Towards a Mathematical Foundation for Service-Oriented Applications Design

Aliaksei Yanchuk, Alexander Ivanyukovich, Maurizio Marchese
Department of Information and Communication Technology, University of Trento, Trento, Italy, USA
Email: 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. However, to achieve this goal a solid software design methodology should be grounded on proper mathematical foundations, specific service-oriented principles, concepts and patterns. This paper contributes to the above goal proposing a lightweight, but complete, mathematical framework capable of capturing the essential components of service-oriented programming paradigm. To this end, we propose mathematical definitions for individual service, service-oriented environment and service-oriented application. Analysis of the properties and the functionalities of these components with respect to data processing mechanisms enables us to introduce a service-oriented application classification schema. For each application class we first identify specific properties and then discuss their use in a service-oriented design methodology.**

Index Terms— **Service Oriented Architecture, Service Oriented Applications, Service Oriented Design**

I. 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 programming paradigms [1]. The latter 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.

In recent years, service-oriented applications are rapidly becoming the de-facto standard for distributed enterprise in supporting collaborative business processes. At present, most enterprises have gained initial experience in deploying predominantly internal business applications by consuming service-oriented applications developed either in-house or offered by third parties [2]. However, so far the main focus has been mostly in assembling applications by consuming web services or by service enabling legacy applications and enterprise information systems, e.g., ERP-systems such as SAP and Oracle Applications.

A common concern is that current emphasis is on such assembling of service-oriented applications rather than on the design principles that guide the development of services, the granularity of services or the development of the components that implement them. One question arises naturally: what is the proper way to design service-oriented applications so that they can be efficiently integrated in business applications, can be assembled into useful business processes, can be managed, reused, priced and metered?

Although, some foundational concepts of service-oriented design are starting to be addressed, [3]–[5], proper mathematical foundations and service-oriented formalized principles and concepts are still not in place.

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, but complete, mathematical framework capable of capturing the essential components of service-oriented programming paradigm. Moreover, based on the analysis of our formalized model we derive and discuss substantial properties for these components, as well as connect them to concrete applications requirements and practical issues.

In Section 2 we briefly discuss existing approaches to Service-Oriented Architectures (SOA) and their main components relevant to our mathematical formalization. In Section 3 we propose our definitions for a mathematical model of SOA main components, namely: service, service-oriented environment and 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. We identify and discuss two main classes, useful in designing practical applications, namely: flow-class and collaboration-oriented class applications. In Section 4, we look into each application class and introduce further

This paper is based on ”A Lightweight Formal Framework for Service-Oriented Applications Design”, by A. Yanchuk, A. Ivanyukovich and M. Marchese which appeared in B. Benatallah, F. Casati, and P. Traverso (Eds.): ICSOC 2005, LNCS 3826, pp. 545551, 2005. © Springer-Verlag Berlin Heidelberg 2005 refinements based on key transition properties. For flow-class applications we examine the structure of application memory; for cooperation-oriented application we examine most commonly used techniques to organize collaboration. In Section 5 we discuss related work. Conclusions and future work close the paper.

II. SERVICE-ORIENTED ARCHITECTURES: DEFINITIONS AND COMPONENTS

Service-Oriented programming paradigm is an approach to loosely coupled, standards-based, and protocol-independent distributed computing, where coarse-grained software resources and functions are made accessible via the network. Service-Oriented Architectures (SOAs) [1], [6], [7] 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. Services that utilize Web services standards (WSDL, SOAP, UDDI) are the most popular type of services available today. 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. SOA's loose-coupling principles - especially the clean separation of service interfaces from internal implementations - can be used as a guide to plan, develop, integrate and manage enterprise-wide and cross-enterprise applications.

In fact, SOAs provide an architectural shape in which business processes, information and enterprise assets can be effectively (re)organized and (re)deployed to support and enable strategic plans and productivity levels that are required by competitive business environments. New processes and alliances need to be routinely mapped to services that can be used, modified, built or syndicated. In addition, business processes need to be easily designed, assembled, and modified. To achieve such requirements, the internal architecture of an SOA evolves into a multi-tier, service-based system, often with a diversified technical implementation. This diversity is the result of a very broad spectrum of business and performance requirements as well as different execution and reuse context.

With the automation available today to produce service wrappers around pre-existing components, such as Commercial-Off the-Shelf (COTS) package components, it is quite tempting to treat Web services like any normal component. In fact, many enterprises in their early use of SOA, think that they can port existing components to act as Web services just by creating wrappers and leaving the underlying component untouched. Since component methodologies focus on the interface, many developers assume that these methodologies apply equally well to service-oriented architectures. Thus, implementing a thin SOAP/WSDL/UDDI wrapper on top of existing applications or components that implement the Web services is by now widely practiced by the software industry. Yet, this is in no way sufficient to construct commercial strength enterprise applications. Unless the nature of the component makes it suitable for use as a Web service - and most have not been built for that - it takes serious thought and redesign effort to properly deliver components functionality through a Web service. While relatively simple Web services may be effectively built that way, a methodology is of critical importance to specify, construct, refine and customize highly volatile business processes from internally and externally available Web services.

In our opinion, at present, the full potential of SOA has yet to be achieved in the software industry: applications are built on the premise that consistently implementing individual service in service-oriented environment will allow solving all computational tasks by means of simply establishing dynamic bindings between individual services. Applying this simple principle to develop enterprise solutions is however hardly possible. In the context of Enterprise Application Integration (EAI) in large organization, specialization is inevitable, therefore a particular user is not interested in dealing with all available services that organization's environment can offer. Instead, she is interested to interact with a subset of these services that helps fulfilling her particular goals. The software industry must shift the focus from the discovery / bind / invoke mechanisms to the provisioning of end-to-end solutions.

One of the reason why SOA is still not widely exploited in the large is its inherent flexibility and generality:

- on one hand, SOA alone doesn't contain detailed methodological and technological guidelines, like the ones found in the widely adopted enterprise architectures like CORBA, J2EE, or DCOM. For the pragmatic software architect, this flexibility of SOA comes at a price of ambiguity on what and how to achieve software development targets.
- on the other hand, software developers who are building a service-oriented application, must define themselves a development methodology, which focuses on analyzing, designing and producing an SOA, in such a way that it aligns with business process interactions between involved partners. The challenge in selecting and following a services design and development methodology is to provide sufficient principles and regulations to deliver the software quality required for business success, while avoiding steps that waste time, squander productivity, and frustrate developers.

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 τ, 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.

Established present-day software engineering technologies and methodologies are centered around the software application concept — rather self-contained computational facility capable of fulfilling particular purpose. Despite the fact that none of current useful applications would be fully independent of their environment, the environment is rarely considered at design time: application design specifies only what the environment should provide in order for the application to run. 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 (i.e. composed services). In the next section, we provide a formal definition for the proposed entities in our conceptual framework.

III. Mathematical Foundation for Service-Oriented Applications Design

A. Service Definition

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:

$$S_i =< P_1^i, P_2^i, \ldots, P_n^i, \Lambda >, \quad (2)$$

where S_i — logical service instance, P_k^i — k^{th} process implementing the logical service functionality f_i through the programmatic interface I_i , and Λ — network communication function between individual processes. Processes of the service S_i may run:

- on a single processor; in this case logical service is identical to physical process instance, or $S_i = P_1^i$, and network coordination function Λ is nil;
- on multiprocessor host, in this case individual logical service consist of a number of internal parallel physical processes, where the Λ function is determined by the specific hardware and software of the host;
- on an heterogeneous distributed system, where the Λ function is implemented on the base of software- and hardware-neutral general purpose protocols.

B. Service-Oriented Environment Definition

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

$$Env_\tau =< S_1, S_2, \ldots, S_n >, \quad (3)$$

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

Generally, service environments may be distinguished in various ways, for instance:

- on the basis of the communication protocol. E.g., HTTP service is distinguished from the FTP service by the protocol specification;
- on the basis of service' content. E.g., news site service environments from major news agencies offer logical services different from eCommerce service environments, albeit using the same protocol

C. Service-Oriented Application Definition

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 τ:

$$F_A =< S1^A, S_2^A, \ldots, S_n^A >_{Env_\tau} \quad (4)$$

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:

$$A =< F_A, V_A > \quad (6)$$

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: i.e., particular service must not have knowledge about the application it participates in [7]. In our framework, the Application Functionality directing graph, V_A, defined in equation5, is the formalization of such capability. In concrete application, V_A may be expresses with existing implementation frameworks for capturing services processes, such as BPEL [8] and WS-Coordination. [9]

IV. 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 functionality and directing graph must be implemented.

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..).

A. Service-Oriented Application Classes

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 [10]. Examples of such QoS requirements are:

- volume conditions: number of tuples in T_{in} set;
- temporal constraints: in what time window should all tuples of the set be processed and how many tuples should be processed in the single time unit;
- security requirements: how tuple confidentiality will be enforced during handling.
- reliability conditions: how long the application will be handling incoming data until first fault; what would be the consequences of the fault(data loss, time loss, denial of service or non usability, and so forth); etc..

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 and on the global service environment.

Database literature has introduced a simple, but effective, data system classification based on the nature of the handled request tuples: *OLTP*[1] and *OLAP*[2] systems. Their key distinction is in the nature of the data and of the queries, although both system may use the same data storage engine: *OLTP*-systems handle relatively large number of relatively simple queries while *OLAP*-systems — relatively small number of complex queries. Present state of the art of database systems offers a wide spectrum of database engines for different tasks — from small local databases [3] to enterprise-level systems[4]. However, in current database technology, the term "transient" has changed. It is now simpler to use database to store data which is considered transient from the business process point of view (e.g., storing session data in database provides easy means of maintaining stateless clusters). In this regard, the *OLTP/OLAP* classification in not any longer descriptive enough to allow the system architect to select the right approach for the particular application. 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

$$\hat{T}_A = M_A \cup T_A \cup \psi_A \tag{7}$$

where M_A — tuple set capturing temporary application memory, T_A — persistent application memory, and ψ_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 [11] implies that, during application runtime, the incoming tuples T_{in} set is reflected on entire set $\hat{T}_A$, that is:

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

For any application A, one or more data entry and data exit[5] 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 (see fig. 1 that illustrates this simple case in the IDEF0 notation). 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

[1] *On-Line Transaction Processing*
[2] *On-Line Analytical Processing*

[3] e.g. FireBird, Cloudscape, Sybase
[4] e.g. DB2, Oracle, MaxDB
[5] Except for the cases where application is designed to retain data indefinitely — see accumulating applications as defined in the following subsection

case, the key task of such *cooperation* class service-oriented application is establishing unambiguous access sharing to the t_i tuple and mutual coordination (fig. 2 illustrates a simple case of this class).

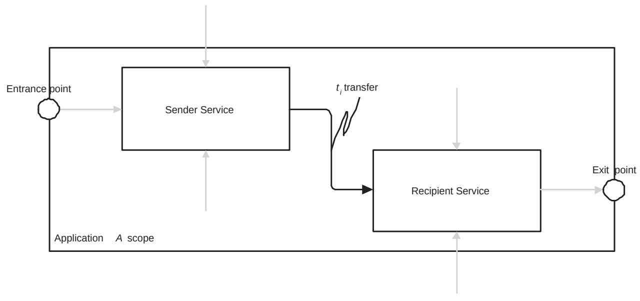

Fig. 1. Example of a simple flow-class service-oriented application: data transfer from one service to another in the scope of a single application

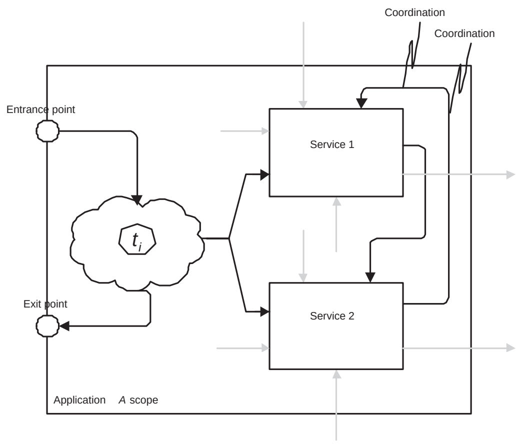

Fig. 2. Example of a simple cooperation-class service-oriented application: service cooperation on shared data space.

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 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.

B. Flow-Class Service-Oriented Application

A flow-class service-oriented application implements its tuple handling functionality by transferring tuple between involved services: each service has exclusive ownership of the tuple. Tuple handling forms an *handling path* — route of tuple from entry point to the exit point through the application's services. By transferring a tuple to the following service, the previous service surrenders exclusive ownership. A relevant number of simple business process applications will map to this class. In fact, one can view the flow-class application as a suitable business process projection into the service-oriented architecture framework.

The flow-class service-oriented application comprises individual service Each service is an independent entity, and possesses its own isolated tuple data space, $\hat{T}_{S_k} = M_{S_k} \cup T_{S_k} \cup \psi_{S_k}$, which defines transient and durable memory of the service.

Durable memory T_A of the application is defined as union of all durable spaces T_{S_k} of individual services:

$$\begin{gathered} T_A = T_{S_1} \cup T_{S_2} \cup \ldots \cup T_{S_n}, \\ T_{S_1} \cap T_{S_2} \cap \ldots \cap T_{S_n} = \oslash \end{gathered} \tag{9}$$

where n — number of services in the application.

Application's durable memory is heterogeneous. Given that T_{saved} is the finite countable set of all durable state tuples under service' control, T_{queued} is the set of tuples that are queued for passing to next services, and T_{out} is the finite countable set of all tuples passed to recipient entities, then the following condition should always be true (data integrity rule):

$$T_A = T_{saved} \cup T_{queued} \cup T_{out} \tag{10}$$

Depending on the size and nature of the tuples in T_{saved}, T_{queued}, and T_{out} sets, and of the virtual tuple set ψ, defined in equation 7, we can distinguish several application sub-classes:

- **Façade and satellite applications** . These applications do not have their own durable memory [6], i.e. $T_A = \oslash, \hat{T}_A = M_A, \psi = \oslash$, and use tuple space of another application (including interface of that application) called *base application*. Tuple transfer and export functions are not defined for such applications. *Façade* applications provide the same functions as base application: eventually they can be adapted to the technical context of the façade application (this usually applies to network protocols, user interface organization, security implementation, etc..). *Satellite* applications use base application's data to deliver new functionality that is not found in the base application (typically these are analytical or reporting functionality).
- **Transient applications** are oriented to handle incoming tuples and to pass on handled tuples to a service-recipient: these applications support primarily data transfer functionalities. For such applications $T_{saved} = \oslash, \psi = \oslash$, and $T_{in} \equiv T_{out}$, given that T_{queued} is not relevant[7]. Key business-tasks of such applications are data reception (as a rule or validation) and conversion; therefore import/export functions could also be defined for such applications.
- **Transient-journal applications** implement the same tasks as mere transient applications, but they also implement temporary memory M_A and a journal, allowing restoring action sequence taken on a tuple, as well as the tuple itself, some time after tuple handling has been completed in full and it has been

[6]this *does not* mean that these applications will not implement database technology internally.

[7]In real cases, the T_{queued} is not empty only due to technical implementation specifics

passed on to service-recipient. These applications are also distinguished from *accumulating* applications (see below) by defining maintenance procedure to remove old (expired) journal records from the temporary memory M_A.

- **Transient-filtering applications** are transient applications by nature, however $T_{saved} \neq \oslash$ and $\psi \neq \oslash$. These applications will not provide all incoming tuples for output, but they will first apply specific business logic filtering.
- **Accumulating applications** are data handling endpoints. A data transfer function is not defined for these applications: $T_{out} = \oslash$. Such applications may implement, for instance, logging or automated tools to remove unnecessary data.

In current work, we are further studying the flow-class applications, in particular flow-class topologies patterns specified by the application functionality graph V_A.

C. Cooperation-Class Service-Oriented Applications

Cooperation-class applications are useful when a tuple has to be handled simultaneously by several services and no exclusive access can be established (either for technical reason or from the business point of view). The focus of these application class is primarily on the coordination of the services that can potentially access simultaneously a tuple in the shared tuple space. An interesting comment is that most of the results from the concurrent programming domain can be usefully exploited in this application class.

Within this class, we can distinguish several application sub-classes. The main ones are:

- **Race cooperation applications**, where several services compete for the temporary exclusive access to the tuple in the shared tuple space. The goal of the overall application is to ensure effective tuple access, so that all services that are interested in gaining exclusive access to the tuple will have opportunity to do so in a timely fashion. Software development practice refers to a state of tuple being exclusively accessible by particular entity as to *locking*. In a *locked* state, other entities but the locker or its delegates generally cannot (or should not) perform read / write operations on the tuple. The semantic of a lock is that it is durable indefinitely until it has been removed by the locker. Issues related to priority-locking, optimistic/pessimistic locking techniques, centralized or distributed mechanisms, fairness and time robustness need to be address within this type of applications.
- **Fork/Join cooperation applications**: where several services process particular aspect of the tuple's data. We can distinguish the following kids of parallelism:
  - Isolated parallelism: each of the services involved in the tuple handling may consider a tuple as being exclusively owned. Requirement is that the tuple is programmed as composition of automata that are exposed for each service.
  - Stronger-isolated parallelism: similar to isolated parallelism, however some of the tuple's properties have related semantics. Services intending read/write/use properties having affect on the semantic meaning on other properties of the tuple must synchronize such operations via chosen synchronization mechanisms.
  - Detached parallelism: the application is programmed in a manner that most operations are done on a tuple copy obtained via synchronized (exclusive) access technique. Such copy of the tuple is *detached* from actual tuple copy. The niche for such applications is long-term low-concurrency data handling where business process essentially supports such parallelism.
- **Critical section application** comprises services that compete to gains exclusive access to the tuple for short period of time. Unlike race-type applications, that gains all-or-nothing access to the tuple, critical section applications introduce more sophisticated tuple access synchronization. The key aspect of the critical section is in the definition of the scope, stability, and "preemptiveness" of the lock. The scope refers to properties of tuple that are being captured by the particular critical section. The stability refers to the operations and their semantics that owner(s) expects would be performed reliably. The preemptiveness refers to the possibility of several critical sections to co-exist simultaneously on the application.
- **Coordinator-based application** that are in charge of synchronization between entities involved in the tuple handling. Coordinator-based applications do not require neither locking, nor critical section mechanisms since coordinator takes responsibility for allocating work and collecting results. Coordinator-based applications are organized similar to 2PC / 3PC[8] implementations. To begin with, each executor service must join with the coordinator to advertise it's availability to the coordinator. Then, the coordinator would communicate a task to each service available.

V. Related Work

The approach presented in this paper is based on the critical assessment of existing design formalization techniques, mainly in the object and component oriented programming domains. Mathematical 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 [12], [13] through the use of refinement techniques to the most used methods for monotonic composition of programs (namely procedures), parallel composition and data encapsulation. In particular,

[8]2-phase and 3-phase commit

in [13], parallel actions in software systems were modelled by their atomic representations, allowing to utilize methods originally developed for sequential systems.

A mathematical foundation for object-oriented paradigm is presented in [14], where message-based automata for modelling object behavior in terms of cleanroom software engineering methodology is described. In [15], software refinements is approached through a mathematical description of all possible transformations, capable to ensure refinement correctness with respect to other software objects.

In [16], 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 [17] 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 [16], [17] 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.

Current on-going activities in software development methodologies in light of the specific features of service-oriented applications are also related to our approach. In particular, the activities related to the tasks of specifying the software development processes (see for instance Rational Unified Process) and of eliciting system's requirements and user's goals (see for instance TROPOS [18] and MAP [19]).

Rational Unified Process (RUP) is one of de facto world-wide accepted standards in the software development industry. Iterative software development, requirements management, quality assurance and other practices (Kruchten, 2000) are now integral parts of the process behind any software project. They have proven to be useful for numerous projects and now can be found as the best practices part of RUP. RUP has its own software development lifecycle model comprising of inception, elaboration, construction and transition. However, lifecycle in RUP is different from the classical waterfall model stages in a sense that it represents the major business decisions points rather than technical evolution milestones. The primary application area of RUP is long-term projects. While RUP belongs to the traditional methodologies family, it covers all phases of the development process in details. Originally it was introduced to the component-oriented design concept; however it does not have strict limitations that will prevent its usage within other conceptual frameworks. Service-oriented concept defines a new level of architectural principles that is related to the existing object- and component-oriented software design concepts. Nevertheless the aspect of loosely-coupled services engenders considerable differences in the way SOA applications are built compared with traditional object- and component-based software. Delivering SOA applications developed under RUP will require customizing the process to address SOA applications planning. These differences have an impact on the way RUP need to be adapted for developing and delivering SOA applications.

The present paper leverages from the above research on software design formalization approaches and aims to extend them to service-oriented programming paradigm.

VI. Conclusions

In this paper we have extended the definition of SOA, and based on this definition, we have proposed a lightweight, but complete, mathematical framework capable of capturing SOA main components. This formalization allowed us to explore the structure of SOA, to derive and discuss substantial properties for its main components with respect to data processing mechanisms, 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. Moreover, service' internal memory structure establishes several application sub-classes.

However much more must be done. Our future work includes: the refinement of the mathematical model ; in-depth exploration of the introduced classification, in particular class topologies patterns; QoS aspects of SOA; patterns for SOA. Such formalization will provide a mature framework for designing practical service-oriented applications that will exhibit expected behavior and performance over time in dynamic services environments.

References

[1] H. K. Gustavo Alonso, Fabio Casati and V. Machiraju, *Web Services Concepts, Architectures and Applications*. Springer, 2004.

[2] M. Cantara, "It professional services forecast and trends for web services," January 2994.

[3] M. P. Papazoglou and J. Yang, "Design methodology for web services and business processes," in *TES '02: Proceedings of the Third International Workshop on Technologies for E-Services*. London, UK: Springer-Verlag, 2002, pp. 54–64.

[4] R. Dijkman and M. Dumas, "Service-oriented design: A multi-viewpoint approach," *International Journal on Cooperative Information Systems*, vol. 13, no. 14, pp. 338–378, December 2004.

[5] D. Quartel, R. Dijkman, and M. van Sinderen, "Methodological support for service-oriented design with isdl," in *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*. New York, NY, USA: ACM Press, 2004, pp. 1–10.

[6] M. P. Papazoglou, "Service-oriented computing: Concepts, characteristics and directions," in *WISE '03: Proceedings of the Fourth International Conference on Web Information Systems Engineering*. Washington, DC, USA: IEEE Computer Society, 2003, p. 3.

[7] D. K. Barry, *Web Services and Service-Oriented Architecture: The Savvy Manager's Guide*. Morgan Kaufmann Publishers, 2003.

[8] "Business process execution language for web services." [Online]. Available: http://www-106.ibm.com/developerworks/library/ws-bpel

[9] "Ws-coordination specifications." [Online]. Available: http://www-106.ibm.com/developerworks/library/ws-coor

[10] J. W. P. Sandeep Chatterjee Ph.D., *Developing Enterprise Web Services: An Architect's Guide.* Prentice Hall PTR., 2003.

[11] R. W. Taylor and R. L. Frank, "Codasyl data-base management systems," *ACM Comput. Surv.*, vol. 8, no. 1, pp. 67–103, 1976.

[12] R. J. R. Back, "Correctness preserving program refinements: Proof theory and applications," 1980.

[13] K. Sere and R. J. R. Back, "From action systems to modular systems," in *FME'94: Industrial Benefit of Formal Methods*, M. B. M. Naftalin, T. Denvir, Ed. Springer-Verlag, 1994, pp. 1–25. [Online]. Available: citeseer.ist.psu.edu/back96from.html

[14] B. Rumpe and C. Klein, "Automata describing object behavior," pp. 265–286, 1996. [Online]. Available: citeseer.ist.psu.edu/rumpe96automata.html

[15] D. Craigen, S. Gerhart, and R. T.J., "An international survey of industrial applications of formal methods," National Technical Information Service, Springfield, VA, USA, Tech. Rep., 1993. [Online]. Available: citeseer.ist.psu.edu/craigen93international.html

[16] M. Broy, "Towards a mathematical concept of a component and its use," *Software - Concepts and Tools*, vol. 18, no. 3, pp. 137–, 1997. [Online]. Available: citeseer.ist.psu.edu/broy96towards.html

[17] M. Broy, "Compositional refinement of interactive systems modelled by relations," *Lecture Notes in Computer Science*, vol. 1536, no. 3, pp. 130–149, 1998. [Online]. Available: citeseer.ist.psu.edu/broy92compositional.html

[18] M. J. Giorgini, P. and R. Sebastiani, "Goal-oriented requirements analysis and reasoning in the tropos methodology," *Engineering Applications of Artificial Intelligence*, vol. 18-2, pp. 159–171, 2005.

[19] R. Kaabi, C. Souveyet, and C. Rolland, "Eliciting service composition in a goal driven manner," in *Proceedings of the International Conference on Service Oriented Computing 2004.* ACM, November 2004, pp. 308–315.

Aliaksei Yanchuk received his Master of Science from the Belarusian State University of Informatics and Radioelectronics with the major in the business systems design in 2002.

He is actively involved with the commercial software development since 1999. Presently he is employed by the Netherlands branch of SaM Service GmbH as software developer / analyst to provide on-site software development services for major European organization. He is affiliated with the Department of Informatics and Telecommunications of the University of Trento.

Alexander Ivanyukovich received his B.S. degree in radio physics from the Belarusian State University, Minsk, Belarus in 2002. He is an IEEE/CS member since 2003. Currently, he is a Ph.D. candidate in computer science at the University of Trento.

He has being actively involved with the commercial software development since 2000. He has founded and lead Zaval Creative Engineering Group - public body aimed contributing to the developers' community with enterprise-quality open source software products and carrying out research and development in field of early-access and emerging technologies. Nowadays he is engaged in research on distributed, performing and fault-tolerant information retrieval and processing systems at the University of Trento.

Maurizio Marchese graduated with full honor in Physics in 1984 at the University of Trento, Italy.

He has been Visiting Researcher at the National Research Council of Canada, Ottawa, Canada; Post-Doctoral Research Associate at the Material Research Laboratory, University of Urbana-Champaign, USA; Visiting Researcher at the Institute for Computer Applications, University of Stuttgart, Germany. He is currently Assistant professor at the Department of Information and Communication technologies at the University of Trento, Italy. Current research interests are: architectures for web services, distributed architectures for digital libraries, service integration in Geographical Information Systems (GIS) environments. He has published more than 60 papers in international journals and conferences.

Dr. Marchese is member of IEEE Computer Society and ACM.