A Service-Oriented Blueprint for COTS Integration: the Hidden Part of the Iceberg

Vina Ermagan, Claudiu Farcas, Emilia Farcas, Ingolf H. Krüger, Massimiliano Menarini
University of California, San Diego – Calit2
{vermagan,cfarcas,efarcas,ikrueger,mmenarini}@ucsd.edu

Abstract

The use of commercial off-the-shelf (COTS) software can greatly reduce the development cost and effort for complex software systems. Reusing software can also improve the general quality of a system by leveraging already proven implementations. One of the limiting factors in the adoption of COTS software is the complexity of integrating it with the rest of the system under development. Often, requirements do not entirely match the functionalities available in COTS components, increasing the complexity of the glue software that needs to be written.

In this paper, we present the blueprint of a Service-Oriented Architecture that can guide the engineer both in specifying the functionalities of a complex software system and as a deployment architecture to seamlessly integrate COTS components implementing such functionalities. The COTS integration concern, typically a deployment issue, is addressed in the service architecture, and is treated as first-class citizen of the development process.

1. Introduction

The term commercial off-the-shelf (COTS) software describes software components used as part of a larger software system but not developed for it. They are generally available only in binary form, preexist the software that uses them, and can be based on arbitrary implementation technologies. Even in-house developed and legacy software are often integrated as components in new larger software systems and treated as COTS.

Incorporating COTS software in large-scale complex applications can be extremely beneficial as it potentially reduces development cost, while increasing the overall software quality. However, the encapsulation, heterogeneity, and complexity of COTS components make the integration work challenging. Furthermore, due to inaccessibility of the source code, reuse of COTS software cannot leverage traditional code-reuse techniques and needs additional glue code for adaptations.

Despite the common tendency towards reuse of COTS software, the lack of a systematic, methodological approach in COTS-based systems development greatly limits the benefits offered by using COTS software. COTS components typically present interoperability problems and interface mismatches, which should be addressed at the architectural level [22]. We believe that the key for COTS software integration is a loosely-coupled architecture that allows specific policies to be enforced by pluggable entities.

Service-Oriented Architectures (SOAs), which partition system functions into logical, homogeneous modules, have emerged as a convenient solution to create low cost, loosely coupled, interoperable systems. An SOA hides the implementation details of the components that provide the functionalities, making it particularly suited for the integration of COTS software. In this paper, we introduce the blueprint of an architecture that maps well to an SOA and that eases the integration of COTS software. The same architecture can also drive the implementation of a deployment architecture based, for example, on service-bus middlwares [8].

We start with an outline of a service-oriented process leading to a service-based specification of system functions. Then, we analyze how such a specification can be refined to take into account explicitly the interaction between the system and the COTS components. In Section 4, we present a blueprint for a service-oriented architecture and discuss patterns that ease the integration of COTS software. An overview of related work and conclusions round out the paper.

2. Service-oriented development process

In this section, we present a service-oriented development process that establishes a clean separation between the logical model (the services provided by the system) and an implementation model (the implementation architecture of the services).
In our interpretation, a service describes an interaction pattern of domain objects that provide a function to the system. We can see this interaction as the realization of a “use case”. Services describe, therefore, interaction scenarios; each service “orchestrates” interactions among system entities to achieve a specific goal [1]. Services are therefore partial specifications of the system behavior [9].

![Figure 1 Service-Oriented Development Process](image)

*Figure 1 Service-Oriented Development Process*

To create a service-oriented system, we follow the two-phase, iterative development process shown in Figure 1. In phase (1), Service Elicitation, we define the logical model as the set of services the system must implement. In phase (2), Architecture Definition, we define a deployment architecture and map the services of the logical model to components of the deployment architecture.

Phase (1) is carried out by first identifying the relevant use cases and their relationships in the form of a use-case graph, which gives us a large-scale, scenario-based view on the system. From those scenarios we derive a set of roles, representing logical entities interacting in the use cases, and then services formally describing the interaction patterns among roles. To formally describe services, we use a dialect of Message Sequence Charts (MSC) [7]. The use of logical roles as participants in the interactions abstracts from implementation details. Roles describe the contribution of an entity to a particular service independently of which component will provide the contribution in the concrete implementation. An implementation component will typically play multiple roles at the same time.

In phase (2) a component configuration is defined. Each role participating in some service is then mapped to one or more components, defining the architectural configuration of the system. The architectural configuration is the starting point for implementation.

The process is iterative both within the two phases, and across. Role and service elicitation feeds back into the definition of the use-case graph; component configurations can be refined and refactored to yield new architectural configurations, which may lead to further refinement of the use cases.

### 3. Implementing services with COTS

As described in Section 2, we define a service by the interactions of roles expressed in MSCs. Each of such services describes a functionality provided by the system. The first step towards implementation is to refine the interaction-based model to a level of granularity that identifies not only the messages exchanged between entities, but also the data elements transported by each message and the local processing executed by each entity on such data. We capture this information in our modeling language by means of message parameters and local actions, respectively. Once the model is refined to this level of detail, the developer should have a good understanding of what functionality each role needs to implement in its local actions.

At this point, we can identify COTS software that can help in implementing each service, and local actions of roles can be replaced by calls to functionalities of existing COTS modules. The role interface shields the rest of the system from the details of interacting with the COTS components. A role can leverage functionalities of different COTS software and, on the other hand, each COTS component can be used by different roles. To capture the, possibly complex, interactions between roles and COTS components in our system, we explicitly describe the interaction pattern of each COTS software with the roles that leverage it. This methodological step lifts the problem of COTS integration from the deployment to the logical level; it provides a description of a COTS software as a service provider (supporting a predefined protocol) and of the roles calling upon its functionalities, as clients of this service provider. Additional roles can be introduced to deal with translations or protocol-adaptations required to fulfill the requests of the original roles.

Thus, COTS software is modeled as additional roles in the service specification of the system. The roles of the original specification (client roles) can now, instead of implementing their local actions, send messages to the relevant COTS roles to request the functionalities (possibly via additional roles that adapt the request to the protocol and data format supported by the COTS components).

These client roles are the interface points between the system and COTS components. Together with the additional roles, introduced to capture the adaptation to COTS protocols, they are responsible for providing the protocol translation, and are capable of implementing specific protocol matching functionalities such as instrumentation and reasoning [17]. We are, therefore, able to account for differences between the communication protocols expected by the various COTS components and the one adopted by the system itself.

We can better explain this approach by a simple example. Assume we want to build an online book store. Some roles that appear in our model are the *buyer*, the *bank*, the *delivery company*, and the *warehouse*. Con-
sider the “acquire book” use case where a buyer wants to buy a book. Assume that, in the corresponding service, the role warehouse needs to have an inventory and performs two functions on it: check-out and remove-the-bought-book. The required functionality can be provided by an old program P already in place. However, P requires additional input such as an authorization in the form of a digitally signed file, not received by the warehouse role in any service it is engaged in. We can then refine the model adding a role P and a role Authorization. When the request is received by the warehouse role, warehouse forwards the request to the authorization role, which generates an authorization file and sends it to P along with the original request. Thus, we have explicitly captured the differences in the protocol and clearly separated what needs to be written to adapt the COTS component and what is handled by it.

As illustrated above, the proposed service-oriented development process helps to manage the complexity of using COTS components by leveraging a clear graphical framework to capture interactions in form of MSC diagrams. In the next section, we introduce an architecture that allows us to readily model COTS-based systems according to the service-oriented approach described and a deployment infrastructure addressing the integration requirements originated by the use of COTS software.

4. A flexible architecture blueprint

The development approach proposed for COTS software integration isolates the interactions with COTS components from the behavior of the rest of the system. To address this key requirement we propose the use of the architecture depicted in Figure 2 both for the logical models (service layer) and as a deployment architecture (service bus).

We take two different perspectives in analyzing the blueprint of Figure 2. First, we show how it provides a structure for the logical architecture, around which services and their integration with COTS software can be defined. Second, we use the blueprint as a deployment architecture for services implementation, describe how such a loosely-coupled architecture can be implemented using readily available software, and show that by leveraging standard architectural integration patterns an easy incorporation of COTS software can be achieved.

4.1. Logical architecture for the service layer

In Figure 2, we capture the proposed logical architecture, inspired by ESB architecture/implementations [8][19], for COTS components integration. The architecture is organized around a message-based communication infrastructure and is hierarchically decomposed into “capability blocks”. Each such a block is connected to a communication channel of other blocks via a “Service/Data Connector”, whose task is to encapsulate the internal structure of the block and offer to the rest of the system a service-oriented view of it.

In our approach, we decompose the system functionality into services that capture the supported use cases. The system is decomposed according to the functionalities it provides, rather than according to implementation concerns. Our communication infrastructure with the “Router/Interceptor” component allows seamless communication between the various services. Moreover, the messaging infrastructure enables loose coupling between the different services and maps well to our interaction-based specification using MSCs.

The communication infrastructure has two layers: the first layer - the “Messenger” component is in charge of moving messages between endpoints, and the second layer - the “Router/Interceptor” component is in charge of addressing the messages to the right set of endpoints. The use of a “Router/Interceptor” layer removes dependencies between services and their relative locations in the logical hierarchy. Therefore, new services can be plugged into such an architecture and provide their functionalities to the system without the need to change or adapt the remaining services.

As an example, we can consider the Encryption Plugin in Figure 2. The task of the Encryption Plugin is to ensure that all messages transiting on an open communication medium get encrypted at the origin and decrypted only at the destination point. A traditional implementation strategy would require changes in the sending and receiving entities and a priori knowledge of the channels and the entities’ configurations to ensure encryption. The suggested architecture, instead, allows us to simply deploy an Encryption Plugin in the system. The routing layer can be informed by the plugin that every message sent to the external network needs to be processed by it. The communication layer is the only entity that needs to have a detailed knowledge of the physical communication infrastructure. Moreover, this supports keeping the encryption concern separate from the functional behavior of the respective end points.

The hierarchical nature of our architecture allows for an easy embedding of COTS components. In our process description, we have outlined how it is possible to explicitly represent a COTS component in the service description and to capture the interaction between the role defined in the service and the COTS component supplying the implementation of the required functionalities. In the proposed architecture, the logical role can be implemented by a capability block
connected to the messaging infrastructure. Internally, the capability block is organized according to our basic block architecture. COTS components can be plugged into the messaging infrastructure by means of a gateway (see the Messaging Gateway pattern [10]). This entails ad-hoc code that deals with the common communication infrastructure and adapts requests and replies to the communication primitives required by the COTS software.

Leveraging this architecture, we can provide plugins that capture all interaction protocols between the various roles and the COTS components, and expose the role interfaces via the “Service Connector” to the rest of the system. An interesting aspect of this approach is that many functions provided by the same COTS software could be exposed via the “Service Connector” of the same “capability block” simply by means of additional plugins.

MSCs can completely define the control part of the communication protocol [7] between COTS and service interfaces. Data transformation and elaboration, however, have to be defined by other means. Therefore, the adaptation will include a message-exchange protocol described by MSC messages, and data transformations captured as local actions. The proposed logical architecture enables the creation of separate plugins to execute the data transformations and to orchestrate message flow. The decoupling of data transformation from the service interaction logic allows for the reuse of transformation code. Moreover, because the logic needed to orchestrate the message flow is captured by an MSC, we could leverage our work on state-machine generation to synthesize it [6][7][9]. Therefore, this approach would likely minimize the amount of code to be written to integrate COTS components. In addition, each module that needs to be hand coded has a clear specification and a simple, well-defined goal.

4.2. Deployment architecture for services

The same architecture of Figure 2 can also be used as an implementation architecture leading to an Enterprise Service Bus (ESB) solution. Similar architectures have been used in enterprise integration efforts with good results [8]. The communication infrastructure can be implemented by message-bus applications available both as free open-source programs, such as ActiveMQ or JBoss Messaging, and as commercial applications, such as WebsphereMQ or SonicMQ. Such a messaging infrastructure provides not only a communication solution, but also allows for configuring channels, routing, and communication policies with a great deal of flexibility. The communication can be configured to use several protocols, from proprietary to standard web-service based ones. Several plugin modules described in Figure 2 can be readily configured from available software without the need to write additional adapters or translators. In fact, the main focus of messaging solutions is to address integration; thus, many adapters are already available to plug into such infra-
structure; for example, authentication, encryption, storage, logging, and policy modules from different providers. In addition, the “Router/Interceptor” layer of the Figure 2 can be readily implemented using available software. Mule [19], for example, is a Java open-source ESB and message-broker component that can fulfill that role in a message-bus based implementation of our blueprint.

An implementation based on standard API and leveraging available messaging infrastructures requires a minimal amount of code to be written and enables the generation of many orchestration and translation components from the service model of the application. The one-to-one mapping between logical and deployment architectures and the decoupling of components both in the logical and implementation layer simplify the evolution of the application during its life cycle.

The blueprint presented in this paper fits other domains as well; for example, the embedded systems domain. In this case, the technology mapping could include, for example, the CAN bus as Messaging infrastructure and leverage AUTOSAR [20] technologies for the “Router/Interceptor” functionality.

4.3. Patterns for COTS integration

The service-oriented integration blueprint presented in this paper leverages many well known architectural patterns. The main architectural decision is to decouple the different parts of the systems by adopting an asynchronous communication pattern. To this end, we adopted the Messaging pattern where the communication is carried out by Messages exchanged over Messages Channels [10]. The other key architectural decision is to provide the “Router/Interceptor” layer, an instance of the Mediator pattern [4] that removes services interdependencies. Furthermore, the Plugin pattern [2] inspired our pluggable architecture to decouple the specification and implementation of services functionalities from configuration details.

Other key patterns used in our architecture are the Façade, Proxy, and Adapter patterns [4]. They inspired our “Service/Data Connector”. The result of using them is the encapsulation of “capability blocks”. A service-oriented interface allows them to interact with the rest of the system, while hiding their internal structure. Finally, the Composite pattern [4] enables the organization of subsystems via the hierarchical nesting of “capability blocks”. This feature, in particular, allows the architecture of Figure 2 to cater extremely well to the needs of COTS software integration. In fact, each “capability block” can adapt one specific COTS module. The interaction with the rest of the system is formally specified by the service signature implemented by the “Service/Data Connector” and the adaptation logic is decomposed in plugins, each providing an elementary and reusable functionality.

5. Related work

Our approach for COTS software integration relies on the notion of “service”, used in many different application domains and on various levels of abstraction in the Software Engineering community [5]. Its roots lie in the domain of telecommunication systems, where features and their interactions play an important role in software development [11][12]. Intensive application of service-oriented approaches can be observed for web services [3], web service-oriented architectures [13][14], and increasingly for complex, embedded automotive systems [15][16].

The complexity of COTS integration arises mostly form architectural mismatch [22]. Conflicting assumption about infrastructure, control or data models, incompatible connection strategies, differences in architecture and component instantiation make tightly integration of COTS often impossible. The use of a loosely-coupled asynchronous messaging infrastructure for COTS integration enables our service-oriented approach to overcome architectural mismatches.

Various wrapper-based approaches have been introduced to deal with COTS integration. For example, in [25] connector wrappers are characterized as protocol transformers, in [24] wrappers are used to adapt COTS that do not completely match the functional requirements of the integrating application, and in [23] wrappers are used to integrate access to legacy data-management systems. Our approach is similar in the fact that a capability block encapsulating some COTS can be seen as a wrapper. However, our methodology provides a guidance to decompose the transformations needed for the adaptation, uses a service-oriented approach that allows focusing only on the interaction used in the application, and promotes the use of a graphical language to specify such protocols. Furthermore, leveraging enterprise integration patterns and a messaging system [10] configured in an adaptable ESB [8] architecture, our approach removes the coupling between services and the transport medium.

Standards for component configuration and deployment, such as the platform-independent OMG standard [18], expect components to be standard compliant and provide configuration metadata, thus, limiting their applicability. Our generic approach can be extended to leverage standard metadata if available, but it does not require the components to adhere any standard.

The use of service-oriented architectures to support COTS integration is not new. In [21] an SOA architecture and four basic adaptation services have been pro-
proposed for COTS integration. Our work expands those concepts by providing a process and a hierarchical service architecture and by suggesting a modeling language (MSC) to specify the integration protocols.

6. Conclusion

We presented a service-oriented development process and introduced an architecture blueprint particularly suited for the integration of COTS software components into complex applications. This provides a unified and general methodology for COTS integration that lifts the concern of COTS integration to the service specification level and subsumes a wide variety of techniques currently in use. The proposed approach minimizes the downsides of using COTS software by leveraging a flexible communication middleware and a clear and formal specification of the glue code needed to accomplish the integration. In addition, parts of the integration code could be automatically synthesized, thus, reducing costs and risks of using COTS software on large-scale systems.

Our tool M2Code already supports the modeling of services using MSCs and the generation of state machines from those models. Furthermore, our code generator is currently able to produce C code for the RT-CORBA real-time communication infrastructure. Nevertheless, in order to fully validate our approach, more experimental work is needed. We plan to support Java and the Mule ESB implementation in our code generation tools to be able to experiment with the deployment architecture described in Section 4.2.

Acknowledgements

This work was partially supported with funds from the California Institute for Telecommunications and Information Technology (Calit2). We thank Matthew Arrott for insightful comments on the architecture blueprint and Barry Demchak for valuable suggestions regarding COTS software.

References