Control Plane Integration for Cloud Services

Sujoy Basu, Sven Graupner, Jim Pruyne, Sharad Singhal

HP Laboratories
HPL-2010-122

Keyword(s):

Service integration, Service-Oriented Architecture, Model-Driven Architecture, source code annotation, introspection.

Abstract:

This paper addresses the problem of control plane integration for management and control of cloud services. Unlike data plane integration, which ensures that services can exchange data during operation, control plane integration ensures proper configuration of services before their use. Examples of control plane integration include creating user accounts or establishing profiles in multiple services to allow them to work together during operation. The heterogeneity of service interfaces in the control plane arises from the different ways in which services are implemented and the different requirements they have for their use. Control plane integration is often needed for service bundling and ad-hoc compositions across services, such as for promotional campaigns that must be developed and deployed rapidly. In this paper, we propose a developer-centric approach to integration of services on the control plane. Our approach is based on using Java code annotation, which is introspected at runtime to create rich service models. A multi-layered architecture allows the rapid modeling, development and implementation of service integration scenarios. We demonstrate our approach with an example of a promotional campaign that uses two external service providers.

External Posting Date: September 21, 2010 [Fulltext] Approved for External Publication
Internal Posting Date: September 21, 2010 [Fulltext]
To be published and presented at ACM/IFIP/USENIX 11th International Middleware Conference, Bangalore, India November 29 - December 3, 2010.

Control Plane Integration for Cloud Services

Sujoy Basu, Sven Graupner, Jim Pruyne, Sharad Singhal

Hewlett-Packard Laboratories
1501 Page Mill Rd, Palo Alto, CA 94304

{sujoy.basu, sven.graupner, jim.pruyne, sharad.singhal}@hp.com

ABSTRACT

This paper addresses the problem of control plane integration for management and control of cloud services. Unlike data plane integration, which ensures that services can exchange data during operation, control plane integration ensures proper configuration of services before their use. Examples of control plane integration include creating user accounts or establishing profiles in multiple services to allow them to work together during operation. The heterogeneity of service interfaces in the control plane arises from the different ways in which services are implemented and the different requirements they have for their use. Control plane integration is often needed for service bundling and ad-hoc compositions across services, such as for promotional campaigns that must be developed and deployed rapidly. In this paper, we propose a developer-centric approach to integration of services on the control plane. Our approach is based on using Java code annotation, which is introspected at runtime to create rich service models. A multi-layered architecture allows the rapid modeling, development and implementation of service integration scenarios. We demonstrate our approach with an example of a promotional campaign that uses two external service providers.

Categories and Subject Descriptors

D.2.2 [**Design Tools and Techniques**]: Modules and interfaces, Object-oriented design methods. D.2.11 [Software Architectures] Service-oriented architectures (SOA).

General Terms

Design, Standardization, Languages

Keywords

Service integration, Service-Oriented Architecture, Model-Driven Architecture, source code annotation, introspection.

1. INTRODUCTION

We focus on the domain of telecom service providers, whose revenues are largely derived from voice or data transmission. Expanding into new revenue sources by offering new, higher-level services and addressing new markets is a common objective for these providers. Typically, telecom service providers partner with external service providers to deliver value-added services using bundling, while retaining operations and business support functions such as help-desks and billing.

To offer a larger portfolio of services, telecom service providers are increasingly integrating external services provided by software vendors in a Software-as-a-Service (SaaS) model. Even though this integration problem has existed for a long time, the proliferation of ecosystems of cloud services has aggravated the problem. The number of services that are being integrated has increased tremendously, and these integrations must be supported with greater agility. Furthermore, such integration leads to complexity that is specific to telecom networks. Customers signing up for external services through their mobile devices must be provisioned at the selected services. This requires executing *provisioning workflows* that connect to the service providers, create accounts for the users, and initialize their profiles and service levels. However, customers expect telecom companies to be the single point of support for services offered through them. Therefore, integration of these services must also be done at the monitoring and management levels at the time they are offered, allowing the service to be monitored from the telecom network. It also allows the customer's usage of the service to be metered and billed by the telecom company. Another benefit of integrations is identity management. The customer can have a single sign-on that works across all subscribed services.

2. CONTROL PLANE INTEGRATION

Service integration technologies typically focus on the messages exchanged between services. A mediator, proxy, adapter or broker intercepts messages sent from one service to another and performs the necessary transformation and coordination tasks. We refer to this layer of message exchanges as the *data plane*.

While service integration in the data plane has been widely explored and a variety of technologies have been developed, there is another domain of service integration that has not been addressed prominently. We refer to this domain as *control plane*. The control plane addresses the configuration and preparation that is needed before services can be used and messages can be exchanged in the data plane. It includes operations such as creating user accounts, providing accounts with information profiles about users, or with information such as subscriptions and payments that are required by the underlying services prior to use.

Similar to the data plane, control plane interfaces and protocols are heterogeneous, and highly specific to services. Techniques similar to those used in the data plane for connecting interfaces, transforming data and coordinating control flows can still be used for control plane integration. For example, both Web Services Description Language (WSDL) and Representational State Transfer (REST) APIs may be present in a single bundling

Middleware'10, 29 November – 3 December, 2010, Bangalore, India.

scenario. However, there are additional problems in the control plane that need to be addressed such as establishing and mapping of user identities, creation and secure transfer of credentials, establishing and mapping of user profile information, and the reliable transfer of funds from sources (in form of payments, chargeable assets, subscriptions) that are required for paid services. This makes the problem of control plane integration different from data plane integration.

Frequently, these control plane tasks are performed manually during service subscription. However, in the service bundling scenarios we are considering, this information must be configured in services for thousands of users. This requires user-specific information available in the telecom business support systems (BSS) to be programmatically configured in the bundled services. Each of the bundled services can have tens of configurable parameters. Developers need to provide integration with the services for all the tasks needed by the workflows. As service logic within individual services evolves, keeping the integration code in sync can become tedious.

The control and data planes differ in other ways. Performance (e.g. message latency and throughput) is important in the data plane *while* the service is being used. It is of lesser importance in the control plane since most interactions occur only once *before* the service is used. Reliability and security have a high priority since control plane integration includes the ability to create new accounts and process payments. Fast development cycles and low development costs are of higher importance as well, since many services created by such integration are short lived. For example, promotional campaigns that only last a few weeks are common.

3. EXAMPLE

We motivate our discussion using an example where a telecom service provider that offers voice and data plans runs a campaign to increase subscriptions of its data plan by voice customers. To advertise its data offerings, the telecom provider bundles its data service offerings with other services in a promotional campaign. In this example, two external service providers are used.

Snapfish.com is an online photo publishing and printing site. *tele-coupons.com* is a hypothetical service that distributes coupons and manages financial settlement among vendors for issuing and redeeming coupons. In the example scenario of the promotional campaign, the telecom service provider seeks to upgrade its voice-only service users to data plans. It arranges with the two business partners a special offer that includes a coupon of $50 redeemable for new smartphones being offered by the telecom service provider, and 50 free photo prints at *snapfish.com*.

In this scenario, the telecom service provider is hosting the promotional campaign and has business agreements with the participating business partners. The technical implications are that logic needs to be defined and implemented in the telecom service provider's backend systems to (a) advertise the promotional campaign to its current voice-only users, (b) connect to the system where users can chose and upgrade their service plans, and (c) when an eligible user chooses to upgrade to a data plan, initiate the interactions on behalf of that user with the two participating service providers.

While (a) and (b) are traditional internal programming tasks within the backend of the telecom service provider, (c) expands the control plane integration into the domains of external partner services. It also requires the definition and implementation of integration logic and is hence a substantial development and integration effort.

We consider the following steps for the scenario:

1. A voice-plan user chooses to upgrade to a data plan. The backend system detects that condition and triggers the integration scenario with the external services. This launches the processes shown in Figure 1.
2. As part of these interactions, the telecom service provider opens an account for the user at *tele-coupons.com* (if such an account is not already present) and creates a coupon of $50 for the purchase of a new smartphone. The login credentials are passed back to the user who then can obtain the coupon from the *tele-coupons.com* site and redeem it with an eligible purchase at a participating retailer. The financial settlement of the coupon payments between the issuer (telecom service provider) and the retailer where the coupon is redeemed is managed by the *tele-coupons.com* service.
3. Similarly, an account is created for the user in *Snapfish.com* and preloaded with the equivalent of 50 photo prints.

The user's root identity is determined by the telecom service provider. Information on existing accounts at the participating services is retrieved from the business support system (BSS). In this example, we ignore corner cases where users have existing accounts unknown to the telecom service provider.

Information about new accounts created on behalf of users is communicated back to users through the telecom service provider's portal. The user can then click the provided URLs and log into the services using the credentials provided by the telecom service provider.

Figure 1 shows the interactions that take place between the telecom service provider and the two external service providers. The telecom service provider is shown on the left in the Figure and the two external services *tele-coupons.com* (S1) and *snapfish.com* (S2) are shown on the right.

The flow inside the telecom service provider's environment shows the use of two highlighted integration building blocks for implementing the interaction logic for adding users and depositing funds into user accounts (one pair used for each external service). Those building blocks are offered as libraries to the service integration developer to help assemble more complex integration scenarios from simpler integration building blocks. The two building blocks encapsulate the actual service invocation through service adapters. In addition, they encapsulate the handling of a simple error case. In case of error, a state is reached in which an error handling method can be invoked which in turn can result in a retry attempt of the operation or in another failure which, according to the logic shown in Figure 1, leads to the abortion of the overall operation for a particular user.

Next, we describe an architecture that enables such integration to be performed rapidly.

4. SERVICE INTEGRATION PLATFORM

We address the specific needs of control plane integration with a dedicated service control plane integration platform. It is also referred to as service integration platform in the rest of this paper.

The diagram in Figure 2 shows the architecture of the service

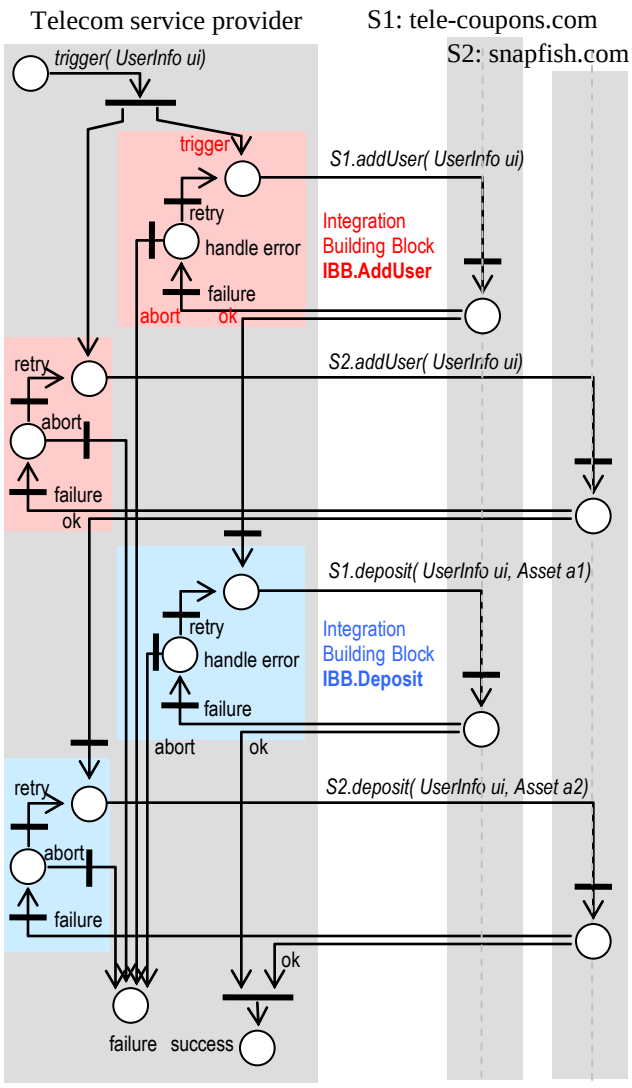

Figure 1: Service integration logic for example scenario.

control plane integration platform. The figure shows four service instances on the left representing external services S_i ($S_1 - S_4$ in the Figure) as they may exist in the cloud (e.g. *tele-coupons.com* or *snapfish.com* as used in the example in section 3). Each service S_i provides control plane interfaces CPI_i through which control plane operations can be performed (shown as interface symbols with horizontal fill pattern, connected by dashed interaction lines with service adapters).

Each service is accompanied by a service model m_i for service S_i. Service models (defined in more detail in Section 5) represent control surfaces available for the service and are created and maintained by specialized service model providers that act as registries. Service model providers smp_j provide service model interfaces $SMI_{j,i}$ for each service model m_i through which control operations can be performed (shown as interface symbols with vertical fill pattern and green interaction lines). Service model providers are web services through which service models are accessible. Each service model is accessible through a distinct URL. For example, while the service snapfish.com may not provide an explicit service model, a hypothetical model provider Servicemodels.com may offer a service model for it using the URL //servicemodels.com/snapfish.

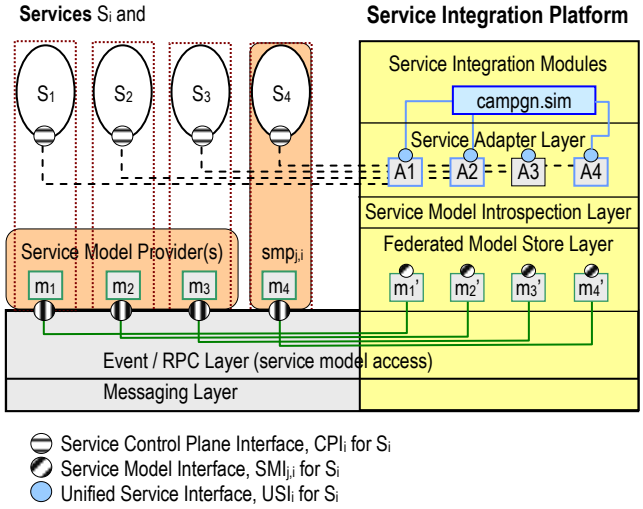

Figure 2: Service Control Plane Integration Environment.

The reason for separating services from service model providers is that most existing cloud services currently do not provide service models for control-plane operations of their services. One service model provider can host service models for multiple services and create and maintain them as a commercial operation. The Figure shows models $m_1 - m_3$ provided by a service model provider while m_4 is provided by the service S_4 itself. In this case, the service must offer the service model interface SMI_4. Service model providers share an event and RPC messaging layer with the service integration platform through which only service model information is accessed and exchanged.

The layers of the service integration platform are shown on the right hand side of Figure 2. The architecture builds on a shared event and RPC messaging layer with the service model providers. An abstraction of a **Federated Model Store Layer** is established as the next layer above. Since service model providers themselves are services in the cloud, access to models in the service integration platform occurs by fetching copies of service model m_i and storing locally inside the federated model store layer as m'_i. An event-based consistency mechanism is implemented in the federated model store layer that allows service model instances m'_i to initiate update operations on service models m_i in the service model providers. Service model providers maintain event distribution lists through which other service model instances m''_i in other service integration stacks can subscribe to model update events. Service model update events caught in the federated model store layer can also be passed to higher layers in the service integration platform and trigger effects in response to model update events that were received from other service model instances.

The next layer is the **Service Model Introspection Layer**, which allows queries and updates in local service models m'_i. In case of updates, they are passed back to the service model provider from which they were initially obtained. The layer is integrated with the programming environment of the service adapter layer. In a java

implementation, classes and java objects are automatically generated for type and instance information respectively, from service models. Many programming environments support classes, objects and introspection today. We favored this style over traditional access through query interfaces such as SQL or JDBC due to the better embedding in programming languages.

Models are used by the **Service Adapter Layer.** Adapters A_i connect to the actual control plane interfaces offered by services. Some cloud services provide programmable interface for that purpose, but typically each is unique and offers a different programming paradigm. Adapters encapsulate this heterogeneity and unify service control plane interactions as operations that are performed on service models through the programming environment. Changes to service models in turn trigger adapter invocations that are passed to the service control plane interfaces CPI_i. Service model operations are exposed through the unified service interfaces USI_i to the next-higher layer of service integration modules (USI_i are shown in the figure with blue interface symbols for adapters A_i).

The layer of **Service Integration Modules** contains modules that connect service adapters with logic for certain service integration tasks. Figure 2 shows one module named *campgn.sim* with connections to three adapters A_1, A_2 and A_4 as they are needed for the promotional campaign in the example in section 3. Integration modules are comprised of logic (code, workflows) using the unified service interfaces USI_i of the adapters.

The advantage of a layered architecture is the introduction of abstractions (messages, events, federated service models, adapters, and service integration modules). It also allows the separation of concerns. Three main roles can be distinguished that participate in the service integration development.

First, *service model developers* concentrate on designing and providing current models of relevant services in the cloud. The richness of these models must allow the control plane integration scenarios and enable adapters to be build based on the information in the models.

Second, *service adapter developers* create reusable service adapters based on service models, reducing the scale of building integration adapters between n services from $O(n^2)$ to $O(n)$. Only one adapter needs to be built per service based on the service model. The main task here is to develop the specific implementation that establishes the connection with the service control plane interfaces CPI_i. In case services already offer service interfaces (e.g. WSDL or REST-based), adapter development can be simple. More advanced technologies must be applied when services do not provide programmatic interfaces. In those cases, user interface recording and replay-scripting techniques [18] can be used for building adapters.

The third is the role of the *service integration developer* with the responsibility of creating specific service integration modules such as the one for the promotional campaign in the example. Service integration developers primarily use service adapters with their unified service interfaces. The heterogeneity of underlying service control plane interfaces is hidden inside adapters allowing the integration developers to concentrate on the integration logic. For this, conventional programming (e.g. in Java) can be used as well as business process or workflow-based development. The result is a service integration module (*.sim*).

A layered architecture enables the separation of concerns and allows tasks being performed by different roles.

5. SERVICE MODELS

Service descriptions typically contain a location where a service can be accessed, along with an interface description which lists operations of the service along with their input and output parameters. This interface description for web services may be in the Web Services Description Language (WSDL) or may be in text format for REST-style services. However, interfaces do not provide adequate description of service behavior. While various annotation schemes for modeling service behavior are possible with WSDL, our approach has been based on the convergence of the principles of Service-Oriented Architecture (SOA) with those of Model-Driven Architecture (MDA). The service models exposed to the Services Integration Platform form the formal representation of the operations, state, associated entities, and meta-data information that capture all service behavior needed for integration.

Models defined as descriptions that are maintained separately have a serious problem in practice—as services are modified and evolved over time, the documents containing these descriptions are often not maintained in synchronization with the service logic. This causes developers, who rely on the documentation to make mistakes that require additional debugging and testing. At a minimum, additional effort is necessary in the development process to maintain both the code and the documentation. In our framework, annotations co-exist with the service logic within Java source and class files. When a particular service is started, the runtime system introspects the code and auto-generates the corresponding service model.

We use rich model descriptions that are based on the Common Information Model (CIM) [19], which is an object-oriented framework standardized by DMTF [20]. CIM is widely used in systems management software. To make CIM more developer-friendly, we introduced extensions to the CIM models that allow classes to be declared as interface classes. Any class declaration can then include such classes with the standard semantics of interface classes.

Figure 3 shows the CIM meta-model as defined by DMTF, extended to include interfaces. Note that it allows classes defined in the resulting models to have model elements such as properties and operations, as well as meta-data information in the form of qualifiers. Subtyping for the purpose of overriding properties and operations is allowed. Unlike standard UML models [21], references to other classes are restricted to association classes.

Models capture the dynamic state of service instances in the runtime system. This allows models to be exchanged and operated upon by the different layers of the service integration platform described in Section 3.1. However, it would make the work of the service adapter developers harder if they had to update these models as the service logic evolves. Hence, the approach taken in our service integration platform is the generation of these models at runtime based on code introspection. This ensures that the generated models are in sync with the code for the service logic.

Service model developers bootstrap the process of generating annotated Java code by writing the model in DMTF's Managed Object Format (MOF) [22] as shown in Figure 4 for class describing coupons in our earlier example.

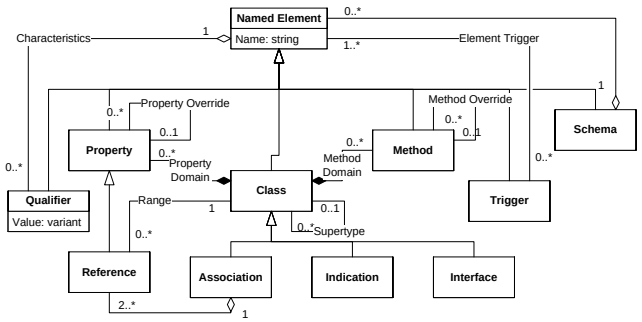

Figure 3: Meta-model used for creating models

A code generator uses the model to generate annotated Java stubs. The service adapter developers then add service logic within the code stubs. This is illustrated in the code segment in Figure 5, which shows the annotated Java class generated from the MOF definition for the Coupon class. The service logic within the setCouponNumber method has been added by the adapter developers. Note that the developers are not restricted to generating code from MOF definitions. They can also modify

```
[Version("1.0"),SuperClassVersion("1.0"),Provider("telecom.svc.coupon")]
class SIE_Coupon : SSP_Entity {
    SINT32 CouponNumber;
 [Write, ValueMap{ "Issued",  "Redeemed", "Invalid"}]
    STRING CouponStatus;
 [Description("Initialize coupon number exactly once to a positive value.")]
    BOOLEAN setCouponNumber(SINT32 couponNumber);
};
```

Figure 4: CIM/MOF class representation for SIE_Coupon.

both the class definitions and the annotations directly in Java, e.g., to add other operations and make them available in the model using @Export annotations. The introspection of the developers' code at runtime results in data exchanges among the service model providers. DMTF's CIM-XML [11], the XML-based serialization format, is used in these data exchanges, over HTTP.

The @ExportClass annotation implies that this Coupon class should be exported by the runtime to the Federated Model Store Layer. Parameters in this annotation provide optional information to the runtime system such as the version of this class and the schema within which this model resides. Since the model creator inherited from the abstract class SSP_Entity in the definition of SIE_Coupon, the generated class in Java inherits all properties and operations from BaseEntity which is defined by the platform run-time and corresponds to the model SSP_Entity. This class allows other services to receive notifications when changes are made to an instance of SIE_ Coupon. The @Export annotation is attached to attributes and operations that the runtime makes available through the model representation. The @Param annotation allows names to be assigned to the parameters in the exported model operations. The method signatures of getCouponNumber() and getCouponStatus() lead the runtime system to conclude that CouponNumber and CouponStatus are attributes that can be read from the model. Our implementation conforms to the CIM specification; hence the presence of a "setCouponStatus()" method, with return type void, indicates that the CouponStatus property is writable by service clients. The method signature of the "setCouponNumber()" method, however, leads the runtime system to conclude that it is an operation on the

```
@ExportClass(classVersion="1.0", schema = "SIE")
public class Coupon extends BaseEntity {

public enum Status {Issued, Redeemed, Invalid} ;
private Status couponStatus;
private int couponNumber = 0;

@Export
public Status getCouponStatus(){
  return couponStatus;
}

@Export
public void setCouponStatus (
   @Param(name="CouponStatus") Status couponStatus){
      this.couponStatus = couponStatus;
}

@Export
public int getCouponNumber(){
  return couponNumber;
}

@Export("Description(\"Initialize coupon number exactly once to a positive
value.\")")
public boolean setCouponNumber(
    @Param(name="couponNumber")  int couponNumber) {
       if ((this.couponNumber == 0) && (couponNumber > 0)) {
            this.couponNumber = couponNumber ;
            return true;
       }
       return false;
}
}
```

Figure 5: Generated stub code for service model.

model and not an operation for writing the CouponNumber property. This is because its return type is Boolean rather than void. The CouponNumber property in the model is not writable, and must be updated only through this operation, which allows the service logic to be enforced.

The service adapter developer can focus on the service logic, and can depend on the runtime system of the service integration platform [9] for event and message handling, and for service discovery, communication and version management. The burden of maintaining service models is minimized. The runtime system verifies that the model versions are compatible across services [10]. The extension of abstract classes in the development environment allows the developer to override standard service operations, such as subscription to model events, as needed.

6. RELATED WORK

Service providers today offer mostly services that are used individually and in isolation. Integration of services remains difficult, mainly due to the lack of programmable interfaces, although this is improving. Even where programmable interfaces exist, their variety and heterogeneity require adapters to be built that translate messages from one service to another. A number of platforms have been developed providing message brokerage capabilities in the data plane. CORBA [1, 2] and DCE [3] were early platforms that emerged before XML and web services. E-speak was one of the first early message broker platforms for web services [4]. Our work focuses on the control plane, and is orthogonal to techniques for message brokerage in the data plane.

Later, service integration progressed towards control flow coordination with various service orchestration and composition languages and frameworks. WSCI [5] and BPEL [6] are examples. Domain-specific process integration frameworks emerged for e-business such as ebXML [7] and RosettaNet [8]. Data translation and data transformation was addressed e.g. in the XML tool set with XPath, XQuery and XSLT. The eclipse ATL environment allows implementing powerful grammar-driven data transformations [12]. Our future work will address the generation of composition logic for service integration scenarios, and will address this part of the related work.

Integration of cloud services has been addressed recently by several companies. Bungee Connect [13] allows application developers to integrate web services with enterprise applications and data services being built using various technologies. Since it is targeted at developers, it is assumed that developers will understand the different technologies needed to integrate services. Cast Iron Systems [14] specializes in integration of cloud and on-premise services. The company builds integration templates that can be configured in their graphical designer. Cast Iron Systems focuses on major cloud services from Google, Salesforce, NetSuite, etc. These solutions are geared towards enterprise customers, while we have proposed a solution for the telecom service provider market.

Integration of services has also been covered in the research community [15, 16, 17]. Our work is distinct in its focus on the annotation of Java code by the developer, and generation of the model at runtime by introspection.

7. CONCLUSION

We have proposed a developer-centric approach to integration of services on the control plane. Our innovative approach is based on annotation of Java code maintained by the developer, which is introspected at runtime to generate the model that captures the dynamic state of the service. The annotation process ensures that the service models remain in sync with the service logic. Integration is driven at runtime by an event-notification layer that informs subscribed services to the changes in the modeled state of a service.

We are currently in the process of delivering our implementation as a capability in a HP product. Developers are integrating the first set of services based on our approach. This can lead to a case study in future. Our research is focused on generating the composition logic for service integration scenarios based on declarative descriptions of valid execution states.

8. REFERENCES

[1] Siegel, J. 1999. *CORBA 3 Fundamentals and Programming*. ISBN 0471295183, John Wiley and Sons, New York, NY.

[2] Object Management Group. 2008. Documents Associated with *CORBA*, 3.1: http://www.omg.org/spec/CORBA/3.1.

[3] The Open Group. *OSF Distributed Computing Environment Download Page*. http://www.opengroup.org/dce/download.

[4] Hewlett-Packard Company. 2001. *E-speak Specification*. Available: http://www.hpl.hp.com/techreports/2001/HPL-2001-138.pdf.

[5] W3C. 2002. *Web Service Choreography Interface (WSCI) 1.0*. Available: http://www.w3.org/TR/wsci.

[6] OASIS. *Web Services Business Process Execution Language Version 2.0*. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.html.

[7] OASIS. *ebXML Specifications*: http://www.ebxml.org/specs/index.htm.

[8] *RosettaNet*: http://www.rosettanet.org.

[9] Srinivasmurthy, V. et. al. *Web2Exchange: AModel-Based Service Transformation and Integration Environment*. In Proceedings of the IEEE International Conference of Services Computing, 2009.

[10] Karin Becker, Andre Lopes, Dejan S. Milojicic, Jim Pruyne, Sharad Singhal: *Automatically Determining Compatibility of Evolving Services*. ICWS 2008: 161-168.

[11] DMTF, WBEM: *CIM-XML*, http://www.dmtf.org/standards/wbem/CIM-XML .

[12] eclipse *ATL*: http://www.eclipse.org/atl.

[13] Bungee Connect. *Bungee Connect is Platform as a Service*. http://www.bungeeconnect.com.

[14] *Cast Iron Systems*, An IBM Company: http://www.castiron.com.

[15] Benatallah, B., Casati, F., Grigori, D., Motahari Nezhad, H.R. and Toumani, F. 2005. *Developing Adapters for Web Services Integration*. CAiSE 2005, DOI=http://dx.doi.org/10.1007/b136788 pages 415-429.

[16] Zhang, L., Zhou, N., Chee, Y., Jalaldeen, A., Ponnalagu, K.,Sindhgatta, R. R., Arsanjani, A., and Bernardini, F., *SOMAME: a platform for the model-driven design of SOA solutions*, IBM Systems. Journal, vol 47, pp. 397-413, July 2008.

[17] Kim, W., Graupner, S., Sahai, A., et.al.: *Web-E-Speak: Facilitating Web-Based E-Services*, IEEE Multimedia, ISSN 1070-986X, Vol. 9, No. 1, pp. 43-55, Jan/Mar 2002.

[18] Bergman, R., et.al. *A visual tool for rapid integration of enterprise software applications*. HP Technical Report, http://www.hpl.hp.com/techreports/2010/HPL-2010-29.html

[19] Distributed Management Task Force (DMTF), *Common Information Model (CIM)*, http://www.dmtf.org/standards/cim.

[20] *Distributed Management Task Force (DMTF)*, http://www.dmtf.org.

[21] Object Management Group (OMG), *Unified Modeling Language (UML)*.

[22] DMTF: *Managed Object Format (MOF)*, http://www.dmtf.org/education/mof.