Autonomic Computing for Virtual Laboratories

Cesare Pautasso[1], Win Bausch[2], Gustavo Alonso[1]

[1] *Department of Computer Science*
Swiss Federal Institute of Technology (ETHZ)
ETH Zentrum, 8092 Zürich, Switzerland
tel +41 01 632 0879 fax +41 01 632 1425
`pautasso@inf.ethz.ch, alonso@inf.ethz.ch`

[2] *AWK Group AG*
Leutschenbachstrasse 45
8050 Zürich, Switzerland
Tel. +41 44 305 97 63, Fax +41 44 305 95 19
`win.bausch@awkgroup.ch`

Abstract. Virtual laboratories can be characterized by their long-lasting, large-scale computations, where a collection of heterogeneous tools is integrated into data processing pipelines. Such virtual experiments are typically modeled as scientific workflows in order to guarantee their reproduceability. In this chapter we present JOpera, one of the first autonomic infrastructures for managing virtual laboratories. JOpera provides a sophisticated Eclipse-based graphical environment to design, monitor and debug distributed computations at a high level of abstraction. The chapter describes the architecture of the workflow execution environment, emphasizing its support for the integration of heterogeneous tools and evaluating its autonomic capabilities, both in terms of reliable execution (self-healing) and automatic performance optimization (self-tuning).

1 Introduction

More and more scientific disciplines are switching from *in vitro* to *in silico* research where natural phenomena are explored using a computer in a virtual laboratory instead of being observed in the field. On the one hand, this is due to the fact that the cost of storing observations has become lower than the cost of making them. On the other hand, scientific workflow tools [15] – such as the one described in this chapter – have been developed in order to make it easier for scientist to process and analyze such observations by composing an increasingly large number of basic analysis and simulation tools.

Although virtual laboratories are typically associated with very large amounts of data, data processing is even more critical than data management due to the

This work is partly supported by grants from the *Hasler Foundation* (DISC Project No. 1820)

sheer computational complexity involved. Given the heterogeneity and complexity of the underlying distributed execution environments and the long duration of the computations involved, it is not feasible to manually manage the lifecycle of such virtual experiments. Instead, a virtual laboratory infrastructure should automate most tasks related to the reliable and reproduceable execution of such computations. Ideally, a virtual laboratory infrastructure should provide a team of scientists with support for easily creating and efficiently running virtual experiments. Additionally, virtual laboratories are rarely designed in a top-down fashion. They typically emerge from a collection of disconnected pieces of data processing code (e.g., written in FORTRAN) and glue scripts (e.g., in PERL [1]) that are developed and maintained by individual scientists. Such an ad-hoc approach leads to systems that are difficult to modify and maintain, cannot be easily shared among researchers and involves rather primitive and unsystematic methods for running, monitoring, and steering the computations.

Considering that all of these problems are a major source of inefficiencies, it becomes clear that an organized way to store and manage information and meta-information about the entire lifecycle of a virtual experiment is critical to its success. Thus, not only high level languages and abstractions to define such computations are needed but also efficient execution tools integrated with user-friendly management and monitoring environments are required.

In this chapter we focus on how this functionality has been provided in JOpera [16], an autonomic process support system specifically tailored for virtual laboratories. The JOpera project has its roots in the BioOpera [5] project and it has been developed at the Information and Communications Systems Research Group of ETH Zurich. JOpera extends the Eclipse platform with a graphical environment where scientists can use a drag, drop and connect programming metaphor to define distributed computations out of reusable components. The resulting high-level models are then automatically compiled into Java bytecode so that they can be efficiently executed by the system. In case of virtual laboratories where a large number of computations are concurrently executed, JOpera can distribute their execution across a cluster of computers in order to provide the appropriate level of performance. Moreover, JOpera includes self-management capabilities, where the distributed engine can automatically determine its optimal configuration based on its current workload. With this, the need for manual intervention and tuning the system's performance is greatly reduced.

The rest of this chapter is organized as follows. We discuss in more detail the problems of virtual laboratories by showing some typical examples in Section 2. In order to address these challenges, scientific workflow tools such as JOpera offer a solution based on two aspects. The first one consists of a language targeted towards modeling virtual experiments at a high level of abstraction (Section 3). The second one – presented in Section 4 – lies in the middleware infrastructure supporting the execution of such a language. An evaluation of the autonomic capabilities of the system is discussed in Section 5 before concluding the chapter in Section 6.

2 Motivation

This section illustrates the issues scientists running large scale virtual experiments need to cope with. Each example represents a pattern frequently encountered in a virtual experiment. Each of these patterns has different characteristics and requires a different type of support from the virtual laboratory infrastructure.

2.1 Structured computations

A structured computation involves a set of applications that needs to be executed in a specific order. These applications run on different operating systems and hardware platforms. They exchange data with each other through a number of input and output mechanisms (e.g., command line input parameters, input and output files, web page downloads) This data is produced at different points in time throughout the computation and may have to be converted between different formats. Programming such application may prove to be too difficult for ordinary users, if appropriate high level programming tools are not available.

In addition to design-time support, run-time support is also important. For instance, considering a distributed environment, manually taking care of routing

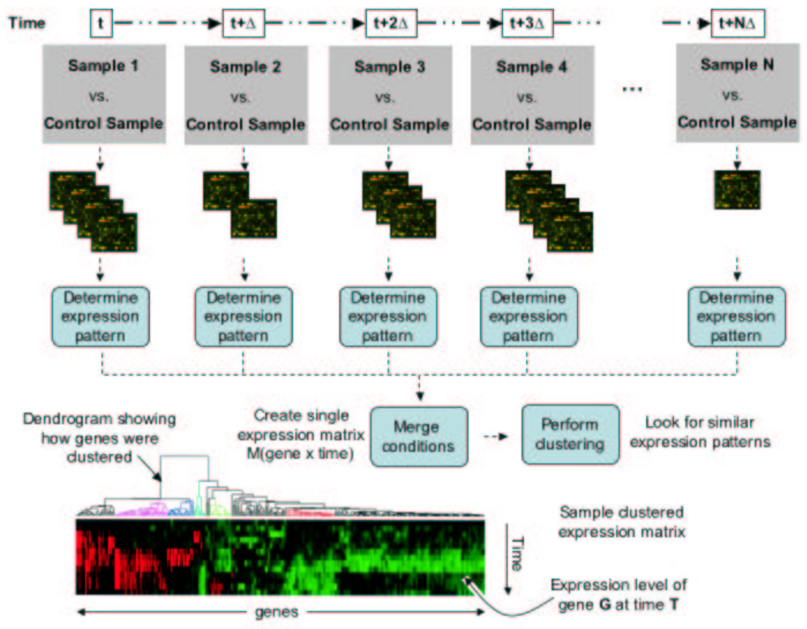

Time
t
t+Δ
t+2Δ
t+3Δ
t+NΔ
Sample 1 vs. Control Sample
Sample 2 vs. Control Sample
Sample 3 vs. Control Sample
Sample 4 vs. Control Sample
...
Sample N vs. Control Sample
Determine expression pattern
Dendrogram showing how genes were clustered
Create single expression matrix M(gene x time)
Merge conditions
Perform clustering
Look for similar expression patterns
Sample clustered expression matrix
Time
Expression level of gene G at time T
genes

Fig. 1. Microarray analysis pipeline: from raw samples to correlated expression patterns

data from task to task at the right time becomes difficult, time consuming and error-prone. Thus, data transfers should be automated, not just to improve the efficiency of the virtual experiment, but also to collect important lineage and data provenance information. The goal is to automatically log all of the necessary meta-data in order to support the correct interpretation of the results of a virtual experiment, i.e., by tracing how this was generated.

An example of this structured computation pattern can be found in the bioscience domain. Microarray technology is a promising approach to find clues concerning the function of specific genes in a cell's metabolism. The idea is to expose the cell to an artificially created stimulus (also called *condition*) and observe the cellular response in terms of the level of activity (or the *expression level*) of some genes over time. Development of appropriate computational models as well as innovation in wet lab equipment have made it possible to move elements of the microarray processing pipeline into virtual laboratories.

Such a virtual microarray experiment involves a range of data extraction, transformation and correction steps that need to be performed prior to a complex statistical analysis of the data. Figure 1 provides a high-level overview of the procedure, which is described in more detail in [3]. This microarray processing pipeline was implemented with BioOpera by integrating existing, standalone, publicly available software packages written in different programming languages and maintained by different reseach groups [5].

2.2 Embarrassingly parallel computations

Whereas the main challenge of the *Microarray analysis pipeline* concerns the specification of the complex interactions between a large set of heterogeneous tools, in this section we deal with the evolution of the execution environment when running long-lived computations.

An embarrassingly parallel computation consists of a set of tasks that can be processed independently of each other. This kind of computations are commonly used in a virtual laboratory setting as, given enough execution capacity, their execution time can be reduced by executing all tasks in parallel. However, when such a pattern is implemented without appropriate support from the virtual laboratory infrastructure, several challenges become apparent. For instance, choices need to be made concerning the granularity of the tasks, how to schedule tasks to run on the available resources (e.g., whether several tasks can share a single processor), and finally, how to handle the failure of individual tasks. Without appropriate support, the onus for such chores lies on the *user*. Not surprisingly, manually and painstakingly maintaining such computation becomes the dominant factor in the overall cost of performing such virtual experiments and does not scale to a large number of tasks running on a large number of computers.

An example of this kind of computation is a sequence alignment, a problem that lies at the heart of comparative genomics. Given an unknown set of nucleo or peptide sequences, the initial step into any inquiry concerning the evolution, structure, and function (e.g., [8,9,20]) of these biomolecules consists of the *cross-comparison* of each sequence in this set against every sequence of a reference data set such as Swiss-Prot [6] - an *All vs. All*, if the two data sets coincide. Typically, a single comparison requires seconds of CPU time, depending on the method that is being used and the length of the sequences being compared, and that the total number of pairwise sequence comparisons is in the order of billions. From this, several years of CPU time are required to perform the whole experiment. Being composed out of a number of pairwise sequence comparisons independent of each other, an All vs. All is embarrassingly easy to parallelize: each alignment can be computed independently.

Ample details concerning a month-long lifecycle of running such a computation with BioOpera can be found in [4]. Throughout the computation, processor availability has been subject to substantial unexpected and uncontrolled fluctuation. Without load balancing or job migration across machines to compensate for resource failures, utilization of the overall available computing resources is bound to be suboptimal. Also, a failure of the node coordinating the computation halts the entire computation. Dealing with these issues manually is indeed inefficient and time consuming. If a computation environment made out of hundreds of hosts is considered, it is clear that all of the previously described aspects of its execution should be controlled automatically.

2.3 Parameter-sweep computations

This pattern represents a combination of the ones discussed in the previous two sections. A parameter sweep computation [2] consists of applying the same algorithm to all parameter value combinations in a predefined parameter space. Since each each parameter combination can be typically processed independently, parameter sweeps are embarassingly parallel and share the requirements for a reliable and distributed execution environment. Concerning structured computations, not only a complex computation is applied to each parameter combination but also traceability needs to be guaranteed (i.e., in order to correlate which results have been produced by which input parameter values).

An example parameter sweep application to which JOpera has been successfully applied involved the simulation of protocols for wireless ad-hoc networks [21]. Communication partners in such networks are in motion with respect to each other and may leave and join the network at any time. Additionally, the network is infrastructureless. Unlike in mobile telephony, for instance, there is no fixed infrastructure that keeps track of nodes and routes data from sender to receiver. Data is directly routed through the mobile nodes and routing paths have to be recomputed as nodes move in and out of transmission range.

The objective of the experiment described here is to compare the simulated behavior of a set of resource reservation protocols under certain assumptions like congestion, network latency or node population distribution. In order to gain a complete understanding of the problem, this parameter space should be explored in its entirety. Although an individual simulation is on average relatively short, on the order of 20 seconds of CPU time, the size of the parameter space makes running the entire simulation challenging. Each simulation depends on 17 parameters, resulting in around 1.5 million independent simulations. Again, parallel execution on a cluster of computers is mandatory to ensure that the results are delivered in a reasonable amount of time.

2.4 Discussion

From the previous examples it is clear that a virtual laboratory infrastructure needs to cope with a variety of design-time and run-time problems. These involve providing good abstractions to model the structure of computations that are built by integrating heterogeneous scientific tools. However, modeling is not enough, as computations need to be reliably and efficiently executed in a distributed (and failure-prone) environment. The main features of such a virtual laboratory infrastructure can be categorized as follows:

Modeling An easy to use, intuitive programming environment should be provided so that scientific computations can be specified at a high level of abstraction by fostering the reuse of existing tools.

Integration Virtual laboratories must cope with heterogeneity, not only regarding data formats but also concerning the environments on which analysis tools are executed.

Distribution Distribution is another property of virtual laboratories, as local and remote (e.g., Web-based) data sources and tools have to be accessed.

Steering In addition to reporting their status and progress, long-running computations require support for interacting with them in order to proactively steer their execution.

Scalability In this context, the notion of scalability needs to be extended to encompass the virtual laboratory infrastructure itself, which should scale to handle a very large number of virtual experiments.

Fault Tolerance Given that most of today's Grid and cluster environments are failure prone, various failure masking and exception handling approaches should be in place in order to minimize the number of troubleshooting activities to be performed.

Combining all of these features and mechanisms with the appropriate self-management strategies yields an autonomic infrastructure for managing virtual laboratories as we are going to describe in the following sections.

3 Modeling Virtual Experiments with Processes

A language for modeling virtual experiments should allow scientists to model all aspects of a virtual laboratory (e.g., which tools to use, what are their dependencies, how to invoke them, where the data should be stored) in a well-defined, formalized way so that these experiments can not only be executed in a fully automatic fashion but the management of related metadata is also automated.

Thus, the main challenge in designing such a language lies in keeping the balance between two extremes. On the one hand, a risk lies in abstracting away too many details – e.g., like the data flow, typically disregarded in many business process modeling languages – that are of primary importance for modeling executable scientific computations. On the other hand, a lower-bound is defined by traditional scripting languages (e.g., PERL or PYTHON). These languages can also be used as the glue to patch together and run virtual experiments. However, they lack the necessary abstractions to deal with issues such as reuse of scientific tools and algorithms, scalable, reliable and persistent execution, simplified orchestration of distributed components, interactive monitoring and steering of computations as well as tracking lineage and data provenance meta-data.

In the following, we give an overview about the abstractions provided by JOpera's languages (Processes and Programs) and how they fit together (Binding and Flow).

3.1 Modeling the flow with processes

Processes can be seen as an executable blueprint of a distributed application built using a pipe-and-filter architectural style [7]. Processes model computations as a combination of heterogeneous tools which are to be executed as the computation goes through its various stages. Processes can be run once over a certain input dataset, or can also be applied over a range of input parameter values.

In JOpera, processes model the interactions between a set of programs. A JOpera process consists of a set of *tasks* linked by *data* and *control* flow dependencies. Tasks represent each step of the computation to be carried out. Executing a task involves the invocation of an external program or the call of another sub-process.

Both the data and the control flow of a process can be formally described as a graph. The edges of the control flow graph link the tasks of a process and define their partial order of execution. These edges can be labeled with boolean expressions in order to select upon which condition they are activated and thus provide support for adding alternative or multiple branches, loops and synchronization points in the control flow. The data flow edges link data parameters of tasks declaring how information is transferred from one program to the next. Processes also have input and output parameters, so that it is possible to pass information to a process when starting it and retrieving its results when it is completed. Data flow and control flow are related since tasks consuming data cannot be started before all tasks producing the required data have successfully completed their execution. Thus, when executing a process, JOpera analyzes its structure and concurrently schedules all tasks that are found to be independent. If enough computing resources are available, these tasks will be executed concurrently.

Traditionally, workflow management tools have used a visual syntax to graphically depict the flow linking the various scientific tools together into a process. This is also the approach followed in JOpera with its JOpera Visual Composition Language (JVCL). With it, both the control flow and data flow of a process can be specified using a very simple, graph-based visual notation. Nevertheless, the

JOpera visual composition language supports advanced constructs (e.g., iteration, streaming, reflection, recursion, nesting, or dynamic late binding) without resorting to ad-hoc (and difficult to interpret) extensions of the visual syntax. We refer the reader to [18] for an in-depth presentation of the JVCL language.

3.2 Binding processes with programs

The notion of binding in JOpera defines the flexible relationship between processes (i.e., the compositions) and programs (i.e., the components). Although processes model how a virtual experiment is composed out of a set of programs, the description of the programs themselves is kept – by design – separate from the processes. This separation has several advantages. It enhances the reusability of the programs, which can be shared among different processes. Likewise, the same process can be reused by binding it with different programs.

More precisely, a binding defines what are the constraints to be satisfied by a program in order to be included in a process [19]. Such a binding can be evaluated along the entire lifecycle of a process: at design-time (early binding), at compilation-time, at deployment-time, at run-time (late and very late binding).

Given the goal of supporting an open and heterogeneous set of programs, JOpera makes very little assumptions about the mechanisms that are used to invoke their functionality. Instead JOpera provides a meta-library of component types that can be used to define programs. Programs wrap existing tools employing the most appropriate invocation mechanism both in terms of performance but also development convenience [17]. Proof of the openness of the JOpera service meta-model is provided in Table 1 where all currently supported component types are listed. Depending on the relevant aspects that should be taken into account when designing a virtual experiment, these components can be classified along the following dimensions:

Granularity Both fine-grained (e.g., Java snippets) and coarse-grained (e.g., Web services) programs are supported by JOpera within a single process. Furthermore, the overhead of invoking each component type is proportional to its granularity. In other words, JOpera can leverage the standardized (but relatively inefficient) SOAP protocol without being constrained by it. If necessary, more efficient invocation mechanisms can still be selected to access fine-grained programs.

Local vs. Remote invocation At run-time, programs can be separated from a process by an increasingly large distance. For example, Java methods are invoked by a thread running within the same Java virtual machine where the process is running. Legacy UNIX applications invoked through the local operating system shell run in a separate operating system process with respect to the one running the JOpera process. Additionally, programs can represent the execution of an application on a remote host through a secure shell connection and, going even further away, jobs submitted to a resource management and scheduling system (e.g., Condor [14] or Globus [10]) to be executed on a cluster of computers in a remote Grid environment.

Component Type		**Description**
Local Computation		
UNIX Application	(`UNIX`)	Execute a command line through the local operating system
Java Method	(`JAVA`)	Call a local Java method
Java Snippet	(`JAVA.SNIPPET`)	Embed a Java snippet into the process
Remote Computation		
Java Remote Method	(`JAVA.RMI`)	Invoke a remote Java method
Web Service	(`SOAP`)	Web service call (using raw SOAP messages)
Web Service	(`WSIF`)	Web service call (using the WSIF framework [13])
Secure Shell	(`SSH`)	Execute a remote command through a secure shell connection
Data Transfer		
Web Page	(`HTTP`)	Download (or upload) a page from a web site
Secure Copy	(`SCP`)	Transfer a file with secure copy
Database		
Database Query	(`SQL`)	Send any SQL statement to a JDBC compliant database
Telegraph Query	(`TELEGRAPH`)	Subscribe to a telegraph stream described by an SQL query
XML transformation		
X-Path Query	(`XPATH`)	Query an XML document with X-Path
Style Sheet Transformation	(`XSLT`)	Transform an XML document with an XSL transformation
Cluster/Grid computing		
Globus [10]	(`GLOBUS`)	Submit a job to a grid managed by Globus
Condor [14]	(`CONDOR`)	Submit a job to a cluster managed by Condor
Internal		
JOpera Echo	(`ECHO`)	Echo a message back
JOpera Process	(`OPERA`)	Spawn another process
JOpera API	(`API`)	Call the API of JOpera
Human-oriented		
Workflow task	(`WF`)	Add a new activity to a user's worklist

Table 1. Summary of the component types currently supported by JOpera

Data-driven vs. Computation-oriented In addition to computations, programs can also be used to manage the data that is required and produced by other programs. Data-driven programs are used to model data transfers (e.g., file-staging through secure copy or GridFTP), access to persistent storage (e.g., SQL database queries), and can play the role of mediators and adapters (e.g., Java snippets or XML data transformations written in XPath, XSLT, or XQuery).

Interaction Style In addition to synchronous (RPC-style) interactions, where a program models the complete invocation of an external tool, we have also applied JOpera's meta-model to provide support for asynchronous interactions, where the execution of a program involves a one-way message exchange or the start (or termination) of an independently running application. In this case, data exchanges between the process and the program can occur at any time, i.e., when the program is started (input), after it has completed (output) but also during its execution (streaming).

Machine-bound vs. Human-oriented Although most computational tools are usually meant to be executed in non-interactive mode, parts of a process may also explicitly include a task requiring some form of human intervention, e.g., to validate partial results and steer the process accordingly or take some manual corrective actions before the computation is carried on.

Data vs. Metadata Reflection and introspection are also two important features of JOpera's visual composition language. With these it becomes possible, e.g., to control the execution of a process from another process, or to dynamically discover properties about the execution environment and use this information from within a process. For example, it is possible to dynamically detect how many resources are available and partition a dataset accordingly or measure the invocation time of a remote Web service to detect whether a service-level agreement has been violated.

Additional component types can be easily added to JOpera by plugging a service invocation adapter into the corresponding extension point, as we are going to show in the next section.

4 An Autonomic Infrastructure for Virtual Laboratories

The architecture of JOpera is composed of a set of Eclipse plug-ins (Figure 2). Following Eclipse's design guidelines, we have separated plug-ins responsible for the user interface (UI) from plug-ins that work with the internal process data model. Along an orthogonal dimension, we have also separated the design-time from the run-time functionality, so that, if necessary, the system can be deployed in a partial configuration (e.g., where only the run-time monitoring features are enabled). The compiler, which links the design-time to the run-time part has been developed in its own plug-in. On the run-time side, the run-time kernel provides the basic process execution infrastructure used by the compiled code. It is extended by the service invocation adapters plug-ins, which implement the

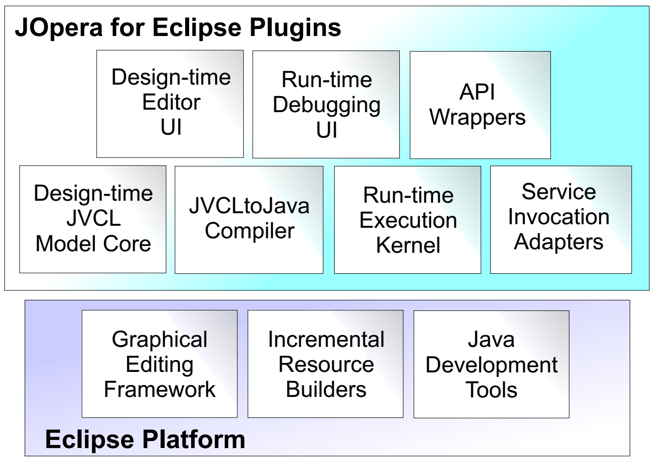

JOpera for Eclipse Plugins
Design-time Editor UI
Run-time Debugging UI
API Wrappers
Design-time JVCL Model Core
JVCLtoJava Compiler
Run-time Execution Kernel
Service Invocation Adapters
Graphical Editing Framework
Incremental Resource Builders
Java Development Tools
Eclipse Platform

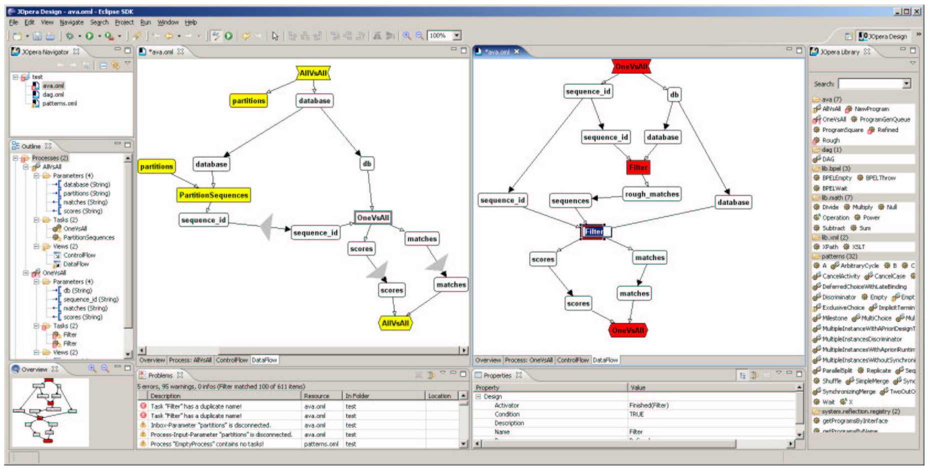
Fig. 2. JOpera is built as a set of Eclipse plugins

mechanisms and provide support for the protocols used to invoke the various kinds of components that were described in the previous section. Finally, the API wrappers are used to expose the functionality of the kernel to clients supporting a variety of protocols.

4.1 Design-time tools

The JVCL model core plug-in contains the functionality used at design-time to manage the information about programs and processes described in the JOpera Visual Composition Language. This includes the ability of internalizing such information loading it from an XML serialization. This plug-in also manages an object-oriented in-memory model of the processes and programs which has been automatically produced from the corresponding schema using a generative programming approach. Clients observing the model may use its event notification facilities to be notified when parts of the model are changed, e.g., to perform some incremental validation or to update the information displayed by the corresponding UI views. This way, after each modification, the model is checked incrementally for consistency with respect to various consistency criteria. In case a violation is detected, a specific problem (or a warning) marker is attached to the part of the model that triggered it. Such verification happens in the background, without user intervention so that errors and potential problems are reported immediately. In an agile development environment, such immediate feedback is nowadays taken for granted as it contributes to reducing the overhead of the typical compose-compile-fix development cycle and it is very important to decrease the slope of the environment's learning curve.

The editor UI plug-in contains the user-interface code that presents the content of the currently open processes to the developer. We use two different kinds



5 errors, 95 warnings, 0 infos (Filter matched 100 of 611 items)

Description	Resource	In Folder	Location
Task "Filter" has a duplicate name!	ava.oml	test	
Task "Filter" has a duplicate name!	ava.oml	test	
Inbox-Parameter "partitions" is disconnected.	ava.oml	test	
Process-Input-Parameter "partitions" is disconnected.	ava.oml	test	
Process "EmptyProcess" contains no tasks!	patterns.oml	test	

Fig. 3. JOpera: Design-time Editor and Background Model Checker

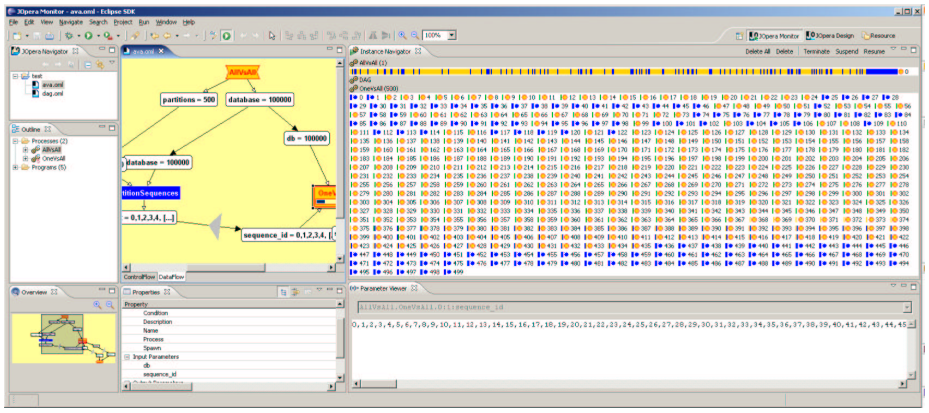


Fig. 4. JOpera: Run-time Process Monitor and Debugger

of visual user interfaces to display and edit the structure of a process. List-based forms are used to choose the services to be composed and to define their interface parameters. Additionally, the control flow and data flow graphs of the processes are edited in a visual environment. Such visual editor is implemented by extending the Graphical Editing Framework (GEF) of Eclipse to use the visual syntax of the JVCL language. In addition to providing a new kind of editor, the UI plug-in reuses the existing Outline, Problems and Property views of Eclipse to display the structure of the active composition, its current error and warning markers and the attributes of its selected graph elements (Figure 3).

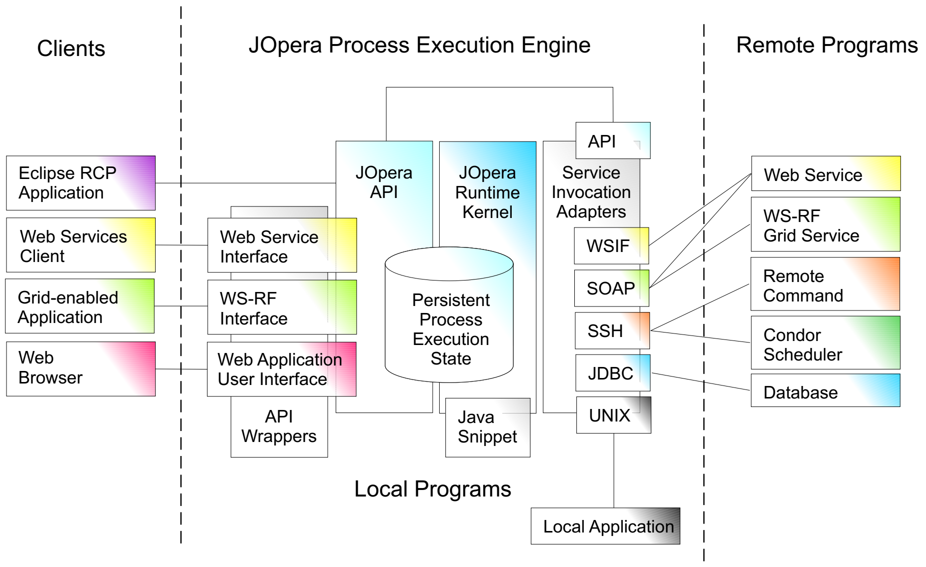


Fig. 5. Layered Architecture of the JOpera Process Execution Engine

4.2 Run-time tools

Following a model-driven approach and by leveraging Eclipse's incremental resource builders, JOpera's JVCLtoJava compiler plug-in incrementally recompiles the modified composition to Java executable code whenever a process is saved. This Java code is then once more compiled by Eclipse's integrated Java compiler into bytecode. The latter is then automatically and transparently re-deployed for execution by dynamically loading it into JOpera's run-time execution kernel.

At this point, a valid, compiled composition is ready to be executed. Unlike most current model-driven environments, the progress of the execution can be followed interactively in the same environment – and most important – using the same visual syntax that was used to define it. Thus, not only JOpera features a so-called reverse model transformation, where the original visual process definition is extracted back from the compiled bytecode, but is also able to join this with the current state of the execution. This way, the visual representation is augmented at run-time with color-coded information representing the state of the execution of each of the service invocations (e.g., white for not yet executed, yellow for active, blue for finished, red representing a failure). Using the tools provided by the debugging UI plug-in (Figure 4), individual data parameters can be inspected, so that – for example, in case a Web service is involved – the actual SOAP request and response messages can be displayed for debugging purposes. Similarly, in case a remote execution fails it is possible to distinguish whether the remote host could not be reached from the actual failure of the execution.

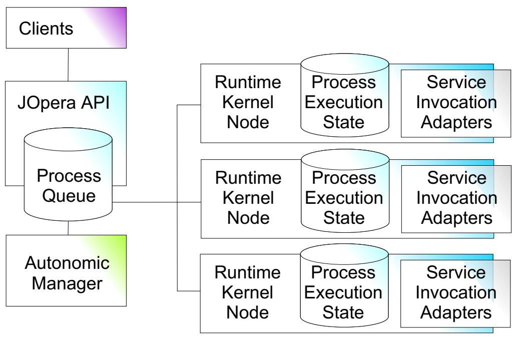


Fig. 6. Scaling the runtime kernel by replication

The persistent state of the execution and the navigation over the control flow graph of the process are managed by the JOpera Process Execution Engine. Figure 5 shows its various interfaces, towards clients used to access the functionality of processes and towards the local or remote programs invoked from a process. Processes are executed by the engine's run-time kernel, which delegates the interaction with different component types to a set of service invocation adapters. Its API can be accessed using a variety of means so that processes deployed in the kernel are automatically published, e.g., as Web and Grid services [12]. In this regard, JOpera can be seen as an open platform for heterogeneous service composition since it is possible to extend the kinds of services that can be composed by adding user-defined service invocation plug-ins.

In order to handle large workloads, the run-time kernel can be distributed on a cluster of computers as shown in Figure 6. Processes submitted by clients for execution are stored into a central queue so that they can be scheduled for execution on a node of the kernel having enough free capacity. As we are going to discuss in the next section, depending on the number and characteristics of the processes to be executed, one node of the cluster may not provide sufficient execution capacity. In this case, additional nodes can be dynamically allocated to the kernel by the autonomic manager component [11].

5 Evaluation

In this section we present some experimental results on the autonomic capabilities of JOpera. They validate the architecture of the system and show that it is possible to automatically deal with a significant set of failures and, in general, changes in the execution environment (self-healing) but also react to changes in the workload to be executed (self-configuration).

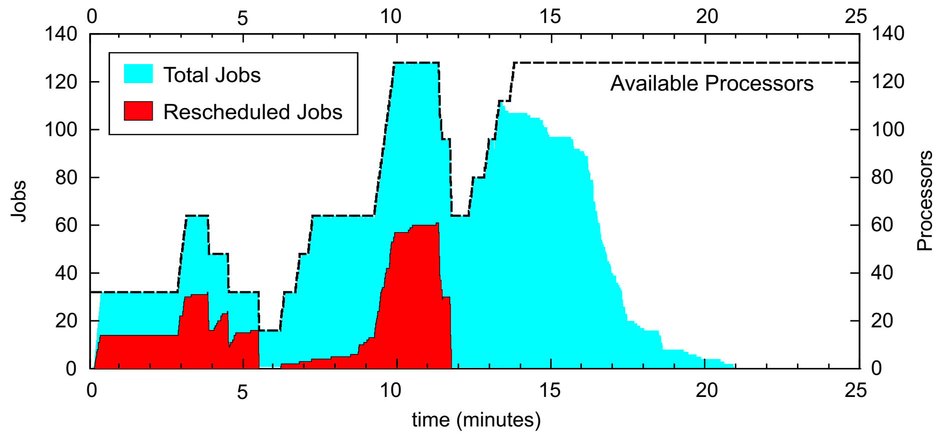


Fig. 7. Dealing with outages and cluster reconfigurations

5.1 Self-Healing capabilities

Dealing with outages in the execution cluster In this experiment we tested the system's ability to cope with changes in the resource set allocated to the execution of the All vs. All process using a reduced input data set. The workload consisted of 256 independent jobs, each requiring an average CPU time of 4 minutes.

Figure 7 shows a trace of the experiment execution using the distributed engine. The y-axis measures both the number of processors in the cluster as well as the number of jobs (each job is allocated to one processor). The dashed line represents the number of available processors. At time t, the Total line indicates the number of jobs running in the cluster. The Rescheduled Jobs line indicates how many jobs at a future point in time are going to be rescheduled due to a failure of the node where they have been running. Thus, the area under this line represent the amount of CPU time lost due to failures.

In general, Figure 7 demonstrates the ability of the kernel to adapt a running computation to the set of available processors, which has shrunk and grown many times throughout the experiment. The kernel is able to take advantage of new machines by immediately scheduling jobs on them and to reschedule lost jobs. Automatic rescheduling can be observed whenever a processor fails: the availability line drops since less processors are available for the computation. Upon such event, the kernel immediately retracts the jobs running on the failed processors to reschedule them on another node. In the graph, this is shown by the Rescheduled Jobs line closely following the number of available processors. Since a copy of the input data used by a task is stored persistently by the kernel as part of the state of the process execution, lost jobs can be recovered by sending a copy of such input data to another processor.

Kernel recovery Recovery of the kernel ensures that process execution resumes in a consistent state after a failure has interrupted the kernel's normal operation. In order to determine the overhead of such recovery, we measured the time taken by the various recovery steps:

1. Re-loading process instance state information from persistent storage;
2. Navigating through them in order to determine what are the tasks to be recovered;
3. Synchronizing the state of the tasks which are remotely executed.

The results of Figure 8 clearly indicate that the recovery times grow linearly with the number of tasks that were active at the time of the failure. More specifically, the most expensive operation is the loading of the instance data from the database, which takes 5 milliseconds when there are no tasks to be recovered, up to 50 seconds when loading 40 process instances composed of 100 tasks each. Since navigation is performed in main memory it is two orders of magnitude faster: less than 0.4 seconds for 4000 tasks. Synchronization with the cluster nodes is the step presenting the most time variability. This can be explained by the fact that when a recovering kernel attempts to contact a remote node to find out about the state of the task being recovered, it blocks either until the remote node responds or until the connection times out, which is the case if the remote node has failed. In addition to this timeout penalty all jobs lost due to node failures are automatically rescheduled adding to the duration of the recovery procedure.

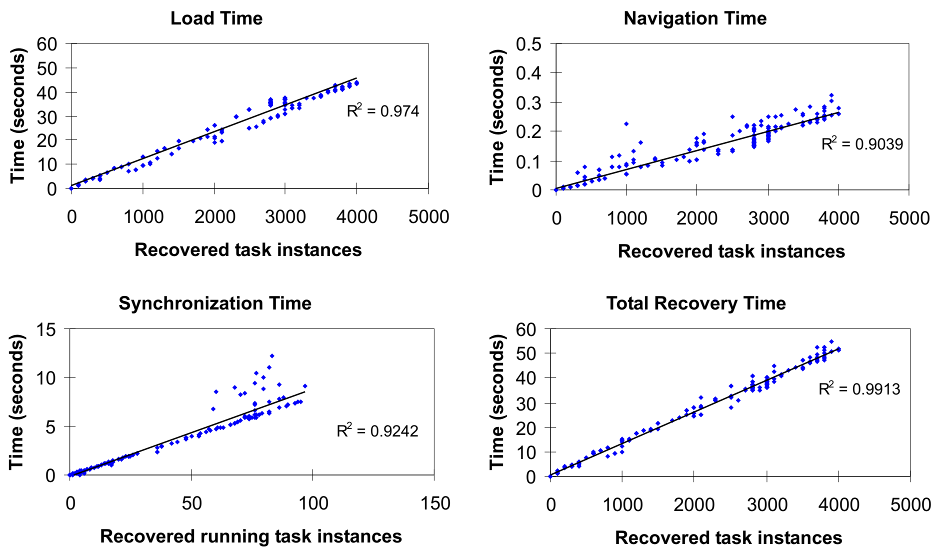


Fig. 8. Overhead of recovering a run-time kernel from persistent state

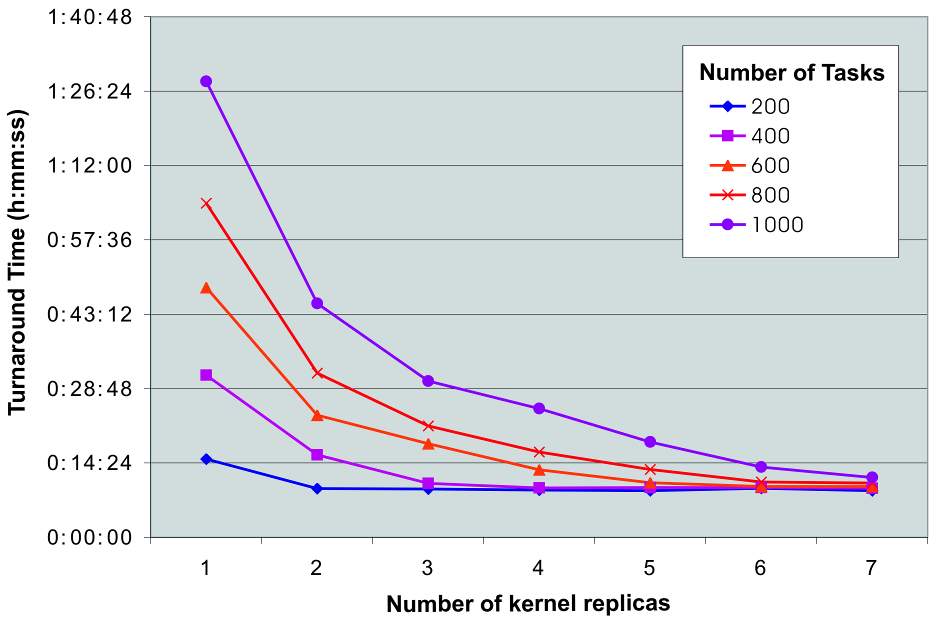


Fig. 9. Impact of replicating the kernel over a cluster

5.2 Self-configuration capabilities

Whereas the previous section described the self-healing capabilities of the system, where the kernel can survive failures of the underlying cluster environment, in this section we explore how the kernel can automatically adapt its configuration to optimally use the available resources. First, we show that the kernel can be replicated in order to service a given workload with better performance. Second, we show that the kernel, through its autonomic manager, can automatically determine a suitable degree of replication for a given workload. To this end, we have been analyzing the effect of a replication strategy where up to 7 copies of the kernel are employed to run the parameter-sweep experiment described in Section 2.3. The process uses from 200 up to 1000 concurrent tasks to computer over an increasingly larger input dataset.

Figure 9 shows the results for the static replication strategy, where the number of replicas (x-axis) of the kernel has been manually configured to study the effect of replication on the process turnaround time (y-axis). Overall, replication has a beneficial impact on turnaround time. The system scales well, as a 5-fold increase in workload can be handled with constant time by a 7-fold increase in the number of kernel replicas. Still, for smaller workloads, it is not necessary to fully replicate the execution environment, as – due to Amdahl's law – the speedup is limited, as it can be observed for the smallest workload (200 tasks), where no improvement can be observed after 2 kernels have been used.

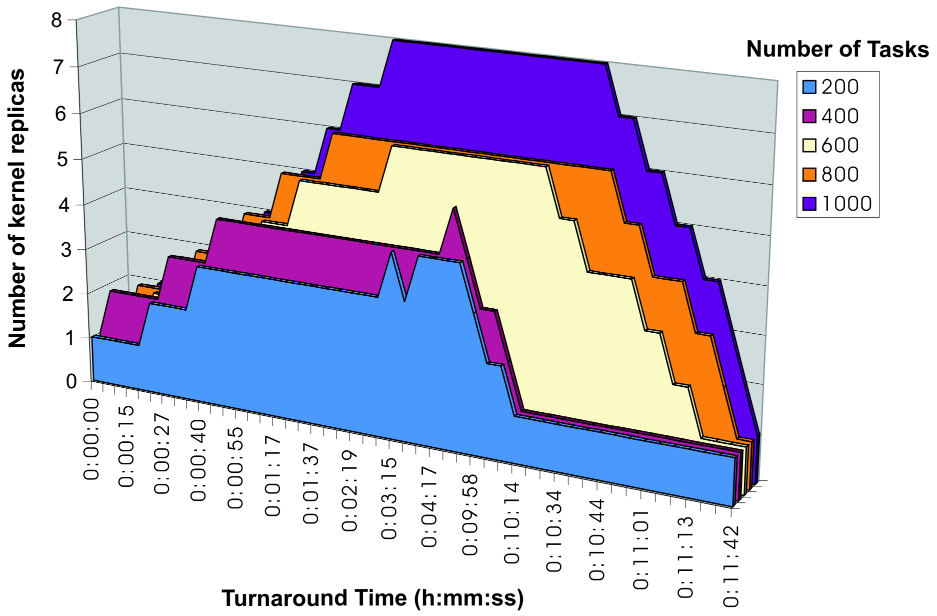


Fig. 10. Automatically adapting the number of replicas

With this, a trade-off can be identified between minimizing the turnaround time of the processes while optimally using the available resources. Due to the potential variability of the workloads, especially if a virtual experiment has been made accessible through a Web service interface, it is important that the process execution infrastructure is capable of automatically adjusting its configuration in response to the current workload.

Self-configuration can be achieved through an autonomic manager, which automatically adapts the degree of replication of the system to fit a specific workload. This component consists of 1) a basic resource manager, which keeps track of the nodes that can be used to run replicas of the kernel; 2) a performance monitoring component that observes the state of the system at regular intervals, detects imbalances and uses the 3) kernel reconfiguration services, to modify the number of replicas without disrupting normal system operation.

More precisely, the manager observes the aggregate number of tasks waiting to be executed by each replica as well as the number of processes waiting to be executed in the central queue (Figure 6). This value gives an indication of the backlog of the system and if it exceeds a configurable threshold, a new replica is added to the system. Conversely, if this value falls below a threshold, the replica with the least amount of work is disabled and shut down.

Figure 10 illustrates the manager's decisions by indicating when a worker has been added or removed from the computation. The x-axis shows the turnaround time, the y-axis the number of replicas involved in the computation (at least one replica is kept active at all times) and the z-axis represents different workload sizes, going from 200 tasks up to 1000 tasks.

These results show the capability of the autonomic manager to adapt the system to the workload without any human intervention. A limited amount of replicas was used to execute small workloads, whereas an increasingly larger number of replicas was used as the workload size increased. On the one hand, clients benefit from this adaptation as it keeps turnaround times low and stable in spite of different workloads. On the other hand, the virtual laboratory infrastructure can automatically adjust the amount of resources dedicated to execute the client's processes.

6 Conclusion

The paradigm shift from *in-vitro* to *in-silico* research, observed in many scientific disciplines, has resulted in the challenge of building virtual laboratory platforms. While early virtual laboratories consisted of a few applications integrated on the user interface level (e.g. in a browser), today's virtual laboratory environments evolved into a workbench supporting teams of scientists in specifying, running, monitoring and evaluating virtual experiments. Crucial to the success of such platforms is its ability to automate all aspects of a computation to the largest degree in order to make large scale computations manageable.

To this end, in this chapter we have presented the JOpera system, which brings autonomic computing techniques to meet the requirements of virtual laboratories. With it, all components (computing nodes, software tools, middleware infrastructure) that deal with the specification and the execution of a virtual experiment can be integrated using an autonomic platform. This platform combines appropriate mechanisms and strategies to 1) raise the level of abstraction at which virtual experiments can be defined, executed and debugged; 2) mask the complexity of dealing with outages in a distributed execution environment and 3) automatically tune the system's configuration for optimal performance. All in all, thanks to its autonomic computing features, JOpera is a significant step towards the goal of providing scientists with an environment that lets them concentrate on doing science while avoiding to deal with the computer science.

Acknowledgements

The authors would like to recognize the hard work of Reto Schaeppi, Andreas Bur, Oliver Deak, who greatly contributed to the BioOpera and JOpera projects.

References

1. Bioperl. http://www.bioperl.org.
2. D. Abramson, J. Giddy, and L. Kotler. High performance parametric modeling with Nimrod/G: killer application for the global grid? . In *Proceedings of the 14th International Parallel and Distributed Processing Symposium (IPDPS 2000)*, pages 520–528, Cancun, Mexico, 2000.

3. A. Alizadeh, M. Eisen, R. Davis, et al. Distinct types of diffuse large B-cell lymphoma identified by gene expression profiling. *Nature*, 403(6769):503–511, 2001.
4. G. Alonso, W. Bausch, C. Pautasso, M. Hallett, and A. Kahn. Dependable Computing in Virtual Laboratories. In *Proceedings of the 17th International Conference on Data Engineering (ICDE2001)*, pages 235–242, Heidelberg, Germany, 2001.
5. W. Bausch, C. Pautasso, R. Schaeppi, and G. Alonso. BioOpera: Cluster-aware computing. In *Proceedings of the 2002 IEEE International Conference on Cluster Computing (CLUSTER 2002)*, pages 99–106, Chicago, IL, USA, 2002.
6. B. Boeckmann, A. Bairoch, R. Apweiler, et al. The Swiss-Prot protein sequence data bank and its supplement trEMBL in 2003. *Nuc. Acids Res.*, 31:365–370, 2003.
7. F. Bushmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *A system of patterns*. Wiley, August 1996.
8. G. Cannarozzi, M. Hallett, J. Norberg, and X. Zhou. A cross-comparison of a large gene dataset. *Bioinformatics*, 16:654–655, 2000.
9. S. Chervitz. Comparison of the complete protein sets of worm and yeast: orthology and divergence. *Science*, 282:2022–2028, 1998.
10. I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputing Applications*, 11(2):115–128, 1997.
11. 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.
12. T. Heinis, C. Pautasso, O. Deak, and G. Alonso. Publishing Persistent Grid Computations as WS Resources. In *Proc. of the 1st IEEE International Conference on e-Science and Grid Computing*, Melbourne, Australia, December 2005.
13. IBM and Apache Foundation. *Web Service Invocation Framework (WSIF)*, 2003. `http://ws.apache.org/wsif/`.
14. M. J. Litzkow, M. Livny, and M. W. Mutka. Condor—A Hunter of Idle Workstations. In *Proceedings of the 8th Int'l Conf. on Distributed Computing Systems*, pages 104–111, 1988.
15. B. Ludaescher and C. Goble, editors. *Special Section on Scientific Workflows*, volume 34 of *SIGMOD Record*. September 2005.
16. C. Pautasso. JOpera: Process Support for more than Web services. `http://www.jopera.org`.
17. C. Pautasso and G. Alonso. From Web Service Composition to Megaprogramming. In *Proceedings of the 5th VLDB Workshop on Technologies for E-Services (TES-04)*, pages 39–53, Toronto, Canada, August 2004.
18. C. Pautasso and G. Alonso. The JOpera Visual Composition Language. *Journal of Visual Languages and Computing*, 16(1–2):119–152, 2004.
19. C. Pautasso and G. Alonso. Flexible Binding for Reusable Composition of Web Services. In *Proceedings of the Workshop on Software Composition (SC 2005)*, Edinburgh, Scotland, April 2005.
20. B. Snel, P. Bork, and M. Muynen. Genome phylogeny based on content. *Nature Genet.*, 21:108–110, 1999.
21. P. Stuedi and G. Alonso. Connectivity in the presence of shadowing in 802.11 ad hoc networks. IEEE Wireless and Communications and Networking Conference (WCNC), 2005.