Towards a Unified Conceptual Framework for Service-Oriented Computing

Philipp Bostan, Colin Atkinson
Chair of Software Engineering
University of Mannheim
Mannheim, Germany
{bostan, atkinson}@uni-mannheim.de

Dirk Draheim
IT Service Management Department
University of Innsbruck
Innsbruck, Austria
draheim@acm.org

Abstract — Given the importance of clients in service-oriented computing, and the ongoing evolution of distributed system and application realization technologies from client/service architectures, through distributed-object and service-oriented to cloud computing, there is a growing need to lower the complexities and barriers involved in the development of client applications. These range from large scale business applications and business processes to laptop programs and small “apps” on mobile devices. In this paper we present a unified conceptual framework in which the basic concerns and viewpoints relevant for building clients of service-oriented, distributed systems can be expressed and related to one another in a platform-independent, non-proprietary way. The basic concerns used to structure the framework are the level of abstraction at which a system is represented and the roles from which the software entities of a distributed system are viewed. Using the various models supported in the framework it is possible to customize each client developer’s view and simplify the way in which service providers develop and maintain their services. This paper provides an overview of our framework and show how it could be applied using a small example.

Keywords— distributed computing, model-driven development, service-oriented architectures

I. INTRODUCTION

Since distributed computing and its technologies became a viable solution for building enterprise systems in the late 1980s, there have been several major waves of technologies, paradigms and “best practices” for building and running them. The first major wave, characterized as “client/server computing”, emphasized the separation of GUI, application logic and data persistence concerns. Key ingredients of this wave were transaction monitors [1] and message-oriented middleware (MOM) [2], for example. The second major wave, characterized as “distributed object computing”, emphasized the use of object-oriented abstractions in the construction of distributed systems. Key examples of technologies supporting this approach include CORBA [3] and J2EE [4]. The third major wave, which is arguably still ongoing, is the “service-oriented architecture” wave which focuses on the Internet as the underlying distribution platform. Key examples of technologies in this wave therefore are Web Services and the emerging REST (Representational State Transfer) services paradigm [5]. Finally, the fourth major wave that is just starting to emerge is cloud computing [6]. This emphasizes the flexible and site-transparent distribution of computing platforms and applications over outsourced data center infrastructures. Key examples of technologies in this wave are the Google Cloud, the Microsoft Cloud, Amazon Web Services and the on-demand data centers of companies like IBM and HP.

Whilst this rapid evolution of distributed computing technologies has provided a rich set of platforms and paradigms for building robust enterprise systems, it has also left a legacy of unresolved problems. The first is a trail of confusion and inconsistency in the concepts used to build distributed systems. Even within individual paradigms there are inconsistent interpretations of some of the most basic concepts, such as whether services are (or should be) stateless, for example. Furthermore, between the different paradigms there is little consensus about the core ingredients of distributed systems, for example, what are the differences and relationships between components, services and objects. The second problem is that the evolution of the different distributed system technologies has been overwhelmingly driven by the server-side concerns of the client/server divide rather than the client side. As a result, developers of regular client applications, business processes or mobile “apps” (e.g. Android) typically have to access server-side assets via low-level, platform-specific abstractions optimized for solving server-side problems rather than via those that fit their needs.

The importance of addressing this problem has grown as client applications have become more visible and as they have started to play a major role in the perceived usability of service-based systems. Moreover, in large enterprise system landscapes [7] services are often clients of each other, and experience has demonstrated that the most successful enterprises have paid particularly special attention to the infrastructure features needed to support flexible service usage [8], [9]. To date, however, SOA best practices have primarily focused on only one aspect of usability - the rapid and straightforward integration of arbitrary technologies (including legacy systems) and platforms at the server side of the client/server divide. However, this ease-of-integration at the server side has come at the price of lower flexibility and ease-of-use of distributed software assets at the client side in client application development.

To address the basic problems and to reduce the artificial complexity involved in building client applications there is a general need for a unified conceptual model of service-
oriented computing that supports both, the needs of client developers and service providers. Our premise is that such a conceptual model should be completely independent of, and implementable on top of the different distributed computed paradigms discussed above since these represent ultimately just implementation alternatives. Not only that, it should also be compatible with the implementation technologies and modeling languages used to realize client applications today, such as high-level programming languages or business process modeling languages. Just as high-level programming languages are optimized for human programmers rather than for realization platforms, we believe that the features of such a conceptual model should be determined by what is best for client developers rather than by the idiosyncrasies of individual implementation platforms.

In a previous EDOC paper [10] we have outlined some of the core conceptual ingredients of a client-oriented model of distributed computing assets in an informal way using a motivating example. In this paper we build on this work by presenting a concrete set of metamodels for the different abstraction levels and views involved in a distributed computing landscape and position them within a single unified conceptual framework. Furthermore, we also present two examples of how specific platforms and technologies are accommodated in the framework. The key difference between our framework and other more general conceptual models for distributed computing like RM-ODP, Service Component Architecture (SCA) [11], CORBA Component Model (CCM) [3] is that our conceptual framework –

- has been designed to accommodate the mainstream platforms and paradigms as special cases, and thus includes a transcending set of concepts that can help to unify and systematize the field of distributed computing in general.
- focuses on the needs of client-developers rather than the traditional server-side concerns (e.g. persistence, interoperability, transactions, robustness, etc.) that dominate distributed computing platforms. For client developers, these manifest themselves as non-functional properties (e.g. performance, reliability etc.) that are part of service-level agreements.

The rest of the paper is structured as follows. In the next section we present the overall structure of the framework, the nature of the various ingredients and how they fit together. In Section 3, we present the core metamodel which captures the core transcending concepts to act as the common supermodel for all of the other metamodels. Section 4 then presents the metamodels for the platform-independent views of a service-oriented system. This is followed by the description of five key PIM-level realization patterns in section 5 that can be use to map client-side abstractions to server-side abstractions and vice versa. In sections 6 and 7 we then show how the model can be applied to particular realization platforms (Java for the client-side and Web Services for the server-side) in the context of a small example scenario. Section 8 then discusses some related work and finally Section 9 concludes with some final remarks.

II. STRUCTURE OF THE CONCEPTUAL MODEL

The basic goal behind the unified conceptual model is to provide a framework in which the concerns and viewpoints relevant for building client applications using distributed services can be expressed and related to one another. The basic concerns that are used to structure the framework are the level of abstraction at which a system is represented and the roles from which the components of a system are viewed.

A. Abstraction Levels

The model-driven architecture (MDA) popularized by the OMG [12] represents software systems using multiple levels of abstraction. The most abstract level at the top is the Computation-Independent Model (CIM) in which models of the business processes to be automated or the environment to be enhanced are described independently of the envisaged IT-based solution. On the next level, a Platform-Independent Model (PIM) describes the key ingredients and the behavior of the envisaged system independently of the idiosyncrasies of specific platforms. A Platform-Specific Model (PSM) describes the system in terms of the concepts supported by a specific platform, but not necessarily in a way that is directly executable. In other words, a PSM is still a model even though it is platform-specific. The lowest level of abstraction in the MDA is the executable implementation which does not require any further manual transformations (assuming the appropriate compilers and virtual machines are available). These levels correspond roughly to the classic abstraction levels of model-driven software engineering. Our approach adopts the basic MDA abstraction levels and the associated terminology. However, we focus only on the two central levels (PIM and PSM) since this is where the key separation of concerns as well as the identification of common abstractions takes place.

B. Roles

As the different technologies of distributed computing have evolved from basic client/server approaches to service-oriented architectures the differences between the concerns of clients (i.e. their developers) and the concerns of services (i.e. their providers) have grown. Moreover, these are set to increase even further as distributed computing evolves further into cloud computing. The two fundamental roles of concern therefore are the client developer and the service provider as depicted in Figure 1.

Figure 1. Roles in Distributed Application Development
This shows an enterprise service infrastructure (ESI), implemented and maintained by one service provider and used by two different client developers. A fundamental premise of our approach is that there is a clear boundary to the enterprise system for each stakeholder. The different stakeholders do not necessarily need to agree on the boundary, but they must each have a clear notion of the system boundary. As depicted in Figure 1, client developer A actually belongs to the organization owning the ESI, and thus is represented as being inside the enterprise system boundary, while client developer B is outside the enterprise boundary and uses the services as a usual customer.

C. Two-Dimensional Modeling Space

The overall structure of the unified conceptual model is derived by regarding the abstraction level dichotomy (PIM/PSM) and the dichotomy of the presented roles as distinct, orthogonal dimensions. The basic goal is to address the concerns of both of the roles at the two abstraction levels as illustrated in Figure 2 resulting in four different models that will be explained in the following.

**Service-Oriented PIM (SPIM):** The role of a SPIM is to provide a platform-independent view of a service landscape from the point of view of a service provider. There is therefore no notion of, or support for, clients in the SPIM. The abstractions used in a SPIM are service-oriented, but transcend particular realization technologies. If there are numerous service providers supporting different parts of a single overall landscape, they will have SPIMs tailored to their own particular view of the landscape.

**Client-Oriented PIM (CPIM):** The role of a CPIM is to provide a platform-independent view of a service landscape tailored to the needs of a particular client developer. A CPIM therefore includes the notion of, and support for, clients. However, it also includes abstractions of server-side software entities that a client type wishes to use and access. As with a SPIM, a CPIM transcends particular realization technologies and represents remote software entities independently of their realization.

**Service-Oriented PSM (SPSM):** A SPSM provides a platform-specific representation of the service landscape from the point of view of a service provider. In terms of the traditional MDA paradigm it extends and refines the SPIM using platform-specific abstractions.

**Client-Oriented PSM (CPSM):** A CPSM has the same relationship to a CPIM as a SPSM to a SPIM, representing a refinement of a platform-independent model that adds detail through platform-specific abstractions. Ideally, this platform- specific detail will only relate to the client-side because the server-side abstractions (e.g. services) should be accessible through PIM level abstractions.

The two-dimensional modeling framework as presented in Figure 2 provides four different views each tailored to the respective role and level of abstraction involved. However, not all views are of interest to all the roles. Each stakeholder has a particular constellation of views that reflect his particular concerns depending on the role that he plays, as illustrated in Figure 3 (relating to the scenario in Figure 1).

A service provider is usually only interested in the server-side abstractions, and since in our example depicted in Figure 1 there is only one service provider responsible for the whole ESI, all services are included in this service provider’s SPIM and SPSM views. Client developer A on the other hand, is only interested in his own client-oriented view of the ESI. This is reflected in his CPIM and CPSM models which only contain abstractions of the services he uses - services B and C. Similarly, client developer B needs client-oriented views tailored to his concerns. His CPIM and CPSM models therefore only contain abstractions of services A and B.

D. Metamodel Architecture

Figure 2 showed the four different kinds of models used by the different stakeholders to provide and use services in a distributed system.

These models are not independent, of course, but are related to one another in carefully defined ways. In fact, we believe that one of the main contributions of the unified conceptual framework is to capture the commonalities and differences between these models. As depicted in Figure 4, this is achieved by arranging the models in a specialization hierarchy rooted in a Core metamodel that contains the core abstractions common to all roles and abstraction levels. In this hierarchy, the CPIM and the SPIM metamodel are

---

1 Standard MDA practice would be to define these as profiles, but we use metamodels since these are essentially equivalent but conceptually cleaner. However, to apply a metamodel we use the stereotype guillemot notation.
specializations of the Core metamodel, while specific PSMs are specializations of the respective CPIM and SPSM metamodel.

III. THE CORE METAMODEL

The Core Metamodel (CM) provides the abstractions that span the main roles and abstractions levels in service-oriented computing identified in the previous section. All of the other metamodels therefore inherit directly or indirectly from the CM. At its heart, therefore, are the basic ingredients of imperative computing – data types, processes and objects. The history of software description can be characterized in terms of the evolution in the way that these have been interwoven. In the early days of computing (up to the late 70’s and 80’s), programs and systems were primarily described in a function-oriented way. This was based on the principle of strictly separating the “functions” in a program from the “data types” that they manipulated, and using the relationships between the former to define the architecture of the system. From the late 70’s, the wisdom of this separation started to fall into question and the notion of object-orientation emerged, based on the idea that functions and data should be tightly bound together and encapsulated. Since objects are essentially data-centric, this meant that data played the dominant role in defining the architecture of a program.

Data Type: A Data Type is essentially a set of values whose use is controlled by a number of rules. The instances of a data types are values that do not have inherent identity of their own – they cannot be created or destroyed and cannot be duplicated. Data Type is an abstract class that has two subclasses – Primitive and Compound Data Type.

Primitive Data Type: These are used to represent the classic data types such as Integer, Character, String, etc.

Compound Data Type: These entities are data types whose values are composed of combinations of primitive and/or compound data types, similar to “records” or “structs” as used in programming languages like Pascal or C.

Process: A Process represents a functional abstraction that includes a set of steps arranged in some well-defined order to achieve some effect or to reach a certain goal. It encompasses programming level abstractions such as subroutines, functions, procedures and methods, as well as high-level notions of processes such as workflows and business processes. Since these involve the sequential execution of sub-steps to achieve a goal, processes have an associated notion of procedural or algorithmic state [10] that represents its current progress through the designated sequence of its steps. Processes can have input and output parameters which can be objects or data types and can also use other processes. The abstract class Process is partitioned by the subclasses Free Process and Allocated Process.

Free Process: A Free Process is a process that is not allocated to any particular object. It therefore represents a purely functional abstraction (akin to a function, procedure or subroutine in older programming languages).

Allocated Process: An Allocated Process is a process that belongs to an object. The existence of an instance of an Allocated Process is therefore tied to the existence of the object that it belongs to and it cannot be used independently of that object. Allocated Processes have direct access to the cohesive state of the object they belong to. They therefore correspond to methods in object-oriented programming languages.

Object: An Object represents the basic object abstraction familiar from object-oriented programs and object-based computing in general. The basic characteristic of an object is that it encapsulates one or more attributes of a Data Type behind one or more Allocated Processes (i.e. methods). Like objects from object-oriented programming they have their own unique identity and can be duplicated. Objects therefore unify the core ingredients of function-oriented computing – processes and data types. The key additional idea for service-oriented computing is that objects come in two basic kinds – Ephemeral and Architectural Objects. The class Object is therefore an abstract class since it has no direct instances. Because they encapsulate data values, objects have an associated notion of cohesive state [10], which captures the effects of the operations that have been applied to the object.

As presented in Figure 5, the Core Metamodel contains the following abstractions arranged as classes within a single hierarchy.

Entity: The class Entity is an abstract class that serves as the root of the inheritance hierarchy. Instances of subclasses of the class Entity therefore may represent any kind of entity within a computing system.

![Figure 5. Core Metamodel](image-url)
Architectural Object (AO): Architectural Objects are stable objects whose lifetime normally coincides with the lifetime of the system as a whole because they are the “components” of the system. The only exception is when architectural reconfigurations are performed. Architectural Objects are in general large, behavior-rich objects and are usually few in number. In fact, most of the time there is only one instance of an Architectural Object type in a system. The notions of components, services and distributed objects found in contemporary distributed computing technologies are encompassed by the notion of Architectural Objects.

Ephemeral Object (EO): Ephemeral Objects are objects which have a temporary lifetime with respect to the lifetime of the complete system. In other words, they are frequently generated and deleted during the lifetime of a system and essentially correspond to data entities, although there are related concepts in specific technologies, such as data transfer objects (DTOs) and entity beans in J2EE [4].

The advent of the object abstraction in the late 70’s led to the object-oriented revolution that still dominates many modeling and programming languages today. In contrast, the concepts and practices of service-oriented development can in fact be understood as a move away from object-orientation back to a more traditional function-oriented style of organizing data and processes. However, the confusion surrounding SOA and its relationship to other paradigms demonstrates that SOA’s conceptual model is not optimal for all aspects of distributed computing. The model proposed in this paper can be viewed as an attempt to introduce a more sophisticated, unified approach that can accommodate both function-oriented and object-oriented viewpoints depending on what is optimal for the different roles involved.

IV. PIM METAMODELS

The two PIM-level metamodels, the SPIM Metamodel and the CPIM Metamodel are both extensions of the Core Metamodel that we have described in the previous section. In the following subsections we provide an overview of both.

A. Service-Oriented PIM Metamodel

The SPIM Metamodel extends the Core Metamodel with general, platform-independent concepts for the client-server perspective of the client/server divide. The basic extensions are the introduction of further subclasses of the classes Ephemeral Object and Architectural Object.

![Figure 6. Service-Oriented PIM Metamodel](image)

As presented in Figure 6, the Ephemeral Object class is divided into two subclasses, Data Object and Behavior Rich Object. The former kind represents a type that is purely a wrapper for sets of attributes. In other words, Data Objects essentially wrap up and encapsulate the data contained in a Compound Data Type by shielding the attributes from direct access via setter and getter operations. They are therefore very similar to DTOs of the kind supported in J2EE. By definition, Data Objects therefore only may contain Create, Read, Update and Delete (CRUD) operations and provide no “rich” behavior beyond the getter and setter operations. This is what basically distinguishes a Data Object type from a Behavior Rich Object type. The latter provides extra functionality by methods that calculate new information that is not directly stored in the attributes. This means that these can no longer be regarded as mere Data Object types.

The Architectural Object class from the Core Metamodel is also divided into two subclasses – Observably Stateful Architectural Object and Observably Stateless Architectural Objects. These capture the distinction between observably stateful and observably stateless services that was introduced in [10]. Observably Stateful Objects are objects that have a cohesive state from the point of view of a client. In concrete terms this means that the methods of the object exhibit different behavior for the same set of explicit input parameters depending on the cohesive state of the object. In contrast, Observably Stateless Architectural Objects have no observable cohesive state.

B. Client-Oriented PIM Metamodel

The CPIM Metamodel extends the Core Metamodel with the general, platform-independent concepts for the client perspective of the client/server divide. As with the SPIM, the main extensions are applied to the Ephemeral Object and Architectural Object class of the Core Metamodel, but the Process class is also extended to include the notion of clients that are in general processes, so called Client Processes. The specializations of the Ephemeral Object and Architectural Object classes reflect the different client-oriented properties that we have presented in [10]. As also indicated in Figure 7, Ephemeral Objects are dynamic objects while Architectural Objects are static.

![Figure 7. Client-Oriented PIM Metamodel](image)

The class Ephemeral Object from the core is divided into two subclasses in the CPIM – Private Ephemeral Object and Shared Ephemeral Object. This reflects whether or not
instances of a class can be shared by various client instances of a client type or whether they remain private to a given client type instance.

The two subclasses of Architectural Object – Observably Stateful Architectural Object and Observably Stateless Architectural Object – are both further divided into two subclasses depending on whether they are shared or private similarly to Ephemeral Objects. In fact, this is orthogonal to the property observably stateful/stateless, and could also have been modeled as a separate generalization set.

The only other extension to the PIM is the addition of the Client Application specialization of Ephemeral Object to represent clients that are applications in the traditional software engineering sense. These are a form of Ephemeral Object because they have an identity, a cohesive state and a lifetime that is often much lower than the lifetime of the system that manipulates them.

V. PIM LEVEL REALIZATION PATTERNS

In this section we define some basic realization patterns which represent mappings between PIM-level abstractions (either in the CPIM or the SPIM). These patterns represent the core strategies used to switch different perspectives of abstractions of a system’s entities. They can also be applied to specializations of the affected abstraction (i.e. at the PSM level) as a practical realization step. However, the long term vision is that the mappings will be performed automatically by (i.e. they will be built into) the technologies used to write client applications to provide the client with an alternative view, and vice versa. These patterns are not necessarily applicable in isolation and are meant to define mapping principles rather than complete solutions. In the long term, we hope they will become part of the vocabulary of service-oriented computing, and will help to reduce the levels of confusion and artificial complexity that exists in SOA development today. Each of the following patterns can theoretically be applied in both directions. However the names of the patterns as presented in Figure 8 reflect one particular direction that we present in this section.

Data Type Reification: This pattern maps a Data Type to an Ephemeral Object. It essentially “reifies” a pure Data Type whose instances represent collections of values, into an Ephemeral Object type (i.e. a class) whose instances represent objects that can store the corresponding values, but as private attributes that can only be accessed by setter and getter operations. In non-technical terms, it is used to turn a naked record into an encapsulated object. This pattern can be used to provide an object-oriented view of the structures (e.g. XML documents) communicated by Web Services over the internet. This essentially corresponds to the idea of DTOs in J2EE [4].

Ephemeral Object Externalization: This pattern maps an Ephemeral Object type to an Architectural Object type. For a given type of the former it defines an equivalent type of the latter that supports the same operations, but with an additional parameter for the identifier of an instance of the Ephemeral Object class. This extra information is needed because a single instance of an Architectural Object type is responsible for storing the state of all instances of the Ephemeral Object type, and these need to be distinguishable. In a sense, therefore, this pattern delegates the responsibility for the cohesive state and functionality of an Ephemeral Object type to an Architectural Object.

Figure 8. Realization Patterns

Ephemeral Object Manager: This pattern also maps an Ephemeral Object type to an Architectural Object type. In fact, it is essentially an extension of the previous pattern. As well as mapping the operations of an Ephemeral Object type to the Architectural Object, with the extra parameters and/or return values, it also adds creation and deletion operations to complete the CRUD functionality, as well as search operations to find instances of the Ephemeral Object based on one or more of its attributes.

Process Externalization: This pattern maps a Process type to an Architectural Object type. A single Process entity, with a given set of input parameters and/or result is mapped to an Architectural Object type with a re-entrant (or idempotent) operation corresponding. Any client wishing to invoke the process therefore simply calls this operation on the singleton instance of this type that is created in a system. How the operation works is completely invisible to the clients of the process. In a sense, therefore, this pattern delegates responsibility for the functionality and algorithmic state of multiple instances of a given Process type to a single instance of an Architectural Object type.

Process Manager: This process pattern also maps a Process type to an Architectural Object type. The difference is that, like the Ephemeral Object Manager pattern, this
pattern provides CRUD operations to create, manipulate and destroy process instances. This requires individual process instances to be identified by a unique identifier (ID) that is passed to or returned by invocations of the CRUD operations. While the Ephemeral Object patterns handle the cohesive state of instances of the Ephemeral Objects, Process patterns handle the algorithmic state of Processes.

These patterns capture some of the strategic decisions that have to be made to realize service-oriented systems, regardless of the specific technology used to implement them. They are therefore strategic platform-independent decisions. It is also important to note that clients are processes or objects, depending on their exact nature. Thus, when the appropriate manager pattern is applied, the algorithmic or cohesive state that is being maintained by the architectural object is the state of the client – in other words, the state of the conversation between the client and an instance of an Architectural Object type. Such conversations are often called sessions and the corresponding identifiers sessions IDs.

VI. EXAMPLE SPSM MODELS

In this section we provide two examples of SPMS models for specific client- and server-side implementation technologies. We therefore consider the most widely known service realization technology – Web Services, and one of the most widely used client realization technologies – Java.

A. Web Service SPSM Metamodel

The first generation of Web Service standards (WSDL and SOAP) and several extensions of these represent one of the most well-known and widely used service realization technologies. However, not all of the concepts of the SPIM are directly supported, so the Web Service SPMS actually restricts the use of some of the concepts in the SPIM. Also, one of the functions of an SPMS is to indicate where a SPIM abstraction has a different name in a specific realization technology. This is achieved by placing the platform-specific name in parentheses after the platform-independent name.

This renaming is most evident in the case of Data Types. Web Services basically support both kinds of Data Types as XML data types, as shown in Figure 9. This SPMS also indicates that Architectural Objects are referred to as services in Web Service technology terms. Both forms of Architectural Objects are also supported, but Observably Stateful Architectural Objects are further divided into two subclasses, Inherently Stateful Architectural Objects and Inherently Stateless Architectural Objects. This distinction reflects the fact that Architectural Objects (i.e. Web Services) that appear to be stateful to clients (i.e. that are observably stateful) may not in fact have any direct state of their own from an implementation point of view, but may delegate the storage of this state to a third party (e.g. a database) that is not visible to the client. Such types of Architectural Objects are referred to as Inherently Stateless Architectural Objects. In contrast, an instance of the type Observably Stateful Architectural Object may indeed encapsulate the state that it exposes to clients. As discussed in [10] the failure to distinguish between these two properties (observable versus inherent) statefulness is the root of much of the confusion surrounding this issue in contemporary service-oriented technologies. By providing an explicit model of the distinctions and relationship between these two concepts and allowing Architectural Objects to be characterized accordingly, the state-related behavior of services can be much more clearly understood by client developers and service providers alike.

Figure 9. Web Service SPSM Metamodel

Another restriction to the SPMS relates to the distinction between Allocated Process and Free Processes. The Web Service standard does not recognize the concept of free processes and requires all processes to be allocated to objects. The Web Service SPMS Metamodel in Figure 9 therefore indicates that an Allocated Process corresponds to a Web Service operation.

B. Java CPSM Metamodel

To illustrate an example of a CPSM metamodel, we have chosen the Java platform that offers one of the most widely used programming languages for writing client applications.
As a mainstream object-oriented language for regular GUI driven programs that run on laptops and PCs and more recently in terms of applications for the rapidly expanding Android market, Java is able to support all of the concepts of the CPIM in a fairly direct way as presented in Figure 10. The only entities from the CPIM that Java does not support are Compound Data Types and Free Processes since data composition and behavioral abstractions are achieved using objects. Therefore, the Java CPSM only contains the process type Allocated Process that plays the role of an operation of a Java class. The type Ephemeral Objects corresponds to a regular Java class, while the type Architectural Object can conveniently be realized as a static Java class, since only one instance is required.

VII. Example Scenario

To provide a small example of how our framework may be applied in practice in a service-oriented application, in this section we show different models for the Mortgage Sales scenario that we introduced in [10]. This represents a scenario that provides IT support for mortgage salesmen working for a bank or credit company. As discussed in the following, this example consists of two client applications and contains a typical mix of different kinds of objects and relationships.

The first client application (Salesman client) supports the salesman responsible for interacting with customers. The salesman can therefore use his client application to create multiple mortgages for a certain customer at the customer’s site and store these for further elaboration in a dictionary. Finally, he submits a mortgage proposal to a priority queue for further processing by financial experts. A second client application (Financial Expert client) is used by the financial experts who are responsible for analyzing proposed mortgages and sending notifications to customers about the acceptance or rejection of mortgage proposals. She or he therefore uses her or his client application to fetch mortgage proposals from the common priority queue to be checked. After a validation, a financial expert sends the customer and the salesman a notification about his decision.

A. PIM-level models

In Figure 11, a CPIM model which captures these types from the perspective of the Salesman client application type is presented.

![Figure 11. Salesman client CPIM](image)

It consists of five classes that are of importance from the client type’s perspective, three of them are of the type Ephemeral Object (EO) whose instances are created and deleted as the system operates, and two of them are of the type Architectural Object (AO) representing permanent instances. Furthermore, the stereotypes indicate that both Architectural Object types are observably stateful, but that one of them, the PriorityQueue type, is shared between multiple instances of different client types, while the other type, the Dictionary is private to an instance of the client type Salesman. Similarly, two of the Ephemeral Object types are shared, while one type is private.

Next we consider the CPIM for the Financial Expert client type, presented in Figure 12. This indicates that there are only four classes of importance, three Ephemeral Object types as well as one Architectural Object type. All but one of them, Notification, also appears in the Salesman client CPIM, with the same properties. The stereotype of this new class, Notification, indicates that this is a private Ephemeral Object type, accordingly.

![Figure 12. Financial Expert client CPIM](image)

After presenting the CPIM models for the Mortgage Sales scenario, we next introduce a SPIM model for the scenario from the perspective of a single service provider working for the same organization developing the two client applications. As shown in Figure 13, any object that is either part of the Salesman or the Financial Expert client models is also in the Mortgage Sales SPIM model. As can be recognized, it contains several Data Ephemeral Object types, one Behavior-Rich Ephemeral Object type as well as two Observably Stateful Architectural Object types.

![Figure 13. Mortgage Sales SPIM](image)

B. PSM-level models

After the introduction of the PIM-level models for the Mortgage Sales example scenario, this subsection presents a subset of PSM-level models for the Mortgage Sales scenario using Java as a platform for an example of one client-
oriented PSM model and Web Services as the platform for the service-oriented PSM model.

First, we consider the Salesman client Java CPSM as presented in Figure 14. Note, that we omit the CPSM of the Financial Expert client for space reasons.

According to the Java CPSM metamodel that we have presented before, Figure 14 contains the same entities as specified in the Salesman client CPIM, but replaced by entities of the Java CPSM. Note, that the properties shared and private are removed in the CPSM since the handling of these issues is deferred to the underlying infrastructure (which is aware of the PIM-level specifications) the applications are cooperating with when being executed. The most important issue here is that Ephemeral Object types are represented by regular Java classes and Architectural Object types are represented by static Java classes in the CPSM to reflect and indicate their behavior the client developer.

Finally, to complete the Mortgage Sales example scenario and to cover the full spectrum of models proposed in this paper, in Figure 15, we present a SPSM based on the Web Service SPSM metamodel that we have introduced in section VI. We therefore apply some of the patterns that have been proposed to map the presented PIM-level abstractions of the SPIM to PSM-level abstractions in the SPSM.

As Ephemeral Objects are not supported in the Web Services SPSM metamodel, we have to apply one of the proposed patterns of section V to be able to create a model on the level of the Web Services SPSM. For reasons of space we pick out two examples at this point. The first is the Customer entity that has been mapped from an Ephemeral Object type to an Architectural Object type applying the Ephemeral Object Manager pattern. We have chosen to provide this entity as an Inherently Stateless Architectural Object in this example. As a second example we pick out the PriorityQueue type that remains an Architectural Object as specified in the SPIM already. For the PriorityQueue we have chosen to provide it as an Inherently Stateful Architectural Object type whose instances maintain their state on their own, while the previous example, the Customer, defers state to an external participating resource like a database for example.

VIII. RELATED WORK

Since Service Oriented Computing (SOC) and Model-Driven Architecture (MDA) are such important paradigms, there has naturally been a great deal of interest in using them together, and over the last decade a number of high profile proposals have been put forward. These range from lifecycle spanning, industrial scale methods such as SOA [13] and SOMF [14] to more academic contributions such as [15] and [17]. As a dedicated language for modeling SOAs, the Service oriented architecture Modeling Language (SoaML) has been standardized by the OMG in [16].

The Service-Oriented Modeling and Architecture (SOMA) approach published by IBM in 2004 [13] was one of the first fully fledged methods targeted at service-oriented architectures. However, it is a very broad spectrum methodology that covers many more aspects of the development life cycle than just the modeling and documentation of the service in a service-oriented architecture. The same goes for the Service-Oriented Modeling Framework (SOMF) developed by Michael Bell [14]. It covers everything from the modeling of business goals and processes to enterprise architectures.

Probably the method most focused on supporting the UML-based modeling of service-oriented architectures per se is the method of Piccinelli and Skene [17], which focuses on the modeling of Electronic Service Systems (ESSs). This uses a mixture of metamodels and profiles (the ESS profile) to support two views of services – one (the capability view) showing how services are composed from capabilities and the other (the implementation view) showing how business assets are assigned to capability roles to make services concrete. However, both views are highly abstract and platform-independent, and provide no support for modeling SOA’s realized in a particular execution platform. The method of Lopez-Sanz et al. [15], also aims to support the modeling of service-oriented software architectures using MDA principles. It does this within the context of the MIDAS model-driven methodological framework by defining a single metamodel that consists of typical concepts from service-oriented computing. Like [17], however, it also focuses exclusively at the PIM level only from a single perspective.

The big difference between these approaches and the unified conceptual model presented in this paper is that they focus on modeling services at only one level of abstraction (PIM level) and usually only from one perspective. Where more than one perspective is supported (e.g. in [17]) these are not focused on the service client or service provision perspectives. The approach presented in this paper is unique in –
1. Highlighting the four possible combinations of the PIM versus PSM perspectives and the service-client versus service-provider perspective as the four most important viewpoints from which to visualize service oriented architectures,
2. Identifying a core set concepts common to all these viewpoints,
3. Specializing these core concepts into four distinct metamodels optimized for the modeling of service-oriented architectures from the point of view of each distinct viewpoint and stakeholder.

IX. Conclusion

In this paper we have presented a unified conceptual framework for describing the ingredients of service-oriented computing systems at two key levels of abstraction (platform-independent and platform-specific) and from the perspective of two key roles or stakeholders (client developer and service provider). By separating concerns in this way it is possible to support distinct views of a system which are customized for the different stakeholders in a particular application optimized for their specific needs. In particular, client developers can be provided with an object-oriented viewpoint of the abstractions involved in a particular business process or application, and can thus be relieved of the burden of writing code (or process specifications) to interact with services at the level of abstraction that was optimized for maximum interoperability rather than for usability by clients. The contribution of this approach lies not only in the shape and structure of the overall modeling framework but also in the separation of concerns and identification of common abstractions that is reflected in the contents of the various metamodels. This is not only the prerequisite for integrating most of the current distributed computing and client programming technologies into the framework in the form of customized platform-specific specializations of the common abstractions, it also facilitates the definition of the basic realization patterns which can be used to support the core client-oriented abstractions on top of all compliant server-side technologies.

Many of the concepts may seem similar to the CORBA [3] and J2EE [4] frameworks, because these also tried to facilitate distributed computing in terms of object concepts. However there are two main differences to our approach. The first is that we focus on client-side issues, and on reducing the artificial complexity experienced by client developers when accessing service infrastructures. Second, CORBA and J2EE both focus on the forward-driven construction of distributed systems (from requirements to components) whereas we place equal, if not more, emphasis on the reuse of existing server-side assets (from components to requirements).

Our approach is related to and builds on existing conceptual models for distributed computing such as RM-ODP and COSMO [18] as well as on general component models such as CORBA Component Model [3] and the Service Component Architecture (SCA) [11]. The main difference is that our approach incorporates different views that support the creation of models that are customized for the different stakeholders. In the coming months we plan to expand the number of concrete implementation platforms incorporated into the framework (as platform-specific models) and to apply the approach to a wider number of case studies.

REFERENCES