SATE- Service Boundary and Abstraction Threshold Estimation for Efficient Services Design

Kalapriya Kannan, Gandhi Sivakumar and Nanjangud C. Narendra

Abstract—Service Oriented Architecture (SOA) provides a flexible set of design principles with the potential to significantly enhance software reusability. The key to successful adoption of SOA is not only reusability but also the ease with which services can be developed, deployed and maintained in a service providing environment. One key guiding principles of SOA is implementation abstraction, that fosters complete encapsulation of implementations, exposing only the primitives as interaction points with the service. Granularity of service abstractions defines the different level of primitives exposed to prospective service consumers. The primitives being the only external representation for a service, their design is extremely important as the granularity not only determines the ease with which the services are identified, but also the ease with which services realization can be achieved. So far there has been no systematic approach to apply design principles such that right level of abstractions can be defined.

To that end, in this paper, we present a bottom-up approach that derives a feasible set of service abstractions and primitives to enable reusable service design. We present a tool called Service Abstraction Threshold Estimator (SATE). Given the available implementation artifacts, SATE allows varying levels of primitives to be defined, elicits the complexity associated with different levels of primitives and provides a threshold estimation engine that displays the optimal level of abstraction that can be achieved in a given service development environment. We demonstrate SATE on a real-world example, and also evaluate SATE against design decisions taken by software architects. The results of our evaluation show that SATE can provide more accurate threshold estimations, and also helps in reducing the time and effort taken to maintain already developed services.

I. INTRODUCTION

Service-oriented Architecture (SOA) [6] has emerged as the next major software architectural style due to its focus on modeling and developing software components as reusable services. Reusability in SOA is achieved by defining abstractions that encapsulate key business logic and concepts whose realization will achieve the desired functionality. One of the important guiding principles to obtain reusable services in SOA is the implementation abstraction, wherein services hide their implementation details completely and expose only primitives to their prospective consumers. Service primitives define service interfaces at various abstraction levels, and are therefore a key component for determining reusability. Typically, higher level abstractions are highly reusable as they reduce the dependency required from service consumers. However, higher abstraction levels come with significant costs for realizing and maintaining services. Therefore designing service primitives that encapsulate the correct level of abstraction is an important criteria for successful adoption of SOA. In this work we argue that for efficient services design, it is imperative to provide equivalent weightage to both reusability and realization factors.

Obtaining optimal threshold levels for service abstractions in a service development environment raises several research issues. First, while SOA defines the principles required for making abstractions reusable, SOA does not specify methods to apply these principles to services design to ensure that the resulting services adhere to these principles. Therefore these principles are open to interpretation by software architects. So far, (subjective) human expertise remains the only design guide for defining services abstractions. This results in varying levels of primitives encapsulating different levels of abstractions, depending on the skill and judgment of the designer. Second, anticipated variations in a given software requirement are considered as main criteria for determining the abstractions. This is primarily due to the fact that these variations are potential candidates for changes in future and good services design should allow these variations to be handled with ease. As this approach provides precedence to reusability, it overlooks the issues associated with development and maintenance of services in a service development environment. Third, with reusability in focus services design targets high level of abstractions. Higher abstractions are highly re-usable due to high separation of concerns while less granular services tend to be less reusable. But these high level abstractions come with significant cost in terms of time, skills and effort to realize them. Higher the abstraction, higher is the level of expertise (skills), process and tool maturity required to realize them. We use the term implementation artifacts to refer to all the factors that affect the feasibility of service realization and maintenance in a given service development environment. We believe that if the software architects are provided with a mechanism that elicits the available implementation artifacts in a given service development environment it will significantly increase the quality of the abstractions achieved.

To that end, in this paper we focus on investigating various aspects of service design from both reusability and realization perspectives. We define methods to determine optimal thresholds of abstraction that will maximize reusability during service design but at the same time ensure abstractions are maintainable. To the best of our knowledge, ours is the first work of its kind to formally and rigorously define a method for determining appropriate abstraction levels that are beneficial to both service requesting and providing environments. With this in mind, we provide a conceptual instrument called Services Abstraction Threshold Estimator (SATE). SATE provides a platform that elicits the different implementation artifacts available in the service development environment, allows varying levels of primitives (based on the variation points) to be defined, elicits the complexity associated with the different level of primitives/abstractions and provides a threshold estimation engine that displays the optimal level of abstraction that can be achieved in a given service development environment. We identify various im-
implemenation artifacts and aspects of the primitives to derive optimal threshold levels. We explore the impact of these implementation artifacts on the services design and propose hypotheses to decide the choice of the primitives thus trading off between reuse and implementation efforts. The real world experiments described in our paper demonstrate the efficacy of SATE.

In summary, we claim the following novel contributions in our paper:

1) Elicitation of the implementation artifacts in a service development environment that affects the services design
   a) Defines the set of implementation artifacts that affect realization of service in a service development environment
   b) Metrics to collectively define different levels of implementation artifacts.

2) A conceptual instrument called SATE that achieves optimal threshold levels of abstractions.
   a) Provides an optimal threshold engine that arrives as a threshold for different services for a software requirement.
   b) Mapping between the levels of implementation artifacts to services primitives/abstractions.

Our paper is organized as follows. In Section II we provide the background, motivation and compare our work against existing literature. The architecture and details of SATE are provided in Section III. The core optimal threshold engine algorithm is presented in Section IV. Section V presents the evaluation of our work, and we finally conclude our paper in Section VI with suggestions for future work.

II. BACKGROUND

The emerging SOA paradigm believes in the implementation abstraction. A service provider develops a piece of code (termed as implementation) and exposes this as a service to external/internal consumers. Logically the service comprises two strata: the service implementation (lower stratum), and the service contract (upper stratum). Service contract is seen by external consumers as well as the service provider and is fortified by the three bands of primitives discussed below.

- Strong primitives: This genre of primitives when reinforced in the service contract ensure total abstraction of the implementation but increase the requirements required in the implementation layer to the largest extent.
- Weak primitives: This genre of primitives ensures the lowest level of abstraction but decrease requirements on the implementation layer.
- Intermediate primitives: This genre of primitives is optimal and ensures intermediate level of abstraction.

During the design phase the designer ensures this abstraction by reinforcing various types of primitives in the service contract. Services design can lead to multiple number of primitives with different combinations. In reality in many cases this leads to the fact of ensuring high level of abstraction with higher level of concept encapsulation but impedes smoother construction of the implementation.

The lower stratum represents the Service implementation which is the actual code. For successful development of SOA solutions and further maintenance, Implementation complexity in this layer is a crucial entity. Different primitives require different implementation artifacts for completion. Abstraction level increases with increasing primitive levels. This increased abstraction effectively increases the service’s capability to perform higher level of functionality as opposed to those designed as weak or intermediate primitives.

A. Motivation- Real World Example

We provide a snapshot of the practices of real world SOA design stages and bring out the issues associated through an example. In the service specification phase the designer normally does the following. First, he/she defines the service interface (operations and data elements); then he/she identifies the service functionality, then the service dependencies and finally, the service bindings to implementations.

At every stage he/she iterates and refines the service interface design. Initially the designer designs the service interface with elements to cater to the current required business functionality. We term such an evolved service as “Basic service”, which possesses the following characteristics: it caters to the current business functionality only, and possesses lower abstraction and hence a lower ability to absorb future changes. Subsequent to this is when the designer reinforces the service interface with various combinations of primitives. The methodology adopted normally to select the primitives is through “Variation Analysis”. We term such an evolved service as “Premium service”. Premium service possesses the following characteristics: flexible enough to absorb changes, and possessing high degree of abstraction to promote reuse.

For ease of exposition we consider a simple commonly used Service termed as AddressManagement encompassing only one operation termed as Retrieve Address. This service currently will perform mere retrieval of matching addresses given an address identifier. Figure 1 provides the basic design of the service response that allows to retrieve a single matching address.

Simple variation analysis elicits some of the foreseen changes to the service, as depicted in Figure 2.

Fig. 1. Service response that satisfies the requirement to retrieve a matching address record

With different level of variation analysis performed by different architects different levels of variation can be achieved. It should be noted that Basic Service is less susceptible for
change due to its strictness for manifesting current requirements as-is while Primitive Service can easily accommodate future changes. Basic service is less abstract as it exposes the requirements clearly to return an address record, while premium service abstracts the information about the purpose of the service. This introduces significant impact on the development of the services. Our aim in this paper is to analyze the various factors that affect implementability and provide the optimal level of abstraction such that the services are not only reusable but also easy to develop and maintain.

B. Related work

Services design work is an ongoing area of intense research [9][10][3]. The importance of service orientation principles has been brought out in several of these works. Those works were followed by different extensions [1], [4], [11] in building conceptual tools and models that enforce service orientation during services design. The main focus of these works has been on designing services keeping the main focus on reusability. In [1] the focus has been primarily on building reusable services. In [11] the focus has been on service quality using service orientation principles. In this work we focus on addressing services design with focus on ease of realization of service in addition to reusability.

The importance of easing development and maintenance of services has been emphasized in [14] [7]. [14] brings out the fact that services designed should be easy to realize. The citation [7] shows the importance of role of maintenance of services in SOA. In this work we explore the different implementation artifacts that affect the development and maintenance of services and their effect on the design of services abstractions.

The citation [5] insists on the importance of reusability and proposes a methodology to identify highly reusable services. It specifies the metamodel of reusable services and reiterates the concepts of atomic and composite services clearly. The method in [3] comprises activities including identification of business processes, specifying business processes, identifying reusable services, and specifying and designing the reusable services. The method specifies an approach to identifying reusable services but does not explain how reusability can be ensured in the service specification phase. We insist on ensuring reusability at service interface level which is equally important because the service comprises of two strata: interface and implementation. Thus a service which qualifies to become reusable from functional perspective alone is incomplete from reusability perspective. This means service interface needs to be processed for reusability as well which can be achieved through abstraction and this in turn impacts implementation which is the focus of our paper.

The citation [6] specifies the key concepts behind service-oriented computing including abstractions. It discusses the complexities associated with intraenterprise, interenterprise, infrastructure and software components and how SOA can help reap the benefits achieved through abstraction. For example they quote a real world example from the health industry and explain the challenges including connectivity among applications and ability of the components to understand each other, and how SOA can ensure implementation abstraction to overcome the challenges when analysing intra-enterprise service ecosystems. However it does not explain the next deeper level of “abstraction”, i.e., how abstraction can be achieved and the impact of abstraction.

The citation [2] discusses the feature-oriented approach to service oriented re-engineering for non-SOA based systems. It clearly specifies the criteria for a functionality of a system to be eligible to become a feature; however it does not define the abstractions to these features when exposed as services.

In [12] the focus has been on service quality using service orientation principles while we focus on addressing services design with focus on ease of realization of service in addition to reusability. The importance of easing development and maintenance of services has been emphasized in [8], [15]. The citation [15] brings out the fact that services designed should be easy to realize. The citation [8] shows the importance of the role of maintenance of services in SOA but does not address the key aspect of trade-off between abstraction and implementation. In this paper we explore the different implementation artifacts that affect the development and maintenance of services and their effect on the design of services abstractions.

III. SATE APPROACH AND ARCHITECTURE

In our work, we proceed to identify those implementation artifacts that limits the development of SOA solutions. Once the factors are identified, complexity involved in developing the abstractions using these implementation artifacts is determined. In a given development environment, the contributing implementation factors are quantitatively measured and utilized for setting threshold levels for the abstractions. Figure 3 presents the system architecture of SATE that describes the different components required for obtaining abstraction threshold.

Functional or non-functional requirements from the consumers are utilized for deriving Service(s), each Service encapsulating certain functionality required for satisfying the end-to-end business requirements. The services are just notional abstractions at this stage. The services are then subjected to variation analysis where foreseen changes and requirements are considered for designing services interfaces. The foreseen changes become the requirements for designing service primitives/interfaces. Depending on the flexibility required to allow future changes to be accommodated the service interfaces are designed with generality or abstractions. It should be noted that at the design phase, although
SOA provides guiding principles it does not provide a mechanism which when applied with make services abstract and reusable. Therefore abstraction is always a relative measure. A service that is being designed can assume different levels of service abstractions (through different service primitives) depending on the context but is largely determined by the expertise of the architect. Therefore our objective function is provided in Equation 1.

\[
\begin{align*}
\text{Min} & \quad I \\
\text{s.t.} & \quad R > \lambda x \\
\text{g.t.} & \quad I < \frac{1}{R}
\end{align*}
\]  

(1)

The input to SATE is the service primitives derived by solving Equation 1 (Step 1 in Figure 3). Each service is also tagged with indicators that can be redesigned for higher or lower abstractions. Different implementation artifacts such as skill set, resources, tools etc., are utilized to measure availability of resources in a service providing environment. The collective effect of the several implementation artifacts is converted to a quantitative measure called the Impediment Factor (IF). This is shown in Step 2 in Figure 3. The IF provides qualifiers for the resources available in the service providing environment for realizing varying levels of abstractions.

The core part of the system is the SATE engine that provides a mechanism to set threshold levels while solving Equation 1. A combinatorial algorithm assigns through an iterative process the right set of primitives that can be selected for a given service providing environment. The engine consists of two important components - primitive threshold planner and primitive combination planner. There is also an ancillary component called primitive level changer. The primitive threshold planner considers the current IF and sets thresholds for different primitives available. It uses the primitive combination planner to check whether the combination of primitives can be assigned to skills. In case the current set of primitives are not achievable the primitive level changers changes the primitive levels of the services to either higher or lower abstractions and returns back the set to the SATE engine which re-runs the algorithm for determining the desirable set of primitives.

A. Identifying Implementation Artifacts

Cross-sectional analysis of about 20 recorded cases of major SOA projects was performed in order to assess the significant artifacts that influence the planning and implementation of services. Data gathered from the case descriptions were then used for determining the variables that characterize implementation processes. Table I provides the different implementation artifacts derived by qualitative analysis performed on the cases. Table II provides the importance scoring of these metrics for the successful development of the SOA projects as provided by software architects of these projects. Significance of these metrics were further confirmed in the research literature as shown in Table II.

Due to the large number of variables reported we utilize factor analysis to remove correlated variables. Let \( U_1..U_n \) be the unique variables that contribute to the factors.

Our aim is to determine the factors \( U_1..U_n \) and the associated weights that each variable has on the factors. In order to achieve this we obtain the data related to relevance scoring of these metrics from the individual architects responsible for end-to-end project management. Relevance scoring measures the importance of these metrics in the project for successful development and maintenance, on a scale of 1 to 10. After applying factor analysis, we narrow down to four significant factors, as depicted in Table III. From Table III it can be shown that factor 1 is influenced by primarily Service Engineering (abstraction, competency, Service engineering), factor 2 is influenced by document, business strategy, resource management, schedule management, project management, selection of tools which are typically processes and Factor 3 is based on the amount of training, common architecture which are typically related to the tools utilized for the development of the projects.

The factors can thus be derived as three, viz., skill factor, processes and tools. These are the three major factors for affecting the implementation of services. Equation 2 provides the mathematical formulation of the IF (Impediment factor).

\[
IF = W_s S + W_p P + W_t T
\]  

(2)

where \( W_s \) refers to the loading of Skill, \( W_p \) refers to the loading of Processes and \( W_t \) refers to the loading of tools on IF. \( S \) refers to a quantitative measure of the skill levels, \( P \) refers to a quantitative measure of process maturity and \( T \) refers to the quantitative measure of tool sophistication.
**TABLE II**

RELEVANCE SCORING OF IMPLEMENTATION ARTIFACTS BY SOFTWARE ARCHITECTS

<table>
<thead>
<tr>
<th>Variable</th>
<th>Factor 1</th>
<th>Factor 2</th>
<th>Factor 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>pi</td>
<td>-0.527</td>
<td>-0.110</td>
<td>-0.490</td>
</tr>
<tr>
<td>competency</td>
<td>0.164</td>
<td>-0.940</td>
<td>-0.054</td>
</tr>
<tr>
<td>domain knowledge</td>
<td>-0.597</td>
<td>-0.768</td>
<td>-0.015</td>
</tr>
<tr>
<td>Service Engg</td>
<td>0.367</td>
<td>0.023</td>
<td>0.863</td>
</tr>
<tr>
<td>Skills</td>
<td>0.026</td>
<td>0.832</td>
<td>0.200</td>
</tr>
<tr>
<td>experience</td>
<td>-0.322</td>
<td>0.325</td>
<td>0.627</td>
</tr>
<tr>
<td>business knowledge</td>
<td>-0.753</td>
<td>-0.205</td>
<td>-0.278</td>
</tr>
<tr>
<td>communication</td>
<td>0.841</td>
<td>0.281</td>
<td>-0.142</td>
</tr>
<tr>
<td>documentation</td>
<td>-0.107</td>
<td>0.081</td>
<td>0.648</td>
</tr>
<tr>
<td>business strategy</td>
<td>-0.915</td>
<td>0.326</td>
<td>0.224</td>
</tr>
<tr>
<td>resource management</td>
<td>7.9062</td>
<td>0.098</td>
<td>0.080</td>
</tr>
<tr>
<td>schedule management</td>
<td>0.901</td>
<td>0.317</td>
<td>0.261</td>
</tr>
<tr>
<td>project management</td>
<td>-0.770</td>
<td>0.464</td>
<td>0.374</td>
</tr>
<tr>
<td>selection of tools</td>
<td>0.357</td>
<td>0.041</td>
<td>-0.802</td>
</tr>
<tr>
<td>training amount</td>
<td>0.240</td>
<td>-0.861</td>
<td>0.320</td>
</tr>
<tr>
<td>change management</td>
<td>-0.983</td>
<td>-0.113</td>
<td>0.033</td>
</tr>
<tr>
<td>reusable factors</td>
<td>-0.546</td>
<td>-0.763</td>
<td>-0.177</td>
</tr>
<tr>
<td>common architecture</td>
<td>-0.983</td>
<td>-0.113</td>
<td>0.033</td>
</tr>
</tbody>
</table>

**TABLE III**

VARIOUS FACTOR LOADING

B. Abstraction Levels and Mapping to Complexity

In SOA abstractions are achieved either at the level of interfaces, attributes on the objects, objects or relationships. For instance Table IV provides a description of the WSDL of service "ConvertAmount". The service operation "ConvertAmount" is the interface through which the service is requested. The arguments "fromCurrency", "toCurrency" and "amount" are the variables defined for the abstraction "ConvertAmount". This service provides specific functionality and the nature of the arguments are very specific to the functionality "ConvertAmount". Let us analyze the level of abstraction that this service has. To begin with, supposing we would like to extend this Service to accommodate a more sophisticated functionality called "ConvertMetric" that can take any metric, from-value and to-value; then the exiting service operation (convert amount) and the variables have to be changed to accommodate this functionality. A better generic service design would have utilized mechanisms such as having additional variables for later utilization. For instance introducing an additional variable called "misc" would have allowed us to use the existing service by utilizing the "misc" variable for storing more information. A more sophisticated mechanism is to use (name, value) pair patterns where the name is indicative of the metrics and the value can be interpretative. This provides complete abstraction wherein any new conversion can be added by just introducing a new name. The major issue is how to measure these service abstractions and categorize them into different levels.

We consider three levels of Service abstraction - Basic, Moderate and High. We have considered three levels for sake of simplicity but the same method can be extended to add more levels. Our aim is to categorize the services that are designed into one of these levels. This can be done by performing a high level analysis of whether the designed service has the potential to accommodate changes. Table V presents the different levels (ranging from specific to generic) of artifacts utilized by services.

From Table V the “Abstraction Factor (AF)” of a service is calculated by Equation 3.

\[ AF = \frac{W_A H}{W_B B + W_M M} \]  

(3)

and

\[ B = W_O N_{op} + W_r N_{ro} + W_{ob} N_{rob} + W_e N_{ro} \]
\[ M = W_O N_{op} + W_r N_{ro} + W_{ob} N_{rob} + W_e N_{ro} \]
\[ H = W_O N_{op} + W_r N_{ro} + W_{ob} N_{rob} + W_e N_{ro} \]
where $W_h, W_b, W_m$ are weighting functions for High abstractions $H$, Moderate abstraction $M$ and Basic Abstractions $B$. $B, H, M$ are calculated as shown above. $W_h, W_b, W_m$ are weighting functions for operations, relations, objects and elements. $N_{hop}, N_{mop}, N_{bop}$ refer to the number of basic, moderate and high abstraction operations respectively, in the service. $N_{hr}, N_{mr}, N_{hr}$ are number of relationships with basic, moderate and high abstraction respectively. $N_{bob}, N_{mob}, N_{hab}$ are number of objects representing basic, moderate and high abstraction, respectively, and similarly $N_{or}, N_{mr}$ and $N_{hr}$ are number of elements representing basic, moderate and high abstractions respectively. $N_{op}, N_{r}, N_{ob}$ and $N_{e}$ are total number of operations, relationship, objects and elements in a service. $B, H$ and $M$ can take a value from $[0-1]$ and the weights similarly can take values from $[0-1]$. Depending on the values of $B, H$ and $M$ AF factor gives a value in the range $[1-0]$ (Equation 3). The resultant metrics can be further subjected to analyses by experts to validate whether the operations are fully convertible to high abstractions.

### C. Mapping abstraction levels to IF levels

Realization of services at different AF levels impose different requirements on the IF, IF is categorized into different categorizes as depicted in Table VI, and based on inputs from several architects. Again for sake of brevity we have considered three levels of IF, but the method is scalable to any number of IF levels.

Once the IF levels are determined it is required to identify levels of abstractions that are feasible to be realized by the IF factors. A basic AF level maps to a high IF level; a high AF level maps to a low IF level; whereas medium AF and IF levels map to each other.

### IV. SATE ENGINE AND ALGORITHM

The SATE engine determines the feasible set of services that can be realized by the available resources (measured in terms of IF) in a service providing environment. It first takes input of all the resources, tools and skills of employees from the service providing environment and calculates the IF factor and levels. It further takes the service abstraction as input and calculates the abstraction levels (AF). Figure 4 shows the different abstraction levels and the IF in a service providing entity. The goal of the SATE engine is provide optimal allocation of the service abstraction to the different individuals/employees (defined by different skill levels).

In order to perform the allocation of the services to the Employees based on the IF factors and levels we provide a variant of the Hungarian allocation algorithm (see [13] for details). Hungarian allocation algorithm considers the cost of individuals performing a task and provides an optimized and feasible set of allocation of tasks to individuals. We adapt this algorithm to our task of finding the optimal allocation of service implementations given the IF values associated with employees in the project. We use a matrix interpretation of the bipartite graph that Hungarian allocation algorithm uses. It takes an input of the cost (Impediment factor) versus employees for performing various tasks. Given $n$ employees and service development tasks, an $n \times n$ matrix containing Impediment factors of assigning each service implementation, we find the cost minimizing assignment through the following steps:

1) Let the set $E = E_1...E_n$ be the set of Employees available with different IF’s in the service providing environment. Let $S = S_1...S_m$ be the set of services, abstraction level of individual services being different. The cost matrix $M$ is populated as follows:

$$M_{ij} = \infty \text{ if } IF > AF$$

$$M_{ij} < 0 \text{ if } IF < AF$$

$M_{ij}$ is assumed to be the minimum when the correct

![Fig. 4. Employees with IF and AF factors](image_url)
abstraction levels and impediment levels are available. The rows of the matrix represent the Employees (along with the IF), the columns represent the Services and the values in the matrix represent the AF levels for services being realized by the employees.

\[
\begin{pmatrix}
E_{11} & E_{12} & E_{13} & E_{14} \\
E_{21} & E_{22} & E_{23} & E_{24} \\
E_{31} & E_{32} & E_{33} & E_{34} \\
E_{41} & E_{42} & E_{43} & E_{44}
\end{pmatrix}
\]

2) Then we perform row operations on the matrix. To do this, the lowest of all \(E_i\) (belonging to 1-n) is taken and is subtracted from each element in that row. This will lead to at least one zero in that row (we get multiple zeros when there are two equal elements which also happen to be the lowest in that row). This procedure is repeated for all rows. We now have a matrix with at least one zero per row. Now we try to assign services \(S_i\) to Employee \(E_i\) such that each employee performs only one task and the penalty incurred in each case is zero. This is illustrated below.

\[
\begin{pmatrix}
0 & E_{12} & 0 & E_{14} \\
E_{21} & E_{22} & E_{23} & 0 \\
0 & E_{32} & E_{33} & E_{34} \\
E_{41} & 0 & E_{4} & E_{44}
\end{pmatrix}
\]

3) Sometimes it may turn out that the matrix at this stage cannot be used for assigning, as is the case in for the matrix below.

\[
\begin{pmatrix}
0 & E_{12} & E_{13} & E_{14} \\
E_{21} & E_{22} & E_{23} & 0 \\
0 & E_{32} & E_{33} & E_{34} \\
E_{41} & 0 & E_{4} & E_{44}
\end{pmatrix}
\]

In the above case, no assignments can be made. Note that Service 1 is done efficiently by both employees A and B. Both cannot be assigned the same Service. Also note that no individual does service 3 efficiently. To overcome this, we repeat the above procedure for all columns (i.e., the minimum element in each column is subtracted from all elements in that column) and then check if an assignment is possible.

4) In most situations this will yield a feasible result, but if it is still infeasible then the procedure described below must be followed.

Initially assign as many services as possible then do the following (assign services in rows 2 and 3)

\[
\begin{pmatrix}
0 & E_{12} & E_{13} & E_{14} \\
E_{21} & E_{22} & E_{23} & 0 \\
0 & E_{32} & E_{33} & E_{34} \\
E_{41} & 0 & E_{4} & E_{44}
\end{pmatrix}
\]

Mark all rows having no assignment (row 1). Then mark all the columns having zeros in that row(s) (Column 1). Then mark all rows having assignments in the given column (row 3). Repeat this till a closed loop is obtained.

Now draw lines through all marked columns and unmarked rows.

The aforementioned detailed description is just one way to draw the minimum number of lines to cover all 0’s.

5) From the elements that are left, find the lowest value. Subtract this from the marked rows, and add this to the marked columns. A feasible assignment might not be feasible in one iteration of the above steps. If an optimum assignment is not feasible steps 1 to 4 are repeated iteratively until optimality is achieved. Optimality is achieved when the minimum number of lines used to cover all the 0’s is equal to the maximum number of individual assignments. We can use dummy variables (usually the max cost) to fill the matrix when the number of people is greater than the number of assignments. If the considered set of individuals and assignments do not converge on optimality, the input set is altered wherein the complexity of the assignments can be reduced. Reducing the complexity of assignments might have an impact on the desired abstraction levels but nevertheless we wanted to achieve abstractions that are easy to implement. It should be noted that proving polynomial time performance of Hungarian algorithm is beyond the scope of the paper.

V. ASSESSMENT AND EVALUATION

We measure the effectiveness of SATE on two metrics

(a) Service abstraction levels that are easy to realize and

(b) Service abstraction levels that are easy to maintain. The former metric is measured tangibly via usage, and the later is measured by utilizing the time required to implement and realize a given set of services.

A. Measuring Effort/Time for Realization

In order to measure the time taken to realize a service we have implemented our algorithm (SATE Engine) with data from 14 projects. We collected the following information from each project:

1) Service Allocation: The employee/individual/group assigned for the services.

2) Expected time to complete: The time by which the implementation of services are expected to be completed by the individuals/group.

3) Actual time taken to complete: The time that was taken for completion of the services by individuals/groups.

4) Complexity of the service implementation: Different levels of service complexity involved in implementing the service.

Table VII provides the statistics of the projects considered for evaluation.
Figure 5 presents the number of services that were not able to meet the expected time by the individuals. The X-axis represents the different domains that we have considered. The y-axis gives the number of services (even when there is marginal difference in the expected time and the actual time the service was developed) that did not meet the expected time. It also compares the number of services that did not meet the expected time after the assignment performed by the SATE engine. It should be noted that SATE could reduce the number of services that missed the development time by about 30-40%. This is due to the fact that SATE considers the optimal allocation of the individuals (based on their impediment factor) to the service development. This ensures that the individuals are allotted services such that it is easier for them to realize.

The above results strongly suggest that the right services abstraction chosen for the available skill levels significantly improves the time required to realize the service.

B. Measuring effort for Service Maintenance

Each service once realized is available for utilization. One of the major constraints (apart from those that are common for any software design and really distinguishes SOA adoption) for the services maintenance is enhancement of its functionality. This implies that new functions are added to the specifications and services are redesigned to cater the additional functionality. SATE in addition to providing optimal allocation of services to resources, provides provision for identifying various levels of service abstractions by tagging the services. With the feasible set of abstraction levels that a service can manifest one can increase or decrease the service abstraction functionality with minimal changes to the existing services.

In order to evaluate the set of services and its importance in the context of functional enhancement we provided services specifications to a set of 8 expert software architects and requested them to design services to meet the functional requirements of the specifications. Once the basic services were designed the specifications were modified to add more functional requirements that expanded the scope of the existing services to accommodate more functionality. The exercise of modifying the existing services to meet the new additional functional requirements was performed by the architectures. The time required to obtain the new services by modifying the existing services were taken as a measure of the cost/effort required to maintain the services expandability. Table VIII provides the time taken to model the set of services required to meet the specifications. It shows the statistics of different architects and the number of service they modeled and the time taken for achieving the same. It also shows the time required to perform the changes to the existing services to support additional functionality.

We then demonstrated the SATE Engine to the architects that allows tagging of services with different abstraction levels and the exercise was asked to be repeated. The results are displayed in Table IX. In the table it can be seen that when the SATE engine was used the time required to accommodate the modifications was reduced to about 40-50%. This is due to the fact that SATE Engine allows tagging of services with different abstraction levels that services can manifest. By using this feature in SATE the services when designed initially can be tagged with information about different abstraction levels supporting different functional requirements. Software architects could use this feature to accommodate the functional requirements with ease. This shows that SATE helps in reducing the effort required in maintaining services when existing services are required to support increased functionality.

From the foregoing discussions it can be seen that SATE engine helps better management of resources and time while developing services functionality(increased efficiency of developing by optimal allocation of Services development to resources). Further it helps in better maintenance of the services (40-50% reduction in time taken to modify the services).

VI. Conclusions

In this paper, we have addressed the non-trivial research issue of achieving optimal threshold levels for service ab-
TABLE IX
EFFICIENCY OF SERVICES DESIGN - WITH SATE ENGINE

<table>
<thead>
<tr>
<th>Architect</th>
<th>Number of services</th>
<th>Time taken for designing services</th>
<th>Number of functional expansion</th>
<th>Time taken to modify the requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architect 1</td>
<td>15</td>
<td>2 weeks</td>
<td>3</td>
<td>2 days</td>
</tr>
<tr>
<td>Architect 2</td>
<td>14</td>
<td>2 weeks</td>
<td>3</td>
<td>1 day</td>
</tr>
<tr>
<td>Architect 3</td>
<td>12</td>
<td>2 weeks</td>
<td>3</td>
<td>2 days</td>
</tr>
<tr>
<td>Architect 4</td>
<td>16</td>
<td>2.5 weeks</td>
<td>3</td>
<td>1 day</td>
</tr>
<tr>
<td>Architect 5</td>
<td>14</td>
<td>2.5 weeks</td>
<td>3</td>
<td>1 day</td>
</tr>
<tr>
<td>Architect 6</td>
<td>13</td>
<td>2 weeks</td>
<td>3</td>
<td>2 days</td>
</tr>
<tr>
<td>Architect 7</td>
<td>12</td>
<td>2 weeks</td>
<td>3</td>
<td>2 days</td>
</tr>
<tr>
<td>Architect 8</td>
<td>12</td>
<td>2.5 weeks</td>
<td>3</td>
<td>2 days</td>
</tr>
</tbody>
</table>

 extractions in service development environments. We have presented SATE, our threshold estimation engine. SATE determines the optimal threshold levels based on the different primitive levels to be defined, and their complexities. Our evaluation of SATE on real-world examples has shown that it performs much better than (human) software architects on two fronts: accurate threshold estimations and reducing time and effort needed to maintain already developed services.

Future work would involve evaluating SATE on larger case studies, especially those involving service maintenance.

REFERENCES


