Continuous Assimilation Policy for Service Component Architecture

Deepali Virkar
Computer Engineering Department,
Pune Institute of Computer Technology
Pune, Maharashtra, India

Pravin Game
Computer Engineering Department,
Pune Institute of Computer Technology
Pune, Maharashtra, India

ABSTRACT
Service Component Architecture (SCA) provides a programming model to support Service Oriented Architecture (SOA). SCA based application has long product development life-cycle. Slight change in one service component may affect functionality of other component. This leads to requirement of continuous checking for stability of integrated systems. If defects are identified in earlier stage and total time required for product development get reduced then it would certainly improve performance. In this paper, we introduce continuous assimilation policy for service component architecture, which gives continuous and rapid development of service components. It focuses on implementation strategies for SOA application.

Keywords
Integrated Environment, Service Component, Distributed Applications.

1. INTRODUCTION
Every day, agile software development delivers dozens of components with some dependency on other components. This integration of components must get complete at the end of the day. As CI principle states [1], programmer should never leave anything unintegrated at the end of the day. Built components must go to testing phase as early as possible to identify issues. As early they are identified, they can be fixed easily. While following Service Component Architecture (SCA), intermediate files are generated which has integration role in product life cycle. If these files-which can be jar, war, tar or archive files-are generated as a part of integration process then it will boost the software development process. Our Continuous Assimilation Policy adds this value to agile software development process for SCA.

In SCA approach of SOA application development, in assembly model, developer defines how components are combined, linked and packaged together. In this stage, some issues may stay undisclosed until developed work gets deployed. To identify such issues in advanced, Continuous Assimilation Policy can help to identify it in earlier stage. While developing code using SCA, developed components are checked in to source control system. CIP policy takes data from source control system, and runs the integration process to create intermediate files. If these files are created then they are deployed on server for further testing. By following this policy, major issues can be identified while generating intermediate files and while deployment.

Unified test framework for continuous integration testing of SOA solutions is proposed by H. Liu , et. al. This [4] framework uses surrogate engine and test case execution engine for continuous integration. Surrogate is proposed by H.Y. Huang, et. al [5] added value to continuous integration testing as it creates surrogate components to test SCA component when they are partially implemented.

Backtracking incremental continuous integration [6] is useful when current build fails. Backtracking makes sure to have working version of any application.

Most of the work in continuous integration approach gives a generic framework to carry out integration testing. There are various open source integration servers available such as Cruise Control, Jenkins. One can use such servers to perform integration testing. There is need to customize this continuous integration testing cycle to improve the development of components. Continuous Assimilation policy gives this customization for SCA artifacts.

The rest of the paper is organized as following: the next section gives brief idea about SCA architecture. Then it gives an introduction to prior arts in the field of continuous integration. After that it explains Continuous Assimilation Policy. The last section concludes this paper and points out some future research directions.

2. SERVICE COMPONENT ARCHITECTURE
Service Component Architecture provides a way to create components and a mechanism for describing how service components work together [2]. The SCA specifications define how to create components and how to combine those components into complete applications. The components in SCA are a building block, which provides one or more services to requestor. It possesses configuration properties and references, which provides dependencies between different components within composite or between two composites. One or more components are combined together to form a Composite.

Composite logically contains components, services, references, the wires that interconnect them and properties that are used to configure the components. Wire is a theoretical illustration of relationship between reference and service. These wires are published to outside world by promotion.

SCA can be broken down into four major parts or models [3]:

A. The Assembly Model, which defines how components are combined, linked and packaged as service independent of the programming language.

B. The implementation model, which defines how services are packages and accessed for specific programming languages.

C. The policy model, which defines the service policies independent of programming code.

D. The bindings model, which defines how components are accessed independent of the programming code.

3. SYSTEM ARCHITECTURE

As discussed in 2nd section of our paper, SCA can be divided in four parts.

Assembly model promises management of services. This includes application configuration by setting properties, logging configuration, and substitution variables. Also, one can add binding to services; promote services and references to the environment, and wire services and references to services and references in other application environments.

Implementation part deals with service composition. In this model, composites are created depending on nature of application. An application may have one or more composites. The output of this stage can be a deployable war, jar, ear or archive file which consists of set of related configuration files: nested composites, resource templates, WSDL files, and substitution variable files.

Policy model and binding model focuses on runtime monitoring and service selection. The main concern of this part is distribution of application to runtime infrastructure that focuses on application execution. One can explicitly distribute application fragments-components and bindings- to one or more node.

As requirement changes or issue arises, SCA artifacts should update to accommodate changes. Application may be composed of dozens of components with rigid dependency with each other where one component change affects other component behavior. When application accumulates all changes, it is needed to test entire application. This is time consuming activity. This may lead to frequent and small releases.

Figure 1 explains system architecture. Continuous Assimilation policy considers that system consists of nodes, service groups, builds, state and integration of builds. In system environment, nodes can have one or more nodes that essentially run services.

4. CONTINUOUS ASSIMILATION POLICY

Continuous Assimilation Policy is based on checking the build health on every check-in command. Source code commit should not create unsteady environment.
System is defined as Sys={C, Pp, Pr, S, Q, IT, BT, J, A, N | parse(composite), assimilate() }

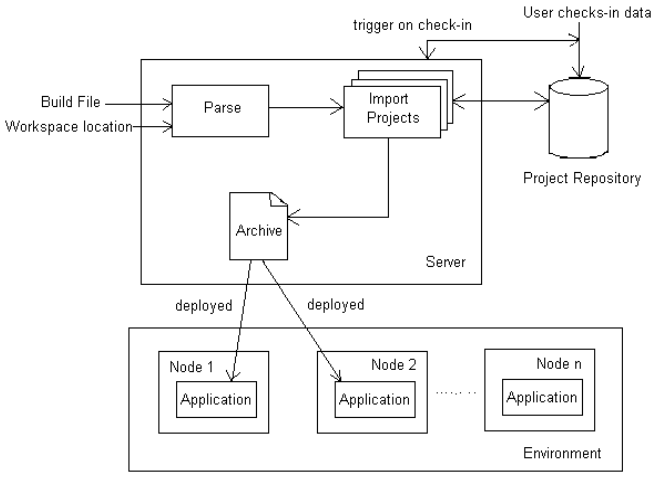


Figure 1. System Architecture

To build a composite, one or more components are required. So composite is defined as,

Composite C = {Q1, Q2, …QN, Pp, Pr} (1)

Where Q is component = <Pp, Pr, S>
Pp is provided port of a composite which is promoted service while Pr is required port of composite which is promoted reference.
In component definition, Pp is provided port, Pr is required port, S is service implemented by that component [9]. Composite is detailed explained as,

$C \subseteq$ Q X IT X BT (2)

IT is Implementation Type of a component. Component can be implemented using any programming language construct. There is one-to-many mapping between component Q and IT. BT is Binding Type which also has one-to-many mapping with component Q.

$f1 : Q \rightarrow IT$ (3)
And $f2 : Q \rightarrow BT$ (4)
where $IT=\{IT_1, IT_2, \ldots..., IT_N\}$
$BT = \{BT_1, BT_2, \ldots , BT_N\}$
Figure 2 shows one-to-many mapping between component and IT.

In Continuous Assimilation Policy, we are building jobs which runs build tool to create archive file.

Job $J = \{j_1, j_2, j_3, \ldots j_N\}$ (5)

These jobs may be running on different environment {env1, env2 , … envn}. Job depends on environment setup.

Build tool continuously look for update of any composite or new check in. Once check is perform, job is triggered which creates archive file and deploys it on predefined node. From this the health of application is identified. At no point in time, any unstable application may be running on any node.

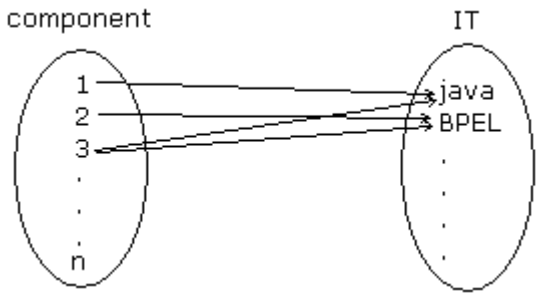


Figure 2. One-to-one mapping between component and IT

For building an archive identifying its implementation type, binding type and WSDL configuration must be checked.

$$\text{Parse (C)} = \exists\ Q \equiv q \wedge [IT]^{+} \wedge [BT]^{+} \quad (6)$$

Since we are going to take in one more components, assimilation of these components is defined as,

$$\text{Assimilate()} = \{\Pi_C \mid c \subseteq IT\ X\ BT\ X\ S\} \quad (7)$$

$$\text{Output of Assimilate() function is } \{q\ X\ q\} \quad (8)$$

Job is partitioned into two subsets {Success, Failure}. Success and failure can be determined from its execution. If executed job gives a stable environment then we say job is executed successfully. Stability of job is defined as when archive is deployed on node N, it does not lead to unsteady Env *t* .

$$\text{Job} = \{\ \Pi_{\text{Archive}} \mid \langle\langle Pp,Pr,S\rangle, BT, IT\rangle \in \text{Success}\} \quad (9)$$

The set Env *t* , belongs to number of available nodes in given environment at time t. Nodes have running application.

$$\text{Env } t = \{\ N_1,N_2,\ldots N_N\ \} \quad (10)$$

where Nx = number of nodes available in given environment at time t.

For current Env *t* ,

$$\text{Node}_N=\{A_1,A_2,\ldots A_N\} \wedge \{S_1, S_2,\ldots S_N\},\ 1<n\leq M \quad (11)$$

where { A1,A2,…AN} is a finite set of applications in given environment at time t and {S1, S2,…SN} is set of services which is operation or group of operations.

Env *t* gives idea about service provided by each node. For Continuous Assimilation Policy, it is assumed that dependency is a part of component sources; a change in dependency causes change in component [6]. It is also assumed that every build of a component has implicit dependency on implementation type and binding type.

5. DISSCUSSION

Implemented Continuous Assimilation Policy is capable of creating and deploying archive file to server along with support for version control system. Also it is capable for checking the extent to which changes made to policy affects implementation of services. Experiments are carried out to test the performance of implementation time for SOA application. The procedure followed is:

Table 1. Currently followed procedure Vs. Steps followed for Continuous Assimilation Policy

Step No.	Steps followed in traditional approach	Steps followed in Continuous Assimilation Policy
1	Identify SOA project and composite to build deployable archive file	Identify SOA project and composite to build deployable archive file
2	Read feature file and identify dependent projects	Set path for technologies used to execute scripts(eg. Set path for Ant and Maven) Perform this step for first time only.
3	Check-out SOA project, dependent projects to local machine from source code repository	Create new job and enter source code repository URL
4	Create script having tasks a. to build SOA and dependent projects and b. to create deployable archive file	Enter details required to create deployable archive a. SOA project name b. composite name Now save configuration and execute job to create and upload deployable archive file
5	Create deployable archive file	
6	Check-in deployable archive file to source code repository	
7	On requisition for service, check out deployable archive file from source code repository to local machine	
8	Create script to upload deployable archive file to server	
9	Upload deployable archive file to server	

In table 1, steps 2,3,4,6,7,8 in traditional approach requires user activity. Like checking in and checking out data from source code repository. Time required to execute these steps varies with level of expertise. Total time required to perform these steps is (considered as user Activity time)150 sec. In addition as number of dependable projects varies, time

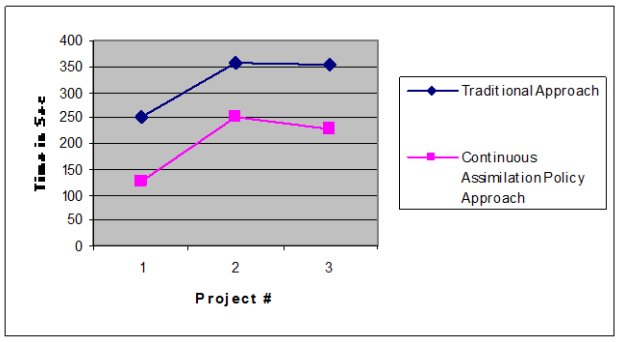
required creating scripts-for archive file and for uploading it to server- varies.
Our performance measurement test considers three projects with different complexity. Project complexity Project complication is decided upon number of components.

Table 2. Project Details

Project #	# of Components	# of ITs	# of BTs
1	1	1	1
2	10	10	15
3	100	100	1

Table 3. Measurement values for two approaches

Project #	Time required using traditional approach (in sec)			Time required using Continuous Assimilation Policy approach (in sec)
	a. Create deployable archive file	b. Upload archive file	Total Time (a+b+ userActivity)	
1	83	18	251	126
2	168	37	355	250
3	173	30	353	226



Graph 1. Performance Graph

From graph it is very clear that time required by Continuous Assimilation Policy for-building SOA project, creating deployable archive file and uploading it to server-is less than time required to perform same tasks using traditional approach.

6. CASE STUDIES

We came across many real time scenarios, which motivated implementing Continuous Assimilation Policy. We are considering three scenarios in this paper.

Issue 1: Building a deployable archive in later stage makes it difficult to isolate the problem.

Consider the first scenario of an application wherein multiple composites, say 10, exists having multiple components in it. Take example of online payment example, which has two of its components as Invoice and Payment. Ideally,

Invoice=<$P_{p\text{-}invoice}$, $P_{p\text{-}question}$, $P_{p\text{-}update}$, $P_{r\text{-}invoice}$, $P_{r\text{-}paymentMode,}$ S>

Payment=<$P_{p\text{-}invoice}$, $P_{p\text{-}paymentMode,}$ $P_{r\text{-}paycheck}$, $P_{r\text{-}delivered}$, S>

A distributed team implements different component. A developer implementing Invoice component does not care about whether the implemented service is going to be used by other component or not. If developed component is going to be used by other component then how it will be used is not considered. Consider, while implementing a component, Payment component is implemented as

Payment = <$P_{p\text{-}invoice}$, $P_{r\text{-}paycheck}$, $P_{r\text{-}delivered}$, S>

While Invoice component is implemented as

Invoice= <$P_{p\text{-}invoice}$, $P_{p\text{-}question}$, $P_{p\text{-}update}$, $P_{r\text{-}invoice}$, $P_{r\text{-}paymentMode,}$ S>

From its implementation, it is very clear that Invoice component is using payment-mode service of Payment component but it is not implemented in Payment component. Now if deployable archive file of an application is created after implementing all 10 composites then it will throw an error and will not create archive file. As number of implementation type and components increases, locating a culprit component becomes time consuming job.

Solution: To avoid such situations, once component is implemented, intermediate archive file should be created to check whether it is implemented as per requirement or not and its dependencies are resolved or not. There should be some policy that will take care of it.

Issue 2: Building a deployable archive from latest executable version is tedious job as lot of change in configuration is required.

Executable is used to create deployable archive file of an application. Let us consider EXE_1 is used for creating first application archive. This executable file is updated to support new runtime environment. When executable is updated, ideally, new version should be use to create deployable archive file. If developer, responsible for creating archive file uses old executable then such case causes issues in production stage such as class not found exception.

Solution: To avoid such situations, it is needed to create archive file using updated executable. Also for backward compatibility testing, archive file creation must be checked with old version as well. For this, some hook up should be there to select which executable version to use without changing other configurations like creating new job for new configuration.

Issue 3: Small change or modification in project cause large change in job configuration.

While implementing Proof-Of-Concept projects, not all situations are taken into consideration. Some real world scenarios may be overlooked like promoting paycheck mode service in our online payment application example. When POCs are deployed on customer end, modifications or updating is required to be done to improve quality of a product as well as to integrate new functionality. Now once these modifications to component and/or composite definitions are done, it is needed to create new archive file. For this, one need to identify which implementation type, binding type is used, which executable should be used, which job to trigger?

Solution: If there is some pre-configured job, which creates archive file by considering all the aspects of archive file generation then with one check-in command pre-configured job will create required archive file.

We have implemented a Continuous Assimilation Policy that takes care of all such above discussed scenarios and many more.

7. CONCLUSION AND FUTURE WORK

Our Continuous Assimilation Policy makes finding and fixing problems in SOA application easier. Traditional approach, which does not follow Continuous Assimilation Policy, takes longer time to create deployable archive file and upload it to server than approach using Continuous Assimilation Policy. This increases application implementation performance. Our future work includes policy customization.

8. ACKNOWLEDGMENTS

We thank to Dr. Sarang Joshi, Santosh Kumar, Wojciech Zurek and Sunil Pawar for their guidance and support.

9. REFERENCES

[1] http://martinfowler.com/articles/continuousIntegration.html accessed on Aug2011 accessed on Aug 2011

[2] David Chapell, Introducing SCA, DavidChappell & Associates, July 2007.

[3] http://www.osoa.org/display/Main/Service+Component+Architecture+Home accessed on Aug2011

[4] Hehui Liu, Zhongjie Li, Jun Zhu, Huafang Tan, Heyuan Huang, "A Unified Test Framework for Continuous Integration Testing of SOA solutions", ICWS 2009, IEEE International Conference on Web Services, pp. 880-887, Jul 2009

[5] He Yuan Huang, He Hui Liu, Zhong Jie Li, Jun Zhu, "Surrogate: A Simulation Apparatus for Continuous Integration Testing in Service Oriented Architecture", SCC'08, IEEE International Conference on Service Computing 2008, pp. 223-230, Jul 2008

[6] Tijs van der Storm, "Backtracking Incremental Continuous Integration", CSMR 2008, 12th European Conference on Software Maintenance and Reengineering, pp. 233-242, Apr2008

[7] Francisco Curbera, "Component Contracts in Service-Oriented Architecture", IEEE Computer, pp. 74-80, Nov 2007

[8] Aliaksei Yanchuk, Alexander Ivanyukovich, Maurizio Marchese, "Towards a Mathematical Foundation for Service-Oriented Applications Design", ICSOC 2005, pp. 545-551, 2005

[9] Dehui Du, Jing Liu, Honghua Cao, "A Rigorous Model of Contract-based Service Component Architecture", 2008 International Conference on Computer Science and Software Engineering, pp. 409-412, Dec2008.