Flexible Provisioning of Web Service Workflows

SEBASTIAN STEIN, TERRY R. PAYNE, and NICHOLAS R. JENNINGS
University of Southampton

Web services promise to revolutionize the way computational resources and business processes are offered and invoked in open, distributed systems, such as the Internet. These services are described using machine-readable metadata, which enables consumer applications to automatically discover and provision suitable services for their workflows at run-time. However, current approaches have typically assumed service descriptions are accurate and deterministic, and so have neglected to account for the fact that services in these open systems are inherently unreliable and uncertain. Specifically, network failures, software bugs and competition for services may regularly lead to execution delays or even service failures. To address this problem, the process of provisioning services needs to be performed in a more flexible manner than has so far been considered, in order to proactively deal with failures and to recover workflows that have partially failed. To this end, we devise and present a heuristic strategy that varies the provisioning of services according to their predicted performance. Using simulation, we then benchmark our algorithm and show that it leads to a 700% improvement in average utility, while successfully completing up to eight times as many workflows as approaches that do not consider service failures.

Categories and Subject Descriptors: I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence—*Intelligent agents; multiagent systems*

General Terms: Algorithms, Experimentation, Reliability

Additional Key Words and Phrases: Web services, semantic Web services, service-oriented computing, workflows, service provisioning, service composition

ACM Reference Format:

Stein, S., Payne, T. R., and Jennings, N. R. 2009. Flexible provisioning of Web service workflows. ACM Trans. Intern. Tech., 9, 1, Article 2 (February 2009), 45 pages. DOI = 10.1145/1462159.1462161 http://doi.acm.org/10.1145/1462159.1462161

This article is a significantly extended version of a previous conference paper that appeared in *Proceedings of the 17th European Conference on Artificial Intelligence (ECAI-06)*, 295–299.
This work was funded by the Engineering and Physical Sciences Research Council (EPSRC) and a BSE Systems studentship.
Authors' address: School of Electronics and Computer Science, University of Southampton, Southampton, SO17 1BJ, United Kingdom; e-mail: {ss2, trp, nrj}@ecs.soton.ac.uk.

1. INTRODUCTION

Due to the proliferation and ubiquity of the Internet, modern computer systems are becoming increasingly distributed in nature. This growing connectivity allows organisations to share expensive computing resources, to automate and outsource business processes, and to offer their services to a worldwide audience [Foster et al. 2001; Medjahed et al. 2003]. In this context, service-oriented computing is gaining popularity as an appropriate system engineering approach for enabling distributed, heterogeneous software components to communicate and interoperate through computer *services* [Singh and Huhns 2005]. These services are software components that are offered over computer networks and that are described using machine-readable descriptions. Whilst existing approaches, such as Web service technologies [Curbera et al. 2002] and CORBA [Yang and Duddy 1996], have concentrated on providing an infrastructure and syntactic service descriptions, recent work on Semantic Web services has explored the use of ontologies to augment service descriptions [McIlraith et al. 2001; Martin et al. 2004]. The overall aim of this effort is to facilitate knowledge-based reasoning about their functionality, and to support interoperability between services within an open environment. Specifically, it is envisaged that annotating service descriptions will allow applications to automatically discover new and previously unseen services, and invoke them as part of their workflows. In so doing, this approach will greatly reduce human effort and address the inherent openness and heterogeneity of the Internet, where service availability changes over time and where services do not generally conform to standardized interfaces and invocation methods [Paolucci and Sycara 2003].

To date, however, most research in this area has viewed services as fully cooperative software components that behave according to their service descriptions. Thus, work has concentrated largely on expressing and reasoning about the functional capabilities of services and has mostly ignored the inherent *unreliability* and *uncertainty* of using remote service providers. However, such uncertainty should be considered, due to the open and dynamic nature of the Internet, where network failures, remote software bugs, transmission delays and competition over limited resources are an unavoidable feature of the environment [Bolot 1993; Long et al. 1995; Schroeder and Gibson 2006].

The resulting uncertainty is further exacerbated by the inherent autonomy of service providers. A key feature of computer services is the fact that they are implemented, maintained and executed on remote machines by independent providers. While this allows for loose coupling and a high level of abstraction, it also means that the service implementation is completely opaque to the consumer. Thus, the provider may use any scheduling algorithm and allocate resources, queue or even reject service requests according to local execution policies and depending on current levels of demand. For this reason, service providers can be viewed as *autonomous, self-interested agents* that follow their own decision-making procedures [Weiß 1999; Jennings 2001]. Therefore, it cannot be assumed that services always behave as advertised. In fact, providers may lie or overstate their capabilities to attract customers, especially in environments where services demand remuneration (e.g., as is emerging in

the context of Grid computing [Buyya et al. 2005]) and where untruthful or misleading service advertisements may result in a financial advantage for the providers.

Such uncertainty in provider behavior cannot be ignored by research on Web services, as it poses critical problems to consumers that rely on services to complete their workflows. Here, a single service failure could jeopardise the overall success of the workflow, and even unexpected service delays may result in an unacceptable completion time. When reliable workflow execution is important for the objectives of their owners (as is the case in most emerging application domains, such as automated business process management [Jennings et al. 2000] or large-scale scientific workflows [Deelman et al. 2003]), workflow failure or delays are highly undesirable and can result in considerable losses to the consumer (e.g., in terms of lost business revenue, time, or penalties incurred by missing contractual deadlines).

To address this problem of service uncertainty, we focus on the *provisioning* of Web services. In short, this is the process of assigning particular service instances to the constituent tasks of abstract workflows after candidate services have been identified by a semantic matchmaker [Zeng et al. 2003; Maximilien and Singh 2004a]. Whilst the matchmaking stage concentrates on matching functional service adverts to abstract task templates, provisioning uses advertised or observed quality-of-service measures to allocate instances in an appropriate manner. Thus, it is possible during provisioning to make predictions about the overall performance of the workflow, to identify particularly failure-prone tasks and to mitigate the effects of such tasks by provisioning services appropriately.

More specifically, in this article, we present a heuristic provisioning algorithm that can be used by service consuming agents to deal with service failures both proactively (by provisioning particularly failure-prone services redundantly) and reactively (by re-provisioning unsuccessful services on-the-fly even when they do not report their failure). In so doing, we employ decision theoretic techniques to explicitly balance the cost of invoking services with the potential reward of successfully completing a workflow. In experiments, we show that this flexible algorithm achieves better results than current approaches that rely only on simple semantic matchmaking. Specifically, our approach achieves an increase in average profit of about 700%, and it successfully completes 98–99% of workflows in most environments, even when individual services are highly unreliable.

The remainder of this article is structured as follows. In Section 2, we discuss the state-of-the-art and position our work in the context of current Web service research. In Section 3, we present an abstract model of a service-oriented system and the types of workflows we investigate. Based on this, we present our flexible provisioning strategy (Section 4), and evaluate it experimentally (Section 5). Finally, we conclude in Section 6 by outlining future work.

2. RELATED WORK

In this section, we examine current approaches for executing workflows consisting of several interdependent tasks. First, we look at appropriate technologies

for describing workflows and how these use Semantic Web technologies for discovering services. Then, we discuss how provisioning has been addressed so far, and why current approaches are insufficient for dealing with unreliable services.

Applications in service-oriented environments rarely rely on single, isolated services, but rather combine the functionality of many distinct service offerings to fulfil their design goals [Milanovic and Malek 2004]. Such service combinations are usually expressed as workflows, which have been applied widely in industry to define the required tasks and appropriate precedence constraints to achieve some overall business goal [Georgakopoulos et al. 1995]. Currently, the predominant approach for defining such workflows of interdependent Web services is WS-BPEL (Web Services Business Process Execution Language) [Weerawarana et al. 2005]. This language is highly expressive and offers some flexibility by allowing workflows to refer to abstract service interfaces rather than concrete instances. This means that services can, in principle, be selected dynamically at run-time, depending on current service availability. However, WS-BPEL is of limited use in large, open environments where services are offered and implemented by distinct, heterogeneous agents. This is because it requires service instances to adhere exactly to the syntactic interfaces specified by the workflow designers—an unrealistic assumption in such systems [Akkiraju et al. 2004].

Semantic Web technologies promise to address this by offering more formal service discovery and composition techniques, thus allowing applications to bind previously unseen and heterogeneous services at run-time without human intervention. Building on knowledge representation formalisms such as OWL [McGuinness and van Harmelen 2004], Semantic Web services present rich metadata annotations that state the semantics of the service descriptions rather than specifying just their syntactic usage [McIlraith et al. 2001]. As exemplified by the emerging OWL-S ontology, such annotations provide information about the functionality of services, the data they operate on, and the context in which they can be invoked [Martin et al. 2004]. Because this metadata is presented using a machine-readable representation language (whose underlying semantics correspond to a well defined and decidable logic theory), it is possible for client applications to use it (in combination with their own knowledge and other assertions available on the Semantic Web) to reason automatically about services (e.g., to discover services that meet certain user requirements [Paolucci et al. 2002; Benatallah et al. 2005] or to translate between heterogeneous services that use different data representations [Szomszor et al. 2005]).

This semantic information has been used to automate the synthesis of service workflows. Most work in the area of service composition employs planning algorithms to search for sequences of service instances that meet a given high-level goal [McDermott 2002; Klusch et al. 2005]. Such approaches promise to require little human intervention and rely solely on the semantic descriptions of services, as well as simple, high-level goals given by the users. However, planning is a computationally complex problem [Bylander 1994], and so it is likely to be infeasible in environments where potentially thousands of services coexist.

To address this complexity, some work on service composition uses abstract workflows, which describe the necessary steps to fulfil common user objectives [McIlraith and Son 2002; Mandell and McIlraith 2003; Sirin et al. 2005]. This work assumes that workflows for particular objectives usually follow the same basic steps, even if the choice of service instances is different each time, depending on the user's personal constraints and current service availability. More specifically, such abstract workflows usually include a number of semantically annotated abstract tasks (e.g., using generic OWL-S *process:SimpleProcess* descriptions in the process model) and a suitable ordering. At run-time, the abstract task descriptions are used to discover service instances, which can then be provisioned for the tasks of the workflow. If necessary, additional planning is used to combine or substitute abstract workflow fragments [Sirin et al. 2005] or to add intermediate services (e.g., to translate between heterogeneous data representations) [Mandell and McIlraith 2003].

Now, a major shortcoming of the approaches discussed so far is that they do not offer satisfactory facilities for dealing with uncertain service performance. In fact, most work relies purely on functional service descriptions, assuming these to be deterministic and truthful. Hence, the resulting workflows are brittle and vulnerable to single service failures.

Traditionally, failures in workflows have been addressed by exception handling mechanisms that follow predefined procedures for mitigating or correcting a problem before continuing the workflow (forward recovery), or that roll-back previous tasks of the workflow to terminate it in a consistent state (backward recovery) [Garcia-Molina and Salem 1987; Eder and Liebhart 1995; Casati et al. 1999]. This approach is also taken by WS-BPEL, which allows workflow designers to specify fault and compensation handlers that are invoked when failures occur during workflow execution [Curbera et al. 2003]. However, relying only on exception handling mechanisms is problematic in the environments we consider. First, they are entirely reactive and so do not allow the consumer to avoid failures proactively (this is especially important when the consumer has a fixed deadline for its workflow). Second, the mechanisms are usually specified manually, which is labor intensive and can be unrealistic for large workflows. Finally, they typically rely on cooperative service providers that signal failures clearly and may allow services to be rolled-back.

To address service failures more proactively, possibly in the presence of malicious providers, a considerable body of research is investigating quality-of-service (QoS) issues for Web services [Menasce 2002; Ran 2003]. This is concerned with non-functional properties of services, including reliability, service costs and durations, and there are a number of approaches for expressing this information. For example, WSLA (Web Service Level Agreements) is an industry-led specification for describing the expected non-functional characteristics of a service interaction in a contractual form between the provider and consumer [Dan et al. 2004]. WSLA allows the specification of penalties for defaulting on a contract and thereby provides some protection to the consumer. However, in case of failures, the service consumer still faces an incomplete workflow (whose value may surpass the received penalty) and it is possible that malicious providers do not compensate the consumer at all.

Other work has used ontologies to describe and reason about QoS information, which is typically assumed to have been obtained through previous interactions or trusted third parties [Maximilien and Singh 2004a, 2004b; Zhou et al. 2004]. These ontologies have generally been used to place restrictions on the types of services to be provisioned, and so complement the functional service descriptions. However, such approaches are inflexible as they make binary choices about which services are feasible, and they do not balance conflicting qualities appropriately (e.g., sometimes a cheap, unreliable service may be preferred to an expensive, reliable one). Hence, these approaches rely on purely qualitative reasoning mechanisms, which are unsuitable for the largely numeric, quantitative information about service qualities. Furthermore, they require a human to annotate workflows accordingly and make feasible restrictions (i.e., that can be satisfied by the available service instances).

Other approaches in the literature take a more flexible approach than hard, binary decisions when provisioning services based on QoS metrics. Sirin et al. [2005] use a simple ranking mechanism that assumes preferential independence between different QoS dimensions and then chooses a service that is pareto optimal regarding all dimensions. However, this approach does not balance conflicting qualities or consider their magnitude. To address this, other work combines QoS constraints with numerical optimization [Zeng et al. 2003; Aggarwal et al. 2004; Jaeger and Mühl 2007; Yu et al. 2007]. Here, the workflow typically has some hard constraints for a number of aggregated QoS parameters (such as the overall duration, reliability or cost of the entire workflow). If these can be satisfied by different service choices for each task, services are selected so that a weighted sum of all parameters is maximised. This approach aims to strike a balance between conflicting parameters, but is inappropriate for several reasons. First, reliability is simply another quality, substitutable at a constant rate with any other quality, which can result in irrational preferences (e.g., if other qualities are raised appropriately, this mechanism will choose a service with a 0% reliability). Second, to avoid this problem, appropriate constraints have to be imposed, which require human effort and unnecessarily restrict the candidate solutions. Third, setting the weights for different parameters is a nontrivial task that requires further human input. Finally, this approach provisions only single service instances for each task and so is likely to fail when all available services are unreliable.

A more promising approach towards addressing unreliable services has been taken in the context of agent-based computing. For example, Collins et al. [2001] use decision-theory to provision services for abstract workflows. In this work, the authors provision services in order to maximise the expected utility of the consumer, which balances the utility of a successful workflow with the incurred cost and the overall success probability. However, they consider an auction scenario that is not directly applicable to current Web service standards, and their approach again relies on single services for each task.

So far, the approaches mentioned here are unsuitable for scenarios where services are highly failure-prone, as a single service failure will always result in an overall workflow failure. To address this, some work has looked into the use of *redundancy* to mitigate the problem of unreliable providers. This is a technique

from reliability theory that has been widely employed to increase the robustness of a system by duplicating critical and failure-prone components [Tillman and Liittschwager 1967]. In the context of fault-tolerant computing, redundancy has been used to design software applications that provide certain guarantees about their behavior despite component failures or malicious attacks [Cristian 1991; Gärtner 1999]. In particular, traditional Web servers often employ redundancy to seamlessly mask failed components (service failover) and to distribute requests to several replicated services (load balancing) [Ingham et al. 1999; Aghdaie and Tamir 2003]. Similarly, the use of redundancy has been suggested to build fault-tolerant Web services [Li et al. 2005; Merideth et al. 2005], but such work has concentrated on designing appropriate software architectures and protocols and is not directly applicable to the problem of provisioning service workflows.

Redundancy has also been employed for the provisioning of services in large-scale distributed systems. Anderson et al. [2002] describe an application in peer-to-peer systems that provisions several functionally-identical service instances for the same task, in order to reduce the probability of failure and to detect malicious data providers. However, the level of redundancy is generally fixed (i.e., there is no notion that some tasks might be more failure-prone and so require higher redundancy than others) and it is assumed that services are inexpensive and numerous. Jaeger and Ladner [2005] show how provisioning redundant providers in parallel can improve the overall success probability and duration of a workflow, but they do not consider stochastic service times or discuss how appropriate levels of redundancy can be chosen autonomously.

A different form of redundancy is used in the work of Friese et al. [2005] on executing WS-BPEL workflows in peer-to-peer systems. They develop a mechanism for dynamically discovering and invoking duplicate services after the first service has failed. A similar approach is taken by Erradi et al. [2006], who propose a policy-based framework for dealing with service failures. In their work, failure handling policies are triggered by predefined events (e.g., the violation of service level agreements or the receipt of an error message) and specify corrective actions that should be taken. These actions include retrying a failed service multiple times, as well as invoking one or more functionally-identical replacement services instead. However, they rely on a human user to specify appropriate failure policies and levels of redundancy. Finally, both Canfora et al. [2005] and Yu and Lin [2005] adapt the numerical QoS optimization techniques discussed earlier to find appropriate replacement services in case of failure, but these approaches still require significant human input, as mentioned above, and will constantly need to replan when services are highly unreliable.

To conclude this section, we have seen that there exist powerful techniques for modelling abstract workflows, which are provisioned automatically at runtime. Such approaches require little human effort to adapt and maintain workflows in dynamic environments. However, current approaches do not deal satisfactorily with unreliable providers. When reliability and other non-functional service characteristics have been considered, they have usually been addressed by imposing simple constraints on the required services or by optimizing a

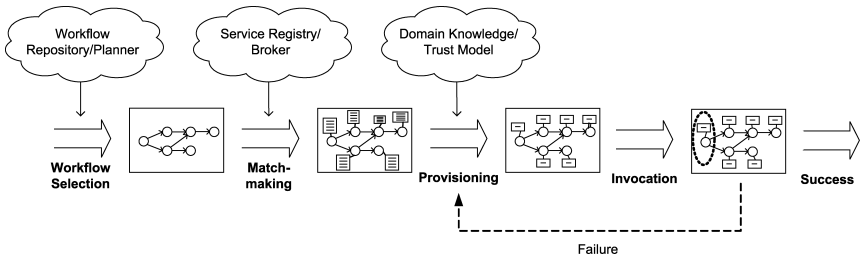


Fig. 1. Lifecycle of a workflow.

weighted sum of parameters. Such an approach relies on appropriate parameters set by a human user and is still vulnerable when services are generally failure-prone. While some research has looked specifically at the use of redundancy to mitigate failures, no work automatically chooses the type and level of redundancy in order to balance the cost of services with their reliability and the benefit of completing a workflow (and the associated time). To address these shortcomings, in the following section, we first formalise our model of a service-oriented system, and we propose our solution for dealing with such uncertain service providers in Section 4.

3. WORKFLOW AND SERVICE MODEL

In this section, in order to provide a formal basis for our work, we define the types of workflows that a consumer typically faces, and the services that it can provision in order to execute these workflows. Specifically, in Section 3.1, we outline the context of our work and describe the lifecycle of an abstract workflow. Then, we discuss the information that is available to the service consumer when provisioning services: in Section 3.2, we define the workflows we consider, and in Section 3.3, we formalise the information that is available about service instances. In Section 3.4, we describe how services are provisioned and invoked for the tasks of a workflow. Finally, in Section 3.5, we highlight and justify some of the simplifying assumptions we have made.

3.1 Workflow Lifecycle

Building on the work on abstract workflows outlined in Section 2, a service consumer in our model proceeds through four stages when executing a workflow (Figure 1):

(1) *Workflow Selection.* First, an abstract workflow is chosen to suit the consumer's current objectives. This is generally created either manually by domain experts or automatically by a planner that uses abstract templates of common service types. Due to the complexity of generating workflows, this may take place offline, allowing the consumer to retrieve suitable workflows from a repository. At this stage, an abstract workflow does not refer to service instances, but rather contains abstract tasks, which are annotated by semantic metadata to describe suitable services.

(2) *Matchmaking*. Once an abstract workflow has been selected, abstract tasks are mapped to candidate service instances via a matchmaking process. Here, the consumer searches a public service registry or requests matching services from a broker. This step uses the semantic annotations provided by the abstract workflow to find suitable service instances.

(3) *Provisioning*. Given lists of matching services, the consumer now provisions individual service instances for each task of the workflow. This decision constitutes a tacit intention by the consumer to invoke the provisioned services for the respective tasks, and so it is not necessarily a binding commitment. The purpose of this stage is to allow the consumer to make predictions about the performance of a provisioned workflow, and to explore the space of candidate provisioned workflows. Specifically, it is possible for the consumer to evaluate and optimize the provisioned workflow using an appropriate utility function that encodes the value of successfully completing the workflow. During this stage, the consumer can make use of its own domain knowledge and possibly service performance information that is available from external sources, to identify particularly failure-prone tasks, and to proactively provision additional services where necessary and where this increases the expected utility of the provisioned workflow.

(4) *Invocation*. When appropriate services have been provisioned, the consumer starts to invoke the chosen services as dictated by the ordering constraints of the workflow. If services fail to complete their tasks, the consumer may provision other services, until the workflow is successfully completed.

As outlined in Section 2, current work on Semantic Web services either views the *provisioning* stage as an intrinsic part of matchmaking, or proposes solutions that are infeasible in environments where services may regularly fail to honour their descriptions. In order to develop a more effective provisioning strategy, we proceed to describe the information that we assume to be available to the service consumer. To this end, we define the structure of an abstract workflow in Section 3.2, and, in Section 3.3, we describe the services that are available for the constituent tasks of the workflow.

3.2 Workflow Description

A workflow is typically a collection of tasks with appropriate ordering constraints. For this reason, we represent it using a directed, acyclic graph (as shown by Figure 2). Formally, we express workflow W as:

$$W = (T, E, \tau, u). \tag{1}$$

Here, $T = \{t_1, t_2, t_3, \ldots, t_{|T|}\}$ is the set of tasks that make up the workflow, and $E \subseteq T \times T$ is a strict partial order over the tasks, where a member, $(t_i \mapsto t_j) \in E$, indicates that task t_i must successfully complete before t_j can be started. The function $\tau : T \to \mathcal{T}$ maps each task to an abstract service description, where $\mathcal{T}$ is the set of all descriptions. Additionally, to represent its value to the consumer, we attach a utility function to each workflow, $u : \mathbb{R} \to \mathbb{R}$, which maps

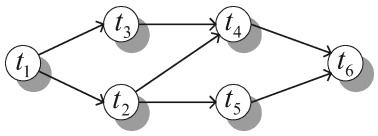


Fig. 2. Example workflow consisting of six interdependent tasks. Circles represent the tasks in T and arrows represent the dependencies as given by E (transitive dependencies are omitted for readability).

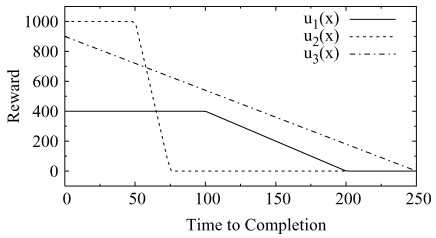


Fig. 3. Examples of some representative utility functions.

the completion time of the workflow to a *utility reward*.[1] We assume that this utility is only awarded to the consumer when the whole workflow is completed successfully and that the utility function is monotonically decreasing.[2] Specifically, we use a general utility function that awards a maximum utility $u_{\max}$ when the workflow is completed within a given deadline $t_{\max}$. When this deadline is exceeded, a penalty rate δ is deducted from $u_{\max}$ for every time step that the consumer is late, until it gains no more positive utility. In this case, the consumer receives a reward of zero, regardless of whether the workflow is completed at a later stage or not. Formally, we express the reward function u as follows (with $u_{\max} \geq 0$, $t_{\max} \geq 0$ and $\delta > 0$):

$$u(t) = \begin{cases} u_{\max} & \text{if } t \leq t_{\max} \\ u_{\max} - \delta(t - t_{\max}) & \text{if } t > t_{\max} \quad \text{and} \quad t < t_{\max} + u_{\max}/\delta. \\ 0 & \text{if } t \geq t_{\max} + u_{\max}/\delta \end{cases} \tag{2}$$

To illustrate this, Figure 3 contains some example utility functions. The function labelled $u_1(x)$ is a typical example with $u_{\max} = 400$, $t_{\max} = 100$ and $\delta = 4$. The function $u_2(x)$ represents an example where time is more critical, and $u_3(x)$ has no specific deadline ($t_{\max} = 0$), rewarding the agent purely based on the amount of time taken.

We now describe the service instances that are available for the tasks of the workflow.

[1]This may be the expected financial gain of completing the workflow, or simply a private utility value, as commonly used in decision theory [Raiffa 1968].

[2]This is consistent with much previous work—Collins et al. [2001] reward a consumer with a fixed utility reward for completed workflows, while Arunachalam and Sadeh [2004] and Irwin et al. [2004] describe utility functions that depend on the time of completion.

3.3 Service Discovery and Performance Information

During the matchmaking stage, the service consumer discovers suitable service instances for each task in T, using the semantic information provided by τ. If we denote the set of all services as $\mathcal{S} = \{s_1, s_2, \ldots, s_{|S|}\}$, then we can formalise this discovery procedure as a function, $\mu : \mathcal{T} \rightarrow \wp(\mathcal{S})$, that maps an abstract service description to a set of suitable services. For task t_i, we denote this set as $S_i = \mu(\tau(t_i))$.

In addition to the set of services for each task, we also assume that the consumer has some knowledge about their predicted performance. However, because the Internet is an open and dynamic environment, where services may leave or enter at will, change their identities and publish false information, this knowledge does not extend to individual service instances. Rather, we assume that it will normally take the form of probabilistic estimates and distributions over the set of all services for a particular task. Specifically, we assume the following information about each task t_i:

—S_i is the set of suitable services.
—f_i is the failure probability of a single service from S_i.
—c_i is the cost[3] of a service from S_i.
—d_i is a probability density function, representing the execution duration of a service from S_i (the total time from invocation to completion). It is conditional on overall success.

These probabilistic measures govern how services behave in our model. In the following section, we briefly outline how this relates to their interactions with the service consumer.

3.4 Service Provisioning and Invocation

As described in Section 3.1, during the provisioning stage, the service consumer allocates service instances to the tasks of the abstract workflow. Once this allocation is completed, the consumer begins to invoke services for the tasks of the workflow according to the ordering constraints given by E. Here, we assume that the consumer can only invoke services at discrete, integer time steps. As is common in the Web services domain, services are invoked *on demand* when they are required. Hence, the cost for each service is paid only at the time of invocation (but regardless of the eventual outcome). When invoked, a service successfully completes the assigned task t_i with probability $1 - f_i$. The duration of a successful service execution is distributed according to d_i, after which the service consumer is notified of success (i.e., the consumer cannot be certain of the outcome until this time has passed). When a service is not successful, we assume that it fails silently (i.e., no response is given to the consumer).

[3]This usually represents a financial remuneration for the service (expressed in the same units as the reward function $u(t)$), but could also quantify the effort and required bandwidth of invoking the service. In reality, these costs are likely to vary across S_i. However, as we use no other service-specific information to distinguish services and as costs may change dynamically between provisioning and invocation, we decided to use a fixed measure, representing the *expected* cost of such a service.

Furthermore, as there may be several matching services for a task, the consumer can invoke more than one service for this task at the same time. In this case, the consumer has to pay each invoked service separately, and the task is completed when the earliest service has executed successfully (if any). When all invoked services seem to have failed, the consumer may decide to provision new services for this task. In this case, the consumer will ignore the previously invoked services and assign the task to the newly provisioned set of services. For multiple invocations, we assume that services execute independently from each other (i.e., the success and duration of all services are independent, identically distributed random variables), and that the consumer is not penalised additionally if several services are successful (apart from paying the associated costs).

In the following section, we summarize and justify the underlying assumptions and associated limitations of the model presented above.

3.5 Model Assumption and Limitations

Although we have striven to present a model that is applicable to a large range of service-oriented scenarios, we have had to make a number of simplifying assumptions about our problem domain that may not hold in all potential application areas. On the one hand, these assumptions were necessary to produce a formal model that is amenable to efficient mathematical analysis, and on the other hand, they allowed us to present and deal with a general problem rather than concentrate on domain-specific constraints that may occur in a concrete application. We believe that our assumptions are reasonable in most large distributed systems and that our model constitutes a solid basis for more specific extensions. In this section, we explicitly list and justify the assumptions we have made. In Section 6.2, we will re-examine some of these and show how our model can be extended to handle them.

(1) *Failure Model.* We have chosen to restrict our failure model to include only silent failures at this time (also known as *crash* failures [Cristian 1991]). In practice, failure messages may sometimes be returned to the consumer, but silent failures are more challenging to deal with (clearly, a consumer receiving such messages will perform at least as well as one that does not). Furthermore, they are realistic in distributed environments, where service providers do not reveal their internal state, and where network or machine failures can lead to communication losses. However, we currently do not deal with Byzantine failures, which include the return of corrupt service results. Hence, we must assume that service results can be tested for correctness (in fact, many intractable problems can be efficiently verified), but we plan to relax this limitation in future work. We also assume that failures (and durations) of different services are independent of each other. We believe that this is generally the case in large-scale distributed systems, where services reside on physically separate machines, use different implementations and do not directly interfere with each other. Despite this, failures may occasionally be correlated—for example, when two services rely on a common third service, or when several systems are attacked by the same virus.

(2) *Performance Information.* As we concentrate on the provisioning problem rather than learning techniques, we assume that the service consumer has accurate performance information about the providers for each task. In practice, such information may be domain knowledge provided by experts during workflow generation [Ng and Abramson 1990], by inference over the task descriptions and related data [Maximilien and Singh 2004a], or by statistical estimation based on previous interactions with similar services, possibly provided by a trusted monitoring service [Teacy et al. 2006]. However, obtaining this knowledge is clearly nontrivial and has been the subject of much ongoing research.

Furthermore, we currently represent uncertain service durations using simple nonconditional probability density functions. This is a common approach for modeling stochastic systems, but it is possible to envisage more detailed joint distributions to be available, for example to model varying service durations at different times of the day.

(3) *Payment Model.* Our model assumes that a service must be paid for regardless of its eventual outcome. Similar to our silent failure model, this is a pessimistic assumption that will not always be true in realistic environments. Rather, cooperative services may refund the consumer on failure or decline payment before the service is commenced. Again, a consumer that is able to deal with this more risky case will perform at least as well when services offer refunds.

Additionally, our model assumes that the service consumer is charged a fixed price per invocation. We believe that this is realistic in scenarios where services are discovered and provisioned dynamically on-demand and where no long-term contracts exist between the provider and consumer. However, it should be noted that other pricing schemes have been proposed, including some that allow multiple invocations of the same service over a certain period of time [Dan et al. 2004].

We also currently assume free disposal of unwanted services, that is, that several successful service invocations for the same task do not incur additional penalties above their normal cost. This may be realistic in Grid scenarios, where the results of data processing services can be disregarded without costs, but in a supply-chain application, the disposal of unused goods may incur additional charges (especially for chemicals or dangerous materials).

(4) *Reward Model.* Our reward function encodes the value of completing a workflow at a given time, and it intuitively follows the general form of many contracts in other domains. However, certain application scenarios might require a more expressive function that depends on multiple dimensions (e.g., the overall time and the perceived quality of some end-product).

(5) *Model Scope.* To obtain a general system model, we currently do not consider specific domain-dependent constraints that may occur in particular workflow applications. For example, we do not cover cases where service instances have mutually exclusive side-effects or where there are dependencies between the instances provisioned for several tasks. We also represent

workflows as directed acyclic graphs, which is consistent with much related work, but we note that realistic applications often require more complex structures, including branches and loops.

Finally, in line with the overall aim of this article, we focus solely on provider failures. Hence, we assume that workflows are correct, that appropriate matchmaking algorithms correctly identify suitable providers and that the consumer is able to translate between heterogeneous data formats. In practice, such problems are far from trivial, but they are not the focus of this work.

This concludes the description of our workflow and service model. In the following section, we build on this model to develop a provisioning strategy that deals with services both proactively and reactively, in order to address the unreliability and uncertainty of service providers.

4. PROVISIONING STRATEGIES

In this section, we outline several strategies for provisioning services for the workflows described in Section 3. We begin in Section 4.1 by outlining a *naïve* strategy that formalises many current approaches towards service provisioning that do not consider service uncertainty. In Sections 4.2 and 4.3, we develop two simple strategies that rely on multiple services to satisfy single tasks (*parallel(n)* and *serial(w)*) and that are broadly based on simple redundant strategies found in related work. These are then combined in Section 4.4 as a *flexible* provisioning strategy that reasons quantitatively about its provisioning decisions, and that constitutes the main contribution of this article.

4.1 The Naïve Strategy

We begin by looking at the currently predominant approach to service provisioning in the literature. This gives us a basic benchmark against which we can evaluate the strategies we develop in this section, and, in doing so, serves to highlight the shortcomings of current work.

Now, as described in Section 2, most current work on Web services focusses solely on the functional descriptions of services. In such research, descriptions are typically assumed to be truthful and deterministic, and so service-consuming agents do not explicitly consider the provisioning stage, but rather pick *any* single service that matches their requirements. Since such a strategy does not consider service failures, we term it *naïve* and describe it more formally as follows:

Definition 4.1 (*Naïve Strategy*). A consumer agent following a ***naïve*** strategy always provisions a single randomly chosen service of the correct type for each task.

A major shortcoming of this *naïve* strategy is that it is highly vulnerable to service failures. A single failure means that the whole workflow is lost, along with all investments already made. To reduce this risk, we discuss two simple techniques in the following sections for dealing with service failures.

4.2 Parallel Provisioning

The first strategy we discuss in this context uses parallel provisioning to *proactively* control the effect of unreliable services. As discussed in Section 3, a feature of service-oriented systems is the fact that several service instances may match a single semantic service description. For this reason, a consumer may benefit by delegating each of its tasks to several providers at the same time, rather than relying on a single service.

To highlight the advantage of this approach, let $X_n \in \{\text{success}, \text{failure}\}$ be a random variable indicating the outcome for a task t_i when n services are invoked in parallel for this task. The probability that a single service ($n = 1$) successfully completes the task is then $P(X_1 = \text{success}) = 1 - f_i$. When invoking two service instances in parallel ($n = 2$), we have a success probability $P(X_2 = \text{success}) = 1 - {f_i}^2$. For the general case with n services, we thus have:

$$P(X_n = \text{success}) = 1 - {f_i}^n \tag{3}$$

This means that the probability of success increases as more providers are provisioned for a single task. However, if a nonzero cost is associated with each provision, then the total cost incurred rises with n. Based on this, we can formulate a strategy that uses parallel provisioning to reduce the probability of workflow failures:

Definition 4.2 (*Parallel(n) Strategy*). A consumer following a ***parallel(n)*** strategy always provisions exactly n randomly chosen services of the correct type for each task.

For this strategy, n is a fixed constant that is determined by a human user. The strategy *parallel(1)* is equivalent to the *naïve* strategy, and a higher value for n implies a generally higher resilience against failures. However, while reducing the probability of workflow failures, the *parallel(n)* strategy lacks any capacity to react to failures *after* they have occurred. This is addressed by the strategy in the following section.

4.3 Serial Provisioning

The second strategy we describe deals *reactively* with service failures. Rather than relying on parallel provisioning, it reprovisions services when it becomes likely that a previously provisioned service has failed. To this end, the consumer first provisions a single service and, after invocation, waits for some time. If the service has not been successful, the consumer tries a different one, waits and repeats the process if necessary, until the task has been completed. However, as services have non-deterministic duration times and because they do not notify the consumer of failure, the consumer has to choose an appropriate waiting period. This period should allow the service a reasonable time to finish, but should not waste unnecessary time when it has most likely already failed.

With this in mind, let $X_{s,w} \in \{\text{success}, \text{failure}\}$ be a random variable indicating the outcome of invoking single service instances in series for a task t_i. Here, s is the number of services that are available in total ($s = |S_i|$), and w is the chosen waiting period. To calculate the success probability of a single

service in this case, we can use the cumulative density function D_i, derived from d_i. Hence, we have $P(X_{1,w} = \text{success}) = (1 - f_i)D_i(w)$, where $1 - f_i$ is that probability that the service will succeed, and $D_i(w)$ is the probability that this will happen within w time steps. Generalizing this for invoking s services in sequence, we get:

$$P(X_{s,w} = \text{success}) = 1 - (1 - (1 - f_i)D_i(w))^s. \quad (4)$$

This is generally less than the success probability of invoking the same number of services in parallel, and the average time taken will also be higher for serial provisioning because of the additional waiting time that is introduced. On the other hand, the average cost drops, because costs are only incurred at the time of invocation.

Hence, we define a new reactive strategy as follows:

Definition 4.3 (*Serial(w) Strategy*). A consumer following a *serial(w)* strategy always provisions exactly one randomly chosen service of the correct type for each task. After a waiting period of w time units, if no success has been registered yet and if there are still more available services, the agent re-provisions a new, randomly chosen service and continues in this manner until the task is completed or no more services are left.

The two approaches discussed above, *serial(w)* and *parallel(n)*, deal with service failures. However, they have several shortcomings that make them less useful for automating the provisioning of complex workflows. First, we have so far considered them separately, whereas they might complement each other by allowing the consumer to provision services both in series and in parallel, as required. Second, we have assumed that the constants n and w are provided by a human user, but choosing these is not trivial. Especially in dynamic environments, they should be chosen automatically depending on current information about the predicted performance of tasks. Finally, we have so far treated n and w as global constants. However, most realistic workflows will have tasks that vary considerably in their reliability, cost and duration distribution. Hence, it may be necessary to use different provisioning strategies for each task in the workflow.

To address these shortcomings, in the following section, we develop a novel strategy that provisions multiple services for tasks in a flexible manner. This approach takes into consideration the performance characteristics of services and the structure of the workflow, and then provisions services accordingly, using a heuristic algorithm to deal with the inherent complexity of this task.

4.4 Flexible Provisioning

Building on the strategies presented in the previous section, we now introduce a novel algorithm for flexibly provisioning services that are part of complex workflows. Because we are interested in building an agent that provisions services automatically, we take a decision-theoretic approach, where the agent provisions services so as to maximise its expected utility. Specifically, it determines automatically how many services to invoke in parallel and it also chooses an appropriate time-out value.

Due to its autonomous decision-making process that adjusts the agent's behavior to its environment, we term this approach the *flexible* strategy and summarize it as follows:

Definition 4.4 (*Flexible Strategy*). A consumer following a *flexible* strategy makes appropriate decisions to provision services for its workflow. To this end, the agent finds suitable numbers of service instances to be invoked and time-out values for each task in the workflow, so that the agent's expected utility is maximised.

This discussion of the *flexible* strategy is divided into several parts. First, we describe our aim in devising this strategy as an optimization task (Section 4.4.1). In the second part, we outline a heuristic local search approach for solving this problem (Section 4.4.2). To conclude the discussion, we provide an illustrative example of how our strategy provisions a complete workflow (Section 4.4.3).

4.4.1 *Problem Formulation.* In this section, we first formulate a more fine-grained decision problem than so far considered. Instead of choosing global values for n and w, as in the previous approaches, we define them as vectors, $\vec{n}$ and $\vec{w}$, with each element corresponding to one task in the workflow. In this notation, the ith element of vector $\vec{n}$, n_i, is the number of services to be invoked for task t_i. Similarly, w_i is the associated time-out value, indicating how long the consumer will wait before invoking another set of n_i services for task t_i.

Now, we are interested in choosing $\vec{n}$ and $\vec{w}$, so that the expected overall utility (or *profit*) $\bar{u}(\vec{n}, \vec{w})$ is maximized (this profit captures the overall utility of a workflow execution to the consumer and so takes into account both the utility reward gained from completing the workflow and the costs incurred from all service invocations). More formally, we let $\bar{u_t}(\vec{n}, \vec{w})$ be the expected utility reward and $\bar{c}(\vec{n}, \vec{w})$ the expected cost. Then we define the expected profit as:

$$\bar{u}(\vec{n}, \vec{w}) = \bar{u_t}(\vec{n}, \vec{w}) - \bar{c}(\vec{n}, \vec{w}). \tag{5}$$

With this, we can specify the service provisioning problem as an optimization task:

$$\max_{\vec{n},\vec{w}\in\mathbb{N}^{|T|}} \quad \bar{u}(\vec{n}, \vec{w}). \tag{6}$$

However, finding a solution for this optimization problem is far from easy. Simply verifying a possible solution (i.e., computing the expected profit $\bar{u}(\vec{n}, \vec{w})$ for given vectors $\vec{n}$ and $\vec{w}$) is very hard. This is because calculating the distribution of the workflow completion time (needed for $\bar{u_t}$) involves the convolution of several probability functions (the duration functions given by $\vec{d}$), which is further complicated by the fact that there are usually interdependencies between the task completion times (as tasks in the workflow depend on their predecessors). In fact, there is currently no known tractable method to solve this problem exactly, even for simple distributions [Dodin 1985; Baccelli et al. 1993].

For this reason, we decided to simplify the problem and devise an algorithm that sacrifices theoretical optimality in favor of a tractable decision algorithm

that produces good results in practice (a *heuristic* algorithm). In particular, we employ a heuristic function for estimating the expected profit, $\tilde{u}(\vec{n}, \vec{w})$. Despite this simplification, we are still faced with the difficult nonlinear integer programming problem of optimizing $\tilde{u}(\vec{n}, \vec{w})$. To address this, we find a good allocation for $\vec{n}$ and $\vec{w}$ by carrying out steepest-ascent hill-climbing [Russell and Norvig 2003], as described in the following section.

4.4.2 *Heuristic Provisioning.* We decided to use a local search algorithm to find a good allocation, because this technique is widely employed for intractable optimization problems [Michalewicz and Fogel 2004]. We chose steepest-ascent hill-climbing specifically, because it is easily implemented and constitutes one of the simplest local search techniques available.[4] This algorithm starts with a random[5] initial allocation for the decision variables $\vec{n}$ and $\vec{w}$, and then gradually improves this by repeatedly picking the best possible neighbor of the current allocation. More specifically, we define a neighbor allocation of $(\vec{n}, \vec{w})$ as $(\vec{n}', \vec{w}')$, so that exactly one component of either vector is different. To restrict the search, we evaluate only a subset of these neighbors by picking those allocations that differ from the original allocation by exactly one integer step, as well as up to four further neighbors (chosen uniformly at random) for each task t_i, so that both an increase and a decrease in n_i and w_i are included. This means that we evaluate up to $8\,|T|$ neighbors at each iteration of the hill-climbing algorithm before proceeding with the best solution. This process is repeated until a maximum is found (i.e., all evaluated neighbors yield a lower or equal estimated profit).

At the centre of this algorithm is clearly the function, $\tilde{u}(\vec{n}, \vec{w})$, which approximates the expected profit of an allocation. Based closely on Eq. (5), we define this as (omitting the parameters for brevity):

$$\tilde{u} = \tilde{r} - \tilde{c}. \tag{7}$$

Here, $\tilde{r}$ and $\tilde{c}$ are estimates of the expected reward and cost of the allocation, respectively (both unconditional on overall success of the workflow). In the following, we describe how these estimates are calculated from a number of parameters for the individual tasks—the success probability p_i, expected cost $\bar{c}_i$, expected completion time $\bar{t}_i$ and variance σ_i^2. First, we outline how the parameters are calculated, given the probabilistic information about service instances discussed in Section 3.3 and an allocation, (n_i, w_i), for each task t_i.

We start by calculating the success probability p_i. This does not depend on n_i, because it is irrelevant for the overall success probability whether services are invoked in series or in parallel. Hence, we let $v_i = |S_i|$ be the total number

[4]This particular choice is not central to our work. We have carried out experiments with a range of local search techniques, including simulated annealing, random restart hill-climbing and simple hill-climbing (where the first better solution is chosen at each iteration), which all achieve similar results.

[5]We generate this by drawing an integer uniformly at random from the the interval $\left[1, \min(|S_i|, \varphi_i)\right]$, where $\varphi_i = \max(10, \lceil -3/\log_{10}(f_i) \rceil)$ for each n_i (this ensures that we do not initially provision an unnecessarily high number of providers), and by drawing a value from the distribution d_i and setting w_i to the nearest integer that is equal or higher.

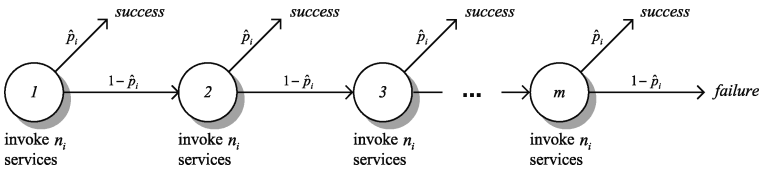


Fig. 4. Possible state transitions as consumer invokes services in sequence.

of service instances for the task, and then re-write Eq. (4):

$$p_i = 1 - (1 - (1 - f_i) \cdot D_i(w_i))^{v_i} \tag{8}$$

Next, we calculate the expected cost $\bar{c}_i$, which depends on the expected number of invocations that are carried out for the task, before it is successful. To illustrate this, Figure 4 shows the possible state transitions of a service-consuming agent. In state 1, the agent invokes the first set of n_i services. With probability $\hat{p}_i = 1 - (1 - (1 - f_i) \cdot D_i(w_i))^{n_i}$ at least one of these is successful, but with probability $1 - \hat{p}_i$ none of them will succeed. In the latter case, the consumer then invokes a new set of n_i services (in state 2). This process repeats until one invocation is successful or no more services are available (for now, we assume that $v_i \bmod n_i = 0$, so that there are up to $m = v_i/n_i$ invocations of exactly n_i services each).

We note from this diagram that the consumer is guaranteed to pay the full cost of invoking all n_i services for task t_i ($n_i c_i$) at least once. After this, the consumer generally has to pay again if the previously invoked set of services has failed (each with probability $1 - \hat{p}_i$). Formally, we let $\hat{f}_i = 1 - \hat{p}_i$ and give the expected cost for task t_i as follows:

$$\bar{c}_i = \underbrace{n_i c_i + \hat{f}_i \cdot (n_i c_i + \hat{f}_i \cdot (n_i c_i + \hat{f}_i \cdot (\cdots + \hat{f}_i \cdot (n_i c_i) \cdots)))}_{m \text{ instances of } n_i c_i} \tag{9}$$

$$= n_i c_i + \hat{f}_i \cdot n_i c_i + \hat{f}_i{}^2 \cdot n_i c_i + \hat{f}_i{}^3 \cdot n_i c_i + \cdots + \hat{f}_i{}^{m-1} \cdot n_i c_i \tag{10}$$

$$= n_i c_i \cdot \left(1 + \hat{f}_i + \hat{f}_i{}^2 + \cdots + \hat{f}_i{}^{m-1}\right) \tag{11}$$

$$= n_i c_i \sum_{k=0}^{m-1} \hat{f}_i{}^k. \tag{12}$$

This summation grows with the number of available services, v_i. To make it more tractable, we note that it is a geometric series and rewrite it as follows (assuming $\hat{f}_i < 1$):

$$\bar{c}_i = n_i c_i \cdot \frac{1 - \hat{f}_i{}^m}{1 - \hat{f}_i}. \tag{13}$$

Equation (13) is the expected cost for task t_i, assuming that $v_i \bmod n_i = 0$. To generalize this result for cases where $v_i \bmod n_i \neq 0$, we note that the consumer will invoke all remaining services on its last try. For this case, we let $m = \lfloor v_i/n_i \rfloor$ be the number of full invocations (n_i services each) and $r = v_i \bmod n_i$ be the remaining number of services after m invocations. Then, the consumer will pay $c_r = c_i r$ for the last invocation if all previous services have failed (which

happens with probability $\hat{f}_i{}^m$). To generalize Eq. (13), we simply include this cost:

$$\bar{c}_i = n_i c_i \cdot \frac{1 - \hat{f}_i{}^m}{1 - \hat{f}_i} + \hat{f}_i{}^m c_i r. \tag{14}$$

Finally, we are interested in calculating the expected time $\bar{t}_i$ until the task is completed. We define this as the mean time until the first service completes the task successfully, conditional on overall success (i.e., that at least one service is successful). First, we let μ_i be the mean duration of a single successful invocation. In other words, given that n_i services are invoked and that at least one completes successfully before time-out w_i, μ_i is the expected duration of the fastest successful service (as observed by the consumer).

To calculate μ_i, we first let $\hat{D}_i(x)$ be the cumulative (nonconditional) probability that at least one out of n_i services has finished successfully by time x:

$$\hat{D}_i(x) = 1 - (1 - (1 - f_i) \cdot D_i(x))^{n_i} \tag{15}$$

With this, we calculate μ_i as follows:

$$\mu_i = \frac{1}{\hat{D}_i(w_i)} \sum_{k=1}^{w_i} k \cdot (\hat{D}_i(k) - \hat{D}_i(k-1)) \tag{16}$$

Now, to calculate the overall expected time of the task, we again assume that $v_i \bmod n_i = 0$ and follow similar reasoning as for the expected cost by considering Figure 4. When the consumer succeeds after state 1, its expected duration is then μ_i, and if it succeeds after state 2, the expected duration is $w_i + \mu_i$. We formulate the general case, after the kth invocation as:

$$\bar{d_k} = (k-1) \cdot w_i + \mu_i. \tag{17}$$

The associated nonconditional probability of this event (succeeding after the kth invocation) is $\hat{f}_i{}^{k-1}(1 - \hat{f}_i)$. Using this, and conditioning on an overall success, we can now write the expected time for task t_i as:

$$\begin{aligned}
\bar{t}_i &= \frac{1}{p_i} \cdot \sum_{k=1}^{m} \bar{d}_k \, \hat{f}_i{}^{k-1}(1 - \hat{f}_i) \\
&= \frac{1}{p_i} \cdot \sum_{k=1}^{m} \big((k-1) \cdot w_i + \mu_i\big) \cdot \hat{f}_i{}^{k-1}(1 - \hat{f}_i) \\
&= \frac{1}{p_i} \cdot \sum_{k=0}^{m-1} (k \cdot w_i + \mu_i) \cdot \hat{f}_i{}^{k}(1 - \hat{f}_i).
\end{aligned} \tag{18}$$

Again, it is possible to rearrange this and write it in closed form. In particular, we again assume that $\hat{f}_i < 1$ and note that $\sum_{k=1}^{\infty} \hat{f}_i{}^k k = \hat{f}_i / \left(\hat{f}_i - 1\right)^2$.

$$\begin{aligned}
\bar{t}_i p_i &= \sum_{k=0}^{m-1} (k \cdot w_i + \mu_i) \cdot \hat{f}_i{}^k \left(1 - \hat{f}_i\right) \\
&= (1 - \hat{f}_i) \sum_{k=0}^{m-1} \hat{f}_i{}^k (\mu_i + k w_i) \\
&= (1 - \hat{f}_i) \left(\sum_{k=0}^{m-1} \hat{f}_i{}^k \mu_i + \sum_{k=1}^{m-1} \hat{f}_i{}^k k w_i \right) \\
&= (1 - \hat{f}_i) \left(\mu_i \frac{1 - \hat{f}_i{}^m}{1 - \hat{f}_i} + w_i \left(\frac{\hat{f}_i - \hat{f}_i{}^m}{(1 - \hat{f}_i)^2} - \frac{(m-1)\hat{f}_i{}^m}{1 - \hat{f}_i} \right) \right) \\
&= \mu_i \left(1 - \hat{f}_i{}^m\right) + w_i \frac{\hat{f}_i - m\hat{f}_i{}^m + (m-1)\hat{f}_i{}^{m+1}}{1 - \hat{f}_i}. \qquad (19)
\end{aligned}$$

To generalize this, when $v_i \bmod n_i \neq 0$, we again let $m = \lfloor v_i / n_i \rfloor$ be the number of full invocations and $r = v_i \bmod n_i$ the remaining services. We also let λ_i be the mean duration to the first success when r services are invoked (calculated analogously to μ_i in Eq. (16)), and we let $\check{f}_r$ be the probability of failure when invoking r services in parallel. Then we can add the impact of the remaining services to extend Eq. (19):

$$\bar{t}_i = \frac{1}{p_i} \left(\mu_i \left(1 - \hat{f}_i{}^m\right) + w_i \frac{\hat{f}_i - m\hat{f}_i{}^m + (m-1)\hat{f}_i{}^{m+1}}{1 - \hat{f}_i} + \hat{f}_i{}^m (1 - \check{f}_r)(\lambda_i + m w_i) \right). \qquad (20)$$

Finally, to calculate the variance, σ_i^2, of the task, we let C_i be a random variable representing the duration of the task, conditional on its success (note, its expected value, $\mathrm{E}(C_i)$, is equal to $\bar{t}_i$). We are interested in the variance of this variable, $\mathrm{VAR}(C_i)$, which we calculate as follows:

$$\begin{aligned}
\sigma_i^2 &= \mathrm{VAR}(C_i) \\
&= \mathrm{E}\left(C_i^2\right) - \mathrm{E}(C_i)^2. \qquad (21)
\end{aligned}$$

We can calculate $\mathrm{E}(C_i)^2$ as given by Eq. (20), but to calculate $\mathrm{E}(C_i^2)$, further steps are necessary. First, we consider two cases, as before: (1) the task is successful during the first $m = \lfloor v_i / n_i \rfloor$ full invocations, and (2) the task is successful in the last invocation with $r = v_i \bmod n_i$ parallel services (if $r \neq 0$). We use two random variables to denote the durations in each case — A_i and B_i, respectively (again, these are conditional on the task being successful in each case). In order to treat both cases separately, we can now rewrite $\mathrm{E}(C_i^2)$, letting P_A be the probability that case (1) occurs, and P_B the probability that case (2) occurs, both conditional on overall success:

$$\begin{aligned}
\mathrm{E}\left(C_i^2\right) &= P_A \mathrm{E}\left(A_i^2\right) + P_B \mathrm{E}\left(B_i^2\right) \\
&= \frac{1 - \hat{f}_i^m}{1 - \check{f}_r \hat{f}_i^m} \mathrm{E}\left(A_i^2\right) + \frac{\hat{f}_i^m (1 - \check{f}_r)}{1 - \check{f}_r \hat{f}_i^m} \mathrm{E}\left(B_i^2\right). \qquad (22)
\end{aligned}$$

Furthermore, we separate each of these durations into the total time spent waiting for unsuccessful invocations that are timed-out (we denote these as A_{Wi} and B_{Wi}) and the time that passes during the last invocation before the first service is successful (denoted as A_{Di} and B_{Di}), and we note that these two components are independent of each other in our model. Beginning with the first case, we thus write:

$$\begin{aligned}
\mathrm{E}(A_i^2) &= \mathrm{VAR}(A_i) + \mathrm{E}(A_i)^2 \\
&= \mathrm{VAR}(A_{Wi}) + \mathrm{VAR}(A_{Di}) + (\mathrm{E}(A_{Wi}) + \mathrm{E}(A_{Di}))^2 \\
&= \mathrm{E}(A_{Wi}^2) - \mathrm{E}(A_{Wi})^2 + \mathrm{E}(A_{Di}^2) - \mathrm{E}(A_{Di})^2 + (\mathrm{E}(A_{Wi}) + \mathrm{E}(A_{Di}))^2 \\
&= \mathrm{E}(A_{Wi}^2) + \mathrm{E}(A_{Di}^2) + 2\mathrm{E}(A_{Wi})\mathrm{E}(A_{Di}). \qquad (23)
\end{aligned}$$

The expected duration of a single invocation, $\mathrm{E}(A_{Di})$, is equal to μ_i, which we calculate using Eq. (16). The expected squared duration, $\mathrm{E}(A_{Di}^2)$, is similarly calculated by multiplying the term inside the summation by k^2 instead of k. The expected waiting time, $\mathrm{E}(A_{Wi})$, is obtained from Eq. (19):

$$\mathrm{E}(A_{Wi}) = \frac{w_i}{(1-\hat{f}_i)(1-\hat{f}_i^m)}(\hat{f}_i - m\hat{f}_i^m + (m-1)\hat{f}_i^{m+1}). \qquad (24)$$

To derive the expected squared waiting time, $\mathrm{E}(A_{Wi}^2)$, we follow similar reasoning as for Eq. (18):

$$\begin{aligned}
\mathrm{E}(A_{Wi}^2) &= \frac{(1-\hat{f}_i)w_i^2}{1-\hat{f}_i^m}\sum_{k=0}^{m-1} k^2\hat{f}_i^k \qquad (25) \\
&= \frac{w_i^2}{(1-\hat{f}_i^m)(1-\hat{f}_i)^2}\big(\hat{f}_i + \hat{f}_i^2 - m^2\hat{f}_i^m \\
&\quad - (2m+1-2m^2)\hat{f}_i^{m+1} + (2m-1-m^2)\hat{f}_i^{m+2}\big) \qquad (26)
\end{aligned}$$

Next, when $v_i \bmod n_i \neq 0$, we also need to calculate the expected squared duration if the consumer is successful on the last invocation, $\mathrm{E}(B_i^2)$. This is done analogously to Eq. (23), simplified by the fact that a constant waiting time (mw_i) is associated with the last invocation:

$$\begin{aligned}
\mathrm{E}(B_i^2) &= \mathrm{VAR}(B_i) + \mathrm{E}(B_i)^2 = \mathrm{E}(B_{Wi}^2) + \mathrm{E}(B_{Di}^2) + 2\mathrm{E}(B_{Wi})\mathrm{E}(B_{Di}) \\
&= (mw_i)^2 + \mathrm{E}(B_{Di}^2) + 2mw_i\mathrm{E}(B_{Di}). \qquad (27)
\end{aligned}$$

The remaining terms, $\mathrm{E}(B_{Di})$ and $\mathrm{E}(B_{Di}^2)$, are calculated as $\mathrm{E}(A_{Di})$ and $\mathrm{E}(A_{Di}^2)$, discussed above.

We have now finished analyzing the performance characteristics of a single task t_i given an allocation (n_i, w_i) and some knowledge about the services available for the task. In particular, we can calculate the success probability of the task (p_i in Eq. (8)), the expected cost of attempting the task ($\bar{c}_i$ in Eq. (14)), the expected completion time of the task, conditional on its success ($\bar{t}_i$ in Eq. (20)), and its variance (σ^2 in Eq. (21)). Given these calculations for each task, we are now interested in estimating the expected reward $\tilde{r}$ and the expected cost $\tilde{c}$ for the overall workflow, which are required for our heuristic utility function given in Eq. (7).

The estimated total cost is the sum of all task costs, each multiplied by the respective success probabilities of their predecessors in the workflow (where r_i is the probability that task t_i is ever reached):

$$\tilde{c} = \sum_{\{i|t_i \in T\}} r_i \bar{c}_i \tag{28}$$

$$r_i = \begin{cases} 1 & \text{if } \forall t_j \cdot \big((t_j \mapsto t_i) \notin E\big) \\ \prod_{\{j|(t_j \mapsto t_i) \in E\}} p_j & \text{otherwise.} \end{cases} \tag{29}$$

Next, to estimate the expected reward of the allocation, we need a duration distribution for the complete workflow (again, conditional on overall success). To this end, we employ a technique from operations research [Malcolm et al. 1959], and evaluate the *critical path* of the workflow (i.e., the path that maximises the sum of all mean task durations along it). To obtain an estimated distribution for the duration of this path, we approximate it with a normal distribution that has a mean λ_W equal to the sum of all mean task durations along the path and a variance v_W equal to the sum of the respective task variances. This approach exploits the central limit theorem, which states that the sum of arbitrary independent random variables can be approximated using such a distribution.[6] Hence, the corresponding probability density function for the workflow duration is:

$$d_W(x) = \frac{1}{\sqrt{v_W 2\pi}} \, \mathrm{e}^{-\frac{(x-\lambda_W)^2}{2v_W}} \tag{30}$$

with

$$\lambda_W = \sum_{\{i|t_i \in \mathcal{P}\}} \bar{t}_i \tag{31}$$

$$v_W = \sum_{\{i|t_i \in \mathcal{P}\}} \sigma_i^2, \tag{32}$$

where $\mathcal{P} = \{t_i \mid t_i \text{ is on the critical path}\}$.

Next, we use the distribution $d_W(x)$ to estimate the expected reward of the allocation. In so doing, we assume that workflow finishing times can be continuous. This allows us to derive a closed, analytical solution, but also means that we may slightly overestimate the reward. In practice, we believe that the introduced error will be negligible, especially when time steps are small, and our results support this. To this end, we assume overall success and denote the corresponding expected reward with $\tilde{r_s}$:

$$\tilde{r_s} = \int_0^\infty d_W(x) u(x) \, \mathrm{d}x. \tag{33}$$

[6]This theorem holds when the number of variables approaches infinity and makes some assumptions about the variables, for example, that their third moments must be finite [DeGroot and Shervish 2002]. However, we have verified that this approximation works well in practice, even when considering small workflows (see Section 5.7).

In order to calculate this, we let $D_W(x) = \int_{-\infty}^{x} d_W(y)\,\mathrm{d}y$ be the cumulative probability function[7] of $d_W(x)$, we let $D_{\max} = D_W(t_{\max})$ be the probability that the workflow will finish no later than the deadline $t_{\max}$ and $D_{\text{late}} = D_W(t_0) - D_W(t_{\max})$ the probability that the workflow will finish after the deadline but no later than time $t_0 = \frac{u_{\max}}{\delta} + t_{\max}$ (both conditional on overall success).

Next, we consider three distinct cases, as derived from Eq. (2) for $u(t)$. First, the workflow may finish within the deadline $t_{\max}$—in this case, which happens with probability $D_{\max}$, the consumer will receive the full reward, $u_{\max}$. Second, the workflow may finish after t_0—this happens with probability $1 - D_W(t_0)$, and here the consumer receives no reward (and so we can ignore it). Finally, the workflow may finish between these two times, which happens with probability D_{late}. Because $u(t)$ is linear on this interval, we can calculate the expected reward in this case by applying $u(t)$ to the mean time on the interval, which we denote by $\bar{t}_{\text{late}}$. Hence, we can rewrite Eq. (33):

$$\tilde{r_s} = D_{\max} \cdot u_{\max} + D_{\text{late}} \cdot u(\bar{t}_{\text{late}}). \tag{34}$$

Now, we calculate $\bar{t}_{\text{late}}$:

$$\begin{aligned} \bar{t}_{\text{late}} &= \frac{1}{D_{\text{late}}} \int_{t_{\max}}^{t_0} d_W(x) x \,\mathrm{d}x \\ &= \lambda_W + \left(e^{\frac{-(t_{\max} - \lambda_W)^2}{2v_W}} - e^{\frac{-(t_0 - \lambda_W)^2}{2v_W}} \right) \frac{\sqrt{v_W}}{D_{\text{late}} \cdot \sqrt{2\pi}}. \end{aligned} \tag{35}$$

Finally, this reward ($\tilde{r_s}$) is only obtained when the workflow is successful. Hence, we calculate the overall probability of success, p, as the product of all p_i:

$$p = \prod_{\{i \mid t_i \in T\}} p_i. \tag{36}$$

This allows us to summarize our heuristic utility function as follows:

$$\tilde{u} = p \cdot (D_{\max} \cdot u_{\max} + D_{\text{late}} \cdot u(\bar{t}_{\text{late}})) - \tilde{c} \tag{37}$$

Using this heuristic function, it is now possible to use steepest-ascent hill-climbing as described at the beginning of this section. Through observations, we have seen that our hill-climbing algorithm quickly converges to a good solution.[8] In particular, the heuristic function $\tilde{u}$ can be solved efficiently in quadratic time. The bottleneck here is the calculation for Eqs. (28) and (29). However, after the initial calculation, only small adjustments need to be made at each iteration of the hill-climbing procedure, further reducing the run-time of calculating $\tilde{u}$. In this case, it is bounded by the critical path problem used in Eqs. (31) and (32), which has a run-time in $O(|T| + |\mathcal{E}|)$ where $|T|$ is the

[7]This is a common function that is usually approximated numerically. In our implementation, we use the SSJ library (`http://www.iro.umontreal.ca/~simardr/ssj`).

[8]On average, around six iterations are needed per task in the workflow. During the experimental evaluation of our algorithm (see Section 5), a solution was typically found within 250 ms (10 tasks) or 5 s (50 tasks) on a 3-GHz Pentium 4 with 1 GB RAM.

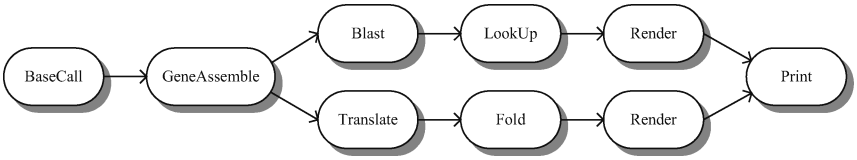


Fig. 5. Example bioinformatics workflow (based on workflows described in Smith et al. [1997], Kochut et al. [2003], O'Brien et al. [2004]).

number of tasks in the workflow and $|\mathcal{E}|$ the number of direct, non-transitive edges.[9]

To illustrate the behavior of our *flexible* strategy, we briefly outline the provisioning of an example workflow in the following section.

4.4.3 *Illustrative Example.* In this section, we discuss how an example workflow is provisioned by our algorithm, and how the various performance measures introduced in Section 4.4.2 are calculated and used in practice. To this end, we use a simple workflow from the bioinformatics domain — an area that relies heavily on computationally intensive services and that has increasingly seen the establishment of large distributed Grid systems for sharing resources, as exemplified by the myGrid project [Oinn et al. 2006]. For our example, we assume that a scientist has just sequenced a previously unknown gene of a bacterium, and is now interested in visualising the shape of the associated protein. For this, she has to carry out a number of tasks, which are shown in Figure 5.

Her initial data comprises a large set of overlapping DNA fragments in the form of chromatograms, as is common in shotgun DNA sequencing [Ewing et al. 1998]. These show characteristic light traces at different wavelengths, corresponding to the four bases found in a DNA sequence. As these traces typically contain some noise and errors, the scientist first needs to run a base-calling service (*BaseCall*). This translates the chromatograms to the corresponding base sequences, attaching a quality value to each base in the process that denotes how accurate the assignment of the base is. The resulting base sequences are then assembled to a single continuous DNA sequence by identifying and merging overlapping fragments, using the quality values to find and repair errors. This task is performed by a sequence-assembling service, which also identifies and isolates the coding region of the gene (*GeneAssemble*).

When the coding region of the gene has been assembled, it is then translated to the corresponding amino acid sequence using a simple translation service (*Translate*). As the primary structure of the protein, this forms the input to the computationally-intensive folding service (*Fold*), which predicts the 3-dimensional shape of the protein based on a search for the conformation with the lowest free energy. The output of this—a file containing the tertiary structural data—is then rendered in high resolution using an appropriate graphics service (*Render*). In parallel with the folding simulation, the scientist is also

[9]We also assume that the probability density functions of service invocation durations and related expected values, as calculated in Eq. (16), can be efficiently calculated (or else approximated).

Table I. Service Types Used in the Example Workflow

Service	Fail. Prob.	Cost ($)	Number	Duration	Mean (min.)	Var.
BaseCall	0.2	1	50	Gamma(1.5,2)	3	6
GeneAssemble	0.1	5	50	Gamma(5,2)	10	20
Blast	0.3	2	500	Gamma(5,3)	15	45
LookUp	0.5	5	10	Gamma(1.5,1.5)	2.25	3.375
Render	0.1	10	25	Gamma(30,3)	90	270
Translate	0.7	0.5	200	Gamma(1,1)	1	1
Fold	0.2	10	5	Gamma(3,30)	90	2700
Print	0.2	2	20	Gamma(2,3)	6	18

interested in comparing the new gene to previously discovered sequences. To this end, she searches through public collections of known proteins to find the closest match using a specialised service (*Blast*), and then accesses commercial database services to retrieve structural information about the protein (*LookUp*). This is rendered again, and both images are printed as part of a report on a local printer (*Print*).

The constituent service types for the scientist's workflow are detailed in Table I, along with their failure probabilities, invocation costs, numbers available, their respective duration distributions[10] and associated means and variances. These were chosen to represent a set of services with variable performance characteristics—for example, *Translate* is a cheap, fast and unreliable service type, while *Render* is expensive, slow and reliable.

Now, for our illustrative example, we assume that the scientist has a deadline of four hours, and values the workflow at $150, which decreases by $1 for each minute that it is late. Figure 6 shows the initial allocation for the workflow. As outlined in Section 4.4.2, the algorithm begins here by randomly provisioning service instances for the constituent tasks of the workflow.

To illustrate the calculations[11] our algorithm performs on this allocation, we consider the upper *Render* task in the workflow (t_4). Here, the algorithm first calculates the probability of success for the task, p_4, using Eq. (8). As there are a number of service instances ($v_4 = 25$), this probability is $p_4 = 1-(1-(1-0.1)\cdot 0.62)^{25} = 1.00$. Next, the algorithm calculates the expected cost, $\bar{c}_4$, using Eq. (14). This is high ($\bar{c}_4 = 1 \cdot 10 \cdot \frac{1-0.4437^{25}}{1-0.4437} = 17.98$), because the initial allocation will ignore any services that finish after the mean duration (even if they are successful). Finally, the expected completion time, $\bar{t}_4$, is calculated using Eq. (20). Again, this is high ($\bar{t}_4 = \frac{1}{1} \cdot (80.22155 \cdot (1 - \hat{f}_4^{25}) + 94 \cdot (\hat{f}_4 - 25 \cdot$

[10]We assume that services in this example follow a gamma distribution Gamma(k, θ) with pdf $p(x, k, \theta) = x^{k-1}e^{-\frac{x}{\theta}}\Gamma(k)^{-1}\theta^{-k}$, which has been chosen because it is well suited for uncertain service times that are always positive, but are not usually bounded above. The gamma distribution also includes common other distributions such as the exponential and Erlang distributions, both of which are often used in the analysis of service and queueing times [Trivedi 2001]. However, this choice is only for illustrative purposes - in practice, an arbitrary distribution can be used to model service durations.

[11]For readability, all values presented here are reported to two decimal places, except where additional precision is necessary during the calculations.

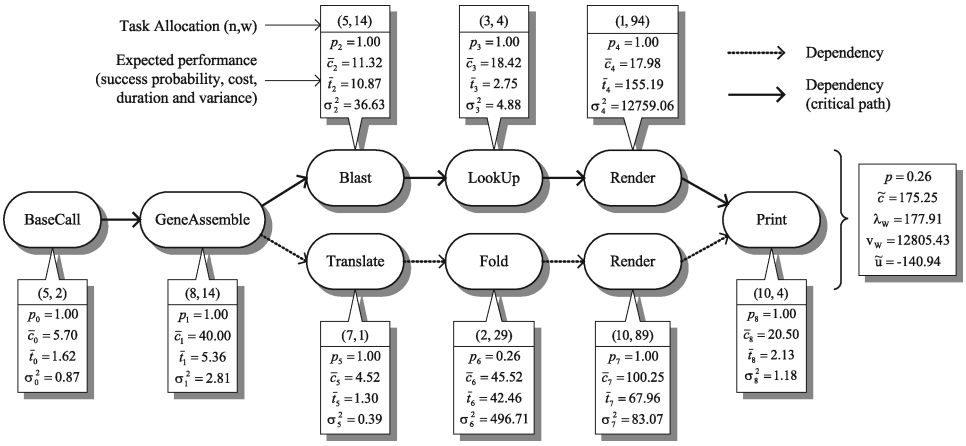


Fig. 6. Initial provisioning allocation.

$\hat{f}_4^{25} + (25-1) \cdot \hat{f}_4^{25+1}) \cdot \frac{1}{1-\hat{f}_4}) = 155.19$, where $\hat{f}_4 = 0.44367$) for the same reason as the expected cost.

Given these values for all tasks in the workflow, the algorithm next derives the overall expected performance measures for the workflow (these are summarized in the box to the right of the workflow). First, the overall success probability, p, is calculated using Eq. (36). This is low, due to the inappropriate time-out value for the *Fold* task (t_6), which results in a high failure probability of that task ($p = \prod_{\{i|t_i \in T\}} p_i = 1.00^7 \cdot 0.26 = 0.26$). The expected cost, $\tilde{c}$, is estimated next using Eq. (28). In this case, we derive an estimated cost of $\tilde{c} = \sum_{\{i|t_i \in T\}} r_i \bar{c}_i = 175.25$ for the whole workflow. After this, the algorithm estimates the distribution of the overall completion time by summing the expected completions times and variances along the critical path, using Eqs. (31) and (32). This yields a mean of $\lambda_W = \sum_{\{i|t_i \in \mathcal{P}\}} \bar{t}_i = 1.620 + 5.356 + 10.867 + 2.747 + 155.187 + 2.130 = 177.91$ and a variance of $v_W = \sum_{\{i|t_i \in \mathcal{P}\}} \sigma_i^2 = 0.87 + 2.81 + 36.63 + 4.88 + 12759.06 + 1.18 = 12805.43$. Using these as the mean and variance of a normal distribution ($d_W(x)$ in Eq. (30), which was derived using the central limit theorem), we estimate that the workflow will finish within the deadline $t_{\max}$ with probability $D_{\max} = \int_{-\infty}^{t_{\max}} d_W(y)\mathrm{d}y = 0.708395$. We also estimate that the probability of finishing between the deadline and t_0 is $D_{\text{late}} = \int_{t_{\max}}^{t_0} d_W(y)\mathrm{d}y = 0.261157$. In the latter case, we calculate the expected completion time using Eq. (35) ($\bar{t}_{\text{late}} = 296.766592$). Finally, using these intermediate values in Eq. (37) yields a total utility estimate of $\tilde{u} = 0.262624 \cdot (0.708395 \cdot 150 + 0.261157 \cdot u(296.766592)) - 175.245220 = -140.94$. This is low because of the high degree of parallelism in the workflow (resulting in unnecessary expenses) and the low overall success probability (resulting in a low estimated reward).

To improve this initial allocation, our algorithm now repeatedly considers a number of neighbor allocations and, at each iteration, picks the one that promises the highest estimated profit. This is repeated until no more improvements can be made. Figure 7 shows the final allocation found by our algorithm, which includes several tasks where providers have been provisioned in parallel,

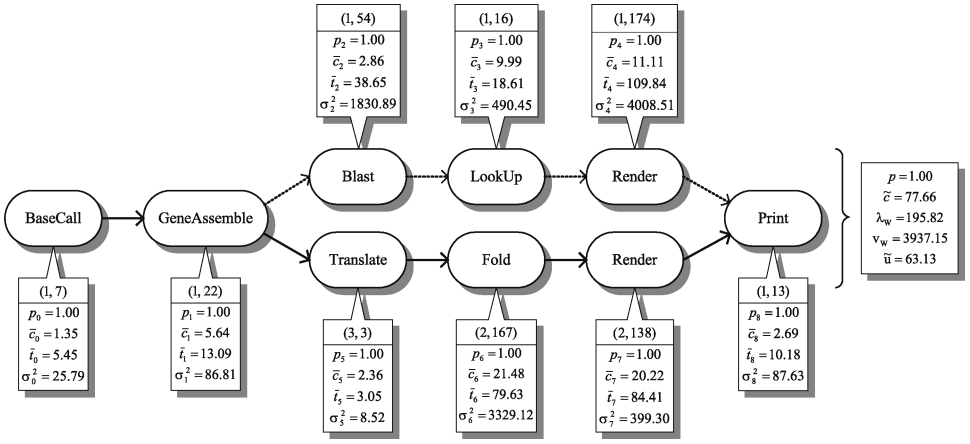


Fig. 7. Finally provisioned workflow.

but mostly relies on serial provisioning as this saves money. Contrasting this with the initial allocation, the improvements are clearly visible—for example, the expected cost of the *Render* task in the upper branch (t_4) has now been reduced from $\bar{c}_4 = 17.98$ to 11.11 and its expected duration has been lowered, simply by choosing a more appropriate waiting time (from $\bar{t}_4 = 155.19$ to 109.84). It is also evident that the structure of the workflow has been taken into account—two providers have been provisioned in parallel for the lower *Render* task (t_7), despite being the same type of service. This means that the task is faster ($\bar{c}_7 = 84.41$), but also more expensive ($\bar{c}_7 = 20.22$) than its counterpart in the upper branch. This is beneficial, because the durations of the lower tasks are generally longer, and so the consumer has to invest more resources in order to meet its workflow deadline. Overall, the consumer now expects to finish within the deadline $t_{\max} = 240$ with probability $D_{\max} = 0.7593$, and between the deadline and $t_0 = 390$ with probability $D_{\text{late}} = 0.2397$. In the latter case, its expected finishing time is $\bar{t}_{\text{late}} = 276.4548$, leading to an overall estimated utility of $\tilde{u} = 0.9977 \cdot (0.7593 \cdot 150 + 0.2397 \cdot u(276.4548)) - 77.6572 = 63.13$.

To give a second example, Figure 8 shows the same workflow in a scenario where the scientist requires her results in a far shorter time period (within 150 minutes), where she values the outcome more highly (the value is now \$1,000), and where the penalty is higher than in the previous example (\$20 per minute). Here, our algorithm is using a far higher level of redundancy than previously, because that allows the agent to finish more quickly and reliably. For example, for the *Render* task in the lower branch, the algorithm has now provisioned five services in parallel, which is very expensive ($\bar{c}_7 = 50.00$), but also results in a low expected duration ($\bar{t}_7 = 73.32$) necessary to meet the overall deadline. Nevertheless, the algorithm still chooses to provision a single service for the *LookUp* task. As before, this is because the tasks on the lower branch take longer, and so the consumer can save some costs by executing the upper tasks in series. Overall, the consumer now expects to finish within the deadline $t_{\max} = 150$ with probability $D_{\max} = 0.78$ and it is late with probability $D_{\text{late}} = 0.22$ (in

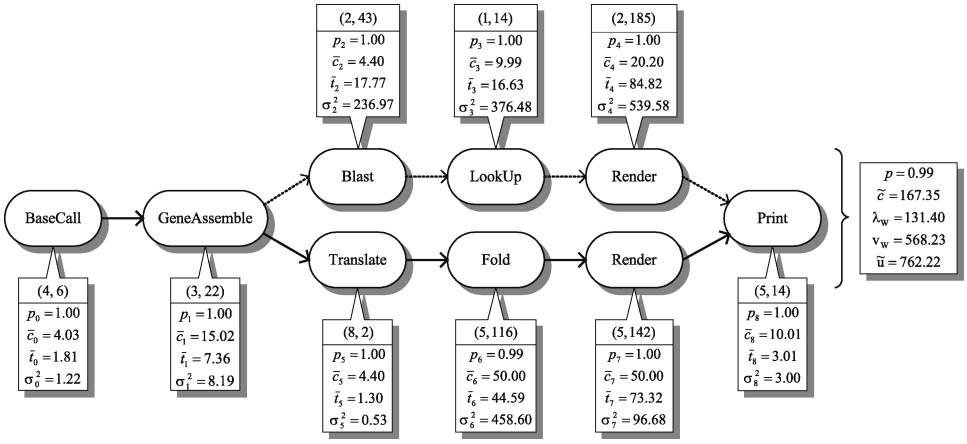


Fig. 8. Provisioned workflow with shorter deadline and higher reward.

which case its expected finishing time is $\bar{t}_{\text{late}} = 163.23$). Due to the high levels of redundancy, the estimated expected cost has now more than doubled compared to the previous case ($\tilde{c} = 167.35$), but the overall higher reward results in a high estimated utility of $\tilde{u} = 762.22$ that justifies the expenses.

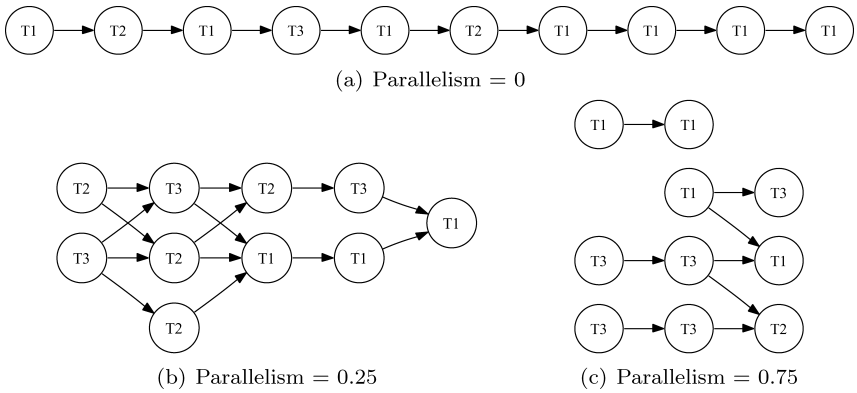
In order to evaluate this strategy and to compare it against less flexible approaches, in the following section, we describe a set of experiments that we carried out to this end.

5. EXPERIMENTAL EVALUATION

In this section, we experimentally compare our proposed strategies to the currently predominant *naïve* approach.[12] The aim of this part of our work is to compare the performance of our strategies to current approaches when there is some uncertainty in the behavior of services. We also intend to verify that our flexible strategy in particular takes appropriate decisions and makes an overall profit over a variety of environments while achieving high success rates. We decided to conduct an experimental study (rather than an analytical one), because of the inherent difficulty of calculating workflow completion distributions (see Section 4.4.1).

To this end, we investigate the average profit gained by all strategies, as well as the average proportion of successfully completed workflows. We begin in Section 5.1 by describing our experimental testbed and our methodology. In Section 5.2, we outline a set of hypotheses to guide our experiments and in Sections 5.3–5.5 we present our results. Then, in Section 5.6, we show how our strategy deals with larger workflows, and in Section 5.7, we compare it to the optimal strategy (for a simplified scenario).

[12]As we assume limited information about each task, the *naïve* strategy also subsumes a number of other QoS optimization approaches that were discussed in Section 2. This is because they rely on more detailed information about individual service instances and user-specified constraints that are not available in our model.



(a) Parallelism = 0

(b) Parallelism = 0.25

(c) Parallelism = 0.75

Fig. 9. Several random workflows with 10 tasks, 3 different services types (indicated by the task labels) and varying degrees of parallelism.

5.1 Testbed and Methodology

In order to analyze our strategies experimentally, we developed a computer simulation of a service-oriented system. In this simulation, the system is populated by agents offering services, as described in Section 3. During each experimental run, a random workflow is first created according to some predefined variables. These include the number of tasks in the workflow, the service types that should be included, and a parameter indicating the parallelism of the workflow. The latter is a variable ranging from 0 to 1, where 0 results in completely linear workflows (i.e., the task dependencies form a total order), while 1 causes workflows to be completely parallel (i.e., there are no dependencies between tasks). Any intermediate value indicates the number of edges that should be introduced as a proportion of the number of edges possible[13] (see Figure 9). This workflow is then executed by a service-consuming agent using one of the strategies outlined in Section 4. These runs are episodic and each involves the execution of exactly one workflow, with no interactions between successive runs.

To analyze the performance of a particular strategy, our simulation executes a large number of experimental runs (the data in this section was collected using 1,000 runs for each experimental setup) and then records the following statistics[14]:

—The proportion of successful workflows for the strategy (where the strategy completes the workflow within time t, so that $u(t) > 0$).

—The average profit of the strategy (the profit of a workflow execution is the difference between the utility reward $u(t)$ for completing the workflow and the incurred cost).

These indicate the extent to which the consumer agent manages to complete its workflows within the given time-constraints and whether it manages to

[13]We implement this by randomly populating an adjacency matrix until the given threshold is reached.

[14]To test for statistical significance, we also record the variances of all averages.

achieve a high average profit at the same time, without making an overall loss.

To concentrate on the core issues of uncertain service behavior and because our approach does not deal directly with differentiating between individual services of the same type at this time, we examined environments with homogeneous service instances (i.e., all services share the same success probability and duration distributions) for a given service type.

For the data presented in this section, we used workflows with 10 tasks and a linearity parameter of 1 (i.e., without parallel tasks). This means that the experiments presented here are particularly relevant to scenarios where workflows are highly interdependent. By using such linear workflows, we were also able to check some of our results analytically to verify that our simulation is correct (in particular, we verified the results presented in Sections 5.3 and 5.4).

Furthermore, we assumed that there were 1,000 services for every task with each service having a cost of 10 and a gamma distribution with shape $k = 2$ and scale $\theta = 10$ as the probability distribution of the service duration. We set a deadline of 400 time units for each workflow, an associated maximum utility of 1,000 and a penalty of 10 per time unit. We also performed similar experiments in a variety of environments, including heterogeneous and parallel tasks, and observed the same broad trends that are presented in the following section (some of these results are presented in Section 5.6).

To prove the statistical significance of our results, we averaged data over 1,000 test runs and performed an analysis of variance (ANOVA) where appropriate to determine whether the strategies we tested produced significantly different results [Cohen 1995]. When this was the case, we carried out pairwise comparisons using the least significant difference (LSD) test. Thus, all results reported in the following sections are statistically significant (at the $p = 0.001$ level).

5.2 Hypotheses

Before discussing the results of our experiments, we outline four hypotheses that drive our investigation. The first two are concerned with the effects of the two basic, nonflexible strategies, *parallel(n)* and *serial(w)*. The aim of these hypotheses is to show that it is possible to achieve better results using simple techniques for handling failures than when relying on the *naïve* strategy.

Hypothesis 1. Adopting strategy *parallel(n)* in uncertain environments can lead to an improvement in the average profit over the *naïve* strategy.

Hypothesis 2. Adopting strategy *serial(w)* in uncertain environments can lead to an improvement in the average profit over the *naïve* strategy.

The other two hypotheses are concerned with evaluating the *flexible* strategy. Here, we present two hypotheses concerned with the average profit and the success probability. This presents the flexible strategy in more detail than the previous two strategies due to its importance to our research.

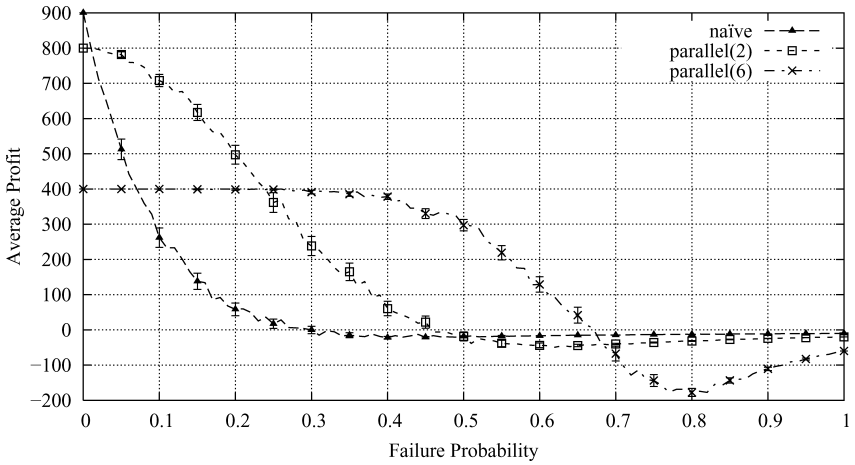


Fig. 10. Effect of provisioning different numbers of services in parallel (data shown with 95% confidence intervals).

Hypothesis 3. The *flexible* strategy produces a higher profit than any of the other examined strategies, averaged over all cases.

Hypothesis 4. The *flexible* strategy successfully completes a higher proportion of workflows than any of the other examined strategies, averaged over all cases.

To evaluate Hypotheses 1–4, we tested each of the four strategies *naïve*, *parallel(n)*, *serial(w)* and *flexible* using the same experimental variables (as outlined in Section 5.1). We summarize the results by discussing each hypothesis separately.

5.3 Parallel Provisioning (Hypothesis 1)

In our first experiment, we compared the performance of strategy *parallel(n)*[15] with the *naïve* approach in environments where services have a varying probability of failure, as shown in Figure 10 (throughout this section, we vary the failure probability in steps of 0.01 from 0 to 1). From this, it is clear that there is a considerable difference in performance between the different strategies—the average profit gained by the *naïve* strategy falls dramatically as soon as failures are introduced into the system. In this case, the average profit gained by provisioning single services falls to around 0 when the failure probability of services is only 0.3. A statistical analysis reveals that the *naïve* strategy dominates the other two when there is no uncertainty in the system. However, as soon as the failure probability is raised to 0.02, *parallel(2)* begins to dominate the other strategies. Between 0.35 and 0.65, *parallel(6)* then becomes the dominant strategy as increased service redundancy leads to a higher probability of

[15]Here, we arbitrarily chose $n = 2$ and $n = 6$ as representative of the general trends displayed by the strategy as more services are provisioned.

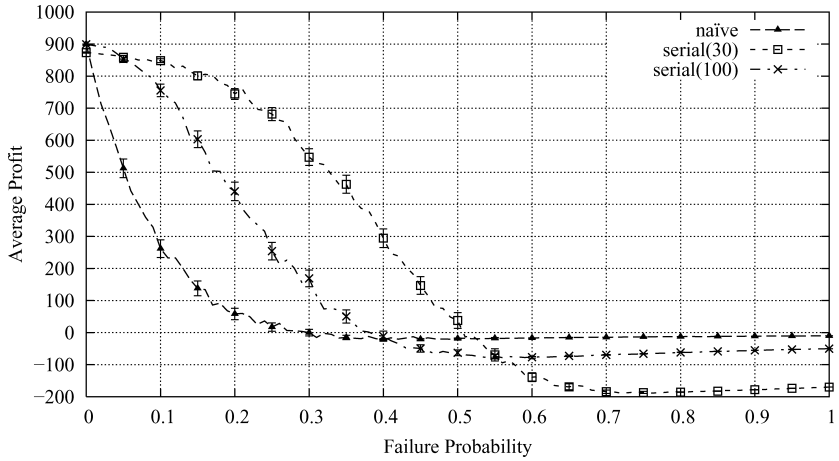


Fig. 11. Effect of different amounts of waiting times for serial provisioning (data shown with 95% confidence intervals).

success. Above this, the parallel strategies do not yield better results than the *naïve* strategy as they also begin to fail in most cases.

Summarizing these trends, it is obvious that parallel provisioning yields a considerable improvement over the *naïve* approach in a range of environments. For example, when the failure probability is 0.2, provisioning two services results in an average profit of 497.2 ± 26.6 (with 95% confidence interval), while the *naïve* strategy achieves only 58.2 ± 17.9. This leads us to conclude that the *parallel(n)* strategy can indeed lead to an improvement and, hence, that Hypothesis 1 holds. However, no parallel strategy dominates the other and they all eventually make losses when the probability of failure increases to such an extent that the chosen redundancy levels do not suffice to ensure success. In this context, it is interesting to note the losses of each strategy become smaller again after a certain minimum is passed (e.g., *parallel(6)* reaches a minimum when the failure probability is around 0.8). This is because the strategies fail earlier in the workflow and therefore lose a lower investment. In conclusion, parallel provisioning is sensitive to the right choice of n and might even lead to an overall loss if the wrong parameter is chosen.

5.4 Serial Provisioning (Hypothesis 2)

We carried out a similar experiment to verify the advantage of serial provisioning over the *naïve* strategy (see Figure 11). Here, again, there is a marked improvement over the *naïve* strategy for failure probabilities up to and including 0.5. This improvement is due to the fact that serial provisioning responds to failures as they occur, while only paying for additional services when necessary. However, as the failure probability rises, this strategy begins to miss its deadlines and hence incurs increasingly large losses.

Overall, a significant improvement in the average profit for some environments leads us to conclude that Hypothesis 2 holds. Again, the strategy is

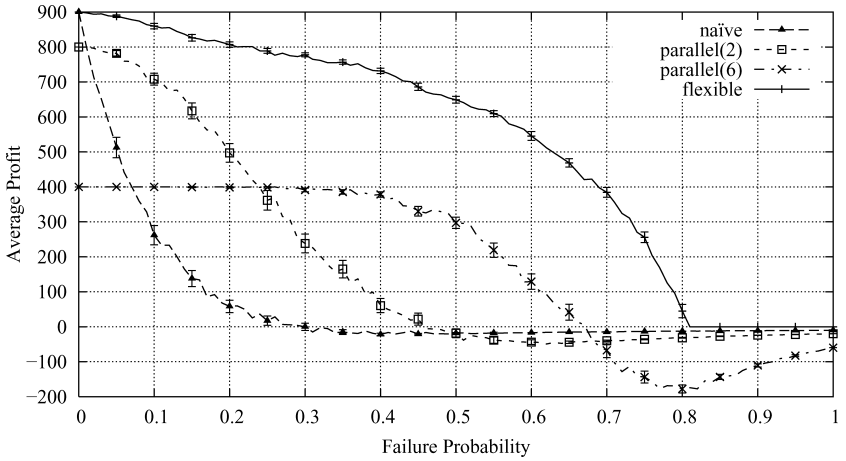


Fig. 12. Average profit of flexible strategy (data shown with 95% confidence intervals).

sensitive to the choice of parameter w, but this time, *serial(30)* dominates *serial(100)* when there is uncertainty, until both make a loss.

5.5 Flexible Provisioning (Hypotheses 3 and 4)

To show how the *flexible* strategy compares against the *naïve* provisioning approach and our nonflexible strategies, Figure 12 plots the average profit of various strategies against the service failure probabilities. Here, it is clear that the flexible approach performs better than any of the other strategies. This is due, in part, to the flexibility of the strategy that allows it to provision more services for later parts of the workflow, where success becomes more critical as a higher investment has already been made. The flexible approach also combines the benefits of the other strategies, allowing the agent to choose between parallel (e.g., when there is little time) and serial provisioning (e.g., when the agent can afford the extra waiting time) or a mixture of the two. Although performance degrades as services become more failure-prone, flexible provisioning retains a relatively high average profit when all other strategies start to make a loss. Furthermore, the strategy avoids making an overall loss due to its prediction mechanism, which ignores a workflow when it seems infeasible.

In Figure 13, we plot the success probability of each strategy against the service failure probabilities. While maximizing the workflow success probability was not the primary aim of devising the *flexible* strategy, the results show that the strategy performs very well over a range of environments. More specifically, it initially completes almost all workflows successfully, and maintains this trend up to a failure probability of 0.8, by which all other approaches have large failure rates. When this failure probability is exceeded, the strategy suddenly begins to ignore all workflows, because it cannot find a feasible allocation to offer a positive return. While the *parallel(6)* strategy still succeeds in a small fraction of workflows, it is incurring significant losses, as explained in the previous section.

From these results, it is clear that hypotheses 3 and 4 hold. While there are some cases where other strategies achieve similar results (e.g., when services

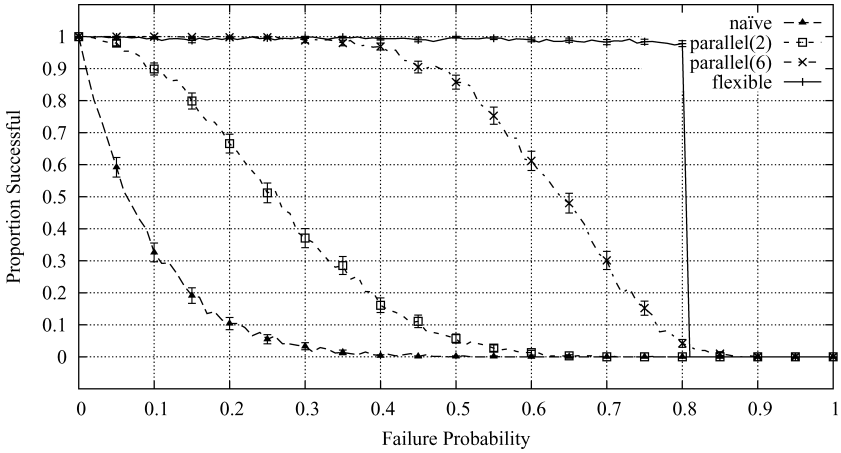


Fig. 13. Success probability of flexible strategy (data shown with 95% confidence intervals).

Table II. Summary of Results with 95% Confidence Intervals

Strategy	Average Profit u_c	Profit vs Naïve	Success Rate p_s
naïve	65.16 ± 1.68	1	0.095 ± 0.002
serial(100)	142.47 ± 2.46	2.19 ± 0.07	0.258 ± 0.003
parallel(2)	177.98 ± 2.37	2.73 ± 0.08	0.272 ± 0.003
parallel(6)	180.06 ± 1.86	2.76 ± 0.08	0.626 ± 0.003
serial(30)	217.12 ± 3.06	3.33 ± 0.10	0.439 ± 0.003
flexible	523.90 ± 2.20	8.04 ± 0.21	0.795 ± 0.003

never fail), the *flexible* strategy achieves consistently good results, and, averaged over all results discussed in Sections 5.3–5.5, dominates all other strategies. This is summarized in Table II, which contains the performance statistics of our representative strategies, averaged over all environments that we tested so far (using the same data as in Figures 10–13). These results highlight the benefits of our strategies, and show that the *flexible* strategy by far outperforms the *naïve* approach. In particular, we achieve an improvement of approximately 700% in average profit and successfully complete around 80% of all workflows. To show that these results also hold in other scenarios, in the next section, we consider a more complex case than the workflows discussed so far.

5.6 Performance in Complex Environments (Hypotheses 3 and 4)

In the previous section, we examined the performance of our strategies in the context of a small, sequential workflow with only one type of service. As mentioned above, this allowed us to verify some results analytically. In this section, we briefly present the results of a more complex problem, and, in doing so, demonstrate that the same overall trends can be observed.

For this experiment, we created random workflows that consist of 50 tasks and have a parallelism parameter of 0.25 (an example is given in Figure 14). We also chose a random service type for each task from a set of seven types that are detailed in Table III. These service types were chosen to display a variety of parameters. For example, T_1 is extremely fast and will almost

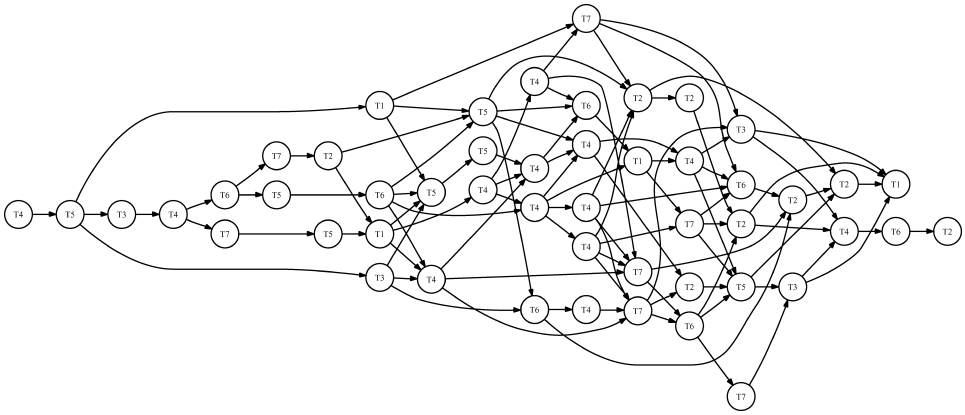


Fig. 14. An example workflow consisting of 50 tasks.

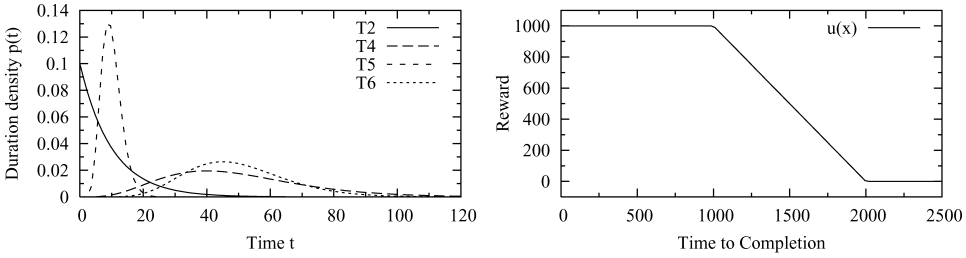
Table III. Service Types Used to Test Complex Workflows

Service	Cost ($)	Duration	Mean **(min.)**	Var.
T_1	0.1	Gamma(1,0.1)	0.1	0.01
T_2	0.1	Gamma(1,10)	10	100
T_3	1	Gamma(5,1)	5	5
T_4	1	Gamma(5,10)	50	500
T_5	2	Gamma(10,1)	10	10
T_6	2	Gamma(10,5)	50	250
T_7	2	Gamma(100,0.1)	10	1

certainly complete by the next time step following its invocation, while, at the other end of the scale, T_4 and T_6 both have a mean duration of 50 time units (Figure 15(a) shows the duration functions for some of the services). Services of type T_1 are also very cheap (0.1 units), while those of T_7 cost 20 times as much.

Furthermore, we assumed that there were 100 instances of each service type, and we used a utility function with a deadline of 1,000 time units, a penalty of 1 per time unit and a maximum utility of 1,000 (this is shown in Figure 15(b)). Again, we tested our strategies in environments where services have different failure probabilities (0,0.01,0.02, . . . ,1), but this time we included some variance in the failure probabilities of different service types. Specifically, during each experimental run for a particular average failure probability f, we assigned a failure probability to each service type that was drawn from a beta distribution[16] with parameters $\alpha = f \cdot 10$ and $\beta = 10 - \alpha$ (unless $f = 0$ or $f = 1$, in which case all services had the same failure probability). This process, which was repeated for all 1,000 runs for each value of f, meant that the average failure probability of all service types would approach f, but still allowed considerable variance between the different types of services.

[16]The beta distribution was simply chosen because it always ranges between 0 and 1.



(a) Example service durations (b) Utility function

Fig. 15. Experimental settings: (a) shows some service duration functions and (b) gives the utility function we use.

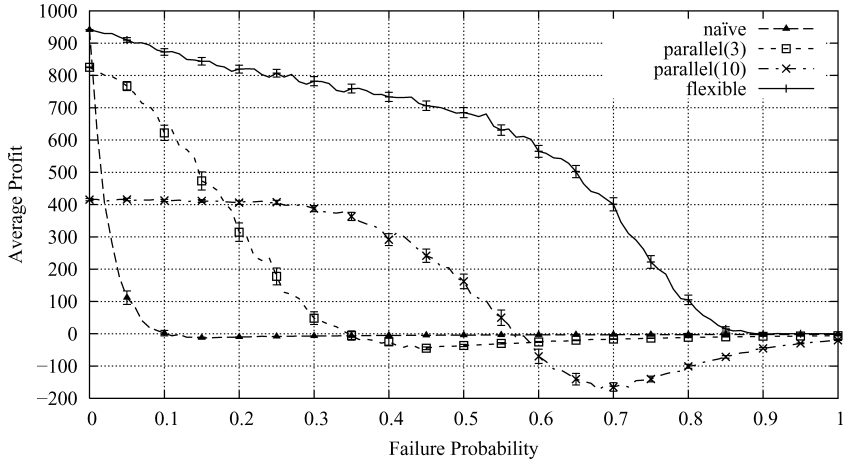


Fig. 16. Average profit for various strategies when faced with complex workflows (data shown with 95% confidence intervals).

With these experimental settings,[17] we again tested the *flexible* strategy against several other approaches (see Figure 16). Here, a similar pattern as shown in Figure 12 emerges and our *flexible* approach clearly dominates the other approaches when service success is uncertain (i.e., when the failure probability is greater than 0). When no services fail (failure probability is 0), the flexible strategy does as well as the *naïve* approach and better than any of the others.

To complete the summary of this experiment, Figure 17 shows the success probabilities of the strategies we tested. Again, the flexible strategy performs very well compared to the other approaches. Although it is initially slightly lower than *parallel(10)*, it stays at a high level and only starts to drop below 90% when the failure probability rises to 70%. Overall, the results presented in this section further highlight the promise of flexible provisioning techniques and show that our strategy is applicable to large workflows with heterogeneous

[17]These parameters were chosen to exemplify the performance of the strategy. We have experimented with other values and observed the same broad trends.

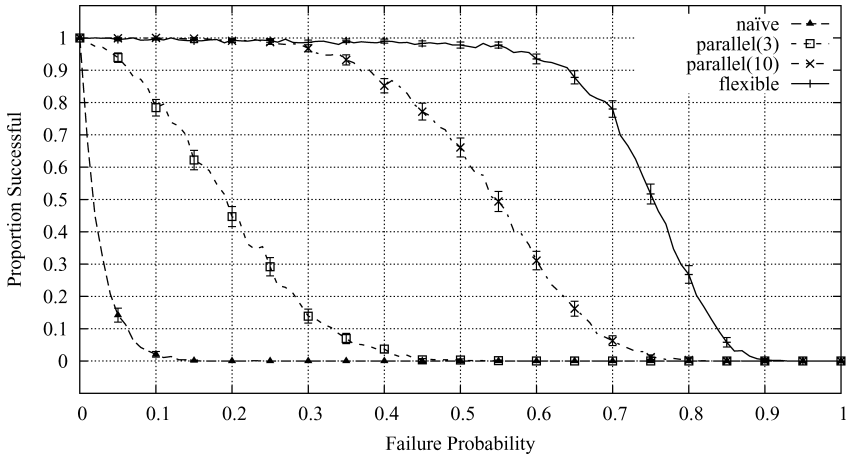


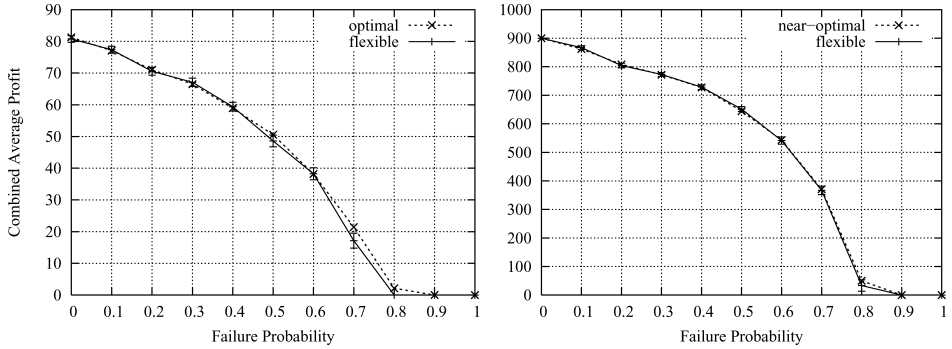
Fig. 17. Success probabilities for various strategies when faced with complex workflows (data shown with 95% confidence intervals).

service types and parallel workflow tasks. In particular, the results confirm that our hypotheses 3 and 4 hold in these environments, as the same trends as in the previous section are observed.

5.7 Optimality of Flexible Provisioning

As discussed previously, the *flexible* strategy uses a heuristic utility function and a hill-climbing mechanism that is not optimal in general. However, adopting this heuristic method has made the provisioning of complex workflows tractable. In this section, we compare the performance of our algorithm to the theoretical optimal. More specifically, we first show our results in a simple environment (we consider a workflow with three sequential tasks, each of which has a cost of 3, duration distribution Gamma(2,4), 20 providers, and a utility function with deadline 30, maximum utility 100 and penalty 10). This scenario allows us to solve our original optimization problem (as given by Eq. (6)) analytically. This is then followed by an analysis of the environment used in Sections 5.3–5.5. Because deriving the optimal solution is intractable in this case, we designed a new *analytical flexible* strategy. This is based on our *flexible* strategy, but accurately calculates the expected utility, rather than relying on a heuristic function. It then repeatedly performs a hill-climbing search with random restarts (we restart the algorithm 200 times with random initial allocations). We believe that this is a reasonable approximation to the optimal, and, in fact, there is no significant difference between its performance and the theoretical optimal in the smaller environment.

Figure 18 shows the average profit of our strategy in these two environments (here, failure probabilities were varied in steps of 0.1 due to the computational cost of calculating an optimal solution). In both cases, while clearly suboptimal, our strategy comes close to the expected utility of the optimal or near-optimal strategies. In fact, when averaging over the failure probabilities we examined, for 3-task workflows (Figure 18(a)), our *flexible* strategy achieves an average



(a) Workflow with 3 Tasks (b) Workflow with 10 Tasks

Fig. 18. Average profit of *flexible* strategy (with 95% confidence intervals), compared to the optimal strategy for 3 tasks in (a) and to the near-optimal strategy for 10 tasks in (b).

utility of 41.7 ± 0.7, compared to the optimal expected utility of 42.5, which corresponds to achieving $98.2 \pm 1.7\%$ of the optimal. For 10-task workflows (Figure 18(b)), we achieve even closer results with an average utility of 512.0 ± 7.0 compared to the near-optimal expected utility of 516.1. In fact, a t-test confirms that this is not a statistically significant difference ($p = 0.764$). This improvement, compared to the smaller workflows, may be due to our reliance on the central limit theorem to estimate the duration distribution. When the workflows become larger, this tends to give more accurate estimates. Overall, these results are promising, because they show that our strategy achieves a level of performance that is close to the optimal in the environments we tested, using a fast heuristic method that is tractable even for large workflows.

6. CONCLUSIONS

In this section we conclude by first summarising the contribution of this article (Section 6.1) and then outlining our future work (Section 6.2).

6.1 Summary

In this article, we highlighted the inherent unreliability and uncertainty of computer services in open, distributed systems. This is becoming a particularly pertinent problem as Semantic Web technologies enable previously unseen services to be discovered and provisioned dynamically at run-time. However, much work in this area has focussed only on functional service descriptions and therefore does not consider the possibility that providers may regularly fail to honour these descriptions or take uncertain amounts of time to execute their services. To address these shortcomings, we developed a heuristic provisioning strategy that allocates multiple services for unreliable tasks to reduce the probability of failure. Moreover, this strategy flexibly varies the number of provisioned services for each task in order to maximize the service consumer's expected utility. In experiments, we showed that this strategy performs well in a variety of environments, and that it consistently outperforms approaches that consider services to be reliable or that rely on simple, nonflexible redundancy.

By focussing on the provisioning of abstract workflows, our work builds on and extends existing research in the area of Semantic Web services. Specifically, we view service provisioning as an additional, intermediate stage between the semantic matching of functional service capabilities and the invocation of service instances. Frameworks such as OWL-S 1.1 [Martin et al. 2004] support both the notion of functional service descriptions, characterising the service in terms of its interface (i.e., *Input*, *Output*, *Condition* and *Result*), and nonfunctional parameters that represent QoS properties for use in service selection and provisioning tasks. To support the QoS metrics assumed by our method within the OWL-S Profile, a set of additional definitions have been specified, such as *provision:SuccessProbability* and *provision:InvocationCost*, that extend the *profile:ServiceCategory* class [Stein et al. 2006]. The DatatypeProperty ranges supported by these new classes contain real numbers, thus facilitating the quantitative reasoning used by the different strategies described in Section 4. An additional ObjectProperty, *provision:sParameterSource*, has been defined to identify where these parameters were generated (e.g., the service provider or another, trusted, third party service, such as the CONOISE quality agent [Norman et al. 2004]).

A key advantage of the method presented in this article is that it does not rely on a particular service framework, description language or matchmaking technique. Rather, we have focussed on providing a principled, abstract approach for provisioning services that can be adapted by agent developers and system designers to deal with uncertainty in their specific applications. In particular, we believe that our flexible strategy can easily be integrated into to a large number of existing systems for invoking abstract workflows in service-oriented systems (e.g., McIlraith and Son [2002] and Friese et al. [2005]). In this context, we envisage our work to be particularly important in scenarios where workflows are of tangible value to the consumer, where some costs are associated with service invocation, and where there are time constraints for workflow completion. Common application domains where these issues arise include automated business process management, high-performance utility computing, peer-to-peer systems and scientific Grids.

However, by devising a general model, we could not cover the multitude of domain-specific constraints that may arise in particular application scenarios, and we have made a number of simplifying assumptions that do not always hold in all possible environments. Some of these will be addressed in future work, as outlined in the next section.

6.2 Future Work

In Section 3.5, we listed a number of assumptions underlying our work. In the following, we describe how we plan to relax them in future work:

(1) *Failure Model.* It is easy to extend our model to include explicit failure messages, for example, by including a new mode of failure, where the provider notifies the consumer of its failure after invocation. This would generally reduce the expected duration of tasks as the consumer does not necessarily need to wait for the specified time-out (Eq. (20)), but would not alter our overall strategy.

Considering Byzantine failures is more challenging, but our strategy forms a solid basis for tackling this problem. As we already rely on redundancy, it is straightforward to include voting schemes that select the majority of several different service outcomes. Dealing with correlated failures also poses new challenges, but there are a number of existing techniques for modelling and learning such correlations, and for avoiding services that are prone to correlated failures [Nicola and Goyal 1990; Weatherspoon et al. 2002; Townend et al. 2005].

(2) *Performance Information.* Although we assume accurate performance information to be available, we have conducted a separate evaluation using inaccurate information and have shown that our strategy is robust to small to moderate inaccuracies in most environments [Stein et al. 2007b]. Also, while we concentrate on scenarios with limited information about service populations in this article, we have extended our model for heterogeneous environments in other work [Stein et al. 2007a].

(3) *Payment Model.* Again, it is easy to modify our model to include either refunds of failed services (with a certain probability) or extra charges that might be incurred for the disposal of additional successful service invocations. Both of these only require small modification to Eq. (14).

(4) *Reward Model.* Our utility function can easily be extended to cover more complex cases, especially as we use a generic hill-climbing algorithm to optimize the overall provisioning allocation.

(5) *Model Scope.* In future work, we will cover more extensive workflow models that may occur in practice, and which will require small modifications to the way we aggregate performance parameters over the workflow. We envisage that a large number of other domain-specific requirements can be easily incorporated into our approach by placing appropriate constraints on the hill-climbing algorithm. For example, when it is impossible to provision multiple services in parallel for a particular task, the corresponding parameter n can be held constant at 1. Similarly, when there are close dependencies between several services offered by a single provider, these can be aggregated and viewed at a higher level of abstraction as a single unit (e.g., a book vendor's *submitOrder* and *payOrder* services might be aggregated, as they only produce the desired result of ordering a book when used in conjunction).

In addition to addressing the above issues, we are currently investigating more complex pricing models where consumers and providers participate in dynamic electronic service markets with constantly changing availability of services [Jennings et al. 2001; Buyya et al. 2005]. In this work, we are particularly interested in devising a strategy that automatically decides when to provision particularly critical or scarce services far in advance, and when to leave sufficient flexibility to deal with unexpected failures or delays by provisioning services on demand.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their comprehensive comments on early drafts of this article.

REFERENCES

AGGARWAL, R., VERMA, K., MILLER, J., AND MILNOR, W. 2004. Constraint driven web service composition in METEOR-S. In *Proceedings of the IEEE International Conference on Services Computing 2004 (SCC 2004)*. IEEE Computer Society Press, Los Alamitos, CA, 23–30.

AGHDAIE, N. AND TAMIR, Y. 2003. Fast transparent failover for reliable web service. In *Proceedings of the 15th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS)*. 757–762.

AKKIRAJU, R., VERMA, K., GOODWIN, R., DOSHI, P., AND LEE, J. 2004. Executing abstract web process flows. In *Proceedings of the ICAPS Workshop on Planning and Scheduling for Web and Grid Services*. 9–15.

ANDERSON, D. P., COBB, J., KORPELA, E., LEBOFSKY, M., AND WERTHIMER, D. 2002. SETI@home: An experiment in public-resource computing. *Comm. ACM 45,* 11, 56–61.

ARUNACHALAM, R. AND SADEH, N. 2004. The 2003 supply chain management trading agent competition. In *Proceedings of the 6th International Conference on Electronic Commerce (ICEC '04)*. 113–120.

BACCELLI, F., JEAN-MARIE, A., AND LIU, Z. 1993. A survey on solution methods for task graph models. In *Arbeitsberichte der IMMD*, N. Götz, U. Herzog, and M. Rettelbach, Eds. Vol. 26 (14). Universität Erlangen-Nürnberg, Erlangen, Chapter Second QMIPS Workshop, 163–183.

BENATALLAH, B., HACID, M.-S., LEGER, A., REY, C., AND TOUMANI, F. 2005. On automating web services discovery. *VLDB J. 14,* 1, 84–96.

BOLOT, J.-C. 1993. End-to-end packet delay and loss behavior in the internet. In *Proceedings of the ACM SIGCOMM '93 Conference on Communications Architectures, Protocols and Applications*. ACM, New York, 289–298.

BUYYA, R., ABRAMSON, D., AND VENUGOPAL, S. 2005. The grid economy. *Proc. IEEE 93,* 3, 698–714.

BYLANDER, T. 1994. The computational complexity of propositional STRIPS planning. *Artif. Intell. 69,* 1-2, 165–204.

CANFORA, G., PENTA, M. D., ESPOSITO, R., AND VILLANI, M. L. 2005. QoS-aware replanning of composite web services. In *Proceedings of the IEEE International Conference on Web Services (ICWS'05)*. IEEE Computer Society Press, Los Alamitos, CA. 121–129.

CASATI, F., CERI, S., PARABOSCHI, S., AND POZZI, G. 1999. Specification and implementation of exceptions in workflow management systems. *ACM Trans. Database Syst. 24,* 3, 405–451.

COHEN, P. R. 1995. *Empirical methods for artificial intelligence*. MIT Press, Cambridge, MA.

COLLINS, J., BILOT, C., GINI, M., AND MOBASHER, B. 2001. Decision processes in agent-based automated contracting. *IEEE Internet Comput. 5,* 2, 61–72.

CRISTIAN, F. 1991. Understanding fault-tolerant distributed systems. *Comm. ACM 34,* 2, 56–78.

CURBERA, F., DUFTLER, M., KHALAF, R., NAGY, W., MUKHI, N., AND WEERAWARANA, S. 2002. Unraveling the web services web: An introduction to SOAP, WSDL, and UDDI. *IEEE Internet Comput. 6,* 2, 86–93.

CURBERA, F., KHALAF, R., MUKHI, N., TAI, S., AND WEERAWARANA, S. 2003. The next step in web services. *Comm. ACM 46,* 10, 29–34.

DAN, A., DAVIS, D., KEARNEY, R., KING, R., KELLER, A., KUEBLER, D., LUDWIG, H., POLAN, M., SPREITZER, M., AND YOUSSEF, A. 2004. Web services on demand: WSLA-driven automated management. *IBM Syst. J. 43,* 1, 136–158.

DEELMAN, E., BLYTHE, J., GIL, Y., KESSELMAN, C., MEHTA, G., VAHI, K., BLACKBURN, K., LAZZARINI, A., ARBREE, A., CAVANAUGH, R., AND KORANDA, S. 2003. Mapping abstract complex workflows onto grid environments. *J. Grid Comput. 1,* 1, 25–39.

DEGROOT, M. H. AND SHERVISH, M. J. 2002. *Probability and Statistics*, 3rd ed. Addison-Wesley, Reading, MA.

DODIN, B. 1985. Bounding the project completion time distribution in PERT networks. *Oper. Res. 33,* 4, 862–881.

EDER, J. AND LIEBHART, W. 1995. The workflow activity model WAMO. In *Proceedings of the 3rd International Conference on Cooperative Information Systems*. 87–98.

ERRADI, A., MAHESHWARI, P., AND TOSIC, V. 2006. Recovery policies for enhancing web services reliability. In *Proceedings of the IEEE International Conference on Web Services (ICWS'06)*. IEEE Computer Society Press, Los Alamitos, CA. 189–196.

EWING, B., HILLIER, L., WENDL, M. C., AND GREEN, P. 1998. Base-calling of automated sequencer traces using phred. I. Accuracy assessment. *Genome Res. 8,* 3, 175–185.

FOSTER, I., KESSELMAN, C., AND TUECKE, S. 2001. The anatomy of the grid: Enabling scalable virtual organizations. *Int. J. High Perform. C. 15,* 3, 200–222.

FRIESE, T., MÜLLER, J. P., AND FREISLEBEN, B. 2005. Self-healing execution of business processes based on a peer-to-peer service architecture. In *Proceedings of the 18th International Conference on Architecture of Computing Systems (ARCS '05), System Aspects in Organic and Pervasive Computing*. Lecture Notes in Computer Science, vol. 3432. Springer-Verlag, Berlin, Germany. 108–123.

GARCIA-MOLINA, H. AND SALEM, K. 1987. Sagas. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data (SIGMOD '87)*. ACM, New York. 249–259.

GÄRTNER, F. C. 1999. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Comput. Surv. 31,* 1, 1–26.

GEORGAKOPOULOS, D., HORNICK, M. F., AND SHETH, A. P. 1995. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distrib. Parallel Dat. 3,* 2, 119–153.

INGHAM, D. B., PANZIERI, F., AND SHRIVASTAVA, S. K. 1999. Constructing dependable web services. In *Advances in Distributed Systems, Advanced Distributed Computing: From Algorithms to Systems*. Springer-Verlag, Berlin, Germany, 277–294.

IRWIN, D. E., GRIT, L. E., AND CHASE, J. S. 2004. Balancing risk and reward in a market-based task service. In *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC-13 '04)*. IEEE Computer Society Press, Los Alamitos, CA. 160–169.

JAEGER, M. C. AND LADNER, H. 2005. Improving the QoS of WS compositions based on redundant services. In *Proceedings of the International Conference on Next Generation Web Services Practices (NWeSP 2005)*. 189–194.

JAEGER, M. C. AND MÜHL, G. 2007. QoS-based selection of services: The implementation of a genetic algorithm. In *Proceedings of the KiVS 2007 Workshop: Service-Oriented Architectures und Service-Oriented Computing (SOA/SOC)*. 359–370.

JENNINGS, N. R. 2001. An agent-based approach for building complex software systems. *Comm. ACM 44,* 4, 35–41.

JENNINGS, N. R., FARATIN, P., LOMUSCIO, A. R., PARSONS, S., SIERRA, C., AND WOOLDRIDGE, M. 2001. Automated negotiation: Prospects, methods and challenges. *Group Decis. Negot. 10,* 2, 199–215.

JENNINGS, N. R., FARATIN, P., NORMAN, T. J., O'BRIEN, P., AND ODGERS, B. 2000. Autonomous agents for business process management. *Appl. Artif. Intell. 14,* 2, 145–189.

KLUSCH, M., GERBER, A., AND SCHMIDT, M. 2005. Semantic web service composition planning with OWLS-XPlan. In *Proceedings of the 1st Int. AAAI Fall Symposium on Agents and the Semantic Web*. 55–62.

KOCHUT, K., ARNOLD, J., SHETH, A., MILLER, J., KRAEMER, E., ARPINAR, B., AND CARDOSO, J. 2003. IntelliGEN: A distributed workflow system for discovering protein-protein interactions. *Distrib. Parallel Dat. 13,* 1, 43–72.

LI, W., HE, J., MA, Q., YEN, I.-L., BASTANI, F., AND PAUL, R. 2005. A framework to support survivable web services. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*. IEEE Computer Society Press, Los Alamitos, CA. 93.2.

LONG, D. D. E., MUIR, A., AND GOLDING, R. A. 1995. A longitudinal survey of internet host reliability. In *Proceedings of the 14th Symposium on Reliable Distributed Systems (SRDS'95)*. 2–9.

MALCOLM, D. G., ROSEBOOM, J. H., CLARK, C. E., AND FAZAR, W. 1959. Application of a technique for research and development program evaluation. *Oper. Res. 7,* 5, 646–669.

MANDELL, D. AND MCILRAITH, S. 2003. Adapting BPEL4WS for the semantic web: The bottom-up approach to web service interoperation. In *Proceedings of the 2nd International Semantic Web Conference*. Lecture Notes in Computer Science, vol. 2870, Springer-Verlag, Berlin, Germany. 227–241.

MARTIN, D., PAOLUCCI, M., MCILRAITH, S., BURSTEIN, M., MCDERMOTT, D., MCGUINNESS, D., PARSIA, B., PAYNE, T., SABOU, M., SOLANKI, M., SRINIVASAN, N., AND SYCARA, K. 2004. Bringing semantics to web services: The OWL-S approach. In *Proceedings of the 1st International Workshop on Semantic*

Web Services and Web Process Composition (SWSWPC 2004). Lecture Notes in Computer Science, vol. 3387, Springer-Verlag, Berlin, Germany. 26–42.

MAXIMILIEN, E. M. AND SINGH, M. P. 2004a. A framework and ontology for dynamic web services selection. *IEEE Internet Comput. 8,* 5, 84–93.

MAXIMILIEN, E. M. AND SINGH, M. P. 2004b. Toward autonomic web services trust and selection. In *Proceedings of the 2nd International Conference on Service-Oriented Computing (ICSOC '04)*. 212–221.

MCDERMOTT, D. 2002. Estimated-regression planning for interactions with web services. In *Proceedings of the 6th International Conference on AI Planning and Scheduling (AIPS'02)*. 204–211.

MCGUINNESS, D. AND VAN HARMELEN, F. 2004. OWL web ontology language overview. Recommendation, W3C. February. (`http://www.w3.org/TR/2004/REC-owl-features-20040210/`).

MCILRAITH, S. A. AND SON, T. C. 2002. Adapting golog for composition of semantic web services. In *Proceedings of the 8th International Conference on Knowledge Representation and Reasoning (KR2002)*. 482–493.

MCILRAITH, S. A., SON, T. C., AND ZENG, H. 2001. Semantic web services. *IEEE Intel. Syst. 16,* 2, 46–53.

MEDJAHED, B., BENATALLAH, B., BOUGUETTAYA, A., NGU, A. H. H., AND ELMAGARMID, A. K. 2003. Business-to-business interactions: issues and enabling technologies. *VLDB J. 12,* 1, 59–85.

MENASCE, D. 2002. QoS issues in web services. *IEEE Internet Comput. 6,* 6, 72–75.

MERIDETH, M. G., IYENGAR, A., MIKALSEN, T., TAI, S., ROUVELLOU, I., AND NARASIMHAN, P. 2005. Thema: Byzantine-fault-tolerant middleware for web-service applications. In *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*. IEEE Computer Society press, Los Alamitos, CA. 131–142.

MICHALEWICZ, Z. AND FOGEL, D. B. 2004. *How to solve it: Modern Heuristics*, 2nd ed. Springer-Verlag, Berlin, Germany.

MILANOVIC, N. AND MALEK, M. 2004. Current solutions for web service composition. *IEEE Internet Comput. 8,* 6, 51–59.

NG, K.-C. AND ABRAMSON, B. 1990. Uncertainty management in expert systems. *IEEE Expert 5,* 2, 29–48.

NICOLA, V. F. AND GOYAL, A. 1990. Modeling of correlated failures and community error recovery in multiversion software. *IEEE T. Software Eng. 16,* 3, 350–359.

NORMAN, T. J., PREECE, A., CHALMERS, S., JENNINGS, N. R., LUCK, M., DANG, V. D., NGUYEN, T. D., DEORA, V., SHAO, J., GRAY, A. W., AND FIDDIAN, N. J. 2004. Agent-based formation of virtual organisations. *Knowl.-Based Syst. 17,* 2–4, 103–111.

O'BRIEN, A., NEWHOUSE, S., AND DARLINGTON, J. 2004. Mapping of scientific workflow within the e-protein project to distributed resources. In *Proceedings of the UK E-Science All Hands Meeting (AHM 2004)*. 404–409.

OINN, T., GREENWOOD, M., ADDIS, M., ALPDEMIR, M. N., FERRIS, J., GLOVER, K., GOBLE, C., GODERIS, A., HULL, D., MARVIN, D., LI, P., LORD, P., POCOCK, M. R., SENGER, M., STEVENS, R., WIPAT, A., AND WROE, C. 2006. Taverna: Lessons in creating a workflow environment for the life sciences. *Concurrency and Computation: Practice and Experience 18,* 10, 1067–1100.

PAOLUCCI, M., KAWAMURA, T., PAYNE, T. R., AND SYCARA, K. P. 2002. Semantic matching of web services capabilities. In *Proceedings of the 1st International Semantic Web Conference (ISWC 2002)*. Lecture Notes in Computer Science, vol. 2342, Springer-Verlag, New York. 333–347.

PAOLUCCI, M. AND SYCARA, K. 2003. Autonomous semantic web services. *IEEE Internet Comput. 7,* 5, 34–41.

RAIFFA, H. 1968. *Decision Analysis: Introductory Lectures on Choices Under Uncertainty*. McGraw-Hill, Englewood Cliffs, NJ.

RAN, S. 2003. A model for web services discovery with QoS. *SIGecom Exch. 4,* 1, 1–10.

RUSSELL, S. AND NORVIG, P. 2003. *Artificial Intelligence: A Modern Approach*, 2nd ed. Prentice-Hall, Englewood Cliffs, NJ.

SCHROEDER, B., AND GIBSON, G. A. 2006. A large-scale study of failures in high-performance computing systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN2006)*. 249–258.

SINGH, M. P. AND HUHNS, M. N. 2005. *Service-Oriented Computing: Semantics, Processes, Agents*. Wiley, New York.

SIRIN, E., PARSIA, B., AND HENDLER, J. 2005. Template-based composition of semantic web services. In *Proceedings of the AAAI Fall Symposium on Agents and the Semantic Web*. 85–92.

SMITH, T. M., ABAJIAN, C., AND HOOD, L. 1997. Hopper: Software for automating data tracking and flow in DNA sequencing. *Comput. Appl. Biosci. 13,* 2, 175–182.

STEIN, S., JENNINGS, N. R., AND PAYNE, T. R. 2007a. Provisioning heterogeneous and unreliable providers for service workflows. In *Proceedings of the 22nd AAAI Conference on Artificial Intelligence*. 1452–1458.

STEIN, S., PAYNE, T. R., AND JENNINGS, N. R. 2006. Flexible provisioning of semantic web service workflows using a QoS ontology. In *Proceedings of the 5th International Semantic Web Conference (ISWC 2006), Online supplement*. (available at `http://eprints.ecs.soton.ac.uk/12992/`).

STEIN, S., PAYNE, T. R., AND JENNINGS, N. R. 2007b. An effective strategy for the flexible provisioning of service workflows. In *Proceedings of the Workshop on Service-Oriented Computing: Agents, Semantics, and Engineering (SOCASE 2007)*. Lecture Notes in Computer Science, vol. 4504. Springer-Verlag, Berlin, Germany, 16–30.

SZOMSZOR, M., PAYNE, T. R., AND MOREAU, L. 2005. Using semantic web technology to automate data integration in grid and web service architectures. In *Proceedings of the Semantic Infrastructure for Grid Computing Applications Workshop in Cluster Computing and Grid (CCGrid)*. 189–195.

TEACY, W. T. L., PATEL, J., JENNINGS, N. R., AND LUCK, M. 2006. TRAVOS: Trust and reputation in the context of inaccurate information sources. *J. Auton. Agents Multi-Agent Syst. 12,* 2, 183–198.

TILLMAN, F. A., AND LIITTSCHWAGER, J. M. 1967. Integer programming formulation of constrained reliability problems. *Manage. Sci. 13,* 11, 887–899.

TOWNEND, P., GROTH, P., AND XU, J. 2005. A provenance-aware weighted fault tolerance scheme for service-based applications. In *Proceedings of the 8th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*. IEEE Computer Society Press, Los Alamitos, CA, 258–266.

TRIVEDI, K. 2001. *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*, 2nd ed. John Wiley & Sons, Inc., USA.

WEATHERSPOON, H., MOSCOVITZ, T., AND KUBIATOWICZ, J. 2002. Introspective failure analysis: Avoiding correlated failures in peer-to-peer systems. In *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems*. 362–367.

WEERAWARANA, S., CURBERA, F., LEYMANN, F., Storey, T. AND FERGUSON, D. F. 2005. *Web Services Platform Architecture*. Prentice-Hall, Englewood Cliffs, NJ.

WEISS, G., Ed. 1999. *Multiagent systems: A modern approach to distributed artificial intelligence*. MIT Press, Cambridge, MA.

YANG, Z. AND DUDDY, K. 1996. CORBA: A platform for distributed object computing. *ACM Oper. Syst. Rev. 30,* 2, 4–31.

YU, T. AND LIN, K.-J. 2005. Adaptive algorithms for finding replacement services in autonomic distributed business processes. In *Proceedings of Autonomous Decentralized Systems (ISADS 2005)*. 427–434.

YU, T., ZHANG, Y., AND LIN, K.-J. 2007. Efficient algorithms for web services selection with end-to-end QoS constraints. *ACM Trans. Web 1,* 1, 6.

ZENG, L., BENATALLAH, B., DUMAS, M., KALAGNANAM, J., AND SHENG, Q. Z. 2003. Quality driven web services composition. In *Proceedings of the 12th International World Wide Web Conf. (WWW '03)*. 411–421.

ZHOU, C., CHIA, L.-T., AND LEE, B.-S. 2004. DAML-QoS ontology for web services. In *Proceedings of the IEEE International Conference on Web Services (ICWS 2004)*. IEEE Computer Society Press, Los Alamitos, CA, 472–479.

Received December 2006; accepted April 2007