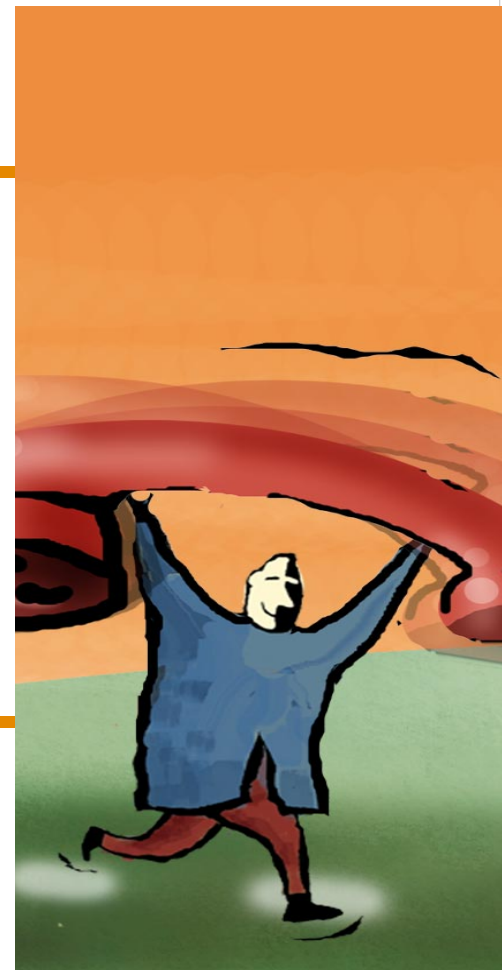

SERVICE ORIENTED DESIGN AND ARCHITECTURE: SOLVING TELECOM SYSTEMS INTEGRATION PROBLEMS

Telecommunications industry faces a number of problems while implementing the services. Cost and complexity of building and maintaining service management systems is very high especially when new services have to be introduced to keep pace in a highly competitive environment. Telecommunications service providers are moving towards the use of off-the-shelf componentware to satisfy their management requirements. However, a crucial problem with such an approach is the ability to integrate components to realize integrated management solutions.

This paper describes the solution architecture that provides an end-to-end solution to a recurring problem in telecom system integration. The architecture and design principles in solution realization are also discussed.



WHITEPAPER



KOTESWARA RAO MEDIDA



Table of Contents

INTRODUCTION 3

BUSINESS FACTORS DRIVING COMPONENT BASED ARCHITECTURES 3

ARCHITECTURE AND DESIGN PRINCIPLES TO MEET
CHANGING REQUIREMENTS 4

INTEGRATION STRATEGIES 5

SOLUTION ARCHITECTURE 12

SOLUTION REALIZATION 12

CONCLUSION 15

REFERENCES 15

ABOUT THE AUTHOR 15

ABOUT WIPRO TECHNOLOGIES 16



INTRODUCTION

Various issues arise with service line architecture involving integration of products with customized solutions. For a smooth application integration to support telecom management business processes one requires a fair understanding of architectural principles, solution architecture, and solution realization. Component integration technologies are expected to be the key to the development of future operational support systems. The state of the relevant component integration technologies is reviewed in the context of these requirements. This paper draws some conclusions on the relative merits of the different types of integration methodologies and makes some suggestions for further work.

BUSINESS FACTORS DRIVING COMPONENT BASED ARCHITECTURES

The liberalization of telecom markets across the world has exposed service providers to a high level of competition. This requires them to reduce costs, improve customer service and rapidly introduce new services. One important way in which these pressures can be addressed is through improved integration of the many software systems operated by a service provider. This includes amongst others, the integration of different operation support systems. Component based reuse is seen as an increasingly important software development aid, both within the telecom industry and in the wider IT community. Building systems from components that interact through well defined interfaces offer a route for reusing software across projects within a telecom system developer and to integrate commodity third party software into the system. Both of these offer development cost savings and improvements in reliability and maintainability. Emerging standards such as Enterprise JavaBeans and CORBA Components are encouraging the development of platforms that directly support component integration. This is prompting the telecom industry to move towards the widespread adoption of component-based architecture and design.

Certain business factors drive requirements for revolutionary integration. For telecommunications service providers, these include reductions in Total Cost of Ownership (TCO) and time-to-market new services, both of which provide a competitive advantage through improved cash flows.

Factors that influence TCO fall in two categories: 1) cost of building the solution and 2) cost of operations and maintenance. The first cost is tied to the time and effort required to build, test, and configure the solution, which is influenced by solution modeling approaches and the availability of appropriate tools.

Many factors influence operations and maintenance costs. These include the effort to integrate new applications, the effort to implement changes in process definition and messaging requirements, the ease of configuring, monitoring and troubleshooting, and so on. The solution should also be future-proof relatively insensitive to technology changes. Choosing the right solution architecture can significantly reduce both these costs. The history of information technology points to more sophisticated solutions using higher levels of abstraction. Higher levels of abstraction lets you focus on the essential features of the problem. One way to do this is to use the right models. The remainder of this article presents a solution architecture that addresses the business imperatives mentioned above. The solution described here is based on the Next-Generation Operations Systems and Software (NGOSS) initiative from the Tele Management Forum (TMF) [3]. Another initiative that addresses the Implementation issues is OSS/J [4] initiative from SUN.



ARCHITECTURE AND DESIGN PRINCIPLES TO MEET CHANGING REQUIREMENTS

Most of the legacy software systems are tightly coupled and are finding it difficult to keep pace with the rapidly changing business scenario. Early software systems were relatively static and any change required in the functionality would demand extensive editing of the source code. In contemporary systems, the changes are anticipated early and the variants in architecture and design are incorporated in early stages. These systems are more dynamic. Variability mechanisms have been present in product line architectures [1]. They are equally applicable in service line architecture also. Variability can be handled at different levels of abstraction. Variation points can be introduced at architecture, detailed design and implementation phase.

The intention of having a provision for a variation point in a system is to be able to insert a variant at a later stage.

In component-based design, the component can be replaced easily without changing the calling component. The source code dependencies between components are minimized. With dynamic binding, the system will be able to use new components at run-time; the system need not be shut down.

A design that doesn't take change into account is likely to face the risk of redesign within a short span of time whereas a system with a design that is robust to changes lasts long. The variable elements in a design need to be identified and addressed early to make the system robust to changes.

Some of the common causes of designing for change and how design patterns address them are [2]:

1. Creating an Object by Specifying a Class Explicitly

Specifying a class name when creating an object commits a designer to a particular implementation instead of a particular interface. This commitment can complicate future changes. To avoid it, create objects indirectly.

Design patterns: Abstract Factory, Factory Method, Prototype

2. Dependence on Specific Operations

When a particular operation is specified, there is one way of satisfying a request. By avoiding hard-coded requests, it is easier to change the way a request gets satisfied both at compile-time and at run-time.

Design patterns: Chain of Responsibility, Command

3. Dependence on Hardware and Software Platform

External operating system interfaces and application programming interfaces (APIs) are different on different hardware and software platforms. Software that depends on a particular platform will be harder to port to other platforms. It may even be difficult to keep it up-to-date on its native platform. It's important therefore to design your system to limit its platform dependencies.

Design patterns: Abstract Factory, Bridge

4. Dependence on Object Representations or Implementations

Clients that know how an object is represented, stored, located, or implemented might need to be changed when the object changes. Hiding this information from clients keeps changes from cascading.

Design patterns: Abstract Factory, Bridge, Memento, Proxy

5. Algorithmic Dependencies

Algorithms are often extended, optimized, and replaced during development and reused. Objects that depend on an algorithm will have to change when the algorithm changes. Therefore algorithms that are likely to change should be isolated.

Design patterns: Builder, Iterator, Strategy, Template Method, and Visitor



6. Tight Coupling

Classes that are tightly coupled are hard to reuse in isolation, since they depend on each other. Tight coupling leads to monolithic systems, where you can't change or remove a class without understanding and changing many other classes. The system becomes a dense mass that's hard to learn, port, and maintain.

Loose coupling increases the probability that a class can be reused by itself and that a system can be learned, ported, modified, and extended more easily. Design patterns use techniques such as abstract coupling and layering to promote loosely coupled systems.

Design patterns: Abstract Factory, Bridge, Chain of Responsibility, Command, Facade, Mediator, and Observer

7. Extending Functionality by Sub Classing

Customizing an object by sub classing often isn't easy. Every new class has a fixed implementation overhead (initialization, finalization, etc.). Defining a subclass also requires an in-depth understanding of the parent class. For example, overriding one operation might require overriding another. An overridden operation might be required to call an inherited operation. And sub classing can lead to an explosion of classes, because you might have to introduce many new subclasses for even a simple extension.

Object composition in general and delegation in particular provides flexible alternatives to inheritance for combining behavior. New functionality can be added to an application by composing existing objects in new ways rather than by defining new subclasses of existing classes. On the other hand, heavy use of object composition can make designs harder to understand. Many design patterns produce designs in which you can introduce customized functionality just by defining one sub class and composing its instances with existing ones.

INTEGRATION STRATEGIES

For telecommunications companies, quickly deploying and offering new services is imperative for competitive advantage. Systems that incorporate this ability have stringent demands in scalability, reliability, and real-time interaction with network elements. For example, creating, deploying, and offering a new service requires complex interactions between several disparate systems, some of which may be legacy systems (Refer Figure 1). The problem is how to achieve this integration in a speedy, cost-effective and flexible manner. It's important to minimize costs for building and maintaining integration solutions. There are various approaches of solving integration problems and the architectures vary. The example given regarding telecommunications applies in any domain. The focus here is on architectural issues – not implementation.

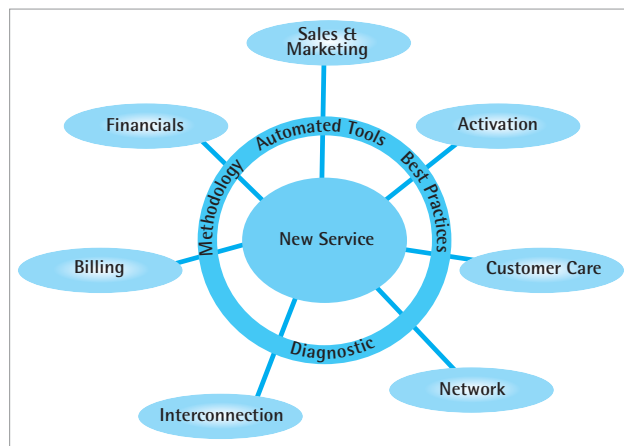


Figure 1 - Application Portfolio for New Service Offering



Point-to-Point Integration

In point-to-point integration, communication channels are developed between each pair of applications (Refer Figure 2). Such a solution practically not possible as the number of interfaces required grows exponentially. With n applications, $n(n-1)$ interfaces may be required since each application may need an interface with very other application. In practice, however, each application will, on average, requires communication with 30 percent of the other applications. So, with eight applications, adding a ninth will require the development of $8 \times 0.3 \times 2$ or about five extra interfaces. The impact of minor changes in communication requirements and that of adding a new application is significant while maintenance is clearly a nightmare.

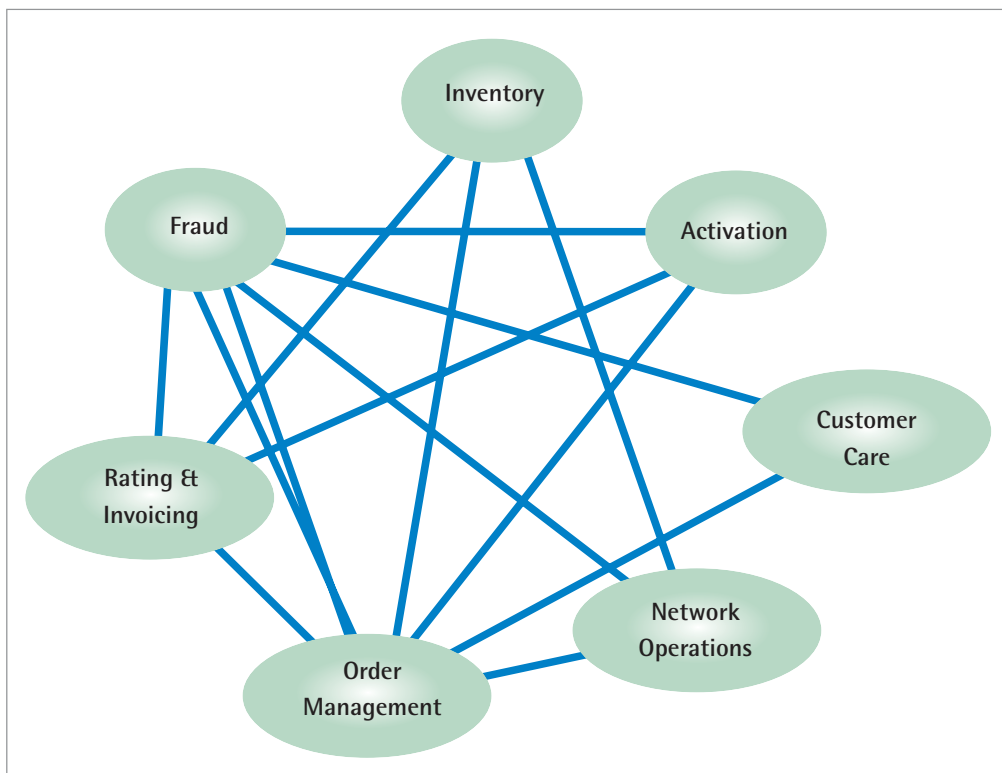


Figure 2 - Point-to-Point Integration



Middleware – Message Oriented Integration

Point-to-point integration exponentially increases the number of interfaces. This can be reduced to a linear increase through the use of middleware – message-oriented or based on the Common Object Request Broker Architecture or CORBA (Figure 3 illustrates this concept).

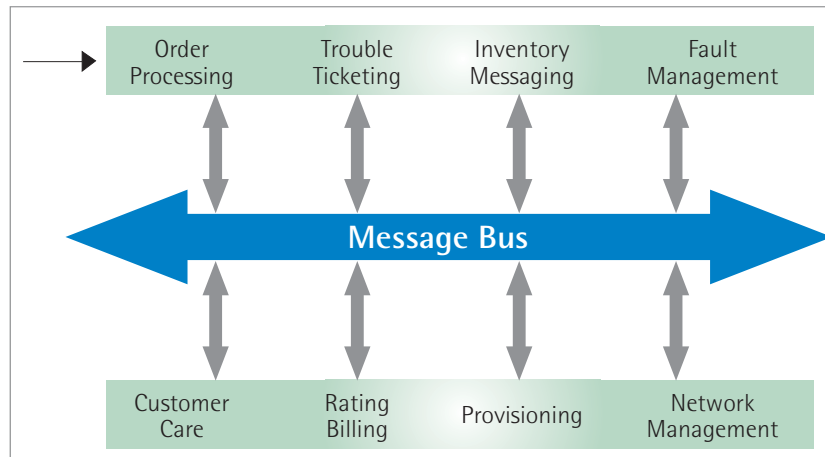


Figure 3 - Integration Using Message Bus

The solution requires interfacing each application to the message bus through an adapter. Each application has only one programmatic interface, the message bus. Applications communicate by publishing a message to the bus, which delivers the message to those who subscribe. Subscription topics or queues let subscribers receive only messages they're interested in. There are variations on this approach; but the concept is the same. Middleware products may also provide value-added services such as guaranteed delivery, certified delivery, transactional messaging, message transformation (using brokers), and so on. However, this solution may not be logically much different from a point-to-point solution if every application needs a different messaging interface with every other application. Even though there's only one programmatic interface through the adapter, that interface consists of several logical messaging interfaces, depending on how many different message types the adapter needs to process. This represents a significant improvement over the point-to-point solution, but there are several important issues:

What's the format in which data is interchanged?

How does the application interpret a message it receives?

What's the impact of adding a new application?

What's the impact of changing a message definition?



This solution depends on pre-defined data structures. The adapter for each application is programmed so it can handle only data structures defined at build time. As for the data interchange format, XML [4] is becoming popular because it's "self-describing." XML is superior to proprietary formats, but it's not a panacea since both sender and receiver must share a common vocabulary. Without a common vocabulary, the receiver won't know how to interpret the data. If a message represents an order, both the sender and receiver must agree that the data encapsulated in an <ORDNO> tag represents an order number. Changes in message definitions will require adapters to be re-coded, with consequent maintenance problems. Another, more serious problem is that the process flow is embedded in the application or adapter logic. Consider Figure 3, the logic regarding what to do next is generally coded into the adapter for a particular system. It's practically impossible to visualize the process flow at a high level of abstraction. Changes in the flow will require changes in application logic. (Actually, even though the application data is loosely coupled, the application processes are tightly coupled.) The solution is to externalize the process flow and have it driven by a process engine.

Process Engine Based Integration

In this approach, a graphical build-time tool typically defines the process flow. This process definition is then exported into some standard format and loaded into the process engine's run-time environment.

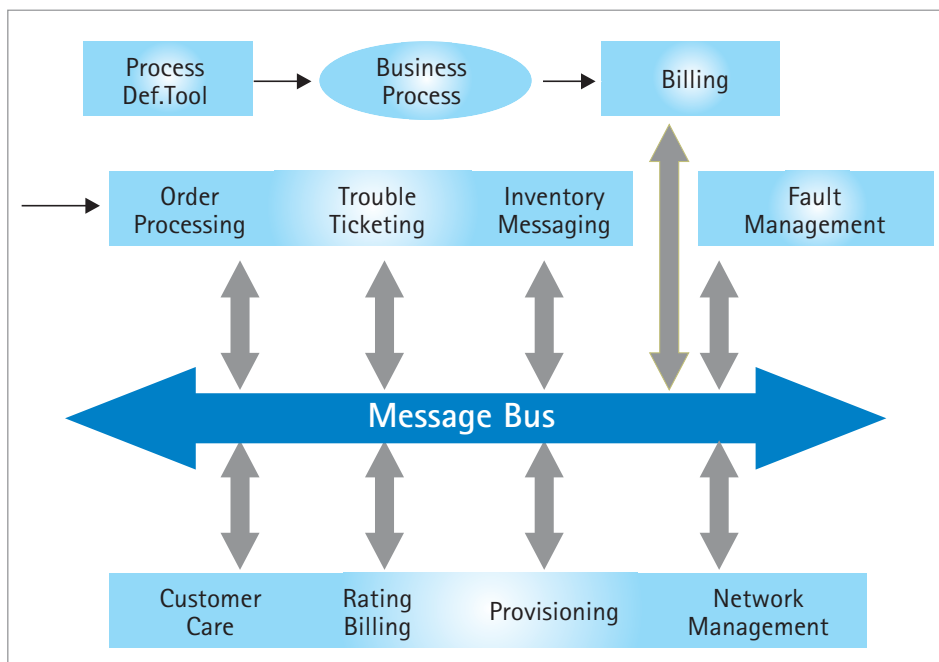


Figure 4 - Integration Using External Process Engine



The benefits include a better understanding of process flow; hence a better design of messaging requirements. Easier maintenance and easier accommodation of changes in process flow are the other advantages. Higher scalability means multiple process engines, and it can support an increasing load.

Although this approach provides significant advantages, there's still considerable coding that must occur when, for example, a new application is introduced. And a number of problems remain: changes in the process flow will require re-coding the process engine adapter, messages are determined statically at build-time and changes in message definitions will still require recoding the adapters. Introducing new applications require a significant amount of configuration and coding. Also, the solution isn't a plug-and-play operation.

Revolutionary Integration

Application integration has been following an evolutionary approach. Now it's time for a revolutionary approach for integration. Table1 shows key architectural principles of revolutionary integration and their benefits. Although these principles primarily target telecommunications OSS, they're generic enough to apply in any problem domain.

Commercial Off-the-Shelf (COTS) is a well-established industry initiative. The Software Engineering Institute (SEI) has a well-documented COTS-based system initiative. A contract describes services offered by a COTS component. The Workflow Management Coalition is actively involved in standards for process definition.

Examples of policies are security policies, which specify authentication, access control, and encryption requirements between components and contract trading policies, which are used to select one from a set of applicable contracts. Work that has been done in other areas, such as policy based network management, can provide important inputs here.

The registry will typically contain information about process definitions, data models, contracts, and the status of COTS components and adapters.



Principle	Benefits	Issues
Use Commercial Off-the-Shelf (COTS) components to build systems	<ul style="list-style-type: none">• Rapid construction of integration solutions	<ul style="list-style-type: none">• Selection of appropriate COTS systems• Mismatch between system requirements and COTS capabilities
Use of contracts to invoke COTS functionality	<ul style="list-style-type: none">• Plug-and-play operation• More robust systems through contract enforcementReduced testing effort	<ul style="list-style-type: none">• Industry wide agreement on contract specifications• Vendor support for standardized contracts
Use of an external process engine	<ul style="list-style-type: none">• Minimal impact of definition• Easier maintainability• Better visualization of process changes in process	<ul style="list-style-type: none">• Standard process definition language• Process engine pluggability
Use of shared framework services such as communication, security, and business process management	<ul style="list-style-type: none">• Better maintainability of the solution due to separation of concerns	<ul style="list-style-type: none">• Use of standardized interfaces or contracts to access these services
Use of policies to regulate system, behavior use of a registry to store build-time and run-time information	<ul style="list-style-type: none">• Superior system administration due to declarative approach• Superior control, monitoring, and troubleshooting• Rapid configuration/re-con-figuration of solutions	<ul style="list-style-type: none">• Standardization of policy scope, definition, and usage• Registry schema definition and interoperability

Table 1 – Revolutionary Integration Architectural Principles



In Revolutionary integration solution, each COTS component provides a service. Furthermore, COTS components can be classified as either Business Service Components (BSCs) or as Framework Service Components (FSCs). BSCs offer services such as order processing, provisioning, and billing. FSCs offer services such as communications, workflow management, policy management, and security. Each COTS component supports a well-defined interface called a contract. The contract specifies a set of services offered by a particular component.

The definition of each service includes name of the service, parameters required providing it defined in terms of objects in the Shared Data Model (SDM), conditions under which the service will be provided (pre-conditions), conditions that may arise after the service is provided (post-conditions), problems that could arise when the service is provided (exceptions), and the type of communication that should be used to invoke the service (message, CORBA, etc.).

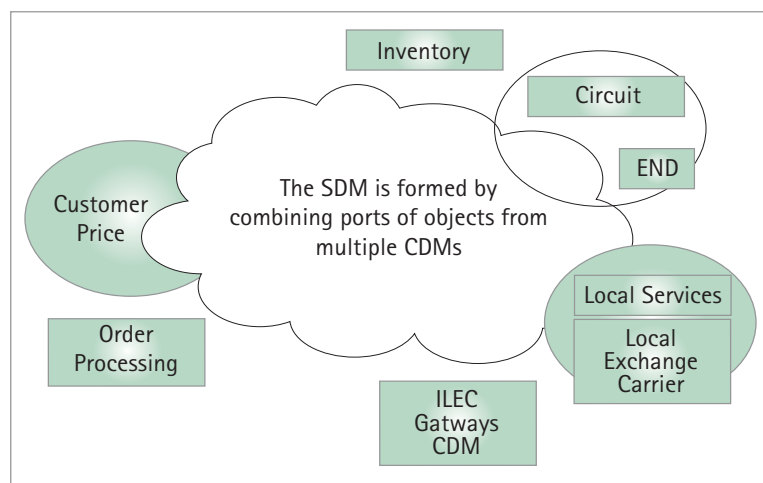


Figure 5 – SDMs and CDMs for Provisioning

Pre-conditions, post-conditions, and exceptions are an important part of a contract specification and go a long way toward building robust systems. The input and output parameters of contracts are specified in terms of objects defined in an SDM. An SDM is a model of data that needs to be shared between COTS components. It represents a common, external representation of information that needs to be exchanged between COTS components. This means that, when a COTS component wants to communicate with another, it will convert it into SDM objects. Each COTS component has its own data model. The portion of this data model that needs to be shared with at least one other application will be called a COTS Data Model (CDM). The SDM, therefore, represents the union of all the CDMs. Figure 5 illustrates this.



SOLUTION ARCHITECTURE

Figure 6 shows a representative architecture for Revolutionary integration solutions. Two important features of this solution are the API and the environment manager. This solution requires deploying a special type of API that will:

- Choose the appropriate type of communication service – message bus, CORBA, Remote Method Invocation (RMI), etc.
- Translate between COTS-specific interactions and contract invocations
- Enforce permissions for contract invocations
- Enforce security policies
- Trade contracts using policies

The environment manager will be used to:

- Configure the environment
- View the status of process instances, API, components etc.
- View alerts caused by abnormal situations

SOLUTION REALIZATION

The architectural principles described earlier are essentially technology-neutral. Realizing these principles will require mapping them to technology-specific implementations, which requires making choices on platforms, standards, and programming languages. Technology specific implementations are beyond the scope of this article, but there's a good likelihood that many of the following technologies will be used to implement revolutionary integration solutions:

- Java 2 Enterprise Edition (J2EE) (OSS/J APIs)[6]
- XML(OSS/J APIs)
- Lightweight Directory Access Protocol (LDAP)
- .NET
- CORBA
- Message-Oriented Middleware (MOM)
- Application servers
- Web services



The revolutionary integration solution will typically operate as follows:

Each COTS component plugs into the environment, indicating its willingness to offer a set of services as defined by contracts.

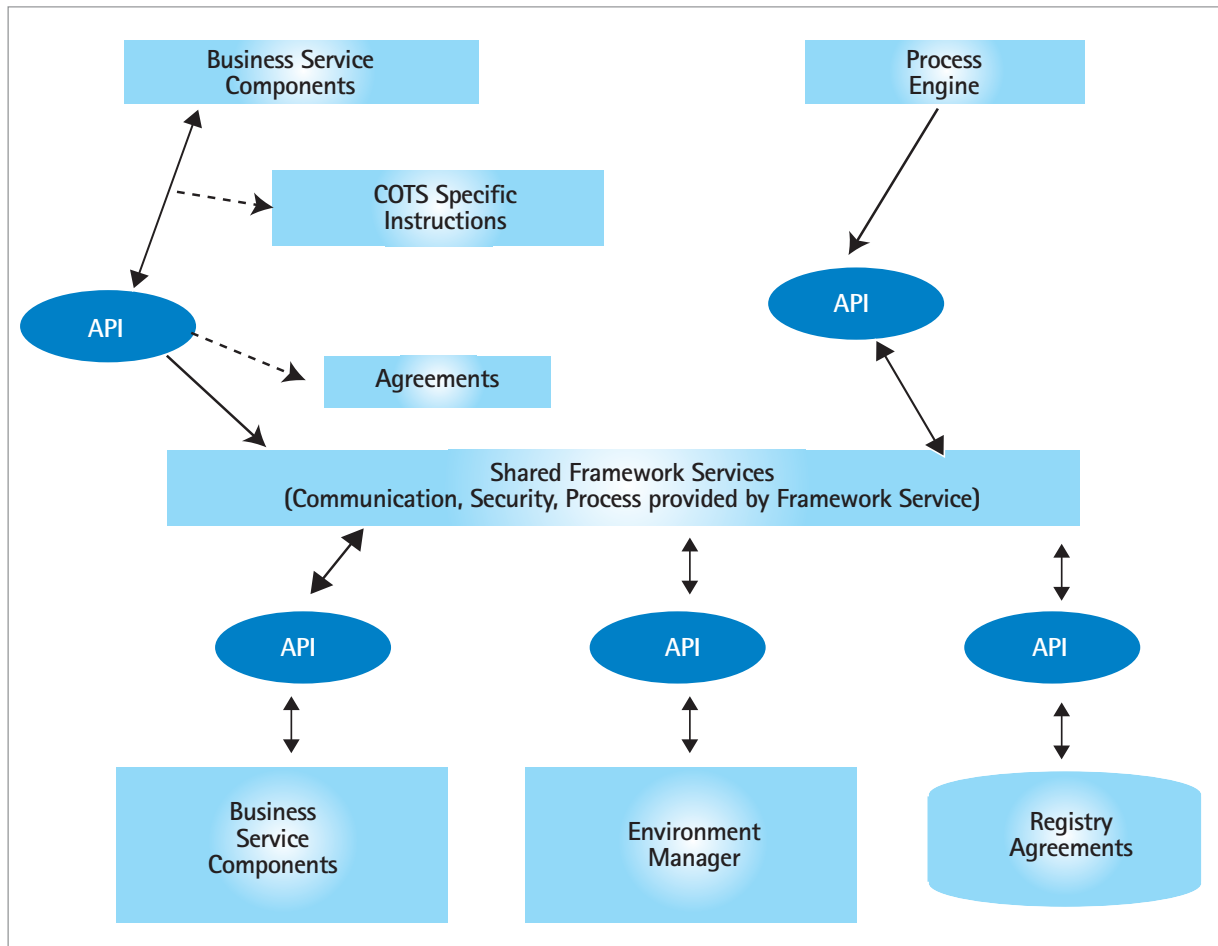


Figure 6 - Revolutionary Integration Architecture

The environment registers the component. It also maintains information about all components and the services they offer. Components that require services can import relevant contracts from the environment. In case more than one provider of the service exists, a component can choose the one best suited to its needs through a trading service provided by the environment.

Normally, the process engine invokes services on BSCs. To avoid tight coupling, a BSC should not directly invoke another BSC. However, it may invoke an FSC.

When a component needs a service, it sends a request to the API, which is responsible for invoking the service provider and returning the results to the requester.

An SDM, accessible by all components, defines objects exchanged between components. The adapter translates between data representations in the SDM and its own private data model or the CDM. This translation is facilitated through map-pings between the SDM and CDM.



The environment is also responsible for enforcing policies (such as security policies) that govern the interactions between components.

Integration Approach	Salient Features	Issues
Point-to-point	Point-to-point	Difficult to maintain
Message oriented	Messaging middleware	No separation of business logic and process logic
Process engine	External process engine	No well-defined interaction semantics
Revolutionary	<ul style="list-style-type: none">• COTS components• Contract-defined Interfaces• Shared data model• Shared framework services• Policy Driven• API support	<ul style="list-style-type: none">• Overcomes previous limitations• Needs to mature solution migration

Table 2 - Summary of different integration approaches



CONCLUSION

This article discussed the issues in service-oriented design and architecture with special emphasis on the telecommunications industry. The approaches were classified into four generations, each representing an increasing level of sophistication. The benefits, architecture, and issues pertaining to revolutionary integration were discussed. Also, it summarizes the features and limitations of the various integration approaches.

REFERENCES

- [1] 'On the notion of variability in software product lines', Proceedings of the IEEE conference on software architecture by J.V Gurf, J.bosch and M. Svalmberg, (WICSA 01), 2001
- [2] 'Design Patterns', by E.Gamma, Rhelm, R.Johnson and J.Vlissides, Pearson Education, 1995
- [3] www.tmforum.org
- [4] <http://java.sun.com/products/oss>
- [5] <http://www.xml.com/>
- [6] <http://java.sun.com/j2ee>

ABOUT THE AUTHOR

Medida Koteswara Rao has eight years of experience in the IT field, out of which more than five years in Telecom domain. He was involved in the development of various projects in the OSS space for various telecommunication operators. He has also



ABOUT WIPRO TECHNOLOGIES

Wipro is the first PCMM Level 5 and SEI CMMi Level 5 certified IT Services Company globally. Wipro provides comprehensive IT solutions and services (including systems integration, IS outsourcing, package implementation, software application development and maintenance) and Research & Development services (hardware and software design, development and implementation) to corporations globally.

Wipro's unique value proposition is further delivered through our pioneering Offshore Outsourcing Model and stringent Quality Processes of SEI and Six Sigma.

For more whitepapers logon to: <http://www.wipro.com/insights>

© Copyright 2004. Wipro Technologies. All rights reserved. No part of this document may be reproduced, stored in a retrieval system, transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without express written permission from Wipro Technologies. Specifications subject to change without notice. All other trademarks mentioned herein are the property of their respective owners. Specifications subject to change without notice.

Worldwide HQ

Wipro Technologies,
Sarjapur Road,
Bangalore-560 035,
India.
Tel: +91-80-844 0011.

U.S.A.

Wipro Technologies
1300 Crittenden Lane,
Mountain View, CA 94043.
Tel: (650) 316 3555.

U.K.

Wipro Technologies
137 Euston Road,
London, NW1 2 AA.
Tel: +44 (20) 7387 0606.

France

Wipro Technologies
91 Rue Du Faubourg,
Saint Honoré, 75008 Paris.
Tel: + 33 (01) 4017 0809.

Germany

Wipro Technologies
Am Wehr 5,
Oberliederbach,
Frankfurt 65835.
Tel: +49 (69) 3005 9408.

Japan

Wipro Technologies
911A, Landmark Tower,
2-1-1 Minatomirai 2-chome,
Nishi-ku, Yokohama 220 8109.
Tel: +81 (04) 5650 3950.
Tel: +97 (14) 3913480.

U.A.E.

Wipro Limited
Office No. 124,
Building 1, First Floor,
Dubai Internet City,
P.O. Box 500119, Dubai.

www.wipro.com
eMail: info@wipro.com