Enforcement of Quality Attributes for Net-Centric Systems through Modeling and Validation with Architecture Description Languages

Jörgen Hansson, Peter Feiler, Aaron Greenhouse

Enforcement of Quality Attributes for Net-Centric Systems through Modeling and Validation with Architecture Description Languages

Jorgen Hansson, Peter H. Feiler, and Aaron Greenhouse

Software Engineering Institute, Carnegie Mellon University
Pittsburgh, PA, USA
{hansson,phf}@sei.cmu.edu

Abstract: In this paper we discuss and demonstrate how to conduct validation of data quality attributes, e.g., security, data accuracy, data confidence, and temporal correctness, can be modeled and validated using an architecture description language such as AADL. We focus on security, specifically confidentiality.

Keywords: Security, Modeling, Validation, AADL, Software Architecture

1. Introduction

Net-centric system are intrinsically distributed system-of-systems comprised of a number of interconnected heterogeneous, geographically dispersed systems, where several systems are embedded and operate under real-time constraints. Sensor networks are increasingly being deployed in applications to monitor, collect, process, and communicate data obtained from the environment, and thus they provide data services to high-level applications. Data quality in a net-centric system can be compromised in a number of ways: confidentiality might be breached due to lack of adequate encryption or incorrectly assigned security clearance for subjects operating on object; confidence level of data might be insufficient in part due to the set size of active sensors or environmental conditions; the accuracy of sensor readings is also affected by environmental conditions; correctness of data is low as it is a function of its temporal coherence. Enforcing 100% accuracy, correctness, and confidence is in practice not always possible nor desirable as the cost becomes too significant with respect to cost, computational power, power consumption, quality of sensor etc. It is more desirable to ensure that data quality is within acceptable tolerance levels of the applications, not jeopardizing their correctness.

Model-driven engineering based on an architectural model is of paramount importance to validating the quality of data, and thereby ensuring that high-level-systems are provided with data of sufficient quality and consistency driven by application requirements. Architectural description languages have successfully been applied to prove system properties, often with a propensity toward task- or component-centric perspectives and, thus, generally preclusive of data quality attributes. We have developed a modelling framework using AADL for validating the enforcement of the previously mentioned data quality attributes prior to the implementation phases of the system. In this paper we present our developed methods for validating quality attributes, specifically security, using AADL. This includes The paper includes examples from modelling and validation of confidentiality under Bell-LaPadula-based frameworks [1,2]. The approach supports other security frameworks, including Chinese Wall [3,4], role-based access control [5], and information flow, [6-8].

2. Validation of Confidentiality

The concept of subjects and objects, where subjects operate on objects by permissible access operations (read, execute, append, write) enables us to model and validate security at both the software and hardware levels. At the software level we can view processes, threads, and software components as subjects and data objects are objects.

Determining the viability of a system given confidentiality requirements of data objects and security clearance by users, one can see the validation as a two-step process: (1) validation of the software architecture followed by (2) validation of the system architecture where the software architecture is mapped to hardware components. Validating the software requires us to

- identify the data elements that we want to protect (objects);
- determine their security requirements;
- identify the components (software components, processes, threads) that should be allowed to access the objects; and
- confirm the access is as specified by access operations.

Thus, we can ensure that data elements are accessed only by authorized users and that confidentiality (as given by security levels) and

integrity (as given by access operations) are enforced.

By mapping the entities of a software architecture (e.g., processes, threads, and partitions) to a hardware architecture (consisting of, for example, CPUs, communication channels, and memory) and the like enables us further to ensure that the hardware architecture supports required security levels. Consider the scenario of two communicating processes, both requiring a high level of security as because the data objects requires secret clearance. Furthermore, the system platform in this scenario consists of a set of CPUs with hardware support for various algorithms that encrypt messages before network transmission. By modeling the system, we can represent and validate that processes and threads (now considered to be objects) can be executed (access mode) on CPUs (subjects) with adequate encryption support. Furthermore, we can validate that CPUs (objects) communicate data (access modes of writing and reading) over appropriately secured communication channels. In a similar fashion, we can enforce design philosophies saying that only processes of the same security level are allowed to co-exist within the same CPU or partition or that they can write to a secured memory.

In this section, we identify and outline desirable criteria to enforce when modeling and validating security using the model-based engineering approach. Our purpose in validating security is to ensure that

- all components only access data they are classified for (through security level and compartments/categories)
- data is only used on a need-to-know basis
- modeling and validating security contributes to identifying identify possible errors
- sanitization (i.e., lowering of security levels) of data is conducted controllably

To model and validate the confidentiality of a system, we distinguish between general and application-dependent validation. General validation of confidentiality is the process of ensuring that a modeled system conforms to a set of common recommendations or design guidelines, expressed as a set of conditions that support system confidentiality independent of a specific reasoning framework for security. Those conditions should hold in the general case; as a result, they are necessary but not sufficient (i.e., satisfying the conditions indicates the system is viable for enforcing confidentiality). General validation of confidentiality assumes that subjects and objects are assigned a security level, such as (cl, c) that is the minimum representation to enforce what are commonly referred to as the basic confidentiality and need-to-know principles.

Application-specific validation refers to validating the system given detailed confidentiality requirements and a specific reasoning-based security framework. For example, Bell-LaPadula-based models represent permitted access patterns between objects and subjects, against which operations are checked before they are allowed to be performed.

These recommendations and design guidelines are conditions that verify an architecture is feasible (i.e., that the modeled architecture does not compromise security and confidentiality). Before elaborating on the conditions, we introduce a graphical notation (see Figure 1) and an example.

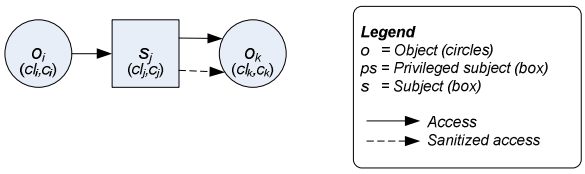

Figure 1: Graphical Notation of $o_i \rightarrow s_j \rightarrow o_k$

Figure 1 shows the general graphical notation we use for subjects and objects. The picture shows a subject s_j accessing the objects o_i and o_k. s_j has security level (cl_j, c_j) clearance, and the objects require that subjects operating on them meet the security classification requirements expressed as (cl_i, c_i) and (cl_k, c_k). The relation $o_i \rightarrow s_j$ implies that o_i is accessed but not modified by s_j (i.e., corresponding to the access operations read and execute). The relation $s_j \rightarrow o_k$ shows that o_k is modified by s_j, corresponding to the access operations append and write. We say that s_j is using o_i as input and produces o_k as output. We represent sanitized access—(cl_k, c_k) does not dominate (cl_i, c_i)—by a dashed arrow, and sanitization is only allowed to be performed on modified objects.

Figure 2 depicts an example consisting of a number of scenarios, potential faults, and errors that we refer to when discussing the required and recommended conditions for performing general validation. For the purpose of simplicity and without loss of generality, we assume that the objects are data objects read $(o_i \rightarrow s_j)$ or written $(s_j \rightarrow o_k)$ by subjects; this notation simplifies the representation, since we know the allowed access operations from the example.

The example in Figure 2 contains a feedback loop ($o_1 \rightarrow s_1 \rightarrow o_1$), subjects reading from and writing to multiple objects (e.g., s_1, s_2), and sanitization ($s_2 \rightarrow o_7$ and $s_2 \rightarrow ps_1 \rightarrow o_7$).

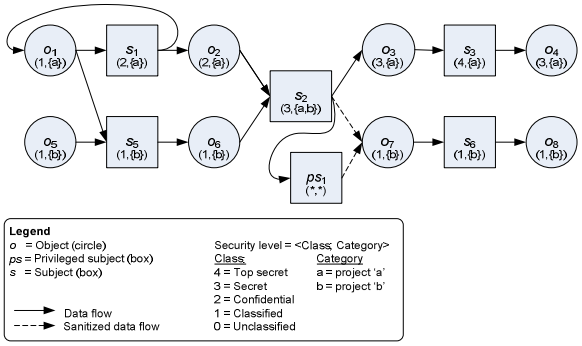

Figure 2: Example with Object-Subject Dependencies

3. Validating Subjects and Objects

Next we outline a number of conditions for validation.

Condition 1 (Basic confidentiality principle):

The basic confidentiality principle states that one should only be granted access if one has the appropriate security clearance, For example, if a document is classified as confidential, the viewer of the document should have confidential clearance or higher (i.e., secret or top-secret). We say that the basic confidentiality principle is enforced if, and only if, for all subjects and objects, the class (cl) of a subject equals or exceeds the class of an object on which it operates. In the example shown in Figure 2, this principle is true for all subjects. Thus, in order to validate the basic security principle, we only need to specify the security class cl for subjects and objects.

Condition 2 (Need-to-know principle):

The need-to-know principle states that one should only be granted access to a resource if there is a need. For example, a person having top-security document clearance should not necessarily be allowed access to all documents but only to those related to his or her function in a project. In our notation, the need is represented as a category c. We say that the principle is enforced if, and only if, the security level (cl,c) of each subject in the system dominates the level of the objects on which the subject operates. In the graphical notation given in Figure 2, adherence to this principle implies that the following relations must hold: (cl_j, c_j) should dominate (cl_i, c_i) and (cl_k, c_k). In Figure 2, we can see that s_5 does not dominate o_1 since s_5 does have membership in project {b}; thus, the example shows an incorrect access. Also, while s_5 dominates o_3 and o_4, it actually has a higher security clearance than necessary for its purpose. We elaborate on this when discussing the principle of least privilege.

Condition 3 (Security level range checking):

The security level assigned to an object should be within the specified security levels of the object and the capacity of the subject. Some systems require that the security measures increase (e.g., when the systems appears to be under a security attack) in which case stronger encryption algorithms are deployed. Those algorithms require more computational power; in addition, the execution times of tasks processing data increase, and message sizes increase and require more network bandwidth. This increased resource demand adds to the end-to-end latency as well.

For applications where power consumption is a major concern (e.g., sensor networks), it is important to maximize the lifetime of the network, which suggests that level of security is traded for operational lifetime. Furthermore, in overloaded systems part of the strategy to resolve an overload is to temporarily enforce the minimum acceptable security, decreasing the workload imposed on the system. Thus, adapting the security policy and the security levels to adequately match the current state and risk levels seems feasible.

Indeed, one can adapt the security policy statically and dynamically. Assigning a security level to each subject based on the security requirements of the accessed objects, as we have previously discussed, is static as it implicitly states how a subject should operate to enforce the security given by an object (e.g., what encryption should be used). By assigning an allowed range of security levels for each object, we are explicitly expressing

the minimum security requirements acceptable

normal security level given normal operation

maximum security level desired for certain situations or system states

We model changing security levels in the following way. Each object o_i can be given a range of security levels, expressed as $[(cl_i^{\min}, c_i), (cl_i^{\max}, c_i)]$. We refer to the security level desired in the normal system state as (cl_i^{normal}, c_i).[1] Recall, going back to a subject's security clearance, that a subject's security clearance represents its maximum security privilege, which also implies that it can manage and operate below this security classification. When

[1] Note that only the level *l* changes; the category remains the same. Nothing prevents the categories from changing with the state, but we have not found an example where there is a practical need for it to do so.

checking the ranges of objects, a subject s_j needs to dominate the specified security classification range of each accessed object o_i— that is, (cl_j, c_j) should dominate $(cl_i^{\max}, c_i)$. Not dominating the range implies there are states when the subject is not authorized to access required object(s).

The desired security level of an object, within its range, normally reflects the current system state. To facilitate adequate and meaningful checking when objects have security level ranges, one should connect each level with a system state to enable analysis for the different states. (In AADL, this connection can be accomplished by modes.)

Condition 4 (Principle of confidentiality constancy):

The principle of confidential constancy means that the security level of an object produced (access is write or append) by a subject as output should dominate the security level of all objects used as input (access is read or execute).

The condition enforces the philosophy that the confidentiality of objects is maintained or increased. For example, a derived data object should be at least as confidential as the most confidential input data object since further analysis has been performed and more intelligence added; thus, its security level should dominate the security level of all input data objects. In the example given in Figure 1, this principle implies that the following relations should hold: (cl_k, c_k) should dominate (cl_i, c_i).

Condition 5 (Controlled sanitization):

Controlled sanitization stipulates that lowering the security level of an object or subject should only be authorized and performed by a privileged subject.

In Figure 2, Condition 5 is enforced in the following flows: $o_1 \rightarrow s_1 \rightarrow o_1$,

$o_1 \rightarrow s_1 \rightarrow o_2 \rightarrow s_2 \rightarrow o_3 \rightarrow s_3 \rightarrow o_4$,

$o_5 \rightarrow s_5 \rightarrow o_6 \rightarrow s_2 \rightarrow o_3 \rightarrow s_3 \rightarrow o_4$, and

$o_7 \rightarrow s_6 \rightarrow o_8$. Subject s_1 takes multiple inputs and produces multiple outputs; the security levels of o_3 and o_7 should at least be 2 given that o_2 was used as input and belong to project {a} or {b}. If this is indeed the desired security level of o_7, we lower the security level by letting a trusted subject authorize sanitization. This state can be viewed as having s_2 writing o_7 over a sanitized port or channel; alternatively, a dedicated subject for sanitization can be invoked. In our example, ps_1 is the privileged subject for performing sanitization (*, *), meaning that it can sanitize objects across all security levels and categories.

Condition 6 (Non-Alteration of Object's Security Requirements):

A subject using an object as input should not alter the security level of the object, even if the object is updated as an output from the subject. The rationale for this condition is that a subject can have a security clearance that exceeds the maximum required security level of an object. Increasing the security level of the object beyond its range implies that security requirements do not align between the subjects that operate on a dependent object. Thus, a subject with less security clearance than an object cannot continue its operation as expected. In Figure 2, consider the flow $o_1 \rightarrow s_1 \rightarrow o_1$. If s_1 is increasing the security level of o_1 to (2,{a}), s_5 is no longer allowed to read o_1, which probably was not intended in the general case.

Condition 7 (Hierarchical)

The hierarchical condition ensures that (i) a component has a security level that is the maximum of the security levels of its subcomponents, and (ii) all connections are checked to determine whether the source component of a connection declaration has a security level that is the same or lower than that of the destination component.

Consider the example given in Figure 3, which is an extract from the one shown in Figure 2 on 3. Hierarchical decomposition supports incremental modeling. In the high-level model where it is a component, s_0 takes o_1 and o_5 as inputs and generates o_5 and o_6 as outputs. Security requirements of the objects are not compromised, because the security level of s_0 dominates their requirements. Developing the refined model, the subcomponents s_1 and s_5 and their relation to the objects are modeled. Similarly, we need to ensure that the s_1 dominates o_1 and o_2, and s_5 dominates o_1, o_5, and o_6. Furthermore, s_0 should

dominate s_1 and s_5 to ensure that they act within the privileges of s_0.

If the security requirements of o_1 would be (2,{a}), Condition 4 is violated. Condition 4 concerns keeping confidentiality at the minimum level derived from maximum security requirements of the inputs. Thus, $o_1 \rightarrow o_6$ seems to be sanitized uncontrollably. In the refined model, we see that o_1 is used as input to derive o_6; thus it is indeed uncontrollably sanitized (as opposed to the circumstance in which only o_5 is used as input to o_6, where security is not compromised).

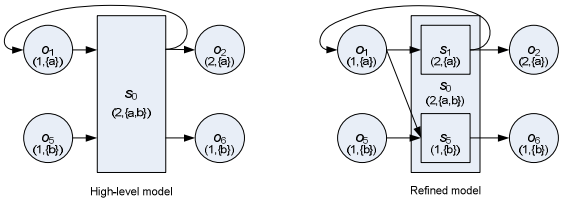

Figure 3: Hierarchical Modeling

Condition 8 (Principle of least privilege)

The principle of least privilege has been identified as important for meeting integrity objectives [10]; it requires that a user (subject) be given no more privilege than necessary to perform a job. This principle includes identifying what the subject's job requires and restricting the subject's ability by granting the minimum set of privileges required. With the object's security requirements specified in an AADL model, the least amount of privileges for the subjects can be generated in a straightforward manner by analyzing the security levels of all objects accessed by the subject. For a set of objects $\mathrm{O}=\{o_1, \ldots, o_n\}$ accessed by s_i, the least privileges (cl_i, c_i) of s_i are given by $\left(cl_i = \max_{l_j \in \mathrm{O}}(cl_j), c_i = \bigcup_{l_j \in \mathrm{O}} c_j \right)$, the maximum security level clearance of accessed objects and required membership in the categories.
Given that the subjects' privileges are specified, a mismatch between the least privilege and what has been specified results in two possible cases:

- The assigned privilege is insufficient. It does not dominate the required least privilege (i.e., the subject has been given incorrect privileges, the object has been wrongly associated with a subject, or there is an unauthorized access).
- The assigned privilege exceeds the minimum privilege, which either is unnecessary or a consequence of that the subject might be associated with other objects that have not yet been described in the model. In Figure 2, privileges of subject s_5 exceed the least privileges necessary given the current model.

The least privileges of subjects represent a snapshot, since it is a function of currently existing objects in the model. This view may change over time as a model is incrementally refined (e.g., as additional objects and subjects are modeled and more detail added).

4. Modeling and Representation in AADL

In the following, we describe our approach to mapping the concepts from the Bell-LaPadula security model to AADL

4.1 SECURITY LEVELS IN AADL

The security level of a subject/object is usually a pair of a security class value and a set of categories. The classification value is drawn from a partially ordered set and denotes how securely an object must be handled and how privileged a subject is. Categories further refine the security level by labelling objects and giving subjects permission to access appropriately labelled objects. This style of security level captures the typical governmental model wherein data is labelled as unclassified, confidential, secret, or top secret and categories are used to further restrict access. Represented this way, a security level $(class_1, categories_1)$ is said to *dominate* another security level $(class_2, categories_2)$ iff $class_1 \geq class_2$ and $categories_1 \supseteq categories_2$.

AADL properties associate security levels with the AADL components and features that represent subjects and objects. Because AADL does not support tuple-valued properties, we use a pair of properties to associate both a classification and a set of categories with each subject/object. The properties `Class` and `Category` that declare an item's classification and set of categories, respectively, are defined in the property set `Security_Attributes`, shown in Table 1.

Table 1: The Property Set `Security_Attributes`

```
property set Security_Attributes is
 -- Specifiy security class of a component.
 Class: inherit Security_Types::Classifications
=>
  value(Security_Types::Default_Classification)
  applies to (data, subprogram, thread, thread
       group, process, memory, processor, bus,
```

```
        device, system, port, server subprogram,
        parameter);
  -- Specify security categories of a component.
 Category: inherit list of
Security_Types::Categories => ()
  applies to (data, subprogram, thread, thread
        group, process, memory, processor, bus,
        device, system, port, server subprogram,
        parameter);
end Security_Attributes;
```

Both properties apply to all component categories as well as to all feature categories. The definitions references two property types, Classifications and Categories, and a property constant, `Default_Classification` that are defined in the secondary property set `Security_Types`. This facilitates customization of the space of security levels and is analogous to the AADL standard's use of the `AADL_Project` property set to allow customization of the property definitions in the otherwise fixed `AADL_Properties` property set.

More specifically, the `Classifications` property type of the `Class` property is expected to be an enumeration. Because AADL treats enumeration literals as being ordered as they are declared in the enumeration type, the `Classifications` type thus provides a totally ordered set of classifications for the space of security levels. The property constant `Default_Classification` provides the default value for the `Class` property. The value of this constant is expected to be the first literal of the `Classifications` enumeration type.

The `Categories` property type is also expected to be an enumeration. In this case, the enumeration is used to define the set of categories applicable to the problem space of the model. Because we are not interested in ordering the categories, we could have used `aadlstring` as the type instead. We chose to use a specific user-defined enumeration type because it provides better support for error checking; the AADL parser flags the use of categories that are not part of the declared enumeration type as syntax errors (i.e., typographical errors in category labels are detected at the time the model is parsed). Because `aadlstring` values cannot be checked this way, errors in category labels would be more difficult to detect (i.e., they would appear as unexpected security level mismatch errors). Acceptable values for the `Category` property are actually lists of values of type `Categories`, thus providing the second component of the security level, the set of categories.

The property types `Classifications` and `Categories` and the property constant `Default_Classification` are declared in the `Security_Types` property set, shown in Table 2.

Table 2: The Property Set Security_Types

```
property set Security_Types is
 -- The levels of security that are
applicable to the system.
 Classifications:
   type enumeration (unclassified,
      confidential, secret,top_secret);

-- This constant should always be set to
-- the first element of the Classifications
-- enumeration.
 Default_Classification:
  constant Security_Types::Classifications
=> unclassified;

 -- The categories for information.
 Categories:
   type enumeration (A, B, C, D);
end Security_Types;
```

The default classifications levels are the standard military classifications levels with the ordering unclassified < confidential < secret < top secret. The default categories are place holders to be replaced. As stated previously, the intent is that the modeler customizes the `Classifications` and `Categories` enumerations based on the domain of the system being modeled.

4.2 SUBJECTS AND OBJECTS IN AADL

In the Bell-LaPadula model, active subjects act on passive objects. In AADL, components communicate through ports and other categories of features. For the most part, data is not explicitly represented in the model; the exceptions are data subcomponents and data access features. Instead, data ports and other features are associated with a data classifier that describes the data objects that flow over connections and into or out of ports. A feature, with its associated attributes and properties, is a proxy for the data that pass through it. These observations motivate us to consider, in general, AADL components as subjects and AADL features as objects.

4.2.1 AADL Components

We treat all AADL components with the exception of data and subprogram components as subjects. Components are sites of activity that coordinate the movement and generation of data throughout the system. Each component is expected to have a security level to describe its clearance to utilize objects. Table 3 shows an example of a component type with a security level.

Table 3: A Thread Component with the Security level (confidential, {A}).

```
thread producer
 // Features, etc. are elided
 properties
  Security_Attributes::Class =>
       confidential;
  Security_Attributes::Category => (A);
end producer;
```

AADL data components are pure objects. Although they can contain subprogram features, data components do not possess an active nature: external threads of control must invoke data component's subprograms. A data component is operated on through data access features that enable direct access to its contents. Thus, a data component is expected to have a security level to describe its contents.

AADL subprogram components are both subjects and objects. They are subjects in the sense that they have the capacity to manipulate objects as a result of being executed. But they are objects in the sense that they need to be executed by an external agent. So a subprogram component is expected to have a security level to describe simultaneously its clearance to use objects and the clearance required to invoke the subprogram.

4.2.2 AADL Port Features

Because of its name, the data port feature is the most obvious starting point for the discussion. A data port feature transmits or receives a data object. It is thus clearly a conduit through which we can observe a component's access to a passive Bell-LaPadula object. A fully specified data port feature includes a data classifier that describes the data objects that pass through the port. When this classifier is available, we retrieve the `Class` and `Category` properties from it to determine the security level of the data port feature; when the classifier is not available, we retrieve the security attributes from the feature itself. This is consistent with the metaphor that the data classifier describes the data that passes through the port. Table 4 shows the declaration of a data port feature output in the thread type producer. The security level of the feature is (confidential, {A}) because that is the security level of the data type A.

Table 4: An Example Declaring the Security level of Features: Data port output gets its security level from the data classifier A.

```
data A
 properties
  Security_Attributes::Class =>
       confidential;
  Security_Attributes::Category => (A);
end A;
```

```
thread producer
 features
  output: data port A;
  interrupt: event port {
   Security_Attributes::Class =>
       confidential;
   Security_Attributes::Category => (B);
  };
 properties
  Security_Attributes::Class =>
       confidential;
  Security_Attributes::Category => (A, B);
end producer;
```

Event data port features and subprogram parameter features are modeled in the same way as data port features. Event data port features differ from data ports only by their delivery semantics, so it is also straightforward to consider them as Bell-LaPadula objects. Because subprogram parameter features also represent the transfer of data objects, we consider them to be objects in the application of the Bell-LaPadula model.

Event ports do not pass explicit data objects between components. The raising of an event, however, can be interpreted as the transfer of an "event happened" data object, one that need not be explicitly represented because there is only one value. Alternatively, one can easily imagine the need to constrain the observation of particular events to those components with an appropriate security level. For example, an event that communicates that an intruder was detected should not necessarily be publicly available because we might not want the intruder to be able to learn of the detection by querying a public access point. We thus also consider event ports to be objects in the Bell-LaPadula model. Because it never has an associated data classifier, an event port feature's security level property values are always retrieved from the feature itself. Table 4 also shows the declaration of an event port feature interrupt in the thread type producer. Its security level (confidential, {B}) must be explicitly declared with the feature.

The security level of a port represents the exact security level of the data that passes through the port. In particular, it does not represent the maximum security level of the data. Such a choice would cause less precise modeling and analysis. We deliberately opted to provide more precise modeling and analysis by having a port's security level represent the exact security level of the data passing through the port.

4.2.3 AADL Port Group Features

AADL port groups aggregate features. From an architectural point of view, they are a container for ports. A fully defined port group feature includes a reference to a port group type. The port group type declares the features of the port group. As with port

features, the security level of the port group feature is obtained from the port group type declaration. As a basic principle of containment, we require that the security level of the port group dominates the security levels of the features in the port group. The feature declarations in the port group determine the security levels of the ports in the port group.

4.2.4 AADL Access Features

Unlike data ports, data access features represent direct access to a data object that is ultimately represented by a data component instance. We must still, however, treat the feature as a proxy for the data because the exact data component being accessed is unknown outside the "providing" component. As might be expected, the rules for checking data access features differ from those for checking data port features due to their differing semantics in AADL.

So `Class` and `Category` as defined previously, do not apply to data access features but to data components. If we were to modify their definitions with the access modifier, as shown in Table 5, however, the properties would apply to data access features but not data components.

Table 5: Modifying Feature Definitions

```
property set Security_Attributes is
 Class:
   access inherit … applies to (data, …);
 Category:
   access inherit … applies to (data, …);
end Security_Attributes;
```

The solution is to declare a second a pair of property names that apply to data (and bus) access features only. This declaration, unfortunately, makes annotating the model more inconvenient, but the strict checking of property applicability by the AADL tool environments prevents the wrong pair of properties from being used for a particular model element.

The additional property definitions are shown in Table 6. Besides their name and applies to clauses, they are defined identically to the `Class` and `Category`. Our use of the secondary property set `Security_Types` now comes into its own because it prevents the modeler from having to modify two pairs of property definitions when the classifications and categories applicable to a model need to be altered.

Bus access features are analogous to data access features, as Table 6 shows.

Table 6: The Property Set `Security_Attributes`, Revised to Handle Data Access Features

```
property set Security_Attributes is
 -- Class & Category defined here...

 -- Specifiy the security class of an
 -- access feature.

 Class_Access: access inherit
    Security_Types::Classifications =>
    value(Security_Types::Default_Classifica
tion)
    applies to (data, bus);

 -- Specify the security categories of an
 -- access feature.
 Category_Access: access inherit list of
    Security_Types::Categories => ()
    applies to (data, bus);
end Security_Attributes;
```

4.2.5 AADL Subprograms as Features

A subprogram feature represents an execution entry point provided by the component. The actual subprogram to which access is provided is described by the feature's associated subprogram classifier. As with port and access features, the security level of a subprogram feature is retrieved from the feature's associated subprogram classifier if it is present or from the feature itself if the classifier is missing.

4.3 ACCESS MODES IN AADL

The Bell-LaPadula model defines four access modes to describe a subject's effects on an object:

- *Execute* access does not permit the subject to observe or alter the contents of the object.
- *Read* access permits a subject to observe, but not to alter, the contents of the object.
- *Append* access permits a subject to alter, but not to observe, the contents of the object.
- *Write* access permits a subject both to alter and observe the contents of the object.

Given our mapping of objects to AADL features and data and subprogram components, we must derive access rights based on the AADL semantics for those features.

4.3.1 AADL Port Features

Data ports transmit and receive data objects by marshalling and un-marshalling, respectively, complete objects between threads through buffers that represent the port in the component. That is, objects obtained from an in data port are read, and objects sent throughout data ports are newly created. Thus, an in data port corresponds to read access, and an out data port corresponds to append access. An in out data port is bidirectional, but not on the same data object: it is more like a port that can be used for both sending and receiving object, but not simultaneously. When used to receive, an `in out data port` thus corresponds to *read* access,

and when used to transmit it corresponds to *append* access.

Event ports communicate events between threads. An in event port facilitates the observation of the event's occurrence and corresponds to *read* access. An `out event port` creates a new announcement of the event and corresponds to *append* access. Again, an `in out event port` is never simultaneously observing and announcing, so its access mode is determined according to its current usage.

An event data port combines the semantics of an event port with that of a data port. For our purposes, an event data port is like a data port; in particular, event data ports transmit complete data objects between threads. Unlike an `in data port`, an `in event data port` queues received objects. You might suspect that an `in event data` port is considered to have a *write* access because it must read from and modify the queue. However, it is worth noting that the queue is not significant for security analysis, since we are interested in the data being transmitted through the queue.

The semantics of subprogram parameter features are similar to those of data port features. Data is copied into the subprogram when it is called and out of the subprogram when it returns. As with data ports, an `in parameter` corresponds to *read* access, an `out parameter` corresponds to *append* access, and an in out parameter has *read* access during the call phase and *append* access during the return phase.

4.3.2 AADL Port Group Features

Fundamentally, a port group is simply a bundle of ports, including those ports of any nested port groups. Conceptually, components interact via the ports contained in a port group not via the port group itself; the port group is simply an abstraction that bundles together related ports. Thus a port group itself has no direction, but the ports it contains do have direction. Therefore, in general, we cannot speak of the access mode of the port group feature, only of the access modes of the individual ports contained in the port group. These are determined as described in Section 4.3.1.

There are, however, two aspects of AADL port groups that complicate the above reasoning:

- A port group might be empty. This aspect supports incremental modeling by allowing empty port groups to be declared to abstractly represent communication between components, even when the exact nature of the communication has not yet been decided upon.
- Two port groups may be directly connected via port group connections. In fact, AADL allows access to the contained features of a port group when connecting a subcomponent and its containing component only; sibling subcomponents must connect port groups in their entirety using port group connections.

There are cases where we must consider the port group as a single entity in terms of data access. In those instances, the security level of the accessed object is the security level of the port group. Just as the security level of a port group must be a maximization of the security levels of its constituent ports, so must the access attributed to the port group be a maximization of the access attributed to its ports. Conservatively, this implies that we treat a port group as having *append* access—the only choice when dealing with an empty port group (because we do not know the directions of the ports yet to be added). We can be less conservative by inspecting the directions of the port group's features: if all the features in a port group (including any nested port groups) are in ports, the port group can be treated as *read* access.

4.3.3 AADL Access Features

AADL data access features represent direct access to shared data. Although AADL distinguishes between `provides` and `requires` data access features, this distinction does not indicate the nature of the access to the data enabled by the port. The standard properties `Required_Access` and `Provided_Access` apply to `requires` access and `provides` access features, respectively, and indicate the kind of access enabled by the feature. Acceptable property values are `read_only`, `write_only`, `read_write`, and `by_method`. The first three indicate direct manipulation of the shared data and map naturally to the access modes *read*, *append*, and *write*, respectively. The `by_method` access form indicates that the shared data object may be manipulated using the sub-program features associated with its data classifier only. Because we do not know what the sub-programs do to the data, we map `by_method` to the *write* access mode because it could read or write the data.

Bus access features are analogous to data access features.

4.3.4 AADL Subprograms

The use of an AADL subprogram within a component implementation calls block corresponds to an invocation of the subprogram. For both subprogram features and subprogram classifiers, invocation corresponds to a Bell-LaPadula *execute* access. Even though the caller must be able to observe the sequence of machine instructions that compose the subprogram in order to invoke it, we do not consider invocation to be a *read* access because the subject is not reading the instructions themselves

and using them as data for a computation. If that computation were performed—for example, to compute a checksum of the program's image in memory or as part of self-modifying code—we would have to consider the subprogram as data with the appropriate *read/write/append* access modes.

5. Conclusion

In this paper we have discussed how model-based engineering can support early modeling and validation of security. Specifically, using the AADL we have defined described common and well-defined security attributes and represented them in the AADL models. The adopted notion is primarily based on the Bell-LaPadula model, which has been recognized as a cornerstone in validating security. Using the AADL and based on the Bell-LaPadula and extended sibling models, one can model and validate security according to flow-based approaches, Bell-LaPadula, Chinese Wall, and role-based access. To support security analysis, we have taken established criteria from the Bell-LaPadula model and defined additional criteria that allow us evaluate how viable a system is to enforce security, given confidentiality requirements of data objects and security clearance by users (e.g., ensure processes and threads are mapped to appropriate hardware, communicate over secured channels, and reside/store data in protected memory). The analysis techniques are implemented in an OSATE plug-in.

The overall objective of a secure system implies that security clearances are given conservatively (as opposed to generously). To this end, we can analyze models to derive the minimum security clearance on components in the model. Or to put it differently, we can use the notion of subjects and objects to determine the minimum security clearance for a subject based on the requirements of the objects being accessed by the specific subject. By also pointing out differences between actual security clearances and the minimum security clearance required, a system designer can evaluate how effective and tight security is. By providing mechanisms to ensure that sanitization is conducted within allowed boundaries, the designer is allowed to analyze and trace these relatively more threatening security risks, as since sanitizing actions are permitted exemptions of security criteria and rules, and as such should be minimized in the system.

References

[1] Bell, D. E. & LaPadula, L. J. Secure Computer Systems: Mathematical Foundations (MITRE Technical Report 2547, Voume 1). 1973.

[2] Bell, D. E. & La Padula, L. J. Secure Computer Systems: Unified Exposition and MULTICs Interpretation (MITRE Technical report ESD-TR-75-306). 1976.

[3] Brewer, David D. C. & Nash, Michael J. "The Chinese Wall Security Policy," 206–214. IEEE Symposium on Security and Privacy, Oakland, CA, May 1–3, 1989.

[4] Lin, T. Y. "Chinese Wall Security Policy—An Aggressive Model," 282–289. Proceedings of the Fifth Aerospace Computer Security Application Conference. 1989.

[5] Ferraiolo, David & Kuhn, Rick. "Role-Based Access Control," 554–563. Proceedings of the 15th National Computer Security Conference, 1992. ACM Press, 1992.

[6] Fenton, J. S. "Memoryless Subsystems." The Computer Journal 17, 2 (May 1974): 143–147.

[7] Denning, Dorothy E. "A Lattice Model of Secure Information Flow." Communications of the ACM 19, 5 (May 1976): 236–243.

[8] Denning, Dorothy E. & Denning, Peter J. "Certification of Programs for Secure Information Flow." Communications of the ACM 20, 7 (July 1977): 504–513.

[9] Hansson, Jorgen & Greenhouse, Aaron. Modeling and Validating Security and Confidentiality in System Architectures. SEI/CMU-2007-TN-005.

[10] Mayfield, Terry; Roskos, J. E.; Welke, Stephen R.; Boone, John M.; & McDonald, Catherine W. Integrity in Automated Information Systems (ADA253990). Alexandria, VA (USA): National Computer Security Center, 1991.