, then the OCL constraint will only return one message, while the C++ interpreter will return a message for each point that violates the Bell-LaPadula Model. The interpreter will output the results to an XML file which can be read from outside the SMAL environment. This enables SMAL to serve as a tool to analyze and provide feedback about information flows in other DSMLs. Figure 17 shows pseudo code that represents the logic used in the BON interpreter.

```
main()
{     for all data
            TreeSearch(data,data)
}

TreeSearch(rootnode,node)
{     if rootnode.integrity
      {
            if node.SecurityLevel > rootnode.SecurityLevel
                  Message(Biba Model Security Level Violation)
            if node.Compartment is not subset of rootnode.SecurityLevel
                  Message(Biba Model Comparment Violation)
      }

      if rootnode.confidentiality
      {
            if rootnode.SecurityLevel > node.SecurityLevel
                  Message(Bell-Lapadula Model Security Level Violation)
            if rootnode.Compartment is not subset of node.SecurityLevel
                  Message(Bell-Lapadula Model Comparment Violation)
      }

      if rootnode.integrity or rootnode.confidentiality
      {
            for each node.children
            {
                  conn = node.getConnectionTo(child)
                  aa = conn.AnticipatedAdverary
                  ks = conn.KeySize
                  alg = conn.EncryptionAlgorithm
                  if aa contains reference to alg and

            }
      }

      for all node.children
            TreeSearch(rootnode,child)
}
```

Figure 17 Pseudo code representation of the C++ interpreter logic

Integrating Security Analysis with Existing Tool Chains

Although there is modeling tool support for analysis of functionality, performance, power consumption, safety, etc., currently available tools incorporate little if any support for security modeling. As a result, security is only addressed once the complete system has been built. We want to leverage the work behind existing tool chains by incorporating security analysis in the system design process. SMAL was created to be a reusable tool that can be integrated with multiple tool chains, thus reducing the effort that would be required to develop custom security analysis for each tool chain.

When adding these security specific concepts to a DSML, it is important to consider what they mean in the context of the entire tool chain. Often the design flow includes other tools for such things as functionality, schedulability, power consumption and safety analysis. The division between these different types of analyses is not always clear-cut. Many times decisions made based on one type of analysis can have an impact on the outcome of other types of analysis. One such example in the context of SMAL is the use of encryption algorithms. The decision to encrypt a communication channel could have a major effect on the schedulability of the system. Also, if there is a code generator for the DSML, it must be modified to support these security properties (i.e. linking to encryption libraries, enforcing the partition model, etc.). The tool developer who is integrating SMAL capabilities to a DSML, must address concerns such as making these other tools aware of the impact the security properties will have on the system. Figure 18 shows a typical design flow for performing security analysis with an embedded system DSML. The portions of this design flow that include the transformation to a SMAL model, the security analysis and the result feedback form an automated process.

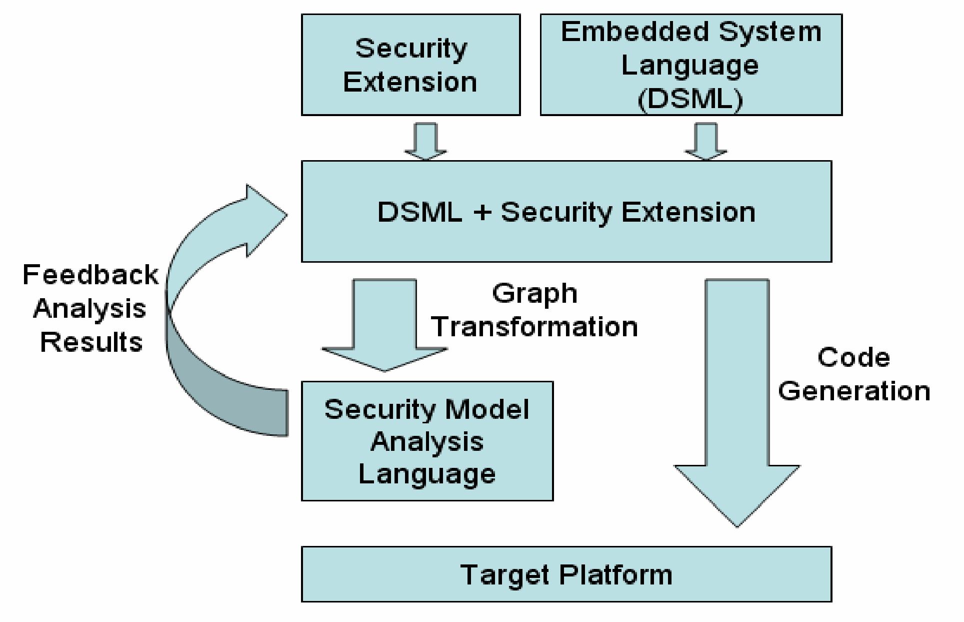

Security Extension
Embedded System Language (DSML)
DSML + Security Extension
Feedback Analysis Results
Graph Transformation
Code Generation
Security Model Analysis Language
Target Platform

Figure 18 Typical embedded system design flow with SMAL

Model Transformation to the SMAL Paradigm

In general, model transformation is a one way function with the domain being the set of all valid models in the original DSML and the range being the set of all valid models in the destination DSML. By defining a transformation that maps models of an embedded system DSML onto SMAL, we can perform information flow analysis and threat model analysis on the embedded systems models. This prevents the repeated effort of developing interpreters that analyze security properties for multiple DSMLs. Only the interpreter that analyzes SMAL models needs to be developed.

GME is integrated with the Graph Rewriting and Transformation language (GReAT) [32]. GReAT is built on top of an execution engine (GReAT-E) which can translate models based on transformation rules specified by GReAT. This mapping specification needs to be created only once for a given DSML and then any valid models for that DSML can be automatically transformed into a corresponding SMAL model. This is achieved with the code generator for GReAT models. Once the transformation rules have been defined the code generator will create a Visual Studio project that compiles to an executable file. This .exe file can be run from the Windows command line.

Modifying Embedded System DSMLs to Support SMAL

In order to define such a transformation, the original DSML must be able to capture those security properties that are need for SMAL to provide a useful analysis. In other words, for information flow analysis, the DSML must be able to model the concepts such as data object, dataflow, partition, security level, compartment, confidentiality and integrity requirements and for threat model analysis the DSML must be able to model the concepts such as encryption algorithm library, encryption algorithm, adversary model, susceptible encryption algorithm, encrypted channel and channel adversary. Typically, the DSML will not have all of the concepts needed to create such a transformation. For example, take a DSML built on the synchronous dataflow model of computation [31]. This DSML would have the concepts such as data objects and dataflow, but none of the other security specific concepts. It would be the responsibility of a tool designer to add the ability to capture these security specific concepts in a DSML. The process of extending a DSML to capture security related properties is not as difficult as it might seem. One of the powerful concepts of the MIC approach is easy composition of metamodels to form new languages. By composing the metamodel of a DSML with concepts from SMAL, it is relatively easy to form these security specific extensions to an existing language. The tool designer can then create the transformation rules that map models in the DSML onto models in SMAL.

Feedback Analysis Results

Since SMAL is only capable of capturing those concepts which are relevant to security analysis, it is not possible to define a transformation from SMAL back onto the original DSML. Those concepts which are unique to the original domain are lost in the translation from the DSML to SMAL. In order for SMAL to provide useful feedback to the user, it is necessary to have the *path* and *id* attributes which belong to partition, information flow, data object, ports, adversary model and encryption algorithm in SMAL. *Path* and *id* store the path and the unique id of an object in the original DSML. In GME, hyperlinks to objects are created using this unique id. For each error message from the SMAL model analysis, one or more hyperlinks point the specific object that caused the security violation. When a SMAL model is analyzed and security violations are identified and the results are output to an XML file. There is a separate interpreter that will read in the XML file and output the results to GME console. Figure 19 shows an example XML file that shows analysis results and Figure 20 shows the console error message corresponding to the XML file.

```
<?xml version="1.0" ?>
<root>
  <System name="SimpleSystem" ID="putGUID">
    <result type="Integrity">
      <data ID=“0062-00000002">/Flow Analysis/PartitionA/Data_A1</data>
      <src ID="0062-00000007">/Flow Analysis/PartitionB/Port_B3</src>
      <dst ID=" 0062-00000013”>/Flow Analysis/PartitionC/Port_C1</dst>
    </result>
  </System>
</root>
```

Figure 19 Example security analysis results XML file

```
Integrity Requirement Violated --
/SimpleSystem/PartitionB/Assembly_B1 has
an integrity requirement which is violated
by the information flow connecting
/SimpleSystem/PartitionB/Port_B2 to
/SimpleSystem/PartitionC/Port_C1.
```

Figure 20 Example console error message

This allows the results to be displayed to the console of the GME instance which has the open DSML model. The results will be fed back to the user of the original DSML in the form of an error messages along with hyperlinks that identify at which point in the original model there is a security violation. Using this approach, a user of a DSML will never have to view the SMAL model. To form this automated feedback process there are three interpreters that are involved: 1) DSML to SMAL Transformation, 2) SMAL analysis interpreter and 3) security results interpreter. The second and third interpreters have been developed for use with SMAL. When integrating SMAL with an existing DSML it is up to the tool developer to develop the transformation interpreter and link it to the other interpreters. The transformation interpreter should generate a SMAL model from a DSML model, invoke the analysis interpreter on the SMAL model and then invoke the security results interpreter that will read in the XML results file. By linking together these interpreters the use will only have to invoke one interpreter from the DSML environment and the rest of the process will be automated.

CHAPTER V

CASE STUDY: INTEGRATING SECURITY ANALYSIS TO AN EXISTING EMBEDDED SYSTEM LANGUAGE

As a proof of concept, we have integrated SMAL with an existing tool for the design of embedded systems, SMoLES [26]. SMoLES alone does not address security concerns. We show how to add the capability for capturing security specific properties to SMoLES. Models in SMoLES are enriched with these concepts and we created a mapping from SMoLES models to objects in SMAL. This mapping allows the creation of a transformation from SMoLES models to SMAL models. The following sections show how results from the analysis of a SMAL model can feed back warnings of security violations in terms of SMoLES models so that the user can correct them accordingly.

Integrating SMAL with SMoLES

Since SMoLES does not capture any security properties, we must add the appropriate concepts, so that we can create security aware models. We call this extended language SMoLES_SEC. SMoLES_SEC allows the modeler to capture the security properties required to perform the two types of analysis that SMAL supports, the information flow analysis and threat model analysis.

First, we address those concepts necessary to perform the information flow analysis. SMoLES already has the concept of dataflows but none of the other concepts used in SMAL. Assemblies in SMoLES are close to the concept of a partition in SMAL. Figure 21 shows the portion of the SMoLES metamodel that defines the Assembly.

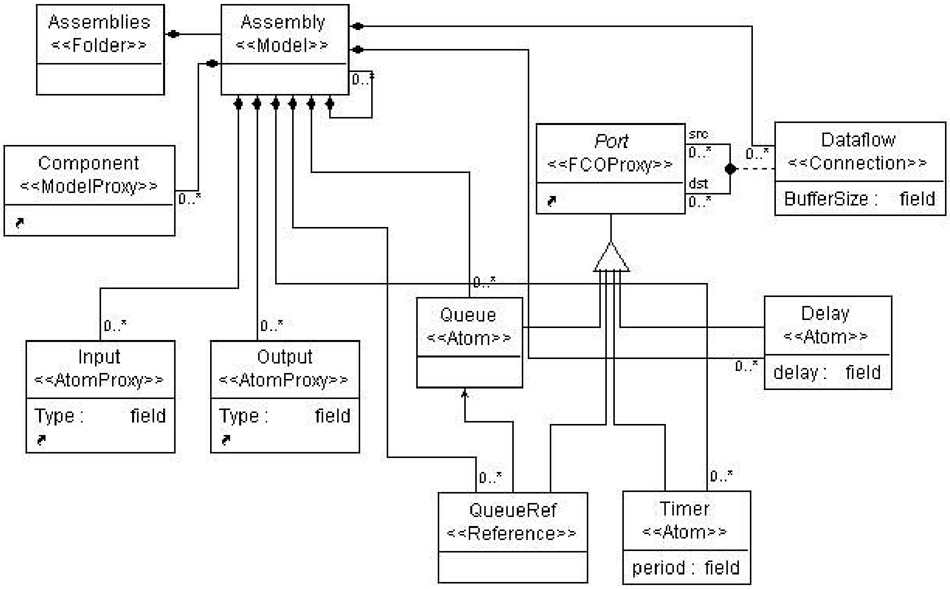


Figure 21 SMoLES Assembly metamodel

One possible approach to add security concepts to SMoLES would be to add the *security level* and *compartment* attributes to assemblies. However, assemblies are hierarchical whereas partitions in SMAL are not. In the root folder of a SMoLES_SEC model will be a Systems folder which can contain one or more System models. So, we introduce the idea of partition to SMoLES_SEC. Partitions will have input and output ports which can be connected by dataflow connections. Like in SMAL, partitions will have *security level* and *compartment* attributes that define their access rights in the context of the Bell-LaPadula and Biba models. A System model contains the SMoLES_SEC Partition models. Assemblies are contained by partitions and inherit the *security level* and *compartment* of the containing partition. SMoLES has the concept of dataflow; however it has no first class object that is a data object. Currently, we assign the *confidentiality* and *integrity* attributes to an assembly in SMoLES_SEC and evaluate these attributes against the dataflows originating from that assembly. Although a more useful and correct approach would be to assign *confidentiality* and *integrity* attributes to SMoLES components. Assemblies are modeling level concepts for usability of the modeling language. They are simply containers used to group components together and have no meaning in terms of generated code. The simply group Figure 22 shows the SMoLES_SEC metamodel with Partition, System and Systems as the security specific extensions to the SMOLES language.

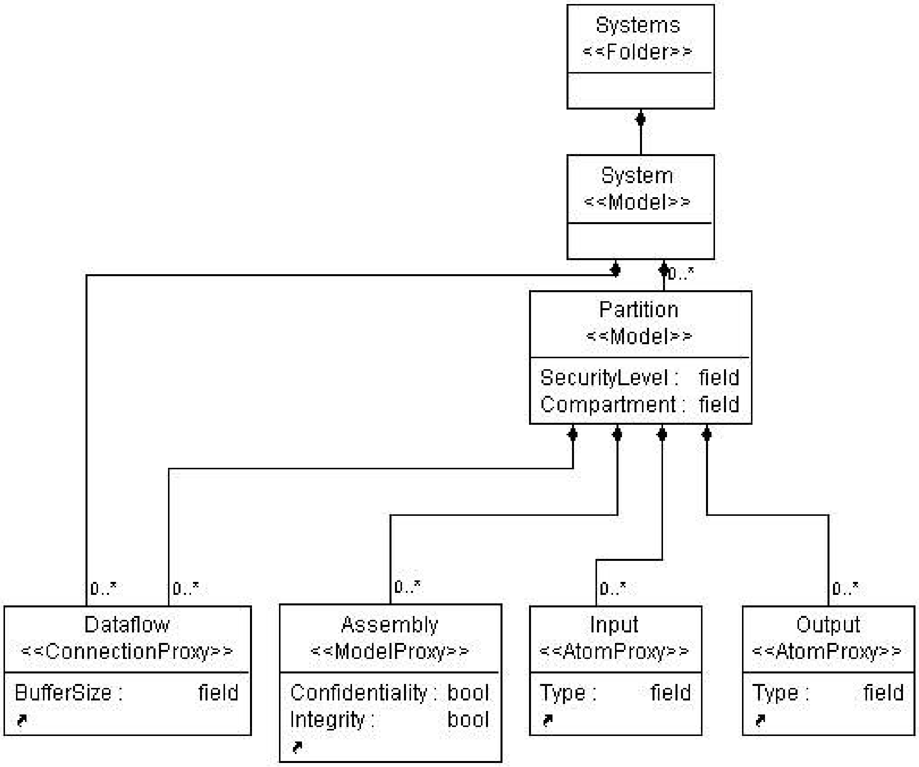


Figure 22 SMoLES_SEC Partition metamodel

Next, we address the concepts necessary to perform the threat model analysis. SMoLES has no concept of encryption algorithms or adversary modeling. We add these concepts to SMoLES_SEC and define them in the same way that they are defined for SMAL. The SMoLES_SEC threat modeling is similar to the threat modeling in SMAL.

In each system, there is an encryption algorithms library with a set of encryption algorithms. Each system contains a set of adversaries and each adversary contains a set of Susceptibility references. Susceptibility references points to an encryption algorithm which represents that the encryption algorithm is vulnerable to the adversary which contains the reference. Each encryption algorithm has an attribute, *MaxKeySize.* Data encrypted with a key size that is less than or equal to M*axKeySize* is vulnerable to attack. Figure 23 shows the metamodel for the SMoLES_SEC adversary.

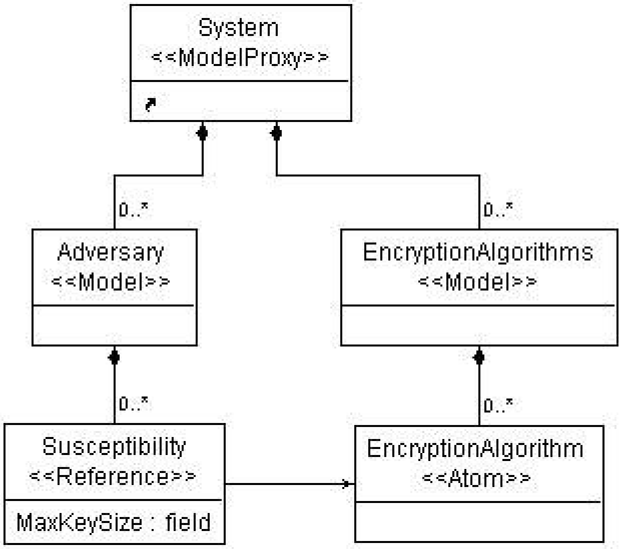


Figure 23 SMoLES_SEC adversary metamodel

We do not associate an encryption algorithm and adversary model directly with dataflows as it is done in SMAL. Rather, SMoLES_SEC can model a deployment diagram where nodes, which represent the execution platform, are connected to other nodes through a link (or bus). A node can be viewed as a set that contain partitions that execute on that node. Likewise, a link can be viewed as a set that contains the dataflows that are transmitted over that link. Each link in SMoLES_SEC has an attribute, *AnticipatedAdversary*, which identifies the adversary model associated with that link. Each link in SMoLES_SEC also has an *EncryptionAlgorithm* and *KeySize* attribute. Dataflows inherit the *AnticipatedAdversary*, *KeySize* and *EncryptionAlgorithm* of the link that they are transmitted across. Figure 24 shows the SMoLES_SEC deployment diagram metamodel.

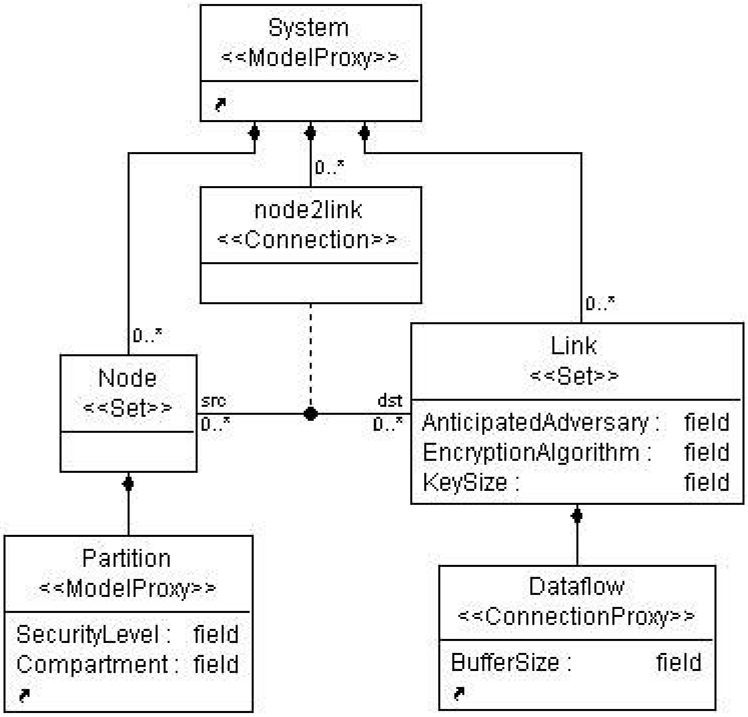


Figure 24 SMoLES_SEC deployment diagram metamodel

The effect that these extensions for SMoLES_SEC have on tools that were written for the SMoLES language must be examined. SMoLES has an interpreter that will generate C++ code from models. This interpreter is unaware of encryption algorithms and their meaning in the context of SMoLES_SEC. There are two solutions to this problem. Either we define a transformation that will map SMoLES_SEC models back onto SMoLES models or we port this code generator to work with the SMoLES_SEC environment. In our case we choose to port the code generator to the SMoLE_SEC environment. To do this we will need to make some slight modifications to the code generator, such as placing the code in separate partitions and linking to a library of encryption algorithms.

Model transformation from SMoLES_SEC to SMAL

Now that the appropriate concepts have been added to extend SMoLE_SEC, we are able to define a model transformation that maps SMoLES_SEC models to corresponding models in SMAL. Once we have defined these rules, the process of converting SMoLES_SEC models to SMAL models will be automated. The GReAT model transformation tool allows us to define these rules.

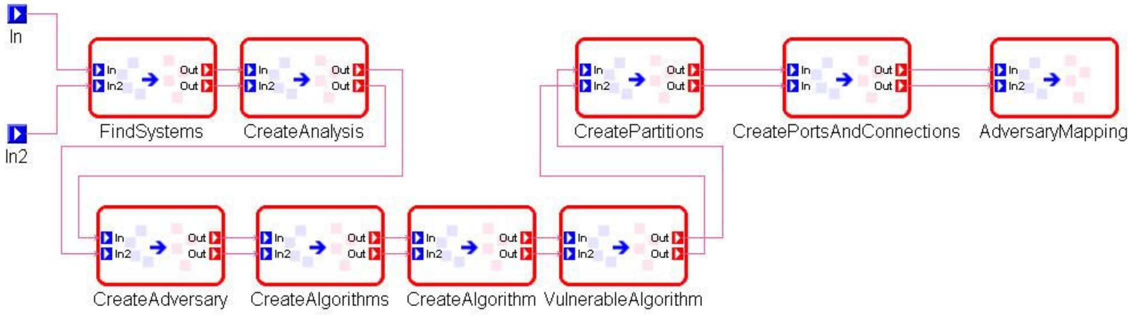


Figure 25 SMoLES_SEC to SMAL Transformation

Figure 25 shows the high level view of the transformation. These are nine rules that define how a SMoLES_SEC model maps into a SMAL model. The rules are as follows:

1) **FindSystems** – finds the Systems folders in the SMoLES_SEC root folder and passes the Systems folders out along with the RootFolder of the new SMAL model.

2) **CreateAnalysis** – creates a new SMAL SecurityAnalysis model for each SMoLES_SEC System model.

3) **CreateAdversary** – for every SMoLES_SEC Adversary a corresponding SMAL Adversary model is created.

4) **CreateAlgorithms** – for every SMoLES_SEC EncryptionAlgorithms model a corresponding SMAL EncryptionAlgorithms model is created.

5) **CreateAlgorithm** – for every SMoLE_SEC EncryptionAlgorithm a corresponding SMAL EncryptionAlgorithm is created in the EncryptionAlgorithms model created in step 4.

6) **VulnerableAlgorithm** – for every reference to SMoLES_SEC EncryptionAlgorithm in the SMoLES_SEC Adversary a corresponding reference to a SMAL EncryptionAlgorithm in a SMAL Adversary is created.

7) **CreatePartitions** – for every SMoLES_SEC Partition model a corresponding SMAL Partition model is created with the same SecurityLevel and Compartment Attributes.

8) **CreatePortsAndConnections** – for every SMoLES_SEC Dataflow that connects Ports on a Partition corresponding SMAL InformationFlow and SMAL Ports are created.

9) **AdversaryMapping –** the value of the AdversaryModel attribute for every SMoLES_SEC Link is passes to the SMAL InformationFlows that correspond to a SMoLES_SEC Dataflow contained by that Link.

Feedback Analysis Results to SMoLES_SEC

In the previous sections, the security extensions to the SMoLES language and the model transformation from the resulting SMoLES_SEC to SMAL were described. The next step to integrate security analysis with the SMoLES language is to automate the process of providing analysis results to the modeler. This feedback automation was discussed in Chapter III. There are three interpreters that are involved: 1) DSML to SMAL Transformation, 2) SMAL analysis interpreter and 3) security results interpreter. The second and third interpreters have been developed for use with SMAL. When integrating SMAL with an existing DSML it is up to the tool developer to develop the transformation interpreter and link it to the other interpreters.

Figure 26 shows how the design flow presented in Figure 18 is used to integrate the security concepts in SMAL with the SMoLES language.

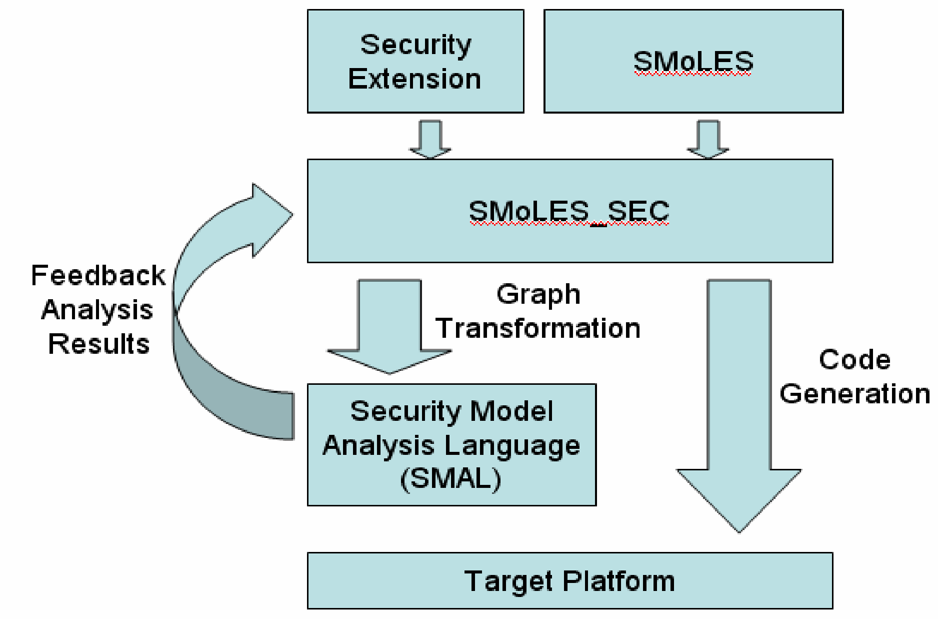


Figure 26 Design Flow: Integrating SMAL with SMoLES

This allows the results to be displayed to the console window of the GME instance which has the open DSML model. The results will be fed back to the user of the original DSML in the form of an error messages along with hyperlinks that identify at which point in the original model there is a security violation. Using this approach, a user of a DSML will never have to view the SMAL model. The transformation interpreter should generate a SMAL model from a DSML model, invoke the analysis interpreter on the SMAL model and then invoke the security results interpreter that will read in the XML results file. By linking together these interpreters the user will only have to invoke one interpreter from the DSML environment and the rest of the process will be automated.

We have written a small script that can be invoked from the SMoLES_SEC environment. This allows the user of the SMoLES_SEC environment to transform their model into a SMAL model in one step, run the information flow and threat model analysis on the SMAL model and receive the analysis results. The user will never view the resulting SMAL model. The analysis on the SMAL model is done in the background.

Example application in SMoLES_SEC

An example SMoLES_SEC model is shown in Figure 27. This is a generic application that demonstrates the capabilities of the security analysis. Real applications of this tool will be too large to cover in the scope of this thesis. There are four partitions in the system. Partitions A, B, and D have a *security level* of 1 and PartitionC has a *security level* of 2. This means that data objects with a *confidentiality* requirement may not flow from PartitionC and those data object with an *integrity* requirement may not flow to PartitionC. In this example, we do not consider partitions with different *compartment* classifications. PartitionB contains an Assembly_B1 that has an *integrity* requirement but no *confidentiality* requirement. Since this assembly is in PartitionB it inherits the security level of 1. Figure 28 shows the deployment diagram. Nodes 1, 2, and 3 are connected by a common link. Components in PartitionA and PartitionB execute on Node1. Components in PartitionC execute on Node2 and Components in PartitionD execute on Node3. All dataflows transfer data across the Link, except the dataflow connecting PartitionA and PartitionB which reside on the same node. Figure 29 shows the threat model of this system. The AnticipatedAdversary model of Link is the Internet_Adversary.

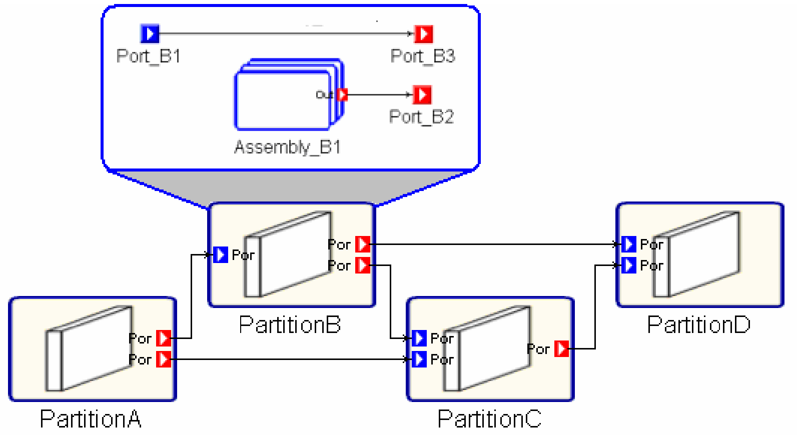


Figure 27 SMoLES_SEC example application: Partitions and dataflows

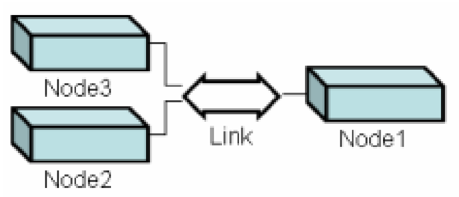


Figure 28 SMoLES_SEC example application: deployment diagram

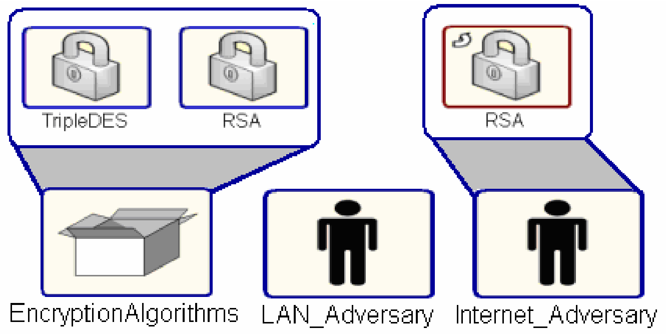
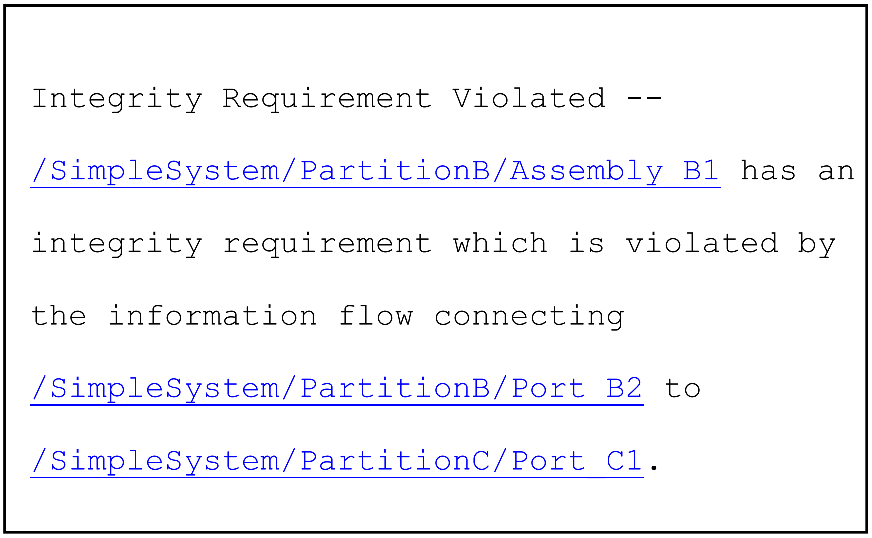


Figure 29 SMoLES_SEC example application: threat model

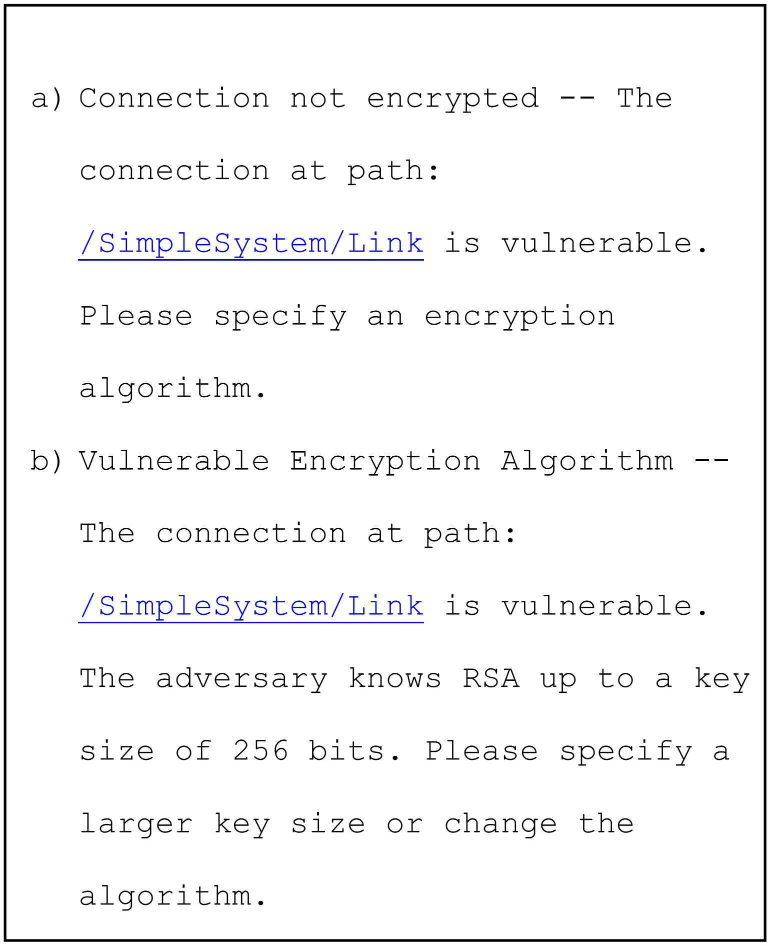
First, we will invoke the information flow analysis on this model. Figure 30 shows the error message that we receive for the information flow analysis. The assembly in PartitionB has an *integrity* requirement, so the dataflows originating from this assembly are evaluated with the Biba model. There is a dataflow that connects PartitionB to PartitionC which represents data objects moving from a low security level to a high security level. This dataflow violates the Biba model. There are a several possible solutions to this error. It is up to the system modeler to determine which solution is appropriate in the context of their system. For this example, we determine that the PartitionB can be classified at a *security level* of 2. When this change is made to the model the security analysis tool does not return any errors.



```
Integrity Requirement Violated --
/SimpleSystem/PartitionB/Assembly B1 has an
integrity requirement which is violated by
the information flow connecting
/SimpleSystem/PartitionB/Port B2 to
/SimpleSystem/PartitionC/Port C1.
```

Figure 30 Error message - information flow analysis

Next, we invoke the threat model analysis. Figure 31a shows the error message we receive. There is an adversary associated with the link, so the channel must be encrypted. The error message warns that Link must be encrypted, so we set the encryption attributes on Link to 256 bit RSA. Internet_Adversary is capable of breaking RSA with a key size of 256 bits or less. The error message in Figure 31b shows the error message we receive. To fix this error message we increase the key size used to encrypt Link to 512 bits. This fixes the security violation the threat model analysis no longer returns any error messages.



```
a) Connection not encrypted -- The
   connection at path:
   /SimpleSystem/Link is vulnerable.
   Please specify an encryption
   algorithm.
b) Vulnerable Encryption Algorithm --
   The connection at path:
   /SimpleSystem/Link is vulnerable.
   The adversary knows RSA up to a key
   size of 256 bits. Please specify a
   larger key size or change the
   algorithm.
```

Figure 31 Error messages - threat model analysis

CHAPTER VI

EVALUATION OF SECURITY ANALYSIS TOOLCHAIN

There have been two methods for identifying vulnerabilities in a SMAL model presented. The first method was analyzing the SMAL model with OCL constraints. The second method analyzed the SMAL model with a C++ interpreter. To compare the performance of both methods, a SMAL model was analyzed with them on a laptop with a 1.73 GHz Pentium M and 512 MB RAM. The model had 320 partitions, 128 data objects, 1214 data flows and 1024 ports. This model is large enough that it is not feasible to analyze the security properties manually yet small enough that the analysis will finish in a reasonable amount of time. Analyzing the model with the OCL constraints took 60 seconds. The BON interpreter took 85 seconds to analyze and write the results to the XML file. It took 92 seconds to read in the XML file and write the results to the GME console. The BON interpreter took 25 seconds longer to analyze the model than the OCL constraints. However, the C++ interpreter identified 4288 vulnerabilities compared to the OCL constraints which only returned 308 vulnerabilities. This difference in number of results is due to the fact that the OCL constraints only return one of each type of result per data object. The BON interpreter will return a result for each vulnerability.

CHAPTER VII

FUTURE WORK AND CONCLUSION

SMAL currently supports modeling of access control policies in the context of the Bell-LaPadula and Biba models. These access control policies are not sufficient for the needs of all applications. We would like for SMAL to have the expressiveness to model other types of access control schemes. Another area that needs to be addressed is how to more tightly integrate the security analysis with the other analysis tools available for a DSML. This would allow the designer to look at tradeoffs made based on security properties such as analyzing the tradeoffs between security and performance. We have shown how SMAL can be integrated with SMoLES which is a dataflow based language. This leads to a simple mapping from SMoLES to SMAL. There needs to be work done to look at how security analysis can be integrated with other classes of DSML such as those based on control flow.

Model driven security approaches have been successfully used in various industrial, governmental and financial applications. Model-Integrated Computing has proven to be a valuable tool in embedded systems design process. We have demonstrated a security analysis tool that is capable of analyzing the flow of data objects through a system and identifying points in a distributed system that are vulnerable to attack. We have outlined a method for composing this type of security tool with existing tool chains for DSMLs. This approach leverages the development efforts that have gone into design of tool suites for existing embedded system DSMLs. Creating a separate analysis language for security properties allows reuse of this tool for multiple DSMLs. The example application shown is a proof of concept that demonstrates the potential of integrating security modeling capabilities with existing languages.

APPENDIX A

The formalization of SMAL information flow and adversary analysis as OCL constraints is presented in this appendix. First the constraints related to information flow analysis (Bell-LaPadula and Biba) are given. Then the constraints related to adversary analysis are given.

```
context meta::Data inv Biba:
self.Integrity implies not self.BibaViolation()

context gme::Atom::BibaViolation: ocl::Boolean defmethod BibaViolation:
self.connectedFCOs("dst",meta::InformationFlow)->iterate( port:
meta::Port; violation = false |
     if(self.parent().oclAsType(meta::Partition).SecurityLevel <
port.parent().oclAsType(meta::Partition).SecurityLevel) then
          true
     else
          violation || port.BibaViolation()
     endif
)

context meta::Data inv BLP:
self.Confidentiality implies not self.BLPviolation()

context gme::Atom::BLPviolation: ocl::Boolean defmethod BLPviolation:
self.connectedFCOs("dst",meta::InformationFlow)->iterate( port:
meta::Port; violation = false |
     if(self.parent().oclAsType(meta::Partition).SecurityLevel >
port.parent().oclAsType(meta::Partition).SecurityLevel) then
          true
     else
          violation || port.BLPviolation()
     endif
)

context meta::Data inv Compartment:
self.Confidentiality || self.Integrity implies not
self.CompartmentViolation()

context gme::Atom::CompartmentViolation: ocl::Boolean defmethod
CompartmentViolation:
self.connectedFCOs("dst",meta::InformationFlow)->iterate( port:
meta::Port; violation = false |
     if(self.parent().oclAsType(meta::Partition).Compartment <>
port.parent().oclAsType(meta::Partition).Compartment) then
          true
     else
          violation || port.CompartmentViolation()
     endif
)
```

```
context meta::Data inv Adversary:
self.Confidentiality || self.Integrity implies
not self.SusceptibleAlgorithm()

context gme::Atom::SusceptibleAlgorithm: ocl::Boolean defmethod
Susceptible Algorithm:
if
(self.attachingConnections("src",meta::InformationFlow)->iterate(conn:
meta::InformationFlow; violation = false |
     if( conn.AnticipatedAdversary <> "") then
            let MKS =
conn.getMKS(conn.connectionPoint("src").target().oclAsType(gme::Atom))
in
            let KS = conn.KeySize in
            MKS >= KS
     else
          violation
     endif
)) then
          true
else
          self.connectedFCOs("dst").exists(t |
t.oclAsType(gme::Atom).SusceptibleAlgorithm())
endif

context::metaInformationFlow::getMKS(source: gme::Atom): ocl::Integer
defmethod getMKS:
source.parent().parent().oclAsType(meta::System).modelParts(meta::Adver
sary)->iterate(a;
MKS = 0 |
     if (a.name = self.AnticipatedAdversary) then
          a.referenceParts(meta::Susceptibility).any(a | a.name =
self.EncryptionAlgorithm).oclAsType(meta::Susceptibility).MaxKeySize
     else
          MKS
     endif
)
```

REFERENCES

[1] Presidential Decision Directive 63, May 22, 1998

[2] The National Strategy for the Physical Protection of Critical Infrastructures and Key Assets, February 2003.

[3] Fernandez, J. D. and Fernandez, A. E. 2005. *SCADA systems: vulnerabilities and remediation*. In *Journal of. Computing Sciences in Colleges.* 20, 4 (Apr. 2005), 160-168.

[4] Amin, M. North America's electricity infrastructure: are we ready for more perfect storms? Security & Privacy Magazine, IEEE Volume 1, Issue 5, Sept.-Oct. 2003 Page(s):19 - 25

[5] Mercuri, R., Neumann, P. 2003. Security by Obscurity. In *Communications of the ACM*. (Volume 46, Issue 11, November 2003) 160.

[6] Kocher, P., Lee, R., McGraw, G., and Raghunathan, A. 2004. Security as a new dimension in embedded system design. In *Proceedings of the 41st Annual Conference on Design Automation* (San Diego, CA, USA, June 07 - 11, 2004). DAC '04. ACM Press, New York, NY, 753-760.

[7] Ravi, S., Raghunathan, A., Kocher, P., and Hattangady, S. 2004. Security in embedded systems: Design challenges. *Trans. on Embedded Computing Sys.* 3, 3 (Aug. 2004), 461-491.

[8] F. Schneider, editor. *Trust in Cyberspace.* National Academy Press, Washington, DC, 1999.

[9] Andrew Hildick-Smith, *Security for Critical Infrastructure SCADA systems.* August 24 2005. SANS Institute.

[10] John Rushby. The design and verification of secure systems. In Eight ACM Symposium on Operating System Principles, pages 12-21, Asilomar, CA, December 1981.

[11] *Unified Modeling Language 2.1.1* Specification. Object Management Group.

[12] Weaver, N., Paxson, V., Staniford, S., and Cunningham, R. 2003. *A taxonomy of computer worms*. In *Proceedings of the 2003 ACM Workshop on Rapid Malcode* (Washington, DC, USA, October 27 - 27, 2003). WORM '03. ACM Press, New York, NY, 11-18

[13] Sztipanovits, J.; Karsai, G. *Model-integrated computing,* Computer Volume 30, Issue 4, April 1997 Page(s):110 – 111

[14] Karsai, G., Sztipanovits, J., Ledeczi, A., Bapty, T.: *"Model-Integrated Development of Embedded Software,"* Proceedings of the IEEE, Vol. 91, No.1., pp. 145-164, January, 2003

[15] Generic Modeling Environment Users Manual

[16] Ledeczi A., Bakay A., Maroti M., Volgyesi P., Nordstrom G., Sprinkle J., Karsai G.: *Composing Domain-Specific Design Environments*, Computer, pp. 44-51, November, 2001.

[17] Tsipenyuk, K.; Chess, B.; McGraw, G.; *Seven pernicious kingdoms: a taxonomy of software security errors* Security & Privacy Magazine, IEEE Volume 3, Issue 6, Nov.-Dec. 2005 Page(s):81 - 84

[18] Microsoft TechNet. Information About Virus-Infected Hotfixes. April 25 2001.

[19] Microsoft TechNet. Microsoft Security Bulletin MS06-007. February 14 2006.

[20] Kevin Poulsen, *Slammer worm crashed Ohio nuke plant network,* August 19 2003.

[21] Jürjens, J. 2005. *Sound methods and effective tools for model-based security engineering with UML*. In *Proceedings of the 27th international Conference on Software Engineering* (St. Louis, MO, USA, May 15 - 21, 2005). ICSE '05. ACM Press, New York, NY, 322-331.

[22] Basin, D., Doser, J., and Lodderstedt, T. 2003. *Model driven security for process-oriented systems*. In *Proceedings of the Eighth ACM Symposium on Access Control Models and Technologies* (Como, Italy, June 02 - 03, 2003). SACMAT '03. ACM Press, New York, NY, 100-109.

[23] Jan Jürjens, *Secure Systems Development with UML*, Springer-Verlag, 2004

[24] Available from the Authors

[25] *Architecture Analysis & Design Language*. SAE Standard AS-5506

[26] Szemethy, T. and Karsai, G. 2004. Platform modeling and model transformations for analysis. *Journal of Universal Computer Science 10*, 10, 1383–1406.

[27] NCSC (1985). "*Trusted Computer System Evaluation Criteria*". National Computer Security Center.

[28] D.E. Bell and L.J. LaPadula. “Secure Computer Systems: Mathematical Foundations and Model,” Mitre Corp. Report No. M74-244, Bedford, Mass., 1975.

[29] K.J. Biba, “Integrity Considerations for Secure Computer Systems,” Mitre Corp. Report TR-3153. Bedford. Mass., 1977.

[30] “Common Criteria for Information Technology Security Evaluation Users Guide”, October 1999.

[31] E. A. Lee and D. G. Messerschmitt, “Synchronous data flow,” *Proc. IEEE*, vol. 75, pp. 1235–1245, Sept. 1987.

[32] Agrawal, A., Karsai, G., Ledeczi, A.: An End-to-End Domain-Driven Software Development Framework. Conference on Object Oriented Programming Systems Languages and Applications, 2003.