Design Web Services: Towards Service Reuse at the Design Level

Wang Chu, Depei Qian
Department of Computer Science and Technology, Xi'an Jiaotong University, Xi'an, China
Email: sdchuw @ 163.com

Abstract—**Service oriented software development has gained more and more importance in the area of e-business. Most researchers focus on the semantic description of Web services and automated composition but pay little attention to how to design Web services for supporting service reuse effectively, thus a substantial amount of modeling and programming is still required. In this paper, a pattern oriented service design method is presented by which all components of different abstraction levels are uniform in regard to their specifications and composition mechanism. Pattern oriented service design model provides a traceable and explicit link from business components to software components so that the top-down service reuse at design level is supported, and the service maintainer profits from the traceability because the impact of requirement or design modifications can be better assessed. The semantic information of component specification can support semantics based component reuse. The pattern oriented service design model can improve development productivity by offering a better chance of reuse through better modularity.**

Index Terms—**Web service design, E-business, Pattern oriented design approach, Service reuse.**

I. INTRODUCTION

Web service technologies provide the necessary mechanisms to expose shareable resources over the network and allow the resources to be consumed by users across heterogeneous platforms, enhancing interaction across organizations. Hence, Web service is fast emerging as the technology of choice to build e-business applications. The pace of business life is much quicker today than in the past, thus the e-business applications need to be deployed much quicker than earlier software applications. Due to the rapid business pace today, the e-business applications also need to be engineered for more flexibility and adaptability [1]. Many companies are adopting service-oriented architecture (SOA). SOAs offer the potential to increase reuse, create new applications from old and new services.

Adopting service oriented paradigm in practice for real software and system development, however, has uncovered several challenging issues, such as maintaining consistent system configuration or integrity of dynamically composed services, or identifying services at the right level of granularity [2]. From a software engineering perspective, e-business application development has certain characteristics that make it different from traditional software development. The e-business applications have the following characteristics [3]:

(1) Web applications constantly evolve. Managing the evolution of an e-business application is a major challenge—much more demanding than a traditional software development.

(2) Web applications have a compressed development schedule, and time pressure is heavy.

One of the main challenges in the development of e-business applications is the provision of methodologies that support the specification and design of compositions of services. Traditional software engineering methodologies are hardly applied in this scenario, where the environment is highly dynamic. Novel design method must be developed to support the refinement from the business process to the final services. Similarly, novel techniques must be devised to construct compositions of services that can provide feedback to business analysis and stakeholders, who can use this information to devise new business strategies at design time. Currently, the development and maintenance of most e-business applications is chaotic and far from satisfactory. Existing Web design approaches face a few issues as following:

• Lack of composition information (i.e. service dependencies). The dependencies between a single service specification and an overall architectural model are a vital part of a methodology which supports life cycle oriented service composition. Process lifecycle management without a complete system model is not possible [4]. Lack of composition information will hinder service composition with respect to service contexts.

• Lack of formal semantics. Existing researches have used UML extension to describe the web system's architecture. However, it is hard to detect the system problems, such as correctness, consistency etc., of the composition of Web services without a formal semantics of web services architecture [5]. UML models contain business elements in software model levels, e.g., objects, operations, events, which are all finer-grained semantics, and lacks of the ability to support coarse-grained semantic modeling [6,7]. Compared with traditional software design, service design embraces two distinctive characteristics: 1) the business processes can be published

as coarse-grained services; 2) the services at different granularities can be discovered, composed, and verified by means of automated tools. Hence the work products created during service design process should have formal semantics.

• Lack of design information. Web services composition require addressing various challenges related to Web service discovery, orchestration, verification and execution monitoring. However, existing Web development approaches just provide the infrastructure mechanisms for service composition at implementation level. The apparent lack of design information is one of the most significant problems of Web services development. The design information is treated only as non-software artifacts, i.e., as detailed documentation in which design solutions are scrutinized based on a standardized description format. This can create a major maintenance problem as the programmers tend to lose sight of the original design. The original design intents of the Web services are obfuscated, or worse, have disappeared altogether. It takes immense effort to implement and test changes as the effects on other services are hard to predict.

Given a set of published services, it is an open question how the services can "drive" software development through all phases of the software lifecycle. This suggests that an important complement to Web services consists of documentation and guidelines that aid developers during requirements specification and analysis to achieve a mapping from the problem domain to the abstractions provided by the services. Clearly it is not enough to search for reusable service components in a repository late in the development lifecycle. It is necessary to introduce a systematic service design methodology to support the service reuse at higher abstraction levels. In this paper, a pattern oriented service design approach is presented that consists of three phases: business analyzing, service modeling, and software component development. In this solution, business entities and services in a service-oriented application are specified as components, and architecture is used as the blueprint for service composition, using component relationships to shorten the gap between business goal and services. We use patterns to package software design expertise with domain knowledge into conceptual building blocks upon which more complex and more flexible software designs can be built. Patterns constitute a promising attempt to moving the emphasis in software engineering away from service-based implementation towards service-based problem solving. We enrich the component specification by pre/post conditions, behavior, and constraints that support automatic verification. Hence, the reliability of service-composition can be improved by checking the correct usage of services. In particular, the pattern oriented service design is an approach for analysis and design of services that support principles such as reusability and componentization.

It is necessary to point out that this paper only concentrates on the service design, other aspects of service development such as publishing, ontology extracting, and so on will not be discussed.

This paper is organized as follows. Section II outlines the related work on service development approach. Section III details the pattern oriented service design process. Section IV depicts the semantic-based component specification. Section V discusses service reuse at design level. Finally, section VI gives the conclusion and ongoing work.

II. Related Work

Our work is most closely related to two classes of research. The first is component based software development, and the second is the service modeling and composition technologies.

Daniela Barreiro Claro et al. propose SPOC (Semantic based Planning for Optimal web services Composition). The problem of composing web services is reduced into four fundamental phases. The first one is planning which determines the execution order of the tasks, a task is considered as a service activity. The second one is discovery that aims at finding candidate services for each task in the plan. The third phase aims at optimizing services composition, and, finally, the fourth concerns execution. The composition of web services starts by creating the initial plan based on task definitions. All the definitions of existing tasks are located in a repository that the planner can consult for obtaining task interfaces [8].

In [9], Ronan Barrett et al. use Web service semantic descriptions to assist the semi-automatic generation of the distribution pattern model. Distribution patterns express how a composed system is to be assembled and subsequently deployed. Distribution pattern models are a form of platform-independent model. These patterns are considered compositional choreographies, where only the message flow between services is modeled. Distribution patterns are modeled using a UML activity diagram in association with distribution pattern UML profile.

In [10], instead of having only a syntactic interface to a component, Joseph R. Kiniry provides a higher-level semantic specification. The key to this new solution is the notion of semantic compatibility. Components are described with domain-specific documentation extensions called semantic properties. These properties have a formal semantics that are specified in kind theory and are realized in specific programming languages. The semantic components are composed automatically through the generation of "glue" code, and such compositions are formally validated. Composition is a constructive operation—its result is a new thing that has some of the properties of its constituent pieces. The semantics of the specific composition operation used dictates the properties of the new construct.

Ioana Sora et al. address the composition problem of a whole system according to a set of requirements by dividing it into sub-problems of layered compositions. The composition strategy is driven mainly by the dependencies established between components by their requirements. The automatic composition problem is the

following: given a set of requirements describing the desired system, and a component repository that contains descriptions of available components, the composition process has to find a set of components and their interactions to realize the desired functionality. The requirements describing the desired system have to be expressed as sets of required properties, defined in the same vocabulary as used for the component descriptions. Rather than enumerating desired system properties, requirements are expressed in a sufficiently high abstraction level domain specific language [11].

Fábio Zaupa et al. present a development environment that focuses on application development process based on services. This environment supports the generation of Web applications based on the Service oriented Architecture (SOA) and the product line approach. The development process consists of a set of activities: 1) Define the application domain; 2) Model the services based on feature models of the product line approach; 3) Instantiate the feature models; 4) Map the instantiated feature model to a corresponding implementation diagram; 5) Implement the service from the implementation diagram; and 6) Generate applications based on the defined services. The concepts of product line are applied to support the modeling of domain services. These services are configured to compose Web applications. Features are abstractions to represent the capabilities of Web applications. Each high-level feature is realized through a domain service. Each service encompasses the operations and data needed to realize the associated feature [3].

Jaejoon Lee et al. propose an approach that identifies reusable services at the right level of granularity. The approach is adapted from the analysis technique of product line engineering, which is the most successful approach for establishing reuse in practice. They present how reusable services can be identified and specified based on features: these features identify variations of a family of products from a user's point of view and thus will be the subjects of reconfigurations of service centric systems at runtime [2].

Kevin Jin et al. present a business-oriented service design and management methodology. The methodology integrates software engineering techniques, such as design patterns to develop IT solutions from a service business perspective [12].

Software composition is the construction of software applications from components that implement abstractions pertaining to a particular problem domain. Raising the level of abstraction is a time-honored way of dealing with complexity, but the real benefit lies in the increased flexibility: a system built from components should be easy to recompose to address new requirements [13].

III. Pattern Oriented Service Design

Patterns represent platform and programming language independent solutions for design problems in a certain design context, which is an organizing means that facilitate rapid mapping from business requirements to infrastructure designs. Designers can make the application development processes more manageable by defining patterns [14,15].

A. Architecture Centric Design Approach

Architectures play a major role in determining the quality and maintainability of a system. The overall architecture of a software system has long been recognized as important to its quality. Architectures are the foundation for designing, communicating, and constructing complex software systems. The intent of architectures is to illustrate a software system's decomposition into the individual components, the communication paths, and the processing resources. Components are widely used for managing distributed applications because they not only capture the software architecture of managed applications as an assembly of components but also permit to dynamically adapt these applications to changing environments [16,17]. In this paper, architecture is used to describe component composition.

The architecture centric design follows the traditional "divide-and-conquer" approach of defining architecture that consists of three activities: 1) Goal decomposing: the objective of this activity is to divide the system goal or requirements into a number of sub-goals and assign them to components. In this activity, "Responsibility-Assignment" relationships between the system goal and the components are created, and they are called γ-relationships; 2) Architecture defining: the objective of this activity is to construct architecture to achieve the system goal. Determining cooperation rule(s) of the components is the major work of this activity. "Take-Part-In" relationships between the components and the architecture are created, and they are called β-relationships; 3) Validating: the objective of this activity is to check whether the constructed architecture meets the system goal or requirements. In this activity, "Achieved-By" relationship between the system goal and the architecture is created, and it is called λ-relationship.

The divide-and-conquer procedure results in a design pattern, written as $L\rightarrow_{\chi}A\{C_i|i\in N\}$, which contains following component relationships: $\{\gamma_i: L\rightarrow C_i|i\in N\}$, $\{\beta_i: C_i\rightarrow A|i\in N\}$, and $\lambda: L\rightarrow A$, where L, C, and A represent "Goal/requirements", "Component", and "Architecture" respectively. The created relationships are used to trace and to understand the architectures.

In this paper, the architecture centric design approach is applied to all of the service design phases: business analyzing, service modeling, and software component development.

B. Business Analyzing

Business analyzing serves as the first step in service design process. By analyzing the business operation and structure of the organization, domain engineers construct business patterns and identify what business functionality is potentially of use to others.

We use the concept of business component as a means to encapsulate the business goal, constituent partners, activities, constraints, and so on.

(1) Identifying business pattern.

Firstly, domain engineers identify the strategic business domains with appropriate stakeholders. This step will allow the definition of business invariants used as criteria to define the optimal e-business patterns, and then specify the business patterns.

Business analyzing needs to be able to model all three aspects: resources, organization and processes. As a first step, build high-level, abstract models of the business goals, business processes, and business entities, with the aim of:

• identifying reusable business components that provide services;

• identifying the business policies that must be obeyed by business components;

• identifying common domain ontology.

Definition 1 (Business pattern) Let R_0, $\{R_i|i\in N\}$ be business components, R be a business architecture, R_0 only contains business goal, and there exist $\{\gamma_i: R_0\rightarrow R_i|i\in N\}$ and $\{\beta_i: R_i\rightarrow R|i\in N\}$. If there exists λ: $R_0\rightarrow R$, then R_0, R, $\{R_i\}$, $\{\gamma_i\}$, $\{\beta_i\}$, and λ constitute a business pattern, written as $R_0\rightarrow_\lambda R\{R_i|i\in N\}$.

Definition 2 (Business model) A business model is a set of business patterns at different levels of granularity, written as $Q_M=\{R_0^j\rightarrow_{\lambda j}R^j\{R_i^j|i\in N\}|j\le n\}$, where n is the number of the business patterns.

Business patterns are specified at several levels of granularity in a consistent way and can be used to describe team, department, or whole organization.

(2) Validating business model.

Domain engineers assess the quality of the business model, using automation tool whenever possible.

The e-business application development starts from the premise that all businesses have a business design. A business design describes how that business works – the processes that it performs; the organizational structure; the economic and market influences that affect how that business achieves its goals; the rules and policies that condition how the business operates. The foundations of business design are business processes that are part of the fabric of a business and contribute to how the business functions and responds to its customers [18]. Business analyzing helps domain engineers understand, communicate, and learn more about the different aspects of business processes in the organization.

A business pattern specifies the business architecture for products, services and information flow. The business patterns are the conceptual and architectural implementation of the business strategy and are the foundation for the service design [19].

C. Service Modeling

The objective of service modeling is to package business operations as service components. This phase results in hierarchical service composition patterns. Service components perform useful business functions through the well-defined interfaces. The main advantage of service components is that they enable practical reuse of assets both within and across organizations. Given a business pattern specifying business goal, packaging business functionality as service components involves the following steps:

(1) Service component defining.

Designers decompose business goal, define service components, and create specifications describing the service component's functionality.

(2) Architecture constructing.

The objective is to define cooperation rule(s) of the service components.

(3) Composition pattern validating.

When a service composition pattern is finished, it is time for developers and users to validate whether the composition pattern meets the given business goal. It is important to assess the operation of the service composition patterns and the potential implications of introduction of the services on the organization. This evaluation should include: possible impact of the introduction of the e-business applications on the organization; the resulting changes in its business processes, and so on.

Service modeling is a top-down procedure. The business components within a business pattern offer high level service components. After the big picture of communication between coarse service components is defined, the coarse service components are divided into smaller service components which encapsulate business processes. Iteration is needed here so that the right size of services is identified and the relationships between service components are modeled explicitly. The component relationships form services dependency graph. Service composition pattern not only supports service reuse but also supports service relationships reuse.

Definition 3 (Service composition pattern) Let $\{G_i|i\in N\}$ be service components, G be an architecture, R_0 be a business component or service component that contains business goal, and there exist $\{\gamma_i:R_0\rightarrow G_i|i\in N\}$ and $\{\beta_i:G_i\rightarrow G|i\in N\}$. If there exists λ: $R_0\rightarrow G$, then R_0, G, $\{G_i\}$, $\{\gamma_i\}$, $\{\beta_i\}$, and λ constitute a service composition pattern, written as $R_0\rightarrow_\lambda G\{G_i|i\in N\}$.

Definition 4 (Service model) A service model is a set of service composition patterns at different abstract levels, written as $U_M=\{R_0^k\rightarrow_{\lambda k}G^k\{G_i^k|i\in N\}|k\le m\}$, where m is the number of the patterns.

The service model may contain coarse-grained business patterns.

The granularity of services must become more like business activities that business components perform and much less like fine-grained software interfaces. Fine-grained services make interoperability between applications and between business partners difficult because the fine granularity is, at least in part, dictated by technology that provisions their business capabilities. It is far easier to interoperate at business activity level because businesses largely perform common business tasks (where interoperability often must take place) in similar ways [20].

Service composition patterns describe how to compose service components and provide a seamless record of trace information from high-level business components

down to simple service components. The simple service components specify software requirements and constraints that are used to design software components.

Modeling service at the architectural level is in its embryonic stage, which is used to specify high-level compositional view of a software application [21].

D. Software Component Development

Software component development consists of two activities: software component design and software component implementation. Software component development results in a software component model.

Software component design is an architecture centric design procedure which is based on decomposing the software requirements contained in service components. The top-level software requirements are decomposed into coarse conceptual components. The conceptual components and the cooperation rule are aggregated into a logical architecture. The logical architecture is validated whether to meet the software requirements. The coarse conceptual components are, in turn, decomposed into fine conceptual components. When all of the conceptual components can be implemented or be used to select pre-existing software components, component design is finished.

Definition 5 (Software component composition pattern) Let $\{T_i|i\in N\}$ be conceptual components or existing software components, T be an architecture, G_0 be a service component or conceptual component that contains software requirements, and there exist $\{\gamma_i:G_0\rightarrow T_i|i\in N\}$ and $\{\beta_i:G_i\rightarrow T|i\in N\}$. If there exists $\lambda:G_0\rightarrow T$, then G_0, T, $\{T_i\}$, $\{\gamma_i\}$, $\{\beta_i\}$, and λ constitute a software component composition pattern, written as $G_0\rightarrow_\lambda T\{T_i|i\in N\}$.

Definition 6 (Software component model) A software component model is a set of software component composition patterns, written as $T_M=\{G_0^l\rightarrow_{\lambda l}T^l\{T_i^l|i\in N\}|\ l\le q\}$, where q is the number of the patterns.

Service model and software component model constitute Web service design model.

Definition 7 (Service design model) A service design model is a 3-tuple $H=(Q_M, U_M, T_M)$, where Q_M, U_M, T_M are business model, service model, and software component model respectively.

The pattern oriented design process results in a hierarchical pattern model. The different components at different abstract levels can be traced by component relationships within the patterns.

An e-business application is a living system. It continues to evolve, change, and grow. Poor design and infrastructure have caused many Web applications to be unable to support the demands placed on them, so they have therefore failed [1]. A sound design model must be in place to support the evolution of an e-business application in a controlled, but flexible and consistent manner. The pattern oriented service design approach helps to create a reusable design model that will allow evolution and maintenance of an e-business application.

The pattern oriented service design approach is based on the idea that each service can be traced back to a certain business component. Having identified the related design patterns, one can document them with formal techniques and provide them as reusable and evolvable assets. Thus, the service based application development becomes such activities as specialization, alteration, and assembly of the patterns.

IV. Semantic-Based Component Specification

Semantics is one of the key elements for the automated composition of components. Web services need a formal model to facilitate the automated composition of components at varying levels of granularity [21].

A component (i.e., business component, service component, software component) C can be characterized as: "C = (*Features* + *Resources* + *Constituent partners* +*Operations*+ *Choreography*) + *Behavior* + *Constraints*".

Features contain three attributes: *Domain*, *Name*, and *Synonyms*. *Domain* gives the area of interest of the component. The *Synonyms* attribute contains a set of alternative characteristics of *Name*. Components take part in architecture through their *Operations*. *Constituent Partners* are components that cooperate with each other and regulated by *Choreography*.

Operations are described at three levels: syntactic, semantic, and operational.

Syntactic properties: *Operations* are syntactically described by the following attributes: *name*, *mode*, *input*, and *output*. Each operation has input parameters, output parameters, or both. A parameter has a name and type associated with it.

Semantic properties: The semantics of *Operations* is crucial to component discovery. It is necessary to include a semantic specification for the meaning of the operations. The information at the "semantic" level can be described by using formalisms like the pre/post conditions. Semantic properties defined for operations include *Pre-condition*, *Post-condition*, and other domain specific properties.

Operational properties: We propose to provide *Scenarios* as operational properties that can be used as interpretation of *Operations* to understand component function and to validate constructed patterns. In addition, *Scenarios* can be used to validate whether the discovered component satisfies business goal, and be used to conduct regression test when the developer/provider change the component implementation, leaving the interfaces unchanged.

Given the dynamic nature of the SOA environment, continuous evaluation of service components is one approach for achieving a level of trust. Standard verification techniques are not sufficient [22]. Online verification is required. The pre- and post-conditions and scenarios can support online verification.

In the component specification, *Behavior* describes another kind of semantic information of *Operations* by using formalisms like finite state transition system. Not all the operation sequences are permitted. *Behavior* is used to determine valid order of *Operations*.

Resources record resources that can be accessed by the specified component.

Besides the previous information, it is also necessary to specify another kind of semantic information concerning with the interaction "protocols" (also named "choreography"). *Choreography* determines the interoperability of *Constituent partners*.

Constraints refer to non-functional "properties" of the component, i.e., security, reliability, performance properties, or business rules. In fact it is often the case that several discovered components possess the same functionalities. The automated selection of one component among these functionally equivalent components may require constraints compatibility.

Definition 8 (Component description) A component description is a tuple $D=(\theta, \Phi, E)$, where:

• θ is a set containing signatures of component description. The signatures are semantically annotated using appropriate domain ontology.

• Φ is a set of functional goals (operations) and constraints.

• E is the context of Φ, including *Choreography*, *Constituent Partners*, and so on.

Component description is used to define component specification independent of specific description logic. Component description can use web services technology. Web services technologies are a collection of technologies that allow services to expose interfaces in ways that are discoverable, network accessible, and cross-platform. By exposing interfaces in this way, they will be usable in the widest variety of environments.

Definition 9 (Scenario) Given a component description $D=(\theta, \Phi, E)$, a scenario for a component operation is a pair (M, V), where:

• M is a transition system structure $(W, w_o, \rightarrow, \Gamma)$, Γ is a set of activities of the given operation.

• V is a valuation function: V: $F \rightarrow W \rightarrow S$, F is formulas over θ (e.g., pre-condition and post-condition), S is the sort of a given formula f. $V(f)(w)$ returns the value of f at state w.

Scenario can be used to test the component compositions.

Definition 10 (Component specification) A component specification is a tuple $C=(D, B)$, $D=(\theta, \Phi, E)$, B is a set of scenarios, and $B \models \Phi$.

$\models$ is defined as following: Given a operation $\varphi:<a(p,o)>\psi$, where p is input, o is output, φ and ψ are precondition and post-condition of $a(p,o)$ respectively, it is said to be satisfied by a scenario $b=(M, V)$, $M=(W, w_o, \rightarrow, \Gamma)$, written as $b \models \varphi:<a(p,o)>\psi$, iff there exists path: $w_0 w_1 ... w_n$, $<w_i, w_j> \in \rightarrow$, i, $j \leq n$, φ holds at w_0 ($V(\varphi)(w_0)$ is true), when $a(p,o)$ is executed, the state arrives at w_n, and ψ holds at w_n ($V(\psi)(w_n)$ is true).

By integrating scenarios into component specification, the operational requirement of the component composition is met. The behavior and properties of the composite component can be checked by automated tool[s].

The pattern-oriented service design focuses on business patterns, which it considers as reusable elements. This promotes the idea of viewing an e-business application as federations of services components connected via well-specified choreography that define service interaction.

V. Service Reuse At The Design Level

Constructing e-business applications by composing prefabricated Web service is an attractive vision for software development. In this paper, architecture centric de-composing/composing mechanisms and semantic-based component specifications are introduced to make the Web service tractable and reusable.

In service based e-business development, service composition is done in three steps: 1) Business architecting: Architects define the business requirements and specify business components, interconnections and configuration; 2) Application planning: By applying service design model, lower-level system skeleton will be produced according to higher-level architecture. The skeleton includes choreography and placeholders for services, but also the constraints guiding later composition. Service design model supports top-down service integration; 3) Application assembling: While all services are discovered or implemented in lower-level languages, automated tools are used to integrate the skeleton and services. Component relationships within patterns can support composition checking which can be done by composition tool.

The application planning problem is the following: Given a business model describing the desired system, and a pattern repository that contains descriptions of available architectural patterns, the planning process has to find a set of patterns to map business goal into services. The requirements describing the desired system have to be expressed in the same ontology as used for the component descriptions. The requirements should be expressed in a sufficiently high abstraction level.

The top-down service reuse process through stepwise refinements is depicted in Figure 1. The overall building process is driven by the business requirements. The requirements for the system are put on the main flow of the system and propagated from that point on. The addition of new components on the flow occurs according to the current requirements, which are those propagated from the initial requirements together with those of the new introduced components. When a requirement matches with business component in service design model then the discovered coarse service is reused. Otherwise if that requirement has sub-requirements then it will have to be fine-tuned, so that its internal sub-requirements are used to discover fine-grained components. A solution is considered complete when the current requirement set becomes empty. It is possible that for certain sets of requirements no solution can be found.

We explain service composition process in Figure 1 as following cases:

Case 1: When business goal g' matches with the business goal g of business pattern $g \rightarrow_\lambda r\{r_1, r_2\}$, the business pattern and related services are reused. This is coarse business process integration.

Case 2: Business goal g' doesn't match with the business goal of business patterns within business model,

thus the goal g' is decomposed into sub-goals r_x and r_y. Sub-goal r_x matches with the business goal r_1 within business pattern $g \rightarrow_{\lambda} r\{r_1, r_2\}$, hence service composition pattern $r_1 \rightarrow_{\lambda 1} r_1'\{r_{11}, r_{12}\}$ and related services are reused.

Case 3: Sub-goal r_y is further decomposed into goals r_{y1} and r_{y2}. Goal r_{y2} matches with a simple service r_{n2}, and service r_{n2} is reused. Goal r_{y1} is decomposed into r_{y11} and r_{y12} that are used as requirements specification to select existing software component or to develop new software components.

An increasing number of software corporations are realizing that most software development projects are not one-of–a-kind efforts. They can develop common component assets for future contracts and products in the same application domain. Thus, they can achieve large-scale productivity gains. Hence new software process model is needed that explicitly distinguishes between software development for reuse which creates core component assets and software development with reuse which uses core component assets to create products. In order to support the top-down service reuse from business model to software components, in this paper, we apply the pattern oriented approach to service design by which a traceable service design model is constructed.

It is important to ensure that the service design model is usable. To be usable, the service design model must be easy to trace, to communicate to stakeholders and to maintain. The traceability is essential to reuse the service design model. Software maintenance profits from traceability because the maintainers understand why a system was built the way it was, and can better assess the impact of requirement or design modifications. During development of an e-business application, engineers can select the services from the service design model that meet stakeholders' needs, and then assemble them.

In the architecture-centric service design process, model transformations from business model and requirements model to component design and component implementation are seamless. The traceability between different models at different abstraction levels can be captured and maintained as a side effect of the design process of service design model. In addition, architectural patterns are used to model and to document service design model, so that the service design model is traceable and reusable.

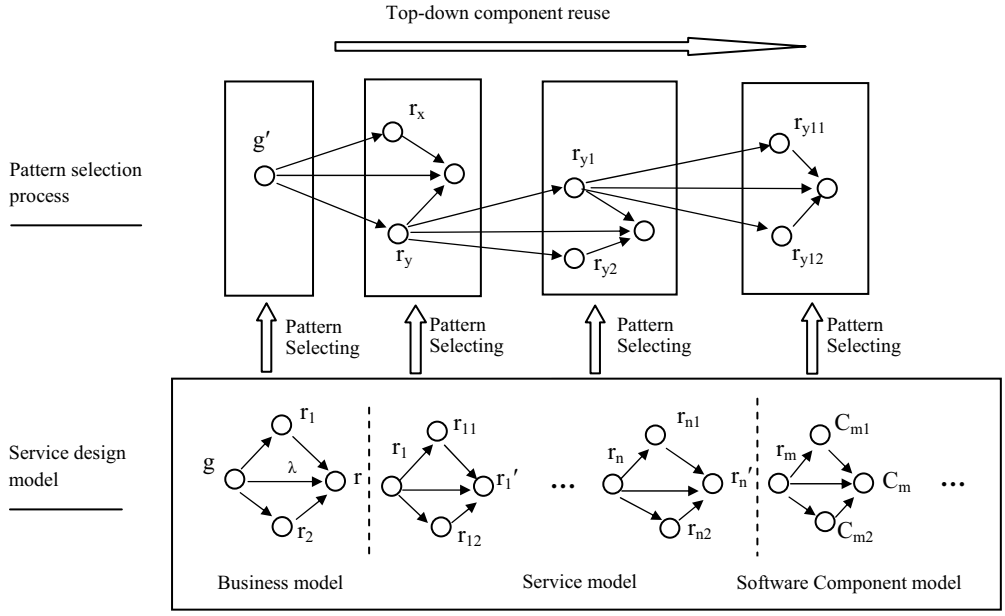

Figure 1. Pattern based service reuse

VI. Conclusion and Future Work

Realizing service reuse is more challenging than realizing the reuse of traditional software component. That is because the service reuse process is conducted automatically at runtime. Compared with the traditional service design methods, the main features of our approach are listed as following:

(1) Domain-specific pattern reuse: From software engineering perspective, the business patterns tend to recur, and the best service design practices tend to recur. Patterns are a means of providing reusable solutions to repeating problems.

(2) Service reuse: The service design model supports service reuse at different levels of granularity. Coarse-grained services are more reusable because any changes of the implementation occur inside the services and are hidden from the user. This enables services to be adapted

and reused more easily. In addition, the semantic information of components and explicit component relationships can support service discovery effectively. Business patterns separate concerns between the business logic of the e-business application and the service components, which also improves reusability of the services. The service design model is a hierarchical structure that supports top-down component reuse, that is, the service reuse is supported when the e-business application is designed.

(3) Service composition: The architectures within the patterns can be used to guide service composition, which contain the configuration skeletons for component composition. The behavior and scenarios of the component specifications support runtime verification of the service composition.

(4) Reduction of time and development costs: As the services of e-business application evolve, the major development effort concentrates on the business logic of each new application. The patterns are matched, and the services only need to be searched and configured to the e-business architecture. This reuse of services reduces much of the necessary modeling and programming needed to construct an e-business application.

(5) Maintainability. Patterns are used to represent service design information while simultaneously providing a traceable and explicit link from business components to software components. The patterns map business goal into services and software components. The service maintainer profits from the traceability because he/she can understand why a service was designed the way it was, and can better assess the impact of design modifications.

Future work includes automated means for semantics based pattern matching, pattern management, and pattern test.

REFERENCES

[1] San Murugesan, Athula Ginige. “Web Engineering: Introduction and Perspectives”. In: *Software Engineering for Modern Web Applications: Methodologies and Technologies*. IGI Publishing, 2008. pp.1-24.

[2] Jaejoon Lee, Dirk Muthig, Minseong Kim, Sooyong Park. “Identifying and Specifying Reusable Services of Service Centric Systems through Product Line Technology”. *Proceedings of the First Workshop on Service-Oriented Architectures and Software Product Lines*. May 2008. pp.D1-D11.

[3] Fábio Zaupa, Itana M. S. Gimenes, Don Cowan, Paulo Alencar, Carlos J. P. Lucena. “A Service-oriented Process to Develop Web Applications”. *Journal of Universal Computer Science*, Vol. 14, No. 8, 2008. pp.1368-1387.

[4] Konrad Pfadenhauer, Schahram Dustdar, Burkhard Kittl. “Challenges and Solutions for Model Driven Web Service Composition”. *Proceedings of 3rd International Workshop on Distributed and Mobile collaboration (DMC)*, 2005.

[5] Yujian Fu, Zhijiang Dong, Xudong He. **“**Formalizing and Validating UML Architecture Description of Web Systems”. *Proceedings of ICWE’06 Workshops*, 2006.

[6] Zhongjie Wang, Xiaofei Xu, and Dechen Zhan. “A Survey of Business Component Identification Methods and Related Techniques”. *International Journal of Information Technology*. Vol. 2 No 4, 2006. pp.229-238.

[7] Jaap Gordijn and Hans Akkermans. “Designing and Evaluating E-Business Models”. *IEEE Intelligent Systems*. July/August 2001. pp.11-17.

[8] Daniela Barreiro Claro, and Patrick Albers, and Jin-Kao Hao. “Web services composition”. *In Semantic Web Service, Processes and Application*. Springer, 2006. pp.195-225.

[9] Ronan Barrett, Claus Pahl. “Semi-Automatic Distribution Pattern Modeling of Web Service Compositions using Semantics”. *Proceedings of the 10th IEEE International Enterprise Distributed Object Computing Conference*. 2006, pp.417–422.

[10] Joseph R. Kiniry. “Semantic Component Composition”. *Proceedings of the Third International Workshop on Composition Languages*, 2003.

[11] Ioana Sora, Pierre Verbaeten, Yolande Berbers. “Using Component Composition for Self-customizable Systems”. *Proceedings of Workshop on Component-Based Software Engineering: Composing Systems from Components*, 2002. pp.23-26.

[12] Kevin Jin, Pradeep Ray. “Business-oriented Development Methodology for IT Service Management”. *Proceedings of the 41st Hawaii International Conference on System Sciences*, 2008.

[13] Oscar Nierstrasz, Theo Dirk Meijler. “Research Directions in Software Composition”. *ACM Computing Surveys*, Vol. 27, No. 2, 1995. pp.262-264.

[14] Hampel A. & Bernroider E. “A Component-based Framework for Distributed Business Simulations in E-Business Environments”. *Proceedings of the Fourth International Conference on Electronic Business-Shaping Business Strategy in a Networked World*. pp.370-375.

[15] Bruce Robertson, Val Sribar. “The e-Business Challenge”. *Enriching the Value Chain: Infrastructure Strategies Beyond the Enterprise*. Intel Press IT Best Practices Series, 2006.

[16] Anthony I. Wasserman. “Principles for the Design of Web Applications”. *http:// www. se-hci.org/bridging /interact 2005 /07_Wasserman.pdf*. 2005.

[17] Julie Street, Hassan Gomaa. “Software Architectural Reuse Issues in Service-Oriented Architectures”. *Proceedings of the 41st Hawaii International Conference on System Sciences*, 2008.

[18] Michael P. Papazoglou, Paolo Traverso, Schahram Dustdar, Frank Leymann. “Service-Oriented Computing: a Research Roadmap”. *Int. J. Cooperative Inf. System*. 17 (2): 223-255, 2008.

[19] Alexander Osterwalder, Yves Pigneur. “An e-Business Model Ontology for Modeling e-Business”. *Proceedings of the 15th Bled Electronic Commerce Conference e-Reality: Constructing the e-Economy***,** 2002.

[20] Thomas B Winans, John Seely Brown. “Policy-driven Service Oriented Architectures”. Working Paper, Deloitte Development LLC., May 2008.

[21] Nikola Milanovic, Miroslaw Malek. “Architectural Support for Automatic Service Composition”. *Proceedings of the IEEE International Conference on Services Computing (SCC 2005)*, USA, 2005, pp.133-140.

[22] Leonardo Salayandía, Ann Q. Gates. “Towards a workflow management system for service oriented modules”. *Int. J. Simulation and Process Modelling*, Vol.3, No.1-2, 2007. pp.18-25.