

# A Lightweight Formal Framework for Service-Oriented Applications Design

Aliaksei Yanchuk, Alexander Ivanyukovich, and Maurizio Marchese

Department of Information and Communication Technology,  
University of Trento, I-38050 Povo (Tn), Italy  
aliaksei.yanchuk@gmail.com, a.ivanyukovich@dit.unitn.it,  
maurizio.marchese@unitn.it

**Abstract.** Leveraging service oriented programming paradigm would significantly affect the way people build software systems. This paper contributes to the above goal proposing a lightweight formal framework capable of capturing the essential components of service-oriented programming paradigm.

## 1 Introduction

The increasing complexity of the software systems has constantly led to the evolution of new programming paradigms: from functional, to object-oriented, to component-oriented, to service-oriented to name a few. Typically each successive paradigm has introduced new design approaches at an higher level of abstraction, encapsulating and sometime adjusting underlying levels. Service-oriented programming paradigm has naturally focused on the next level of abstraction over object- and component-oriented ones [1]. Established and mature paradigms are supported by well-defined analysis and design methodologies (e.g. UML notation) and supporting tools (e.g. Rational Rose). Such methodologies and tools have emerged and have become highly usable and effective due to a significant effort towards the formalization of the underlying fundamental concepts of object-oriented and component-oriented paradigms, together with an evolving and shared understanding of the abilitating technologies.

Although, some foundational concepts of service-oriented design are starting to be addressed, [2, 3, 4], proper mathematical foundations and service-oriented formalized principles and concepts are still lacking. We think that such formalization is crucial for the identification of suitable software design methodologies and supporting tools capable to meet the specific challenges of service oriented applications, e.g. composability, adaptability and platform independence.

This paper contributes to the above effort by proposing a lightweight formal framework capable of capturing the essential components of service-oriented programming paradigm. Our approach is based on the critical assessment of existing design formalization techniques, mainly in the object and component oriented programming domains. Formalization in these software paradigms covers aspects mainly related to system refinement (such as modules composition techniques,

operations parallelism and analysis of intrinsic constraints in distributed systems. Such formalization is grounded on refinement calculus [5] through the use of refinement techniques to the most used methods for monotonic composition of programs (namely procedures), parallel composition and data encapsulation [6]. Parallel actions in software systems were modelled in [6] by their atomic representations, allowing to utilize methods originally developed for sequential systems. A mathematical foundation for object-oriented paradigm is presented in [7], where message-based automata for modelling object behavior in terms of cleanroom software engineering methodology [8] is presented. There software refinements is approached through a mathematical description of all possible transformations, capable to ensure refinement correctness with respect to other software objects.

In [9], a descriptive functional semantic for the component-oriented design is proposed and used for the definition of a formal model for the interfaces of the components. This work investigates the relationships between compositional operators for synchronous and parallel components designs and system refinement techniques. In contrast to previously referenced works, this component-oriented design approach operates with a black-box interface view on the system's components. Further research in component-based design [10] has led to precise definition of components through their behavioral characteristics as well as to the introduction of parallel composition techniques with feedbacks, enabling modelling of concurrent execution and interaction. However, functional time dependency introduced in [9, 10] does not take into account possible temporal execution of the functionality specified within the interfaces' contracts, but rather limits itself to input/output interrelations. It is important to note that the support for such temporal execution sequences is particularly important in service-oriented applications.

The present paper leverages from the above research on software design formalization approaches and aims to extend them to service-oriented programming paradigm. The structure of the paper is organized as follows. In Section 2 we briefly discuss existing approaches to Service-Oriented Architectures (SOA) and their main components. We then propose formal definitions SOA main components, namely: service, service-oriented environment, service-oriented application. Further elaboration of these models with respect to data transition properties allowed us to introduce a classification scheme for service-oriented applications in Section 3. Conclusions and future work close the paper.

## 2 Formal Foundation for Service-Oriented Applications Design

Service-Oriented Architectures (SOAs) [1, 11, 12] are emerging to support the specificity of service oriented applications. In a SOA, the software resources are considered "services," which are well defined, self-contained, and are independent of the state or context of other services. Services have a published interface and communicate with each other. The basic SOA defines an interaction between

software agents as an exchange of messages between service requesters and service providers. This interaction involve the publishing, finding and binding of services. The essential goal of a SOA is to enable general-purpose interoperability among existing technologies and extensibility to future purposes and architectures. Simply put, an SOA is an architectural style, inspired by the Internet and the Web, for enabling extensible interoperability. In our opinion, the “basic SOA trinity” of a service, broker, and client doesn’t display enough features to capture all service-orientation features . It is rather a *platform pattern*, that is used to design robust, essentially distributed applications and environments. Complimentary to the platform pattern, a service orientation *principle* may be formulated as “a set of computing capabilities of a service-oriented environment for any given moment  $\tau$ , determined by the kind of the dynamically available (deployed) services”. Particular set of conventions for software designed for such environment makes up particular Service-Oriented Architectures.

From the above reasoning, we propose that:

$$SOA = Principle + Platform \quad (1)$$

The importance of this statement emerges in the context of Enterprise Application Integration in large organization: in fact it is practically impossible to provide a universal platform that would strike a perfect fit for all tasks. On the other hand, the service orientation principle enables different products to be designed independently but ensuring their potential integration viability. We believe that in order to fully exploit SOA the following entities must be considered on the same importance level in a conceptual framework for service orientation: individual services, service-oriented environments, and service-oriented applications. In the next sub-sections, we provide a formal definition for the proposed entities in our conceptual framework.

## 2.1 Logical Service, Service-Oriented Environment and Service-Oriented Application

In our framework, a given logical service  $i$  is deployed into an environment to provide the useful functionality  $f_i$ , expressed as a programmatic interface  $I_i$ . Important feature of a service is its capability to interact dynamically, in the given environment, with other services and non-service entities (such as end users).

Logical service’ implementation is thus a set of coordinated and interacting processes:

$$S_i = \langle P_1^i, P_2^i, \dots, P_n^i, \Lambda \rangle, \quad (2)$$

where  $S_i$  — logical service instance,  $P_k^i$  —  $k^{\text{th}}$  process implementing the logical service functionality  $f_i$  through the programmatic interface  $I_i$ , and  $\Lambda$  — network communication function between individual processes.

Service-oriented environment consists of a finite countable set of all accessible logical services implementation for the given moment of time  $\tau$ :

$$Env_\tau = \langle S_1, S_2, \dots, S_n \rangle, \quad (3)$$

where  $n$  — number of logical services deployed in the environment.

Overall functionality  $F$  of a service-oriented application  $A$  is determined by the logical services involved in the provision of the application in a given environment for a given moment of time  $\tau$ :

$$F_A = \langle S_1^A, S_2^A, \dots, S_n^A \rangle_{Env_\tau} \quad (4)$$

Moreover, we introduce the *Application Functionality directing graph*, defined as:

$$V_A = (F_A, G) \quad (5)$$

having vertexes from the  $F_A$  set and verge set  $G$  formalizing the coordination between individual logical services of the  $F_A$  set.

Finally, the Service-Oriented application  $A$  may be formalized as the set:

$$A = \langle F_A, V_A \rangle \quad (6)$$

The defined service-oriented application  $A$  is characterized by the following properties:

- to achieve computation goal, at least two logical service must be involved (otherwise *SOA* degrades to client-server architecture);
- services involved in the application must “coordinate” their work to solve the computational problem. Here, we mean “coordination” as any kind of interaction between involved services that aims to achieve the application goal. Coordination may be implemented by different means, including but not limited to, exchanging data, exchanging messages, service provisioning, service control, service monitoring etc.

The presence of the coordination capability is required in a service-oriented application due to the fact that a consistent implementation of a service must be designed to be context-invariant [12]: i.e., particular service must not have knowledge about the application it participates in. In our framework, the Application Functionality directing graph,  $V_A$ , defined in equation 5, is the formalization of such capability. In concrete application,  $V_A$  may be expressed with existing implementation frameworks for capturing services processes, such as BPEL<sup>1</sup> and WS-Coordination.<sup>2</sup>

### 3 The Application Class Concept

In order to ground a service-oriented design methodology on our proposed framework, the following steps must be considered: first application requirements are collected to drive functionalities definition; then application’s functionalities are decomposed into individual services; thus the appropriate Application Functionality graph is created; finally to realize a concrete application  $A$ , both service functionality and directing graph must be implemented.

<sup>1</sup> <http://www-106.ibm.com/developerworks/library/ws-bpel/>

<sup>2</sup> <http://www-106.ibm.com/developerworks/library/ws-coor/>

In the following, we introduce the concept of the service-oriented application "class" in order to capture general properties of  $A$  and to support the system architect to devise appropriate design strategies (i.e. design patterns, implementation templates, etc..).

In general, the computational task of a given service-oriented application is to process all incoming requests (tuples) from a set  $T_{in}$  in such a way that processing will comply with requirements determined by specified Quality of Service agreement set,  $QoS$  [13]. The feasibility of a given  $QoS$  requirements set depends on the implementation of the individual services involved in the service-oriented application  $A$ , on the implementation of the Application Functionality graph  $V_A$  and on the global service environment  $Env_\tau$ .

In our opinion the structure as well as the access method of the application's memory plays a major role in determining the types and sub-types of a specific application. For any service-oriented application,  $A$ , there is a data space,  $\hat{T}_A$ , containing all data of the application

$$\hat{T}_A = M_A \cup T_A \cup \psi_A \quad (7)$$

where  $M_A$  — tuple set capturing temporary application memory,  $T_A$  — persistent application memory, and  $\psi_A$  — virtual tuple set, encapsulating all tuples that were deleted from temporary and persistent memories.  $T_A$  includes all those tuples that were delivered to the  $A$ , except for those that left it (passed on further or discarded), *having durable state* — a state having impact on business processes. Data integrity [14] implies that, during application runtime, the incoming tuples  $T_{in}$  set is reflected on entire set  $\hat{T}_A$ , that is:

$$\forall t_i \in T_{in} : \hat{t}_j \in \hat{T}_A, t_i \rightarrow \hat{t}_j \quad (8)$$

For any application  $A$ , one or more data entry and data exit<sup>3</sup> points may be established. To implement the specified functionality of the service-oriented application  $A$ , the following two main scenarios are possible and we use then to define the basis *classes* of a service-oriented application:

- Each tuple  $t_i$  should pass a handling path through number of services. To achieve this, it is sufficient that the service-sender would be able to pass the tuple  $t_i$  to service-recipient. These type of applications constitutes the *flow* class service-oriented application.
- Each tuple  $t_i$  is saved in a shared data space where it is simultaneously accessible to all services that would be involved in the tuple handling; in this case, the key task of such *cooperation* class service-oriented application is establishing unambiguous access sharing to the  $t_i$  tuple and mutual coordination.

Some real applications may find it necessary to combine features of both classes. With such application, *fork* and *join* points may be established in the

<sup>3</sup> Except for the cases where application is designed to retain data indefinitely — see accumulating applications as defined in [15].

Application Functionality graph. Fork point is transfer of tuple from exclusive service' memory into shared data space, and join point is tuple transfer from shared tuple space into service' exclusive memory.

## 4 Conclusions

In this paper we have extended the definition of SOA and we have proposed a lightweight formal framework capable of capturing SOA main components. This formalization allows us to explore the structure of a SOA and to introduce a service-oriented application classification schema. In particular tuple access methods (exclusively owned/ shared) lead to establishing two main classes of service-oriented application: the flow-class and the cooperation-class. However much more work must be done and is currently in progress. In particular, in-depth exploration of the introduced classification, class topologies patterns, QoS aspects of SOA, patterns for SOA. A more exhaustive report on current work can be found in [15].

## References

1. Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services Concepts, Architectures and Applications*. Springer, 2004.
2. Mike P. Papazoglou and Jian Yang. Design methodology for web services and business processes. In *TES '02: Proceedings of the Third International Workshop on Technologies for E-Services*, pages 54–64, London, UK, 2002. Springer-Verlag.
3. R.M. Dijkman and M. Dumas. Service-oriented design: A multi-viewpoint approach. *International Journal on Cooperative Information Systems*, 13(14):338–378, December 2004.
4. Dick Quartel, Remco Dijkman, and Marten van Sinderen. Methodological support for service-oriented design with isdl. In *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*, pages 1–10, New York, NY, USA, 2004. ACM Press.
5. R. J. R. Back. Correctness preserving program refinements: Proof theory and applications, 1980.
6. K. Sere and R. J. R. Back. From action systems to modular systems. In M. Bertran M. Naftalin, T. Denvir, editor, *FME'94: Industrial Benefit of Formal Methods*, pages 1–25. Springer-Verlag, 1994.
7. B. Rumpe and C. Klein. Automata describing object behavior, 1996.
8. D. Craigen, S. Gerhart, and Ralston T.J. An international survey of industrial applications of formal methods. Technical report, National Technical Information Service, Springfield, VA, USA, 1993.
9. Manfred Broy. Towards a mathematical concept of a component and its use. *Software - Concepts and Tools*, 18(3):137–, 1997.
10. Manfred Broy. Compositional refinement of interactive systems modelled by relations. *Lecture Notes in Computer Science*, 1536:130–149, 1998.
11. Mike P. Papazoglou. Service-oriented computing: Concepts, characteristics and directions. In *WISE '03: Proceedings of the Fourth International Conference on Web Information Systems Engineering*, page 3, Washington, DC, USA, 2003. IEEE Computer Society.

12. by Douglas K. Barry. *Web Services and Service-Oriented Architecture: The Savvy Manager's Guide*. Morgan Kaufmann Publishers, 2003.
13. James Webber Ph.D. Sandeep Chatterjee Ph.D. *Developing Enterprise Web Services: An Architect's Guide*. Prentice Hall PTR., 2003.
14. Robert W. Taylor and Randall L. Frank. Codasyl data-base management systems. *ACM Comput. Surv.*, 8(1):67–103, 1976.
15. A. Yanchuk, A. Ivanyukovich, and M. Marchese. Technical report *dit-05-059*: Towards a mathematical foundation for service-oriented applications design. Technical report, Department of Information and Communication Technology, University of Trento, <http://eprints.biblio.unitn.it/>, 01 July, 2005.