JOpera: Autonomic Service Orchestration

Cesare Pautasso, Thomas Heinis, Gustavo Alonso
Department of Computer Science
ETH Zurich
8092 Zurich, Switzerland
{pautasso, heinist, alonso}@inf.ethz.ch

1 Introduction

The increasing interest in new software engineering technologies for application integration such as Service Oriented Computing and Service Orchestration has resulted in a proliferation of workflow management systems as the underlying representation and execution platform for service composition [7]. Workflow management system are also being applied to new domains (e.g., virtual scientific laboratories [1], Grid computing [12], service delivery and provisioning [6]). For these new applications, workflows are seen as the modeling metaphor behind the notion of *straight through processing* and *virtual organizations* where a collection of existing heterogeneous systems are composed into an integrated solution.

In all these settings workflow engines are at the core of a complex combination of applications and clustered computers. As such, they have become rather difficult to deploy and configure, let alone tune to obtain maximum performance. This problem is not unique to workflow and service composition engines but it is more difficult to address in these settings because there is only a limited understanding of the execution procedures behind a workflow engine. In this short paper we report on our ongoing work to design and develop an autonomic workflow engine that can be used for large scale service composition. The challenge we face in doing this is threefold. First, we need to design an execution procedure for service compositions that is amenable to autonomic treatment. Second, this procedure needs to be realized in an architecture that supports the deployment of different modules of the system across a computer cluster in order to achieve the desired level of performance. Third, an autonomic controller and appropriate control policies need to be developed to automatically provision the optimal amount of resources to the engine.

In what follows we provide a high level description of how we have accomplished these three goals and give a brief account of the performance of the system. The implemented system is part of the JOpera project. JOpera is an advanced SOA tool for Eclipse, which provides modeling, execution, monitoring and debugging tools for workflow-based Web service orchestration. A more detailed presentation of the autonomic capabilities of JOpera, including an extensive experimental evaluation of the approach can be found in [4, 11].

2 Web Service Orchestration with Workflows

In Service Oriented Architectures (SOA) workflow modeling languages have found a good application to define an executable model of the flow of information between a set of services [7]. A workflow process defines the

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

interactions between a set of services by scripting (or orchestrating) the exchange of messages between them. To simplify its integration and reuse, the resulting workflow is also typically published as a service.

As an example, we illustrate a small workflow for providing a value-added service out of the composition of two basic ones. In particular, the workflow shows how a service providing stock prices converted in any currency can be built out of the composition of two services: one returning stock prices in U.S. dollars and the other one returning currency exchange rates between dollars and the requested currency.

The screenshot of Figure 1 shows how the workflow is developed using JOpera. The outline view on the left contains the structure of the workflow in terms of its tasks and also lists the services to be composed. The editors on the right show two graphs defining the control flow and data flow relationships between the tasks of the workflow. The control flow graph (at the top) defines the order of execution of the tasks of the workflow. Since the stock quote and currency exchange services are independent of each other, the tasks invoking them can be executed in parallel. Once both of these tasks complete, the task computing the converted price is executed. The data flow graph of the workflow (shown in the bottom editor) defines where the information required by each service comes from. The result of the entire workflow, to be returned to its client, is produced by the StockQuote service invocation task for the OriginalPrice and by the PriceConversion task for the ConvertePrice. This task receives its input from the result of the invocation of both the StockQuote and the CurrencyExchange service. These are invoked passing data (the Currency and the Symbol identifying the stock) provided by the client as input of the whole workflow.

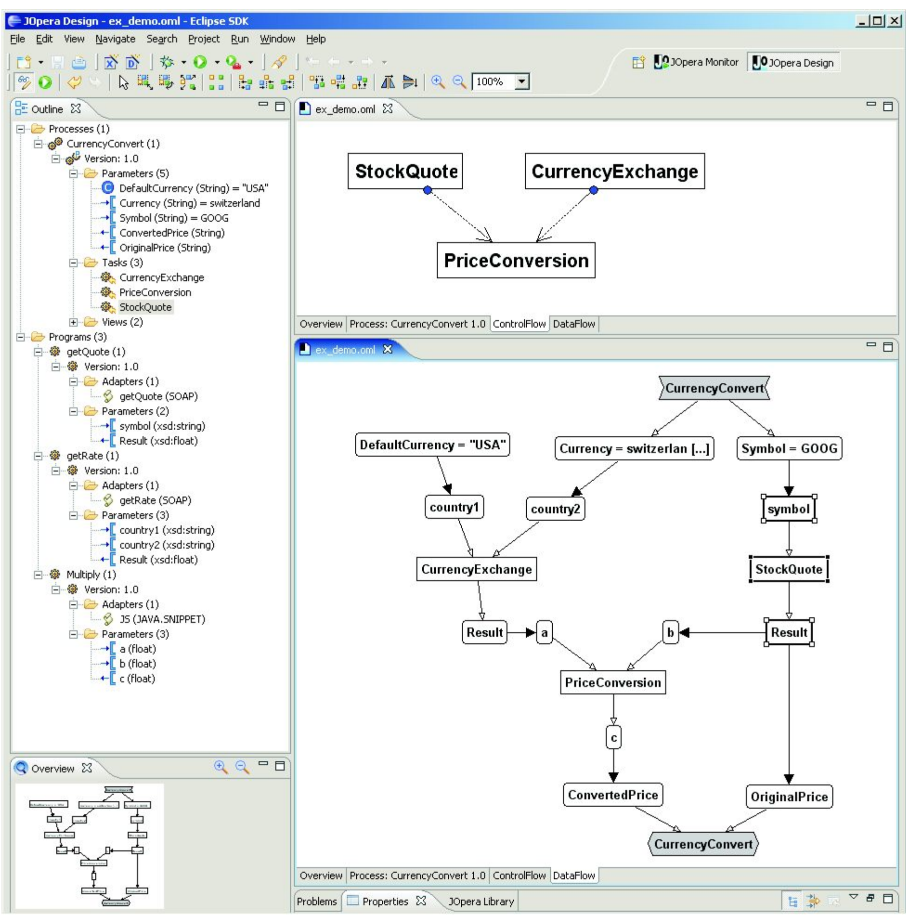

Figure 1: Defining a simple Web service orchestration in JOpera

In addition to the separate visualization of the control flow and data flow aspects of a workflow and the use of control flow extraction algorithms to ensure the automatic reconciliation between the two, JOpera also offers efficient means of binding each task of the workflow to the service to be invoked while executing it [9]. In this example, the tasks devoted to collecting information from the external providers of stock prices and currency rates are bound to a standard-compliant Web service (described using plain WSDL, accessed using SOAP messages). With the price of additional complexity and overhead, this ensures the interoperability between the workflow engine and the service provider and removes the need of developing customized adapters to make the engine access external sources of information. For the third task, responsible for computing the converted price by running a multiplication between the original price and the corresponding exchange rate, it should not be necessary to pay the overhead of a remote SOAP call. In JOpera, the PriceConversion service can be implemented using a so-called Java snippet, which is invoked with the overhead comparable to a local Java method call.

3 Stage-based Workflow Execution in JOpera

Running such a workflow process involves executing the tasks of the workflow in the correct order and passing the data produced by one task to its successors. In the context of Web service orchestration, tasks are typically bound to service invocations and their execution involves the exchange of messages between the engine and an external service provider. Messages are also exchanged in the reverse direction, when clients of the workflow engine want to initiate the execution of a new process instance. Upon receipt of such message, the engine begins running a new process instance, analyzes its control flow structure and determines which tasks need to be executed next. Then, for each active task, the engine selects the service to be invoked, fetches data from the process variables to compose a message, which is then sent to the corresponding service provider. Once this invocation completes, the state of the process needs to be updated with the results so that other tasks can access them. Process execution continues until all tasks have been executed, or an explicit termination point in the workflow has been reached.

Clearly, workflow engines are capable of running more than one instance of a workflow at the same time. This feature is also very important in the context of Web service orchestration: once processes are published as a Web service, clients can send messages to the engine for starting a new process at any time. Given the limited amount of resources (i.e., CPU threads and memory) available to the engine, it becomes important to restructure the execution of a workflow so that the engine can scale to run a large number of concurrent process instances.

In this regard, the simple solution of permanently assigning a thread to run each process instance suffers from a number of limitations. The number of concurrent threads that are available in a virtual machine would set a limit to the number of processes that can be run by the engine at a given time (a few hundred). Furthermore, such threads would be underutilized as they would dedicate most of their time to I/O operations, i.e., sending and receiving SOAP messages. Finally, assigning one thread to each process instance would limit the amount of intra-process parallelism supported by the engine. In other words, even if the control flow structure of a process defines a partial execution order over its tasks, this engine threading model would serialize the execution of all tasks within a process.

One of the innovative design decisions of the JOpera engine lies in employing a threading model which effectively decouples the process instances from the threads executing them. Apart from shifting the factor limiting the maximum number of concurrent processes that can be executed from the number of threads to the amount of available memory, this decision also makes it possible for the same engine architecture to scale out from a centralized to a distributed configuration [10].

To do so, we have partitioned the execution of a process in two *stages*. The first involves the, so-called, process *navigation*, i.e., making the control and data *flow* through the process instance by using a graph traversal algorithm to determine which tasks of the process are to be executed next based on their dependencies to the already completed tasks. The second stage – *dispatching* – involves the actual execution of the tasks, which boils down to the synchronous or asynchronous exchange of messages with the provider of the Web service to which the task has been bound.

In the architecture of JOpera, these two execution stages have been assigned to two different (and loosely coupled) active components of the engine: the navigator and the dispatcher. The navigator runs processes, the dispatcher runs tasks. As it can be seen from Figure 2 they communicate asynchronously using queues. Whenever the navigator has determined that a new task is ready to be executed, the information required to perform such execution is added to the *task queue*. The dispatcher takes tasks from such queue and performs the corresponding Web service invocation. Once the invocation is complete, the dispatcher puts its results in the *event queue*. The navigator collects them, updates the state of the corresponding process instance and continues running it by sending the next tasks to be executed to the dispatcher.

Given an appropriate implementation of such task and event queues, the navigator and dispatcher components can be run by threads which are distributed on different physical hosts, e.g., a cluster of computers [5]. To achieve a large task execution capacity dispatchers can be run by a large thread pool. Similarly, navigators running different (and independent) process instances can also be replicated among a pool of threads.

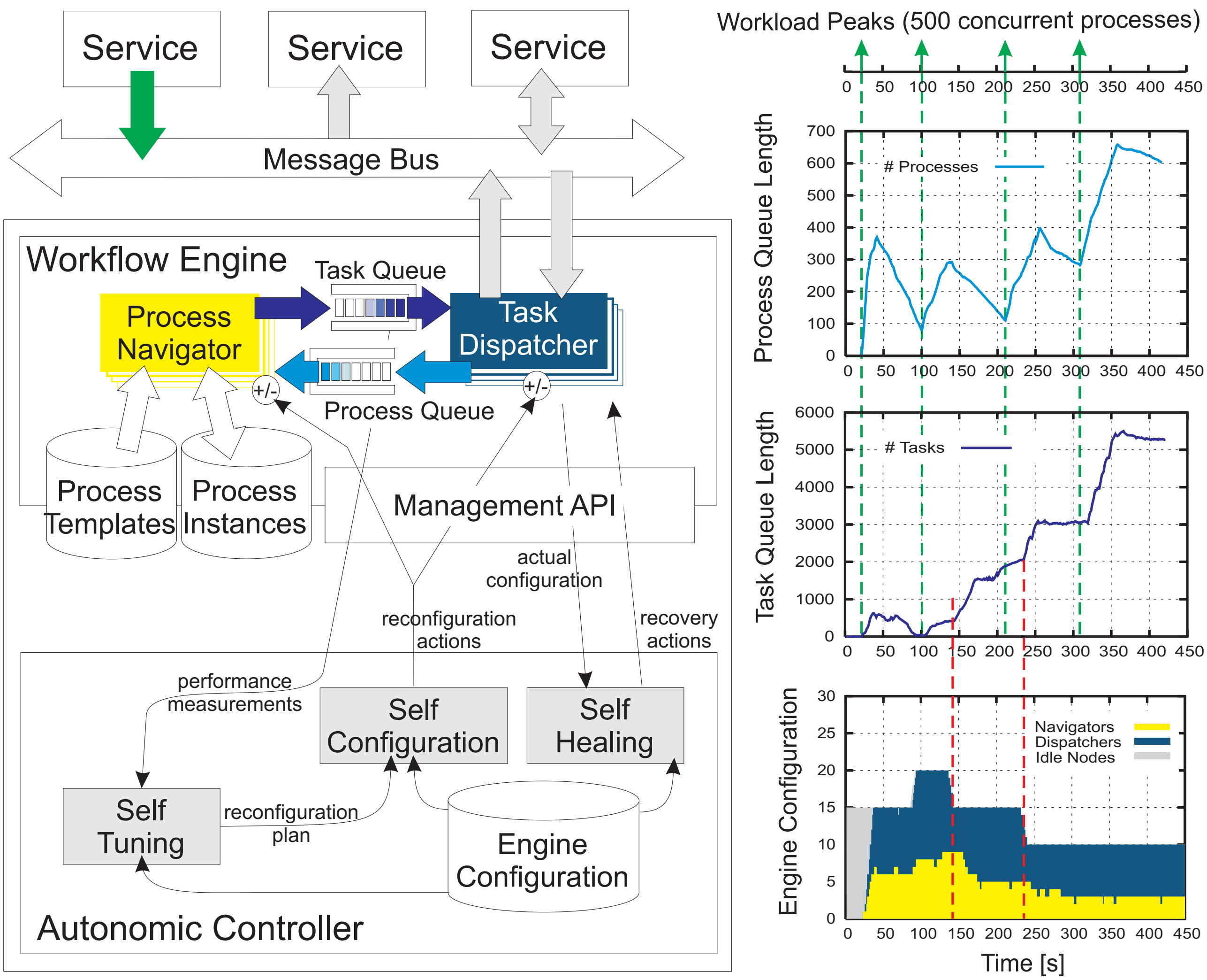

Figure 2: Architecture of a self-managing workflow engine (left) and performance evaluation (right)

4 A Deployment Dilemma

Although such stage-based architecture delivers the necessary flexibility to adapt the number of dispatcher and navigator components to the workload executed by the engine, it opens up another important problem, related to the *deployment* and *management* of such system. The engine may face an unknown number of internal and external clients that can define and run an unpredictable number of processes concurrently. Thus, the structure of these processes and the number of their instances that will have to be executed in response to clients cannot be determined a priori. This makes it difficult to choose between a centralized or a distributed solution for the deployment of the system. Moreover, in case a distributed approach is chosen to provide the required level of performance, the correct amount of resources must be provisioned and these must be managed and optimally configured, in terms of how the resources are allocated to navigators and dispatchers.

To illustrate this problem, in Figure 3 we include an example showing the sensitivity of the system to its configuration. In this example we started 1000 concurrent processes and measured their total execution time using different configurations of the engine.

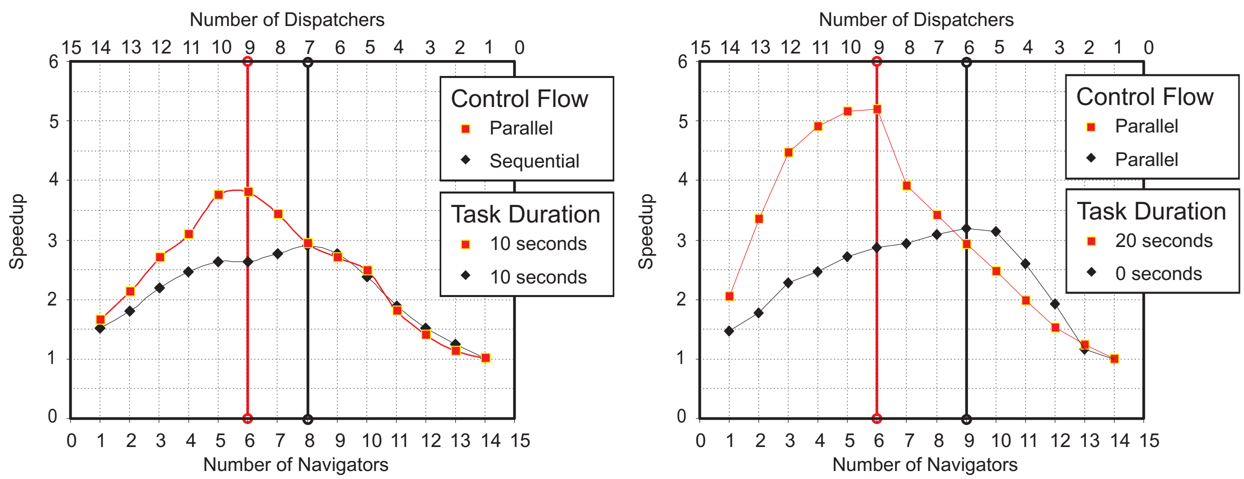

Figure 3: Speedup relative to the slowest configuration of four different workloads over all possible configurations of the engine deployed on a cluster of 15 nodes

Tuning the engine by allocating the "right" amount of navigators and dispatchers can – in this example – achieve a 5x speedup in the execution of the same workload. Still, the optimal configuration for one kind of workload could turn out to be sub-optimal for a different load. In the example (Figure 3 right), we use two workloads characterized by different task durations. Fast tasks can be consumed quickly by the dispatchers and shift the load onto the navigators, which have to resume executing processes after having issued the tasks which immediately complete. For short tasks duration, the highest speedup is found with a configuration which allocates more resources to navigators. The opposite occurs with slow tasks, which keep the dispatchers busy for a longer time. In Figure 3 left, we show that with tasks of the same duration, second order effects due to the process control flow structure become apparent.

From this example, it is clear that it is not enough to base the deployment decisions on estimating the performance of a certain configuration of the system by making assumptions about the properties of its workload [2]. Once these assumptions no longer hold, the system will be misconfigured and use the available resources inefficiently. Instead of a static solution, we choose to follow a dynamic approach based on a closed feed-back loop. As we are going to illustrate in the following section, we have extended the workflow engine with self-management capabilities so that it can adjust its configuration on the fly based on measurements of its performance under the actual workload.

To do so, we introduce an autonomic controller whose algorithms and policies do not make any assumption about the structure of the processes to be executed. This controller can dynamically grow and shrink the size of the system based on the number of processes that are currently active. Such self-managing engine can be initially be deployed in a centralized configuration and gradually evolve as a distributed engine as its workload gets larger. This avoids the problem of resource overprovisioning, where the workflow engine would have to be dimensioned for peak capacity at all times.

Such a solution requires the engine to support dynamic reconfiguration. Clearly, stopping the entire engine in order to migrate some of its components and to alter its configuration is not possible as it would affect the availability of the processes that are published as a Web service through the engine. Instead, with our design, to grow the size of the engine, i.e., to increase its process or task execution capacity, it is enough to provision an additional thread for executing the navigator or dispatcher components. In case of overprovisioning, threads can be relinquished and the size of the engine reduced accordingly.

5 Autonomic Deployment

The engine's architecture has been designed to provides all of the necessary extension points to follow a "bolt on" approach to achieve self-management [3]. A management API for monitoring the performance of the engine and for applying reconfiguration actions is available. This interface can be used both for manual system administration tasks, as well as automatic self-management when the appropriate autonomic controller component is added.

As opposed to measuring raw hardware metrics (e.g., CPU utilization) of the physical hosts running the engine, in our approach we have chosen to observe how the workload affects the performance of the engine at a higher level of abstraction. The workload for the engine can be defined as the number of active process instances that are being executed. This can be measured as the workload affects both execution stages and influences the length of the process queue serviced by navigators as well as the number of tasks to be invoked by dispatchers. The execution of a large number of processes will generate a large number of tasks. Still, monitoring the queues gives a precise picture of the performance of each stage. The amount of queued tasks depends on the degree of parallelism within a process and will change for processes having different control flow structures. Additionally, the size of the process queue will grow if several tasks complete their execution at the same time.

As shown in the lower part of Figure 2, this performance information is fed into the autonomic controller, which processes it and reacts by applying the appropriate reconfiguration and recovery actions to the engine. The controller is structured into three functional components: self-healing, self-tuning and self-configuration, which interact asynchronously and share a common model of the engine's configuration. As a first approximation, this model includes information about the available resources of the cluster and their allocation state. The configuration information is kept up to date by the self-healing component which periodically monitor its consistency with respect to the actual configuration of the cluster. Once a mismatch occurs, the self-healing component detects a failure in the engine, updates the configuration model and performs the appropriate recovery actions. For example, tasks executing on a failed dispatcher have been lost and have to be requeued to be retried.

The goal of the self-tuning and self-configuration component is to work together in order to keep the engine provisioned with the optimal amount of resources and ensure that the current configuration provides a good performance. The self-tuning component periodically reads performance measurements (i.e., the size of the process and task queues) from the engine and uses this information to detect imbalances in the current configuration. Measuring the length of the event and task queues makes it easier to define high level policies to control the engine's configuration. Such policies map observed variables into a reconfiguration plan. For example, if the task queue grows beyond a certain threshold, a dispatcher should be added to increase the rate of task execution.

This abstract plan (e.g., to add one dispatcher) is executed by the self-configuration component. Decoupling planning from execution is important not only because it takes time to perform the actual reconfiguration, but also because it allows the controller to choose the optimal resource targeted by the reconfiguration plan. While a reconfiguration is taking place, the self-tuning component can continue to observe the system's behavior and possibly update the reconfiguration plan with new decisions based on more up-to-date information. At the same time, the self-configuration component can choose the appropriate resource on which to apply the plan by minimizing the distruption caused by the reconfiguration.

6 Evaluation

To evaluate the architecture of the engine and its autonomic capabilities we have performed a number of experiments which (1) motivate the need for adding self-management capabilities to the engine (2) show that the controller can indeed automatically reconfigure and heal the system [4] and (3) compare the performance of different control policies [11].

Due to space limitations, in this section we only describe the results of a self-healing experiment. In addition to adjusting the configuration in response to changes in the workload applied to the system, in this case, the controller also reacts to external changes in the system configuration. The right side of Figure 2 shows a trace of how the engine evolves, from the point of view of the controller. The top two graphs include measurements of the performance of the engine in terms of the length of the task and process queues. The bottom graph shows various snapshots of the configuration of the engine over time, defined as the number of nodes of the cluster that have been allocated to run dispatchers and navigators.

Given the lack of benchmarks for autonomic workflow engines, we have performed a basic load test, where the system is periodically hit by a peak of n messages that are handled by starting the execution of the same number of processes in parallel. To simplify the analysis of the results, these processes have the same structure and contain the same number of tasks. In the experiments, four peaks of 500 processes arrive at $t = 20s, 100s, 205s, 305s$. The controller notices that the workload has increased by observing the evolution of the process queue length. When such queue gets longer, it means that the engine needs to allocate more process execution capacity. Thus, the controller allocates up to 5 navigators to service the process queue. Once the processes begin execution, also the task queue gets filled up and, in the first part of the experiment, the controller allocates up to 10 dispatchers to deliver the required task execution capacity.

While the second peak arrives, at $t = 100s$, the engine undergoes a maintenance operation. First, 5 nodes are added to the pool of resources of the engine, then other 5 nodes are taken out of the pool for maintenance (at $t = 140$). This manual node rotation is part of the normal maintenance of the system and should not disrupt its operations. The controller immediately makes uses of the newly added resources by allocating 3 additional dispatchers and 2 additional navigators. Still, once 5 nodes are taken out of the pool, the self-healing component notices their disappearance and recovers the tasks and processes that were running on the failed nodes. Also, the configuration of the remaining nodes is out of balance. This will be corrected before the next peak of processes arrives. At $t = 230$ the newly added nodes fail and the engine continues running with only 10 nodes. Clearly its performance has decreased as both task and process queues get increasingly longer. The controller tries to make use of the remaning nodes and the system keeps running.

7 Conclusion

In the same way that using database engines provides considerable savings in terms of code to be developed in large applications, workflow engines greatly simplify the orchestration problem in application integration settings. Yet, for developers to be able to take advantage of such savings in coding, the performance of existing workflow engines needs to be significantly improved.

In this paper we show how the processing capacity of a service composition engine based on the workflow paradigm can be extended automatically in response to changes in the load. Although we have shown how to apply it to JOpera, our approach is independent of the particular workflow engine, as its general principles can be applied to any process execution engine for Web service orchestration (e.g., other implementations of WS-BPEL [8]).

The solution described builds upon three important ideas: separating dispatching from navigation in the process execution, implementing them as separate modules, and designing appropriate policies for determining how many dispatchers and how many navigators are needed according to the current workload. The fact that the system can dynamically adjust the number of navigation and dispatching modules it utilizes by itself is an important property that frees up the developer and system administrator from having to worry about tuning and deployment configurations.

Downloading JOpera

The latest release of JOpera for Eclipse, including several examples to get started, can be downloaded from `www.update.jopera.org`. Additional publications and documentation can be found on `www.jopera.org`.

Acknowledgements

Part of this work is funded by the European projects: IST-FP6-004559 SODIUM (Service Oriented Development In a Unified fraMework) and IST-FP6-15964 AEOLUS (Algorithmic Principles for Building Efficient Overlay Computers).

References

[1] G. Alonso, W. Bausch, C. Pautasso, M. Hallett, and A. Kahn. Dependable Computing in Virtual Laboratories. In *Proc. of the 17th International Conference on Data Engineering (ICDE2001)*, pages 235–242, Heidelberg, Germany, 2001.

[2] M. Gillmann, W. Wonner, and G. Weikum. Workflow Management with Service Quality Guarantees. In *Proc. of the ACM SIGMOD Conference*, pages 228–239, Madison, Wisconsin, 2002.

[3] R. A. Golding and T. M. Wong. Walking toward moving goalpost: agile management for evolving systems. In *First Workshop on Hot Topics in Autonomic Computing*, Dublin, Ireland, October 2006.

[4] T. Heinis, C. Pautasso, and G. Alonso. Design and Evaluation of an Autonomic Workflow Engine. In *Proc. of the 2nd International Conference on Autonomic Computing*, Seattle, WA, June 2005.

[5] L. jie Jin, F. Casati, M. Sayal, and M.-C. Shan. Load Balancing in Distributed Workflow Management System. In G. Lamont, editor, *Proc. of the ACM Symposium on Applied Computing*, pages 522–530, Las Vegas, USA, 2001.

[6] R. Khalaf, A. Keller, and F. Leymann. Business Processes for Web Services: Principles and Applications. *IBM Systems Journal*, 45(2):(to appear), 2006.

[7] F. Leymann, D. Roller, and M.-T. Schmidt. Web services and business process management. *IBM Systems Journal*, 41(2):198–211, 2002.

[8] OASIS. *Web Services Business Process Execution Language (WSBPEL) 2.0*, 2006.

[9] C. Pautasso and G. Alonso. From Web Service Composition to Megaprogramming. In *Proc. of the 5th VLDB Workshop on Technologies for E-Services (TES-04)*, pages 39–53, Toronto, Canada, August 2004.

[10] C. Pautasso and G. Alonso. JOpera: a Toolkit for Efficient Visual Composition of Web Services. *International Journal of Electronic Commerce (IJEC)*, 9(2):104–141, Winter 2004/2005.

[11] C. Pautasso, T. Heinis, and G. Alonso. Autonomic Execution of Service Compositions. In *Proc. of the 3nd International Conference on Web Services*, Orlando, FL, July 2005.

[12] J. Yu and R. Buyya. A Taxonomy of Workflow Management Systems for Grid Computing. *Journal of Grid Computing*, 2006 (to appear).