Enforceable and Verifiable Stale-Safe Security Properties in Distributed Systems

JIANWEI NIU AND RAM KRISHNAN AND JARED F. BENNATT AND
RAVI SANDHU AND WILLIAM H. WINSBOROUGH
University of Texas at San Antonio
Department of Computer Science Technical Report CS-TR-2011-02

Attribute staleness arises due to the physical distribution of authorization information, decision and enforcement points. This is a fundamental problem in virtually any secure distributed system in which the management and representation of authorization state are not globally synchronized. This problem is so intrinsic that it is inevitable an access decision will be made based on attribute values that are stale. While it may not be practical to eliminate staleness, we can *limit* unsafe access decisions made based on stale user and object attributes. In this article, we propose two properties and specify a few variations which limit such incorrect access decisions. We use temporal logic to formalize these properties which are suitable to be verified, for example, by using model checking. We present a case study of the uses of these properties in the specific context of an application called Group-Centric Secure Information Sharing (g-SIS). We specify the authorization information, decision and enforcement points of the g-SIS system for the case with only a single user, object, and group (the small enforcement model) in terms of State Machine (SM) and show how these SMs can be designed so as to satisfy the stale-safe security properties. Next, we formally verify that the small model satisfies these properties and enforces a g-SIS authorization policy using the NuSMV model checker. Finally, we show that by generalizing the verification results of the small model that a large enforcement model, comprising an unbounded number of users, objects, and groups, satisfies these properties.

Categories and Subject Descriptors: D.4.6 [**Operating Systems**]: Security and Protection—Access controls; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—Unauthorized access

General Terms: Security, Verification

Additional Key Words and Phrases: Attribute-Based Access Control, Temporal Logic, Model Checking, Security Properties, Stale-Safety

Author's Address: Jianwei Niu (niu@cs.utsa.edu), Jared F. Bennatt (jbennatt@cs.utsa.edu), Ravi Sandhu (ravi.sandhu@utsa.edu) and William H. Winsborough (wwinsborough@acm.org), Institute for Cyber Security and Department of Computer Science, UTSA. Ram Krishnan (ram.krishnan@utsa.edu), Institute for Cyber Security, and Department of Electrical and Computer Engineering, UTSA.
Preliminary version of some of the results reported in this article first appeared in R. Krishnan, J. Niu, R. Sandhu and W. H. Winsborough, "Stale-Safe Security Properties For Group-Based Secure Information Sharing," *ACM Workshop on Formal Methods in Security Engineering (FMSE)*, 2008.
Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 20YY ACM 0000-0000/20YY/0000-0001 $5.00

1. INTRODUCTION

The concept of a stale-safe security property is based on the following intuition. In a distributed system, authoritative information about user and object attributes used for access control is maintained at one or more secure authorization information points. Access control decisions are made by collecting relevant user and object attributes at one or more authorization decision points, and are enforced at one or more authorization enforcement points. Because of the physical distribution of authorization information, decision and enforcement points, and consequent inherent network latencies, it is inevitable that access control will be based on attributes values that are stale (i.e., not the latest and freshest values). In a highly connected high-speed network these latencies may be in milliseconds, so security issues arising out of use of stale attributes can be effectively ignored. In a practical real-world network however, these latencies will more typically be in the range of seconds, minutes and even days and weeks. For example, consider a virtual private overlay network on the internet which may have intermittently disconnected components that remain disconnected for sizable time periods. In such cases, use of stale attributes for access control decisions is a real possibility and has security implications.

We believe that, in general, it is not practical to eliminate the use of stale attributes for access control decisions.[1] In a theoretical sense, some staleness is inherent in the intrinsic limit of network latencies. We are more interested in situations where staleness is at a humanly meaningful scale, say minutes, hours or days. For example, a SAML (Security Assertion Markup Language) assertion produced by an authorization decision point includes a statement of timeliness, i.e., start time and duration for the validity of the assertion. It is up to the access enforcement point to decide whether or not to rely on this assertion or seek a more timely one. Likewise a signed attribute certificate will have an expiry time and an access decision point can decide whether or not to seek updated revocation status from an authorization information point.

Given that the use of stale attributes is inevitable, the question is how do we safely use stale attributes for access control decisions and enforcement? The central contribution of this article is to formalize this notion of safe use of stale attributes. We demonstrate specifications of systems that do and do not satisfy this requirement. We believe that the requirements for *stale-safety* identified in this article represent fundamental security properties the need for which arises in secure distributed systems in which the management and representation of authorization state are not centralized. In this sense, we suggest that we have identified and formalized a *basic security property* of distributed enforcement mechanisms, in a similar sense that non-interference [Goguen and Meseguer 1982] and safety [Harrison et al. 1976] are basic security properties that are desirable in a wide range of secure systems.

[1]Staleness of attributes as known to the authoritative information points due to delays in entry of real-world data is beyond the scope of this article. For example, if an employee is dismissed there may be a lag between the time that action takes effect and when it is recorded in cyberspace. The lag we are concerned with arises when the authoritative information point knows that the employee has been dismissed but at some decision point the employee's status is still showing as active.

Specifically, we present formal specifications of two properties, one strictly stronger than the other. The most basic and fundamental requirement we consider deals with ensuring that while authorization data cannot be propagated instantaneously throughout the system, in many applications it is necessary that a request should be granted only if it can be verified that it was authorized at some point in the recent past. The second, stronger property says that to be granted, the requested action must have been authorized at a point in time after the request and before the action is performed. We believe that the first property, *weak stale-safety*, is a requirement for most actions (*e.g.*, read or write) in distributed access control systems. We also believe that the second property, *strong stale-safety*, is (further) required of some or all actions in many applications. We further show how these two properties can be strengthened to bound the acceptable level of staleness in terms of time elapsed between the point at which the request was last known to have been authorized and the point at which the action is performed.

We show how these properties can be applied in a specific application domain called group-centric secure information sharing (g-SIS) [Krishnan et al. 2009a; 2009b]. We formalize the properties in first-order linear temporal logic (FOTL), as it is a natural choice for supporting unbounded number of users, objects, and groups in an information sharing system. We develop an attribute-based g-SIS enforcement model, each component being represented by state machines (SMs).

The stale-safe properties require an enforcement model to ensure that a requested action is only performed if that action was authorized during a previous refresh of authorization information. Manually determining the satisfaction of FOTL properties can have high complexity. To alleviate this problem, we use model checking [Clarke et al. 1986] to obtain an automated proof for the small g-SIS enforcement model containing one user and one object within a single group.

Finally, we demonstrate that if a policy and stale-safety hold with the small model, they hold for a large system applied to many or unbounded number of users, objects, and groups. This is shown by applying the small enforcement model to every user, object, and group triplet (or tuple). We show formally, that this implies a global policy, *i.e.*the small model policy holds globally.

The remainder of the article is organized as follows. In section 2, we discuss the group-centric secure information sharing problem which will be used throughout the article to illustrate the stale-safe properties. In section 3, we formally specify the stale-safe security properties using FOTL. In section 4, we specify the g-SIS system using SMs and formally verify the stale-safe security properties against the small enforcement model in section 5. In section 6, we extend the verification results for small model to unbounded finite systems. In section 7, we discuss related work and conclude in section 8.

2. A RUNNING EXAMPLE

In this section, we discuss a simple distributed system that will be used in this article as an example to illustrate and study the stale-safety problem. We consider the problem of information sharing amongst a group of authorized people. We refer to this problem as Group-Centric Secure Information Sharing or g-SIS.

2.1 Objectives

We consider the following set of objectives:

—In a group, a user may access an object depending on the temporal order of join/leave and add/remove events. (A user is a representation of a human being in the system and an object represents information). For instance, a user may be allowed to access only those objects added after her join time or all objects regardless of add time. Similarly, after leaving, the user may lose access to all objects or may retain access to objects authorized at leave time. Any variety of such policies may apply (see [Krishnan et al. 2009b] for more examples).

—Group membership is expected to be dynamic. That is, a user may join and leave and an object may be added and removed multiple times.

—A server, called the Control Center (CC), facilitates operation of the system. It maintains attributes of various entities in the system such as membership status of each user and object in each group.

—Authorization decisions can be made offline. That is, for every access attempt, the CC need not be involved for making the access decision. Clearly, an appropriate client-side enforcement mechanism is required to enforce group policy. To this end, we assume a user-side Trusted Reference Monitor (TRM) to enforce group policies in a trustworthy manner. The TRM caches authorization information such as user and object attributes (e.g., user join time, leave time, etc.) locally on the user's access machine (a computer with an appropriate TRM) and refreshes them periodically with the server.

—Objects are made available via super-distribution. In the super-distribution approach, protected group objects (encrypted with a group key for instance) are released into the cloud (cyber space). Users may obtain such objects from the cloud and may access them if authorized. For instance, a user may directly email a group object to another user or transfer it via a USB flash drive. Thus objects need not be downloaded from the CC for each access attempt. A group key is provisioned on users' access machines in such a manner that only a TRM may access the key to encrypt and decrypt group objects. The TRM faithfully enforces group policies based on user and object attributes.

2.2 Enforcement Model for g-SIS

Figure 1 shows one possible enforcement model for g-SIS and illustrates the interaction of various entities in g-SIS. A Group Administrator (GA) controls group membership. The Control Center (CC) maintains authorization information (e.g. attributes of group users and objects) on behalf of the GA.

—*User Join*: Joining a group involves obtaining authorization from the GA followed by obtaining group attributes from the CC. In step 1.1, the user contacts the GA using an access machine that has an appropriate TRM and requests authorization to join the group. The GA authorizes the join in step 1.2 (by setting AUTH to TRUE). The TRM furnishes the authorization to join the group to the CC in step 1.3 and the CC updates the users JoinTS in step 1.4. In step 1.5, the CC verifies GA's authorization and issues the attributes. JoinTS is the timestamp of user join (set to a valid value), LeaveTS is the time at which a user leaves

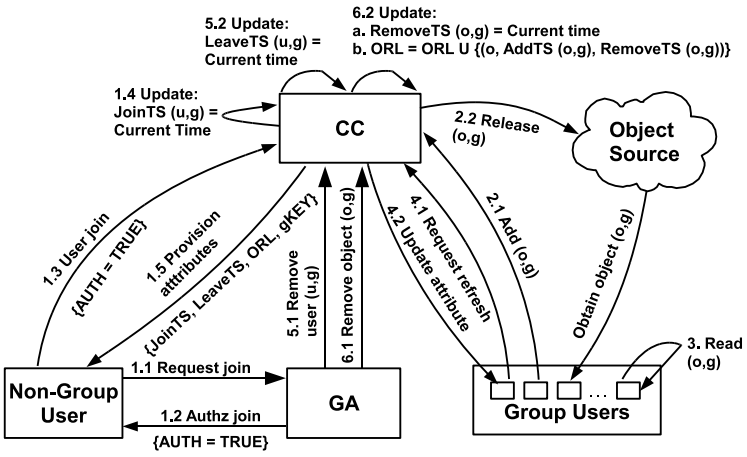

Fig. 1. g-SIS System.

the group (initially set to time of join), gKey is the group key specifying which group objects can be decrypted, ORL is the Object Revocation List which lists the objects removed from the group. We assume that these attributes may be accessed and modified only by the TRM and not by any other entity in the user's access machine.

—*Policy Enforcement*: From here on, the user is considered a group member and may start accessing group objects (encrypted using the group key) as per the group policy and using the attributes obtained from the CC. This is locally mediated and enforced by the TRM. Since objects are available via super-distribution and because of the presence of a TRM on user's access machines, objects may be accessed offline conforming to the policy. For example, the TRM may enforce the policy that the user is allowed to access only those objects that were added after she joined the group and disallow access to objects added before her join time. Further users may lose access to all objects after leaving the group. Such decisions can be made by comparing the join and leave timestamps of user, add and remove timestamps of object. Objects may be added to the group by users by obtaining an add timestamp from the CC. The CC approves the object, sets the AddTS and releases the object into the cloud (steps 2.1 to 2.2). We assume that the AddTS attribute that reflects the time of the last add is embedded in the object itself. However, the RemoveTS cannot be embedded in the object because when the object is removed every copy of the object would have to be found and modified (this is exactly the action taken by a refresh). If this were possible, staleness allowing users to access removed objects would not be possible. Instead, an Object Revocation List (ORL) with elements of the form (o, AddTS, RemoveTS) is provisioned to the access machine. Note that the ORL lists the objects removed from the group and it is required to maintain the triple since objects could be removed and re-added.

—*Attribute Refresh*: Since users may access objects offline, the TRM needs to refresh attributes with the CC periodically (steps 4.1-4.2). How frequently this

is done is a matter of policy and/or practicality. In certain scenarios, frequent refreshes in the order of milliseconds may be feasible while in others refreshes may occur only once a day.

—*Administrative Actions*: The GA may have to remove a user or an object from the group. In step 5.1, the GA instructs the CC to remove a user. The CC in turn marks the user for removal by setting the user's LeaveTS attribute in step 5.2. In the case of object removal, the ORL is updated with the object's AddTS and RemoveTS (steps 6.1-6.2). These attribute updates are communicated to the user's access machine during the refresh steps 4.1 and 4.2.

As one can see, there is a delay in attribute update in the access machine that is defined by the refresh window. Although a user may be removed from the group at the CC, the TRM may let users access objects until a refresh occurs. This is due to attribute staleness that is inherent to any distributed system. We discuss this topic in detail in the subsequent sections. Note that building trusted systems to realize the architecture in figure 1 is well-studied in the literature[2] and is outside the scope of this article. The above system is an instantiation of a more general system in which the policy information point (the CC that maintains authorization information) and policy decision/enforcement point (in this case the TRM) are distributed.

3. STALE-SAFE SECURITY PROPERTIES

As discussed earlier, in a distributed system, access decisions are almost always based on stale attributes which may lead to critical access violations. In practice, eliminating staleness completely may not be feasible. Specifically, the principle of stale-safety is as follows:

PRINCIPLE 3.1 PRINCIPLE OF STALE-SAFETY. *The* principle of stale-safety *states that when it is necessary to rely upon stale authorization information, if the user is granted access an object, the authorization to access that object should have* definitely *held in the recent past.*

In this section, we propose stale-safe security properties following the spirit of this principle. We first discuss a scenario in which stale attributes lead to access violations in g-SIS and informally discuss the stale-safe properties. We formalize them next.

3.1 System Characterization

The g-SIS system consists of users and objects, trusted access machines with TRMs (using which users access objects), a GA and a CC. Access machines maintain a local copy of user attributes which they refresh periodically with the CC. AddTS is part of the object itself. A removed object is listed in the ORL which is provided to access machines as part of refresh. For the purpose of this illustration, we assume that each user is tied to an access machine from which objects are accessed and there is a single GA and single CC per group. We assume a group policy that a user is allowed to access an object as long as both the user and object are current

[2]See related work on trusted computing (http://www.trustedcomputinggroup.org) for example.

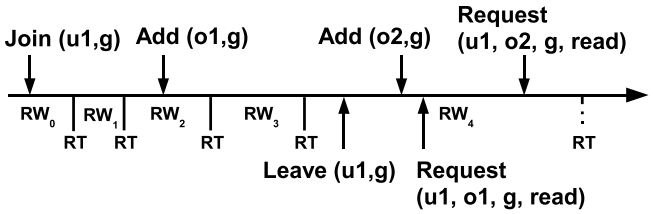

Fig. 2. Allowing u1 to access o2 will violate the Principle of Stale-Safety.

members of the group and the object was added after the user joined the group. Thus the g-SIS system can be characterized as follows:

User attributes	$\{\text{JoinTS}, \text{LeaveTS}\}$
Object attributes	$\{\text{AddTS}, \text{RemoveTS}\}$
Group attributes	$\{\text{gKey}, \text{ORL}\}$
Access Policy	$\text{Authz}_{\text{E}}(u, o, g, op) \rightarrow \text{JoinTS}(u, g) \leq \text{AddTS}(o, g) \wedge$
	$\text{LeaveTS}(u, g) = \text{NULL} \wedge$
	$(o, \text{AddTS}(o, g), \text{RemoveTS}(o, g)) \notin \text{ORL}(g)$

in which u, o, g, and op represent a specific user, object, group, and access operation, respectively. For simplifying the presentation we omit the parameters when it is clear in the context.

Figure 2 shows a timeline of events involving a single group that can lead to access violation due to stale authorization information. User u1 joins the group and the attributes are refreshed with the CC periodically. RT represents the time at which refreshes happen. The time period between any two RT's is a Refresh Window, denoted RW_i. After join, RW_0 is the first window, RW_1 is the next and so on. Suppose RW_4 is the current Refresh Window. Objects o1 and o2 were added to the group by some group user during RW_2 and RW_4 respectively and they are available to u1 via super-distribution. In RW_4, u1 requests access to o1 and o2. A local access decision will be made by the TRM based on the attributes obtained at the latest RT. Thus allowing u1 to read o2 will violate the principle of stale-safety.

Clearly, our access policy allows access to both o1 and o2. However it is possible that u1 was removed by the GA right after the last RT and before Request(u1, o1, read) in RW_4. Ideally, u1 should not be allowed to access both o1 and o2. However note that authorization to access o1 held at the most recent RT while authorization to access o2 *never* held.

Furthermore, from a confidentiality perspective in information sharing, granting u1 access to o1 even if u1 had left the group is relatively less of a problem than granting access to o2. This is because u1 was authorized access to o1 and hence the CC can assume that information has already been released to u1. However, there is never a time that u1 obtained authorization to access o2 and letting u1 access o2 means that u1 may gain knowledge of new information that u1 is not privileged to. This is a critical violation and should not be allowed. In brief, it is OK for a user to access information the TRM can verify they once had authorization to (o1 in this example) but should *not* allow access to objects for which it cannot verify whether or not the user ever had authorization (o2 in this example).

Note that access violation due to staleness illustrated above is not limited to super-distribution. It applies to any enforcement model in which management and representation of authorization information is not centralized. Consider an alternate model in which group objects are obtained from an object distribution server. The distribution server maintains AddTS of objects as they are added to the group. A removed object is listed in the ORL, maintained by the CC. Like earlier, the TRM refreshes attributes periodically with CC. Users may obtain objects from the distribution server anytime and read them if authorized. Clearly, we have the same access violation illustrated earlier.

One of the properties we discuss considers attributes to be stale if it is timestamped later than the last refresh time of the TRM. A stricter property may require the TRM to refresh attributes before granting any access. That is, when u1 requests access to o1, the stricter version of the stale-safe property mandates that the access machine refreshes the user attributes before making an authorization decision.

3.2 Formal Property Specification

In this section we use first-order linear temporal logic (FOTL) to specify stale-safety properties of varying strength. FOTL differs from the familiar propositional linear temporal logic [Manna and Pnueli 1992] by incorporating predicates with parameters, constants, variables, and quantifiers. Temporal logic is a specification language for expressing properties related to a sequence of states in terms of temporal logic operators and logic connectives (e.g., $\wedge$ and $\vee$). The future temporal operator $\Box$ (read henceforth) represents all future states. For example, formula $\Box p$ means that p is true in all future states. The past operators $\ominus$ and $\mathcal{S}$ (read previous and since respectively) have the following semantics. Formula $\ominus p$ means that p was true in the previous state. Note that $\ominus p$ is false in the very first state. Formula $(p \,\mathcal{S}\, q)$ means that q has happened sometime in the past and p has held continuously following the last occurrence of q to the present.

Our formalization uses the following predicates:

request (u,o,g,op)	u requests to perform an action op on o in group g.
Authz (u,o,g,op)	u is authorized to perform an action op on o in g.
join (u,g) and leave (u,g)	Join & leave events of u in g.
add (o,g) and remove (o,g)	Add & remove events of o in g.
perform (u,o,g,op)	u performs op on o in g.
RT (u,g)	TRM contacts CC to synchronize attributes for u.

3.2.1 *Access Policy Specification.* We first specify the example access policy discussed in section 3.1 using FOTL. Note that, in distributed systems such as g-SIS, events such as remove and leave cannot be instantaneously observed by the TRM. Such information (that a user or an object is no longer a group member) can only be obtained from CC at subsequent refresh times (RT's). Thus, we have a notion of ideal or desirable policy that assumes instant propagation of authorization information (like that of a centralized system). This is enforceable only at the CC. However, while designing the TRM (that is, in a distributed setting), one has to re-formulate this ideal policy using available authorization information so that it is enforceable locally by the TRM. We call the former Authz_{CC} and the latter $\text{Authz}_{\text{TRM}}$.

Authz_{CC} below is an FOTL representation of group policy (section 3.1) in a centralized setting. Figure 3 illustrates Authz_{CC}. It states that u is allowed to perform an action op on o if the object (o) was added to the group sometime in the past and both the user (u) and object (o) have not left the group since it was added. Also, the user joined the group prior to the time at which the object was added and has not left ever since. As the name implies Authz_{CC} can be enforced only by the CC and not the TRM. This is because the leave and remove events at CC are not visible to the TRM until the next refresh. When a request is received, the TRM cannot ensure that no leave occurred since join or no remove occurred since add. Authz_{CC} is formally specified as follows:

$$\begin{aligned}&\forall u : \text{U}.\forall o : \text{O}.\forall g : \text{G}.\forall op : \text{P}.\\ &\Box(\text{Authz}_{\text{CC}}(u, o, g, op) \leftrightarrow \quad ((\neg\text{remove}(o, g) \wedge \neg\text{leave}(u, g))\, \mathcal{S}\\ &(\text{add}(o, g) \wedge (\neg\text{leave}(u, g)\, \mathcal{S}\, \text{join}(u, g)))))\end{aligned}$$

Since Authz_{CC} is not enforceable locally at the TRM, we need to formulate another version that is enforceable at the TRM. Recall from section 2 that once a user joins the group, authorization information such as object add time is available instantaneously to the TRM via super-distribution. Thus whether an object add event occurred can be verified by the TRM without contacting the CC. However, verification of whether a user join or leave event or an object remove event occurred can be ascertained only at refresh time (RT) with the CC. $\text{Authz}_{\text{TRM}}$, below, shows the re-formulation of Authz_{CC} as enforceable by the TRM. As shown, the occurrence of join, leave and remove are ascertained at RT. However, add is not subject to this constraint and its occurrence is ascertained independently of RT.

$$\begin{aligned}&\forall u : \text{U}.\forall o : \text{O}.\forall g : \text{G}.\forall op : \text{P}.\\ &\Box(\text{Authz}_{\text{TRM}}(u, o, g, op) \leftrightarrow\ (\neg\text{RT}(u, g)\, \mathcal{S}\, (\text{add}(o, g) \wedge (\neg\text{RT}(u, g)\, \mathcal{S}\, (\text{RT}(u, g) \wedge\\ &(\neg\text{leave}(u, g)\, \mathcal{S}\, \text{join}(u, g)))))) \vee (\neg\text{RT}(u, g)\, \mathcal{S}\, (\text{RT}(u, g) \wedge \text{Authz}_{\text{CC}}(u, o, g, op))))\end{aligned}$$

$\text{Authz}_{\text{TRM}}$ is a disjunction of two cases. The first part addresses the scenario in which the requested object was added after the most recent RT. This is illustrated in case (a) of figure 4 where we are only able to verify that the user was still a member at RT. Since the object was not added prior to that point, we are unable to do a similar check for the object. The second part handles the situation where the object was added before the most recent RT. This is illustrated in case (b) of figure 4 where we are able to verify that at RT both the user and object are current members. Note that in both cases (a) and (b), our evaluation of policy is based on authorization information (except add) available at RT.

Let us introduce formula φ_0 to represent the TRM authorization of u performs an operation op of o in g:

$$\begin{aligned}\varphi_0(u, o, g, op) \equiv \ominus((\neg\text{perform}(u, o, g, op) \wedge (\neg\text{RT} \vee (\text{RT} \wedge \text{Authz}_{\text{CC}}(u, o, g, op))))\, \mathcal{S}\\ (\text{Authz}_{\text{TRM}}(u, o, g, op) \wedge (\neg\text{request}(u, o, g, op) \wedge\\ \neg\text{perform}(u, o, g, op))\, \mathcal{S}\, \text{request}(u, o, g, op)))\end{aligned}$$

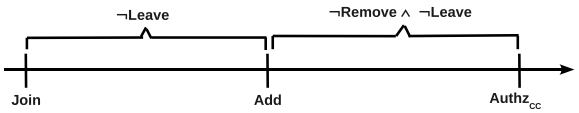

Fig. 3. Ideal Access Policy (Authz_{CC}).

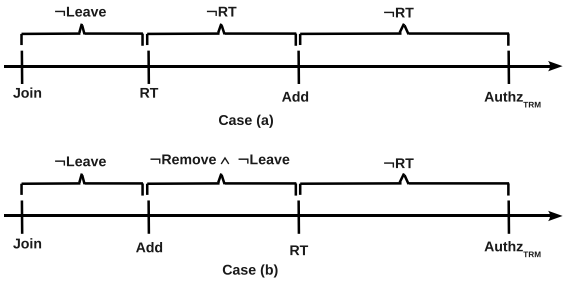

Fig. 4. Approximate Access Policy ($\text{Authz}_{\text{TRM}}$).

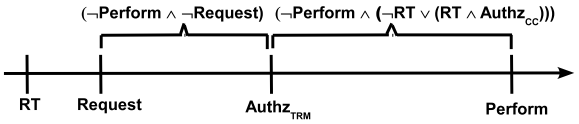

Fig. 5. Formula φ_0.

Figure 5 is a pictorial representation of formula φ_0. It illustrates how a TRM traditionally reacts to a request to access an object. When a request arrives from a user, the TRM subsequently verifies if the policy ($\text{Authz}_{\text{TRM}}$) holds. If successful, the TRM allows the user to perform the requested action. Note that if a refresh occurs in the meantime, the TRM re-evaluates the policy (at the time of refresh it is feasible to verify Authz_{CC}) with updated attributes. Thus, φ_0 states that the operation was authorized sometime between request and perform. Clearly, formula $\forall u : \text{U}.\forall o : \text{O}.\forall g : \text{G}.\forall op : \text{P}.\ \Box\ (\text{perform}(u, o, g, op) \rightarrow \varphi_0(u, o, g, op))$ reflects this behavior of the TRM.

In contrast, formula $\forall u : \text{U}.\forall o : \text{O}.\forall g : \text{G}.\forall op : \text{P}.\ \Box\ (\text{perform}(u, o, g, op) \rightarrow \varphi_0'(u, o, g, op))$ is not enforceable as argued earlier since Authz_{CC} cannot be locally verified by the TRM at request time.

$$\begin{aligned}\varphi_0'(u, o, g, op) \equiv \ominus((\neg\text{perform}(u, o, g, op) \wedge (\neg\text{RT}(u, g) \vee (\text{RT}(u, g)\\ \wedge\ \text{Authz}_{\text{CC}}(u, o, g, op)))) \,\mathcal{S}\, (\text{Authz}_{\text{CC}}(u, o, g, op) \wedge\\ (\neg\text{request}(u, o, g, op) \wedge \neg\text{perform}(u, o, g, op)) \,\mathcal{S}\, \text{request}(u, o, g, op)))\end{aligned}$$

However, observe that verifying if $\text{Authz}_{\text{TRM}}$ holds at the time of request in φ_0 will allow the user to access objects that were added after RT in figure 5. We illustrated this in case (a) of figure 4. As discussed earlier, it is unsafe to let users access these objects before a refresh can confirm the validity of their group membership.

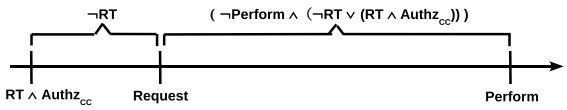

Fig. 6. Formula φ_1.

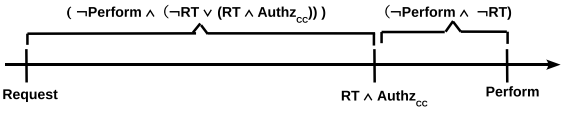

Fig. 7. Formula φ_2.

Next, we specify two stale-safe security properties of varying strength. The weakest of the properties we specify requires that a requested action be performed only if a refresh of authorization information has shown that the action was authorized at that time. This refresh is permitted to have taken place either before or after the request was made. The last refresh must have indicated that the action was authorized and all refreshes performed since the request, if any, must also have indicated the action was authorized. This is the *weak stale-safe security property*. By contrast, the *strong stale-safe security property* requires that the confirmation of authorization occur after the request and before the action is performed.

3.2.2 *Weak Stale-safe Security Property.* Let us introduce two formulas formalizing pieces of stale-safe security properties. Intuitively, φ_1 can be satisfied only if authorization was confirmed prior to the request being made. On the other hand, φ_2 can be satisfied only if authorization was confirmed after the request. Note that weak stale safety is satisfied if either of these is satisfied prior to a requested action being performed.

$$\begin{aligned}
\varphi_1(u,o,g,op) \equiv & \ominus((\neg\mathrm{perform}(u,o,g,op) \wedge (\neg\mathrm{RT}(u,g) \vee (\mathrm{RT}(u,g) \wedge \\
& \mathrm{Authz_{CC}}(u,o,g,op)))) \,\mathcal{S}\, (\mathrm{request}(u,o,g,op) \wedge \\
& (\neg\mathrm{RT}(u,g) \,\mathcal{S}\, (\mathrm{RT}(u,g) \wedge \mathrm{Authz_{CC}}(u,o,g,op)))))\\
\varphi_2(u,o,g,op) \equiv & \ominus((\neg\mathrm{perform}(u,o,g,op) \wedge \neg\mathrm{RT}(u,g)) \,\mathcal{S}\, (\mathrm{RT}(u,g) \wedge \\
& \mathrm{Authz_{CC}}(u,o,g,op) \wedge ((\neg\mathrm{perform}(u,o,g,op) \wedge (\neg\mathrm{RT}(u,g) \vee \\
& (\mathrm{RT}(u,g) \wedge \mathrm{Authz_{CC}}(u,o,g,op)))) \,\mathcal{S}\, \mathrm{request}(u,o,g,op))))
\end{aligned}$$

Figure 6 illustrates formula φ_1. φ_1 says that prior to the current state, the operation has not been performed since it was requested. Also since it was requested, any refreshes that may have occurred indicated that the operation was authorized $(\neg\mathrm{RT} \vee (\mathrm{RT} \wedge \mathrm{Authz_{CC}}))$. Finally, a refresh must have occurred prior to the request and the last time a refresh was performed prior to the request, the operation was authorized.

Observe that formula φ_1 mainly differs from φ_0 on the point at which $\mathrm{Authz_{CC}}$ is evaluated. Referring to figure 6, evaluating $\mathrm{Authz_{CC}}$ at the latest RT guarantees that requests to access any object that may be added during the following refresh window will be denied.

Note that φ_1 is satisfied if there is no refresh between the request and the perform. It requires that any refresh that happens to occur during that interval indicate that the action remains authorized. In our g-SIS application, this could preclude an action being performed, for instance, if the user leaves the group, a refresh occurs, indicating that the action is not authorized, the user rejoins the group, and another refresh indicates that the action is again authorized. For some applications, this might be considered unnecessarily strict.

Figure 7 illustrates formula φ_2. φ_2 does not require that there was a refresh prior to the request. Instead it requires that a refresh occurred between the request and now. It further requires that the operation has not been performed since it was requested and that every time a refresh has occurred since the request, the operation was authorized.

Note that φ_2 can be satisfied without an authorizing refresh having occurred prior to the request, whereas φ_1 cannot. Thus, though φ_2 ensures fresher information is used to make access decisions, it does not logically entail φ_1 as it is satisfied by traces that do not satisfy φ_1.

Thus the formula $\text{perform}(u, o, g, op) \rightarrow \varphi_1(u, o, g, op)$ requires that a confirmation of authorization occur after the request has been received. The formula $\text{perform}(u, o, g, op) \rightarrow \varphi_2(u, o, g, op)$ requires that confirmation of authorization is obtained after the request, before the action is performed.

Definition 3.2 Weak stale safety. *An enforcement model has the* weak stale-safe security property *if it satisfies the following FOTL formula:*

$$\forall u : \text{U}.\forall o : \text{O}.\forall g : \text{G}.\forall op : \text{P}.\Box(\text{perform}(u, o, g, op) \rightarrow (\varphi_1(u, o, g, op) \vee \varphi_2(u, o, g, op)))$$

3.2.3 *Strong Stale-safe Security Property.* This property is strictly stronger than weak stale safety. For this reason, and because, unlike weak stale safety, it is a reasonable requirement for higher assurance systems, we give it a second name.

Definition 3.3 Strong stale safety. *An enforcement model has the* strong stale-safe security property *if it satisfies the following FOTL formula:*

$$\forall u : \text{U}.\forall o : \text{O}.\forall g : \text{G}.\forall op : \text{P}.\Box(\text{perform}(u, o, g, op) \rightarrow \varphi_2(u, o, g, op))$$

Note that the formulas (φ_0, φ_1 and φ_2) were concerned about the occurrence of RT with respect to the time at which the request came from the user, the time at which the requested action is performed and the verification if authorization held. This separation of request and perform is important to differentiate weak-stale safety from strong-stale safety property. In weak-stale safety, although authorization held at RT prior to request, it is possible for an RT to occur between request and perform. If fresh attributes are available, it is important to re-check if authorization holds in light of this update. Formula φ_1 requires that authorization continue to hold at such occurrences. On the other hand, strong-stale safety mandates that after request, the action cannot be performed until authorization is verified with up-to-date attributes.

We have shown that the weak and strong properties can be extended to express requirements that bound acceptable elapsed time between the point at which attribute refresh occurs and the point at which a requested action is performed. In addition, we are able to demonstrate that the stale-safety properties that we have

defined takes an authorization policy that is a safety property [Lamport 1977; 1985; Alpern and Schneider 1987] and specifies the right conditions under which it may hold in a distributed system in which authorization information may potentially be stale. See appendixes A and B for a detailed discussion of timely stale-safety and stale-safety versus safety properties.

3.3 Stale-safe Systems

We discuss the significance of the weak and strong stale-safe properties in the context of stale-safe systems designed for confidentiality or integrity. Confidentiality is concerned about information release while integrity is concerned about information modification. Both weak and strong properties are applicable to confidentiality; the main trade-off between weak and strong here is usability. Weak allows authorization decisions to be made using stale attributes while strong forces an attribute refresh before a decision can be made. Depending on the security and functional requirements of the system under consideration, the designer has the flexibility to choose between weak and strong to achieve stale-safety. In the case of integrity, the weak property can be risky in many circumstances; the strong property is more desirable. This is because objects modified by unauthorized users may be used/consumed by other users before the modification can be undone by the server. For instance, in g-SIS, a malicious unauthorized user (i.e. a malicious user who has been revoked group membership but is still allowed to modify objects for a time period due to stale attributes) may inject bad code and share it with the group. Other unsuspecting users who may have the privilege to execute this code may do so and cause significant damage to the system. In another scenario, a malicious user may inject incorrect information into the group and other users may perform certain critical actions based on such information. Thus, although both weak and strong properties may be applicable to confidentiality and integrity, the weak property may not be sufficient for integrity.

4. FORMAL SPECIFICATION OF ENFORCEMENT MODEL

In this section, we develop a collection of attribute-based g-SIS enforcement models according to the architecture illustrated in figure 1. A small enforcement model is specified as a composite extended-state-machine (ESM), describing how the four components, GA, CC, UI, and TRM, interact to enforce the g-SIS policy with respect to a single user, a single object, within a single group. Next, we parameterize the ESM for a given user, object, and group and show that many instantiations of this parameterized ESM can be composed to create an enforcement model for an unbounded finite number of users, objects, and groups.

4.1 Modeling Notation

We use hierarchical transition systems (HTSs) [Niu et al. 2003; Niu 2005] to specify the g-SIS enforcement model's four components and their interactions. There are two primary reasons for choosing HTSs to describe the behavior of the enforcement mechanism. First, it is an expressive yet intuitive ESM, supporting control state hierarchy, conditional transition, and a rich set of composition operators. Thus, an HTS control state can represent a collection of system states (i.e., automaton states), supporting a higher-level of abstractions. Second, the semantics of HTS

is formally defined to enable the automation of the proof using model checking. Third, the average programmer should be able to implement a model based on the HTS specification.

An HTS is an ESM that consists of transitions and a hierarchical set of control states, given by $\langle S, S^H, S^0, S^F, E, V, V^0, T\rangle$. S is a set of control states. Each control state $s \in S$ is either a basic state or a super state that contains other states. S^H is the state hierarchy, which defines a partial ordering on states, with the basic states as maximal elements and root state (a state not contained in any other state) of an HTS as the minimal element. Each super state must have a default state. $S^0 \subseteq S$ is the non-empty set of initial states. $S^F \subseteq S$ is the set of final basic states. E is a finite set of events, consisting of input and generated events. V is a finite set of typed variables. V^0 is a predicate describing the initial values of variables. T is a finite set of transitions. A transition will transform the system from the source state to the destination state. A transition label may include optional elements such as a triggering event, a condition on the triggering event, and actions, which can be assignments to typed variables or event generation.

The semantics of an HTS is formally defined using customizable semantic templates [Niu et al. 2003]. In this paper, we adopt the following semantics. A transition executes only if it is enabled: the HTS is currently in the source state; the positive triggering events happen; the negative events are absent; and the condition evaluates to True. If more than one transition is enabled, the transition whose source state has the highest rank (innermost) is chosen to execute first (rank is the distance of the root from the state in question). If multiple transitions, whose source states have the same rank, are enabled, the ones labeled with triggering events have higher priority. A state may have an *entry* transition, which executes first whenever the state is entered. The event generation and variable update take into effect in the next state due to the execution of a transition's action.

For instance, in figure 8 (discussed in detail later) the TRM_Unsafe HTS, for any user **u**, any object **o**, in any group **g**, has three basic states: *ready*, *authorized*, and *refreshed*. (In the following, we often skip the parameters of states, events, and variables, when they are obvious from the context.) Initially, the HTS is in the *ready* state and variable *pendResponse* is set to False. If event *request* occurs and event *refresh* does not occur (represented as negated event): in the case that condition Authz_E evaluates to False, the loop transition executes and generates event *fail* sent to the UI HTS; in the case that Authz_E evaluates to True, the leftmost transition executes and its action sets variable *pendResponse* to True. Authz_E represents Authz_TRM in terms of the attributes maintained by the TRM_Unsafe HTS, and is defined as below.

$$\text{Authz}_\text{E}(u, o, g, op) \equiv (\text{JoinTS}(u, g) \leq \text{AddTS}(o, g)) \wedge (\text{LeaveTS}(u, g) \leq \text{JoinTS}) \wedge (o, \text{AddTS}(o, g), \text{RemoveTS}(o, g)) \notin \text{ORL}(g)$$

In the case when the HTS is in the *refreshed* state, the loop transition will have priority over all the other transitions with that state as the source state because the loop transition is labeled with event *refresh* and the others are not.

A specification of a distributed system is a hierarchical composition of HTSs. Concurrency, synchronization, and communication are introduced via composition operators, such as parallel, rendezvous, and interrupt. The composition operator,

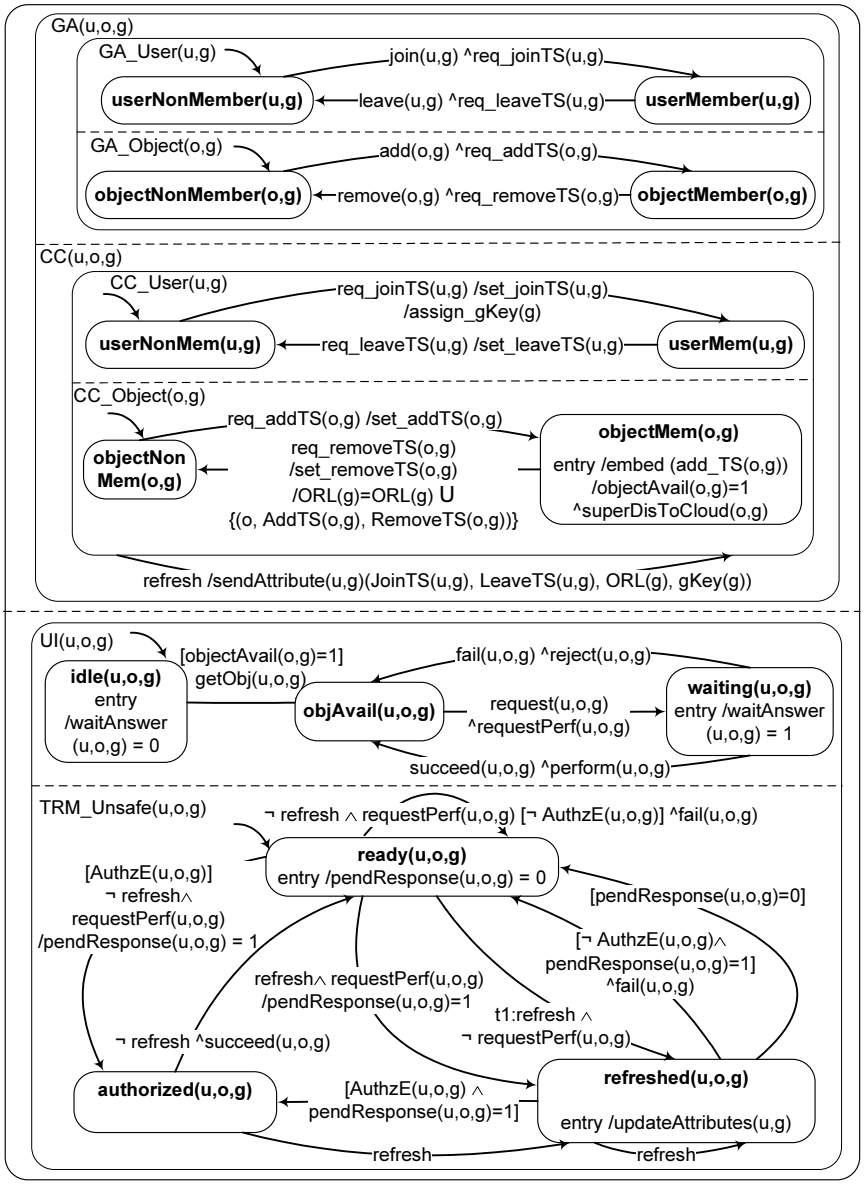

Fig. 8. HTS specification of the g-SIS architecture. The TRM machine, TRM_Unsafe, is not stale-safe.

| *op* |, controls which HTSs execute and how the HTSs share data (e.g., generated events being made available to other HTSs). In the parallel composition of two HTSs, both execute if they are enabled simultaneously. Otherwise, the enabled one executes in isolation. In the rendezvous composition, exactly one transition in the sending HTS generates a rendezvous event that triggers exactly one transition in the receiving HTS, and both transitions execute together. If only one HTS is en-

abled by (or generates) a synchronization event in the same step, then the first HTS is forced to wait until the other one is ready. Otherwise, the behavior of the HTSs is interleaved, only one HTS executing a transition that does not involve rendezvous events. Composition operators can be nested, a component in the composition can be a single HTS or composed HTSs (CHTS). The interrupt composition allows control to pass between components via a set of interrupt transitions. In the interrupt composition, either transitions in the source, transitions in the destination, or an interrupt transition can be chosen to execute. In the context of this article, it is always the case that the interrupt transition has the lowest priority. For instance, the the transition triggered by the event *refresh* in the CC machine has lower priority than inner transitions–that is the *refresh* event occurs *after* requesting and setting the timestamps.

4.2 Formal g-SIS Enforcement Model

We consider a **parameterized** enforcement model, which can be instantiated by any user **u**, any object **o**, and any group **g** (see figure 8) in the domains of $\mathcal{U}$, $\mathcal{O}$, and $\mathcal{G}$ respectively. For simplicity, we skip the parameters of HTS, states, events, and variables in the following presentation. The enforcement model is specified as four CHTSs, consisting of GA, CC, UI, and TRM. The GA is modeled as a parallel composition of two HTSs: GA_User, which models the behavior of a user joining and leaving a group, and GA_object, which models the behavior of an object being added and removed from the group. Without loss of generality, we omit the two step join process discussed in figure 1 and let the GA join a user to the group by sending a request directly to the CC. Furthermore, we assume that only the GA may add/remove an object to/from a group.[3]

The CC is modeled as a CHTS containing two parallel HTSs, CC_User and CC_object, and an interrupt transition to respond to the *refresh* event. This transition has lower priority than any transitions in the CC CHTS. When the GA authorizes a user to join or leave the group, a request of timestamp event is generated. This event causes the CC_User machine to set the appropriate timestamp (JoinTS or LeaveTS). CC_Object and GA_Object behave similarly. The GA and CC are composed using the rendezvous composition: the GA_User and CC_User are synchronized on the *req_joinTS* and *req_leaveTS* events; the GA_object and CC_object are synchronized on the *req_addTS* and *req_removeTS* events.

Consider the interaction between GA_User and CC_User machines in figure 8 first. In GA_User, when event *join*, from a user, occurs in *userNonMember* state, a *req_joinTS* event is generated and *userMember* state is entered. The *req_joinTS* event is captured by the CC_User machine; the *JoinTS* and *gKey* are set for the user. The timestamp is obtained from a global Clock (meaning there is only a single Clock that is *not* parameterized). The Clock machine (not shown here) models a clock by monotonically incrementing a tick variable which is used by the CC to set the timestamps. Similarly, when event *leave* occurs in the *userMember* state in GA_User, a *req_leaveTS* event is generated and *userNonMember* state is entered and the *LeaveTS* is subsequently set by CC_User. Note that the user's membership

[3]Note that none of these changes affect stale-safety verification since the properties are only concerned with authorization synchronization between CC and TRM.

state is always synchronized between the GA and CC machines. As shown, the GA_Object machine handles object membership states and behaves similarly to the GA_User machine. However, in the case of CC_Object, when *req_addTS* is received in the *objectNonMem* state from GA_Object, *AddTS* is set and *objectMem* state is entered. Once this state is entered, the object o's *AddTS* is embedded in the object itself and thus the TRM can obtain it directly from the object when an access decision needs to be made. The *superDisToCloud* event models the fact that the protected object is distributed into the cloud. When *req_removeTS* is received from GA_User, the RemoveTS is set, the ORL is updated and CC_Object enters *objectNonMem* state.

The TRM machine TRM_Unsafe[4] and UI are also composed using the rendezvous synchronization on events *requestPerf, succeed* and *fail*. The user machine UI represents the interface to a user in the system. UI can obtain an object from the cloud, represented by event *getObj*. If a user's request to perform an action on an object is received, represented by event *request*, such as to read an object, UI sends a *requestPerf* event to TRM_Unsafe and moves to the *waiting* state. TRM_Unsafe decides whether the perform is allowed (event *succeed*) or denied (event *fail*) and accordingly the user may or may not be allowed to perform the requested action.

The TRM_Unsafe machine maintains local copies of CC attributes, including *JoinTS, LeaveTS, AddTS, RemoveTS* and refresh timestamp, *RT*. It has three states. In the *ready* state, the machine responds to event *requestPerf* from the UI machine and event *refresh*. The *authorized* state represents the fact that the requested action is authorized to be performed (i.e., Authz_E is True). The *refreshed* state is entered whenever a refresh occurs, representing the fact that the TRM's copy of attributes is updated. If a *requestPerf* is received and Authz_E is not satisfied, a *fail* response is sent back to UI. But if Authz_E holds, the machine enters the *authorized* state and generates a *succeed* event and transitions back to the *ready* state. As shown, the *succeed* event is captured by UI which subsequently allows the user to perform the requested action. However, if a *refresh* occurs in the *authorized* state, TRM_Unsafe transitions to the *refreshed* state, updates the attributes and verifies if Authz_E holds in light of the updated attribute values. This ensures that if the machine is in *authorized* state and new attribute values are available, Authz_E is re-evaluated based on the new attribute values. Note that the variable *pendResponse* keeps track of whether a request has been processed or not. The enforcement model represented as a composition of HTSs in this section is similar to the security automaton in Schneider's framework [Schneider 2000]. See [Niu et al. 2011] for a detailed discussion.

Recall that a parameterized g-SIS enforcement specified in HTSs enables us to model a secure information system of unbounded numbers of groups, users, and objects. Determining the satisfaction of a g-SIS policy and stale-safe security properties in FOTL for the enforcement model can have high complexity. To manage the complexity, we will first consider a small finite enforcement model, each of the domains $\mathcal{U}$, $\mathcal{O}$, and $\mathcal{G}$ containing one element only. Next, we generalize the results obtained for the small model to a large model, consisting of an unbounded instances (one for each $\langle u, o, g\rangle$ tuple) of the enforcement model, such as the one shown in

[4]This machine is named so because it is not stale-safe, as will be shown later.

figure 8.

5. MODEL CHECKING STALE-SAFETY

A small enforcement model consists of a single instance of the CHTS, such as the one in figure 8. In this section we explain the mapping from CHTS to SMV modules. The FOTL stale-safe properties then can be converted into propositional LTL formulas that NuSMV can handle. Finally, we use model checking tool NuSMV to verify whether small enforcement models enforce the g-SIS policy and to demonstrate whether they are weak or strong stale-safe.

5.1 Representing Small Enforcement Model using NuSMV Modules

We can represent the FOTL stale-safe properties into propositional LTL formulas by limiting ourselves to small domains, each containing only one element. Still, it can be hard or impossible to reason about stale safety even regarding the small enforcement model by manual means due to the complexity exhibited by its behavior. To alleviate this problem, we base our verification on proofs obtained by model checking a small g-SIS enforcement model. Model checking is an automated verification technique to verify finite systems, usually represented by finite state machines (FSMs). Model checking exhaustively explores the state space based on the transition relation of a finite system to determine if a given property, usually expressed in temporal logic, holds. If not, the model checker generates a counterexample trace showing why the property in question fails. We choose to use the open-source model checker called NuSMV [Cimatti et al. 2002] to verify stale safety of the small enforcement model because it supports past temporal operators and it is a BDD-based, highly-optimized model checking tool that can handle a relatively large state space.

We briefly describe NuSMV by focusing on the features that we use. It accepts FSMs encoded in the SMV language. States are given by assignments of values to variables, which must be of finite type. Variables are declared in the *VAR* section using the syntax $\langle variable \rangle : \langle type \rangle$ where *variable* is an identifier that refers to the name of a data structure and *type* refers to a finite type such as a boolean, enumerated type, range of integers, or SMV module. The transition relation is defined in the *ASSIGN* section. The initial states are defined by using *init* statements of the form $init(x) := EXP$, which defines the value or set of values x can take on initially. Transition relation is represented by using a collection of *next* statements of the form $next(x) := EXP$, which defines the value or set of values that x can assume in the following state (by taking the transition in the current state). NuSMV supports derived variables, which are essentially macros that expand to expressions over state variables. Derived variables are defined by using assignment statements of the form $x := EXP$ and are not explicitly represented in any state. Instead, variable x is replaced by expression EXP, which refers to state variables. A NuSMV model consists of a main *MODULE* and a set of other MODULEs, each containing a set of VAR, ASSIGN, and DEFINE blocks. Below, we show how to translate the small model into SMV modules.

Component Mapping. All of the HTSs except GA_User/Object are represented as Modules in the NuSMV language (GA_User/Object HTSs are modeled inside of the CC_User/Object modules). The GA_User/Object machines, are condensed into

the CC_User/Object modules. In the machines it is shown that the GA triggers a request action (to join, leave, add, or remove) which initiates a set-timestamp action in the CC. In modeling, we have the CC_User/Object do all of the requesting. Well-formedness constraints are part of the specification so that, for instance, a request join does not occur for a member that is already joined. The CC reacts to events generated by the CC_User/Object, setting the timestamps appropriately after each generated event. The TRM keeps track of the value of authzTRM through random refreshes and super distribution. Finally, the UI is the user interface which generates random requests to access an object from the given TRM. The TRM notifies the UI after each request with a *succeed* or $fail$ signal based on the value of authzTRM. For a more detailed, code specific, description of the NuSMV model see appendix D.

Representation of Composition. We use the parameter passing mechanism among NuSMV MODULES to represent the composition of enforcement components. The following code listing shows the main MODULE.

```
1 MODULE main
2    VAR
3        user: CC_User();
4        object: CC_Object();
5        cc: CC(user, object);
6        trm: TRM(cc, ui);
7        ui: UI(trm);
```

The main MODULE consists of instances of five other MODULEs, CC, TRM, UI, CC_Object, and CC_User. The CC_Object and CC_User MODULEs represent the behavior of GA. CC MODULE takes CC_User and CC_Object as parameters to represent the synchronization between GA and CC. The TRM MODULE takes the CC and UI as input to represent the concurrent execution and synchronization among them.

We do not model the ORL because it isn't a finite type; even with a single object, it grows each time the object is removed. Also the policy we enforce only depends on the latest remove and add events so it's unnecessary to keep track of the complete history.

When comparing two timestamps for two events, the information we want to gain is which event happened first, not necessarily the specific time at which each event occurred. Thus for any set of timestamps we can immediately convert them into an ordered list. There are four distinct user/object events and therefore we need at least four values for each of those timestamps (to correctly order them). The CC holds and maintains such timestamps. The TRMs do not hold timestamps, rather the evaluation of Authz_{E} is abstracted based on values from the CC during refresh and the user's membership state during super distribution.

We adapt ideas presented in [Tsuchiya and Schiper 2007] to model these ordered timestamps. A NuSMV model can be defined using INVAR and TRANS definitions which impose invariants and transition relations. There are two TRANS and two INVAR definitions, for a total of four conditions. The two transition relations ensure that: (1) the order of timestamps is preserved between consecutive states among timestamps' whose event is not triggered in the next state, and (2) if a timestamp

is set in the most recent state, then it's value is N (the largest possible value). The two invariants ensure that the smallest timestamp has value 0 and that the values are consecutive, *i.e.*if $ts_i = 2$ then $\exists j : ts_j = 1$. The 1st TRANS definition ensures that every evaluation of equality and inequality is consistent between the actual timestamps and the ordered timestamps. The invariant part for the ordered timestamps is what makes the their values different from the actual timestamps'.

5.2 Unsafe Enforcement Model

We denote the enforcement system presented in figure 8 as Δ_0-system.

DEFINITION 5.1 Δ_0-SYSTEM. *Let Δ_0 represent a g-SIS enforcement system defined as below:*

$$\begin{aligned}\Delta_0(u,o,g) \equiv &(GA(u,o,g) \mid rendez \mid CC(u,o,g)) \mid parallel \mid \\ &(UI(u,o,g) \mid rendez \mid \text{TRM\_Unsafe}(u,o,g))\end{aligned}$$

In which, GA and CC, and UI and TRM_Unsafe rendezvous (rendez) respectively, and then are composed via parallel composition. Both the CC and GA have two parallel HTSs.

An unbounded finite (UF) enforcement model contains any number of instances of a Δ-system (Δ_0, Δ_1, or Δ_2), ranging over all users, objects, and groups, executing in parallel, formally,

$$\Delta_i^{UF} \equiv \mathop{||}_{u \in \text{U}.o \in \text{O}.g \in G} \Delta_i(u,o,g), 0 \leq i \leq 2$$

In the following sub-sections we present several theorems concerning the behavior of each of the three Δ systems. In all cases, we only model checked the small carrier case. We later show, in Section 6, that this is sufficient for proving the unbounded finite system also models the FOTL formulas presented. For example, we proved through model checking the following formula for the Δ_0 system:

$$\Delta_0(u,o,g) \vDash \Box(\text{perform}(u,o,g) \rightarrow \varphi_0(u,o,g))$$

Using Theorem 6.1 in Section 6, we show that this implies that the unbounded finite system also models this formula for each of its parameters:

$$\Delta_0^{UF} \vDash \forall u : \text{U}.\forall o : \text{O}.\forall g : \text{G}.\Box(\text{perform}(u,o,g) \rightarrow \varphi_0(u,o,g))$$

For brevity, this is not explicitly stated below, but the process for arriving at each of the following theorems relies on this implication from the results of model checking the small carrier system.

THEOREM 5.2 UNENFORCEABLITY THEOREM. Authz_{CC} *is not enforced by the Δ_0^{UF}-system. That is:*

$$\Delta_0^{UF} \nvDash \forall u : \text{U}.\forall o : \text{O}.\forall g : \text{G}.\Box(\text{perform}(u,o,g) \rightarrow \varphi_0'(u,o,g))$$

This theorem states that TRM_Unsafe does not enforce Authz_{CC}. This is self-evident because, as discussed earlier, certain events such as user leave and object remove occurring at CC are not immediately visible to TRM_Unsafe (they are visible only at refresh times). Thus there will be instances in which TRM_Unsafe will allow a user to perform even when Authz_{CC} is False. The model checker confirms this by generating a counter-example as explained in appendix D.4.1.

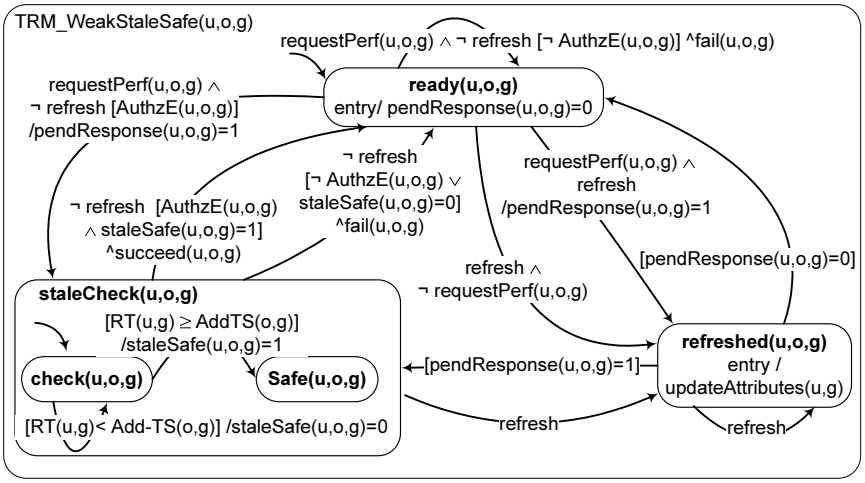

Fig. 9. TRM_WeakStaleSafe: This TRM machine satisfies weak stale-safety.

THEOREM 5.3 ENFORCEABILITY THEOREM. *The* Δ_0^{UF}*-system enforces* $\text{Authz}_{\text{TRM}}$. *That is:*

$$\Delta_0^{UF} \vDash \forall u : \text{U}.\forall o : \text{O}.\forall g : \text{G}.\Box(\text{perform}(u, o, g) \rightarrow \varphi_0(u, o, g))$$

This theorem states that $\text{Authz}_{\text{TRM}}$ is enforceable by the Δ_0-system. Recall that $\Box(\text{perform} \rightarrow \varphi_0)$ ensures that the user can perform if $\text{Authz}_{\text{TRM}}$ is satisfied (section 3.2.1).

THEOREM 5.4 WEAK UNSAFE TRM THEOREM. *The* Δ_0*-system does not satisfy the Weak Stale-Safety property. That is:*

$$\Delta_0^{UF} \nvDash \forall u : \text{U}.\forall o : \text{O}.\forall g : \text{G}.\Box(\text{perform}(u, o, g) \rightarrow (\varphi_1(u, o, g) \vee \varphi_2(u, o, g)))$$

This theorem states that the Δ_0-system is not stale-safe. Specifically, it fails the weak stale-safety security property. Recall that this was illustrated earlier in figure 2. The counter-example generated by NuSMV confirms this theorem.

COROLLARY 5.5 STRONG UNSAFE TRM THEOREM. *The* Δ_0*-system does not satisfy the Strong Stale-Safety property. That is:*

$$\Delta_0^{UF} \nvDash \forall u : \text{U}.\forall o : \text{O}.\forall g : \text{G}.\Box(\text{perform}(u, o, g) \rightarrow \varphi_2(u, o, g))$$

Evidently, if Δ_0-system fails the weak property, it would also fail the strong property. The same counter-example generated by NuSMV for theorem 5.4 applies here.

5.3 Weak Stale-Safe TRM

Figure 9 shows one possible design of a TRM that satisfies weak stale-safety. This TRM machine, TRM_WeakStaleSafe, has a super-state called *staleCheck* that ensures that any authorization decision made is weak stale-safe. When the request is received, the machine first checks if Authz_{E} holds based on local attributes of the TRM. If Authz_{E} holds, it enters a super state called *staleCheck* and verifies if

the decision is weak stale-safe. This is achieved by verifying that the object that is being requested access to was added before the most recent refresh time—that is, RT $\geq$ AddTS. Here RT is a variable that maintains the timestampe for the most recent refresh event. This ensures that any object that is received via super-distribution which was added after the TRM's last refresh time is unsafe to access. As illustrated in figure 2, the user could potentially leave the group between the most recent RT and the object add time but the TRM is not aware of this until the next refresh. This could result in a situation where a user is granted access to an object to which he/she was never authorized. We now prove that the Δ_1-system satisfies Weak Stale-Safety property.

DEFINITION 5.6 Δ_1-SYSTEM. *Let Δ_1 represent a g-SIS enforcement system defined as below.*

$$\begin{aligned}\Delta_1(u,o,g) \equiv &(GA(u,o,g) \mid rendez \mid CC(u,o,g)) \mid parallel \mid \\ &(UI(u,o,g) \mid rendez \mid \text{TRM\_WeakStaleSafe}(u,o,g))\end{aligned}$$

where GA and CC, and UI and TRM_WeakStaleSafe *rendezvous respectively, and then execute concurrently via parallel composition (note that GA, CC, and UI are the same CHTSs as in Figure 8).*

THEOREM 5.7 WEAK STALE-SAFE TRM THEOREM. *Δ_1^{UF} satisfies the Weak Stale-Safe Security Property.*

$$\Delta_1^{UF} \vDash \forall u : \text{U}.\forall o : \text{O}.\forall g : \text{G}.\Box(\text{perform}(u,o,g) \rightarrow (\varphi_1(u,o,g) \vee \varphi_2(u,o,g)))$$

NuSMV successfully verifies that the Δ_1-system satisfies the weak stale-safe security property. Obviously, the Δ_1-system does not satisfy the strong stale-safety property.

$$\Delta_1^{UF} \nvDash \forall u : \text{U}.\forall o : \text{O}.\forall g : \text{G}.\Box(\text{perform}(u,o,g) \rightarrow \varphi_2(u,o,g))$$

As expected, NuSMV generates a counter-example showing that the Δ_1-system does not satisfy strong stale-safety. The explanation of the generated counter-example is given in D.5.2.

5.4 Strong Stale-Safe TRM

Figure 10 shows a straight forward way to satisfy strong stale-safety. Note that TRM_StrongStaleSafe refreshes the attributes with CC every time a request from the user is received. If Authz_E holds after refresh, the user is allowed to perform, else the request is rejected. Clearly, this should satisfy strong stale-safety because formula φ_2 requires that a refresh be performed after request before verifying if Authz_E holds, which is consistent with the model in figure 10.

DEFINITION 5.8 Δ_2-SYSTEM. *Let Δ_2 represent a g-SIS enforcement system defined as below.*

$$\begin{aligned}\Delta_2 \equiv &(GA(u,o,g) \mid rendez \mid CC(u,o,g)) \mid parallel \mid \\ &(UI(u,o,g) \mid rendez \mid \text{TRM\_StrongStaleSafe}(u,o,g))\end{aligned}$$

where GA and CC, and UI and TRM_StrongStaleSafe *rendezvous respectively, and then execute concurrently via parallel composition (note GA, CC, and UI are the same as in Figure 8).*

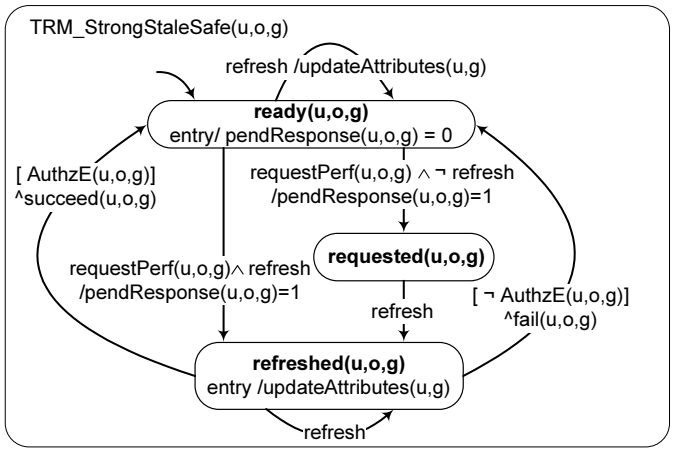

Fig. 10. TRM_StrongStaleSafe: This TRM machine satisfies strong stale-safety.

THEOREM 5.9 STRONG STALE-SAFE TRM THEOREM. Δ_2 *satisfies the Strong Stale-Safe Security Property.*

$$\Delta_2^{UF} \vDash \forall u : \mathrm{U}.\forall o : \mathrm{O}.\forall g : \mathrm{G}.\Box(\mathrm{perform}(u, o, g) \rightarrow \varphi_2(u, o, g))$$

NuSMV successfully verifies that the Δ_2-system satisfies strong stale-safety.

Until now we have only considered a generic perform. For simplicity we have assumed that this perform was always a read operation. In this way, objects are not modified by users, otherwise the number of possible objects created is unbounded since every write causes a new version of the object to be created.

6. FINITE UNBOUNDED ENFORCEMENT SYSTEMS

In the previous section we presented Δ systems describing three different enforcement models (through their differing TRM HTS's). The model checking results for stale safety regarding these systems rely on small models (containing a single user, single object, and single group). In this section we show how to extend our results to multiple or even an unbounded number of users, objects, and groups.

6.1 Parameterized Model Theorem

When model checking the small carrier Δ systems, there is nothing special about the choice of user, object, and group–they are anonymous. So intuitively, it should be the case that a system composed of many Δ systems would still model the same theorems as the single Δ systems for all of its users, objects, and groups. The following formalizes this idea by introducing a parameterized system, $\Omega(N)$, which consists of many independent formulations of the same FSM.

We consider a parameterized system (Ω), containing N processes, each of which is an instantiation of the same module. The processes execute in an independent and parallel manner. A module is a deterministic FSM, which is a 4-tuple:

$$FSM \equiv \langle S^0, S, \rho, \Sigma \rangle$$

Where S is the set of states, $S^0 \in S$ is the (single) initial state, ρ is the transition relation $\rho \subseteq S \times S$, and Σ is the alphabet. $\Omega(N)$ is represented as the parallel composition of N instantiations of the FSM:

$$\Omega(N) \equiv \mathop{||}_{i=1}^{N} FSM_i$$

in which FSM_i is an instantiation of FSM. In the Ω system, S^0_Ω is given by an array $[1..N]$ of initial states, S_Ω is given by an array $[1..N]$ of states. Σ_Ω is given below; the i^{th} elements in the arrays of tuples correspond to states in each FSM_i.

$$\Sigma_\Omega = \begin{cases} \mathrm{X} & : \textit{set of global input} \\ y & : \textit{array [1..N] of tuples} \\ z & : \textit{array [1..N] of tuples} \end{cases}$$

in which set X is the global input to each process (i.e. the set of input that is shared among all processes). Array y and the array z should have the same number of elements, where for FSM_i, $y[i]$ is the parameterized input into the deterministic values in $z[i]$.

Since all processes are independent, they should all take their transitions, respectively, in each step. The transition relation is formally defined as:

$$\rho(s, s') = \bigwedge_{i=1}^{N} z'[i] = exp(\mathrm{X}, \mathrm{X}', y[i], y'[i], z[i]) \tag{1}$$

in which $s, s' \in S_\Omega$ are the current and next state, respectively. We use *exp* to denote an expression that maps given input values (the X and y and their primed versions) to a value in the domain of z (i.e. a tuple that has values for each element in the tuple). This is used to show that *all* values in z are deterministic; there can be *no* free variables in each tuple of the array z. The lack of transition relations for the variables X and y is because they are inputs which are *not* determined by the system(Ω). Finally, note the intrinsic independence by the fact that $z[i]$ refers *only* to $y[i]$ and itself, never $y[j]$ or $z[j]$, $j \neq i$.

We are specifically interested in safety formulas of the form:

$$\Psi \equiv \forall i : \Box\psi(\mathrm{X}, y[i], z[i]) \tag{2}$$

where ψ can contain any boolean operators, tests for equality and inequality, and past temporal operators, but *no* existential or universal quantifiers. This again shows the independence of the processes. Each process fails or succeeds on its own.

Theorem 6.1 Parameterized Model Theorem. *An Ω system of arbitrary size (any n) models the property Ψ if and only if the same system of size 1 models the corresponding property Ψ.*

$$\forall n \in \mathbb{N} : n > 0, \Omega(n) \models \Psi(n) \leftrightarrow \Omega(1) \models \Psi(1) \tag{3}$$

Proof. First we show, by contrapositive, that if an arbitrarily sized system models a given property then the small system also models that property. We do this by showing that if the single process system, $\Omega(1)$, fails the property, then so

Table I. Assignments to the variables X, $x[i]$, and $y[i]$ in FSM_i

	X	$x[i]$	$y[i]$
$\sigma_i[1]$	X_1	$x_1[i]$	$y_1[i]$
$\sigma_i[2]$	X_2	$x_2[i]$	$y_2[i]$
$\sigma_i[3]$	X_3	$x_3[i]$	$y_3[i]$
$\vdots$	$\vdots$	$\vdots$	$\vdots$

Table II. Matrix of parallel traces, one for each FSM_i

	$\sigma_\Omega[1]$	$\sigma_\Omega[2]$	$\sigma_\Omega[3]$	...
σ_1	$\sigma_1[1]$	$\sigma_1[2]$	$\sigma_1[3]$	...
σ_2	$\sigma_2[1]$	$\sigma_2[2]$	$\sigma_2[3]$	...
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
σ_N	$\sigma_N[1]$	$\sigma_N[2]$	$\sigma_N[3]$	...

too does the large system. This is very easy to see. If a single process system fails, then it's just a matter of constructing a large system where at least one of the parameterized components matches the behavior of the single process system. Therefore if the single process system fails, then a large system can be constructed to fail as well.

Secondly, in order to prove in the other direction (also by contrapositive), it will be equivalent to prove that if a system of size n doesn't model a property $\Psi(n)$, then the system of size 1 *also* doesn't model its property, $\Psi(1)$:

$$\neg q \rightarrow \neg p = \exists n \in \mathbb{N} : n > 0, \Omega(n) \nvDash \Psi(n) \rightarrow \Omega(1) \nvDash \Psi(1)$$

For a given n and particular process j, showing that the $\Omega(n)$ system doesn't model this formula means that a finite trace of length $k+1$ exists, $\sigma_\Omega[..k]$, such that Ψ is not satisfied. The negation of Ψ means that there exists a process, j, such that $\Box\psi(\mathrm{X}, y[j], z[j])$ failed.

$$\Omega(n) \nvDash \Psi(n) \rightarrow \exists j : \neg\Box\psi(\mathrm{X}, y[j], z[j])$$

Table I shows how each FSM_i's traces (σ_i) are just assignments to the three types of variables. Note that all three are really assignments to *several* variables–all that are in set X and the tuples x and y. Also note that $y_j[i]$ refers to the values in y in the j^{th} state along the trace, for the i^{th} FSM.

As shown in Table II, the trace for the Ω system can be thought of as many parallel traces, σ_i, each assigning values to the $y[i]$'s and $z[i]$'s, as well as set X (all σ_i have to agree on set X). This is an important feature of the Ω system. The processes run in parallel rather than being interleaved.

Note that the formula $\Box\psi(\mathrm{X}, y[j], x[j])$ that failed, involves *only* the variables X, $y[j]$, and $z[j]$. This means that this system's trace (σ_Ω) caused a failure because of the assignment to σ_j (process j's trace). Specifically, $\sigma_i, \forall i \neq j$ does not affect the invalidity of the formula $\Box\psi(j)$.

We can create a new trace from this counterexample, $\tilde{\sigma}_\Omega$, which assigns σ_j (from above) to all other σ_i. Now in the trace $\tilde{\sigma}_\Omega$, all of the $y[i]$'s and $z[i]$'s are the same, and since they came from variables that caused a failure of the property, we can say that *all* $\psi(i)$'s fail:

$$\forall i, \neg\Box\psi(\mathrm{X}, y[i], z[i]) \tag{4}$$

Now we can truncate the system *and* the trace to produce a system of size 1, $\Omega(1)$, such that its only trace is σ_j from above which, we know failed the property from the larger system $\Omega(n)$. Therefore if a counterexample exists in the larger system $\Omega(n)$, then a counterexample exists in the single system $\Omega(1)$. Thus if the large system does not model a property, then neither does the single system.

□

6.2 Formal Definition of Δ^{UF}-System

It remains to be shown that the three Δ systems discussed earlier are indeed instances of the above Ω systems. Also we must show that the properties we wish to check are examples of Ψ formulas. At first, it may appear that the Δ systems are *not* instances of Ω systems. However by associating integers with each tuple of input to the Δ systems we can show that they do exhibit the same behavior and thus the above proof is also valid for our Δ systems.

The small carrier Δ is defined as follows:

$$\Delta_{u,o,g} \equiv \langle S^0_{u,o,g}, S_{u,o,g}, \rho_{u,o,g}, \Sigma_{u,o,g} \rangle$$

We can associate consecutive integers with each tuple needed to complete the power set $\mathcal{U} \times \mathcal{O} \times \mathcal{G}$. Formally this association is a function:

$$f : \mathbb{N} \mapsto \langle u, o, g \rangle \in \mathcal{U} \times \mathcal{O} \times \mathcal{G}$$

There exists a finite number, N, of distinct tuples $\langle u, o, g \rangle$– these tuples are the input for each Δ system, therefore $\Delta(u, o, g) \equiv \Delta(f(i)) \equiv \Delta^i$, where $f(i) = \langle u, o, g \rangle$, thus i refers to the i^{th} tuple of $\mathcal{U} \times \mathcal{O} \times \mathcal{G}$. We use a super-script to distinguish the i^{th} tuple from the 3 different enforcement models presented (Δ_0, Δ_1, and Δ_2). The unbounded finite Δ system, Δ^{UF}, is a parallel composition of multiple small carrier Δ-systems:

$$\Delta^{UF} \equiv \mathop{||}_{i=1}^{N} \Delta^i$$

where N is the number of elements in $\mathcal{U} \times \mathcal{O} \times \mathcal{G}$. For example, the unbounded finite system for the Δ_0 system can be written as:

$$\Delta_0^{UF} \equiv \mathop{||}_{i=1}^{N} \Delta_0^i$$

Since this is clearly an Ω system (now that we have associated integers with each combination of user, object, and group). The last thing to do is to show that the FOTL formulas have the same form as the $\Psi(N)$. All of the formulas we are concerned with can be written in the form:

$$\forall u : \mathrm{U}. \forall o : \mathrm{O}. \forall g : \mathrm{G}. \Box \psi(u, o, g)$$

In exactly the same way we parameterized the parameters for the Δ-systems, we can associate integers (with the *exact* same function) to the tuples that are needed as parameters into each ψ:

$$\forall i. \Box \psi^i \equiv \bigwedge_{i=1}^{N} \Box \psi^i \equiv \Psi(N)$$

Finally we can use Theorem 6.1 to show that for Δ_0, Δ_1, and Δ_2:

$$\Delta_i(u, o, g) \vDash \Box \psi(u, o, g) \leftrightarrow \Delta_i^{UF} \vDash \forall u : \mathrm{U}. \forall o : \mathrm{O}. \forall g : \mathrm{G}. \Box \psi(u, o, g)$$

This statement is implicit in Section 5 when presenting each of the FOTL theorems.

7. RELATED WORK

To the best of our knowledge, this is the first effort towards formalization of the notion of stale-safety and proof-of-concept verification in distributed systems. The work of Lee et al [Lee et al. 2007; Lee and Winslett 2006] is the closest to ours that we have seen in the literature, but focuses exclusively on the use of attribute certificates, called credentials, for assertion of attribute values in trust negotiation systems. Lee et al focus on the need to obtain fresh information about the revocation status of credentials to avoid staleness and propose different levels of consistency amongst the credentials used to make a decision. Our formalism is based on the notion of a "refresh time," that is the time when an attribute value was known to be accurate. We believe the notion of refresh time is central to formulation of stale-safe properties. Because Lee et al admit only attribute certificates as carriers of attribute information there is no notion of refresh time in their framework. Further, formal specification of stale-safe properties using temporal logic allows designers to ensure stale-safety in their enforcement models using automated verification techniques such as model checking.

Furthermore, as [Lee et al. 2007] note, there is a rich body of literature on achieving consistency in distributed systems (see [Tanenbaum and Van Steen 2007] for a survey). The fundamental problem addressed in this domain is to achieve consistency in data replication systems and balance it with performance and availability. For example, [Yu and Vahdat 2002] discuss a continuous consistency model that explores the range of consistency levels between strong consistency (where consistency requirement is absolute) and optimistic consistency (where consistency requirement is not critical). In contrast, stale-safety is concerned about making safe authorization decisions at an enforcement point using attribute values that may be out-of-date. We envision that stale-safe security properties identified in this article would be used for making authorization decisions in replicated systems.

The work of Schneider ([Schneider 2000]) characterizes security policies enforceable using a security automata by monitoring system execution. Specifically, it ensures that a program (or the HTS model in our case) in fact enforces the security policy and terminates the program in case of failure to do so. In comparison, our work strengthens this by ensuring that the enforced security policies are stale-safe. While precise enforcement of security policy is critical in high-assurance and safety-critical systems [Lamport 1985], stale-safety allows a designer to make trade-offs in systems where availability is equally desirable. The weak stale-safety property enables safe authorization decisions to be made when an authoritative entity is not available to make a decision using up to date authorization information. The strong property, on the other hand, is desirable in safety-critical systems. The timely-weak and time-strong properties (appendix A) enable one to bound acceptable staleness in terms of time elapsed between the point at which the request was last known to have been authorized and the point at which the requested action is performed.

The use of model checking in automating analysis of security policies, properties, and protocols has attracted a lot of research attention. There has been fruitful research in model checking security protocols since Gavin Lowe's seminal work [Lowe 1996; 1997] using model checker FDR for CSP to reveal a subtle attack of the Needham-Schroeder authentication protocol and cryptographic protocols.

Recently, model checking has been increasingly employed to reason about security properties, especially in RBAC and trust management systems. Sistla and Zhou [Sistla and Zhou 2006; 2008] propose a model-checking framework for security analysis of RT policies [Li et al. 2002]. Jha and Reps [Jha and Reps 2004] verify such properties as authorization, availability, and shared access of the SPKI/SDSI policy language using model checking. Fisler et.al. [Fisler et al. 2005] analyze the impact of policy changes on RBAC systems and verify the separation of duty properties using their own model checking tool called Margrave. Schaad et. al. [Schaad et al. 2006] verify separation of duty properties in RBAC systems through NuSMV [Cimatti et al. 2000]. Hansen et. al. [Hansen and Oleshchuk 2005] utilize an explicit model checking tool called Spin to verify various static and dynamic separation of duty properties in RBAC. Additionally, other works [Bandara et al. 2003; Gilliam et al. 2005; May et al. 2006; Zhang et al. 2005; 2008] also leverage formal techniques to verify properties of security and privacy policy specifications.

The notion of timestamps creates a large problem for model checking due to state explosion. Counter abstraction is a key insight into solving these types of problems. We used the algorithm presented in [Tsuchiya and Schiper 2007] to model the timestamps in our model. They present code for handling 3 events, so it was necessary to create code for handling 2 and 4 events for our model. We automated this process by writing C code to generate a NuSMV model for handling n events (given as input to the executable). This generic module is then used by instantiating it with the given events as well as their initial order.

In extending the small carrier (single) model to the large carrier (multiple entities), we used a similar approach as in [Pnueli et al. 2001] to prove Theorem 6.1. There are two major differences in our approach to that of [Pnueli et al. 2001]. The first is that our parameterized model differs from their definition of a BDS (Bounded-discrete data system). The BDS they present has a specific transition relation which makes the many processes transition in an interleaved fashion. This is evident by the fact that their transition relation has an existential quantifier, meaning that at least one process transitions (it can be many, but only one *has* to). Contrasting with our model, our transition relation involves a universal quantifier, indicating that all processes transition at once, in every state. The second major difference is that our processes cannot modify any values from the other processes. For these reasons it was necessary to synthesize a new theorem for proving the large carrier case. We used a similar approach as used to prove the Small Model Theorem in [Pnueli et al. 2001]–that is we found a counterexample in a large system then showed that this counterexample could be generated by a smaller system (in our case a system of size 1).

8. CONCLUSION

Attribute staleness is inherent to any distributed system due to physical distribution of user and object attributes. While it is not possible to eliminate staleness entirely, we can still manage and limit its impact. In this article, we proposed two stale-safe security properties and specified a few variations on them by using temporal logic. This formalization not only enabled us to precisely state the properties but also allowed systems to be formally verified. With model checking we proved the small

model satisfies the properties and were able to generalize this result to a large model using the Parameterized Model Theorem.

In our current work, we assumed that authorization information is maintained centrally at an authoritative server. Our future work involves investigating the notion of stale-safety in scenarios where a user's authorization information may be distributed across multiple authorization servers, each of which may maintain a specific set of attributes. This raises an issue of synchronization among the many CCs. Theoretically, the CCs can be synchronized with a global clock which means that comparing timestamps from two different CCs is consistent. However, in practice, it would be impossible to get an exact synchronization. This could change the policy and thus could affect staleness properties for such a system. One possible solution would be to allow for a tolerance in comparing timestamps; this tolerance could be set according to latencies in synchronizing the CCs.

REFERENCES

Alpern, B. and Schneider, F. B. 1987. Recognizing safety and liveness. *Distributed Computing 2*, 117–126.

Bandara, A. K., Lupu, E. C., and Russo, A. 2003. Using event calculus to formalise policy specification and analysis. In *Proceedings of the 4th International Workshop on Policires for Distrubuted Systems and Networks (POLICY 2003)*. 26–39.

Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., and Tacchella, A. 2002. Nusmv 2: An opensource tool for symbolic model checking. *Lecture Notes in Computer Science*, 359–364.

Cimatti, A., E.M.Clarke, Giunchiglia, F., and Roveri, M. 2000. NuSMV: A new symbolic model checker. *International Journal on Software Tools for Technology Transfer 2,* 4, 410–425.

Clarke, E. M., Emerson, E. A., and Sistla, A. P. 1986. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Language and Systems (TOPLAS) 8,* 2, 244–263.

Fisler, K., Krishnamurthi, S., Meyerovich, L. A., and Tshchantz, M. C. 2005. Verification and change-impact analysis of access-control policies. In *ICSE*. ACM Press, 196–205.

Gilliam, D., Powell, J., and Bishop, M. 2005. Application of lightweight formal methods to software security. In *Proceedings of the 14th IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 2005)*.

Goguen, J. and Meseguer, J. 1982. Security policies and security models. *IEEE Symposium on Security and Privacy 12*.

Hansen, F. and Oleshchuk, V. 2005. Conformance checking of RBAC policy and its implementation. In *Information Security Practice and Experience*. LNCS, vol. 3439. Springer Berlin/Heidelberg, 144–155.

Harrison, M. A., Ruzzo, W. L., and Ullman, J. D. 1976. Protection in operating systems. *Comm. of the ACM*, 461–471.

Jha, S. and Reps, T. 2004. Model checking SPKI/SDSI. *Journal of Computer Security 12*, 317–353.

Krishnan, R., Sandhu, R., Niu, J., and Winsborough, W. H. 2009a. A conceptual framework for group-centric secure information sharing. In *Proc. of the 4th ACM International Symposium on Information, Computer, and Communications Security*. 384–387.

Krishnan, R., Sandhu, R., Niu, J., and Winsborough, W. H. 2009b. Foundations for group-centric secure information sharing models. In *Proc. of the ACM Symp. on Access Control Models and Tech.* 115–124.

Lamport, L. 1977. Proving the correctness of multiprocess programs. *IEEE Trans. Softw. Eng. 3,* 2, 125–143.

Lamport, L. 1985. Logical foundation, distributed systems-methods and tools for specification. *Lecture Notes in Computer Science 190*.

Lee, A., Minami, K., and Winslett, M. 2007. Lightweight cnsistency enforcement schemes for distributed proofs with hidden subtrees. *Proceedings of the 12th ACM symposium on Access control models and technologies*, 101–110.

Lee, A. and Winslett, M. 2006. Safety and consistency in policy-based authorization systems. *Proceedings of the 13th ACM conference on Computer and communications security*, 124–133.

Li, N., Mitchell, J. C., and Winsborough, W. H. 2002. Design of a role-based trust-management framework. *Security and Privacy, IEEE Symposium on 0*, 114.

Lowe, G. 1996. Breaking and fixing the needham-schroeder public-key protocol using fdr. In *Second International Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*. LNCS, vol. 1055. Springer, 147–166.

Lowe, G. 1997. Casper: A compiler for the analysis of security protocols. In *10th IEEE workshop on Computer Security Foundations*. 18–30.

Manna, Z. and Pnueli, A. 1992. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, Heidelberg, Germany.

May, M. J., Gunter, C. A., and Lee, I. 2006. Privacy APIs: Access control techniques to analyze and verify legal privacy policies. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop*.

Niu, J. 2005. Template semantics: A parameterized approach to semantics based model compilation. Ph.D. thesis, University of Waterloo, School of Computer Science.

Niu, J., Atlee, J. M., and Day, N. A. 2003. Template semantics for model-based notations. *IEEE Transactions on Software Engineering 29,* 10 (October), 866–882.

Niu, J., Krishan, R., Bennatt, J. F., Sandhu, R., and Winsborough, W. H. 2011. Enforcable and verifiable stale-safe security properties in distributed systems. Tech. Rep. CS-TR-2011-02, UTSA.

Pnueli, A., Ruah, S., , and Zuck, L. 2001. Automatic deductive verification with invisible invariants. In *TACAS: Proceedings of Tools and Algorithms for the Construction and Analysis of Systems*. 82–97.

Schaad, A., Lotz, V., and Sohr, K. 2006. A model checking approach to analysis organizational controls. In *Proceedings of the 11th ACM Symposium on Access Control Models and Technologies (SACMAT06)*. 139–149.

Schneider, F. B. 2000. Enforceable security policies. *ACM Trans. Inf. Syst. Secur. 3,* 1, 30–50.

Sistla, A. P. and Zhou, M. 2006. Analysis of dynamic policies. In *Proceedings of Foundations of Computer Security and Automated Reasoning for Security Protocol Analysis*. 233–262.

Sistla, A. P. and Zhou, M. 2008. Analysis of dynamic policies. *Information and Computation 206,* 2-4, 185–212.

Tanenbaum, A. and Van Steen, M. 2007. *Distributed Systems: Principles and Paradigms*. Pearson Prentice Hall.

Tsuchiya, T. and Schiper, A. 2007. Model checking of consensus algorithms. In *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems*. 137–148.

Yu, H. and Vahdat, A. 2002. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Trans. Comput. Syst. 20,* 3, 239–282.

Zhang, N., Ryan, M., and Guelev, D. P. 2005. Evaluating access control policies through model checking. In *Proceedings of the 8th Information Security Conference*. LNCS, vol. 3650. Springer-Verlag, 446–460.

Zhang, N., Ryan, M., and Guelev, D. P. 2008. Synthesising verified access control systems through model checking. *Journal of Computer Security 16,* 1, 1–61.

Appendix

A. FRESHNESS OF AUTHORIZATION

Let us now consider how to express requirements that bound acceptable elapsed time between the point at which attribute refresh occurs and the point at which a requested action is performed. We refer to this elapsed time as the degree of *freshness*. For this we introduce a sequence of propositions $\{P_i\}_{0\leq i\leq n}$ that model n time intervals. These propositions partition each trace into contiguous state subsequences that lie within a single time interval, with each proposition becoming true immediately when its predecessor becomes false. They can be axiomatized as follows:

$$P_1\,\mathcal{U}\,(\Box\neg P_1 \wedge (P_2\,\mathcal{U}\,(\Box\neg P_2 \wedge (P_3\,\mathcal{U}\,(...\,\mathcal{U}(\Box\neg P_{n-1} \wedge \Box P_n)...)))))$$

This partially defines correct behavior of a clock, given by a component of the FSM. Note that is it not possible to express in LTL that the clock transits from P_i to P_{i+1} at regular intervals of elapsed time. taken by the FSM, P_i holds for some time interval $i \in [1..n]$, and that the FSM proceeds through the time intervals in order. The current time can be interrogated by the other FSM components with which it is composed. It can also be referred to in the variant stale-safe properties presented in the following paragraphs. If the clock is accurate with respect to transiting from P_i to P_{i+1} at regular intervals, the enforcement machine obeying these variant properties will enforce freshness requirements correctly.

We now formulate variants of φ_1 and φ_2 that take a parameter k indexing the current time interval. These formulas use two constants, ℓ_1 and ℓ_2 which represent the number of time intervals since the authorization and the request, respectively, that is considered acceptable to elapse prior to performing the requested action. The formulas prohibit performing the action if either the authorization or the request occurred further in the past than permitted by these constants.

$$\begin{aligned}
\varphi_1(u,o,g,k) \equiv \ominus((\neg\mathrm{perform}(u,o,g) \wedge (\neg\mathrm{RT} \vee (\mathrm{RT} \wedge \mathrm{Authz}_{\mathrm{CC}}(u,o,g))))\,\mathcal{S}\\
(\mathrm{request}(u,o,g) \wedge \bigvee_{max(0,k-\ell_2)\leq i\leq k} P_i \wedge\\
(\neg\mathrm{RT}\,\mathcal{S}\,(\mathrm{RT} \wedge \mathrm{Authz}_{\mathrm{CC}}(u,o,g) \wedge \bigvee_{max(0,k-\ell_1)\leq i\leq k} P_i))))\\
\varphi_2(u,o,g,k) \equiv \ominus(\neg\mathrm{perform}(u,o,g) \wedge \neg\mathrm{RT})\,\mathcal{S}\\
(\mathrm{RT} \wedge \mathrm{Authz}_{\mathrm{CC}}(u,o,g) \wedge \bigvee_{max(0,k-\ell_1)\leq i\leq k} P_i \wedge\\
((\neg\mathrm{perform}(u,o,g) \wedge (\neg\mathrm{RT} \vee (\mathrm{RT} \wedge \mathrm{Authz}_{\mathrm{CC}}(u,o,g))))\,\mathcal{S}\\
(\mathrm{request}(u,o,g) \wedge \bigvee_{max(0,k-\ell_2)\leq i\leq k} P_i))
\end{aligned}$$

With these formulas, we are now able to state variants of weak and strong stale safety that require timeliness, as defined by the parameters ℓ_1 and ℓ_2.

Definition A.1 Timely, weak stale safety. *An FSM has the* timely, weak

stale-safe security property *if it satisfies the following LTL formula:*

$$\forall u : \mathrm{U}.\forall o : \mathrm{O}.\forall g : \mathrm{G}.\Box \, (\bigwedge_{0 \le k \le n} (\mathrm{perform}(u, o, g) \wedge P_k) \rightarrow (\varphi_1(u, o, g, k) \vee \varphi_2(u, o, g, k)))$$

Definition A.2 Timely, strong stale safety. *An FSM has the* timely, strong stale-safe security property *if it satisfies the following LTL formula:*

$$\forall u : \mathrm{U}.\forall o : \mathrm{O}.\forall g : \mathrm{G}.\Box \, (\bigwedge_{0 \le k \le n} (\mathrm{perform}(u, o, g) \wedge P_k) \rightarrow \varphi_2(u, o, g, k))$$

B. STALE-SAFETY AND SAFETY PROPERTIES

The stale-safety properties that we have defined takes an authorization policy (e.g. $\mathrm{Authz}_{\mathrm{TRM}}$) that is a safety property [Lamport 1977; 1985; Alpern and Schneider 1987] and specifies the right conditions under which it may hold in a distributed system when authorization information may potentially be stale. In [Schneider 2000], the author defines a class of enforcement mechanisms called EM that is capable of enforcing security policies in the domain of a safety property. We establish the relationship between our definition of stale-safety with that of the work of [Schneider 2000].

B.1 Background

A trace, σ, is a finite or infinite sequence of states. In g-SIS, for example, a trace is a finite or infinite sequence of states, where in each state some set of events hold. (A sample trace in g-SIS could be $\mathrm{join}(u_1, g)$, $\mathrm{join}(u_2, g)$, $\mathrm{add}(o, g)$, $\mathrm{leave}(u_2, g)$,...). Let Ψ be the universe of traces and Π be a set of traces. Intuitively, a safety property stipulates that something bad never happens. A security policy $\mathcal{P}$ is a *safety property* if it satisfies the following three intuitive properties (see [Schneider 2000] for a formal definition):

(1) A security policy $\mathcal{P}$ holds for a set traces Π if for each σ in Π, $\widehat{\mathcal{P}}(\sigma)$ holds, where $\widehat{\mathcal{P}}$ is a policy predicate on individual traces. That is, a security policy holds for a set of traces if it holds independently for each trace in that set. (In other words, the policy can be enforced in a given trace without referring to other traces).

(2) If the evaluation of $\widehat{\mathcal{P}}$ fails for a finite prefix τ' of a trace, then $\widehat{\mathcal{P}}$ also fails for an evaluation of $\tau'\sigma$ for each σ in Ψ. ($\tau'\sigma$ denotes a finite trace τ' followed by a finite or infinite trace σ). That is, once $\widehat{\mathcal{P}}$ fails at some point in a trace, it is impossible to recover in the future.

(3) Finally, if $\widehat{\mathcal{P}}$ fails for a given trace σ then it should fail after a finite period. (That is, it should be possible to identify a specific state in σ at which point $\widehat{\mathcal{P}}$ fails).

B.2 Safety characteristics of g-SIS policies

It is straight-forward to observe that the g-SIS policy, $\mathrm{Authz}_{\mathrm{TRM}}$, is a safety property.

Proposition B.1. $\mathrm{Authz}_{\mathrm{TRM}}$ *is a* safety *property.*

Observing the structure of $\text{Authz}_{\text{TRM}}$, it satisfies the first requirement above because authorization in g-SIS can be determined without having to refer to multiple traces. That is, given a trace, $\text{Authz}_{\text{TRM}}$ can precisely say whether a user is authorized to perform the requested action in that trace independent of actions in other traces. Further inspecting the structure of $\text{Authz}_{\text{TRM}}$, it is prefix-closed. That is, if $\text{Authz}_{\text{TRM}}$ fails to hold in a particular state of a given finite prefix of a trace, it will remain so regardless of the actions that may follow that state. This is because $\text{Authz}_{\text{TRM}}$ does not contain any future temporal operators. The third requirement is also trivially satisfied by $\text{Authz}_{\text{TRM}}$ because when authorization fails, it fails at a finite point in the trace.

PROPOSITION B.2. $\text{Authz}_{\text{TRM}}$ *and the stale-safe security properties are enforceable by* EM.

That is, $\text{Authz}_{\text{TRM}}$ is enforceable by the class of enforcement mechanisms EM specified in [Schneider 2000] since it satisfies the three requirements of a policy that are enforceable by EM. Clearly, following the earlier argument, the four stale-safety security properties that we have defined are also safety properties and are enforceable by EM. In section 4, we define finite state machines that enforce g-SIS policies and show the equivalent mechanism in EM by constructing a security automata.

This establishes that stale-safe security properties specified on systems that enforce authorization policies that are safety properties are enforceable by EM. Thus the characterization of stale-safety ensures that the security policies enforced by EM are stale-safe in a distributed system where authorization information available to EM may be potentially out-of-date.

C. G-SIS ENFORCEMENT MODEL AS SECURITY AUTOMATON

We argue that the enforcement model represented as a composition of HTSs is a security automaton [Schneider 2000]. We first show that an HTS, $\langle\ S,\ S^H,\ S^0,\ S^F,\ E,\ V,\ V^0,\ T\ \rangle$ is a security automaton, denoted as a 4-tuple $\langle\ Q,\ Q_0,\ I,\ \delta\ \rangle$.

—Q is a countable set of states, where each state is a 3-tuple $\langle CS,\ IE,\ CV\rangle$. CS is the set of current control states, where $CS \subseteq S$, and for any basic state $s \in CS$, so are all of its ancestor states (i.e., states that include basic states). IE is the set of current internal events that are generated by the system, where $IE \subseteq E$. The set CV is a function that maps each variable $v \in V$ to its current value.

—Q_0 is a countable set of initial automaton states, which is given by : $\{\langle CS, \emptyset, CV)\rangle \mid CS \subseteq S^0 CV \models V^0\}$.

—I is a countable set of input symbols, which are given by the the input events, $I \subseteq E$

—δ is a transition relation, $\delta : (Q \times I) \rightarrow 2^Q$.

For instance, the TRM_Unsafe HTS is a security automaton. Q is a set of 3-tuples, $\langle CS,\ IE,\ CV\rangle$, where CS is a subset of $\{ready,\ authorized,\ refreshed\}$, IE is a subset of $\{succeed,\ fail,\ requestPend\}$, and CV maps each variable in $\{pendResponse, \text{JoinTS}, \text{LeaveTS}, \text{AddTS}, \text{ORL}\}$ to their current values. The initial state Q_0 is a singleton set of a tuple, $\{\langle\ \{ready\},\ \emptyset,\ \{(pendResponse, 0),$

(JoinTS, 0), (LeaveTS, 0), (AddTS, 0), (ORL = $\emptyset$)$\rangle$}. The input $I = \{refresh\}$. A transition relation defines the set of transitions that the TRM_Unsafe HTS can take. For example, transition $t1$ is triggered by input $refresh$ and updates all the TRM time stamps with the current time stamp values at CC, for instance, (JoinTS = 2), (LeaveTS = 0), (AddTS = 3), (ORL = $\emptyset$)$\rangle$. Transition $t1$ moves the HTS from state $\langle$ {$ready$}, $\emptyset$, {($pendResponse$, 0), (JoinTS, 0), (LeaveTS, 0), (AddTS, 0), ($ORL = \emptyset$)$\rangle$ to state $\langle$ $refreshed$, $\emptyset$, {($pendResponse$, 0), (JoinTS, 2), (LeaveTS, 0), (AddTS, 3), (ORL = $\emptyset$)$\rangle$. Note that when the HTS is in control state $ready$, the TRM time-stamp variables may carry different values. This suggests that a single HTS control state corresponds to multiple states in Q and transition $t1$ represents a collection of fine-grained transitions in security automaton. The representation of the other HTSs in the enforcement model as security automata follows the same way.

HTSs can be composed via various composition operators to form a tree of HTSs, denoted as a CHTS. A CHTS is a security automaton, where a state is a collection of states, one from each HTS, the input is the union of the inputs for all the HTSs, and the transition relation is from one collection of states to a set of collection of states by taking a set of input events. In a CHTS, the composition operators determine which HTSs to execute. HTSs take a step according to their respective transition relation and the state collection is a tree structure, matching the CHTS composition hierarchy.

D. MODEL CHECKING RESULTS

This appendix presents the design of the NuSMV model for the g-SIS enforcement mechanism specified in HTSs and the complete code listing.

D.1 Structure of SMV Input Language

The following is a simplified version of a TRM module (without timestamps). The parameter cc gives the TRM access to a CC from which to set it's policy. The user interface isn't present here, which doesn't affect unsafe or weakly safe machines. However, the strongly safe model requires knowledge about when a request is made, so it is not shown in this example.

```
1 MODULE TRM(cc) -- <=== "--" signfies a comment
2  VAR
3   policy : boolean;     --                          this is authzTRM
4   userJoined : boolean; --               status during last refresh
5   refreshed : boolean;--authzTRM has not changed since last refresh
6   refresh : boolean;    --  non-deterministic refresh (it's random)
7  ASSIGN
8   init(policy) := FALSE;               --initial states are chosen
9   init(strongPolicy) := FALSE;
10  init(refreshed) := FALSE;
11  init(userJoined) := FALSE;
12  next(refreshed) := case                    -- always set to true
13  next(refresh) : TRUE;                      --when refresh occurs
14  policy xor next(policy) : FALSE;     -- means the policy changes
15   TRUE : refreshed;                   --   in the next state thus
16  esac;                                --        authzTRM is stale
17  next(userJoined) := case
18   next(refresh) : next(cc.userJoined);           --only get value
19   TRUE : userJoined;                             --       from CC
20  esac;
21  next(policy) := case     --  short circuit on refresh, just grab
```

```
22   next(refresh) : next(cc.policy);              -- value from CC
23   next(cc.super_distro) : !next(cc.falsePositive) &
24       next(userJoined) & next(cc.objectAdded);
25   TRUE : policy;             --the above essentially just figures
26  esac;                       --   out if the object add timestamp
27 DEFINE                       --      is greater than or less than
28  UNSAFE_POLICY := policy;    --     the TRM's user join timestamp
29  WEAK_POLICY := refreshed & policy;
```

The three different sections are defined by the placement of the keywords VAR (line 2), ASSIGN (line 7), and DEFINE (line 27). The alphabet consists of the variables *policy*, $userJoined$, $refreshed$, and $refresh$. All variables are declared as booleans (takes on values TRUE or FALSE).

The ASSIGN section is used to define the transition relation. The transition relation is not given as a boolean expression, rather it is given as a relation between unprimed and primed versions (current and next values) of the state variables. There are essentially 2 ways to define the transition relation: by specifying the current value, or by specifying the next value. It is incorrect to specify both such as $x := 1$ and $next(x) := 2$. This relation cannot be satisfied, because x should equal 1 in every state, therefore it cannot equal 2 in the next. Usually variables specified by their next value also have an init declaration to assign a value in the initial state. If an init isn't given for variable x *and* it isn't defined by it's current value then x takes on a random value from its domain in the initial state. If, in addition, no next declaration is given for variable x, then this variable is completely free (i.e. $refresh$). In this example all variables (except $refresh$) are initialized to a single initial state.

The case construct allows for a piecewise definition of the transition relation. The case construct returns values if the condition is met. For instance on line 22, if $next(refresh) =$ TRUE, then the next value of *policy* will be assigned the value obtained directly from the CC. On line 14, if $next(refresh) =$ FALSE *and* the policy value changes in the next state, then $refreshed$ is assigned the value FALSE in the next state. The keyword esac defines the end of the most recently entered case statement. Notice that the last condition for all case statements must be TRUE, so that a value is always guaranteed to be returned. Non-deterministic values can be returned by returning a set of values such as {TRUE, FALSE}. The value will be assigned non-deterministically from the given set. The best way to understand the symantics of the case construct is that the conditions are essentially a long chain of if-else if-else if ... else. So if two or more conditions are True, the return value will be determined by the condition listed *first* in the case statement (i.e. it short-circuits on a match). For instance on lines 22 and 23, if super distribution happens in the same state that a refresh occurs, then the return value for super distribution is ignored, and the TRM just grabs the policy from the CC since the $refresh$ case is listed first.

The DEFINE section creates macros–values that can be computed by looking at the current state. In addition to declaring the variables by their next value, you can also define them by their current value (e.g. $x := next(policy)$. All macros could be defined in this way, but this would add to the state space. Because the SMV input language does not allow macros to access any next values, if you wish to define such a macro it has to be part of the state variables and can be declared

like the variable x above (using any expression involving current and next values). Obviously such a variable could not reference it's own next value to try and evaluate it's current value. In this example the DEFINE section is used to define the unsafe and weak policies. The unsafe policy is exactly that, unsafe, so $authz_{TRM}$ is always returned from a request. On the other hand, weak stale safety requires "non-stale" attributes, thus the machine must be refreshed *and* $authz_{TRM}$ is TRUE.

Now it is helpful to understand a little better how the ASSIGN section is defining the behavior of this finite state machine. During a refresh the TRM gains two pieces of information: whether the user is joined (line 18) and the value of the CC's policy (line 22). The user's status is used to compute $authz_{TRM}$ during super distribution (lines 23 and 24)–notice that if $next(userJoined)$ is FALSE, this will return False; $cc.falsePositive$ means that although the object is available in the CC, it's timestamp is less than the user's join timestamp and thus $authz_{TRM}$ should be FALSE. The only other event this machine responds to is when *policy* changes values without a refresh which can only come from super distribution. This notifies the machine that it's attributes have become stale. Finally, if there is no refresh, no super distribution, and no change in *policy*, then all values remain the same.

D.2 Structure of NuSMV Specification for g-SIS Enforcement Model

The above was a simplified version of our TRM, that basically abstracted away the timestamps. Instead of modeling timestamps, the above modeled the result of the timestamps. That is, the result of the evaluation of the timestamps which is ultimately a boolean expression.

The NuSMV model consists of five top-level modules to represent the CC_User, CC_Object, CC, UI, and TRM HTSs respectively. The GA is not modeled (with its own module), instead the actions of the GA are included as part of the CC_User and CC_Object modules. The TRM refreshes at random times and gets the authorization information from the CC, also, when super distribution happens the TRM is able to update it's authorization information based on the new object add time stamp. Modules communicate via parameter passing. The NuSMV model also has a main module as the entry point. The main module is responsible for creating instance of these five modules.

CC Module:
The CC maintains "correct" time stamps on each of four events: user join, user leave, object add, and object remove events. The CC is only capable of reporting the correct and accurate *order* of these events. The idea is taken from [Tsuchiya and Schiper 2007]. The meaning of these timestamps is described in the TimeStamps Module section. Because the CC does not maintain actual timestamps, this presents a limitation in modeling super distribution. The TRM contains only a snapshot of the timestamps once held in the CC and therefore the TRM's timestamps cannot be compared against the CC's (when a refresh happens the TRM's previous timestamps are discarded and replaced with the ones from the CC). Because of this, super distribution is *only* allowed to happen during the state the object is added. In this way when super distribution happens, it is known that the TRM's object add time stamp should be set to a value larger than all of it's previous timestamps.

CC_Object and CC_User Modules:
These modules essentially model the interface between the GA and the CC. These modules keep track of which state (member or nonMember) the object is in (user or object really). They go in and out of membership states based on the events request_join/add_TS and request_leave/remove_TS; these events are sent immediately to the CC to tell it how to keep up with the time stamps (the CC's timestamps are set in the same state that the event occurs). The states are used so that an "add" request never happens while in a state of membership, likewise a remove request never happens when already removed.

UI Module:
This module models how a user will access information, the UI module is assumed to be in parallel with the CC_User module that is also created (that is it is assumed that this user is the user that CC_User's time stamps refer to). Likewise, all requests to perform on objects are assumed to be the CC_Object module created in parallel. This module is very simple, the UI makes requests at random intervals. The UI has three states: idle, objAvail, and waiting. The UI never makes a request while waiting and never requests before it has visited the objAvail. The idle state acts as an entry point to model the time before the user becomes aware of the object. However, once the user is aware of the object (by going into the objAval state) it never goes back to idle. After getting a response (leaving the waiting state), the UI will go back to the objAvail state.

TRM_Unsafe Module:
This module is extremely simple, it always responds immediately to requests with it's current evaluation of AuthzE.

TRM_Weak Module:
This module mirrors very closely, the FSM presented for the Weak TRM. Just to elaborate a little, it has three states: idle, staleCheck, and refreshed. The machine *always* goes into the refreshed state when a refresh event occurs. Also, when a request is made, the machine always goes to the staleCheck state and then if no refresh occurs, a response is given. The response, succeed or fail, depends on the value of AuthzE and staleSafe boolean which just checks whether or not the refresh time stamp is greater than the object add time stamp. This is accomplished, by using the Timestamps module (with only 2 timestamps) given the refresh event and the request_add event from the CC. Although technically speaking, the TRM shouldn't have access to this information (from the CC), it is only used in the TimeStamps module to determine the correct evaluation of which came first, the object add event or the refresh event. In reality the TRM would have an actual time stamp, but we cannot use actual timestamps due to computational restrictions.

TRM_Strong:
This module is identical to the FSM presented for the Strong TRM. It has three states: idle, requested, refreshed. Whenever a request occurs the state changes to

either requested (if no refresh happens) or refreshed (if a refresh does happen). Like in the weak model, once a request happens the only way to get back to idle is to eventually get to the refreshed state (in the weak machine it was the staleCheck state). Once in the refreshed state, the TRM immediately gives a response based on it's current evaluation of AuthzE. Notice that the refreshed state can ONLY be reached after a request occurs. Once in the requested state, the machine waits for a refresh to enter the refreshed state where it will then give a response.

AuthzTRM Module:
This module handles the evaluation of the AuthzE for the TRM. It takes the CC (to get the authorization information) and the refresh event. There are two events that affect the value of AuthzE: super distribution and refresh. The refresh event is very straight forward: during a refresh AuthzE = AuthzCC. Because we don't use actual time stamps, we have to exploit the fact that super distribution *only* happens when an object add event occurs. So in this way we can reason that if the user was a member during the last refresh then the object add time stamp *must* be greater than the user's add time stamp, therefore authzE will evaluate to true. On the other hand, if the user was *not* a member during the last refresh, then AuthzE will still be false. Notice that if a refresh occurs in the same state as super distribution that the super distribution is ignored, that is AuthzE is just set to AuthzCC.

TimeStamps Module:
This module handles the time stamps for the CC module and this is how we are able to do an "infinite" simulation—that is the simulation isn't bounded by a finite clock. The idea is very simple: only the order matters in the evaluation of the policy, not the actual values of the time stamps. We also produced a finite model which used a clock module that essentially just incremented an integer value in each state until a maximum value was reached, then time stopped and no more transitions were allowed. This approach is limited because no matter how large you make the "end time" it will always be finite. Also having a large end time causes a very large number of states for NuSMV to explore.

To instantiate a TimeStamps module, you need n booleans which correspond to the n events you are tracking. Internally there are $n + 1$ integer time stamps corresponding to each of these events which range from $t = 0$ to $t = n$. In any state where an event's boolean becomes true (meaning the event is happening in this state) its time stamp will be overwritten to be the highest value, indicating that this event happened most recently. There are four constraints which can be summarized as follows: 1. The order of the time stamps is always preserved from state to state excluding time stamps who's triggering event happens (this is how order changes), 2. The oldest event (happened first) is given time stamp = 0, 3. During the state in which the event happens, the time stamp is set to n (this is why $0..n$ is needed instead of $1..n$), and 4. There are no gaps in the ordering of the time stamps except time stamps being set in the current state—that is in the next state, the time stamps currently set to n will be set to some value less than n. For more details read section 6.1 of [Tsuchiya and Schiper 2007]. One major difference between our specification and theirs is that we do not use a unit delay for events;

events that become true will set the time stamp in *this* state rather than the next.

This module is not uniquely made for our system rather it is a general framework suggested by [Tsuchiya and Schiper 2007]. It is virtually impossible to hand write the code given the invariant and transitional relations laid out, certainly for large numbers of events. So a C program was developed to automate the process of creating this module. The program creates a module capable of handling n events as given by the input. Our particular model required 4 events: ga_action_join_user, ga_action_leave_user, ga_action_add_object, and ga_action_remove_object. We only need these four because these are the only events' time stamps tested in the policy for the TRM. The actual module has no idea which event is which so it is up to the CC to remember how it instantiates the TimeStamps module and then query the correct time stamp when needed. Since the TRM solely relies on the CC for its time stamps, the TRM is completely unaware of this module.

Well-Formedness Constraint:
The well-formedness constraints are enforced through the INVAR statements in the UI, CC_Object, and CC_User modules.

D.3 Counter-Examples

We discuss the counter-example traces generated by NuSMV while verifying various properties that are specified in the NuSMV model. For readability, we use Authz_{CC} and $\text{Authz}_{\text{TRM}}$ in the LTL specification shown here to denote the corresponding LTL formulas shown in section 3.2.1. However, in the NuSMV code for verification, we used the actual LTL formulas shown in section 3.2.1. This is not the exact output from NuSMV, it is abbreviated to show only the relevant parts. The actions *leave*, *join*, *add*, *remove*, *rt* (refresh), *perform*, and *request* (request perform) and values for *authztrm*, *authzcc* are shown. These are macros in the main module, and are all set from values from their respective module. Not shown are all of the necessary intermediate values, such as timestamps and state values for each of the modules.

D.4 Verification against Δ_0-system

D.4.1 *Verification of Unenforceablity Theorem*

```
-- specification  G (perform ->  Y ((!perform & (!rt | (rt & ((!remove & !leave)
S (add & (!leave S join)))))) S ((((!remove & !leave) S (add & (!leave S join)))
& ((!request & !perform) S request))))  is false

State: 1.1

leave = 0
join = 0
add =  0
remove = 0
authzcc = 0
athztrm = 0
rt = 0
perform = 0
```

```
request = 0

State: 1.2

leave = 0
join = 1
add =  1
remove = 0
authzcc = 1
athztrm = 1
rt = 1
perform = 0
request = 0

State: 1.3

leave = 1
join = 0
add =  0
remove = 1
authzcc = 0
athztrm = 1
rt = 0
perform = 0
request = 1

State: 1.4

leave = 0
join = 1
add =  1
remove = 0
authzcc = 1
athztrm = 1
rt = 0
perform = 1
request = 0
```

As shown in the counter-example above, NuSMV successfully proves that Authz_{CC} is not enforceable at TRM_Unsafe. That is, $\Delta_0 \not\models \Box(\text{perform} \rightarrow \varphi_0')$. In state 2, the user is joined, the object is added, and a refresh happens. This causes both *authzcc* and *authztrm* to be true. In the third state, the user leaves and the object is removed, without a refresh (rt = 0). This causes *authzCC* to become false, while *authztrm* remains true. Also in the third state, the user requests and the TRM responds to the request. The user is allowed to perform in the next (fourth) state because in the third state *authztrm* is still true. This shows that although *authzcc* is false the user is still allowed to perform showing that *authzcc* is not enforceable.

D.4.2 *Verification of Enforceability Theorem*

```
-- specification G(perform -> Y ((!perform & (!rt | (rt &
authzcc))) S (authztrm & ((!request & !perform) S request))) is true
```

NuSMV successfully verifies $\Box$(perform $\rightarrow \varphi_0$) against the Δ_0-system as shown above.

D.4.3 *Verification of Weak Stale-Safety against Δ_0-system*

```
-- specification  G (perform -> ( Y ((!perform & (!rt | (rt & ((!remove &
!leave) S (add & (!leave S join)))))) S (request & (!rt S (rt & ((!remove &
!leave) S (add & (!leave S join))))))) |  Y ((!perform & !rt) S ((rt & ((!remove
& !leave) S (add & (!leave S join)))) & (!perform S request)))))  is false

State: 2.1

leave = 0
join = 0
add =  0
remove = 0
authzcc = 0
athztrm = 0
rt = 0
perform = 0
request = 0

State: 2.2

leave = 0
join = 1
add =  0
remove = 0
authzcc = 0
athztrm = 0
rt = 1
perform = 0
request = 0

State: 2.3

leave = 1
join = 0
add =  1
remove = 0
authzcc = 0
athztrm = 1
rt = 0
perform = 0
```

```
request = 1

State: 2.4

leave = 0
join = 1
add =  0
remove = 1
authzcc = 0
athztrm = 1
rt = 0
perform = 1
request = 1

State: 2.5

leave = 1
join = 0
add =  1
remove = 0
authzcc = 0
athztrm = 1
rt = 0
perform = 1
request = 0
```

The counter-example generated by NuSMV clearly shows that the Δ_0-system fails the weak stale-safety security property. In state 2 the user is joined and a refresh occurs, but since the object has not been added yet, both *authzcc* and *authztrm* remain false. In state 3, the object is added and the user leaves making *authzcc* false. However the object super distributed to the TRM and thus the *authztrm* becomes true (since there was no refresh to update the user_leave_ts). In this same state the user requests and is allowed to perform (in the next state). It's irrelevant that *authzcc* does not match *authztrm*. The important aspect is that there is *no* refresh where *authztrm* held before allowing the user to perform int state 4.

D.5 Verification against Δ_1-system

D.5.1 *Weak Stale-Safety*

```
-- specification  G (perform -> ( Y ((!perform & (!rt | (rt &
((!remove & !leave) S (add & (!leave S join)))))) S (request &
(!rt S (rt & ((!remove & !leave) S (add & (!leave S join))))))) |
Y ((!perform & !rt) S ((rt & ((!remove & !leave) S (add &
(!leave S join)))) & (!perform S request)))))  is true
```

NuSMV successfully verifies that the Δ_1-system satisfies the weak stale-safe security property.

D.5.2 *Strong Stale-Safety*

```
-- specification  G (perform ->  Y ((!perform & !rt) S ((rt & ((!remove &
!leave) S (add & (!leave S join)))) & (!perform S request))))  is false

State: 2.1

leave = 0
join = 0
add =  0
remove = 0
authzcc = 0
athztrm = 0
rt = 0
perform = 0
request = 0

State: 2.2

leave = 0
join = 1
add =  1
remove = 0
authzcc = 1
athztrm = 1
rt = 1
perform = 0
request = 0

State: 2.3

leave = 0
join = 0
add =  0
remove = 1
authzcc = 0
athztrm = 1
rt = 0
perform = 0
request = 1

State: 2.4

leave = 1
join = 0
add =  0
remove = 0
authzcc = 0
```

```
athztrm = 1
rt = 0
perform = 0
request = 0

State: 2.5

leave = 0
join = 1
add =  1
remove = 0
authzcc = 1
athztrm = 1
rt = 1
perform = 1
request = 1
```

As shown in the counter-example generated by NuSMV above, the Δ_1-system fails strong stale-safety. In State 2 the the object is added, the user joins and a refresh occurs. The refresh causes both user_join_ts and object_add_ts to be updated at the TRM thus both *authzcc* and *authztrm* hold in state 2. In State 3, the user requests, the object is removed, the user leaves and the TRM responds to the users request. Because there is no refresh, *authztrm* remains true and the user is then allowed to perform in State 4. Strong stale-safety fails because there was no refresh by the TRM between the request in State 3 and perform in State 4.

D.6 Verification against Δ_2-system

D.6.1 *Strong Stale-Safety*

```
-- specification  G (perform ->  Y ((!perform & !rt) S ((rt & authzcc)
& (!perform S request)))) is true
```

As shown in the NuSMV report above, the Δ_2-system satisfies strong stale-safety.

D.7 NuSMV Code Listing

The following sub-section contains the entire code listing for the models that were checked using the NuSMV (refcimatti2002nusmv) model checking tool. The following code represents 12 SMV modules: **Main**, **UI**, **CC_User**, **CC_Object**, **CC**, **Authz**, **AuthzTRM**, **TRM_Unsafe**, **TRM_Weak**, **TRM_String**, **TimeStamps2**, and **TimeStamps4**.

The code can be compiled by concatenating all of the following code into a single file. For the **Main** module to work properly, *only* one TRM module should be included in that file (i.e. one of **TRM_Unsafe**, **TRM_Weak**, or **TRM_String**. Each of these TRM modules defines the "same" SMV module (TRM) which is used in the main module, hence only one can be included at a time. This should create three SMV files with a single main module, one for unsafe, one for weak, and one for strong. The only difference in each of these modules is the definition of the TRM.

D.7.1 *Main Module*

```
MODULE main
VAR
user: CC_User();
object: CC_Object();
cc: CC(user, object);
trm: TRM(cc, ui);
ui: UI(trm);

DEFINE
request := ui.request;
perform := ui.perform;
rt := trm.refresh;

authztrm := trm.authzTRM;
authzcc := cc.authzCC;

remove := object.req_remove_TS;
add := object.req_add_TS;
join := user.req_join_TS;
leave := user.req_leave_TS;


-------------------------simplified versions, but can't put this into NuSMV
--------------------

--perform ->
--LTLSPEC G(
--perform -> Y(
--!perform S (authztrm & ((!request & !perform) S
request))
--)
--);

--perform -> '
--LTLSPEC G(
--perform -> Y(
--(!perform & (!rt | (rt & (authzcc) ))) S
--((authzcc) & (!request & !perform) S request)
--)
--);

----------------------- this is where the ACTUAL formulas being tested start
-------------------

--perform ->
```

```
LTLSPEC G(
perform -> Y(
!perform S ((!rt S (add & (!rt S (rt & (!leave S join
))))) | (!rt S (rt & ((!remove & !leave) S add) & (!leave S join))) & ((!request
& !perform) S request))
)
);



--perform -> '

LTLSPEC G(
perform -> Y(
(!perform & (!rt | (rt & (((!remove & !leave) S (add &
(!leave S join)))) ))) S
((((!remove & !leave) S (add & (!leave S join)))) &
(!request & !perform) S request)
)
);




--weak-stale safety

--unsafe perform should fail, weak and strong should pass
LTLSPEC G(
perform -> Y ((!perform & (!rt | (rt &
((!remove & !leave) S (add & (!leave S join)))))) S (request & (!rt S (rt &
((!remove & !leave) S (add & (!leave S join)))))) |
Y ((!perform & !rt ) S ((rt &
((!remove & !leave) S (add & (!leave S join)))
) & (!perform S request))));




--strong-stale safety

--unsafe perform and weak perform should fail, strong should pass
LTLSPEC G(
perform -> Y ((!perform & !rt ) S ((rt &
((!remove & !leave) S (add & (!leave S join)))
) & (!perform S request))));
```

```
--strong check (strong peform should fail this)
LTLSPEC G(
!(perform & Y ((!perform & (!rt | (rt &
((!remove & !leave) S (add & (!leave S join)))
))) S (request & (!rt S (rt &
((!remove & !leave) S (add & (!leave S join)))
)))) &
!(Y ((!perform & !rt ) S ((rt &
((!remove & !leave) S (add & (!leave S join)))
) & (!perform S request))))));



--checking negations

--!(perform -> )
LTLSPEC G(
perform -> !Y(
!perform S ((!rt S (add & (!rt S (rt & (!leave S join
))))) | (!rt S (rt & ((!remove & !leave) S add) & (!leave S join))) & ((!request
& !perform) S request))
)
);



--weak stale-safety
LTLSPEC G(
perform -> !(Y ((!perform & (!rt | (rt &
authzcc))) S (request & (!rt S (rt & authzcc)))) |
Y ((!perform & !rt ) S ((rt & authzcc) & (!perform S
request)))));



--strong stale-safety
LTLSPEC G(
perform -> !Y ((!perform & !rt ) S ((rt & authzcc) & (!perform S
request))));
```

D.7.2 *UI Module*

```
MODULE UI(trm)
```

```
VAR
state : {idle, objAvail, waiting};

getObj : boolean;
request : boolean;
ASSIGN
init(state) := idle;

next(state) := case
getObj : objAvail;
request : waiting;
trm.succeed | trm.fail : objAvail;
TRUE: state;
esac;

DEFINE
perform := trm.succeed;

INVAR --never get object or request while waiting
!(getObj & state = waiting);
INVAR
!(request & state = waiting);
INVAR --never request while idle
!(request & state = idle);
```

D.7.3 *CC_User Module*

```
 MODULE CC_User()
VAR
state : {nonMember, member};
req_join_TS : boolean;
req_leave_TS : boolean;

ASSIGN
init(state) := nonMember;
next(state) := case
(req_join_TS) : member;
(req_leave_TS) : nonMember;
TRUE : state;
esac;

INVAR
!(state = member & req_join_TS) &
!(state = nonMember & req_leave_TS);
```

D.7.4 *CC_Object Module*

```
 MODULE CC_Object()
VAR
```

```
state : {nonMember, member};
req_add_TS: boolean;--these both come from GA
req_remove_TS: boolean;
ASSIGN
init(state) := nonMember;
next(state) := case
(req_add_TS) : member;
(req_remove_TS) : nonMember;
TRUE : state;
esac;

INVAR --never request add while already added
!((state = member) & req_add_TS);

INVAR --never request leave while already not a member
!(state = nonMember & req_remove_TS);
```

D.7.5 *CC Module*

```
MODULE CC(user, object)

VAR
timestamps : TimeStamps4(user.req_join_TS, user.req_leave_TS,
object.req_add_TS,
object.req_remove_TS,0,0,0,0);

_authzCC : Authz(user_join_TS, user_leave_TS, object_add_TS,
      object_remove_TS);

super_distro : boolean;

ASSIGN
init(super_distro) := FALSE;


--super distribution always happens when an add event occurs
--(in the next state)
next(super_distro) := case
next(object.req_add_TS) : TRUE;
TRUE: FALSE;
esac;

DEFINE
authzCC := _authzCC.authz;

user_join_TS := timestamps.ats1;
user_leave_TS := timestamps.ats2;
object_add_TS := timestamps.ats3;
```

```
object_remove_TS := timestamps.ats4;
object_add_event := object.req_add_TS;
```

D.7.6 *Authz Module.* This module represents Authz_{CC}:

```
 MODULE Authz(user_join_TS, user_leave_TS, object_add_TS, object_remove_TS)
DEFINE
authz := user_join_TS > user_leave_TS &
user_join_TS <= object_add_TS &
object_add_TS > object_remove_TS;
```

D.7.7 *AuthzTRM Module.* This module represents the evaluation of $\text{Authz}_{\text{TRM}}$.

```
 --By setting withSuperDistro = TRUE, you get the evaluation of authzTRM
--if you accessed the last time stamps for the super distributed object
--unless a refresh happens after the super distribution occured


MODULE AuthzTRM(cc, refresh, withSuperDistro)

VAR
authzTRM: boolean;

isMember: boolean;

ASSIGN
init(isMember) := FALSE;

--the status of whether the user is a member
-- only changes during refresh
next(isMember) := case
next(refresh) : next(cc.user_join_TS) >
next(cc.user_leave_TS);
TRUE: isMember;
esac;

init(authzTRM) := FALSE;

--if withSuperDistro = FALSE, then authzTRM is ONLY set
-- after a refresh.
next(authzTRM) := case
next(refresh) : next(cc.authzCC);
withSuperDistro & next(cc.super_distro) : case
next(isMember) : TRUE;
TRUE : FALSE;
esac;
TRUE : authzTRM;
esac;
```

```
--DEFINE
--refresh := trm.refresh;
```

D.7.8 *Unsafe TRM Module (Δ_0)*

```
 MODULE TRM(cc, ui)
VAR
state : {ready, authorized, refreshed};

authzTRMwithSD : AuthzTRM(cc, refresh, TRUE);

fail: boolean;
succeed: boolean;
refresh :boolean;

ASSIGN
init(state) := ready;

next(state) := case

TRUE : state;
esac;

init(fail) := FALSE;

next(fail) := case
ui.request & !authzTRM : TRUE;
TRUE : FALSE;
esac;

init(succeed) := FALSE;
next(succeed) := case
ui.request & authzTRM : TRUE;
TRUE: FALSE;
esac;

DEFINE
authzTRM := authzTRMwithSD.authzTRM;
```

D.7.9 *Weakly Safe TRM Module (Δ_1)*

```
 MODULE TRM(cc, ui)
VAR
state : {ready, authorized, refreshed};

authzTRMwithSD : AuthzTRM(cc, refresh, TRUE);

fail: boolean;
succeed: boolean;
```

```
refresh :boolean;

ASSIGN
init(state) := ready;

next(state) := case

TRUE : state;
esac;

init(fail) := FALSE;

next(fail) := case
ui.request & !authzTRM : TRUE;
TRUE : FALSE;
esac;

init(succeed) := FALSE;
next(succeed) := case
ui.request & authzTRM : TRUE;
TRUE: FALSE;
esac;

DEFINE
authzTRM := authzTRMwithSD.authzTRM;
```

D.7.10 *Strongly Safe TRM Module (Δ_2)*

```
 MODULE TRM(cc, ui)
VAR
state : {idle, requested, refreshed};

authzTRMwithSD : AuthzTRM(cc, refresh, TRUE);

fail : boolean;
succeed : boolean;

refresh : boolean;

ASSIGN

init(state) := idle;

next(state) := case
state = idle : case

--if state = idle
--ui.request can ONLY be true if
```

```
--  state = idle...
ui.request : case
--if refresh goto refreshed else to
-- requested
refresh : refreshed;
--else
TRUE : requested;
esac;
--else stay in idle state
TRUE : idle;
esac;

state = requested : case
refresh : refreshed;
TRUE : requested;
esac;

--else it MUST be in state = refreshed
-- therefore the TRM should respond and go back to idle
TRUE: idle;
esac;




init(succeed) := FALSE;

next(succeed) := case
state = refreshed & authzTRM : TRUE;
TRUE : FALSE;
esac;


init(fail) := FALSE;

next(fail) := case
state = refreshed & !authzTRM : TRUE;
TRUE : FALSE;
esac;

DEFINE
authzTRM := authzTRMwithSD.authzTRM;
```

D.7.11 *Timestamps Module for Two Events.* This is used in **TRM_Weak** to order the refresh and the object add event.

```
 MODULE TimeStamps2(cp1, cp2, ats01, ats02)
VAR
```

```
ats1 : 0..2;
ats2 : 0..2;

ASSIGN
init(ats1) := ats01;
init(ats2) := ats02;
--transition and invariant relationship as described in "Model Checking of
Consensus Algorithms", Tsuchiya and Schiper, Section 6.1
TRANS
((!next(cp1) & !next(cp2)) -> (
((ats1 = ats2) -> (next(ats1) = next(ats2))) &
((ats1 < ats2) -> (next(ats1) < next(ats2))) &
((ats1 > ats2) -> (next(ats1) > next(ats2)))
));
TRANS
( next(cp1) <-> (next(ats1) = 2) ) &
( next(cp2) <-> (next(ats2) = 2) )
INVAR
(
(ats1 != 2) |
(ats2 != 2)
) -> (
(ats1 = 0) |
(ats2 = 0)
)
INVAR
( (ats1 < ats2) & (ats2 != 2) -> (
( (ats1  + 1) = ats2 ) ))
&
( (ats2 < ats1) & (ats1 != 2) -> (
( (ats2  + 1) = ats1 ) ))
```

D.7.12 *Timestamps Module for Four Events.* This is used in the **CC** module to order the four events of JoinTS, LeaveTS, AddTS, and RemoveTS.

```
 MODULE TimeStamps4(cp1, cp2, cp3, cp4, ats01, ats02, ats03, ats04)
VAR
ats1 : 0..4;
ats2 : 0..4;
ats3 : 0..4;
ats4 : 0..4;

ASSIGN
init(ats1) := ats01;
init(ats2) := ats02;
init(ats3) := ats03;
init(ats4) := ats04;
--transition and invariant relationship as described in "Model Checking of
```

```
Consensus Algorithms", Tsuchiya and Schiper, Section 6.1
TRANS
((!next(cp1) & !next(cp2)) -> (
((ats1 = ats2) -> (next(ats1) = next(ats2))) &
((ats1 < ats2) -> (next(ats1) < next(ats2))) &
((ats1 > ats2) -> (next(ats1) > next(ats2)))
)) &
((!next(cp1) & !next(cp3)) -> (
((ats1 = ats3) -> (next(ats1) = next(ats3))) &
((ats1 < ats3) -> (next(ats1) < next(ats3))) &
((ats1 > ats3) -> (next(ats1) > next(ats3)))
)) &
((!next(cp1) & !next(cp4)) -> (
((ats1 = ats4) -> (next(ats1) = next(ats4))) &
((ats1 < ats4) -> (next(ats1) < next(ats4))) &
((ats1 > ats4) -> (next(ats1) > next(ats4)))
)) &
((!next(cp2) & !next(cp3)) -> (
((ats2 = ats3) -> (next(ats2) = next(ats3))) &
((ats2 < ats3) -> (next(ats2) < next(ats3))) &
((ats2 > ats3) -> (next(ats2) > next(ats3)))
)) &
((!next(cp2) & !next(cp4)) -> (
((ats2 = ats4) -> (next(ats2) = next(ats4))) &
((ats2 < ats4) -> (next(ats2) < next(ats4))) &
((ats2 > ats4) -> (next(ats2) > next(ats4)))
)) &
((!next(cp3) & !next(cp4)) -> (
((ats3 = ats4) -> (next(ats3) = next(ats4))) &
((ats3 < ats4) -> (next(ats3) < next(ats4))) &
((ats3 > ats4) -> (next(ats3) > next(ats4)))
));
TRANS
( next(cp1) <-> (next(ats1) = 4) ) &
( next(cp2) <-> (next(ats2) = 4) ) &
( next(cp3) <-> (next(ats3) = 4) ) &
( next(cp4) <-> (next(ats4) = 4) )
INVAR
(
(ats1 != 4) |
(ats2 != 4) |
(ats3 != 4) |
(ats4 != 4)
) -> (
(ats1 = 0) |
(ats2 = 0) |
```

```
(ats3 = 0) |
(ats4 = 0)
)
INVAR
( (ats1 < ats2) & (ats2 != 4) -> (
( (ats1  + 1) = ats2 ) |
( (ats1  + 1) = ats3 ) |
( (ats1  + 1) = ats4 ) ))
&
( (ats1 < ats3) & (ats3 != 4) -> (
( (ats1  + 1) = ats2 ) |
( (ats1  + 1) = ats3 ) |
( (ats1  + 1) = ats4 ) ))
&
( (ats1 < ats4) & (ats4 != 4) -> (
( (ats1  + 1) = ats2 ) |
( (ats1  + 1) = ats3 ) |
( (ats1  + 1) = ats4 ) ))
&
( (ats2 < ats1) & (ats1 != 4) -> (
( (ats2  + 1) = ats1 ) |
( (ats2  + 1) = ats3 ) |
( (ats2  + 1) = ats4 ) ))
&
( (ats2 < ats3) & (ats3 != 4) -> (
( (ats2  + 1) = ats1 ) |
( (ats2  + 1) = ats3 ) |
( (ats2  + 1) = ats4 ) ))
&
( (ats2 < ats4) & (ats4 != 4) -> (
( (ats2  + 1) = ats1 ) |
( (ats2  + 1) = ats3 ) |
( (ats2  + 1) = ats4 ) ))
&
( (ats3 < ats1) & (ats1 != 4) -> (
( (ats3  + 1) = ats1 ) |
( (ats3  + 1) = ats2 ) |
( (ats3  + 1) = ats4 ) ))
&
( (ats3 < ats2) & (ats2 != 4) -> (
( (ats3  + 1) = ats1 ) |
( (ats3  + 1) = ats2 ) |
( (ats3  + 1) = ats4 ) ))
&
( (ats3 < ats4) & (ats4 != 4) -> (
( (ats3  + 1) = ats1 ) |
```

```
( (ats3  + 1) = ats2 ) |
```