university of groningen

University Medical Center Groningen

University of Groningen

Interface inheritance for object-oriented service composition based on model driven configuration

Andrea, Vincenzo D'; Fikouras, Ioannis; Aiello, Marco

Published in:
EPRINTS-BOOK-TITLE

IMPORTANT NOTE: You are advised to consult the publisher's version (publisher's PDF) if you wish to cite from it. Please check the document version below.

Document Version
Publisher's PDF, also known as Version of record

Publication date:
2004

Link to publication in University of Groningen/UMCG research database

Citation for published version (APA):
Andrea, V. D., Fikouras, I., & Aiello, M. (2004). Interface inheritance for object-oriented service composition based on model driven configuration. In *EPRINTS-BOOK-TITLE* University of Groningen, Johann Bernoulli Institute for Mathematics and Computer Science.

Copyright
Other than for strictly personal use, it is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license (like Creative Commons).

The publication may also be distributed here under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license. More information can be found on the University of Groningen website: https://www.rug.nl/library/open-access/self-archiving-pure/taverne-amendment.

Take-down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Downloaded from the University of Groningen/UMCG research database (Pure): http://www.rug.nl/research/portal. For technical reasons the number of authors shown on this cover page is limited to 10 maximum.

Interface inheritance for object-oriented service composition based on model driven configuration

Vincenzo D'Andrea
DIT, Univ. of Trento
Via Sommarive, 14
38100 Trento
Italy
dandrea@dit.unitn.it

Ioannis Fikouras
BIBA, Univ. of Bremen
Hochschulring 20
28359 Bremen
Germany
fks@biba.uni-bremen.de

Marco Aiello
DIT, Univ. of Trento
Via Sommarive, 14
38100 Trento
Italy
aiellom@dit.unitn.it

ABSTRACT

In today eCommerce environments, customers have to deal with a wide variety of alternatives, both in terms of service offerings as well as service providers. They risk to be overwhelmed by the complexity of alternatives, thus reducing the usefulness of the experience and consequently the likelihood of transactions. There is an increasing need for new ways to reduce the perceived complexity. Service-oriented computing can help the user cope with this problem. With services, interfaces no longer hide units of code, but provide access to complex functionality equivalent to that of entire conventional applications.

We introduce a methodology for extended service composition derived from model-driven configuration and object-oriented systems. By focusing on the concept of interfaces, and applying it to the object-oriented concept of inheritance, we propose an innovative approach to composition that takes into account how the *composed* services can be recognized or accessed via the *composing* service. In order to set the stage, we discuss the similarities between Service Oriented Computing, Object-Oriented Configuration and Object-Orientation. In addition, we provide an overview of knowledge-based systems, described as software systems built by capturing the knowledge used by experts, and more specifically object oriented configuration for implementing service composition.

Categories and Subject Descriptors

H.1 [**Information Systems**]: Models and Principles; D.1.5 [**Software**]: Programming techniques—*Object-oriented Programming*

General Terms

Web services, Object-oriented programming, Model driven configuration

1. INTRODUCTION

"It is the customer who determines what a business is" [13] by attempting to address specific needs and express his personality through custom-made products and services [26]. Customers thus drive vendors to strive for product palettes with an ever increasing number of variants. Consequently the pursuit of differentiation through variety leads to unique products and services [16, 19]. This strategy is known as "mass customization". Mass customization is defined as "when the same large number of customers can be reached as in mass markets of the industrial economy, and simultaneously they can be treated individually as in the customized markets of pre-industrial economies" [12]. According to [31] the objective of mass customization is "to deliver goods and services that meet individual customers needs with near mass production efficiency". Online transactions and specifically eCommerce environments differ greatly from conventional commercial transactions. Online transactions achieve greater execution speeds and can bridge greater distances than traditional commerce. Furthermore purely digital products (i.e., information services or digitized media) can be discovered, adapted, evaluated, purchased, paid for and delivered by a single service platform within a very short timeframe at any time of day or place on earth [32]. Moreover such platforms compared to conventional sales facilities (i.e., brick and mortar stores) are quick and cheap to implement as well as adapt to new requirements even in not previously predetermined ways [32]. This allows for the rapid and inexpensive deployment of on-line stores offering advanced functionality (such as rearranging the product palette for individual customers) impossible to implement in brick-and-mortar facilities.

On the other hand, customers in an eCommerce environment are faced with more information, resulting from a wider variety of alternatives both in terms of service offerings as well as service providers. However the processing of this information occurs based on the same knowledge and information processing capacity available to the customer as in conventional shopping scenarios [32]. These constraints, unaffected by new technologies, result in a significant drawback to high variety strategies. A customer overwhelmed by the amount of available products or frustrated by their complexity is less likely to complete the transaction and purchase the product, and more likely to delay the decision or leave the shop altogether [18]. This behaviour illustrates the need for new ways for retailers to reduce the perceived complexity of their products. Advanced functionalities are designed to help the user cope with a large amount and at the same time a significant complexity of product data. The advance functionality necessary to accomplish the vision of mass customization may be offered by service composition

functionality implemented in a service-oriented infrastructure.

Service oriented computing (SOC) is a new computing paradigm in which complex systems are built on the basis of basic distributed autonomous services by abstracting on the actual implementation and location of the various services [24]. This paradigm allows for a high distribution of the workload, for the building of complex system yet dynamically and easily scalable. Following the "Service Oriented Computing Manifesto" [25], SOC is more formally defined in terms of services, that is:

> Services are autonomous platform-independent computational elements that can be described, published, discovered, orchestrated and programmed using XML artifacts for the purpose of developing massively distributed interoperable applications.

The best-known example of service-oriented technology is that based on web services. In [10], web service are described as

> a networked application that is, able to interact using standard application-to-application Web protocols over well defined interfaces, and which is described using a standard functional description language.

In the SOC paradigm the emphasis shifts from the engineering of appropriate isolated applications towards the integration, orchestration and choreography of a set of independent services over a network. Typical distributed systems properties [8] become of paramount importance in this setting, most notably: heterogeneity, openness, security, scalability, failure handling, concurrency, transparency. Furthermore, in the SOC model no fixed synchronous bindings are established, but rather the computational elements follow the *find-bind-use* model.

If the scene is that of a web of autonomous computational elements that offer simple services exposing their interfaces, then the challenge is that of creating massively distributed applications offering added value by taking advantage of the basic services. In other words, service composition is the cornerstone for the success of the SOC vision.

Various approaches to service composition have been proposed in the literature. On one extreme are those who consider composition as a run-time process in which services are composed on the fly, e.g., [20]. To achieve this, semantic annotation of services going beyond a simple operational interface is mandatory. Efforts involving semantic web technology are blooming, most notable is the semantic web service initiative (`www.swsi.org`), but others based on temporized automata have also been proposed [4]. On the other extreme, many approaches consider composition as an engineering process that starting from user requirements, data or knowledge models arrives at a service composition satisfying the requirements. Examples of this approach are [7, 23].

In [14], we have shown how Model Driven Configuration theory can be exploited for service composition and orchestration, in [11] we have shown the analogies relating object-oriented programming and service-oriented design. In this paper, we propose a methodology for extended service composition derived from model-driven configuration and object-oriented systems, having the notion of service as the central building block. By focusing on the concept of interfaces and applying it to the object-oriented concept of inheritance, we propose an innovative approach to composition that takes into account how the *composed* services can be recognized or accessed via the *composing* service. We propose a classification of service composition, derived from the concepts of inheritance, interface inheritance, and object composition. For instance, from the notion of object composition we derive the definition of *Opaque Composition*, that is, a service is composed by other services without informing the external world of the details of the composing services.

The paper is organized as follows. In Section 2 we provide an overview of knowledge based systems. The paper then proceeds focusing on the use of knowledge-based construction systems, specifically object oriented configuration for implementing service composition. In Section 3, we present a discussion of the similarities between Service Oriented Computing, Object-Oriented Configuration and Object-Orientation, in order to bridge the gap between model driven configuration and services. Section 4 presents the main results of the paper, that is, a methodology for the composition of services based on object-oriented configuration. Concluding remarks and open issues are summarized in Section 5.

2. KNOWLEDGE-BASED SYSTEMS

We focus on the use of knowledge-based construction systems, specifically model-driven variant configuration for implementing service composition. The following section gives thus a broad overview of knowledge-based systems.

Knowledge-based systems are defined in [1] as:

> computer programs which (a) use knowledge and inference procedures (b) to solve problems which, if addressed by a human, would be regarded as difficult enough to require significant expertise.

For the purposes of this paper we use the following definition of *software systems built by capturing the knowledge used by experts and structuring it according to a specific method, in order to solve problems requiring application domain specific knowledge.* Such knowledge stored in a knowledge-base can be organized according to a number of different methods depending on the underlying concept for storing and managing knowledge. Methods in use include rules-based systems using lists of rules for describing dependencies and conditions, case-based systems that use libraries of predefined descriptions of past cases, and finally object oriented or model-driven systems that store knowledge in an object hierarchy with the help of a domain specific data model [29]. Furthermore knowledge-based systems are split into three broad categories Diagnosis, Simulation and Construction, according to the type of problem they attempt to solve [27,

28]. In our work, we focus on the category of Construction and, in particular, on Configuration problems.

The goal of Construction is the creation of a new solution out of a set of existing components. Construction problems include the configuration of products, processes, resources or services. Configuration is the design of a system through identification, parameterization and combination of instantiations of appropriate components types out of a predefined component set [17]. Configuration focusing on the modification of existing constructions is termed Variant Configuration.

Variant Configuration [29] is a process were complex products are composed out of elementary components. A configurator in this sense is a knowledge-based system implementing such process, based on predefined goals as well as domain specific knowledge. Design goals can be constraints, functional requirements, predetermined components or various quality criteria [21]. Such systems do not follow a single predefined method, but rather a strategy based on a series of small steps, each step representing a certain aspect or assumption leading to the configuration of the composite service. Configuration is therefore considered as the solution to a single exercise and not the solution to a whole problem or problem class that has first to be methodically analyzed. This implies the following, see Figure 1:

- The set of all possible solutions is finite.
- The solution sought is not innovative, but rather is a subset of the available parts.
- The configuration problem is known and well defined.

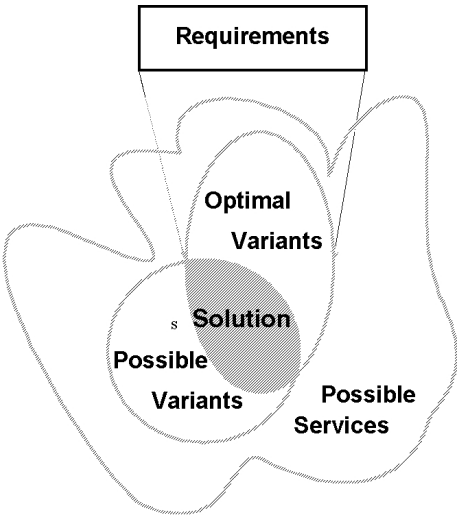

Figure 1: Variant Configuration solution-space.

Configuration as a knowledge-based system requires a knowledge-base as the source of its domain specific knowledge. The structure of this knowledge-base determines to a large degree the configuration process itself. Currently three major types of variant configuration are defined: (i) rules based configuration, (ii) case-based configuration, and (iii) model driven or object oriented configuration.

Object-oriented Variant Configuration is based on the concept of iterative composition of the final product out of a set of elementary components that have been previously organized according to a product data model into a structure, known as the object hierarchy that contains all knowledge related to the product in question. The relationships between components and how they fit together are described with the help of constraints.

Constraints are constructs connecting two unknown or variable components and their respective attributes which have predefined values (taken from a specific knowledge domain). The constraint defines the values the variables are allowed to have, but also connects variables, and more importantly, defines the relationship between the two values [30]. In other words, constraints contain general rules that can be applied to make sure that specific components are put together in a correct fashion without having to specify any component-related rules or calculations [30]. The constraint satisfaction problem is defined as follows [2]:

- There is a finite set of variables $X = \{x_1, \ldots, x_n\}$.
- For each variable x_i, there exists a finite set D_i of possible values (its domain).
- There is also a set of constraints, which restrict the possible values that these variables are allowed to take at the same time.

The object hierarchy contains all relevant objects and the relationships between them in an "is-a" relationship that defines types of objects, object classes and subclasses, and their properties. The configuration process creates objects on the basis of this information according to the products being configured. In one specific hierarchy (as depicted in Figure 2 for the configuration of automobiles), classes for specific car types (i.e., coupè, minivan, etc.) are connected by "is-a" relationships to the main "car" class. This hierarchy also allows the breakdown of a product into components with the help of further "has-parts" relationships. These "has-parts" relationships are the basis for the decision-making process employed to create new configurations. An example of such a relationship would be the relationship between a chassis and a wheel. A chassis can be connected to up to four wheels in a passenger car, but the wheels are represented only once, with appropriate cardinality.

The greatest hurdle to be resolved when creating new configurations is the fact that the software is required to make decisions that are not based on available information. Such an action can possibly lead to a dysfunctional composition or simply to a combination that does not conform to user requirements. In this case all related configuration steps have to be undone (backtracking) in order to return to a valid state. The longer it takes for the configuration to detect that a mistake has been made, the more difficult it is to correct the error in question [21]. The configuration process itself is composed of three phases [9]:

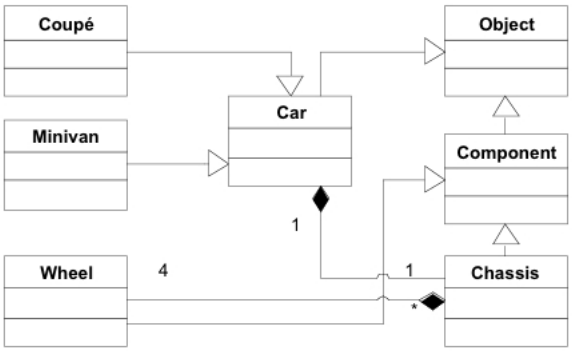

Figure 2: Automotive object hierarchy.

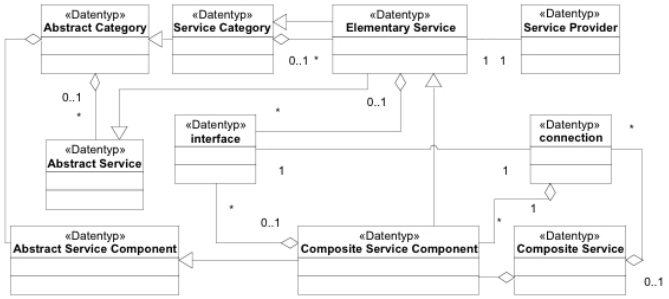

Figure 3: Object hierarchy for composition of services.

- Analysis of the product in order to define possible actions.
- Specification of further configuration actions.
- Execution of specified actions.

These actions are:

- Disassembly of the product into its components. This is meant to reduce the complexity of the problem and create a large number of smaller objectives in the manner of conventional top-down specification.
- Assembly of components, integration and aggregation. This step creates a product out of its components in a bottom-up manner.
- Creation of specialized objects. Object classes are specialized through the definition of subclasses.
- Parameterize objects. Define attributes and parameters for the specified objects that can be used for the application of constraints or other configuration mechanisms.

3. MODEL DRIVEN CONFIGURATION AND OBJECT ORIENTATION

A service composition engine based on object-oriented configuration implemented by project NOMAD [22] employs the following data model for composition of services. It divides services conceptually into two categories, Elementary Services and Composite Services, cf. Figure 3. Elementary Services represent a specific instantiation of a service and contain all data needed to describe it. Composite Services act as templates designed to provide the default knowledge required to produce a specific composition and consist of groups of Components derived individually from Elementary Services. Interfaces can be defined between Elementary Services, Composite Services, Service Categories and Service Providers. For a detailed discussion of the NOMAD service composition data model the reader is referred to [15].

The relationship between interfaces and elementary services matched by the filters contained in an interface resembles the one between plugs and sockets, whereby interfaces as sockets match multiple plugs. Henceforth, connections to Elementary Components that have a direct reference to an interface via its unique identifier will be referred as *sockets* and components that are matched by a socket will be referred to as *plugs*. An interface object is not restricted in its scope to use by only one pair of services, but rather implements a generic rule (constraint) that can be used by multiple components for describing their interfaces. For a detailed discussion of the NOMAD service composition engine the reader is referred to [14].

One of the common metaphors used in textbooks on Object-Oriented programming (OOP) is to view objects in terms of the services they provide, describing them in "service oriented" terms (see for instance [5]). Building on abstraction and encapsulation, the key idea is to hide programming details that provide object functionalities. An interface describes these functionalities in terms of methods and properties, providing a logical boundary between operation invocations and their implementations. Then an object is just a "server" of its own methods.

Objects in this respect closely resemble services with their plug and socket interfaces as implemented based on the above model-driven configuration service composition engine. Furthermore, similarities between the Object-Oriented paradigm and the Service Oriented paradigm as illustrated by this composition engine extend to a number of properties typical of objects and Object-Orientation. Referring to Figure 4, we draw a parallel. The concept of an ontology is fundamental to both paradigms. Any development is based on an ontology appropriate to the application domain in question. Based on this ontology in object-oriented terms use cases and scenarios are defined. These usually lead to a class diagram detailing the architecture to be implemented. This is analogous to the object hierarchy produced from the object-oriented model employed by model-driven configuration. Another common mechanism used to convey semantics related to the overall architecture and propagate best-practice design are design patterns. In order to achieve a certain type of composition in an efficient way (based on best practice) default knowledge is required. This knowledge is provided by composite service templates previously described. Design patterns directly correspond to such composite service templates.

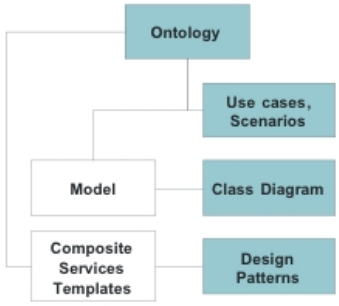

Figure 4: Relations between model driven configuration concepts and object orientation.

Based on these parallels further similarities can be established, see Figure 5. Elementary or composite service definitions directly correspond to classes. Categories of services, providing convenient ways of sorting large amounts of instances of services, are the equivalent of abstract classes, that describe common features but can not produce objects through instantiation. Constraints on the other hand are the equivalent of preconditions and post-conditions commonly used in object-oriented development.

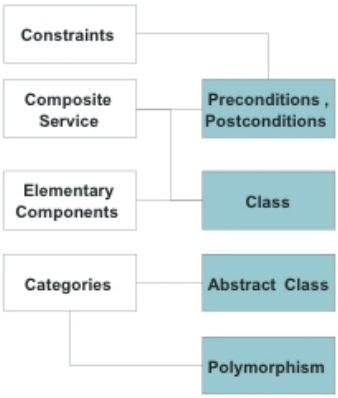

Figure 5: Additional relations between model driven configuration concepts and object orientation

More object related concepts can move into the service oriented world in order to enhance the technology and, perhaps, clarify the role and scope of web services. Here are the most immediate example of concept migration:

Inheritance. Two modes of inheritance are used in OOP: code inheritance and interface inheritance. Interface inheritance is the most immediate to apply to web services. Consider a payment service, which could be subtyped in a service with acknowledgment of receipt. In a workflow, the former could be substituted by the latter as it is guaranteed that the same port types are implemented in the subtyped service. Inheritance enables service substitution, service composition and it induces a notion of inheritance on entire compositions of services. Consider a workflow A built on a generic service and another one B with the same data and control links, but built on services which subtype the services of A. Could we say that B inherits from A or that B is a specialization of A?

Polymorphism. Both inclusion polymorphism and overloading can be extended to the service paradigm. A composition operation in a workflow may have different meanings depending on the type of the composed services. For example, composing a payment and a delivery service may have a semantics for which the two services run in parallel; on the other hand, the composition of two subtyped services in which the payment must be acknowledged by the payers bank and the delivery must include the payment transaction identifier have the semantics of a sequencing the execution of the services.

Composition. A formal and accepted notion of composition is currently missing in the SOC domain and, as just proposed, inheritance and polymorphism could induce such precise notions of composition over services. Some of the gaps left by standards which do not have a clear semantics, most notably, BPEL [3], could benefit from semantically funded definitions of composition.

4. INTERFACE INHERITANCE FOR SERVICE COMPOSITION

Composition is a central issue both in the object-oriented paradigm and in service oriented computing. By means of composition an entity can access other independent entities during the execution of its operations. On the other hand, the concept of inheritance, which is quite central in object oriented systems, does not have a relevant role in the service oriented paradigm.

In object oriented systems, the term *inheritance* is used to describe the mechanism allowing the derivation of a class C_2 from another one C_1. Class C_2, the inheriting class, is said to be a subclass of C_1. The subclass class will have to present the same external interface of C_1, in addition to its own public interface. In other words, it is possible to treat as object O_2 of class C_2 as if it is of class C_1: that is O_2 will accept the same messages of objects of class C_1.

The behaviour of C_2 could extend or limit the behaviour of C_1, but the important fact is that it is defined with respect to C_1 behaviour. One may distinguish between several forms of inheritance [5], but in our discussion we focus on inheritance for *specialization*; a class is defined in terms of specialization of an already existing one – this is expressed by the "is a" relationships. For instance, if we state that a TextWindow is a Window, we mean that the TextWindow has all the properties and behaviors of the Window, plus some additional property and/or behaviour.

Specialization usually implies a semantic coherence between the two classes. When this is true, C_2, that is, the specializing class or subclass, is also called a *subtype* of the class C_1. If semantic coherence is not granted the subclass will just have the same names as C_1 for public variables and methods, but the meanings attached to these interface elements can arbitrarily change. In other words, the subclass requires only a syntactical match, while the subtype guarantees also a semantical match between the involved classes.

The concepts of subclass/subtype are also related to a distinction commonly made between what is sometime referred to as "true" or "code" inheritance versus interface inheritance. The former is used when, besides presenting the same external interface, a class includes also the same code of the inherited class. As a consequence, a subclass formed via code inheritance will also be a subtype unless it explicitly overrides the behaviour of the inherited class. The latter term, interface inheritance, is used when a class has the same external interface of the inherited one, but it has no direct access to its code. In this case, a subclass becomes a subtype only when the behaviour of the inherited class is reproduced with the same semantics.

In terms of implementation, a simplifying model is to view inheritance as a special form of composition. Composition generally implies wrapping the interface of the included classes, and filtering the communication between these classes and the external world – the composition operation could be completely hidden. In the case of inheritance, the interface of the inherited class is added to the one of the inheriting class, letting the external world know of the relationship between the two classes. In addition, in the case of code inheritance, the operation of the inherited class will also be available.

Inheritance can be described as if the inheriting class incorporates (composes with) the inherited one, but without filtering the communication; the inherited class interface is directly accessible. An object of the inheriting class responds to the same invocations as an object of the inherited class. If the subclass is also a subtype, the results will also be the same. To think at inheritance (subtyping) as a form of composition which maintains the interface (behaviour) of the composed object, makes it easier to reason about similar concepts in the service world.

Before presenting the application of interface inheritance for service composition (see Section 4.2), in the next section we discuss the role of composition in the service oriented paradigm and its relationship with the similar concept in OOP.

4.1 Object composition versus service composition

A large amount of effort in research literature and in industry standards is devoted to service composition. As representatives of the approaches mentioned at the end of Section 3, we refer on one hand to authors focusing on designing the composition of service (e.g., [6, 33]) and on the other hand to authors defining how semantically annotated services can be automatically composed (e.g., [20]).

Service composition based on model driven configuration addresses the problem of creation of composite services during run-time. This is achieved through the iterative composition of elementary components into a composite service based on well-defined constraints. Connections between elementary services are realized based on the aforementioned plugs and sockets concept where composition dependencies (connections) that make use of an interface component are referred to as sockets and components that are matched by a socket are referred to as plugs. A composition created based on this process consists of a group of elementary services connected via their interfaces in order to produce a more complex effect defined to be the composite service. Consequently a composite service can synthesize its functionality out of the functionality of a number of other services, e.g., a location based weather forecast service that is composed out of a service providing positioning data and a service providing weather forecast information. This behaviour can be cleanly mapped to the type of composition employed in the context of object oriented development, where the composite service functions as an inheritor and composing elementary services play the role of the parents.

In comparison, composition in Object Oriented development is a design-time activity mainly dealing with statically designing the architecture of the system. To state that an object is composed of another one, means that in the class diagram a containment relationship between the two corresponding classes exists. In this relationship the containing object is able to use the contained one, possibly shielding it from other parts of the system (see Section 4.2).

An additional level of detail, related to composition in the object oriented world, is grounded in the difference between the abstract view of classes and the instantiation process, that is, the creation of the actual objects. A composition relationship between classes C_1 and C_2 will lead to the fact that an object O_1 (instance of class C_1) will contain an object O_2 (instance of class C_2). This result can be achieved in two radically different ways: exclusive or non-exclusive composition. In the former case, the instantiation of O_1 will create O_2, a new instance of C_2; when O_1 will be destroyed, O_2 will also be deleted. In the latter, non-exclusive, case, O_1 will make use of O_2', an already existing instance of C_2; in this case, deleting O_1 will not affect O_2'.

Recapitulating, the main difference between service composition and composition of objects is that composite services are not statically designed, but rather are composed at run-time, as services providing the required supporting infrastructure are composed using dynamic discovery mechanisms. Consequently, the service paradigm provides the capabilities for dynamic, runtime composition instead of a statically planned architecture.

The dynamic nature of service composition has several consequences. A significant one is that negotiation and contractual agreements cannot be accomplished off-line, they have to be dealt with at run-time. The role of catalogues and the discovery mechanism have no counterpart in the world of objects and components.

Services demand a transition from static binding between objects or components that are to be integrated to the dynamic binding of services. From the point of view of the design there is the need of a transition from designing an architecture to designing the *enabling medium*, that is, the infrastructure for runtime composition.

Furthermore, a composite service functioning as the inheritor retains all the interfaces of its individual elementary components playing the role of the parents. This behaviour can be cleanly mapped to object-oriented inheritance mech-

anisms.

4.2 Interface inheritance for service composition

Interface inheritance allows to treat in the same way two elements of a composition relationship: with interface inheritance, a member of a composition can be substituted its inheritor (descendant). Interface inheritance for services guarantees the presence in the inheriting service of a specific interface: the inherited one.

An example is a service A designed for informing client services about conformance to certain policies, for instance, acceptance of a certain kind of credit card or availability of express shipping. A business process could then be designed in terms of requests to A and decisions based on its responses. If a second service B is built inheriting A interface: in addition to its own operations, it will respond to the A-like requests regarding card acceptance or shipping. Moreover, interface inheritance guarantees that the format of the requests accepted by B is the same as the ones of A. We can then substitute A with B in the business process. In addition, the service B may have further interface elements which do not affect the process.

We identify four different composition scenarios, which differ on the basis of the kind of operations performed and on the relationship between service interfaces. Table 6 summarizes the four scenarios, illustrated in Figures 7–9 and discussed in the reminder of this section. In Table 6 we use two categories for describing composition scenarios. Along the vertical dimension, we discriminate services according to the fact that the composing service presents (or not) to external applications the same interface elements as the composed services. On the horizontal dimension, we differentiate services according to the additional operations that are performed in addition to using the composed service. We define as *value-added* the operations that significantly change the nature of the operation of a composed service, while we define *pass-through* the operations that are only rearranging or reformatting data in addition to activating the composed service operation.

	Value-Added Operations	*Pass-Through Operations*
Same Interface	Sub-class composition	Sub-type composition
Different Interface	Opaque composition	Transparent composition

Figure 6: Composition and Inheritance.

In Figures 7–9, we represent a service with an oval in the diagrams and with capital letters in the text. Elements of the interface (that is, service operations) are represented by small shapes positioned on the oval boundary. Different shapes represent different operations, the same shape in two services indicates that the two services offer the same operation. In the text, interface elements are identified with i_x, i_y, and so on. Arrows represent requests or invocation of service operations. The + inside an oval of a service indicates that the service adds its own processing to a request, instead of just passing it to a composed service operation, possibly with some trivial data transformation. This second case is represented by a line connecting the interface element with the activation of the composed service. We also include simplified UML class diagrams, indicating the object oriented relationship from which we originate our description.

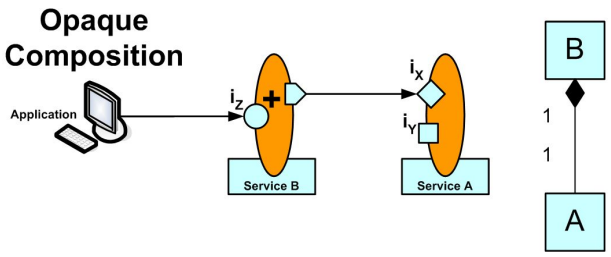

Figure 7: Opaque composition.

In the case of Opaque composition, see Figure 7, service B is composed by another independent services: A. The interface of service B is not related to the one of A. For an external application, there is no indication that B contains service A.

A request i_z to service B will be performed by activating operation i_x in service A. Besides requesting operations to A, B will perform additional work when it receives request i_z.

On the outside of B there is no notion of A operations.

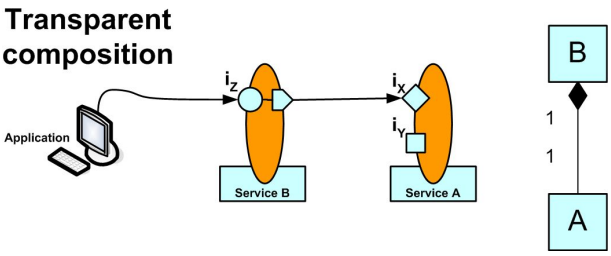

Figure 8: Transparent composition.

Transparent composition, see Figure 8, differs from Opaque composition because B does not process request i_z, but differently from the following cases, B's interface, i_z, is not the same as A's interface, i_x. For this reason, rearranging i_z data to match i_x format does not change the nature of this composition.

For instance, B could be a commercial service which is using A, a credit card validation service. Beside using a validation operation i_x of A, B could offer to external applications a validation operation i_z, using a different name and different parameters from i_x. Upon receiving request i_z, B will reorganize the request parameters and it will in turn ask A to perform i_x.

From the point of view of the external application, there is no connection between operation i_z of B and operation i_x

of A. They just happen to have a similar scope.

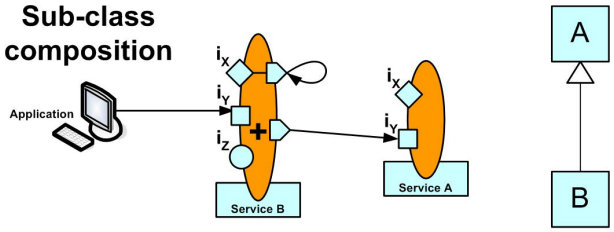

Figure 9: Sub-class composition.

With Subclass composition, see Figure 9, the role of inheritance starts to appear. Since service B inherits service A interface, it has to present to external applications the same interface as A, in addition to its own operations.

In Figure 9, B has i_x and i_y operations, with the same names and parameters as A operations. Since this is a subclass, there is no requirement to guarantee that B will produce the same results as the requests of the same these operations to service A. In fact, B could assign a completely different meaning to these operations.

Since B is not using A, there is no composition between the services. Nevertheless, from the point of view of an external application, B could be treated as an A, since having the same interface it will accept the same requests.

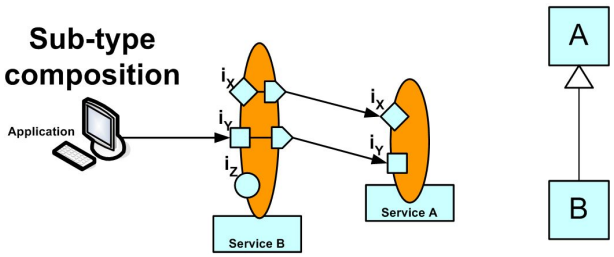

Figure 10: Sub-type composition.

As for Sub-class composition, in the case of Sub-type composition, (Figure 10), service B has the same interface as A, possibly with additional elements. The important difference is that B has to preserve the meaning assigned by A to its own operations.

One possible description of the case in Figure 10 is that B just receives the requests i_x and i_y, passing them on to A. It this way B guarantees that an accessing application will be able to treat B as if it was an A service, obtaining the same results.

Substitution of A with service B is possible also in the previous case, but without being semantically coherent.

4.3 Discussion

The widespread use of composition in systems based on service oriented architectures will ultimately lead to complex business models, relying on advanced service composition functionality. We suggest that the concept of composite services can be extended in a useful manner by allowing access to individual elementary services through interfaces exposed on the composite service. Examples scenarios where this type of extended composition would be useful are location based services (LBS). LBS typically require the integration of at least one service providing positioning data. Consequently every invocation of any composite LBS, like for instance a location based weather service, would also require the invocation of a service providing access to a positioning system, i.e., cellular positioning. If a user makes continuous use of composite LBS, a business model providing cost saving is to allow an already invoked composite LBS to participate in a new composition. In the new composition, the composite LBS would play the role of an elementary service, using only a subset of its functionalities. According to our model, the composite LBS would be used via the positioning system service interface only. In a different scenario, the motivation for this type of extended composition could be the provision of value-added services based on simpler versions provided by elementary components of a composition.

Such business models pose additional requirements for controlling the way the functionalities of the elementary services are composed and made accessible to the composite service consumer. Based on the concept of object-oriented inheritance, and of interface inheritance in particular, we propose a number of extended types of composition, supporting differentiated access modes to the functionalities and interfaces of elementary services.

Transparent, opaque, sub-type and sub-class composition can be compared to public, private and private protected interfaces in object-oriented terms. Much like object-oriented development makes use of such mechanisms to selectively expose interfaces to outside users or direct inheritors of a class, we propose access control mechanisms to achieve similar results when dealing with interfaces of elementary services participating in a more complex composition.

5. CONCLUDING REMARKS

The object oriented paradigm has a solid formal background and is a well-established reality of today's computer science. Service oriented computing is, on the other hand, a new emerging field, which tries to realize global interoperability between independent services. To meet this goal, service oriented technology will need to solve a number of challenging issues, such as how to manage service composition and orchestration. We have proposed a methodology based on model variant configuration by 'borrowing' concepts from the object oriented world. In particular, we have shown how the concepts of interface inheritance induce four forms of service composition.

Future investigation will be pursued in two directions. On the one hand, the utility of the approach will be tested by implementing a tool for designing compositions of services based on the proposed methodology. On the other hand, the added value of semantical enrichments of the interfaces will be investigated.

6. REFERENCES

[1] A. Barr and E. Feigenbaum, editors. *The Handbook of Artificial Intelligence*. Kaufman, 1981–82. Vols. 1–2.

[2] R. Barták. *Week of Doctoral Students (WDS99)*. MatFyzPress, 1999.

[3] BEA, IBM, Microsoft, S. AG, and Siebel. Business Process Execution Language for Web Services, 2003. `http://www-106.ibm.com/developerworks/library/ws-bpel/`.

[4] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. Automatic composition of e-services that export their behavior. In E. Orlowska, M. Papzoglou, S. Weerawarana, and J. Yang, editors, *Int. Conf. on Service Oriented Computing (ICSOC 03)*, LNCS, 2910, pages 43–58. Springer, 2003.

[5] T. Budd. *An Introduction to Object-Oriented Programming*. Addison Wesley, 2002. (3rd edition).

[6] F. Casati, S. Ilnicki, L. Jin, V. Krishnamoorthy, and M.-C. Shan. eFlow: a platform for developing and managing composite e-services. Technical report, Hewlett Packard, 2000.

[7] F. Casati and M. C. Shan. Definition, execution, analysis and optimization of composite E-Services. *Bullettin of the IEEE Computer Society Techincal Committee on Data Engineering*, 24(1):29–34, 2001.

[8] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design*. Addison Wesley, 2001. (3rd edition).

[9] R. Cunis, A. Günter, and H. Strecker. *Das PLACON-Buch. Informatik Fachberichte*. Springer, 1991.

[10] F. Curbera, W. Nagy, and S. Weerawarana. Web services: Why and how. In *Workshop on Obejcet Orientation and Web Services OOWS2001*, 2001.

[11] V. D'Andrea and M. Aiello. Services and objects: Open issues. In G. Piccinelli and S. Weerawarana, editors, *European workshop on OO and Web Service*, pages 23–29, 2003. IBM Research Report. IBM. Computer Science, (RA 220).

[12] S. Davis. *Future Perfect*. Addison-Wesley, 1987.

[13] P. Drucker. *The Practice of Management*. New York: Harper, 1954.

[14] I. Fikouras and E. Freiter. Service discovery and orchestration for distributed service repositories. In E. Orlowska, M. Papzoglou, S. Weerawarana, and J. Yang, editors, *Int. Conf. on Service Oriented Computing (ICSOC 03)*, LNCS, 2910, pages 59–74. Springer, 2003.

[15] I. Fikouras and F. Ramme. Service orchestration with generic service elements. In *Proceedings of the 6th International Symposium on Wireless Personal Multimedia Communications*, 2003.

[16] R. Glazer. Winning in smart markets. *Sloan Management Review*, 40:59–69, 1999.

[17] M. Heinrich. Ressourcenorientieres konfigurieren. *Künstliche Intelligenz*, 7(1):11–15, 1993.

[18] C. Huffman and B. E. Kahn. Variety for sale: Mass customization or mass confusion? Technical Report R98-111, Marketing Science Institute, 2004. `http://www.msi.org/msi/publication_summary.cfm?publication=491`.

[19] B. Kahn. *Dynamic relationships with customers: high-variety strategies*, volume 26. Sage, 1998.

[20] S. McIlraith and T. C. Son. Adapting Golog for composition of semantic web-services. In D. Fensel, F. Giunchiglia, D. McGuinness, and M. Williams, editors, *Int. Conf. on Knowledge Representation and Reasoning (KR2002)*, pages 482–493, 2002.

[21] B. Neumann. Configuration expert systems: a case study and tutorial. In H. Bunke, editor, *Artificial Intelligence in Manufacturing, Assembly and Robotics*. Oldenbourg, 1988.

[22] NOMAD. Ist-2001-33292. Project Web-Site: `http://www.ist-nomad.org`.

[23] B. Orriëns, J. Yang, and M. P. Papazoglou. Model driven service composition. In E. Orlowska, M. Papzoglou, S. Weerawarana, and J. Yang, editors, *Int. Conf. on Service Oriented Computing (ICSOC 03)*, LNCS, 2910, pages 75–99. Springer, 2003.

[24] M. P. Papazoglou and D. Georgakopoulos. Service-oriented computing. *Commun. ACM*, 46(10):24–28, 2003.

[25] M. Papazoglou et al. SOC: Service Oriented Computing manifesto, 2003. Working draft available at `http://www.eusoc.net`.

[26] F. Piller. *Kundenindividuelle Massenproduktion: Die Wettbewerbsstrategie der Zukunft*. Carl Hanser Verlag, 1998.

[27] F. Puppe. Expertensysteme. *Informatik Spektrum*, 9(1), 1986.

[28] P. Schnupp, H. Nguyen, and T. Chau. *Expertensystem-Praktikum*. Springer, 1987.

[29] W. Tank. Wissensbasiertes konfigurieren: Ein überblick. *Künstliche Intelligenz*, 7(1):7–10, 1993.

[30] E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.

[31] M. Tseng and J. Jiao. Mass customization. In G. Salvendy, editor, *Handbook of Industrial Engineering*, pages 684–709. Wiley, 2001.

[32] P. West, D. Ariely, S. Bellman, E. Bradlow, J. Huber, E. Johnson, B. Kahn, J. Little, and D. Schkade. Agents to the rescue? *Marketing Letters*, 10(3):285–300, 1999.

[33] J. Yang and M. Papazoglou. Web component: A substrate for web service reuse and composition. In *CAiSE*, pages 21–36, 2002.