A Design and Verification Framework for Service Composition in the Cloud

Matthew L. Hale       Michael T. Gamble       Rose F. Gamble
Software Engineering and Architecture Team
Tandy School of Computer Science
University of Tulsa
Tulsa, OK USA
{matt-hale, mtg, gamble}@utulsa.edu

Abstract—The service cloud model allows hosted services to be dynamically provisioned and composed as part of larger, more complex cloud applications. Compatibility of interaction and quality of service are important to provisioning similar services available in the cloud to a client request. Auditing individual services, the composition and its outcome, and the overall cloud resources, used for monetary assessment or to ensure critical operations, also provide properties for reasoning over service and composition capabilities. Security policies and potential violations pose a threat to the composition since sensitive data may be leaked if information flow control guarantees cannot be proven. Service engineering lacks design principles and an expression infrastructure for formal representation and reasoning within a service cloud model. Reasoning over service compositions requires a formal language that can express multiple service and cloud properties. In this paper, we use coordination language techniques to express services, their interaction capabilities and information sharing constraints, and the infrastructure of a service cloud model in which services can be accurately provisioned, composed and reasoned over to provide necessary guarantees. We discuss lessons learned from the process of formulating the service representation and cloud model infrastructure.

Keywords: service composition; service cloud; coordination languages

I. INTRODUCTION

The Cloud utility computing paradigm is based on a dynamic service provisioning model [1-3] that composes application services together to satisfy elastic consumer demands. Service clouds [1] provide a set of re-usable resource templates, in the form of composed web services [4-6], virtual machine (VM) images [4, 7], and database schema [8], to application developers using APIs that promote scalability and usage metering. For developers, leveraging multi-tenant cloud resources to construct web applications allows for increased monetary opportunities through increased service delivery efficiency, improved resource use, and decreased costs.

All of the provisioned and composed services for a cloud application operate in a cloud application session. Cloud session management systems should reflect the constraints, properties, and policies of the services chosen for the application. For example, if each service has a security policy dictating how acquired and produced information can be shared, then the session management system must be able to manage information at each exchange point between services to protect against external threats. Two notable directions have been taken in service provisioning for securing cloud applications as it relates specifically to information disclosure. First, policy issues related to information flow control among services in a service-oriented architecture have been examined [3, 9]. This direction focuses on developing algorithms or protocols for exchanging policy data and credentials across a chain of composed services. The goal is to ensure that none of the policies are violated, otherwise the composition becomes invalid. A second direction involves ensuring the services themselves are provisioned securely [2, 7, 8, 10]. Many web service composition algorithms exist [3, 9, 10], but become increasingly complex when service interactions include assessing multiple policies or properties, sensitive data access, and dynamic storage.

Auditing and monitoring audit records for quality of service and security issues are often needed for service compositions and session management in the cloud. Research has examined auditing as it relates to session management [6] and has explored the role of Service Level Agreements (SLAs) [5, 11, 12] for monitoring cloud application properties to guarantee security constraints. Auditing approaches focus on detecting anomalies with respect to service operations across compositions, while combining the different auditing perspectives that exist in the cloud into a single perspective and ways to use this information, together with real time monitoring, to proactively improve cloud service offerings and facilitate incident reporting [6]. Difficulty arises in proper representation and monitoring across the service composition when each service may create a local audit trail to log specific events visible only from its perspective. At the same time, the cloud might log information related to provisioning and communication of the services. Yet, central monitoring needs to capture more shared information to provide a global perspective of application usage and outcomes.

SLAs are used by service providers to provide contractual agreements to customers guaranteeing certain quality of service (QoS) levels. Various research examines how service properties can be directly embedded in SLAs.
for use in service discovery [5, 11, 13, 14], matchmaking [5, 11, 13] and monitoring [14]. These efforts provide a business-level means for service customers to understand the security QoS constraints, referred to as quality of security service (QoSS), relating to how services are provisioned, audited, and protected against security risks.

Despite these research initiatives, service clouds lack formal techniques for ensuring cloud applications and communication sessions correctly provide and share information as expected. An infrastructure is needed first to accommodate the representation and reasoning over services, their properties, their composition, and the resulting cloud applications. Because of the lack of formal notation, there is subsequent limiting of design guidance for configuring and verifying cloud application designs given the diverse set of services hosted on a cloud that may be dynamically provisioned to form the application.

In this paper, we apply coordination language techniques to construct a formal cloud application specification language capable of expressing and reasoning over service compositions in order to ensure correctness and provide design guidance. Using this language we design a service cloud infrastructure that can be used formally to demonstrate various service information exchange constraints and ensure services are designed and interact correctly. We apply our infrastructure to a simple case study to demonstrate the language constructs, including a session manager specification within the cloud. Given the design constraints set forth in the service specification, we prove session containment among provisioned services, a core property needed for further reasoning over sessions. The goal of the effort is to enable cloud application design verification and to provide application developers guidance for designing cloud-based applications utilizing the service cloud, based on sound formal principles.

II. RELEVANT RESEARCH

Composing web services in Service Oriented Architectures (SOAs) has been the study of multiple research efforts focusing on service discovery [11, 15-17] and selection [5, 14, 16]. Service discovery involves the use of publishing mechanisms such as an UDDI [15] or the WSO2 Governance Registry [17] that allow services to advertise their capabilities. Services are marked up using standards such as WSDL [13] which allows semantics to be attached to service specifications [13, 14]. Selection research focuses on exploiting the attached semantics to analyze advertised services and rank them according to how well their capabilities, Quality of Service (QoS) guarantees, and pricing meet client needs and/or align with other services. For instance, Hu et al. propose WSRank [16] which adapts page rank principles to collect, index, and rank the WSDLs for cloud-services based on how they are connected to other services or referenced online.

The potential for sensitive information disclosure exists even in the sessions where service security policies allow for information sharing with other provisioned services in the application. This threat can be manifested if a service is spoofed, if a message is intercepted, or if the existence of some data used in the application is known, even if not directly accessible. Thus, when security is of concern the session should tightly encapsulate service communication, such as through a secure session, and protect data shared among the services, such as through secure channels. These policies are threatened by a number of security risks including local (pairwise between services) and global (end-to-end of service composition) violations of data integrity, data confidentiality, and service availability which are introduced when services exchange information as part of the execution of a cloud application. These risks stem from trust [6, 9] and communication [3, 6, 9] issues that may exist between services in compositions.

Overcoming the variety of increased security challenges has been the focus of numerous works [2, 3, 5, 9-12]. Singaravelu et al. [9] examine the access control, encryption, and trust implications associated with end-to-end messaging in web service compositions. They introduce WS-FESec as a WS-Security extension that improves the end-to-end handling of confidentiality and integrity constraints in messages [9]. She et al. [3] propose a message exchange protocol for information flow control among composed services to reduce violations. Their carry-along policy and pass-on certificate advertise service flow control policies and prevent information leakage by providing appropriate credentials, but it incurs a large overhead due to the increased interaction among the services required to process the policy directives.

Beyond web service security, Bleikertz et al. [7] identifies three core security issues associated with virtualized service cloud architectures. These including virtual isolation (separate execution environment for each client), operation correctness (services operate as expected), and failure resilience (no single point of failure). A graph theoretic approach is used to express each of these issues as a graph rule, e.g. for isolation two nodes outside of the same security domain may not communicate. These graph rules are mapped onto the formal language VALID [7] which uses model checking to compare the run-time state of virtual machines (VMs) in cloud against the defined security goals.

Coordination languages have been successfully used to formally verify operational correctness [18, 19] and security properties over static SOA specifications [19-21]. The primary coordination language construct is a tuple space. Tuple spaces are a well-studied data structure [20-23] that provides a data-driven model of component interactions. Its usage originated with the parallel programming language Linda [23], which provided the three operators in, out, and rd to allow tuples to be respectively entered, removed, or read from the tuple space. Merrick et al. [22] established a set of scoping rules that allowed Linda tuple spaces to be nested or isolated from one another. KLAIM [19] first introduced permissions to tuple spaces providing access control mechanisms capable of limiting which processes can manipulate or read the content of tuples in a space. Their
approach relies on a set of static access control policies to limit certain processes to their respective spaces and is coarse grained meaning single tuples and data fields cannot be restricted within a tuple space.

Recently, Linda-like tuple spaces have been applied by Bravetti et al. [24] to model and secure coordination in untrusted interaction environments that occur between composed services. Their work developed SecSpaces [24] which refines Linda tuples with asymmetric key-based control fields for attaching fine-grained access control parameters. A Linda tuple may only be read, i.e. rd, if the provided credential matches the stored private key. However, despite these advances from previous Linda iterations, SecSpaces lacks the formal proof logic necessary to prove temporal interaction properties.

X-UNITY [21, 25], pronounce “cross-unity”, offers all of the fine-grained access control and encryption capabilities of SecSpaces, but also brings to bear a powerful temporal proof logic associated with its predecessors [18, 26]. Of the coordination languages available, it provides the most fertile ground for representing services in the cloud and the best proof theory for proving temporal properties over service specifications. Thus, we base our work on X-UNITY, describing it in the next section, as well as within our cloud design and verification landscape.

III. X-UNITY

X-UNITY [25] inherits features from Context UNITY [18] and extends it to build a representation platform for systems of system specifications, including managing incompatible interactions. X-UNITY provides support for representing local and global security policy awareness related to access control, encryption, and auditing of exposed (shared) variables in the tuple space [21]. Fig. 1 shows a Program specification as a core X-UNITY entity.

| Program ProgramName (parameters) | declare | internal – internal variable declarations | exposed – exposed variable declarations | context – context variable declarations | initially – initial conditions of variables | assign – assignment statements to declared service variables | context – context rules affecting context variables | policy – policy declarations that associate security controls with exposed variables using access, audit, encrypt keywords | end ProgramName |

A Program consists of a set of declared variables, initial conditions, and assignment statements that operate over the declared variables. Internal variables are private to the program. Exposed variables are represented as shared tuples accessible by other programs operating within the same System (as shown in Fig. 2). Each exposed variable has an underlying tuple structure represented by a var table as shown in Table 1. Tuples are customarily referenced by their name, i.e. η, which for representational simplicity can be assigned their value ω.

Formally, the var table allows fields within the tuple to be referenced using the notation var[η].field where field is some entry in Table 1. For instance, the notation var[η].ω is equivalent to the value of the exposed variable x.

<table>
<thead>
<tr>
<th>TABLE 1: X-UNITY var TABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
</tr>
<tr>
<td>π</td>
</tr>
<tr>
<td>η</td>
</tr>
<tr>
<td>τ</td>
</tr>
<tr>
<td>ω</td>
</tr>
<tr>
<td>α</td>
</tr>
<tr>
<td>κ</td>
</tr>
<tr>
<td>ϵ</td>
</tr>
<tr>
<td>δ</td>
</tr>
</tbody>
</table>

Returning to Fig. 1, context variables allow a program to access and manipulate the exposed variable tuples of other programs in the same coordination space or System. The corresponding context section of a program asserts the rules which manage these interactions via exposed Program variables that reside in the tuple space. A built-in credentials variable is used to determine which tuples a context rule can access and modify within the tuple space. Credentials are embedded in each program and checked against exposed variable’s access control policies, α in Table 1, when used in a context rule.

Assignment statements that are not part of program interactions are handled in the assign section. Assignment can be one of three types: normal assignment statements on internal or exposed variables represent a state transition of the form x := x’, where the after state of x is x’. Transactions denote an ordered series of normal assignments that execute atomically, and reactive statements execute to fixed point [18] in response to a specific defined state change denoted by the key word reacts-to <state expression>.

To express security policy awareness, X-UNITY includes the policy section in Fig. 1 [21]. The policy section may contain the keywords access, encrypt, and audit to dictate the policy implementation conditions, such as

policy <exposed variable> access|audit|encrypt <params>,

which ensures that certain exposed variables meet the policy parameters defined when the program is deployed. The access keyword identifies components within the system that are allowed to read and/or write to the associated exposed variable. Similarly, audit specifies the types of auditable events that should produce audit records in the component’s audit log and encrypt denotes the need to encrypt the variable’s value with a specific encryption algorithm [21].
Fig. 2 shows the systems of systems (SoS) infrastructure specification within X-UNITY. A set of Systems can be expressed as part of the SoS. Multiple programs are referenced and governed within the X-UNITY System representation, also shown in Fig. 2. The set of Components denotes the instantiated programs in the System with explicit parameters. The Governance section of a system provides system-wide statements that apply to all exposed variables in the associated tuple space. Promote is a Governance operator that makes explicit which exposed variables in a Program can be accessible to other systems in the SoS. Though not shown, Promote can dictate which systems can see the exposed variable and any aliases each variable has for particular systems. For simplicity, we do not use its full specification. Thus, all promoted variables can be seen by all Services (as systems, see Fig. 8) in the cloud.

<table>
<thead>
<tr>
<th>SoS SoSName</th>
</tr>
</thead>
<tbody>
<tr>
<td>System SystemNamei</td>
</tr>
<tr>
<td>Programi &lt;reference and parameters&gt;</td>
</tr>
<tr>
<td>Components</td>
</tr>
<tr>
<td>Governance</td>
</tr>
<tr>
<td>Global impact statements such as Promote</td>
</tr>
<tr>
<td>end SystemNamei</td>
</tr>
<tr>
<td>Service SystemNamek ... end SystemNamek</td>
</tr>
<tr>
<td>end SoSName</td>
</tr>
</tbody>
</table>

Figure 2: X-UNITY SoS and System Specification

X-UNITY retains the non-deterministic weakly fair execution model [18] and proof theory delivered by Context UNITY. Thus, X-UNITY’s execution model examines all assignment statements and context rules within a program and chooses one non-deterministically with weak fairness, in which every statement is chosen infinitely often. At the SoS level, each System is chosen in the same manner, as well as each of its programs, for statement execution. UNITY provides an associated temporal logic and proof theory given the execution model that allows proofs based on execution traces.

IV. CASE STUDY

A service cloud use case composes a set of services as part of a business process workflow to satisfy a client request. The composition may be from a historical cache of cloud applications or may be dynamically provisioned to address the request when it is made. We compose two services to address a client request through end-to-end, round-trip messaging. The case study includes a travel service that as part of its operations invokes a weather service for the forecast on the dates for client travel as depicted in Fig. 3. When provisioned to the composition, each service invokes a proxy. This example is used throughout the paper to motivate and exemplify the service cloud model and coordination language approach to specification of and reasoning over a service cloud.

Figs. 4, 5, and 6 express each service proxy using X-UNITY in which communication between a client proxy and a travel service proxy is through shared tuples tRequest and tResponse and communication between a travel service proxy and a weather service proxy is through shared tuples wRequest and wResponse. Fig. 4 shows the client Proxy, which is a combined Program and Component instantiation of the program within a Service, which takes the place of a X-UNITY System. We use ∅ to mean any null value of the appropriate type for notational convenience. Another notation convenience is that of the multi-assignment statement. In the initially section of Fig. 4, four assignment statements are performed concurrently. In this statement, the proxy ID is assigned that which is assigned in its parameters, the userInput is assigned its request that invokes the session, and the plan and travelPartner are both initialized to ∅.

Context rules are used to model the communication that occurs between the client and the two services. A single context rule writes (impacts) or reads (becomes) information to or from exposed variables, owned by other instantiated proxies in the same coordination space, using the defined local context variables. The uses keyword in context rules denotes the exposed variables being used in the communication space. All of the contextual assignment statements are stated in the where section and are executed provided that the conditions in the given section are met.

After receiving some userInput via the Get userInput() function, the only eligible context rule to execute is the first one in which the Session Manager has initiated a session by providing a communication channel for the client. The use of the Session Manager is derived from segregating cloud architecture components for better manageability [27]. Its purpose in this case study is to establish the session and the communication channels needed for improved and deliberate information sharing.

We use s! as a local dummy variable for internal reference of the shared object. Once the channel assigns the travelPartner for communication, the client makes an initial request to the targeted travel service by pushing contents to the tRequest variable using the second context rule shown in Fig. 2. The tRequest variable is defined as the tuple (fromdate, todate, from, to) where the fromdate is the requested departure date, todate is the requested return date, from is the departure location, and to is the destination location.

This state transition sets up the travel service to do its processing. The Client-Proxy is then in a wait state until the travel service responds. At which point, the third context rule becomes eligible for execution, obtaining the travel plan, that includes local weather, and then resetting the channel and shared variables to end the session.
Figure 4: X-Unity Client Proxy Specification

The Travel-Proxy (Fig. 5) will grab its channel information when it becomes available using the first context rule. Given this information, the travel service performs some calculations (shown in the assign section in Fig. 5) and reads a \textit{tResponse}, which is a \texttt{json} document representing the travel plan. However, part of the response \texttt{json} includes a weather report. Before the travel service can respond, it must request the forecast from the weather service. Thus, it makes a weather request using the second context rule in Fig. 5 on the \textit{wRequest} variable, which is defined as the tuple \texttt{(date, location)} in the \textit{weather} service where \texttt{date} is the start date of the 10-day forecast and \texttt{location} is the destination location of travel.

Weather has also been provisioned for the session using the \textit{weatherChannels} construct established by the Session Manager. Only the context rule cannot be executed until \textit{weatherPartner} is set. Since the weather service is the last service on the chain, it is unique in that the channel is reflexive so that it is its own communication partner. This expression better separates the request sharing between travel and weather and the response sharing between weather and travel.

The requested weather report is generated by the \textit{Weather-Proxy} using the \textit{CreateForecast} function shown in the assign section of Fig. 6. This report is then passed back to the travel service as a \textit{wResponse}, which is another \texttt{json} document containing the weather forecast. The travel service pulls the \textit{wResponse} into its \textit{forecast} context variable using third context rule in Fig. 5. The \textit{forecast} \texttt{json} is then added to the travel plan by the \textit{CreateTravelPlan} function (shown in the assign section of Fig. 5) and made available to the client using \textit{tResponse}. Also in this third context rule, the channel is closed and the associated variables are reset. Finally, the client proxy stores the \textit{tResponse} \texttt{json} in the \textit{plan} context variable using the third context rule in Fig. 4 as discussed earlier.

Figure 5: Travel Service Proxy Specified in X-Unity

Figure 6: Weather Service Proxy Specified in X-Unity
V. DESIGN AND VERIFICATION PLATFORM

Service clouds lack formal techniques for ensuring cloud applications correctly provide and share information for interaction, auditing, and security. To accomplish this, an infrastructure is needed to accommodate the representation and reasoning over services and their composition as cloud applications. This section designs a service cloud infrastructure that houses service proxies, orchestrates them into compositions, and can be used formally to demonstrate information exchange and sharing constraints to ensure service proxies are designed and interact correctly. We develop the infrastructure using the case study context to demonstrate the language constructs.

One issue that arises in the service cloud is how proxies are provisioned into compositions. Once provisioned, the service proxies begin interoperating. If one or more of the proxies are designed incorrectly the logical flow of the composed cloud application might be incorrect. In this case the cloud application may never successfully complete or may return incorrect information. For critical systems, such as in the medical or military fields, incorrect application functionality could cost money or, worse, lives. Another issue in the service cloud is inter-proxy communication. There is the potential for information leakage or disclosure if communication channels are not successfully implemented or protected.

A. Managing Sessions

To reason about service cloud issues our infrastructure includes a Session Manager [6] cloud service. The Session Manager orchestrates service compositions by enabling proxies of advertised services to facilitate communication via channels. Represented as SessionMgr in Fig. 8, it captures incoming cloud requests and ensures orchestration, provisioning, and proxy communication are correctly established. X-UNITY does not have the means for dynamically instantiating proxies. Therefore, we abstract this to an expression of a discrete (arbitrarily high), maximum number of potential proxies for each available service that are inactive until provisioned. SessionMgr retains the list of currently inactive client, travel, and weather proxies.

Client requests are captured by the assign statement in Fig. 7 by the function GetTravelRequest(), which returns true when there is an unhandled request for provisioning travel related services. A non-deterministic choice of inactive clients, travel, and weather proxies results in adding a new triple (c, t, w) to activeSessions, removing c, t, and w from their respective inactive sets, and instantiating communication channels between c and t, t and w, and w and c, as the last service in the chain. The channels are disabled and the proxies returned to inactivity after the session is complete by the proxies.

The channels are stored as dictionary mappings (Dict) from PID to PID using the arrow notation “→”. The objective is to confine the proxies to communicating only along the channel during the session. By defining them as dictionaries they can be referred to using the notation

*Channels[key] = <value>*

where key is the id of the proxy and value is the proxy assigned to it by the SessionMgr. The mappings assign every instantiated client, travel, and weather proxy to a channel. Null channels assignments mean the cloud is not providing a channel for the denoted proxy, e.g. clientChannels[1] = ∅ means that the client proxy with PID 1 does not have a communication partner. On the other hand, the notation clientChannels[1] = 7 means the client proxy with PID 1 can interact with the promoted variables (e.g. the iRequest and iResponse variables) in the proxy with PID 7.

Using the encrypt keyword means the channel exposed variables are encrypted. This means that only a proxy with the correct valid certificate, embedded in its built-in credentials variable, can access the respective channel.

```
Program SessionMgr(sid : SID, m : Integer, n : Integer, k : Integer)

declare
  internal
  id : SID
  active : Set of (client : PID, travel : PID, weather : PID)
  inactiveClients, inactiveTravels, inactiveWeathers : Set of PID

exposed
  clientChannels : Dict {PID → encrypt RSA}
  travelChannels : Dict {PID → encrypt RSA}
  weatherChannels : Dict {PID → encrypt RSA}

initially
  id, inactiveClients := sid, [1..m]
  inactiveTravels, inactiveWeathers := {[1..n],[1..k]}
  clientChannels := [{1→∅}…{m→∅}]
  travelChannels := [{1→∅}…{n→∅}]
  weatherChannels := [{1→∅}…{k→∅}]

assign
  if GetTravelRequest()
    then (c, t, w : 
      c ∈ inactiveClients, t ∈ inactiveTravels, w ∈ inactiveWeathers : 
      activeSessions := activeSessions ∪ {c,t,w});
      inactiveClients := inactiveClients \ {c};
      inactiveTravels := inactiveTravels \ {t};
      inactiveWeathers := inactiveWeathers \ {w}
      clientChannels[c] := t;
      travelChannels[t] := w;
      weatherChannels[w] := c
    end

```

Figure 7: SessionMgr Program in X-UNITY

B. Specifying the Infrastructure

Together all of the service proxies in the case study composition are represented in a Cloud SoS specification, Travel_Service_Cloud shown in Fig. 8. This cloud contains a number (m, n, and k respectively) of potential proxies in each Client, Travel, and Weather Service as well as the SessionMgr program in the Platform Service. Each Service also uses promote governance statements to denote which exposed proxy variables in each service system communication space are being promoted up to the
SoS space to allow other system proxies to interact with them. The Client Service has $m$ deployed proxies which uses the $tRequest$ and $tResponse$ variables promoted from the Travel system. Travel promotes $tRequest$ and $tResponse$ for each of its $n$ proxies, and uses the $k$ $wRequest$ and $wResponse$ variables promoted by Weather system. The SessionMgr promotes $clientChannels$, $travelChannels$, and $weatherChannels$ to establish restricted communication sessions between the proxies as discussed in the previous section.

Cloud Travel_Service_Cloud
  Service Client
    Proxy Client-proxy[1..m]
  Service Travel
    Proxy Travel-proxy[1..n]
  Governance
    promote $tRequest[1..n]$, $tResponse[1..n]$
  Service Weather
    Proxy Weather-proxy[1..k]
  Governance
    promote $wRequest[1..k]$, $wResponse[1..k]$
  Service Platform
  Program SessionMgr
    Governance
    promote $clientChannels[1..m]$, $travelChannels[1..n]$, $weatherChannels[1..k]$
end Travel_Service_Cloud

Figure 8: Case Study Deployed in a Cloud System of Systems

VI. VERIFYING SESSION CONTAINMENT

To reason over the program logic and ensure proper implementation, designers can refer to the formal specification of each proxy and the overall Cloud SoS. Using these specifications, the execution model can be traced to understand the series of possible state transitions allowed by the cloud application. This allows for formal properties to be established over the cloud compositions.

One property is session containment which is defined as “once a proxy is provisioned and assigned to a session, it remains sharing information with only those session variables until the session has ended and becomes inactive.” This property is a staple for proving stronger properties related to access control and information flow control related to session management.

We define a provisioned proxy $i$ as having a context variable $x: PID$ that is the range value of a channel for shared information such that $(i, x) \in \text{*Channel}$ where $*$ denotes a regular expression for the channel name and the proxy name. Then

$$\text{provisioned}(*.\text{Proxy}[i]) \equiv \exists x: \text{PID} | (i, x) \in \text{*Channel}$$

This state is established for a proxy by the $sessionMgr$ assignment statement. Note that $sessionMgr$ does not have the ability to alter the channel once it is activated. Nor can it prematurely cancel the session. Only the proxies can complete or abort the session and reset the shared and channel variables.

Our specification solution requires that for the service design to accommodate session handling it must support having a communication partner context variable that captures the channel information. We use the UNITY temporal operator, $\text{leads-to}$, to clarify this.

$$\text{provisioned}(*.\text{Proxy}[i]) \text{ leads-to } \text{*Partner} = \text{*Channel[i]}$$

In UNITY, a $\text{leads-to} b$ means that when state $a$ holds, there exists a set of statements that can transition the program to state $b$ without requiring $a$ to continue to hold. To understand how this temporal transition occurs, $sessionMgr$ will establish three channels for client, travel, and weather. Each proxy has a partner variable and a context rule that assigns the channel to the partner if the channel is available. Recall that the context rules are nondeterministically chosen with weak fairness for execution. Thus, the order of channel assignment to partner across the proxies does not matter.

Once the communication partner has knowledge of the channel, the shared variable updates can occur. By examining each assignment statement and context rule in each proxy, we can guarantee that the channels are not reset until the shared variables have been. Additional leads-to operators are required for formal proof.

Another consideration is if other proxies have access to channels external to their session. Because they are promoted within the $Travel_Service_Cloud$ specification, their existence can be known, but their contents cannot be known because the Session Manager restricts access to particular channels unless they share a session designated by $[c, t, w]$.

VII. DISCUSSION AND CONCLUSION

Based on the representation and formal reasoning mechanisms, we discuss the design expectations of using a coordination language for the service cloud infrastructure and conclude the paper.

A. Direct Information Sharing through Channels

The specification of the channels allows the Session Manager to act as a cloud service intermediary. This indirect communication yields the same cloud application results from the composition, but provides a centralized service that can examine SLA compliance and QoS attributes prior to provisioning. In addition, the Session Manager can capture audit records generated by the services and port them to a monitoring component to determine if there are security threats, such as information flow control and message communication attacks. Xie and Gamble [6] show that the use of a Session Manager as an intermediary cloud service adds significant overhead, but the evidence captured for security monitoring far outweighs the overhead cost.

B. Explicit Sharing and Exchange

By modeling the service cloud using a coordination language, the specific forms of reasoning over shared tuples in a tuple space can identify consequences of
service interactions, improper service provisioning, and poor session management. Temporal predicates can be formulated that are specific to the property to be assessed, such as access control constraints, QoS provisioning restrictions, and security policy compliance. However, to form these predicates, application developers must have knowledge of what shared entities can be exchanged and how. If access control and encryption are important, than the shared variable should be able to enforce its own policies, such as X-UNITY provides by the tuple structure (i.e. \texttt{var} table) of exposed variables. If security monitoring is important, then the history of the variable accesses must be available to the cloud. Thus, application developers should create a service that can obtain the information stored in that field of the \texttt{var} table of the exposed variable.

The X-UNITY execution model and proof theory as borrowed from Context UNITY [18] and UNITY [26] allows for reasoning over the session variables from the perspective of coordinated tuples. By using X-UNITY to express a service cloud model, we enable this reasoning over the services provisioned for a particular session. The use of the Session Manager as an intermediary provides a segregated tuple space from which the session can be understood and proven compliant with specific properties needed for the cloud application.

ACKNOWLEDGMENT

This material is based on research sponsored in part by the Air Force Office of Scientific Research (AFOSR), under agreement number FA-9550-09-1-0409.

REFERENCES