

## Reasoning about Hybrid System of Systems Designs

M.T. Gamble      R.F. Gamble

*Department of Computer Science*

*University of Tulsa*

*michael-gamble@utulsa.edu, gamble@utulsa.edu*

### Abstract

*Constructing a complex system-of-systems (SoS) involves integrating two or more components. When integrations overcome isolating mechanisms inherent to heterogeneous components, a SoS is a software hybrid. SoS designers are challenged to create a viable hybrid that reuses significant value from autonomous, component systems while allowing for new, SoS-wide properties to emerge and be reliably maintained. Thus, a SoS is a super-system integrating many designs, yet not identical to any of them. Formalisms beyond architecture definition languages for mismatch are needed to express SoS designs to terms of the properties and structures that promote the determination of the cause of hybrid failure and its resolution. This paper extends a formal specification language to SoS designs within a paradigm based on software speciation, where software systems experience divergent evolution under various forms of isolation to become incompatible. Examples from security audit are used to illustrate the application of the formalism.*

### 1. Introduction

Reuse of existing software components causes the emergence of the detailed interactions of programs and systems with their environment and the local mechanisms that impact their compatibility. Thus, both recurring behavior and integration issues surface as part of the expression of the component interaction. Previously [1], we discuss how the use of commercial-off-the-shelf (COTS) software can impose constraints on designs, where those constraints are inherited from COTS components into the systems that use them. Likewise, reuse of systems implies similar constraints on a system of systems (SoS).

We invoke a conceptual metaphor of the isolated evolution of biological species [2], [3] as a design metaphor that is useful in understanding how software elements that were developed independently have divergent and incompatible designs [1]. When systems are developed in isolation, their design solutions naturally vary from one another. Such variances result from forces distinct to a system's local environment or *context* and the scope of its execution. These forces shape design decisions during development and evolution. If variations are significant, the systems resemble natural species in that they have localized *isolating mechanisms* (specific to their environment). Building a non-trivial SoS requires the construction of *hybrids* with components coming from a variety of isolated sources.

Tools are needed to handle this design complexity, while also providing essential reasoning capability over component designs to detect and defeat isolating mechanisms. One potential tool is a framework for formal analysis of software hybrids. However, this requires the representation of a SoS in such a way that subsystems still retain some individual identity yet are scoped accordingly, while composed to reliably and robustly interact at an intersystem level. Modeling languages, such as [4] and [5], and distributed system languages, such as [6, 7], do not capture such context-dependent isolation in a SoS hierarchy. Thus, modeling language extensions are needed to existing formal representations to express and reason about hybrid system design properties.

In this paper, we discuss SoSs as hybrid systems, subject to isolating mechanisms that inhibit interoperability. We specify and identify isolation in candidate hybrid system designs as well as provide a framework for correcting them with design solutions that result in robust and reliable hybrid SoSs. We introduce X-UNITY (pronounced *Cross-UNITY*), an extension of Context UNITY [8], to capture

programmatic, structural, and scoping properties of hybrid SoSs. An example involving security audit functions is used to illustrate X-UNITY's potential.

## 2. Relevant Background

In this section, we review software isolation and background related to definitions of SoSs to conclude that SoSs are hybrid systems. We give a brief foundation for our formal language choice and its extensions representing and reasoning about SoSs.

### 2.1. Isolation in Software Systems

The term *isolating mechanism* was first used to describe properties of biological systems [9] where gene exchange is limited between species by blocking reproduction. For software, isolating mechanisms impair integration of systems from distinct sources. Difficulties arise when trying to classify and scope isolation types in order to discover isolating mechanisms. Table 1, revised from its first introduction in [1], partitions types of isolating mechanisms across pre- and post-integration phases within the framework of an integration lifecycle. The classification is similar to the theory of biological evolution [9], but recast into a software viewpoint.

Examining the pre-integration phase in Table 1, *structural isolation* occurs when message transmission is thwarted due to lack of connection, e.g. when a

physical pathway through a network is not available, or when the architectural styles [10] of components are mismatched. *Ecological isolation* arises when adaptations to foster execution in a previous environment remain with the component in the new environment.

Even with successful message transfer, isolating mechanisms exist that prevent integration and cause message processing to fail. *Interface isolation* occurs when a message is incompatible with the expectations of the receiving component. *Protocol isolation* results when the behavior required by one component differs from what is expected by another.

Continuing the biological analogy, once the isolating mechanisms in the pre-integration phase have been defeated, a *hybrid* is formed. The robustness of the resulting hybrid is determined by its *viability* (i.e., capability to execute properly) and its *fertility* (i.e., ability to participate in further compositions). In biology, hybrids that are inviable or have reduced viability may die out.

Inviability of a SoS goes beyond simple mismatch problems and requires a deeper understanding of both component and environment factors. Properties or side effects may emerge from the composition, to cause the hybrid to execute incorrectly, including new functions incorporated to defeat pre-integration isolating mechanisms. If the hybrid is not usable in further compositions, it is *sterile*. This is directly analogous to its biological meaning.

**Table 1: Forms of Isolation**

	INTEGRATION PHASE	ISOLATION TYPE	DESCRIPTION
PRE- INTEGRATION	<i>Pre-message transfer</i>	<i>Structural</i>	No interconnection between components or interconnection limited to specific structural forms.
		<i>Ecological</i>	Preference or limitation to a specific partition of the same domain, non-concurrent activity; inability to synchronize or rendezvous
	<i>Post-message transfer</i>	<i>Interface</i>	Incompatibility in parameter types, structure or order; different variables refer to the same entity; the same variable names refer to different entities.
		<i>Protocol</i>	Message sequencing or composite actions do not agree. The logical meaning of similar protocol actions or sequences does not agree.
POST- INTEGRATION	<i>1<sup>st</sup> Hybrid</i>	<i>Inviability</i>	Incompatibility with the environment due to large scale ecological isolation. Internal inconsistencies violate requirements and behavior expectations.
		<i>Sterility</i>	Pre-message isolation disallows further communication. Internal problems with subsystems cause invalid behavior in post-message transfer.
	<i>2<sup>nd</sup> Hybrid</i>	<i>Breakdown</i>	The result of hybrid reuse with similar hybrids or parent system types is weak and subject to inviability or sterility

Inviability may be observed when components of a hybrid demand centralized control. When multiple components assume “ownership” of system wide control an isolating mechanism emerges. Sterility can occur when required interfaces fail to be exposed for composition. Either their variables are private to a system or are covered by being already used up in other interactions intrinsic to the system.

A second generation hybrid takes a first generation hybrid and composes it with another individual from one of its parent species. The assumption is that the first generation hybrid is “fit” (i.e., viable and not sterile). The common reuse of a hybrid as a software component can result in further revelations of integration problems, known as hybrid *breakdown*, where isolating mechanisms surface that were dormant in other compositions and now cause inviability or sterility.

## 2.2. A System of Systems as a Hybrid

A SoS retains constructs that are derived from its component systems. A recurring theme in the description of a SoS is that it is formed by reusing instances of existing systems, that is, reuse as a service as opposed to reuse as a copy. These restrictions imply that candidate systems for inclusion in the SoS are isolated prior to the SoS being composed. For example, centralized functions may need to be merged across competing/cooperating sub-systems. However, systems which are good candidates for reuse in SoSs have little or no centralized control, facilitating the formation of composites [11]. Therefore, constructs, such as centralized control, are strong indicators of isolation.

A SoS is often defined by describing its characteristics. Maier [12] describes a SoS as having (1) operational and managerial independence of elements, (2) evolutionary development with emergent behaviors, and (3) large scale distribution. Maier’s first characteristic has significant implications for governance that can potentially create complex and unwieldy situations where system elements are highly independent. For instance, “operational independence” implies an ongoing isolation at least at some levels. Morris [11] provides a set of governance characteristics considered “good practice” for SoSs. It should be the case that (1) collaboration and authority are distributed, (2) motivation and accountability are shared, (3) support for multiple governance models is provided, (4) expectations of evolution are understood, (5) highly fluid processes are assumed, and (6) minimal centrality demands are made.

Modern implementations of software grids by definition [13] embody the implementation of a SoS for stakeholders in the scientific community. Multiple, existing computational and/or data resources are knitted together to form larger SoSs in the form of grids. Foster [13] provides a checklist of characteristics to define a grid as something that (1) coordinates resources that are not subject to centralized control, (2) uses standard, open, general-purpose protocols and interfaces, and (3) delivers nontrivial qualities of service. Note that the implication of Foster’s criteria is that functional capabilities of the resources are highly autonomous (not subject to centralized control) yet the infrastructure characteristics, such as the underlying communications and coordination technologies (i.e., middleware) are subject to the constraints of being “standard, open” and “general purpose”.

These descriptions imply that a SoS is a hybrid and one that is likely at risk of being inviable or sterile. This situation varies based on the overall SoS goals and the system elements used. For example, it is assumed that systems on a grid forming a SoS actually utilize the same substrate of technological interface standards in the form of common middleware. Grid systems “inherit” their interface designs from a common source that exists in the grid middleware and is reused in each grid application. The desirable outcome of this is that it would seem more likely that resulting SoS designs would be able to integrate more “deeply” and thereby yield Foster’s third criteria of delivering non-trivial qualities of service. This requires careful use of Morris’ first point, management of collaboration and authority in governance, to maintain a consistent middleware throughout the population of grid systems potentially reusable in SoS designs.

Foster also implies prescriptive control through coordination of the behavior of the overall SoS. A grid’s purpose is explicit, not emergent as Morris and Maier argue. This might imply that grids are a subset of a larger, more complex SoS viewpoint developing in the research community. In either case, hybrids are required because the component systems used come from isolated development efforts spread across autonomous organizations. Despite this very loose coupling, the resulting SoS needs to satisfy overall properties. Formal specification allows reasoning about such composite properties.

## 2.3. Formal Languages

Specification formalisms for software hybrids must portray hierarchical composition, where intermediate results can be formed and then further composed. Current languages do not focus on this need in the

context of reuse Membranes [14] represent domains and migration of program entities across domains. Pi-calculus [6] provides a process algebra for reasoning about process-centric system specifications. Mobile Ambients [15] model processes and capture their movement between administrative domains. Architecture definition languages address structural relationships and interface matching [5].

We focus on Context UNITY [8] which extends UNITY’s programming model [16] to include distribution and interactions with an operational environment through context programs. The primary unit of specification in Context UNITY is the program. Context UNITY represents systems in both an imperative manner (using actual program statements) and a declarative manner (stating program properties). In addition to its specification constructs, it contains an execution model and a proof logic that allows temporal reasoning. We review Context UNITY to the extent that is needed for our extension and examples. Figure 1 shows its basic structure.

Context UNITY program,  $P$ , describes a state transition system consisting of variable declarations (**declare**), initial values for variables (**initially**), and assignment statements (**assign**). Statements are executed with weak fairness in that they are executed non-deterministically, infinitely often.

---

```

System SystemName
Program P_{<parms>}
  declare
    exposed // public variables
    internal // private variables
    context // handles to other public vars
  initially // initial values for public/private vars
  assign // programmatic state changes
  context // programs that use context variables
           // to interact with environment
end P
  Components // the instantiated programs in the system
  Governance // global impact statements
end SystemName

```

---

**Figure 1: Context UNITY Specification Structure**

The **declare** section is divided into variable types for **exposed**, **internal**, and **context**. The **context** program following the **assign** section specifies how changes in the environment state are reflected in the values of exposed variables, which in turn can influence another program’s context variables. Thus, the context program provides components with explicit and individualized interactions within their contexts. The **Components** section is used to define **Program** instances. The **Governance** section contains rules for

behaviors that have a global impact on the system. These rules rely on the state of exposed variables throughout the larger system to affect other exposed variables in the system.

Given that programs are actually code, they must be instantiated to form a system. A system may “run” many instances of a program. Program instances in Context UNITY are distinguished by passing parameters (depicted by  $\langle parms \rangle$ ) during system initialization that includes a unique instance identifier. Thus, Context UNITY provides an initial foundation for structuring the specification of hybrids and, explicitly, SoS designs.

## 4. X-UNITY

Given SoS characteristics, we must adhere to a specification framework that can indicate isolation in component design. This means we need a structure that reflects the concepts of scope, interaction, and reuse while providing mechanisms to support reasoning and proof. The framework requirements should (1) allow multiple architectures for governance and control [11], (2) represent component layering and hierarchies, (3) include imperative and declarative viewpoints, (4) express abstract design and their instantiations, (5) specify different interaction styles (e.g. explicit, implicit, indirect) and (6) depict the concept of reuse of existing program and system types to model the inheritance of software behaviors from classes or species of software artifacts.

### 4.1. Extending Context UNITY

In this paper, we focus on three important extensions of Context UNITY to define X-UNITY (pronounced *Cross-UNITY*).

- (1) We formally induce a hierarchy of module specifications to represent the SoS configuration by augmenting the specification labels of **System** and **Program** with **SoS**.
- (2) We differentiate explicitly between a system design specification and an instance of the specification as a particular use of the design to reflect the concept of reuse. This is done with the introduction of **include** for reusing an encapsulated entity and **System Instances** as the instantiation of **Systems**.
- (3) We allow program variable exposure outside of the scope of the reused system through the introduction of **promote**.

These extensions allow X-UNITY to express a SoS so that it can be reasoned about in the context of other systems, not just programs.

Figure 2 shows the basic structure for X-UNITY specifications, illustrating the notation and hierarchy extensions. Though **System** specifications are similar to Context UNITY, our introduction of **include** lets us refer to programs that may be specified elsewhere. Thus, **include** allows the module name to represent its entire specification template. This convention leads to simpler specifications of higher-level systems and provides a consistent notation for reuse using module names. It also makes system composition explicit in X-UNITY, which facilitates reasoning about hybrids.

A similar convention is introduced at the **SoS** level that encompasses all modular entities. Where a **Program** serves as a template for instances of **Components** in Context UNITY, we extend this approach to include the instantiation of **System** specifications into particular **System Instances**. Thus, when we name a system template as **include System**, this name serves as a reuse symbol in another SoS. When we give an instance of a template a unique identifier as in **Components** or **System Instances**, that instance may be referenced explicitly during execution.

This makes a **System** a type, while a particular **System Instance** is a parameterized occurrence of that type. A **SoS** describes a particular interacting set of system instances. If a SoS is to be reused as a component of another system, it too can be considered a **System** type if needed.

---

```

System SystemName
  include Program ProgramName1
  include Program ProgramName2
  ...
  Components // specific program instances
    // ProgramName1(1),...,ProgramName(n)
  Governance
    promote x as w in *
end SystemName

SoS SoSName
  include System SystemName1
  ...
  System Instances
    // configuration of specific system
    // instances within the SoS
end SoSName

```

**Figure 2: X-UNITY Basic Specification Structure**

---

The notational extension of **promote** alters the scope of the exposed variable  $x$  to include all systems

(\*). **Promote** makes  $x$  available under the alias  $w$  in a system  $s$  such that  $w \in s.\text{exposed}$ ; where  $s.\text{exposed}$  is the set of names for all exposed variables available in  $s$ . Alterations to the **var** [ ] table in Context UNITY for defining exposed variables are needed to formally represent **promote**. For  $x$ 's scope to now include  $s$  means that a **Program** in  $s$  can select the variable  $x$  using its attributes, including its aliases, in the **Program context** rules.

**Promote** appears in the **Governance** section of a **System**. Recall that systems are allowed to have system-wide governance, while SoSs are not. This approach conforms to the definitions of a SoS as a collection of autonomous systems. Thus, **promote** is a core concept for X-UNITY to provide a form of selective composition. It helps capture the unique compositional properties of systems in a SoS. If two systems are formally composed in UNITY, it is done by a union theorem that forces all exposed variables to be public to other systems. The pairing X-UNITY's **promote** with Context UNITY's **uses** rules constrains this formal union to make it possible to defeat a number of isolating mechanisms by allowing more remote interactions to appear as if they are in a local environment. This induces a reaction based on context variables that are quantified over the local environment via **uses**. If such constructions are effective, they make for good hybrids, even when the elements of the composition are of different software species.

## 5. Modeling Hybrid SoSs in X-UNITY

An example from a distributed security audit serves as a vehicle for exercising the features of X-UNITY that we focus on for hybrid system specification and analysis. The NIST document SP 800-53 [17] recommends security controls for U.S. federal information systems and includes a number of sections across many topics including access control and privilege management. One particular set of controls, denoted AU, deals with audit and accountability. For our examples, we focus on AU requirements for handling auditable events shown in Table 2. These requirements apply both to simple systems as well as to complex SoS designs.

AU-2	The information system generates audit records for events per system as chosen by the organization.
AU-2(1)	The information system provides the capability to compile audit records from multiple components throughout the system into a system wide (logical or physical), time-correlated audit trail.
AU-2(2)	The information system provides the capability to manage the selection of events to be audited by individual components of the system.

**Table 2: NIST Security Controls for Audit**

## 5.1. Specifying a System in X-UNITY

We first specify a system of components that capture detected auditable events and generate event notifications within X-UNITY. We limit the code specification to only security audit properties.

Notifications are retained in an exposed variable, *notify*, for later review by auditors. The *notify* variable can be any type of local storage that is accessible to other programs. Here, it is a set of audit records, each of which is an ordered pair with a timestamp and audit information fields, such as the type of event and the component identifier. We assume the function *detectEvents()* returns the set of events detected in the local environment since it was last invoked. In *AuditableComponent* (Figure 3), once the events are saved in *notify* there are no further state changes.

---

```

Program AuditableComponent
  declare
    exposed
      notify: Set of AuditRecord
    initially
      notify :=  $\emptyset$ 
    assign
      notify := notify  $\cup$  detectEvents()
  end AuditableComponent

```

**Figure 3: The Program *AuditableComponent***

---

*AuditCollector* (Figure 4) uses a context variable, *auditCache*, to collect the notifications from components in its same system with the exposed variable, *notify*, where *notify* contains notifications that are not yet in the audit trail. Thus, it does not execute as a stand-alone component. The **uses** statement “loops” over all *p*, in which *n* is local to the scope of the loop. Effectively, component interfaces are advertised by their exposed variables and selected for use by the quantification of **uses** context rules.

In *AuditCollector*, variables named *notify* are selected from all programs *p* that satisfy the **given** clause and are bound to the handle *n*. The “!” notation is used to associate a temporary handle, *n*, to each

matched instance of *notify*. **Becomes** is assignment. The *auditCache* values are eventually assigned to the exposed variable *auditTrail* using the statement in the program’s **assign** section. Weak fairness of UNITY’s execution model assures that all statements are selected for execution infinitely often. Given the rules in its **context** program, other components, such as those that instantiate *AuditableComponent*, must provide their audit notifications as exposed variables for collection.

---

```

Program AuditCollector
  declare
    exposed
      auditTrail: Set of AuditRecord
    context
      auditCache: Set of AuditRecord
    initially
      auditTrail, auditCache :=  $\emptyset$ ,  $\emptyset$ 
    assign
      auditTrail := auditTrail  $\cup$  auditCache
    context
      auditCache
      uses n!notify in p
      given  $\neg(n \subseteq \text{auditCache})$ 
      where auditCache becomes auditCache  $\cup$  n
  end AuditCollector

```

**Figure 4: The Program *AuditCollector***

---

In Figure 5, we specify a **System** of the components in Figure 3 and Figure 4 using the **include** statement to indicate reuse by copy of the previously defined program types. Without **include**, we would have to repeat the entire program specifications within the system. Reuse results in more complex specifications because all relevant component information must be considered. Therefore, the benefit of **include** is that it provides a construct for better management of these complex system representations, while mimicking actual reuse and composition.

---

```

System CollectedAuditSystem
  include Program AuditableComponent
  include Program AuditCollector
  Components
     $\langle \square i :: \text{AuditableComponent}(i) \rangle$ 
     $\square \text{AuditCollector}$ 
  end CollectedAuditSystem

```

**Figure 5: The System *CollectedAuditSystem***

---

A program template (named in an **include** statement) is instantiated in the **Components** section as needed. Instances have unique identities that make

them available for later reuse as services by other systems in the **SoS** specification. The notation ‘ $\square i ::$ ’ means that there are ‘ $i$ ’ *AuditableComponents* that execute asynchronously, each with a unique identifier.

To show the compliance of *CollectedAuditSystem* with the requirement of AU-2(1) in Table 2, we formulate a UNITY progress property **leads-to** to generically state that eventually there is at least one component which has the complete representation of a system-wide audit trail. Because we have retained the UNITY execution model, we have use of its temporal proof logic. A “dot” notation expresses the hierarchy of modules and variable names within the X-UNITY specification.

$$\begin{array}{l} \exists c \in C \mid \hspace{15em} P1 \\ \langle \forall t \in C \mid e \in t.\text{notify} \text{ leads-to } e \in c.\text{auditTrail} \rangle \end{array}$$

Given the system *CollectedAuditSystem* the statement P1 reasons over all program components ( $\forall t \in C$ ) that are reused within a composite system, such that when  $t$  receives an event ( $e \in t.\text{notify}$ ) there is at least one component ( $\exists c \in C$ ) that will eventually acquire the event in its audit trail ( $e \in c.\text{auditTrail}$ ). Without the X-UNITY extensions, this statement would not be easily expressible or provable, and further reasoning about hybrids would prove difficult. It can be directly seen that *auditTrail* in *AuditCollector* of *CollectedAuditSystem* contains all notifications from all  $i$ , such that *AuditableComponent*( $i$ ), to comply with P1.

## 5.2. Specifying a SoS in X-UNITY

Figure 6 introduces a simple SoS specification as a composition of system specifications, showing X-UNITY’s capability to explicitly denote the structural relationship between a SoS and its component systems. In Figure 6, **SoS ASReplicas** includes multiple system instances of *CollectedAuditSystem* (Figure 5).

---

```

SoS ASReplicas
  include System CollectedAuditSystem
  System Instances
   $\langle \square i :: \text{CollectedAuditSystem}(i) \rangle$ 
end ASReplicas

```

**Figure 6: The SoS ASReplicas**

The constraints of X-UNITY force system-to-system interactions to be made explicit. This convention provides a degree of encapsulation where systems, much like biological systems, have an inherently defined individuality even if they interact in larger systems. Compositions occur via well defined

interfaces that are explicitly exposed and controlled by processes designed for those types of interactions.

In the case of **SoS ASReplicas** each individual *CollectedAuditSystem* can be proven to satisfy P1. They are conspecific in this respect. However, when reused as multiple instances of a system, **SoS ASReplicas** fails to satisfy P1. This is a classic case of an ecological isolating mechanism – each system expects to actively use external information but does not allow access to their internal information.

The result is the lack of a SoS-wide audit trail. This is because each *AuditCollector* within each *CollectedAuditSystem* maintains an audit trail with the records confined to events detected within each component system. This result is a deliberate artifact of modeling systems as autonomous entities that are, by default, closed. However, X-UNITY relies on explicitly declared shared variables to define system-to-system interactions at the SoS specification level.

To defeat this and similar habitat isolating mechanisms so that **SoS ASReplicas** complies with P1, we add to Figure 5 a **Governance** section and introduce **promote** to make an exposed variable explicitly visible to other systems. This results in **System CollectedAuditSystem-2** as shown in Figure 7.

---

```

System CollectedAuditSystem-2
  include Program AuditableComponent
  include Program AuditCollector
  Components
   $\langle \square i :: \text{AuditableComponent}(i) \rangle \square \text{AuditCollector}$ 
  Governance
  promote AuditCollector.auditTrail as notify in *
end CollectedAuditSystem-2

```

**Figure 7: System CollectedAuditSystem-2**

The **promote** statement elevates visibility of specific exposed variables from the **System** level to the **SoS** level. Its use requires clear design intent. Here P1 guided the choice of the *auditTrail* variable to be renamed as *notify* in order to allow its reference by existing logic in peer systems. In this case, all peer systems as denoted by ‘**in** \*’. The peer systems are selectable under quantification by **uses** statements in the context programs of peer systems within the SoS.

With this introduction of the **Governance** section and the **promote** statement, every *AuditCollector* component (Figure 4) can access the notifications from each *AuditableComponent* (Figure 3) within its respective system, as well as the top level audit trails of every other copy of *CollectedAuditSystem-2* (Figure 7) within the **SoS ASReplicas** (Figure 6). As an alternative to this centralized approach, *notify* in each

*AuditableComponent* could have been promoted. However, we choose a centralized design via the **Governance** section because it more closely follows the design approach of making system level interfaces explicit. Promoting *auditTrail* and renaming it to *notify* allows the system to be reused as a component in the hierarchy, while preserving P1 over system quantification.

Note that governance, or control, is either modeled explicitly within a special section of X-UNITY or implicitly through the cooperating code of multiple programs. By definition, it is rare to apply explicit governance to SoS models. Our Context UNITY extensions promote governance rules across the full spectrum of modeling unit granularities (programs, systems and systems-of-systems). For governance to occur explicitly, it must be implemented “outside” the modeling construct (e.g., within the environment) in question. For governance to occur in the SoS, each contributing system must accept some degree of outside control. This control can be in the form of various types of integration middleware that “glue” the systems together. The middleware itself is part of the SoS solution that can be modeled as (1) a subsystem in and of itself or (2) a “governance” function represented explicitly outside of the other SoS parts.

### 5.3. Hybrid Sterility

When all pre-integration isolating mechanisms are overcome (Table 1), a hybrid may be formed. Though, interactions occur in the SoS, sterility can cause the failure of further compositions. Sterility can occur if the exposed interfaces of a system are used up or “covered” making them inaccessible for further compositions. This is often a concern when reusing an encapsulated system within a SoS. There may be limitations on the cardinality of interfaces of the system. Sterility is revealed in a specific composition configuration. To illustrate this, we specify a program *LimitedAuditCollector* (Figure 8) with a practical limit on the number of audited components. The change from *AuditCollector* (Figure 4) is manifested in the context variable *client* that induces the limit.

In *LimitedAuditCollector*, context rules are used in two ways. Audit notifications are collected only from known *client* components to create an audit trail. The special variable  $\pi$  indicates the component identifier for a particular program instance. Clients become known via a simple selection rule that chooses them if they have not been chosen before and the total number of clients is less than or equal to 10. In this scenario,

*LimitedAuditCollector* cannot be effectively used in systems with more than 10 auditable components.

Figure 9 defines the **System** *LimitedAuditSystem*, that substitutes the **Program** *LimitedAuditCollector* for the **Program** *AuditCollector* in Figure 7. Note that the **promote** statement reflects the substitution.

A hybrid **SoS** *HybridLimitedCollected* is formed by composing the ‘collected’ (Figure 7) and ‘limited’ (Figure 9) to create an audited system.

---

```

Program LimitedAuditCollector
declare
  exposed
    auditTrail: Set of AuditRecord
  context
    auditCache: Set of AuditRecord
    clients: Set of ComponentID
  initially
    auditTrail, auditCache, clients :=  $\emptyset, \emptyset, \emptyset$ 
  assign
    auditTrail := auditTrail  $\cup$  auditCache
  context
    auditCache
      uses  $n!notify, i!\pi$  in p
      given ( $i \in clients$ )  $\wedge \neg(n \subseteq auditCache)$ 
      where auditCache becomes auditCache  $\cup$  n
    clients
      uses  $i!\pi$  in p
      given ( $i \notin clients$ )  $\wedge (|clients| \leq 10)$ 
      where clients becomes clients  $\cup \{i\}$ 
end LimitedAuditCollector
  
```

**Figure 8: Program *LimitedAuditCollector***

---

```

System LimitedAuditSystem
include Program AuditableComponent
include Program LimitedAuditCollector
Components
   $\langle \square i :: AuditableComponent(i) \rangle$ 
   $\square$  LimitedAuditCollector
Governance
  promote LimitedAuditCollector.auditTrail as notify
end LimitedAuditSystem
  
```

**Figure 9: System *LimitedAuditSystem***

---

---

```

SoS HybridLimitedCollected
  include System CollectedAuditSystem-2
  include System LimitedAuditSystem
  System Instances
    <[] j :: CollectedAuditSystem-2(j)>
    [] <[] k :: LimitedAuditSystem(k)>
end HybridLimitedCollected

```

**Figure 10: SoS *HybridLimitedCollected***

While the number of *AuditableComponents* may be controlled by the system in *LimitedAuditSystem* instances, its behavior when joined with other systems is not as certain. Given that all instances of *CollectedAuditSystem-2* should be viewed as auditable components by instances of *LimitedAuditSystem*, there must be enough client connections remaining in *LimitedAuditSystem* instances to see all components in the hybrid. If, as define above,  $i + j \geq 10$ , then no additional *CollectedAuditSystem-2* instances can be composed into the SoS. The cardinality constraint results in compositions using *LimitedAuditCollector* that can become sterile.

For any further composition involving *Hybrid-LimitedCollected* as a component system of other SoS, the same constraints disallow further components to have their audit trails collected into *LimitedAuditSystem* instances. The key isolating mechanism causing sterility in is the failure of the quantifier in the context rule for *clients* in the *LimitedAuditCollector* program. Once the set of clients is saturated at 10, this rule never fires again. To overcome sterility the limit can be redefined, but that is subject to influences of the system's owners and other constraints. Similar constraints influence the design of *LimitedAuditCollector*, adapting it to relatively small systems.

#### 5.4. Hybrid Inviability

Beyond sterility is the concern that the resulting hybrid is viable (i.e., it meets required liveness criteria). To support reasoning about hybrid systems, X-UNITY must represent hybrid designs that may not be correct under tests for viability. Thus, our formalism explicitly allows the description of different species of software within the same specification that may be isolated.

Extending the audit example further, we introduce multiple systems that satisfy the requirements yet do so using different algorithms and system structures. These systems are composed into a hybrid SoS which may be examined to confirm or deny whether the SoS also satisfies the requirements at the hybrid system level.

```

Program AuditConsumer
  declare
    exposed
      auditTrail: Set of AuditRecord
    context
      auditCache: Set of AuditRecord
  initially
    auditTrail := auditCache :=  $\emptyset$ 
  assign
    auditTrail := auditTrail  $\cup$  auditCache :
    auditCache :=  $\emptyset$ 
  context
    auditCache
    uses n!notify in p
    given  $\neg(n \subseteq \text{auditCache})$ 
    where auditCache becomes auditCache  $\cup$  n
     $\emptyset$  impacts n
end AuditConsumer

```

**Figure 11: Program *AuditConsumer***

The program *AuditConsumer* (Figure 11) reproduces much of the logic of *AuditCollector* (Figure 4). It differs by ‘consuming’ the audit events once they are copied to the context variable *auditCache*. The variable *notify* remains the source of audit records in other components throughout the system. Now, *notify* is cleared by an **impacts** statement in the context program. This behavior is captured in Figure 12, *ConsumedAuditSystem*, that instantiates *AuditableComponent* (Figure 3) and the central *AuditConsumer* to gather the audit records for the entire system.

---

```

System ConsumedAuditSystem
  include Program AuditableComponent
  include Program AuditConsumer
  Components
    <[] i :: AuditableComponent(i)>
    [] AuditConsumer
  Governance
    promote AuditConsumer.auditTrail as notify in *
end ConsumedAuditSystem

```

**Figure 12: System *ConsumedAuditSystem***

We specify the **SoS** *ConsumeCollectHybrid* (Figure 13) as the hybrid of both collecting and consuming audit system types. One collects audit records while leaving their original variables undisturbed, while the other consumes such records and continually clears the source variables. Both report the results as exposed variables named *auditTrail* and promote these variables to peer visibility at the SoS level.

### SoS *ConsumeCollectHybrid*

**include System** CollectedAuditSystem-2

**include System** ConsumedAuditSystem

#### System Instances

$\langle \square i : 1 \leq i \leq N :: \text{CollectedAuditSystem-2}(i) \rangle$

$\square \langle \square j : 1 \leq j \leq M :: \text{ConsumedAuditSystem}(j) \rangle$

**end** *ConsumeCollectHybrid*

**Figure 13: SoS *ConsumeCollectHybrid***

An interesting point to observe about SoS properties is that they must satisfy requirements as if they were a system while not violating properties in their component systems. Unfortunately, in Figure 13, we have a case where such property violations make the hybrid inviable. While the top level audit trail created in the *ConsumedAuditSystem* (Figure 12) satisfies P1, the individual audit trails of its instances in Figure 13 are no longer valid. Their collection algorithm is interfered with by the consumer algorithm within *AuditConsumer* programs in *ConsumedAuditSystem* instances. When the *auditTrail* variables are promoted as *notify*, they too become subject to the **impacts** statements in the context rules of *AuditConsumer* programs and are set to empty sets. This violates P1 for component systems of the SoS. Specifically, instances of *CollectedAuditSystem-2* fail since they no longer have a system-wide audit trail after execution of the **impacts** statement.

## 9. Conclusion

In this paper, we explore SoS as hybrid systems which may be prone to inviability and sterility. We introduce X-UNITY as initial machinery to specify and reason about hybrid systems in a hierarchical format. X-UNITY allows the expression of a SoS that may have isolating mechanisms, i.e., they are not completely and correctly integrated. This allows us to reason about the presence of isolating mechanisms, focusing on their expressions within hierarchical relationships, reuse by copy vs. by service, and variable scoping. Once their existence is determined, we can move to defeat if required. Additional research is required to exercise the complete proof system of safety and progress properties across a number of hybrid system examples.

**Acknowledgement.** This material is based on research sponsored in part by US Air Force Office of Scientific Research award FA9550-05-1-0374. The U.S. Government is authorized to reproduce and

distribute reprints for Governmental purposes notwithstanding any copyright notation therein.

## References

1. Gamble, M.T. and R.F. Gamble, *Isolating Mechanisms in COTS-Based Systems*, in *Sixth International IEEE Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems (ICCBSS'07)*. 2007, IEEE: Banff, Canada.
2. Coyne, J.A. and H.A. Orr, *Speciation*. 2004: Sinauer Associates.
3. Mayr, E., *What is a species, and what is not?* *Philosophy of Science*, 1996. **63**(2): p. 262-277.
4. Bock, C., *UML 2 Composition Model*. *Journal of Object Technology*, 2004. **3**(10): p. 47-73.
5. Allen, R. and D. Garlan, *A Formal Basis for Architectural Connection*. *ACM Transactions on Software Engineering and Methodology*, 1997. **6**(3): p. 213-249.
6. Milner, R., J. Parrow, and D. Walker., *A calculus of mobile processes, parts I and II*. *Information and Computation*, 1992. **100**(1): p. 1-77.
7. McCann, P.J. and G.C. Roman, *Compositional programming abstractions for mobile computing*. *IEEE Transactions on Software Engineering*, 1998. **24**(2): p. 97-110.
8. Roman, G.-C., C. Julien, and J. Payton, *Modeling adaptive behaviors in context UNITY*. *Theoretical Computer Science*, 2007. **376**(3): p. 185-204.
9. Dobzhansky, T., *Genetics and the Origin of Species*. 3rd ed. 1951, New York: Columbia University Press.
10. Shaw, M. and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*. 1996: Prentice Hall.
11. Morris, E., P. Place, and D. Smith, *System-of-Systems Governance: New Patterns of Thought*. 2006, Carnegie Mellon University.
12. Maier, M.W., *Architecting principles for systems-of-systems*. *Systems Engineering*, 1998. **1**(4): p. 267-284.
13. Foster, I., *What is the Grid? A Three Point Checklist*. *Grid Today*, 2002. **1**(6).
14. Boudol, G. *A generic membrane model*. in *Second Global Computing Workshop*. 2004.
15. Cardelli, L. and A. Gordon., *Mobile ambients*. *Theoretical Computer Science, Special Issue on Coordination*, 2000. **240**(1): p. 177-213.
16. Chandy, K.M. and J. Misra, *Parallel Program Design: A Foundation*. 1988: Addison-Wesley.
17. (NIST), U.S.N.I.o.S.a.T., *Recommended Security Controls for Federal Information Systems, rv. 1*. 2006, U.S. Dept. of Commerce, .