A Framework for Adapting Service-oriented Applications based on Functional/Extra-functional Requirements Tradeoffs

Raffaela Mirandola
Politecnico di Milano
DEI, Milano, Italy
mirandola@elet.polimi.it

Pasqualina Potena
Univ. degli Studi di Bergamo
DIIMM, Dalmine (BG), Italy
pasqualina.potena@unibg.it

Elvinia Riccobene
Univ. degli Studi di Milano
DTI, Crema (CR), Italy
elvinia.riccobene@unimi.it

Patrizia Scandurra
Univ. degli Studi di Bergamo
DIIMM, Dalmine (BG), Italy
patrizia.scandurra@unibg.it

***Abstract*—This paper introduces an adaptation framework for service-oriented applications based on trade-offs between functional and extra-functional (e.g., availability, performance, and adaptation cost) requirements. The framework relies on an optimization method for adaptation space exploration based on the combined use of meta-heuristic search techniques and of functional and extra-functional patterns (e.g., architectural design patterns and tactics). A formal service-oriented component model, called SCA-ASM, is also adopted for the specification and functional analysis of service-oriented applications. Through a sample application, we exemplify the methodology with emphasis on the use of extra-functional patterns.**

***Keywords*-Service-oriented applications; software adaptation and evolution; extra-functional adaptation patterns.**

I. Introduction

Service-oriented applications are playing an important role in several application domains (e.g., health care, defense and aerospace) since they offer advanced and flexible functionalities in widely distributed environments by composing, possibly on demand, different types of services. These applications may require dynamic adaptation to changing user needs, system intrusions or faults, changing operational environment and resources. Foundational theories and notations are required to support the engineering of such applications. Also required are techniques for monitoring and evaluating the behavior and performance of these applications, fully integrated in a software engineering process that reflects the *closed-loop paradigm* (e.g., the MAPE-K loop in the context of autonomic computing) [1] are required.

Extra-functional properties of services are often specified as quality of service (QoS) constraints and their values are dynamic [2]. For example, the system response time depends on environmental factors among which input data, server load, and network latency. The adaptation decisions for implementing the single changes should be triggered whenever unsatisfactory behaviors and values are reported by monitoring modules or required by the user (or "system designer" or "system maintainer"), and the right trade off among the functional requirements, software qualities and the adaptation cost should be considered. A decision, for example, taken for modifying the dynamic of a service may be good for the satisfaction of the system reliability, but at the same time it may require a high adaptation cost for adapting the interfaces of services [3].

This paper presents an adaptation framework based on functional and extra-functional requirements trade offs. It is based on a formal *service-oriented component model*, named SCA-ASM [4], for the specification and analysis of service-oriented applications, and on a runtime *optimization method* for adaptation exploration that uses a mixed approach of metaheuristic search techniques [5], of functional and extra-functional adaptation patterns, such as architectural design patterns and tactics, or also software actions defined by the maintainer based on his/her experience. The adopted optimization approach enhances the one defined in [3] by taking into account also functional issues that allow, among other things, to relax the independence assumption between adaptation actions for different adaptation requirements.

According to the *design for adaptability* vision in [6], our framework supports both evolution (at re-design time) and self-adaptation (at run time). The second form of adaptation regards temporary modification (such as the re-execution of an unavailable service or a substitution of an unsuitable service) permitting to respond to changes in the requirements and/or in the application context. However, when changes regard critical aspects and should be applied permanently to the system, they should be considered as evolution steps, and therefore fast answers are not essential since adaptation strategies are evaluated and carried out at (re-)design time.

This paper is organized as follows. Section II reports related works. Section III provides background on SCA-ASM. Section IV describes our adaptation methodology. Section V presents the overall architecture of our framework. Section VI exemplifies our methodology through a sample application. Finally, Section VII sketches some future work.

II. Related work

A survey on adaptation approaches and frameworks can be found in [1]. Most of them typically adapt a system by adopting different service selection policies, varying system parametrization or exploiting the inherent redundancy of the Service-Oriented Architecture (SOA) environment.

Some frameworks exist for the dynamic generation of a service composition, but usually they adapt a system only in a reactive way, after the adaptation request triggered by a user. They support the service selection with respect to a service composition defined by the user (e.g., the VRESCo runtime environment [7]) or choose the service composition, which they have generated together with a finite set of other candidates, that better fulfill the required quality (e.g., [8]).

With respect to the state-of-art, our work is the first framework (to the best of our knowledge) that supports the adaptation of service-oriented applications (including both static and dynamic aspects) at runtime and at re-design time. It uses a mixed approach of metaheuristic search techniques, whose effectiveness and efficiency has been already demonstrated for supporting the service selection activity at runtime [2], and of functional and extra-functional adaptation patterns [9]. Existing approaches typically do not take into account functional aspects and assume that a component is functional equivalent to its alternatives [10]. Concerning design solutions, existing approaches (e.g., [9] and [11]) do not quantify or predict the impact of the adoption of one or more solutions on the system quality and functionality. As opposite, we address such a problem.

III. Background on SCA-ASM

SCA-ASM [4] [12] is a formal and executable modeling language based on: (i) the open standard *Service Component Architecture* (SCA) [13] for heterogeneous service assembly, and (ii) the *Abstract State Machines* (ASM) formal method [14], which is an extension of FSMs where *states* are arbitrary complex data (multi-sorted first-order structures) and the *transition relation* is specified by *rules* describing how functions change from one state to the next. The SCA-ASM formalism is able to model service interactions, orchestration, compensations, and services internal behavior.

An SCA assembly (or composition) of service-oriented components can be graphically produced using the Eclipse-based SCA Composite Designer (an inner module of the SCA tools), and also stored or exchanged in terms of an XML-based file that is then used by the SCA runtime to instantiate and execute the system. The ASM formalism complements the structural description of the SCA assembly with a formal and executable behavioral description of the assembled components. Figure 1 shows an example of an SCA assembly of a stock trading application (better described in Sect. VI), while Figure 2 shows an ASM fragment of the `OrderDeliveryComponent` component behavior.

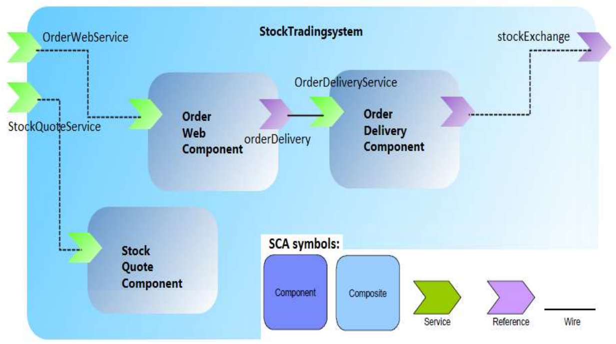

Figure 1. Stock Trading System

```
module OrderDeliveryComponent
//@Provided service interface
import OrderDeliveryService
//@Required service interface
import StockExchangeService
...//Other module imports
signature: //ASM function declarations
//@Reference to the external stock exchange system
shared stockExchange:Agent−>StockExchange
//@Backref back reference to the requester
shared client: Agent −> Agent
//Other function declarations for internal computation
controlled order: Agent −> Order
definitions:
//ASM rules for the provided service operations
@Service
rule r_place($client in Agent,$o in Order)= ... //to place buy or sell orders
...
//ASM rule for the component's agent behavior
rule r_OrderDeliveryComponent=
  seq
   r_wreceive(client(self),"place",order(self))
   //direct service invocation
   r_place(client(self),order(self))
   r_wreplay(client(self),"place",order(self))
  endseq
//constructor rule
rule r_init($agent in OrderDelivery)=$ ... //do initialization (if any)
```

Figure 2. ASM module of the `OrderDeliveryComponent`

IV. The Adaptation Methodology

An SCA assembly can be adapted through the following *actions*: adding/removing components, component services, references, properties, reference-service wires and promotion wires (component interactions); changing a component implementation (but keeping its shape); changing component properties values; changing SCA domains (components re-deployment). It is also possible to change the component interaction style in synchronous/asynchronous, stateful or not, unidirectional or bidirectional. See [13] for details.

Actions can be combined into an *adaptation plan*, which is a set of actions modifying the static and dynamic parts of a system architecture to address a certain requirement. Adaptation plans may differ for adaptation cost and/or for the system quality achieved after their application.

The proposed *adaptation process* starts from a set of initial SCA-ASM assemblies (initial *candidates* or *population*) fulfilling the existing/new functional requirements. It proceeds iteratively till stop criteria are satisfied. Currently, we use a predefined number of iterations to determine the end of the search. More sophisticated stop criteria could use convergence detection and stop when the global optimum is probably reached. At each iteration step, new candidates are generated from the initial population (whose size depends on the specific search technique) by two subprocesses executed in parallel: (i) *metaheuristic search* by applying user adaptation plans, service selection and service re-deployment; (ii) *functional/extra-functional patterns application* by exploiting architectural design patterns and tactics. Then the functional and quality analyses of the resulting candidates are performed together with an assessment of the adaptation costs. In case of self-adaptation, SCA-ASM assemblies are automatically selected as solutions according to predefined selection criteria (e.g., cost minimization). In case of evolution, the solution can be more accurately selected also considering a possible feed-back from the user [6].

Metaheuristic search techniques. Several metaheuristics [5] with different characteristics could be adopted depending on the problem: for example, considering the system reliability, a possible heuristic is to regard as increasing the whole reliability of the system when the reliability of the most used components increases. As remarked in [10], there exist design options for which we have no prior knowledge on how they affect the extra-functional property of a particular system. To this extent, undirected operations could be performed (e.g., random choices or exhaustive evaluation of all neighboring candidates).

Architectural design patterns and tactics. Architectural patterns are templates for concrete software architectures. They are adopted to embody functional requirements and, in particular, to enable self-adaptability by introducing sensors/effectors components (e.g., Microkernel pattern, Reflection pattern, Interception pattern) [1]. To build new design solutions embodying extra-functional requirements, we adopt architectural tactics [9], which are reusable architectural building blocks that provide a generic solution to issues pertaining to quality attributes.

V. The Adaptation Framework

Figure 3 shows the main modules of the framework. The core of the framework is an optimization approach (implemented by the *Reasoner* module) that adapts (through the *Executor* module) an SCA-ASM assembly (developed by the *System Model Creator*) of a service-oriented application with respect to the functional requirements, system qualities, and adaptation costs. Adaptation actions can be triggered automatically (after receiving alerts from the *Monitor* module) and/or by the user (through the *User Requests Manager* module that also interacts with the *Monitor* module to figure out internal or context changes). An *Analyzer* assists during the adaptation process for functional and quality analysis purposes. A description of each module follows.

System Model Creator and Executor. This module consists of two sub-modules (the *creator* and the *executor*) and relies on the integration of the SCA tools and runtime platforms (like Tuscany, FraSCAti, etc.) with the ASM toolset ASMETA [15], to graphically model, compose, analyze, deploy, execute, and introspect service-oriented applications.

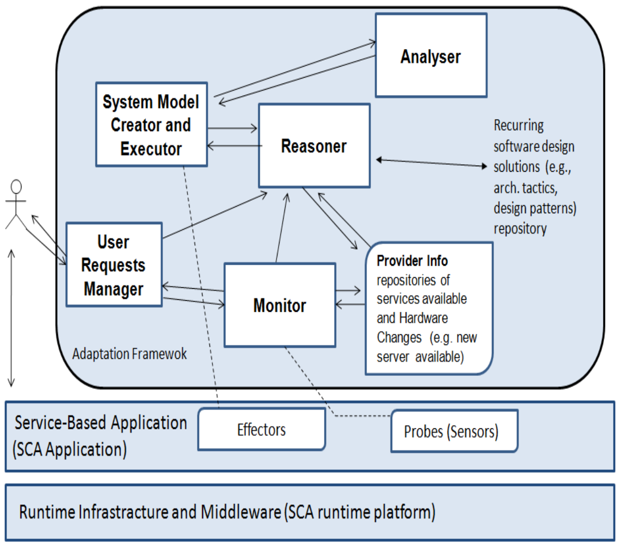
Analyser
System Model Creator and Executor
Reasoner
Recurring software design solutions (e.g., arch. tactics, design patterns) repository
User Requests Manager
Monitor
Provider Info repositories of services available and Hardware Changes (e.g. new server available)
Adaptation Framewok
Service-Based Application (SCA Application)
Effectors
Probes (Sensors)
Runtime Infrastructure and Middleware (SCA runtime platform)

Figure 3. The Adaptation Framework

An SCA-ASM model (or assembly) of an application can be produced from scratch, or generated from an existing system implementation. Analysis techniques can be employed to assure consistency between the architecture and the implementation. Another feature of the *System Model Creator* is allowing, by exploiting the SCA Policy Framework [13], the designer to specify for components necessary metadata annotations. These are useful for providing metrics that can be extracted from the model for nonfunctional analysis purposes, and for representing policies that can be guarantee by the runtime platform. It also allows the application of design patterns and tactics to an SCA-ASM assembly, leading to a chain of adaptation actions. To guarantee the functional correctness of the resulting assembly and that changes claimed by the adaptation actions do not compromise the satisfaction of existing functional requirements, an interaction with the *Functional and Quality Analyzer* is required. Different adaptation actions of the SCA assembly may be enacted manually (as suggested by the *User Request Manager*) or automatically (by the *Reasoner*).

The *System Model Executor* implements the adaptation actions suggested by the *Reasoner*. Through the use of effectors, changes applied at model level must be related to the underlying mechanisms and runtime infrastructure. To this extent, guidelines of existing approaches supporting dynamic service invocation and of the ones for dynamically adapting the system behavior could be exploited. In the case of SCA, mechanisms like introspection and reconfiguration, for managing and enacting self-adaptation [16] are applied.

User Requests Manager. It allows users to make adaptation requests by providing appropriate adaptation plans. It assures that plans of different adaptation requirements are independent between each other, i.e., changes claimed by a plan do not compromise the application of other plans.

Reasoner. It is activated after receiving either adaptation requests from the user or alerts from the *Monitor*. By using an optimization approach, it produces a set of software adaptation actions. Through the help of the *System Model Creator*, it generates the new system architecture model, i.e., the new SCA-ASM assembly model including both structural and behavioral aspects. The adaptation space exploration process implemented by the reasoner is iterative and is based on the combined use of meta-heuristic search techniques and of functional and extra-functional adaptation patterns (i.e. architectural design patterns and tactics). A detailed description of the optimization method and related techniques are out of the scope of this paper.

Monitor. It controls the system at runtime through the use of probes (sensors). It may trigger self-adaptation when detecting relevant context and internal changes or an evolution cycle of the system for introducing important and permanent changes [6]. For implementing such a module, several monitoring approaches exist in the literature (see, e.g., [1]). The monitor can also continually measure the services' QoS attributes. The providers can improve the estimate of the services' non-functional properties by monitoring them.

Functional and Quality Analyzer. It consists into a set of external tools that can be invoked for different analysis purposes. Essentially two sub-modules can be identified: one for the *functional*, the other one for the *non-functional analysis*. The *functional analyzer* is linked with ASMETA [15], a set of tool for the ASMs. It is invoked when a preliminary analyses of the functional requirements satisfiability of the SCA-ASM assembly would be performed by easier techniques as simulation or scenario-based validation. Later, heavier formal verification techniques (as model checking) can be exploited when more complex functional properties [17] must be proved to guarantee behavioral system correctness. The functional analyzer is also invoked when correctness must be proved upon a refinement step of the SCA-ASM assembly due to adaptation actions. Techniques for checking correctness of model refinement as supported by the ASMETA tool-set. The *non-functional analyzer* exploits external tools for performance and reliability analysis like `qnetworks` [18] and `LQNsolver` [19]. The system qualities (e.g., performance and reliability) and the adaptation costs are predicted exploiting the SCA-ASM model of the system. Examples of adaptation costs can be found in [3]. Considering quality analysis, different approaches/strategies can be used depending on several factors due mainly to the use of our framework for evolution (at re-design time) or self-adaptation (at run time). If permanent changes, for example, are requested or a safe-critical service has to be adapted, precise (often expensive) analysis must be performed (e.g., see [20] for performance analysis). As opposite, if runtime changes are claimed and these require, for example, only the adaptation of parameters without using more sophisticated analysis, faster approaches must be adopted allowing a prompt run-time adaptation (see, e.g., techniques for estimation of quality at runtime, such as [21]).

VI. THE STS CASE STUDY

We describes the adaptation methodology by a sample application from the Stock Trading System (STS) in [9]. Figure 1 shows the SCA assembly of the STS. Briefly, an STS user, through the `OrderWebComponent` interacting with the `OrderDeliveryComponent`, can check the current price of stocks, placing buy or sell orders and reviewing traded stock volume. Moreover, he/she can know stock quote information through the `StockQuoteComponent`. STS interacts also with the external Stock Exchange system, which we do not model.

Figure 2 shows a fragment of the ASM (abstract) model for the `OrderDeliveryComponent` behavior. The main service of this component (the rule `r_place` annotated with `@service`) is to place buy or sell orders when requested (see the blocking receive action and the replay action preceding and following, respectively, the service invocation within the component's main rule `r_OrderDeliveryComponent`). The ASM definition for the provided and required interfaces of the `OrderDeliveryComponent` are reported in Figure 4. They are ASM modules containing only declarations of business agent types (the subdomains `OrderDelivery` and `StockExchange` of the predefined ASM `Agent` domain) and of business functions (parameterized ASM out functions) used as temporary locations to store service computation results.

```
module OrderDeliveryService
import ... //Other module imports
signature:
// the domain defines the type of the provider component's agent
domain OrderDelivery subsetof Agent
// business function value
out place: Prod(Agent,Order) −> Order

//@Remotable
module StockExchangeService
import ... //Other module imports
signature:
domain StockExchange subsetof Agent
out sendOrder: Prod(Agent,Order) −> Rule
```

Figure 4. ASM modules of the `OrderDeliveryComponent` interfaces

Below, we apply to the STS case study some adaptation strategies adopted by our methodology. Specifically, first we describe the application of a simple metaheuristic technique, and then we show the use of some tactics as examples of extra-functional adaptation patterns. Details on the experimental data set used in this case study can be found in [23]. *Metaheuristic search:* Figure 5 shows an example of instantiation of our optimization process by considering, on the STS example, the *steepest-ascent hill-climbing* metaheuristic [5] that tries to adapt the system minimizing

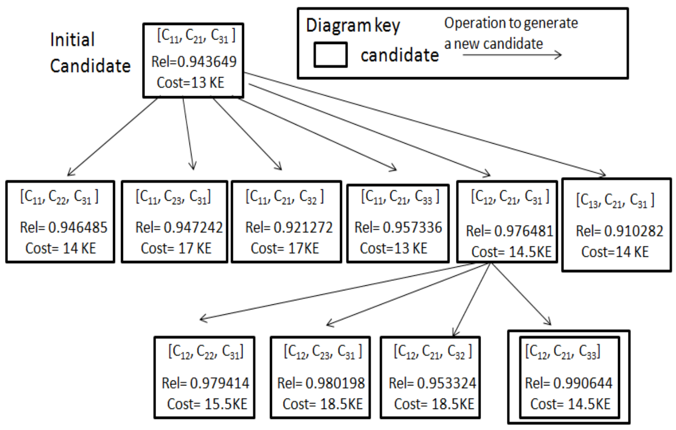
Diagram key
candidate
Operation to generate a new candidate
Initial Candidate
[C11, C21, C31] Rel=0.943649 Cost=13 KE
[C11, C22, C31] Rel= 0.946485 Cost= 14 KE
[C11, C23, C31] Rel= 0.947242 Cost= 17 KE
[C11, C21, C32] Rel= 0.921272 Cost= 17KE
[C11, C21, C33] Rel= 0.957336 Cost=13 KE
[C12, C21, C31] Rel= 0.976481 Cost= 14.5KE
[C13, C21, C31] Rel= 0.910282 Cost=14 KE
[C12, C22, C31] Rel= 0.979414 Cost= 15.5KE
[C12, C23, C31] Rel= 0.980198 Cost= 18.5KE
[C12, C21, C32] Rel= 0.953324 Cost= 18.5KE
[C12, C21, C33] Rel= 0.990644 Cost= 14.5KE

Figure 5. Example of steepest-ascent hill-climbing application

the adaptation costs and assuring a level of system reliability greater than 0.98. The initial candidate is the vector $[C_{11}, C_{21}, C_{31}]$ (see Figure 5), where C_{ij} denotes the jth instance available on the market for the component C_i with C_1 indicating the `OrderWebComponent`, C_2 the `StockQuoteComponent` and C_3 the `OrderDeliveryComponent`. Each vector comes with two parameters: the resulting system reliability and the cost of the solution (predicted using the reliability and cost model used in [22] and reported in Table 1 in [23]). At each iteration step, a set of new candidates is generated by replacing, one at a time, an existing component with one available on the market. The best candidate is then selected as the one improving the system reliability and minimizing the adaptation costs. It becomes the basis for next candidates generation. The process terminates either if no better candidates can be found or the reliability threshold is reached. In our case, the optimization process returns the solution $[C_{12}, C_{21}, C_{33}]$ with reliability equals to 0.990644 and cost equals to 14.5 KE.

Application of extra-functional adaptation patterns: We here show how availability and performance tactics can be used to embody extra-functional requirements of the STS example into its architecture. Let us assume the following extra-functional requirements (taken from [9]):

NFR1. *The STS should be available during the trading time (7:30 AM6:00 PM) from Monday through Friday. If there is no response from the system for 30 s, the STS should notify the administrator.*

NFR2. *The system should be able to process 300 transactions per second, 400,000 transactions per day. A client may place multiple orders of different kinds (e.g., stocks, options, futures), and the orders should be sent to the system within 1 s in the order they were placed.*

To address NFR1 the *Fault Detection Tactic* for the detection and notification of a fault to a monitoring component or to the system administrator can be adopted. Such kind of tactic can be refined into other ones (e.g., *Ping/Echo*, *Heartbeat* and *Exception* tactics [9]). As done in [9], we support NFR1 combining *Ping/Echo* and *Heartbeat*.

As in [9], NFR2 is supported combining the *FIFO* (for the Resource Arbitration) and *Introduce Concurrency* (for the Resource Management) tactics. The *FIFO* tactic allows clients to place each type of orders (e.g., stocks, options, futures) to a dedicated queue for immediate processing. Finally, to handle considerable amount of transactions by their kinds within a very short time, as suggested in [9], NFR2 can be also supported by reducing the blocking time of transactions on I/O, which can be realized by the combined use of the *FIFO* and *Introduce Concurrency* tactics (i.e., by concurrent dispatching of the same kind of orders).

Figure 6 shows how it would change the SCA assembly by composing these tactics: the assembly is extended to add the new `Queue` component (for the *FIFO* tactic) and the `Monitor` component (for the *Fault Detection Tactic*). The `OrderWebComponent` is refined for concurrently producing orders to place into the `Queue`. Similarly, the `OrderDeliveryComponent` is refined for adding the monitoring functionality and for the concurrent consuming of different kinds of orders placed into the `Queue` component. Of course, this implies a change of the components shape (i.e. in the required/provided interfaces) and of their behavior. The behavior, for example, of the `OrderDeliveryComponent` is refined in ASM as shown in the fragment reported in Figure 7: the consuming and sending of different kind of orders (stock, option, or future) are executed in parallel (i.e. concurrently) by the `par` rule.

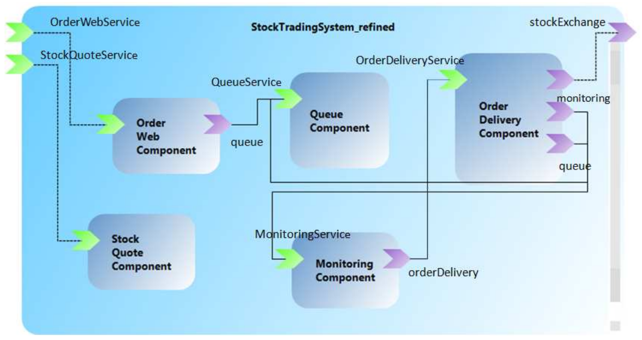

Figure 6. Adapting the STS by applying tactics for NFR1 and NFR2

It is possible to prove that the behavior of the `OrderDeliveryComponent` in Figure 7 is a correct refinement [14] of that in Figure 2, and, therefore, all initial functional requirements are still guaranteed. Moreover, the impact of the adoption of the tactics should be quantified with respect to the existing system quality. For example, the introduction of new components could decrease the maximum level of reliability. In the STS example, after the embedding of new components into the `OrderDeliveryComponent` for NFR1, if the probability of failure of the instance available for this component

```
module OrderDeliveryComponent
...
rule r_OrderDeliveryComponent=
... seq
      par //Queue consuming
        r_wsendreceive[client(self),"dispatch","Stock",stockorder(self)]
        r_wsendreceive[client(self),"dispatch","Option",optionorder(self)]
        r_wsendreceive[client(self),"dispatch","Future",futureorder(self)]
      endpar
      par //Order sending to the Stock Exchange system
        r_wsend(stockExchange(self),"sendOrder",(self,stockorder(self)))
        r_wsend(stockExchange(self),"sendOrder",(self,optionorder(self)))
        r_wsend(stockExchange(self),"sendOrder",(self,futureorder(self)))
    endpar
  endseq ...
```

Figure 7. The refined behavior of the `OrderDeliveryComponent`

increases (for example, from 0.00006 to 0.0002 [23]), then the reliability of the overall solution will decrease (from 0.990644 to 0.970639 [23]). Therefore, it could happen that the reliability constraint is not satisfied any more (in the example indeed, the system reliability is not greater than 0.98). Note that, also the reliability of the new *Queue* component may contribute to decrease the system reliability.

VII. Conclusion and Future Directions

This paper presented an adaptation framework for service-oriented applications that relies on design-for-adaptability principles while supports the closed-loop paradigm. With such a kind of support, a system is able to monitor itself and its context to detect significant changes, decide how to react on the base of functional/non-functional trade offs, and execute such decisions at runtime or at re-design time.

We intend to enhance our framework towards several directions. Currently, we are implementing a prototype to compare different implementations of our optimization process (e.g., with heuristics depending on application domain or quality attributes) on realistic examples. We intend to support the right trade-off between the adaptation overhead (due, e.g., to the frequent execution of the reasoning algorithms) and the accrued benefits of changing the system.

References

[1] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 4, no. 14, pp. 14:1–14:42, 2009.

[2] F. Rosenberg, M.B. Müller, P. Leitner, A. Michlmayr, A. Bouguettaya, and S. Dustdar, "Metaheuristic optimization of large-scale qos-aware service compositions," in *Proc. of the IEEE Int. Conf. on Services Computing*, 2010, pp. 97–104.

[3] R. Mirandola and P. Potena, "Self-adaptation of service based systems based on cost/quality attributes tradeoffs," in *Proc. of WoSS at SYNACS 2010*, pp. 493–501.

[4] E. Riccobene, P. Scandurra, and F. Albani, "A modeling and executable language for designing and prototyping service-oriented applications," to appear in *Proc. of EUROMICRO SEAA 2011*.

[5] C. Blum and A. Roli, "Metaheuristics in combinatorial optimization: Overview and conceptual comparison," *ACM Comput. Surv.*, vol. 35, no. 3, pp. 268–308, 2003.

[6] A. Bucchiarone, C. Cappiello, E. Di Nitto, R. Kazhamiakin, V. Mazza, and M. Pistore, "Design for adaptation of service-based applications: Main issues and requirements," in *ICSOC/ServiceWave 2009 Workshops*, ser. LNCS, 2010, pp. 467–476.

[7] F. Rosenberg, P. Celikovic, A. Michlmayr, P. Leitner, and S. Dustdar, "An End-to-End Approach for QoS-Aware Service Composition," in *EDOC*, 2009, pp. 151–160.

[8] D. Chiu, S. Deshpande, G. Agrawal, and R. Li, "A Dynamic Approach toward QoS-Aware Service Workflow Composition," in *ICWS*, 2009, pp. 655–662.

[9] S. Kim, D. Kim, L. Lu, and S. Park, "Quality-driven architecture development using architectural tactics," *Journal of Systems and Software*, no. 8, pp. 1211–1231, 2009.

[10] H. K. A. Martens, "Automatic, model-based software performance improvement for component-based software designs," in *Proc. of FESCA 2009*, vol. 253, no. 1, 2009, pp. 77 – 93.

[11] K. Vallidevi and B. Chitra, "Effective self adaptation by integrating adaptive framework with architectural patterns," in *Proc. of A2CWiC 2010*, ACM, pp. 67:1–67:4.

[12] E. Riccobene and P. Scandurra, "Specifying formal executable behavioral models for structural models of service-oriented components," in *Proc. ACT4SOC 2010*, pp. 29-41.

[13] "Service Component Architecture (SCA)" `www.osoa.org`, 2007. [accessed: May 18, 2010]

[14] E. Börger and R. Stärk, *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer, 2003.

[15] "The ASMETA tooset," `http://asmeta.sf.net/`, 2006. [accessed: April 26, 2011]

[16] L. Seinturier, P. Merle, D. Fournier, N. Dolet, V. Schiavoni, and J. Stefani, "Reconfigurable sca applications with the frascati platform," in *Proc. of Int. Conf. on Services Computing*, IEEE, 2009, pp. 268–275.

[17] C. Attiogbé, P. André, and G. Ardourel, "Checking component composability," in *Software Composition*, ser. LNCS, W. Löwe and M. Südholt, Eds., 2006, pp. 18–33.

[18] M. Marzolla, "The `qnetworks` toolbox: A software package for queueing networks analysis," in Proc. ASMTA 2010, Springer.

[19] G. Franks, P. Maly, M. Woodside, D.C. Petriu, and A. Hubbard, "Layered Queueing Network Solver and Simulator User Manual, LQN software documentation," 2006.

[20] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni,"Model-based performance prediction in software development: A survey," *IEEE Trans. Software Eng.*, no. 5, pp. 295–310, 2004.

[21] I. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburrelli, "Model evolution by run-time parameter adaptation," in *Proc. of ICSE'09*, pp. 111-121.

[22] V. Cortellessa, F. Marinelli, and P. Potena, "An optimization framework for "build-or-buy" decisions in software architecture," *Computers & OR*, vol. 35, no. 10, pp. 3090–3106, 2008.

[23] R. Mirandola, P. Potena, E. Riccobene, and P. Scandurra, "A framework for adapting service-oriented applications based on functional/extra-functional requirements tradeoffs: the Stock Trading System case study," TR Univ. of Bergamo (Italy), http://cs.unibg.it/potena/AdaptationFramework/TRExpResults.pdf, 2011. [accessed: July 21, 2011]