Security Policy Foundations in Context UNITY

M. Todd Gamble
Department of Computer Science
University of Tulsa
Tulsa, OK 74104 USA
+1 918 631 2000
mtg@utulsa.edu

Rose F. Gamble
Department of Computer Science
University of Tulsa
Tulsa, OK 74104 USA
+1 918 631 2000
gamble@utulsa.edu

Matthew L. Hale
Department of Computer Science
University of Tulsa
Tulsa, OK 74104 USA
+1 918 631 2000
matt-hale@utulsa.edu

ABSTRACT
Security certification includes assessing an information system to verify its compliance with diverse, pre-selected security controls. The goal of certification is to identify where controls are implemented correctly and where they are violated, creating potential vulnerability risks. Certification complexity is magnified in software composed of systems of systems where there are limited formal methodologies to express management policies, given a set of security control properties, and verify them against the interaction of the participating components and their individual security policy implementations. In this paper, we extend Context UNITY, a formal, distributed, and context aware coordination language to support policy controls. The new language features enforce security controls and provide a means to declare policy specifics in a manner similar to declaring variable types. We use these features in a specification to show how verifying system compliance with selected security controls, such as those found in the NIST SP800-53 document, can be accomplished.

Categories and Subject Descriptors
K.6.5 [Management of Computing and Information Systems]: Security and Protection – Authentication, Insurance, Physical security

General Terms
Design, Security

Keywords
Security certification, security controls, UNITY

1. INTRODUCTION
Security policies form a governing set of constraints on all system architectures as dictated by certification and accreditation processes such as the DIACAP [1]. These policies include a minimum of access control, audit, data protection, contingency planning, and non-repudiation. The practices of examining documentation, relying on previously established

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SESS’11, May 22, 2011, Waikiki, Honolulu, HI, USA Copyright 2011 ACM 978-1-4503-0581-5/11/05... $10.00.

functionality, and testing from expansive test cases form the majority of the certification efforts which generally occur after systems of systems development and integration. The result is often continuous patching and partially verified systems. A more disciplined approach requires a formal specification and verification process.

The NIST SP800-53 [2] specifies a set of required security controls as part of its Risk Management Strategy of the Security Life Cycle. The first life cycle step categorizes the information system according to security priorities. The second step selects applicable security controls based on the priorities. These controls range across three broad classes – technical, operational, and management. The third step implements the security controls in the system and then assesses them in step four. Steps five and six authorize the deployment of the system and then monitor it for continuing compliance. Within this life cycle, we focus on steps 2 through 4, specifically working with the technical class of security controls.

Critical security vulnerabilities discovered at system design time can direct design changes and risk mitigation strategies early in the life cycle. This discovery requires an expressive and complementary representation of both system behavior with respect to its implemented security policies and the security control requirements for managing those policies. Thus, we must devise a transparent means to capture both security capabilities and the administration of capability configurations selected during design. In this paper, we extend Context UNITY [3] to express and reason about security management policies and their administration as related to exposed, or shared, program variables that can be accessed by external components as a form of interaction. We model the implementation and configuration of selected security controls, making configuration choices transparent within the design model through explicit specification language features, context dependent behavior, and parameterized execution. Through examples we show verification of compliance with selected security controls.

2. BACKGROUND
Three major security control documents are the NIST SP800-53 [2], the Common Criteria (CC) [4], and the DoDI 8500.2 [5]. Each provides a structure for and grouping of security controls expressed at varying levels of abstraction. Mappings are not 1:1 among the controls and in many cases there is no mapping. While we use the NIST SP800-53 in our example, we put in parentheses associated controls within the CC and DoDi 8500.2.

As part of a model-based security engineering approach, Jürjens et al. [6] analyze compliance at the physical, logical and policy level using a model-based approach founded in UMLsec.
Predefined UMLsec stereotypes for security constraints are embedded in UML diagrams to assess the presence of particular security requirements given a similar diagram of the concrete system implementation, highlighting the use of automated tools for compliance checking. Similarly, Best et al. [7] explore the implementation of a search feature being added to the existing corporate intranet architecture of a German car manufacturer, again using automated model checking tools to determine if the system model delivers the security functionality. This approach relies on the soundness of the UML diagrams and the tools associated with UMLsec models.

Our approach uses a formal foundation grounded on the concept of data sharing via tuple spaces – a form of distributed shared memory consisting of ordered lists (tuples) accessible to all cooperating processes. Context UNITY [3] has features of shared data coordination languages, based on the model originally proposed in Linda [8]. While Linda captures the uncertainty of system evolution and integration, it is well known that is does not include an inherent security model. The recent resurgence in interest around coordination language has resulted in multiple approaches to extending the basic Linda coordination model to include security features. See [10] for a comprehensive summary.

Fundamentally, existing approaches to securing distributed tuple spaces rely on one of two mechanisms to provide security [9, 10]. First, the shared tuple space is partitioned into multiple spaces by providing for a space ‘owner’ or ‘server’ for each partition. These servers must secure the tuples that they provide to other programs. Second, tuples are protected by various encryption and digital signing mechanisms so that tuple access requires presentation of these credentials to access its contents. Our approach allows the implementation of both models. The set of exposed variables declared by a Context UNITY program partitions the tuple space that is ‘owned’ by the program. We refine the access control to provide for access control lists on each exposed variable. Encryption is also supported in a generic fashion and may be used at the framework level (e.g., to protect an entire tuple space partition) or at a single tuple to provide confidentiality as required for that specific variable’s use.

Inclusion of these features in the formal modeling framework is done in such a way as to make the intent of the design more transparent. Declarations of tuples, tuple spaces, and the access control mechanisms for them are encoded within the specification to show both the security mechanism as well as its explicit configuration in a particular implementation instance. Given our objective to enable certification, access control is structured to complement audit since it is in fact audit controls which dominate much of the verification methods for security certification. Our reframing of formal language features to support audit in addition to access control is unique from prior approaches.

Context UNITY augments the notation of Mobile UNITY [11] to provide formal representation and reasoning of context-aware programs. Both retain the base specification language constructs of UNITY [12]. Two fundamental aspects of UNITY are carried into the extensions. The first is the non-deterministic, weakly fair execution of all assignment statements (including context rules, defined later in the section). The second is the temporal proof logic based on the execution model which provides a basis for proof of safety and liveness properties.

<table>
<thead>
<tr>
<th>System</th>
<th>SystemName</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td>ProgramName (parameters)</td>
</tr>
<tr>
<td>declare</td>
<td>internal – internal variable declarations</td>
</tr>
<tr>
<td></td>
<td>exposed – exposed variable declarations</td>
</tr>
<tr>
<td></td>
<td>context – context variable declarations</td>
</tr>
<tr>
<td></td>
<td>initially – initial conditions of variables</td>
</tr>
<tr>
<td>assign</td>
<td>assignments to declared variables</td>
</tr>
<tr>
<td>context</td>
<td>Context rule definitions affecting context variables - can pull information from and push information to the environment via interaction with exposed variables of other programs</td>
</tr>
<tr>
<td>end</td>
<td>ProgramName</td>
</tr>
<tr>
<td>. . .</td>
<td>additional program definitions . . .</td>
</tr>
</tbody>
</table>

Components
Instances of Programs declared with parameters and unique names
Governance
Global impact statements that can access any exposed program variable
end | SystemName |

Figure 1: Context UNITY Specification Structure [3]
A Context UNITY system specification is structured as in Figure 1 (comments in normal font). The System can have multiple Program specifications instantiated with unique ids in the Components section. Each Program has a variable declaration section. Internal variables are private to the program, exposed variables store public data accessible outside the program, and context variables are local variables that can push and pull information to and from exposed variables of other programs. Context UNITY requires all programs to have two built-in exposed variables: a component ID (id) that cannot be changed and an internal variable credentials which can be changed and whose value supplies the necessary parameters for access control to external variables.

The assign section contains three statement types: normal assignment statements to internal and exposed program variables, transactions for executing multiple simple assignment statements atomically, and reactive assignment statements that execute given a specific state change. Reactive statements execute to fixed point before returning execution control.

Table 1: The Original Context UNITY var Table

| t | The variable’s unique id |
| α | The id of the owner component |
| γ | The name |
| ρ | The type |
| υ | The value (possibly a complex data type) |
| α | The access control policy |

The context section contains the rules which manage the component’s interaction (interface) with the exposed variables of other components that are uniquely represented by a tuple defined by the var table shown in Table 1. This structure is required for an external reference to an exposed variable by an external component. In the context section, reactive context rules execute to fixed point after the execution of any simple or transactional assignment statement in the assign section. The reactive context rule in Figure 2 with its var table expansion is taken directly from [3], where c is a declared context variable. Formally, c is assigned (becomes, c := var(a[t, ω]) the value of external variable a’, chosen non-deterministically from exposed
variables in external programs satisfying condition Q, i.e., \( a = a'.Q \). The conditions state that the value of \( a' \) is greater than local variable \( x \) and the credentials of the program reading from and writing to \( a \) meet access control requirements from \( a \) in Table 1. This assignment is performed in parallel (\( ( \) ) with impacts, which assigns 0 to \( a \) (affecting another program). The notation is often shortened where \( a \) represents its value.

<table>
<thead>
<tr>
<th>Context rule:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( c ) uses ( a )</td>
</tr>
<tr>
<td>( \text{given } a &gt; x )</td>
</tr>
<tr>
<td>( \text{where } c \text{ becomes } a )</td>
</tr>
<tr>
<td>( 0 \text{ impacts } a )</td>
</tr>
<tr>
<td>reactive</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Formal notation for context rule:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;a':a'{\text{var}[a]}.v &gt; x \land {r} \subseteq \text{var}[a].\alpha(\text{credentials})\rangle)</td>
</tr>
<tr>
<td>( \land {w} \subseteq \text{var}[a].\alpha(\text{credentials}))</td>
</tr>
<tr>
<td>( \</td>
</tr>
</tbody>
</table>

Figure 2: Context Rule and Formal Notation

### 3. POLICY IMPLEMENTATION

We represent an exposed variable (tuple) security policy implementation by extending Context UNITY with:

- Refinements to access control mechanisms provide an interface for initializing and altering access control lists
- An encryption function which carries the type, keys, and other meta data to provide a clearly defined encryption mechanism for the variable
- An audit history as a stream of events impacting each exposed variable that are identified as `auditable events`
- An audit function, which provides the means for encoding `auditable events` for a variable.

We represent a component security policy implementation by extending Context UNITY with:

- A policy declaration associating security controls with exposed variables via the use of access, audit, encrypt keywords
- Presentation variables (such as audit_log) for defined aggregates of policy results

We form a system policy implementation by using the Governance section in Figure 1 to express system-wide concerns. The Governance section allows access and impact to the entire space of exposed variables across all components in the system. We specify a compliance model by extracting functional requirements from selected security controls and specifying them as safety and progress properties over the system and components. Proof of these properties indicates compliance with the respective security controls.

#### 3.1 Access Control

Exposed variables are owned by the components that declare them. As mentioned in Section 2, a partition of the entire tuple space of exposed variables is delimited by this ownership relation as identified by the built-in \( id \) (\( \pi \) in Table 1). Thus, access control for exposed variables is controlled by the owning component.

In our access control model, subjects may perform actions on objects, where objects are exposed variables. When creating access control lists, assignments are provided as ordered pairs of the form \( (\text{subject, action}) \) and associated with a specific exposed variable. Table 2 lists what a subject can be, with a subject of a particular user considered the most restrictive. Note that when access is opened by the `**` subject wildcard notation, audit records would still include the real identity of subjects bound at runtime when exposed variables are accessed. The model provides for further extension of these choices and Table 2 should be considered an example.

We do not provide the detail here for brevity, but the access control function \( a \) (Table 1) contains the access control metadata (e.g., access control lists) defining relations between users and their access rights to an exposed variable. Assignments of access control lists in the specification populate this metadata.

<table>
<thead>
<tr>
<th>Table 2: Forms of subject</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
</tr>
<tr>
<td>Group</td>
</tr>
<tr>
<td>Role</td>
</tr>
<tr>
<td>CompID</td>
</tr>
<tr>
<td>ProgramName</td>
</tr>
<tr>
<td>Component Group</td>
</tr>
<tr>
<td>*</td>
</tr>
</tbody>
</table>

We make explicit the setting of the access control policy values for an exposed variable during declaration using the keyword access or during execution by invoking the \( a \) (Table 1) attribute on the left-hand side of an assignment statement as shown below. For example, Bill has write access to \( a \) and all students have read access. Sue has read and write access to \( a \).

```
exposed a : Integer access (\{Bill\},\{w\}), (role:student,\{r\})
assign var[a].\alpha := (\{Sue\},\{r,w\})
```

We also introduce policy section notation with the capability to define access control across multiple variables (via simple templating and wildcarding) as well as to override or augment existing access control lists for exposed variables. This extension will be described more as it relates to Figure 4.

Note that careful choice of subject/action values and wildcards within the policy section templates can enable collections of exposed variable policy configurations to be aligned with broad protection criteria. Examples include:

- All variables accessible to a single user.
- All variables writable by a role.
- All components involved with the same variable (e.g., all clients of the same database).
- All technology types permissible for an exposed variable.
- The collection of all variables accessed by a subsystem

#### 3.2 Encryption

We define the functions encrypt and decrypt where

\[
y' := encrypt(y, c, x), y := decrypt(y', c, x)
\]
assigns the encrypted version of variable \( y \) to \( y' \). The encryption is performed using algorithm \( x \) and the credentials \( c \) of the current process. Variables \( y \) and \( y' \) are assumed to be of the same type. The function \( decrypt \) performs the reverse operation and assigns to \( y \) the decrypted version of \( y' \) using algorithm \( x \) and credentials \( c \). For a proper roundtrip to occur, the values of \( x \) in both cases must be identical.

We extend the \( \text{var} \) table to include the attribute \( \kappa \) signifying which cryptographic algorithm is used to encrypt an exposed variable. For any exposed variable \( a \) that is encrypted, the decrypted version of the variable is given by,

\[
decrypt(\text{var}[a], \nu, \text{credentials}, \text{var}[a], \kappa),
\]

where \( \text{credentials} \) holds program attributes such as passwords, certificates and cryptographic keys as defined originally in Context UNITY [3]. If the variable \( a \) is not encrypted then \( \text{var}[a], \kappa = \emptyset \) and any invocation of the \( decrypt() \) function when this field is \( \emptyset \) returns the current value in \( \text{var}[a], \nu \).

Whenever an exposed variable is read or written, the required \( encrypt \) and \( decrypt \) operations are implicitly assumed. If the \( \text{credentials} \) provided are inadequate to perform the encrypt or decrypt operations (e.g. a private key or passphrase does not meet access requirements, or a \( \tau \) is not allowed), then the access control policy will have caused the operation to fail in advance of attempting any encrypt/decrypt operations. Effectively, the application of the access control policy acts as a guard for any encryption or decryption that might occur. Using the context rule in Figure 2 and assuming that \( a \) is encrypted, we show the formal notation supporting both the existing access control and the new encryption.

\[
\begin{align*}
\text{exposed} & : a = a' \quad (\text{var}[a], \nu) > x \\
& \quad \land \quad [\nu] \subseteq \text{var}[a], \omega(\text{credentials}) \\
& \quad \land \quad [\omega] \subseteq \text{var}[a], \omega(\text{credentials}) \\
& \quad || \text{var}[a], \nu \Rightarrow decrypt(\text{var}[a], \nu, \text{credentials}, \text{var}[a], \kappa) \\
& \quad \text{reacts-to} \quad true > \\
\end{align*}
\]

In order for exposed variables in any given component to have an encryption mechanism, we must make assignments to their respective \( \kappa \) attributes in the \( \text{var} \) table using an \( encrypt \) modifier during variable declarations. Assume that we have access to the cryptographic algorithms DES, AES, and TripleDES. Sample exposed variable declarations are

\[
\begin{align*}
\text{exposed} & : user_id : \text{String} \quad encrypt \text{ } \{ \text{DES} \} \\
& \quad \text{mail_queue : Sequence of Mail_Item} \quad encrypt \text{ } \{ \text{Triple-DES} \} \\
& \quad \text{credit_card_no : String} \quad encrypt \text{ } \{ \text{AES} \}
\end{align*}
\]

This approach allows us to annotate existing specification code with encryption guidance without modifying the code. Each use of the \( encrypt \) keyword causes the augmented \( \text{var} \) table entry \( \kappa \) for the referenced variable to be updated with the corresponding encryption algorithm. Therefore, to determine the encryption used for any specific exposed variable, we need only refer to its \( \kappa \) attribute in the \( \text{var} \) table.

To determine the set of encryption algorithms used within any given program component, we can perform a union across all \( \text{var}[a], \kappa \) entries for variables assigned to that component. The same holds true for determining the encryption algorithms across an entire system. Note that we may also discuss partitions of programs within a system based on their encryption algorithms (i.e., a common, compatible securing infrastructure). For example, we could prove that two programs which deal with similar data (e.g., privacy data for users) do or do not share common algorithms.

### 3.3 Auditing

For an audit management policy, we must provide a means to detect and record changes to an exposed variable as part of the policy implementation. Potential actions on an exposed variable’s value (\( \nu \) in Table 1) are shown in Table 3. Both successful and failed actions are represented to form a list of auditable events which may be selected for audit by an organization for any given variable.

The audit configuration for a component dictates which of the events below are recorded in each exposed variable’s history. In addition, these base events may be augmented by additional relations that detect variable transitions that are particular to the needs of a specific program instance. Registration of relational tests allows the audit system to detect if the relation holds true for the current value of the exposed variable. The value tested is after other actions are performed that can alter the value (i.e., \( \text{write}, \text{create}, \text{and destroy} \)). The variable’s value must change from one of these prior actions for the relational test to result in an entry in the history, otherwise all variable accesses would “fire” the relational test.

<table>
<thead>
<tr>
<th>Relation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>The variable’s value is successfully read</td>
</tr>
<tr>
<td>write</td>
<td>The variable’s value is successfully written</td>
</tr>
<tr>
<td>create</td>
<td>The variable’s value is initialized</td>
</tr>
<tr>
<td>destroy</td>
<td>The variable’s value is reset to uninitialized</td>
</tr>
<tr>
<td>read-fail</td>
<td>Unauthorized attempt to read ( \nu )</td>
</tr>
<tr>
<td>write-fail</td>
<td>Unauthorized attempt to assign ( \nu )</td>
</tr>
<tr>
<td>create-fail</td>
<td>Unauthorized attempt to assign ( \nu ) when its prior value was ( \text{null} )</td>
</tr>
<tr>
<td>destroy-fail</td>
<td>Unauthorized attempt to set ( \nu ) to ( \text{null} )</td>
</tr>
</tbody>
</table>

Relations may be unique per variable type. For instance, registered auditible events for an Integer variable might include:

- \(< 0 \), the variable’s value became negative
- \( \geq 10 \), the variable’s value became 10 or greater
- \( = 5 \), the variable’s value became equal to \( 5 \)

Auditable events (AEs), such as those requiring organizational definition in NIST SP800-53, are defined by specific patterns of operation or relations on exposed variables that occur within a context rule. The patterns inform a special function, \( \omega \), newly defined in the \( \text{var} \) table for each exposed variable. For exposed variable \( a \), the AE patterns that should result in log entry are defined for \( \text{var}[a], \omega \). The definition of AE patterns to be added to each exposed variable within a program occurs by invocation of the \( \text{audit} \) macro either in exposed variable declarations directly or in the new policy section of a program where patterns may be declared over multiple exposed variables sharing similar types. When invoked, \( \text{var}[a], \omega \) forms the log entries corresponding to tests of the audit patterns against the current value of \( a \) (an assumed parameter) and any other operations referenced via explicit parameters (e.g., a failed read attempt).

To create an audit record for each exposed variable that has an audit pattern (i.e., \( \text{var}[a], \omega \neq \emptyset \)), we further extend the \( \text{var} \) table with a \( \text{history} \) element. For a variable \( a \), the \( \text{history} \) is
denoted by \( \text{var}[a].\delta \). This completes our extensions to the original \( \text{var} \) table (Table 1) which are shown in Table 4.

### Table 4: Extended var Table

| \( I \) | The variable's unique id |
| \( \pi \) | The id of the owner component |
| \( \eta \) | The name |
| \( \tau \) | The type |
| \( \omega \) | The value (possibly a complex data type) |
| \( \alpha \) | The access control policy |
| \( \kappa \) | The encryption method |
| \( \omega \) | The audit settings (what is an auditable event) |
| \( \delta \) | The history of actions on this variable |

Since the variable history must record all auditable events with respect to an exposed variable, it contains a set of audit records with the elements in Table 5.

### Table 5: Audit Record Format

| Component ID | Owner of the audit log |
| Subject | Credentials of the component that caused the log creation |
| Object | Exposed variable that is accessed |
| Action | The attempted action on the variable |
| Result | Success or failure of the action |
| Time | Global clock time when action occurred |

Using the context rule in Figure 2 again, we show the capture of the auditable event as follows

\[
\langle a : \alpha \rangle [\text{var}[a].\delta > x \wedge \tau \leq \text{var}[a].\alpha(\text{credentials}) \wedge (w \leq \text{var}[a].\alpha(\text{credentials})) \mid (c := \text{var}[a].\nu, \text{var}[a].\delta := \text{var}[a].\alpha(\tau)) \mid \| \text{var}[a].\nu := 0 \mid \text{var}[a].\delta := \text{var}[a].\alpha(w)) \rangle
\]

\[
\langle a : \alpha \rangle [\text{var}[a].\delta > x \wedge \tau \leq \text{var}[a].\alpha(\text{credentials}) \mid (c := \text{var}[a].\nu, \text{var}[a].\delta := \text{var}[a].\alpha(\tau)) \mid \| \text{var}[a].\nu := 0 \mid \text{var}[a].\delta := \text{var}[a].\alpha(w)) \rangle
\]

Note that parallel assignment to the history due to both read failure and write failure can occur because two different history entries are formed. We could avoid this complication for access control violations by refining the definition of \( \alpha \) to always include logging of failed actions. We can further expand the statement above to include encryption in the same manner as in the first example.

Use of the \( \ominus \) function is dependent on the AE sets being defined with the new audit keyword. For example, below are two exposed variables, audited_var and num_users that have an audit keyword in their declaration. Audited_var relies on a pattern definition and comparison operators that reference the current value of the exposed variable as that value is initialized and altered throughout the program. Num_users relies on a template definition of the audited action which is further defined in the policy declaration section described next.

\[
\text{exposed}
\]

\[
\text{av}/\text{audited_var} : \text{Integer audit} \{w,w\text{-fail},<10,=x\}
\]

\[
\text{un}/\text{num_users} : \text{Integer audit always_positive}
\]

As shown below, the policy section can be used for a broader application of policies in a more concise manner, such as defining templates. The policy section can also be a dynamic policy implementation on an exposed variable when it relies on input parameters to the program (see program in Figure 4).

\[
policy
\]

\[
*: \text{Username audit} \{w\}
\]

\[
*: * \text{audit} \{t\text{-fail}\}
\]

\[
\text{always_positive} : \text{audit template} \{<0\}
\]

The first declaration in the example above states that logging should be performed for any write operation to a variable of type Username. The second declaration says to log all failed reads that are due to the exposed variable’s access control policy. The third declaration defines the template always-positive as a pattern to audit the variable when it becomes negative.

The auditable events for a program can be specified as the aggregation of events defined over the exposed variables of that program. Since the \( \text{var} \) table identifies components for each variable, we can select the variables for a given program instance (a component) and reference the AE patterns for each to create a complete set.

Assume that every program supplies a function time which returns the current time taken from a global clock. We require that all programs declare a built-in exposed variable called audit_log that is a set of audit records of the form in Table 5. Audit_log aggregates all of the audit activity in the program. The declaration can contain encryption and access control constraints as dictated by the program. Individual tuples are referred to by audit_log[{v},t] for id and time pair. Addition to audit_log is denoted by \( \ominus \). Deletion is not permitted.

With the capture of a history of the actions on exposed variables that cause the creation of a log, we can formally represent the audit_log as the union over all exposed variable histories for a component \( C \).

\[
C.audit_log = \bigcup\text{var}[x].\delta\text{where var}[x].\pi = C.id
\]

### 4. EXEMPLIFYING THE NOTATION

In this section, we specify a home monitoring system using our notational extensions to Context UNITY for modeling. We follow the system specification with compliance verification properties derived from three NIST SP800-53 audit controls.

#### 4.1 System Specification

Figure 3 shows a system specification of a Home Monitoring system. We use references to programs in the system notation for clarity and expand the programs in Figures 4 and 5. There are four sensor components for the water meter, alarm system, thermostat, and electric meter, and three monitor components. Each sensor has a parameter list that is input to the component: the component id (\( \text{parms}[0] \)), the device to sense (\( \text{parms}[1] \)), a set of access controls (\( \text{parms}[1] \)), and a set of audit controls (\( \text{parms}[3] \)). For example, the alarm system sensor can be read only by the role Home with all reads and read failures being logged. The monitor parameter list has the component id (\( \text{parms}[0] \)), the set of devices to monitor (\( \text{parms}[1] \)), and the monitoring domain as a role (\( \text{parms}[1] \)). For example, the Mobile monitor expects to read the water meter, alarm system, and electric meter.

The Governance section of Figure 3 non-deterministically chooses components \( p \) and \( q \), assigning handles (via \( \text{via} \)) to audit_log (handles \( a \) and \( b \)) and id (handles \( i \) and \( j \)) that satisfy
the constraint that \( i \) in \( p \) is at most \( j \) in \( q \). The reactive governance rule then copies the \( b \) into \( a \). Eventually, given weak fairness, the program with the minimum \( id \) contains all audit logs from all components.

**System** HomeMonitoring  
**Program** Sensor  
**Program** Monitor  

**Components**  
- Sensor\( (id, \text{‘water’, \{(*, [t], \emptyset)\}) \)  
- Sensor\( (id, \text{‘alarm’, \{(*, [t], \emptyset), ([t, r-fail])\}) \)  
- Sensor\( (id, \text{‘thermostat’, \{(*, [t], \emptyset)\}) \)  
- Monitor\( (id, \text{‘alarm’, ‘thermostat’, ‘electric‘}, \text{‘Home‘}) \)  
- Monitor\( (id, \text{‘water’, ‘alarm’, ‘electric‘}, \text{‘Mobile‘}) \)  
- Monitor\( (id, \text{‘water’, ‘thermostat’, ‘electric‘}, \text{‘Internet‘}) \)

**Governance**  
- \( a \) /_audit_log, \( i/id \) in \( p \); \( b \) /audit_log, \( j/id \) in \( q \)  
  where \( i \leq j \Rightarrow a \) becomes \( a \cup b \)  
- reactive  
  end HomeMonitoring

**Figure 3:** Home Monitoring System

Figure 4 specifies the Sensor program. Note that for simplicity, no parameter checking is performed. No internal variables or context rules are needed so we omit these sections. External variables are declared for the component \( id \), the device which allows read access, the ‘sensed’ value of the device, the last time the value was updated which is always available, and the time of the last sample which is audited when reset. The external variable declarations are static, while the policy section dictates the policy implementation conditions on the exposed variables based on input parameters. In this case, \( value \) and \( lastSampled \) have additional access constraints from input \( parms[1] \) and \( value \) has audit requirements based on \( parms[3] \).

**Program** Sensor\( (parms : \text{List of Parameters}) \)  
 declare  
 exposed  
 \( id : \text{ComponentID} \)  
 \( audit\_log : \text{set of AuditRecord encrypt \{DES\}} \)  
 \( device : \text{Mechanism access \{(*, [t]\})} \)  
 \( value : \text{Integer} \)  
 \( lastDetected : \text{Time access \{(*,[t])\}} \)  
 \( lastSampled : \text{Time audit \{\{w\}\}} \)  

policy  
\( value, lastSampled \) access \( parms[1] \)  
\( value \) audit \( parms[3] \)  
initially  
\( id, lastSampled, lastDetected, device = parms[0], 0, 0, parms[1] \)  
assign  
\( value, lastDetected := \text{detect}(device), time() \)  
end Sensor

**Figure 4:** The Sensor Program

We use the shorthand UNITY notation in the initially and assign sections of the Sensor program that allow parallel assignment based on list position, eliminating the use of the parallel bars. Thus, \( id \) is initialized to \( parms[0] \), \( lastSampled \) to 0, and so on. Similarly, in the assign section, \( value \) is set using a detect method applied to the device simultaneously with lastDetected being set to the clock time.

In the Monitor program (Figure 5) there are no policy declarations or assignments, so these sections are omitted. An internal variable holds the devices to be sensed as initialized by the input parameters. The only exposed variables are \( id \) and \( audit\_log \) (though there is no audit policy implemented for Monitor), so we omit that section. A context variable \( m \) is declared to hold triplets of the monitored sensor values. A tuple identifying the role of the monitor expressed by \( parms[1] \) is added to Context UNITY’s built-in credentials variable.

The context rule in Figure 5 non-deterministically chooses exposed variables \( value \), \( device \), \( lastSampled \) and \( lastDetected \) from a system component \( p \) that satisfy the given condition. If \( d \) is a device to be monitored and the last sensor update \( (ld) \) is greater than the last sampled time \( (ls) \), then \( m \) acquires the triplet containing the device sampled \( (d) \), the value at the time of sampling \( (v) \), and the last detection time \( (Id) \). The last sampled time \( (ls) \) of \( p \) is set to the current time.

**Program** Monitor\( (parms : \text{List of Parameters}) \)  
 declare  
 internal  
 \( devices : \text{set of Mechanism} \)  
 context  
 \( m : \text{Set of (Mechanism, Integer, Time)} \)  
 initially  
\( id, devices = parms[0], parms[1] \)  
\( \| credentials = credentials \cup \{\text{role, parms[1]}\}\)  
 context  
\( m \) uses \( \forall value, device, ls/lastSampled, ld/lastDetected \) in \( p \) given  
\( d \in devices \land ld \geq ls \)  
 where  
\( m \) becomes \( m \cup \{d, v, ld\} \)  
\( time() \) impacts \( ls \)  
end Monitor

**Figure 5:** The Monitor Program

Given the specification in Figure 3, the Monitor with the role ‘Mobile’ will violate an access control policy of the alarm system. Without the extended access control policy, there would exist no formal mechanism to specify or detect this occurrence. Since the audit policy for the alarm system includes \( r-fail \), the history in the alarm system Sensor program for the \( value \) and \( lastSampled \) exposed variables should contain the access denial at the time the mobile monitor attempts to read the sensor. The formal notation of the context rule in Figure 5 would include the two assignment statements below and thus, the read failures are asserted into the histories and the assignments to \( m \) and \( ls \) are not performed.

\[ v : v \cdot t \not\in \text{var}\{v\}.o(\text{credentials}) \]
\[ v : \text{var}\{v\},d \leftarrow \text{var}\{v\},o(\text{r-failed}) \]
\[ l : l \cdot t \not\in \text{var}\{l\}.o(\text{credentials}) \]
\[ l : \text{var}\{l\},d \leftarrow \text{var}\{l\},o(\text{r-failed}) \]

### 4.2 Compliance Verification

In this section, we state a set of security control enhancements from the NIST SP800-53 [2]. We have examined these controls in terms of defining methodologies to derive functional requirements [13], and rely on that investigation to develop the verification properties for Context UNITY specifications. The controls are listed with their mnemonic for their control family (e.g., AU for audit), the control number within their family, and any enhancement numbers given for the control. We state three controls and show how compliance is assessed in the Home Monitoring System. We omit formal proofs due to space.
**Non-Repudiation:** AU-10(1) The information system associates the identity of the information producer with the information. (See also FAU_GEN.2 [4], ECAT-1 [5])

This audit control dictates that when a log is created the system must identify who performed the auditable event. The history (8) extension to the var table and the creation of the audit-log of each program use the record format of Table 5, such that var[a].δ.subject identifies the information producer. Since this occurs within the context rule, it ensures the subject is captured at the time of the auditable event and its authenticity is validated by the use of the identifying credentials within the log.

**Protection:** AU-9(3) The information system uses cryptographic mechanisms to protect the integrity of audit information and audit tools. (See also FCS_COP.1 [4], ECAT-1 [5])

We require audit_log to be a built-in exposed variable of every program available to all components in the system. However, it is only encrypted if explicitly defined, as in the Sensor program. This flexibility is allowed in order to model variations in actual system implementations. Therefore, one compliance check examines if this declaration exists. Note that it does not exist for the Monitor program. However, the Monitor program has no audit policy implemented for any exposed variables. Therefore, its audit_log would be empty. It is up to the certifier to determine if this condition – unencrypted, but empty audit information – satisfies the control. Our encoding makes this distinction obvious and verifiable. Thus, the invariant statement for verification is: For all components, var[audit_log].κ≠null.

We declared that each tuple’s history is private to the tuple and no access is allowed beyond the owning component. Therefore, that audit information is not available to external components and does not require encryption.

The Governance section in the system specification freely accesses the audit_log of all programs and augments certain logs without deletion. By definition of the Governance section in Context UNITY, the system operating through the use of governance rules has the necessary credentials for full access to all exposed variables, and thus can perform the decrypt and encrypt functions as needed.

**Audit Generation:** AU-12(2) The information system produces a system-wide audit trail composed of audit records in a standardized format. (See also FAU_STG [4], ECAT-1 [5])

Because this audit control is a system level property, it is facilitated by the Governance section’s access to relevant component variables. The governance rule eventually merges the audit logs of all components into the audit_log variable of the component with the lowest id creating the system-wide audit trail. There are multiple methods by which this could have been achieved. Each component in the system could be required to expose a variable that the Governance section would actively select and ‘command’ one of them to build the audit trail. Another method would be for each component to build the audit trail using a context rule (essentially every component could have a copy of the global log). In all cases, compliance verification requires proof of the following progress property: Among the system components, there exists at least one component whose audit_log eventually contains the audit_log of all components in the system.

5. **DISCUSSION AND CONCLUSION**

In this paper, we extend the notation of Context UNITY to allow for security policy implementation. We build on the foundation of the exposed variable as a shared tuple and introduce policy implementation mechanisms that are declared within the program specification. These extensions, coupled with requirements derived from security controls, allow for verifying the compliance of the system specification. This work lays the foundation for specification constructs which focus on capturing those elements of security policy that need certification. Included in the set of elements needed are not only the functional specification of controls such as access control, but also the configuration specification of other control families that make them more transparent to verification properties.

6. **ACKNOWLEDGMENTS**

This material is based on research supported by the Air Force Office of Scientific Research, under Award No. FA-9550-09-1-0409.

7. **REFERENCES**


