On Guaranteeing Global Dependability Properties in Collaborative Business Process Management

Valeria Cardellini, Emiliano Casalicchio, Vincenzo Grassi, and Francesco Lo Presti

Dipartimento di Informatica, Sistemi e Produzione
Università di Roma "Tor Vergata"
Via del Politecnico 1, 00133 Roma, Italy
{cardellini,casalicchio}@ing.uniroma2.it
{vgrassi,lopresti}@info.uniroma2.it

Abstract. The Service-Oriented Architecture (SOA) paradigm supports a collaborative business model, where business applications are built from independently developed services, and services and applications build up complex dependencies. Guaranteeing high dependability levels in such complex environment is a key factor for the success of this model. In this chapter we discuss issues concerning the design of such software systems, evidencing the limits of the proposed approaches, and suggesting directions for advancements in this field. Moreover, we also discuss issues concerning the case of self-adaptive SOA systems, whose goal is to self-configure themselves, to cope with changes in the operating conditions and to meet the required dependability with a minimum of resources.

1 Introduction

We are witnessing an increasing trend toward globalization and competition, where enterprises retain only core competencies, and rely on external partners for carrying out their business. Advances in Internet-based communications have provided the technological support for this collaboration process. As a result, today's business processes are cross-organizational in nature, involving extended partners of enterprises including, for example, suppliers, partners, and dealers.
One of the key motivations for this trend is the need for enterprises to achieve business agility, *i.e.*, the capacity of responding in a timely and effective way to changes in business models, business opportunities, and market conditions. Enterprise thus interact according to Collaborative Business Processes (CBPs) [25,1], that orchestrate their activities to achieve some specific goal.

*The original publication is available at http://www.springerlink.com/ in *Business System Management and Engineering*, Claudio A. Ardagna, Ernesto Damiani, Leszek A. Maciaszek, Michele Missikoff and Michael Parkin (eds.), LNCS Vol. 7350, pp. 48-70, 2012.

Generally, a CBP could include both fully automated and human activities, *e.g.*, a loan approval process typically includes human steps. However, in this chapter we will focus on CBPs defined and executed as an orchestration of fully automated software services. Similarly to us, other chapters in this book consider only business processes realized through fully-automated software services [3,28,36]. On the other hand, the chapters by Badr et al. [7] and Dubois et al. [20] take into account the incorporation of humans in service-based applications: the first in the context of knowledge-intensive business service firms, the latter to capture business requirements. Friesen et al. [21] discuss the differences and relationships between the business layer of a CBP, where the activities mostly involve humans, and the ICT layer of a CBP, where the software aspect is prevalent.

The Service Oriented Architecture (SOA) paradigm provides the architectural guidelines for software systems able to support a collaboration-based business model, as it emphasizes the construction of software systems through the dynamic composition of network-accessible services offered by loosely coupled providers. To be effective, a SOA-based implementation of a CBP must be able to guarantee some overall Quality of Service (QoS) level to the CBP users. In this chapter, we focus on the QoS facet concerning the dependability of a software system implementing a given CBP, expressed both in terms of its *availability* (probability that the system is accessible to its users) and of its *reliability* (probability to successfully carry out the task connected to a given request, within a suitable maximum time frame) [6,47]. Guaranteeing a high dependability level for a given SOA-based CBP implementation is a key factor for its success in the envisioned competitive world, where different implementations may co-exist with different QoS and cost attributes [8,29]. Thus, in this chapter we discuss issues concerning the realization of a SOA system that implements a collaborative business process, with the goal of meeting some specified dependability requirements.

In this respect, we note that such a system will typically operate in an evolving environment, where providers may modify the exported services, new services may become available, existing services may be discontinued by their providers. A promising way to cope with these issues is to make the system able to self-configure in response to changes in its environment (*e.g.*, available resources, type and amount of user demand). In this way, the system can timely react to (or even anticipate) environment changes, trying to use at best the available resources, thus avoiding long service disruptions due to off-line repairs [17]. Hence, we also include in our discussion issues concerning the realization of a dependability-driven self-configurable SOA system.

Methodologies to assess the QoS of a SOA system and to drive its self-configuration have been already presented. Some of them specifically focus on the fulfillment of dependability requirements (*e.g.*, [22,49]), while others consider multiple QoS attributes including dependability (*e.g.*, [5,10,12,15,14,35,46,47]). Most of these methodologies base the self-configuration on the runtime selection for each CBP task of a single service that implements it, to which that task will be dynamically bound. The methodologies presented in [15,14,22,49] extend this idea consider-

ing the possibility of selecting at runtime redundant implementations for each CBP task, based on existing functionally equivalent services, to improve the system ability to meet a given dependability requirement.
However, these proposals are based on assumptions (often only implicitly stated) that restrict the class of CBPs they can be applied to. For example, they generally do not consider the case of long-running CBPs made of multiple atomic transactions with possibly different dependability requirements. Moreover, when redundancy-based implementations are considered for CBP tasks, they do not discuss the impact that different failure modes could have on the effectiveness of these implementations, thus basically assuming a single failure mode.
Building upon these proposals, in this chapter we present a general modeling framework to architect a dependability-driven SOA system that implements a CBP with, possibly, self-configuration features. The framework allows us to take into account in a unified way:

- CBPs consisting of multiple transactions;
- different CBP utilization scenarios (single user requests versus sustained flow of requests generated by different users);
- different failure modes for the partners of a CBP;
- CBPs with stateless/stateful tasks.
- CBPs with centralized/distributed self-configuration management.

The chapter is organized as follows. In Sect. 2 we present a reference model of CBP that we use as basis for our discussion on dependability-driven CBP configuration. In Sect. 3 we discuss configuration actions that can be performed to meet dependability goals of a CBP. In Sect. 4 we outline the mathematical formulation of a system model that can be used to drive the configuration of the CBP and discuss related issues. In Sect. 5 we discuss issues concerning the design of an architecture that can support the self-configuration of a SOA system and describe some decentralization issues regarding the architectural style of the system. Finally, Sect. 6 concludes the chapter.

2 Model

In this section, we define the CBP model we refer to, the failure model we consider in our discussion about dependability impairments, and the contract model used for the specification of the respective obligations and expectations of service users and providers.

2.1 Collaborative Business Process Model

An *abstract CBP* consists of:

- a set of *tasks*;
- a set of *roles*;
- a set of *atomic transactions*;
- a *collaboration scheme*.

In this definition, each *role* consists of one or more *tasks* that must be performed to carry out the collaboration. In a SOA based implementation, we assume that a task corresponds to the execution of an operation belonging to some service interface[1]. The roles define a partition of the overall set of tasks. We assume that the partitioning of tasks into different roles implies the existence of some kind of logical relationship among tasks belonging to the same role, that actually corresponds to the sharing of some state information. According to this model, a task that has not any such relationship with other tasks of the CBP corresponds to a role consisting of that task only.

Tasks are also partitioned into distinct *atomic transactions*, where all the tasks belonging to the same transaction must be executed according to an all-or-nothing rule. Typically, an atomic transaction consists of a subset of the overall set of CBP tasks, possibly belonging to different roles, that engage in a short-running collaboration. Thus, an overall short-running CBP corresponds to a single atomic transaction, but, in general, we want to consider long-running CBPs consisting of several atomic transactions, that possibly need not to be all successfully completed, or completed within a single time window, to consider the overall CBP successfully completed [33].

Finally, the *collaboration scheme* specifies how the tasks are composed, according to some *composition rules*. Typical composition rules are: (i) sequence, (ii) conditional selection, (iii) loop, and (iv) parallel.

To make clearer the meaning of the different elements of this CBP model, we propose in Tables 1 and 2 a mapping from these elements to the specific terminology of two well-known languages for CBP specifications, the Web Service Choreography Description Language (WS-CDL, [41]) and the OMG's Business Process Modeling Notation (BPMN, [31]), which is becoming the de-facto standard for modeling intra-organizational processes. For a comprehensive overview of business process modeling languages, we refer the reader to [27], while we refer to [19] for an identification of the key requirements of service choreography languages, along with their assessment.

To be actually carried out, an abstract CBP must be mapped to a *concrete CBP* that implements it, consisting of:

- a set of *participants*
- a set of *concrete tasks*
- a *task-to-implementation mapping*

Each *participant* provides services (*concrete tasks*) that implement tasks belonging to one (or possibly more) of the specified roles. The *task-to-implementation mapping* maps each task of the abstract CBP to an implementation based on the services offered by the CBP participants. Given the meaning of a role in the CBP model we are considering, we assume that any such mapping must satisfy the following constraint: *tasks belonging to the same CBP role must be bound to services offered by the same participant.*

[1] We note that the *user* (or *client*) of a given CBP can be considered as a special case of role. Tasks belonging to this role could include starting the CBP, and collecting some final result.

Table 1. Abstract model concepts and mapping with the constructs of WS-CDL and BPMN

abstract model	WS-CDL	BPMN
task	`activity`	`Activity`
role	`roleType`	`PartnerRole`
atomic transaction	`workunit`[a]	`Transaction`

[a] Transactions are not explicitly addressed in WS-CDL, but some facility can be used to satisfy some basic transaction properties [39].

Table 2. Composition rules of the abstract model and mapping with the constructs of WS-CDL and BPMN

Rule	WS-CDL	BPMN
sequence	`sequence`	`Sequence Flow`
structured loop	`repeat` and `guard` attributes of workunit	`Activity Looping`, `Sequence Flow Looping`
conditional selection	`choice`	`Exclusive Gateway`
parallel	`parallel`	`Parallel Gateway` (fork and join)

As an example of the rationale for this constraint, think for example of a *Provider* role in some e-commerce CBP, that includes the *orderFulfillment* and the associated *Shipping* tasks. It would make little sense to assign the former task to one participant, and the latter one to a different one, which has received no order (and no money!) for the good it should deliver.

The discussion of how to devise dependability-driven methodologies for the definition, possibly in an automatic way, of mappings from an abstract CBP to a suitable concrete CBP, is the main goal of this chapter. Besides fulfilling some dependability (and cost) requirements, the methodologies we consider aim also at maximizing some suitable utility function. We also discuss issues related to the use of such methodologies to support the self-configuration of the overall system implementing the CBP. In this case, the system is intended to define by itself at runtime the mapping between each task of the abstract CBP and some corresponding implementation, dynamically modifying this mapping if some change occurs that makes the previous mapping no longer suitable for the new environment.

2.2 Failure Model

Software systems may fail according to different *failure modes*. These failure modes can be characterized with respect to different viewpoints (we refer to [6] for a thorough discussion of this issue). With respect to the *failure domain* viewpoint, some relevant failure modes are:

- *content* failures, where the content of the service output deviates from the correct one, given the input provided to the service;
- *timing* failures, where the delivery time of the service output deviates from the correct one, given the time when input was provided to the service;
- *halt* failures, where no response is received at all (they could be considered as simultaneous content and timing failures).

Given these domain-based failure modes, a useful *mode-dependent* dependability measure can be defined as follows:

- *reliability*: the probability that, when invoked, the service completes its task correctly with respect to a given failure mode[2].

On the other hand, a *mode-independent* dependability measure is defined as follows:

- *availability*: the probability that the service is accessible and ready to accept user requests.

We note that, according to these definitions, reliability implies availability, in the sense that that to deliver a correct output the system must be available and ready to accept the corresponding input. On the other hand, availability does not necessarily imply reliability, as accepting an input does not guarantee that the corresponding output will be correct.

With respect to the *detectability* viewpoint, relevant failure modes are:

- *signaled* failures, where some detection system is able to check the correctness of the delivered service;
- *unsignaled* failures, where such a detection system does not exist.

Considering the detectability viewpoint, we assume that, in a SOA environment, it mainly refers to the ability of an external observer (different from the service provider: it could be, for example, the service user) to detect the occurrence of a failure during the execution of a service. Detecting a timing or halt failure is straightforward for such an observer, and hence we can assume that they are always signaled. More questionable could be to assume that content failures are always signaled, as it could not be simple for the external observer to devise some function able to check the correctness of the delivered output (in that case, it can only rely on the ability and willingness of the service provider to signal such a failure). We further discuss this issue in the next subsection on contract definition, from the viewpoint of the measurability of the dependability measures defined above.

Finally, with respect to the *consistency* viewpoint, relevant failure modes are:

- *consistent* failures, where all the service users perceive the same (correct or incorrect) output;
- *inconsistent* failures, also known as *Byzantine* failures, where different users may perceive different kinds of output.

[2]This measure is called *successful execution rate* in [47].

2.3 Contract Definition

The overall dependability and cost of a particular concrete CBP depend on the dependability and cost of the services provided by the CBP participants. In our framework, we assume that the involved parties regulate their interactions and state the required dependability and cost values in a Service Level Agreement (SLA), *i.e.*, a contract that explicitly states the respective obligations and expectations [18]. This contract specifies the conditions for service delivery, its cost, duration, and penalties for non-compliance.

The SLA model we consider is related to the CBP model presented in Sect. 2.1, where we have assumed that a CBP is partitioned into a set of transactions. We may think that, in general, not all the transactions within a CBP require the same type of dependability guarantees: some of them could correspond, for example, to "optional" parts of a CBP that are not strictly required for the overall CBP to be considered successfully completed. For example, this could be the case of a travel insurance transaction within a travel planner CBP, which could be only optionally required by a user of this CBP. Hence, the SLA model we consider consists of:

- a *global* SLA, stating requirements for the overall CBP implementation;
- a set of *local* SLAs, one for each transaction of the CBP, where each SLA states requirements for the corresponding transaction.

In general, a SLA definition may include a large set of parameters, referring to different kinds of functional and non-functional attributes of the service/process it refers to, and different ways of measuring them (*e.g.*, averaged over some time interval) [18,40]. In this chapter, we focus on dependability requirements, concerning reliability and availability, and the corresponding cost the user is willing to pay for them. Hence, the SLA definition we consider includes the following parameters (for both the local and global SLAs):

- a_{min}: a lower bound on the transaction/CBP availability expected by its user;
- r_{min}: a set of lower bounds on the transaction/CBP reliability expected by its user (one bound for each different failure mode considered in the SLA);
- c: the unitary service cost paid by a transaction/CBP user for each submitted request.

For a dependability parameter to be included in a SLA, it should necessarily be measurable by all the parties involved in the SLA (or by some trusted third party), to avoid disputes about non compliance. In this perspective, we note that the act of submitting a request to a service directly implies for the submitting entity (the service user) the possibility of detecting the occurrence of a timing or halt failure, and, provided that a suitable checking function exists, also the occurrence of a content failure. This means that reliability with respect to halt and timing failures (and content failures, with the indicated limitation) is a directly *measurable* dependability attribute for both the provider and the user of a service. Hence, it can be safely introduced in a SLA between service

user and provider. On the other hand, availability can be hardly detected in the absence of an explicit request addressed to a service, or an explicit notification from the service itself of its transitions between the available/unavailable states. We should note that addressing requests to a service just to check its availability, without any actual need of that service, could be too costly for several reasons. Hence, the inclusion of availability in a SLA should be carefully considered. We refer to [38] for a thorough discussion about these issues.

The global SLA associated with an overall CBP also includes the following item, besides the dependability attributes listed above:

- a logical predicate on the successful/unsuccessful completion of the CBP transactions, stating transaction completion patterns that correspond to the successful completion of the overall CBP.

The logical predicate is specified using some suitable logic notation, from simple Boolean operators to more expressive notations, like some kind of temporal logic (*e.g.*, LTL [34]).

We have outlined in the introduction that two different utilization scenarios could be considered, corresponding to a single request addressed by a user, or an entire flow of requests addressed by one or more users. In the former case, the SLA parameters mentioned above refer to the single request under consideration, irrespective of other requests concurrently addressed to the same system. In the latter case, a_{min} and r_{min} must be intended as calculated over the entire flow of requests, while c still refers to the cost for each request in the flow[3].

For the "flow of requests" utilization scenario, it seems quite unreasonable to state in a SLA dependability and cost requirements irrespective of the load generated by the user. Hence, for this scenario, we assume that the SLA also includes the following additional parameter:

- L: an upper bound on the load the user is allowed to submit, expressed in terms of average rate of requests (requests/time unit).

We assume that the parameters of a SLA defined according to this model are the result of a negotiation between each prospective user of the CBP or of some of its parts (*i.e.*, some specific transaction of that CBP) and the configuration entity that manages it. Hence, several SLAs of this kind can co-exist at a given time interval and may have, in general, different values for these parameters. However, it is possible that the managing entity proposes to the CBP/transaction[4] user(s) a predefined set of differentiated service levels, to drive the user indication of a service level.

All these co-existing SLAs define the dependability objectives that the CBP managing entity must meet in that interval, provided that (in case of the "flow of requests" scenario) the flow of requests generated by the users in that interval does not exceed the limits stated by the L values in the existing SLAs. Moreover, they also define the expected cost for the

[3] Alternatively, c can also correspond to the flat price for the overall flow of requests.

[4] To simplify the notation, in the following we will write CBP rather then CBP/transaction, when we discuss issues that applies to CBPs as well as to single transactions of a CBP. However, unless explicitly specified, it should be intended that our discussion concerns both of them.

the CBP use (and, correspondingly, the expected income for some CBP "owner").
To meet these objectives, the CBP managing entity must try to exploit at best the services provided by the CBP participants. To this end, we assume that a SLA has been negotiated with each of these participants, stating the QoS and cost parameters of each service they offer to implement some CBP task. As, in general, these services are opaque (their internal organization is not known), they can be considered equivalent to a single transaction. Hence, the SLAs negotiated with their providers are defined according to the local SLA model presented above.
The set of all these SLAs defines the constraints within which the CBP managing entity can organize a suitable (self-)configuration policy able to meet the SLA negotiated with the CBP users.

3 Configuration Actions

We discuss in this section the configuration actions that can be performed by the managing entity of a SOA system implementing a given CBP, to make it able to meet the dependability requirements of its users, stated in their SLAs.
Each request addressed by a user generates a corresponding set of one (or more) request(s) for each CBP task. To configure a SOA system means to bind these latter requests to suitable concrete implementations, based on services provided by the CBP participants. Hence, to devise an effective configuration policy we have to consider the following two issues: (a) identification of a set of possible concrete implementations for each CBP task; and (b) selection within this set of the implementation better suited to meet a given dependability requirement. We discuss these two issues in the following two subsections, respectively.

3.1 Task Implementation

Several existing methodologies only consider implementations consisting of a single service provided by some CBP participant ([5,10,12,35,46,47]). In this case, the set of possible implementations for a given task corresponds to the set of functionally equivalent services implementing that task, provided by the (candidate) CBP participants.
However, it is possible that a user arrives with high dependability requirements, which cannot be satisfied by any single service. Rather than rejecting this user (which could cause an income loss and/or a reputation decrease), other possible actions could be tried:

1. to identify additional participants, implementing the same task with higher dependability;
2. to "increase" the dependability which can be attained exploiting the services provided by the already identified participants.

The former action has two drawbacks. It requires additional effort to discover such participants and negotiate with them suitable SLAs. Worse yet, such participants could not even exist.

The latter action does not suffer from these drawbacks. It is based on the idea of using *redundancy* schemes to get a dependability level higher than that guaranteed by each single service, at the expense of a higher cost (basically equal to the sum of the costs of all the invoked services). According to these schemes, a request for a task is logically bound to a set of two or more services implementing it (coordinated according to some redundancy pattern), rather than to a single one. The full potential of this approach can be exploited in a SOA environment where more participants are available, providing different implementations for the same task [16]. However, it can also be exploited in a more limited form when only one participant is available, that offers an implementation for a specific task. In this respect, we point out that, according to a service-oriented perspective, we are talking here of redundancy schemes implemented and managed by some entity *external* to each service, to get a dependability level higher than that guaranteed by the provider of that service. The provider of each service could use as well redundancy schemes for its *internal* implementation of that service, to get that dependability level, but their use is hidden to an external observer (that hence cannot control their configuration).

In this chapter, we consider the following redundancy schemes for the implementation of CBP tasks, assuming that they are sufficiently representative for a discussion of some relevant issues (we refer to [45,16,49] for a more thorough description of redundancy schemes in a SOA environment):

- *local retry*: sequentially repeated execution of a task by the same participant (up to a pre-defined maximum), until correct service is delivered, or the maximum number of executions is reached[5];
- *non-local retry*: sequentially repeated execution of a task by different participants, until correct service is delivered, or the number of different participants is reached;
- *parallel-or*: parallel execution of a task by different participants, taking as result the first delivered correct output;
- *majority voting*: parallel execution of a task by different participants, taking as result the output delivered by the majority of the participants.

Given these redundancy schemes, we can formalize as follows the set of implementations we can potentially consider for a task T_i belonging to a given CBP. Let us denote by $\mathcal{K}_i = \{k_{i1}, k_{i2}, \ldots, k_{in_i}\}$ the set of functionally equivalent services implementing T_i, provided by the CBP participants. The overall set of all the possible implementations for T_i can be described as the union of the following sets:

- *local retry*: the set k_i^+ of all the sequences k_{ij}^n ($k_{ij} \in \mathcal{K}_i, n \geq 1$); a given sequence k_{ij}^n means that service k_{ij} is tried sequentially at most n times;
- *non-local retry*: the set $\mathcal{K}_i^+$ of all the ordered lists of elements belonging to $\mathcal{K}_i$, where each element appears at most once (excluded the empty list); a given sequence means that the listed services are tried sequentially starting from the first one;

[5]This technique is the only one that can be meaningfully used when only one participant is providing the implementation of a given task.

- *parallel-or*: the set $\mathcal{K}_i^{par}$ of all the subsets of elements belonging to $\mathcal{K}_i$ (excluded the empty set); a given subset means that services in it are activated in parallel;
- *majority voting*: the set $\mathcal{K}_i^{vot}$ of all the subsets of elements belonging to $\mathcal{K}_i$ with odd cardinality (greater than or equal to three); a given subset means that services in it are activated in parallel, with majority voting on the delivered results.

Existing methodologies for the dependability-oriented self-configuration of SOA systems (*e.g.*, [15,14,16,22,24,49]) consider the whole union of these sets (or some variant of it, depending on the considered redundancy schemes) as the set from which an implementation of a given task should be selected. They basically drive the selection process by identifying within this set the subset of those implementations able to guarantee the required dependability level, and then discriminating among different dependability-equivalent implementations only on the basis of the respective cost (and possibly performance penalty). As an example, for the same dependability level, a non-local retry implementation costs less than a parallel-or implementation, as it involves on the average the use of less services, but causes on the average a higher task completion time.
However, the union of the sets listed above only describe *potential* implementations of a CBP task. Existing methodologies do not take into account some relevant issues, that can actually lead to a restriction of the set of possible implementations to be considered for a given task. They concern the effectiveness of the considered redundancy schemes with respect to different *failure modes*, and their utilization in the presence of *stateful* tasks (that, in our CBP model, correspond to tasks belonging to the same multi-task role). We discuss below these two issues.

Redundancy Schemes under Different Failure Modes As discussed in Sect. 2.2, failures that occur in a system and affect its dependability can be classified according to several modes. Different redundancy schemes may have different effectiveness, depending on the failure mode they have to cope with. Table 3 summarizes the effectiveness of the considered redundancy schemes with respect to the failure domain viewpoint. In this respect, we point out that both the retry and parallel-or schemes are based on the ability of explicitly detecting the occurrence of a failure. As discussed in Sect. 2.2, this ability could not be guaranteed in a SOA environment for content failures, contrarily to timing and halt failures. As a consequence, we have remarked in Table 3 the possibly limited effectiveness of these two redundancy schemes for content failures.
Besides this, we also remark that the local-retry scheme is actually effective only in case of short transient failures of a service (in case of long service disruption, it makes no sense to try again with that same service). Hence, unless there is only one participant implementing that task (in that case no other scheme can be used), it could be better to prefer the other schemes, that exploit the design/implementation diversity offered by different participants.

Finally, we point out that none of the redundancy schemes considered here is able to cope with inconsistent failures in a SOA distributed environment. In this case, other, Byzantine-tolerant techniques should be used (*e.g.*, [48]).

Table 3. Redundancy schemes with respect to some failure domains

	content failure	**timing failure**	**halt failure**
local retry	effective only if signaled; ineffective if unsignaled	ineffective (as it adds additional delays)	effective provided that it does not violate timing requirements
non-local retry	effective only if signaled; ineffective if unsignaled;	ineffective (as it adds additional delays)	effective provided that it does not violate timing requirements
parallel or	effective only if signaled; ineffective if unsignaled	effective	effective
majority voting	effective	effective (but more costly than parallel-or)	effective (but more costly than parallel-or)

Redundancy Schemes with Stateful Tasks We recall from Sect. 2.1 that we have assumed that all the tasks belonging to the same role form a set of related *stateful* tasks, sharing some state information. Let us consider a set of tasks belonging to the same role, to be sequentially executed, and let us assume that for one of these tasks an implementation is selected based on the non-local retry scheme. If the j-th invoked service implementing the task succeeds, we are forced to use one implementation provided by the same participant for the next task of that role, according to the stateful assumption. Hence, it is infeasible to adopt also for that next task a non-local retry scheme, to get a higher dependability level. Analogous considerations hold for the other considered redundancy schemes, except for local retry.

To the best of our knowledge, only the methodologies proposed in [5,14,50] consider explicitly this issue. In [14] we deal with it by simply excluding at all the possibilities of using redundancy schemes involving multiple providers to improve the dependability of related stateful tasks.

If we want instead to exploit such schemes also for stateful tasks, we should probably adopt a "per role" viewpoint, instead of the "per task" viewpoint adopted by all the existing methodologies we are aware of. This means that we should select different participants, asking each of them (in sequence or in parallel, depending on the redundancy scheme) to execute the whole set of tasks of a given role. To pursue this approach, it is necessary: (a) at the methodological level, to extend existing methodologies based on the "per task" viewpoint, to compositionally calculate the

whole dependability of a CBP implementation, based on the dependability of each whole role implementation; and (b) at the architectural level, to coordinate the interactions of different participants implementing the same role with other tasks involved in the same transaction.

3.2 Task Implementation Selection

Based on the considerations discussed above, the CBP managing entity identifies the actual set of possible implementations to be considered for each task T_i of a given CBP. Once these sets have been identified, the managing entity configures the CBP by determining a suitable *task-to-implementation mapping*. This corresponds to selecting within these sets the implementation of each T_i, taking into account the SLA negotiated with the CBP user(s), and the SLA negotiated with the CBP participants providing the k_{ij}'s services. According to the SLA model outlined in Sect. 2.3, the SLA negotiated with the provider of a service k_{ij} can be denoted by the tuple $\langle a_{ij}, [r_{ij}]^*, c_{ij}, (L_{ij})\rangle$, as it corresponds to a single transaction. In this tuple, a_{ij} is the agreed on bound on the k_{ij} availability, $[r_{ij}]^*$ is a list of agreed on bounds on its reliability with respect to a given list of different failure modes, c_{ij} is the service cost, and L_{ij} is the bound on the load the user is allowed to submit to the service.

Analogously, the SLA negotiated with a user u can be denoted by a set of tuples $\langle a^u_t, [r^u_t]^*, c^u_t, (L^u_t)\rangle$, defining the local SLAs (one per each transaction t belonging to the CBP), plus a tuple $\langle a^u, [r^u]^*, c^u, (L^u), P\rangle$ defining the global SLA, where P is the predicate on the transactions completion introduced in Sect. 2.3[6].

For the sake of generality, we are assuming in these schemes that each CBP participant could manifest different failure modes, and provide possibly different dependability guarantees for each of them. Correspondingly, a user could have different dependability requirements with respect to different failure modes. If this information is missing, the managing entity could make, for example, more or less conservative assumptions about the failure mode it should cope with, depending on the dependability level it is willing to achieve.

The implementation selection performed by the CBP managing entity actually corresponds to two different actions, depending on the envisioned utilization scenario.

In a *single request* scenario, where the dependability requirements of a single request addressed to the CBP must be fulfilled, the implementation selection corresponds to a *0-1 choice* of one implementation for each task, from the available ones.

In a *flow of requests* scenario, instead, we have to consider simultaneously all the requests belonging to the flow generated by each CBP user. Hence, the implementation selection action corresponds in this case to determining, for each CBP task, which is the *fraction* of the overall set

[6] In the SLA templates, we put in parentheses L_{ij}, L^u_t and L^u as these parameters could be absent in a SLA concerning a single request.

of requests generated for that task by a user that will be bound to a given concrete implementation.
For this scenario, we point out that, as the CBP managing entity could deal simultaneously with several users having different requirements, requests coming from different users are likely to be routed differently to the available implementations. For requests coming from the same user, it is possible as a special case that all the requests for a task are routed to a single implementation, but in general it may happen that subsets of these requests are routed to different implementations.
In the next section we outline how a mathematical model can be formulated to determine a suitable selection of implementations for the CBP tasks.

4 A Model to Drive CBP Configuration

We call a *configuration policy* a decision taken by the CBP managing entity about the implementation(s) to be bound to each CBP task for the request (flow of requests) generated by a given user $u \in U$. We can model this policy by a vector $\boldsymbol{x}^u = [x_1^u, \ldots, x_m^u]$, where $x_i^u = [x_{iJ_i}^u]$. In this definition, index i of $x_{iJ_i}^u$ ranges over the set of all the CBP tasks T_i, while index J_i ranges over all the possible implementations of T_i, determined according to what discussed in Sect. 3.1.
The $x_{iJ_i}^u$ variables are defined in two different ways, depending on the considered utilization scenario:

- *single request* scenario: each $x_{iJ_i}^u$ takes only the 0 or 1 value, where $x_{iJ_i}^u$=1 means that implementation J_i is selected for T_i;
- *flow of requests* scenario: each $x_{iJ_i}^u$ takes any value in the $[0, 1]$ interval, and denotes the fraction of the user u requests for task T_i which are bound to implementation J_i.

In both cases, it holds the constraint: $\sum_{J_i} x_{iJ_i}^u = 1$.
As an example of the meaning of the $x_{iJ_i}^u$ variables in the flow of requests scenario, consider the case of four concrete services $S_{i.1}, \ldots, S_{i.4}$ offered by different participants implementing a given task T_i and assume that the policy $\boldsymbol{x}_i^k$ for a given user u specifies the following values: $x_{i.\{S_{i.1}\}} = 0.3$, $x_{i.\{S_{i.3}\}} = 0.3$, $x_{i.\{S_{i.2},S_{i.4}\}^{par}} = 0.4$ and $x_{i.J_i} = 0$ otherwise. This policy implies that 30% of user u requests for task T_i are bound to service $S_{i.1}$, 30% are bound to service $S_{i.3}$, while the remaining 40% are bound to a parallel-or implementation based on the pair of services $\{S_{i.2}, S_{i.4}\}$ (see Fig. 1).
Determining a suitable value for the $x_{iJ_i}^u$ variables can be formalized as the solution of an optimization problem, which takes the following general form:

$$\begin{aligned} \textbf{max } & F(\boldsymbol{x}) \\ \textbf{subject to: } & Q^{\alpha}(\boldsymbol{x}) \leq Q^{\alpha}_{\max} \\ & Q^{\beta}(\boldsymbol{x}) \geq Q^{\beta}_{\min} \\ & S(\boldsymbol{x}) \leq L \\ & \boldsymbol{x} \in A \end{aligned} \tag{1}$$

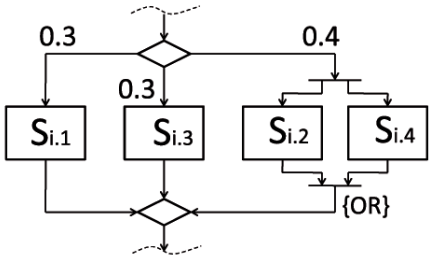

Fig. 1. Example of adaptation policy

In this model, $\boldsymbol{x} = (\boldsymbol{x}^1, ..., \boldsymbol{x}^{|U|})$ is the decision vector defined above, $F(\boldsymbol{x})$ is a suitable utility function, $Q^{\alpha}(\boldsymbol{x})$ and $Q^{\beta}(\boldsymbol{x})$ correspond, respectively, to SLA parameters whose values are settled as a maximum and a minimum (typically, they correspond to cost and dependability parameters, respectively), $S(\boldsymbol{x})$ is the constraints on the offered load determined by the SLAs with the service providers (in case of flow-based SLAs), and $\boldsymbol{x} \in A$ is a set of functional constraints.

The latter set of constraints includes the constraints $\sum_{J_i} x^u_{iJ_i} = 1$. It in general includes other constraints on the $x^u_{iJ_i}$ values, that could be used, for example, to take into consideration the *stateful* nature of tasks belonging to the same role, according to the CBP model of Sect. 2.1. Example of this kind of constraints can be found in [5,14].

Depending on the considered utilization scenario, we point out that solving this model corresponds to solving an *integer programming* optimization problem (for the single request scenario), or a *linear programming* optimization problem (for the flow of requests scenario). This model (or variants of it) underlies most of the proposed methodologies for the self-configuration of SOA systems [5,12,15,14,24,46,47,50]. These methodologies differ in the proposed solution techniques (in particular for the single request scenario, where heuristics are often proposed to cope with the NP-hard nature of integer programming), and in the way the model parameters (*e.g.*, Q^{α} and Q^{β} matrices) are calculated, that basically depends on the considered QoS metrics, composition rules for the CBP tasks, and redundancy schemes.

Self-configuring at runtime a SOA system corresponds to building at runtime a new instance of this optimization problem, and solving it, to determine the $\boldsymbol{x}$ value that describes the new system configuration. The construction and solution of this new instance is triggered, in general, when the managing entity detects some relevant event for which the current configuration is no longer suitable. For example, such events could include: *a*) a change in the utilization profile of the CBP tasks; *b*) a change in the CBP definition, because tasks and/or participants are added or removed; *c*) a detected violation in the negotiated SLA parameters; *d*) the arrival of a new user who submit requests for the business process. Existing methodologies consider this model as a single global model of the overall CBP, and hence implicitly assume some kind

of centralized managing entity, that maintains this model (for example keeping up to date its parameters) and solves it to drive the system self-configuration.

Indeed, most methodologies for the self-configuration of SOA systems are based on an underlying orchestration model, in which the execution of a target composite service requires the selection and runtime binding of a number of concrete implementations for realizing the functionalities of the business process, but these selected implementations do not interact with each other, *e.g.*, [5,15,24,46,47]. On the other hand, in a choreographic environment the participants may be or not willing to rely on a centralized managing entity that may have a global view and therefore chooses the collaborators of each participant. When the choreography participants want to maintain a decisional autonomy without delegating the configuration decisions to a centralized entity, for example because they do not prefer to disclose their collaborators/partners, they can rely on a more limited information regarding only the collaborating participants with whom each participant directly interacts. To the best of our knowledge, only the service selection methodology in [23] explicitly addresses a choreography environment that supports local autonomy requirements of the CBP participants with the aim to maximize the CBP reliability. However, this approach suffers from the partial versus global tradeoff, because each participant takes its decision only on the basis of some local knowledge; therefore, QoS properties regarding the overall CBP can be hardly satisfied. On the other hand, a single centralized entity owns detailed information about all implementations and therefore can meet at best the global QoS requirements of the CBP.

Another limitation of self-configuration methodologies for business processes regards the type of failure modes taken into account, because only a single failure mode (either timing or halt mode) is typically supported by the proposed methodologies.

Finally, most methodologies for the self-configuration of SOA systems consider a CBP with only a single transaction. However, as discussed in Sect. 2.1, a CBP could consist of multiple transactions, with possibly different dependability requirements. Moreover, the SLA model we have introduced in Sect. 2.3 suggests that different CBP users could have different views of the same CBP, corresponding to different definitions of the logical predicate on the transactions completion stated in their global SLA. This implies that multiple instances of the optimization problem should be probably simultaneously considered to determine the configuration of a given CBP, corresponding to different transactions and/or to different views. Each of these different instances of the optimization problem could refer to a subset of the $x^u_{iJ_i}$ variables (for example, only those referring to the tasks of a specific transaction of the overall CBP). These instances can be considered as partially uncoupled. They could be still managed by a single centralized entity, but considering the possibility of distributing their management among more entities would be probably closer to the principles of the SOA paradigm.

5 Architectural Issues

In the previous sections we discussed the main issues that should be tackled to guarantee a given dependability level for a SOA-based CBP implementation. Here we discuss the architectural issues involved in the design of the managing entity that implements the CBP self-configuration methodology presented in Sects. 3 and 4. The main task of this architecture is to drive the adaptation of the CBP it manages to fulfill the QoS goals stated in the SLAs with the CBP users, given the SLAs it has negotiated with each of the CBP participants that provide implementations of the used concrete tasks. Moreover, the CBP managing entity also aims at optimizing a global utility goal.

A key design issue regards the self-adaptation capabilities of the CBP managing entity. To this end, we organize its architecture according to the MAPE (Monitor, Analyze, Plan, and Execute) feedback control loop [37]. The components of the CBP managing entity - namely, the *Collaboration Manager*, the *Execution Engines*, the *Configuration Manager*, the *Coordination Manager*, the *Execution Monitor*, the *Admission Control Manager*, and the *SLA Monitor* - are therefore organized according to the MAPE stages. We first present the managing entity components from a functional point of view and then briefly discuss in Sect. 5.1 the alternatives for their topological organization considering the trade-off between centralized and decentralized architectural style of the system.

The *Execute* subsystem of the MAPE loop, which comprises the components in charge of executing the business logic, includes the Collaboration Manager, the Execution Engines, and the Coordination Manager.

The main functions of the *Collaboration Manager* are the specification of the collaboration scheme (*i.e.*, the choreography) with a suitable notation (*e.g.*, BPMN or WS-CDL), the discovery of the set of functionally equivalent services implementing the CBP tasks, and the negotiation and establishment of the SLAs with the corresponding set of participants. In this context, the Collaboration Manager should also specify different types of transactional behavior, *e.g.*, atomic or long-running transactions.

The *Execution Engines* are the software platforms (*e.g.*, Activiti BPM Platform [2] and Bonita Execution Engine [11] for BPMN, Apache ODE [4] and Oraclw BPEL Service Engine [32] for BPEL) managed by the CBP participants where the tasks of the collaborative process are executed. Since the CBP execution is typically distributed across different organizations, the global specification defined by the abstract business process needs to be implemented by the participants in a distributed way. Therefore, the global specification is usually decomposed into a set of local specifications, that are implemented and deployed by each involved participant (each participant implements its local specification typically using BPEL, but not necessarily so). When the user invokes the collaborative business process, the Execution Engines manage altogether a new executable instance of the process itself. Each generated instance can be different, according to the configuration instructions received by the Configuration Manager (described below).

The *Coordination Manager* provides the foundation for implementing transactional service interactions by defining the coordination context of a transaction and the protocols for registering services therein. To this end, the Coordination Manager implementation can exploit the WS-Coordination specification [30], which is a general transaction framework that describes the protocols for participant registration and defines a transaction context.

The *Configuration Manager* is the core component of the *Plan* subsystem of the MAPE loop, since it decides for the runtime configuration of the collaborative business process. Upon receiving a notification of a significant variation of the system model parameters, it finds out whether new configuration actions must be performed. To this end, it determines the configuration policy (*i.e.*, it solves a new instance of the optimization problem sketched in Sect. 4), passing to it the new instance of the system model with the new values of the system parameters. The calculated configuration policy provides indications about the configuration actions that must be performed to optimize the use of the available concrete tasks with respect to the global utility criterion. Based on this solution, the Configuration Manager transmits suitable directives to the participants, that implement them through their Execution Engines, so that future instances of the collaborative business process will be generated according to these directives. In the CBP management we envision, the selection of participants is done at runtime; therefore, it must be ensured that the participants are made aware dynamically of the selection. To this end, we note that a drawback of WS-CDL and BPMN is that service selection is not fully supported [19] and is often left to engine-specific deployment configurations; BPEL4Chor, which is a BPEL extension for modeling service choreographies [19], includes a `selects` attribute for `participant` that can turn out useful in our case to implement the runtime binding, since it allows to specify which service selects which other services.

The latter three components of the managing entity (*i.e.*, Execution Monitor, SLA Monitor, and Admission Control Manager) form collectively the *Monitor* and *Analyze* subsystem of the MAPE control loop, which checks the system execution and senses the environment and, when something is not proceeding as planned, triggers the *Plan* subsystem to fix the detected anomaly. Specifically, the *Execution Monitor* collects information about the CBP usage, calculating estimates of the model parameters. The *SLA Monitor* collects information about the dependability level perceived by the CBP users and offered by the participants that provide implementations of the used concrete tasks, and about the requests generated by the users. Both the Execution Monitor and the SLA Monitor rely on a hierarchical organization, according to which monitoring agents located at the participants collect local information and transmit it to the respective global component which is responsible for aggregating the monitored data on the managing entity. The *Admission Control Manager* determines whether a new CBP user can be accepted, given the associated SLA, without violating existing SLAs for already present users.

On the one hand, the Execution Monitor, SLA Monitor, and Admission Control Manager components play the Monitor role of the MAPE loop, because they check and maintain up to date the parameters of the model of the CBP operations and environment. These parameters include the invocation frequencies of the concrete tasks, the rate of arrival of service requests (in case of the flow of requests scenario), the dependability and cost of the used concrete tasks. On the other hand, the three components also play the Analyze role: when they observe significant variations in the model parameters, they signal these events to the Configuration Manager. Summing up, the Admission Control Manager (in case of the flow of requests scenario) signals events related to the fluctuation of workload intensity parameters, while the Execution Monitor signals abnormal fluctuation in the CPB usage, and the SLA Monitor signals abnormal events, such as unreachability of a concrete task and/or variation of its dependability level.
We observe that the monitoring subsystem plays a crucial role to keep track of the CBP behavior and find out whether anomalies have occurred and a new configuration plan is needed. While business process monitoring can be more easily achieved in a service orchestration context, where there is a centralized entity in charge of the synchronization of the component services (*e.g.*, [9] for monitoring of BPEL processes), the monitoring of CBPs in a multiple organizational setting present additional challenges, because it may require that different participants interchange monitoring data. A recent research effort toward this direction can be found in [42], that proposes an event-based monitoring approach based on BPEL4Chor.

5.1 Centralized vs Decentralized Architecture

We currently devise the CBP managing entity as a single centralized broker (except the Execution Engines and the local monitoring agents, that are both located on the participants), which has a complete knowledge of the system model and therefore can identify the best configuration policy. Such a centralized approach may suffer from the single point of failure and scalability issues that can be addressed by distributing and replicating the components of the managing entity so to not impact on the managed CBP dependability and scalability. A similar architecture in the context of a broker providing a single composite service is described in [13].
More generally, we can envision a CBP managing entity architected as a decentralized system consisting of a set of federated brokers, where the distribution and replication of the components take place at the level of the MAPE subsystems rather than at the level of the single components. In this architecture, the brokers coordinate themselves according to a *master-slave* scheme. The slave brokers implement only the Monitor and Execute functions of the MAPE loop, while the master broker (which can be replicated for improving the system scalability and dependability) aggregates and analyzes monitored data from slaves, and uses them to build and solve an overall optimization problem (through its Configuration Manager component). As in the centralized approach, the cal-

culated configuration policy is then transmitted to slave brokers that implement it through their respective Execution Engine components. To enable an efficient exchange of large volumes of messages among the system components placed at geographically distributed locations, the overall system can rely on a publish/subscribe messaging system similarly to the distributed architectures for business process execution and service choreography proposed in [26] and [44], respectively.
However, this decentralized architecture still presents a single coordinator which needs to transmit and receive messages from the other system components; therefore, its location in the distributed environment may affect the performance of the overall system. Furthermore, the centralized coordination might be difficult to enforce administratively in a cross-organizational setting. To address these issues, a more scalable and decentralized solution would consist in devising a distributed solution of the overall optimization problem, that would mainly require a change in the CBP self-configuration policy presented in Sect. 4. This latter solution is in the direction of investigating decentralized self-adaptive systems, that present a number of challenges to be addressed in future research [43].

6 Conclusions

A SOA-based implementation of a CBP should be able to guarantee in an effective way the dependability levels that have been negotiated between the CBP providers and users. Based on the premise that to achieve this goal we must introduce in the system automated self-configuration features, we have discussed some issues to be considered. We have based our discussion on a quite general model of CBP, and have focused our attention mainly on methodological aspects, touching also some architectural issues. We have evidenced that existing approaches only address a limited part of the general CBP model, thus suggesting open problems to be addressed.

Acknowledgments

Work partially supported by the Italian PRIN project D-ASAP and by the project Q-ImPrESS (215013) funded under the European Union's Seventh Framework Programme (FP7).

References

1. van der Aalst, W.M.P., ter Hofstede, A.H.M., Weske, M.: Business process management: a survey. In: BPM '03. LNCS, vol. 2678, pp. 1–12. Springer-Verlag (2003)
2. Activiti: Activiti BPM Platform (2011), `http://www.activiti.org/`
3. Anisetti, M., Ardagna, C.A., Damiani, E.: Container-level security certification of services, chap. 6. This book, Springer-Verlag (2011)

4. Apache Software Foundation: Apache ODE (2011), http://ode.apache.org/
5. Ardagna, D., Pernici, B.: Adaptive service composition in flexible processes. IEEE Trans. Softw. Eng. 33(6), 369–384 (2007)
6. Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.E.: Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. Dependable Sec. Comput. 1(1), 11–33 (2004)
7. Badr, Y., Peng, Y., Biennier, F.: Digital ecosystems for business e-services in knowledge-intensive firms, chap. 2. This book, Springer-Verlag (2011)
8. Baresi, L., Di Nitto, E., Ghezzi, C.: Toward open-world software: Issue and challenges. IEEE Computer 39(10), 36–43 (2006)
9. Baresi, L., Guinea, S.: Self-supervising bpel processes. IEEE Trans. Software Eng. 37(2), 247–263 (2011)
10. Berbner, R., Spahn, M., Repp, N., Heckmann, O., Steinmetz, R.: Heuristics for QoS-aware Web service composition. In: IEEE ICWS '06. pp. 72–82 (2006)
11. Bonita: Bonita Execution Engine (2011), http://www.bonitasoft.org/
12. Canfora, G., Di Penta, M., Esposito, R., Villani, M.: A framework for QoS-aware binding and re-binding of composite web services. J. Syst. Softw. 81(10), 1754–1769 (2008)
13. Cardellini, V., Iannucci, S.: Designing a broker for QoS-driven runtime adaptation of SOA applications. In: IEEE ICWS '10. pp. 504–511 (2010)
14. Cardellini, V., Casalicchio, E., Grassi, V., Iannucci, S., Lo Presti, F., Mirandola, R.: MOSES: a framework for QoS driven runtime adaptation of service-oriented systems. IEEE Trans. Softw. Eng. (2012), to appear
15. Cardellini, V., Casalicchio, E., Grassi, V., Lo Presti, F., Mirandola, R.: QoS-driven runtime adaptation of service oriented architectures. In: ACM ESEC/SIGSOFT FSE '09. pp. 131–140 (2009)
16. Chan, P., Liu, M., Malek, M.: Reliable web services: methodology, experiment and modeling. In: IEEE ICWS '07. pp. 679–686 (2007)
17. Cheng, B.H.C., Giese, H., Inverardi, P., Magee, J., de Lemos, R.: 08031 – software engineering for self-adaptive systems: A research road map. In: Software Engineering for Self-Adaptive Systems. Dagstuhl Seminar Proceedings, vol. 08031. IBFI, Schloss Dagstuhl, Germany (2008)
18. Dan, A., Davis, D., Kearney, R., Keller, A., King, R., Kuebler, D., Ludwig, H., Polan, M., Spreitzer, M., Youssef, A.: Web services on demand: WSLA-driven automated management. IBM Systems J. 43(1) (2004)
19. Decker, G., Kopp, O., Leymann, F., Weske, M.: Interacting services: From specification to execution. Data Knowl. Eng. 68(10), 946–972 (2009)
20. Dubois, E., Kubicki, S., Ramel, S., Rifaut, A.: Capturing and aligning assurance requirements for business services systems, chap. 5. This book, Springer-Verlag (2011)

21. Friesen, A., Theilmann, W., Heller, M., Lemcke, J., Momm, C.: On some challenges in business systems management and engineering for the networked enterprise of the future, chap. 1. This book, Springer-Verlag (2011)
22. Guo, H., Huai, J., Li, H., Deng, T., Li, Y., Du, Z.: Angel: Optimal configuration for high available service composition. In: IEEE ICWS '07. pp. 280–287 (2007)
23. Hwang, S.Y., Liao, W.P., Lee, C.H.: Web services selection in support of reliable web service choreography. In: IEEE ICWS '10. pp. 115–122 (2010)
24. Hwang, S.Y., Lim, E.P., Lee, C.H., Chen, C.H.: Dynamic web service selection for reliable web service composition. IEEE Trans. Serv. Comput. 1(2), 104–116 (Apr 2008)
25. Ko, R., Lee, S., Lee, E.W.: Business Process Management (BPM) standards: a survey. Business Process Management J. 15(5), 744–791 (2009)
26. Li, G., Muthusamy, V., Jacobsen, H.A.: A distributed service-oriented architecture for business process execution. ACM Trans. Web 4(1), 1–33 (2010)
27. Mili, H., Tremblay, G., Jaoude, G.B., Lefebvre, E., Elabed, L., Boussaidi, G.E.: Business process modeling languages: Sorting through the alphabet soup. ACM Comput. Surv. 43(1), 1–56 (2010)
28. Niemöller, J., Freiter, E., Vandikas, K., Quinet, R., Levenshteyn, R., Fikouras, I.: Composition in heterogeneous service networks: requirements and solutions, chap. 9. This book, Springer-Verlag (2011)
29. Nitto, E.D., Ghezzi, C., Metzger, A., Papazoglou, M.P., Pohl, K.: A journey to highly dynamic, self-adaptive service-based applications. Autom. Softw. Eng. 15(3-4), 313–341 (2008)
30. OASIS: Web Services Coordination (WS-Coordination) Version 1.2 (Feb 2009)
31. OMG: Business Process Model and Notation (BPMN) Version 2.0 (Jan 2011), `http://www.omg.org/spec/BPMN/2.0/`
32. Oracle: BPEL Service Engine (2011), `http://www.oracle.com/us/technologies/soa/soa-suite/`
33. Papazoglou, M.: Web services and business transaction. World Wide Web: Internet and Web Information Systems 6, 49–91 (2003)
34. Pnueli, A.: The temporal logic of programs. In: Proc. of 18th Annual Symposium on Foundations of Computer Science. pp. 46–57. IEEE Computer Society (1977)
35. Qu, Y., Lin, C., Wang, Y., Shan, Z.: Qos-aware composite service selection in grids. In: GCC '06. pp. 458–465. IEEE Computer Society (2006)
36. Rodriguez, I.B., Halima, R.B., Drira, K., Chassot, C., Jmaiel, M.: A graph grammar-based dynamic reconfiguration for virtualized web service-based composite architectures, chap. 11. This book, Springer-Verlag (2011)
37. Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. ACM Trans. Auton. Adapt. Syst. 4(2), 1–42 (2009)
38. Skene, J., Raimondi, F., Emmerich, W.: Service-level agreements for electronic services. IEEE Trans. Softw. Eng. 36(2), 288–304 (2010)

39. Sun, C., el Khoury, E., Aiello, M.: Transaction management in service-oriented systems: Requirements and a proposal. IEEE Trans. Services Computing 4(2), 167–180 (2011)
40. Toktar, E., Pujolle, G., Jamhour, E., Penna, M., Fonseca, M.: An XML model for SLA definition with key indicators. In: IP Operations and Management. LNCS, vol. 4786. Springer (2007)
41. W3C: Web Services Choreography Description Language Version 1.0 (Nov 2005), http://www.w3.org/TR/ws-cdl-10/
42. Wetzstein, B., Karastoyanova, D., Kopp, O., Leymann, F., Zwink, D.: Cross-organizational process monitoring based on service choreographies. In: SAC '10. pp. 2485–2490. ACM (2010)
43. Weyns, D., Malek, S., Andersson, J.: On decentralized self-adaptation: lessons from the trenches and challenges for the future. In: SEAMS '10. pp. 84–93. ACM (2010)
44. Yoon, Y., Ye, C., Jacobsen, H.A.: A distributed framework for reliable and efficient service choreographies. In: WWW '11. pp. 785–794. ACM (2011)
45. Yu, J., Buyya, R.: Taxonomy of workflow management systems for grid computing. Journal of Grid Computing 3(3-4) (2005)
46. Yu, T., Zhang, Y., Lin, K.J.: Efficient algorithms for Web services selection with end-to-end QoS constraints. ACM Trans. Web 1(1), 1–26 (2007)
47. Zeng, L., Benatallah, B., Dumas, M., Kalagnamam, J., Chang, H.: QoS-aware middleware for web services composition. IEEE Trans. Soft. Eng. 30(5) (May 2004)
48. Zhao, W.: Design and implementation of a byzantine fault tolerance framework for web services. J. Syst. Softw. 82(6), 1004–1015 (2009)
49. Zheng, Z., Lyu, M.R.: A distributed replication strategy evaluation and selection framework for fault tolerant web services. In: IEEE ICWS '08. pp. 145–152 (2008)
50. Zheng, Z., Lyu, M.R.: A QoS-aware fault tolerant middleware for dependable service composition. In: IEEE/IFIP DSN '09. pp. 239–248 (2009)