JOpera: A Toolkit for Efficient Visual Composition of Web Services

Cesare Pautasso and Gustavo Alonso

ABSTRACT: Web services are attracting attention because of their ability to provide standard interfaces to heterogeneous distributed services. Standard interfaces make it possible to compose more complex services out of basic ones. This paper tackles the problem of visual service composition and the efficient and scalable execution of the resulting composite services. The effort revolves around the JOpera system, which combines a visual programming environment for Web services with a flexible execution engine that interacts with Web services through the simple object access protocol (SOAP), described with Web services language description (WSDL) and registered with a universal description discovery and integration (UDDI) registry. The paper explains the syntax and implementation of the JOpera Visual Composition Language (JVCL) and its ability to provide different quality of service (QoS) levels in the execution of composite services.

KEY WORDS AND PHRASES: JOpera, scalable process execution, visual programming languages, Web service composition.

Web service technologies may not solve every interoperability problem, but they show great promise for reducing the complexity of integrating heterogeneous software components over the Internet. They provide standard protocols for invoking (SOAP), describing (WSDL), and discovering services (UDDI) published on the Internet in a platform-, programming language–, and vendor-independent manner [34, 43, 44]. A most natural evolution of these technologies concerns the ability to compose complex Web services from basic ones [12]. Especially in e-business scenarios, researchers have proposed many ways to standardize the integration of Web services into business processes [15, 25, 26, 30, 39, 45, 46]. None of these is yet well established in practice, although the Business Process Execution Language for Web services (BPEL4WS) specification seems to be ahead at the moment [25, 41].

The standardization efforts behind Web services, and the increasing number of aspects that are being formalized, open the possibility of reducing the development costs and complexity of large distributed information systems. In particular, a visual approach to Web service composition may very well be a suitable complement to existing XML-based composition standards. The use of a visual programming language may help to bridge the different standards and will certainly make Web services much more designer-friendly. In such a system, the order of invocations of service, data exchanges, and failure-handling behavior could all be specified with a simple visual syntax. Having a visual programming language for Web service composition is not enough,

This work was supported in part by grants from the Hasler Foundation (DISC Project No. 1820) and the Swiss Federal Office for Education and Science (ADAPT, BBW Project No. 02.0254 / EU IST-2001–37126).

International Journal of Electronic Commerce / Winter 2004–5, Vol. 9, No. 2, pp. 107–141.
Copyright © 2005 M.E. Sharpe, Inc. All rights reserved.
1086-4415/2005 $9.50 + 0.00.

however. The visual programming environment also needs a set of tools for efficient, scalable, and reliable execution of such composite applications.

This paper presents the JOpera system, a visual programming environment and execution engine for Web service composition [35]. JOpera is a research platform used to explore several different aspects of software composition at the Information and Communication Systems Research Group of the Swiss Federal Institute of Technology in Zurich.

JOpera brings visual programming to the composition of Web services by means of the JOpera Visual Composition Language (JVCL), in which services are composed into processes defined by a simple visual notation based on data and control flow graphs [36]. This visual approach can be very useful both for the manual programming of such processes and for the automatic support of semantic-based Web service composition. Given the complexity of real business processes [10], providing an understandable representation of the composite Web services that are built automatically based on semantic annotations is a key factor in the success of such environments.

The discussion in this paper shows how the JVCL language can be mapped to the BPEL4WS specification, and vice versa. In respect to the efficient and scalable execution of composite Web services, it describes a novel architecture for a process-support system in which a flexible kernel for process execution can be tailored to provide different quality-of-service guarantees. This is achieved through a set of well-defined abstractions for storing the state of a process, propagating events, and executing tasks. Switching between different implementations of these abstractions facilitates the achievement of different quality-of-service levels in terms of both scalability and reliability.

This flexible architecture lets the replication be applied to any system component in order to increase the throughput of the overall system. Unlike most workflow systems, JOpera achieves high performance execution by compiling process descriptions into executable code rather than interpreting them with a generic navigation algorithm.

The JOpera approach is validated in this paper with an extensive set of experiments that systematically compare the performance of different system configurations.

Web Services Composition

The description of the JVCL will be preceded by the presentation of a model for building applications by composing different Web services. The model is used both to motivate the language's design and to give an overview of its major features.

Web Services

Web services technologies provide open standards for interactions among heterogeneous applications running on different platforms across the Internet. XML-based mechanisms have been standardized for describing service inter-

faces (WSDL), publishing and discovering services (UDDI), and invoking them over different communication protocols (SOAP) [34, 43, 44].

Once it is possible to interact with individual services, the ability to compose and describe relationships between basic services becomes important [2]. Furthermore, a single Web service may export multiple operations that need to be invoked following a certain interaction pattern. Several terms to denote these ideas have been proposed: choreography (WSCI), orchestration (BPEL), automation (XLANG), flow (WSFL), coordination (WS-Coordination), collaboration (BPSS), and conversation (WSCL) [15, 25, 26, 30, 39, 45, 46].

In the present case, the term "composition" is preferable because the focus is on developing applications by composing existing and reusable building blocks. Not all of these blocks need to be Web services. The JOpera process-execution kernel is flexible enough for the integration of components accessible through a wide variety of invocation mechanisms. For example, a component can represent the execution of a command in a UNIX shell, a remote procedure call, a Java remote method invocation, a job submitted to a cluster batch-scheduling system, or a request to access a given Grid service [7]. As stated by the jbpm.org project [4]:

> BPEL4WS, BPML, WSCI are all "workflow standards" based on Web services. While Web services are cool and is [*sic*] a nice buzzword, we think it is a big limitation to restrict a workflow engine to only Web services. There are so many other nice protocols like HTTP, RMI, CORBA, EJB, TCP/IP, UDP/IP, JMS. . . . As a workflow engine is mostly used for enterprise application integration, it seems ridiculous for an engine to support only Web services and ignore all other protocols. In our opinion, a workflow engine should communicate with each system in the technology that is most appropriate and not force the development and maintenance of Web service wrappers.

The present authors are in full agreement with this view, and have also designed the JOpera system to support components that can be accessed through a variety of protocols, including, but not limited to, those compliant with Web services. As a result, the developer of a composite application is not restricted to the use of components accessible only through the SOAP protocol. Moreover, in the case of components supporting several protocols, the system can use the one that is most appropriate in terms of convenience, performance, security, and reliability.

Although this should be kept in mind, the exposition in the rest of the paper, for the sake of simplicity, will focus on components modeling individual calls to Web services and will treat the terms "program," "component," and "service" as synonyms.

Composition Through Processes

The notion of process is used to model the composition of independent but related software components. A *process* consists of a set of tasks that can represent either a service invocation (activities) or a call to other processes (subprocesses). All the information necessary to instantiate and execute a task is derived

at runtime to support a form of *late binding,* meaning that the actual implementation of a service is located at the latest possible moment, based on the constraints imposed on the task.

In general, a task involves the execution of an operation that may require some input data and may produce some output results. Processes exchange data with other processes or with the user by means of input and output parameters. A process includes a data flow graph to describe the connections between the input and output parameters of its tasks. From the data flow graph of a process, one can derive a control flow graph defining the partial order of execution of the tasks of a process. Like a data-driven data flow language, a task cannot be started until all of its data dependencies are satisfied [22]. Unlike traditional data flow models, an explicit description of the control flow of a process is included. This is useful for an overview of the order of execution of the tasks and allows users to specify additional control dependencies that cannot be derived from the data flow. As will be described later, the development environment enforces the appropriate editing constraints to keep the two graphs synchronized.

The services and processes to be composed as the tasks of a process are chosen from a library of existing, reusable components. JOpera provides a set of tools with which to manage the component library. For example, external services can be looked up in UDDI registries, and their interfaces can be automatically imported. This is done by translating the corresponding WSDL descriptions into JVCL visual notation: Each service's operation is imported as a separate activity whose input and output parameters match the corresponding parts of the request and response messages.

JOpera Visual Composition Language

Because a process and the relationships between its tasks and parameters can be modeled using control and data flow graphs, the structure of a process can be described directly with a visual programming language instead of using a textual syntax. The discussion in this section informally defines the visual syntax of the JVCL language used to compose a set of Web services into a coherent application. This graphical notation is used during the development phase to design the processes and, augmented with color-coded information, during the monitoring phase to track the state of execution of the processes at runtime.

A process is programmed by drawing a set of directed graphs. The nodes of a graph represent tasks and their data parameters. The edges of the graph represent control flow or data flow dependencies. As shown in Figure 1, a task is drawn as a box with its name inside. An activity box has a single border; boxes for subprocesses have a double border to indicate nesting.

Data Flow

Each task has a set of input and output data parameters. An input parameter is used to pass information to the task when it is started. An output parameter

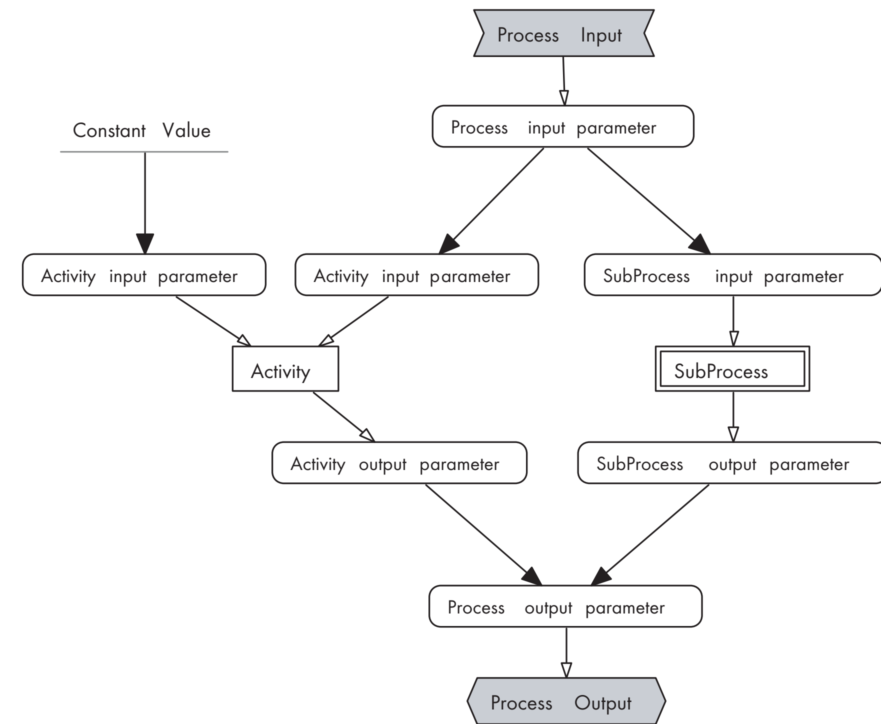


Figure 1. Sample Data Flow Syntax

is filled with the information returned from the task once it is finished. This property is reflected in the graph with incoming edges connecting input parameters to their tasks and outgoing edges connecting tasks to their output parameters (*see Figure 1*). These edges are not removable, because a parameter box cannot exist without its task.

Data parameters may contain values of any data type encoded as a string. Optionally, the user may associate a type identifier to a parameter and turn on the static type-checking facilities of the development environment. This allows the user to reject connections between parameters of mismatching data types.

The two gray shapes in Figure 1 represent the input and output interfaces of a process to which the corresponding parameters are attached. The input parameters can be initialized when the user starts the process or can receive their data from the calling process. The output parameters can be read as soon as the process has finished its execution. The input and output parameters of a process are displayed linked to two separate boxes in order to improve the readability of the diagram.

In the case of activities representing a call to a Web service, each input parameter corresponds to a *part* of the SOAP request message, while each output parameter is extracted from the SOAP response. Thus, the information

exchanged with the Web service can be modeled in detail, as opposed to dealing only with entire SOAP messages.

Data Flow Bindings

The data flow relationships between parameters define how data is transferred between tasks—a data flow *binding* is represented as an edge going from an output parameter box of a task to an input parameter box of another task. As shown in Figure 1, constant values can be bound to the input parameters of tasks.

Multiple data bindings to and from the same parameter are allowed. One output parameter box can be linked to multiple input boxes. On the other hand, edges from multiple output boxes of different tasks that converge on the same input box are only useful in the case of a loop or when the corresponding control flow merges from two or more alternative execution paths. The JOpera runtime environment uses a *last writer wins* semantic—the value of the input box will be copied from the output box attached to the task finishing last.

The development environment enforces a set of editing rules that prevent the user from drawing invalid bindings and explain with an error message why an edge is not allowed. For example, data always flow from the output parameters of tasks to the input parameters. The input parameters of processes can only be connected to the input parameters of tasks, and the output parameters of processes can receive data only from the output parameters of tasks. The same constant can be connected to multiple input parameters, but an input parameter bound to a constant value cannot have any other incoming data flow edge. Thus, the consistency of the data flow graph is maintained at all times.

System Parameters

In addition to the user-defined data flow parameters, each task has a set of system parameters and properties that can be used for a variety of purposes. More precisely, the system output parameters contain metadata about the execution of the process and can be used to access them from within the process. The system input parameters can be used to control the execution of the individual tasks. The same visual syntax applies to the system parameters and the user's data flow parameters. However, the former are colored gray and their names always begin with the SYS prefix. Connections between user and system parameters are supported.

Figure 2 shows some examples. In the case of activities representing Web service calls, the two system parameters called *xmlin* and *xmlout* give direct access to the XML content of the SOAP request and response messages. The host and priority system parameters can be used to specify additional scheduling constraints. The host parameter may be used, for example, when composing a stateful conversation out of a set of operations belonging to the same

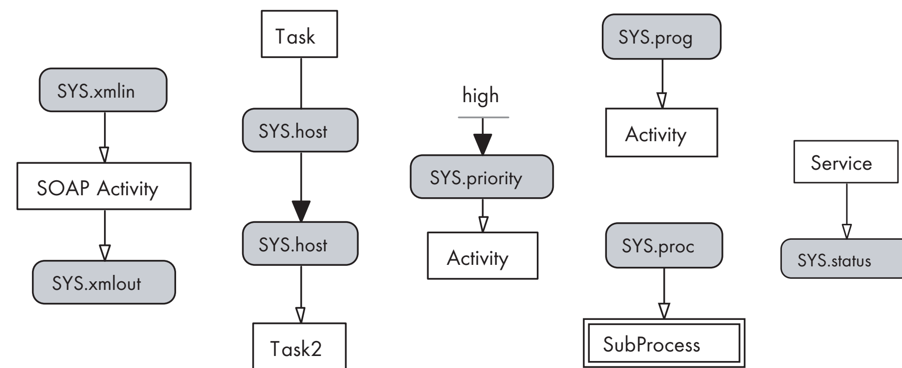


Figure 2. Example of System Parameters

Web service. In this case, the first operation may be scheduled to contact any of the available service providers, but the rest of the operations should be forced to interact with the same service provider as the first one. This scheduling constraint can be visually modeled by connecting the host system parameter of the first task to the same parameter of the others. The priority system parameter may be used to manually raise (or lower) the scheduling priority of tasks located on the critical path of the process. System parameters can also be used to support late binding of tasks to services. The choice of which service (or process) to invoke when executing a certain activity (or subprocess) is dynamically based on the value of the *prog* (or *proc*) system parameter. The system output parameter called *status* can be used to detect why the invocation of a task has failed. In the case of Web services, a failure can occur, for instance, because the service provider cannot be contacted or a fault message has been returned. Different exception-handling tasks can be executed depending on the actual cause of the failure.

Control Flow

Control flow defines the partial execution order between the tasks inside a process. Each process has one control flow graph, with tasks as nodes and control flow dependencies as directed edges.

A control flow edge from node A to node B is used to show that task B cannot start until task A has reached a specified execution state. Valid states are either finished (by default), failed (when an error occurs), or aborted (after a user kills the task). Figures 3 and 6 show examples of control flow graphs.

By definition, a data flow connection between two tasks implies a control flow dependency. This is because it is not possible to transfer data from task A to task B unless task A has successfully finished execution and B has not yet been started. It follows that a subset of control flow dependencies can be derived from the data flow specification. Extra control flow dependencies can be introduced directly in the control flow graph to model constraints that are not explicit in the data flow.

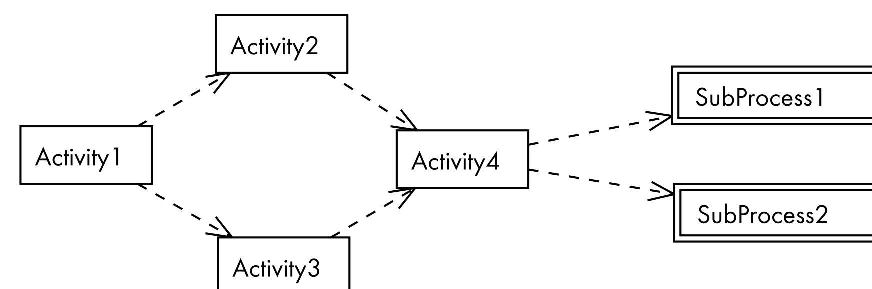


Figure 3. Sample Control Flow Syntax

If there is more than one incoming control flow edge to a node C, the default execution semantic is to *and* all dependencies. For example, if there is a dependency coming from service A and another from B, task C cannot be started until tasks A and B have both finished. One exception to this rule is when the incoming connector is part of a loop in the graph, in which case the semantic is to *or* the loop dependency with the others.

Alternative execution paths are modeled by associating a start condition to each node. This is a Boolean expression referencing the value of certain data parameters. A task can only be started when this condition evaluates to true. Currently, start conditions may be specified only textually as one of the task properties.

Failure-handling behavior is specified in the control flow graph by using connectors that are triggered by the failure of a task. An exception-handling task may be added to a process by drawing such connectors from one or more tasks to it. With start conditions applied to the appropriate system parameter, it is possible to discriminate between different types of failures and activate the appropriate exception handler. By setting a link from the exception handler back to the failed task, it is possible to retry its execution after the exception handler has finished.

Iteration

Supporting iteration in a language based on the data flow paradigm requires the introduction of auxiliary constructs [31]. The authors' approach relies on two constructs with different degrees of generality. A special data flow connector is used to perform sequential parallel operations on lists. Explicit arbitrary loops are being used experimentally in the control flow graph.

List-based loops can repeat the same operation on a given set of values. When no data dependencies hold, the operation can be performed in parallel. Otherwise, the task must be applied sequentially on each value. To achieve this, a pair of special data flow connectors, called *split* and *merge*, is introduced. As in other graph rewriting schemes [8], the overall effect at runtime is to replicate a set of nodes for each value of the input parameter list (*see Figure*

8). In the case of a sequential split connector, the appropriate control flow dependencies between the tasks of the sequence are automatically inserted when the loop is unrolled. If the multiple instances of the task produce output, the merge connector can be used to conveniently concatenate it into a single parameter when the execution of the replicas has completed. In cases of multiple incoming split operators on the same task, the execution of the task will fail if the lists produced by the split operators have different numbers of elements.

Arbitrary cycles in the control and data flow graphs may be used to explicitly model loops in the execution of a process. To avoid endless iteration, the user should assign the correct conditions to enter and exit the loop.

Recursion is another way of modeling repeated behavior. In the simplest case, this can be achieved with a subprocess calling the container process. This enables the tasks composing the process to repeat as long as the condition associated with the subprocess holds true.

BPEL Mapping

The discussion in this section shows the extent to which it is possible to map the authors' visual composition language to the Business Process Execution Language for Web services (BPEL), an emerging XML-based specification for Web service composition, and vice versa [25]. The main goal of such mapping is to be able to use the JOpera platform for visually composing Web services into processes that can later be translated into BPEL or any other equivalent specification for external execution. Conversely, a BPEL document can be imported into JOpera to take advantage of its scalable execution facilities and visual monitoring environment.

Mapping to BPEL

The components of a JVCL process can be accessed using various mechanisms, some of which are not compatible with SOAP/WSDL. To keep the mapping feasible, it will be assumed either that all the tasks of a process represent Web service invocations or that Web service wrappers for the other classes of components can be readily provided. The wrapping could be done automatically as part of this mapping procedure or manually in a separate step.

Partners

For each of the JVCL activities invoking a Web service, a BPEL *partner* is created that contains a service link corresponding to the activity's program, and a BPEL *invoke* activity is also prepared. For each of the JVCL subprocesses, a *link* is created to the JOpera systems where the process is accessible, or, alternatively, a new *scope* is added to the BPEL *process* to make the final BPEL document self-contained.

Control Flow

Given the arbitrary topology of the JVCL control flow connections, it is not always possible to reduce JVCL to the block-structured control flow description of BPEL. However, the control flow graph of a JVCL process can always be mapped to a single BPEL *flow* activity composed of all the tasks of the process, with a direct translation of the dependencies between the service invocations. In cases where control flow dependencies are used to model exception-handling, specific BPEL constructs can be employed. In a case where loops in the JVCL control the flow graph, they can be detected and mapped to a BPEL *while* block.

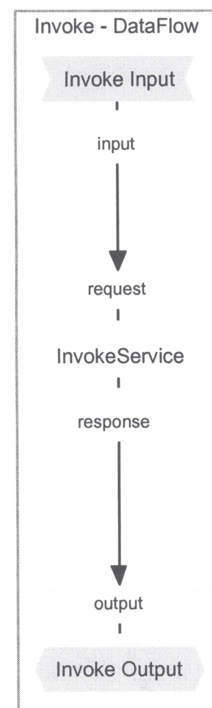
Data Flow

In BPEL, unlike JVCL, the flow of information between the services is not explicitly modeled with a data flow graph. Instead, global variables (or containers, depending on the version of the specification) are used as temporary storage for the messages exchanged by the services, and XPath expressions are used to refer to individual data elements of the messages. To map the data flow graph of a JVCL process, a BPEL *variable* is created to store the request/response messages of each service, and a BPEL *assign* activity is inserted before and after the service invocation represented by the BPEL *invoke* activity for each data flow connection in the graph. This assignment activity contains the XPath expressions used to access the individual JVCL parameters (or message parts). As an alternative, a BPEL *variable* can be added for each JVCL data flow parameter. An example of a basic data flow mapping is shown in Figure 4.

A few of the JCVL constructs have no equivalent construct in BPEL. In addition to explicit control flow loops, JVCL offers iteration through list-based split/merge data flow operators. It is unclear how this could be mapped to an existing standard BPEL construct. Furthermore, the system parameters of JVCL—which, for example, give access to metadata about the process and its tasks, and can be used to specify the late binding of a service interface to its implementation—cannot be mapped to standard BPEL expressions.

Mapping from BPEL

BPEL mixes elements of two different types in the same process-modeling language. On the one hand, there are structural constructs for modeling the control flow and data flow of a process. On the other hand, there are several different basic activities used to model the synchronous and asynchronous invocation of Web services, as well as to handle of events and alarms. Elements of the first type can be mapped to constructs of the JVCL language, whereas those of the second type cannot be mapped directly but are implemented using a library of BPEL components.



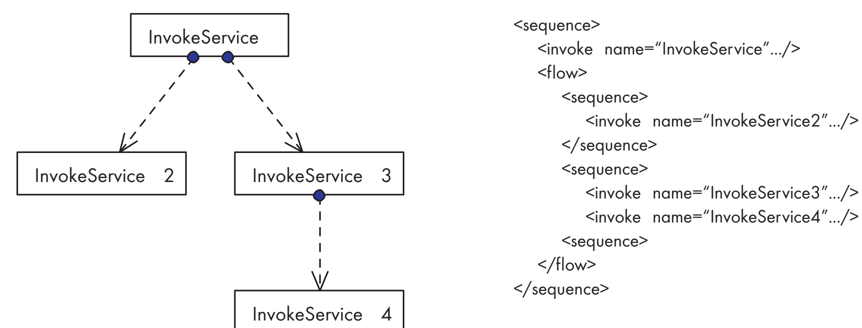

```
<process name="Invoke">
  <variables>
    <variable name="input"/>
    <variable name="output"/>
    <variable name="request"/>
    <variable name="response"/>
  </variables>

  <sequence>
    <receive name="receiveInput"
             portType="tns:Invoke"
             operation="initiate"
             variable="input"
             createInstance="yes">
    </receive>
    <assign>
      <copy>
        <from variable="input" part="parameters"
              query="//value"/>
        <to variable="request" part="parameters"
            query="/Service/value"/>
      </copy>
    </assign>
    <invoke name="InvokeService"
            portType="tns:Service"
            operation="Service"
            inputVariable="request"
            outputVariable="response">
    </invoke>
    <assign>
      <copy>
        <from variable="response" part="parameters"
              query="//result"/>
        <to variable="output" part="parameters"
            query="/onInvokeResult/result"/>
      </copy>
    </assign>
    <invoke name="replyOutput"
            portType="tns:InvokeCallback"
            operation="onResult"
            inputVariable="output">
    </invoke>
  </sequence>
</process>
```

Figure 4. A Process with a Single Web Service Invocation Represented Both in JVCL and in BPEL
The corresponding parts of the process are shown side by side.

Control Flow

As the BPEL control flow is expressed using both block and graph structures, it is always possible to map it to a purely graph-based model (*see Figure 5*). Thus, BPEL constructs like *sequence*, *flow*, *pick*, *while*, and *switch* can be replaced by a corresponding combination of control flow dependencies and conditions.

BPEL-structured exception-handling (based on *throw*, *catch*, and *catch all* activities) can be mapped to JVCL by introducing rule-based exception handlers




```
<sequence>
  <invoke name="InvokeService"…/>
  <flow>
    <sequence>
      <invoke name="InvokeService2"…/>
    </sequence>
    <sequence>
      <invoke name="InvokeService3"…/>
      <invoke name="InvokeService4"…/>
    <sequence>
  </flow>
</sequence>
```

Figure 5. A Process Invoking Multiple Web Services Shown Both in JVCL and in BPEL
There is a clear correspondence between the block-based structure of the BPEL representation and the control flow graph of the JVCL.

and an ad hoc task in the BPEL library that always fails and is used to represent the *throw* activity.

Data Flow

The mapping of the data flow of a process is not so straightforward, given the use of global variables (or containers) and arbitrary *assign* activities in BPEL. There are two possibilities: Either an *assign* activity can be mapped to a direct data flow connection between a pair of JVCL parameters or an explicit task can be added that runs the XPath expression contained in the *assign* activity and accordingly transforms the input into the output parameter.

Messaging

BPEL *invoke* activities can be directly mapped to JVCL activities with a reference to a program representing the corresponding service invocation. Mapping BPEL activities like *send*, *receive*, and *wait* can all be done by using JVCL tasks of the BPEL library, for which the corresponding functionality is implemented in the BPEL subsystem of the JOpera kernel.

Events

A similar approach is used with the *onAlarm/onMessage* constructs. In this case, a process (or part of a process) could be set up as follows: There is a task that corresponds to the *onAlarm/onMessage*. This task terminates its execution on receipt of a message or on the occurrence of an alarm. The block of actions to be carried out when such an event occurs is translated as before, with the additional dependency from the task corresponding to the original *onAlarm/onMessage*.

Visual Development Environment

The JOpera visual process development environment provides an integrated tool kit for managing the whole life cycle of a process. This begins with the program library, where Web services can be imported as reusable components. The user can search the library, select a set of services, and drag and drop them into a process. Then the data flow graph needs to be specified. This operation is partially automatic, because the editor can automatically bind parameters with matching names and make recommendations based on the parameter types. Manual intervention is only required to resolve ambiguities and connect parameters that could not be automatically matched, but the user can view and edit the control flow graph in order to get an overview of the order of execution of the tasks and add additional constraints. The development environment is responsible for keeping the two graphs synchronized: Whenever a new data flow binding is established, the necessary control flow dependency is added. Conversely, when a control flow dependency is deleted, all of the corresponding data flow bindings are removed. Optional warning messages may notify the user of the consequences of these actions, which otherwise are carried out in a transparent manner.

Once all the services have been connected, the process is compiled and uploaded to a JOpera runtime environment for execution. A user interactively monitoring a running process can watch its progress as indicated by the colors of the task boxes and can click on the parameters to inspect their contents. The user can interact with a running process or its tasks, and can abort, pause, continue, and restart them at will. Once a process has completed its execution, the user can access the content of all the parameters and measurements of the execution time of each task, until the process is explicitly deleted from the system.

Example

A process used to compare the prices of books sold at various Internet stores will serve as an example. The process receives an ISBN number as input and returns a URL as output for a report containing the price comparisons for the book. As stores in different countries return prices in their own currencies, the user may specify the currency to be used in the report as an optional input parameter. The process contains the steps needed to perform the currency conversion. The report also contains the book's author and title, retrieved from a library database, and a listing of the top five results returned by a Web search engine looking for the author and the title of the book. This simple example can be used to present the most important features of the JVCL without having to describe too many application-dependent details.

Process BookPrices

Figure 6 shows the control flow graph for the price-comparison process. The process is composed of three activities (Library, GoogleSearch, MergeReport) and one subprocess (QueryBookPrice). As its name suggests, QueryBookPrice

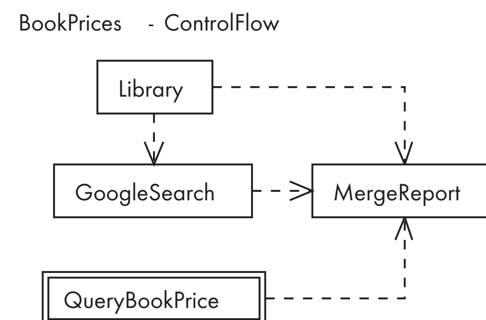


Figure 6. Control Flow Graph of the BookPrices Process

involves contacting a bookstore to inquire about the price of a certain book identified by its ISBN. While this is happening, the Library activity retrieves the author and title of the book. The Web search is started when the library query finishes, and the report is generated when all of the previous tasks are finished.

The data flow graph of this process has been partitioned into two different views to enhance its readability. Figure 7 shows one view with data parameters and bindings of the Library, GoogleSearch, and MergeReport activities. The second view, in Figure 8, shows the data flowing through the QueryBookPrice subprocess. The first view shows one of the input parameters of the process (*isbn*) passed both to the Library and MergeReport activities (*see Figure 7*). Given the *isbn* as input parameter, the Library activity returns the corresponding author and title. These two parameters are passed on to the GoogleSearch activity, which runs a Web search using them as keywords and returns the top five results. The MergeReport activity receives the title, the Web search results, and the author and ISBN of the book. It uses this to generate a report and returns a *url* where it can be found. When the process is finished, this value is returned as the *reporturl* output parameter of the process.

The rest of the data flow can be seen in Figure 8, which shows an example of the parallel split and merge iteration constructs. This allows the process to call in parallel different services having the same interface. Both *isbn* and destination *currency* process input parameters are passed to the processQuery subprocess, which also receives the identifier of the bookshop service to be called and the source currency of the price returned by the service. At runtime, a parallel copy of the processQuery subprocess will be executed for each element found in these two input parameters. In the example, the service and source parameters are bound to constants with a list of four strings that contain service identifiers (BooksCH, AmazonCOM, AmazonDE, BNCOM) and the corresponding currency identifiers (CHF, USD, EUR, USD). Note that a parallel instance of the processQuery subprocess will be started for each pair of parameter values. The prices returned by the parallel instances of the processQuery subprocess are merged into the prices input parameter of the MergeReport activity. Both views show the same data flow connection binding the output of the last activity with the output of the process.

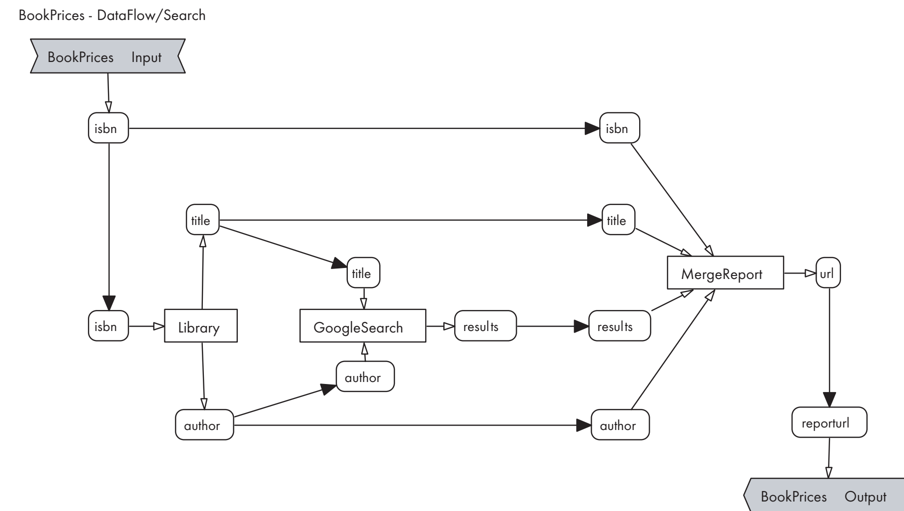


Figure 7. First Data Flow View of the BookPrices Process

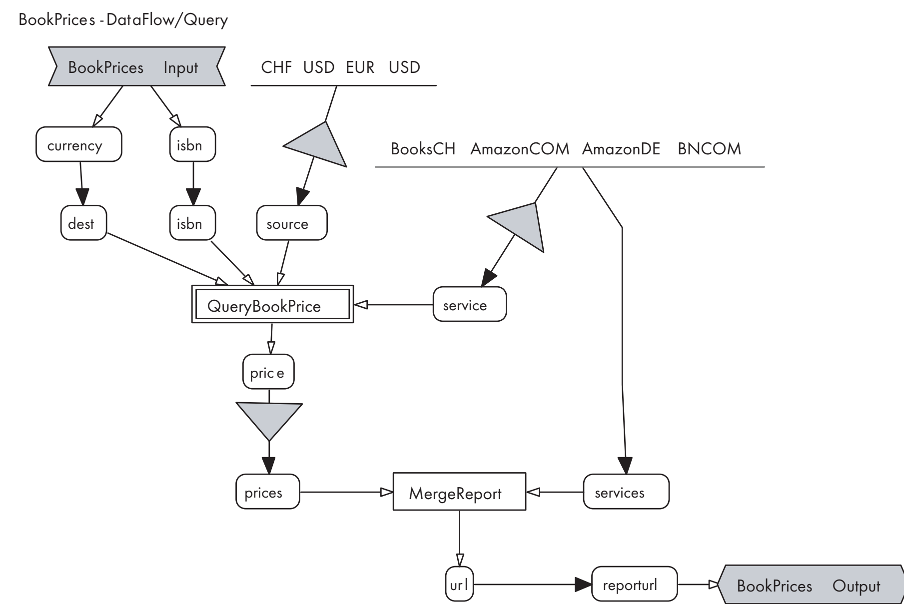


Figure 8. Second Data Flow View of the BookPrices Process with Parallel Split Merge Operators

Process QueryBookPrice

The QueryBookPrice process is called from within the BookPrices process. It contacts two Web services in order to ask the book's price and convert it to the

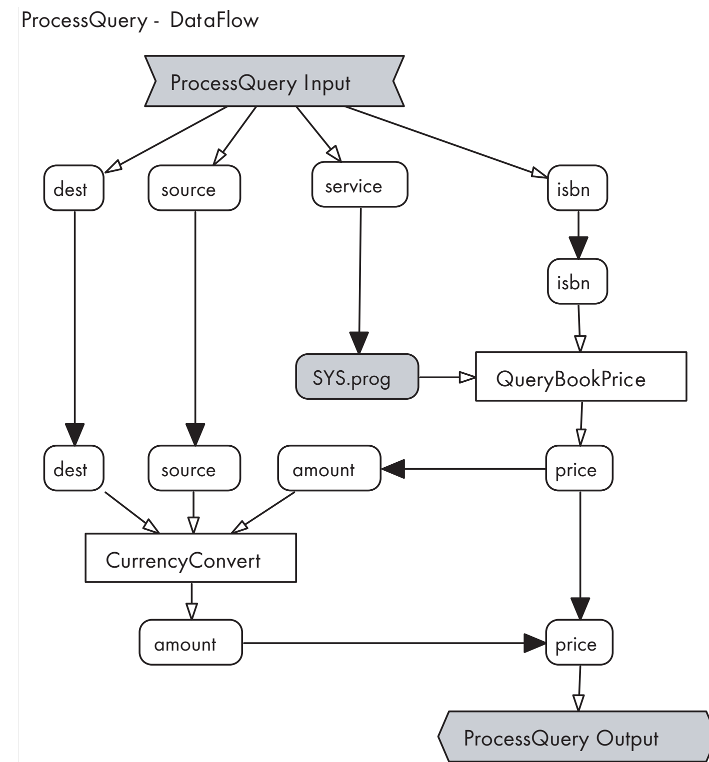


Figure 9. Main Data Flow View of the Query Process

desired currency. Figure 9 shows its data flow graph. This process contains two activities: *QueryBookPrice* and *CurrencyConvert*. The input and output parameters of the process match the ones of the processQuery subprocess. The *isbn* of the book is passed to the *QueryBookPrice* activity. To choose the services to call, the actual service name is assigned to the *SYS.prog* parameter of the activity, resulting in the invocation of the corresponding service. After the query has completed, the resulting price and the source and destination currencies are passed to the *CurrencyConvert* service, which will return the corresponding amount. When the process finishes, the converted price is returned to the caller. Note that the *CurrencyConvert* service is not invoked when the currencies are the same—in this case, the price is returned directly from the result of the query.

JOpera Process Execution Kernel

It is now time to present the architecture of the JOpera process execution kernel. The main purpose of the kernel is to provide an execution platform for the JVCL that can be tailored to different levels of performance.

The relationship between the JOpera visual development environment and the process execution kernel is as follows: Processes defined in the JVCL lan-

guage are created and edited using the development environment. As will be explained in the following section, once the processes are complete and ready to be executed, they are compiled into Java, and the resulting process template plugins are then dynamically loaded into the kernel for execution. Once a new process instance has been started, its execution is managed by the kernel, which may be run independently of the development environment. However, users may connect the development environment to an existing kernel to monitor the activity and progress of their processes.

Before preceding to a description of the architecture of the JOpera kernel, it will be useful to present some background on how the so-called navigation algorithm can be used to execute the description of a process written in JVCL.

Process Navigation

Navigation is the procedure whereby the system determines the set of tasks to be executed next, given the current state of the process and its control flow graph, specifying the partial order of execution of the tasks. The navigation procedure interprets the information in a directed graph, where the nodes represent the tasks and are labeled with their current state, and the edges represent control flow dependencies between the tasks. When a task undergoes a state change, the algorithm proceeds in two steps. First, in order to determine the set of tasks affected by the state change, it follows all outgoing control flow dependencies. Then it evaluates the starting conditions of these tasks to check whether they are ready to be started. This makes it possible, after every change in task state, to determine the set of tasks to be started next.

The foregoing approach is very similar to mapping the process description to a set of event, condition, and action (ECA) rules. State changes of tasks trigger events that cause evaluations of the conditions associated with the set of dependent tasks, and when these rules fire, the actions required to start the tasks can be carried out. More specifically, during navigation, the system mainly performs two types of actions. The first type of action concerns the actual task execution, that is, packing all the necessary information into a job that can be submitted to the scheduler responsible for finding a suitable machine to run the task or the correct provider to which a request message should be sent in order to invoke the service. The second type of action groups operations that access or modify the state information of a process. These include, for example, copying the data from the parameters of one task to another as specified in the data flow of the process, as well as setting metadata values, such as the starting time of a task, or accessing the state of a set of tasks to determine whether they have failed.

Executing Navigation

In practice, it is not necessary to compile a process description into a generic ECA-like representation to be interpreted by the process engine. Instead, the process-navigation algorithm is implemented by building on the idea of mapping

the process description to a program that embodies the specific rules corresponding to the process description, generated using an ordinary programming language. Thus, the language's compiler can be used to produce executable code that can then be dynamically loaded and linked into the kernel's runtime environment to be executed. This approach has the potential to provide better performance. First of all, the executable code is generated in a standard programming language—in the present case, Java—which then is compiled one more time. This way, the process model can be mapped to standard language constructs that can be efficiently executed. Moreover, during code generation, it is possible to analyze the structure of the process and perform optimizations.

The generated program is completely stateless, because it only contains a mapping of the process structure. To navigate over a particular process instance, the program reads its state as input. Therefore, it is possible to navigate over many instances of the same process using the same program code, which only needs to be loaded once. This clear separation between the state of a process and its structure is missing in many systems, and therefore, both types of information need to be loaded from the persistent repository before each invocation of the navigation procedure, incurring unnecessary overhead.

Architecture

The core infrastructure necessary to run the processes written in JVCL is depicted in Figure 10. The kernel of the JOpera process support system includes mechanisms to (1) run the navigation algorithm, (2) schedule and (3) dispatch tasks for execution in the correct environment, (4) access and modify state information about tasks and processes, and (5) exchange event notifications triggering the execution of the navigation algorithm. Note that the navigation algorithm is independent of the actual implementation of these basic facilities, which are described in the rest of this section.

Navigator

The navigator is the kernel component responsible for handling incoming process events, which are generally triggered by changes in the state of tasks or represent user requests. When an event occurs—for example, when the dispatcher has finished executing a task—the navigator runs the algorithm that decides what task should be executed next. The navigator acts as a container for the process plugins, which embody process-specific versions of the navigation algorithm. Upon receipt of events concerning a particular process, the navigator, if necessary, dynamically loads the appropriate plugin.

Task-Execution Scheduler

The task-execution scheduler couples the navigator, which generates task-execution requests, with the dispatcher, which manages the actual task execu-

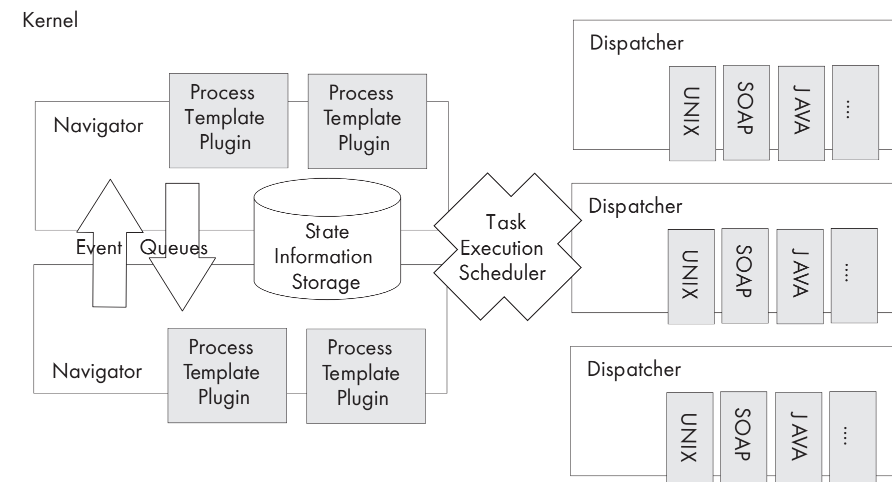


Figure 10. Architecture of a Distributed Kernel

tion. In a distributed kernel, the scheduler receives task-execution requests from a number of navigators and forwards them to a set of dispatchers (*see Figure 10*). This is a key component in the scalability of the system, because its throughput limits the rate at which tasks can be executed.

Dispatcher

If the navigator is in charge of deciding what tasks should be started next, the dispatcher is the component that actually starts executing the tasks by dispatching them to the appropriate execution subsystem. In order to increase the navigator's throughput, the actual task startup operation has been decoupled from the navigation step that triggers it. Thus, the navigator may asynchronously issue multiple task-startup requests to the task-execution scheduler, which queues and forwards them to one or more dispatcher components. Once the dispatcher receives a job, it checks what the job's characteristics are and sends it to a matching execution subsystem. The current prototype contains mechanisms that execute jobs containing Unix programs, SOAP requests [43], Java method calls, XML transformations, and subprocess invocations. Once the job's execution has completed, the dispatcher sends an event encapsulating its results to the navigator.

State-Information Storage

State-information storage is the component responsible for storing state information about the process instances. Its design has been influenced by many requirements, such as performance, reliability, and portability across different data repositories. The component's interface supports only a simple, key-value-based data model, where the key is structured as the following tuple (Process, Task, Instance, Box, Parameter) and is used to uniquely identify a

certain data value across the system. The definition of the key reflects the structure of the information to be stored: A process is composed of a set of tasks, of which there can be many instances. Each process/task instance has multiple parameters that are grouped into three boxes (or logical name spaces): system, input, and output.

The main advantages of this approach are summarized in the following arguments. First, because the information in the key is neutral with respect to the physical locations of the data, the data can be moved transparently to exploit locality and even to replicate them in different physical repositories to improve their availability. Furthermore, the hierarchical nature of the key suggests a natural data-partitioning strategy. Another advantage is that changes and extensions to the data model of the processes' state information do not affect the storage component, because this low-level data representation is mostly independent of the data and metadata that need to be stored [1]. Finally, as shown in Figure 11, the data layer can be implemented with a wide variety of mechanisms. These range from centralized memory-based data structures (e.g., hash maps) to traditional forms of persistent storage (e.g., network file systems, relational databases) or distributed storage systems (e.g., Linda-like tuple spaces like TSpaces or JavaSpaces) [11, 16, 29].

Event Queues

The various kernel components communicate by exchanging event notifications managed by event queues. The sources for the events consumed by the navigator components are the user interface, other navigators, and the dispatchers. Events are sent by the user interface in order to start, stop, and, in general, interact with a process instance. The dispatcher notifies the navigator with an event every time a task finishes its execution. Navigators also exchange events, for example, when a subprocess completes its execution and navigation over the calling process, managed by a different navigator, needs to be triggered. The priorities of these three classes of events can be adjusted.

In a distributed kernel, event communication is also quite important in respect to the system's scalability. The authors have been experimenting with several implementations of the event queues, each having different scalability properties.

First, as a reference, a single tuple space server was used in which all the kernel components were connected and exchanged events by writing and taking tuples. As expected, this centralized event queue quickly became a bottleneck, because all the events sent by multiple dispatchers to a set of replicated navigators had to go through it. Therefore, a second design used a multilayered approach that distributed the event queue across all the navigator components. This approach reflected the heuristic that the navigator responsible for handling incoming events should be kept as close as possible to the events themselves. With this approach, dispatchers can directly send "task-finished" events to the appropriate navigator. Events not sent to a specific navigator still go through the central queue, which in this configuration needs to handle relatively less traffic. For example, user-generated process-startup requests

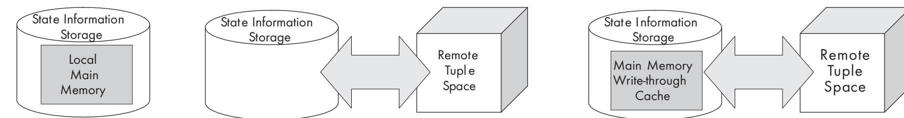


Figure 11. State Information Storage Implementations for the Monolithic Kernel

are queued centrally, and the corresponding "start-process" events can be retrieved by idle navigators. To further reduce the communication overhead, events generated by a navigator that can be processed by the same navigator are kept locally and do not need to be sent over the network.

Parallel Navigation

The navigation algorithm was parallelized based on the observation that every process instance is a fully independent entity. Changes to the state of one instance do not affect other process instances. Therefore, it is possible to partition the system's workload at the granularity level of the process instance and to perform navigation on different, independent process instances in parallel. As a consequence, the navigation algorithm presented here does not need to be changed, because it can be implemented in a thread-safe manner. However, the underlying infrastructure needs to support the concurrent execution of the algorithm, triggered by events concerning independent process instances.

Once the system is capable of performing parallel navigation, it is necessary to deal with such issues as load balancing and fault tolerance. The current prototype supports two different load-balancing strategies: Either process instances can be statically partitioned among different parallel navigators (load sharing), or events and state information can be dynamically moved between different navigators in order to keep the system balanced. Moreover, recovery from failures occurring in the navigator is completely transparent because each navigational step is executed atomically within a transaction, and the state information can be stored remotely and persistently. In fact, under a dynamic load-balancing strategy, process instances belonging to a failed navigator can be immediately assigned to another navigator as soon as the failure is detected.

Deployment Scenarios

This section presents some of the configurations in which the flexible kernel architecture can be deployed (*see Table 1*) and discusses their main performance advantages and disadvantages. Flexibility is important not only for performance reasons, but also because it allows the system to be adapted to different requirements and thus to be deployed in environments and configurations that match specific workload targets. For example, the architecture can be deployed as a lightweight process-simulation engine attached to a process-development tool. Similarly, it can be embedded into standalone Java

	Event queues	State information storage	Task execution scheduler	Dispatcher	Navigator
(a)	Local	Volatile	Local	Single	Single
(b)	Local	Persistent	Local	Single	Single
(c)	Centralized	Volatile	Remote	Multiple	Single
(d)	Centralized	Persistent	Remote	Multiple	Single
(e)	Distributed	Volatile	Remote	Multiple	Multiple
(f)	Distributed	Persistent	Remote	Multiple	Multiple

Table 1. Deployment Scenarios.

applications that require process-enactment capabilities to coordinate the invocation of different components and services. This way, the coordination logic specified as a process can be directly executed within the context of the application. Alternatively, the JOpera kernel can be used as a reliable service-orchestration platform running inside an application server that can also scale to handle very large workloads, using a cluster-based configuration. Flexibility is one of the basic requirements of an infrastructure capable of exhibiting autonomic behavior. To this end, it is important that the system can be reconfigured dynamically following the decisions of an autonomic system controller that monitors the current workload conditions and determines the optimal system configuration [6].

The simplest configuration (a) is a so-called *monolithic kernel*, where one navigator and one dispatcher run on the same machine. The state-information storage, event queues, and task-execution services are implemented using the appropriate main memory data structures. As all the data are kept in the main memory, this configuration trades recoverability from failures for very fast access to the state information. Given its centralized nature, such an architecture does not scale well with large workloads. In addition to the ease of deployment, its main benefit lies in its very low overhead with small workloads.

The next configuration (b) is the monolithic *persistent* kernel. Again, one navigator and one dispatcher run on the same machine, but the storage of the state information is implemented using a remote, persistent data repository. This makes the kernel recoverable, but at the cost of a larger overhead, as can be seen from the results in Figure 12.

The limitations of these centralized configurations pertain to all five main system components: the navigator, the dispatcher, the task-execution scheduler, the state-information storage, and the event queues. If one component is replicated in order to improve its throughput, another component will very soon become a bottleneck. For example, if a set of navigators send task-execution requests to the dispatchers through a centralized scheduler, the throughput of the scheduler limits the rate at which tasks can be executed. Similarly, if the performance of the state-information storage improves, the navigator will be able to produce and consume events at a faster rate, putting a greater burden on the event queues. Thus, one must take care to keep the system well balanced when scaling it up and configuring it with replicated components (*see Figure 10*).

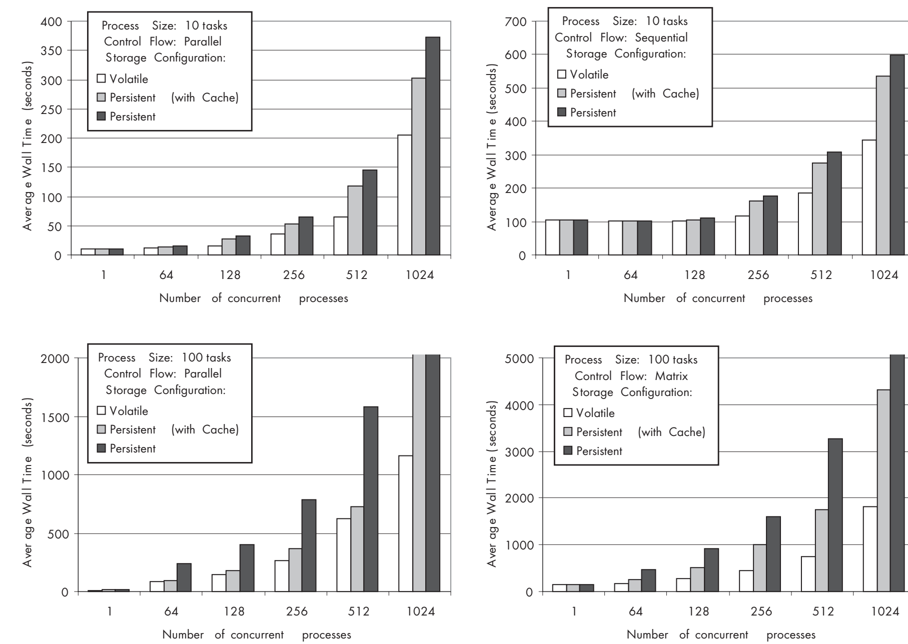


Figure 12. Performance Degradation of a Centralized Process Support System Under Increasingly Large Workloads

The first replicated configuration presented here concerns the dispatcher component. In this case, a single navigator, with (d) or without (c) persistent storage, manages the processes, whose tasks are executed by an increasingly large number of dispatchers. One might expect that the increase in task-execution capacity would make the system capable of handling a larger workload. As the measurements show, this is only true when the task duration is long enough (e.g., more than 10 seconds). For tasks lasting a shorter time, the actual bottleneck lies in the navigator component.

This problem is addressed by configurations (e) and (f), where the navigator component is also replicated, keeping the number of dispatchers and the corresponding task-execution capacity constant. As the measurements indicate, the system's scalability is now bound by the persistent storage service. In fact, using a centralized data repository with an increasingly large number of clients (the navigators) only scales up to a certain limit. Therefore, another configuration (e) was tested in which storage of state information was localized at the navigator. Because of the improved performance of the storage service, the limiting factor shifted to the event-communication service, which also had to be partitioned in order to keep the system functioning.

Measurements

The goal of the experiments was to analyze the performance of a significant subset of the deployment and configuration options described above. First of

all, they indicated the scalability limits of a centralized system that uses the local main memory to implement all the data-storage, event, and job-scheduling services. Then external persistent storage for the state information was added to determine the cost of adding persistence to the system. The dispatcher and navigator components were then replicated, and the changes to the system's throughput were observed.

Hardware Setup

The hardware and software setups for the experiments were as follows: The navigator and dispatcher kernel components were run on a cluster of dual Pentium-III 1000 MHz PCs with 1024 MB of RAM using Java 1.4.1 running on Linux v2.4.17. Each of the three tuple space servers dedicated to state-information storage, task-execution scheduling, and event communication was run on a separate dual Athlon 1.5 GHz with 1024 MB of RAM, Java 1.4.1 on Linux v2.4.19 and used the IBM's TSpaces implementation version 2.1.2 [24].

Workload Description

The behavior of the system was affected by the properties of the workload, as defined by the control variables listed in Table 2. The number of processes indicates the size of the batch of concurrent processes to be executed. The size of a process is the number of tasks composing it. Three different process sizes were used: 1, 10, and 100 tasks. Larger processes require more storage space and generate more jobs and events. The duration of the tasks affects the navigator's throughput, because the longer a task runs, the longer the delay between a job-startup request and the corresponding termination event. During this time, the navigator(s) can process other events or remain idle.

Finally, different topologies of the control flow of the processes generate different patterns of event exchanges. In the case of a process composed of a single task, there are no degrees of freedom concerning the control flow, but as soon as the size of the process increases, it is possible to connect the tasks in different ways. The system was tested with a variety of control flow graphs. Two topologies were used in the case of 10 tasks, one sequential, where the tasks are executed sequentially, and the other parallel, where all the tasks are executed concurrently. The same parallel topology was also used with the larger process composed of 100 tasks. In the case of a large process, the experiment also tested a more complex control flow graph modeling a matrix-like computation.

Measured Variables

Measuring the user-perceived effect of the different configurations was the first subject of interest. This effect was measured by determining how long a process took to complete. More precisely, the average wall-clock time was calculated for all the concurrent processes of a given batch.

Variable	Values
Number of concurrent processes	1, 64, 128, 256, 512, 1024, 2048
Number of tasks	1, 10, 100
Task duration (seconds)	0, 1, 10, 30
Control flow topology	Sequential, Parallel, Matrix

Table 2. Workload Control Variables.

Second, the batch-execution time was recorded for every experiment—this was how long it took to run an entire batch of concurrent processes. In the case of tasks running for 0 seconds, the execution time of the batch of processes was used to compute the average throughput of the system, defined as the number of processed tasks per second. This value indicated the overall speed of the system in performing the operations (navigation, scheduling, running, and result gathering) required to execute the tasks.

Third, in order to observe the system's internal behavior, the state-information storage services were instrumented to measure the time necessary to create the image of a new process instance. This critical step was a potential performance bottleneck, because it is not possible to perform navigation until an instance has been created. Process instantiation was expected to be expensive, because, depending on the size of the process, (a lot of) information about the process, its tasks, and their parameters needs to be written out to the state information storage service.

Results

Reliability Overhead

Analyzing the performance of a centralized process-support system illustrates the limitations of centralized architectures. Such a system is built with a single component dedicated to process navigation, which uses a centralized repository to keep track of the state of the execution of the processes. As has often been observed, centralization and persistence both generate significant overhead in process-support systems under a heavy workload [28, 38]. Figure 12 quantifies the user-perceived behavior of a centralized system while running four different types of processes.

As the results show, the system's response time—that is, the average wall-clock duration of a process—grows as a function of the system's workload, defined as the number of processes running concurrently within the system. Relative to an unloaded system, where only one process at a time is executed, the response time in the worst case grows about 200 times when the workload size is increased a thousandfold. The actual performance degradation depends both on the type and size of the processes and on the specific properties of the system's configuration (*see Figure 11*). First of all, it can be observed that a relative performance improvement can be obtained by sacrificing the reliability of the system. In fact, using the local, volatile memory of the process-navigation

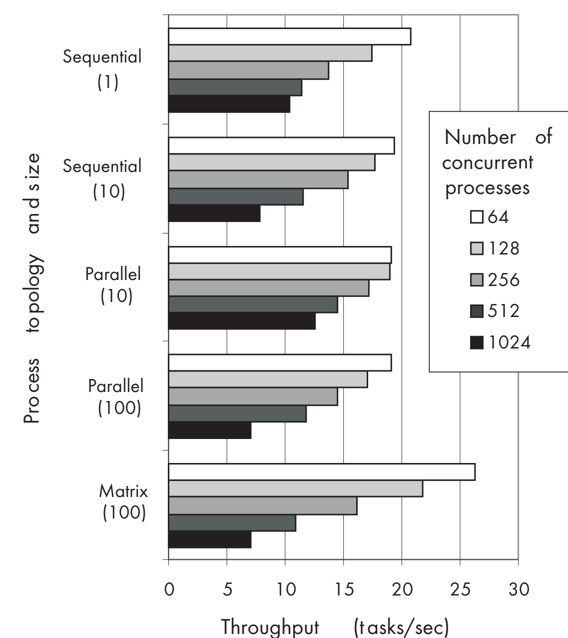


Figure 13. Throughput Degradation of a Centralized Process Support System Under Increasingly Large Workloads

component to store the processes' state information can lead to response times up to 50 percent shorter than the time required to perform navigation over a persistent state.

In an attempt to combine the benefits of both configurations, a write-through cache was added between the navigation component and the persistent storage. As the results indicate, a cache significantly reduces the penalty of using a remote storage service but still has limited scalability.

Monolithic Kernel

In addition to the results in Figure 12 concerning the degradation of the response time of a centralized system under increasingly large workloads, Figure 13 presents the degradation of the corresponding throughputs. This set of measurements was made with a monolithic kernel configured to use volatile storage and up to 64 threads for local task execution—that is, its execution capacity is limited to 64 concurrent tasks.

For all process types, the maximum throughput was achieved when the smallest workload was running. As the number of concurrent processes increased, the throughput decreased to a minimum. The actual degradation rate was dependent on the process topology, as the overhead of navigation is more important for larger and more complex processes.

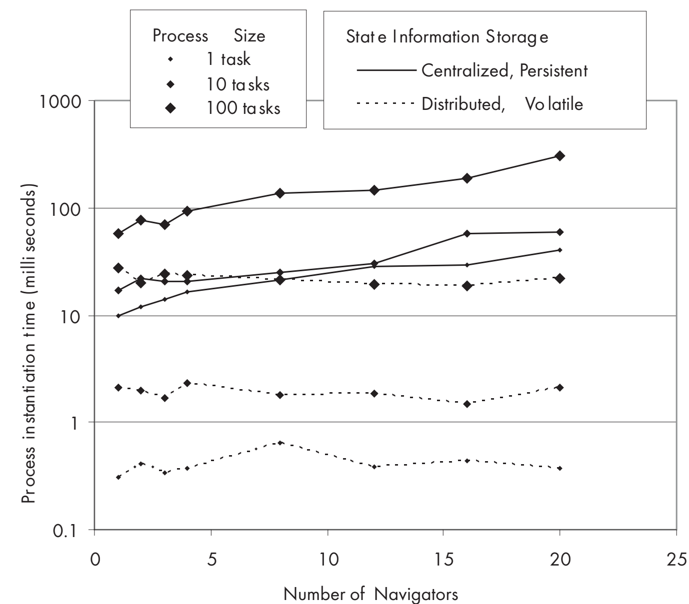


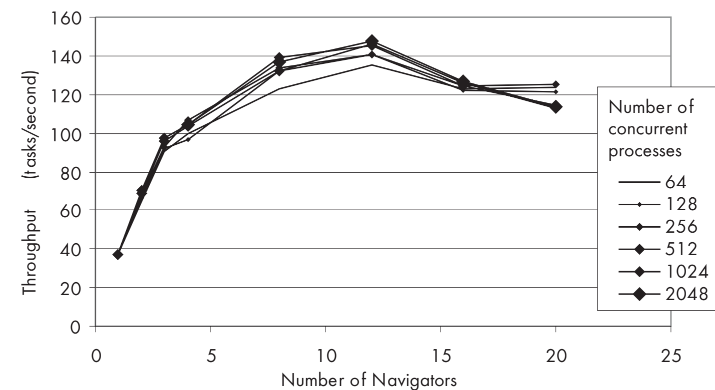
Figure 14. Scalability of the Process Instantiation

Process Instantiation

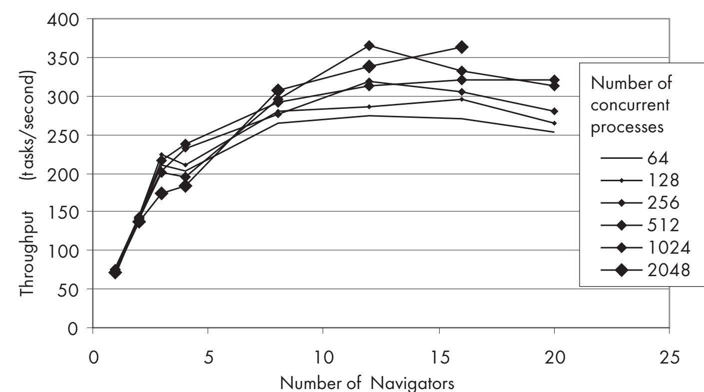
Figure 14 displays the average process-instantiation time as a function of the number of navigators, the size of the process, and the configuration of the state-information storage. As was expected, the instantiation time grew linearly with the process size—the higher number of tasks, the more information about them needed to be written to the data repository. The figure shows two interesting results. Not only is the instantiation time using persistent storage more than one order of magnitude longer than the time with volatile storage, but the volatile storage scales well with the number of navigators, because the process-instantiation time remains constant. On the other hand, the performance of the centralized repository degraded as more navigators stored data in it about their new processes. As has often been suggested, replicating the persistent storage would alleviate this problem [28, 38]. In every case, the instantiation time remained well below the 1-second boundary.

Scalable Process Navigation

Figure 15 shows the average system throughput with processes of 10 parallel tasks run with a variable number of navigators, 25 dispatcher components, and different workload sizes. (a) With persistent storage, the throughput for all workload sizes peaked at 12 navigators at about 140 tasks per second. This was a significant improvement with respect to a centralized system, especially considering that the throughput did not degrade as more and more processes ran concurrently. In (b), the throughput actually improved as the workload size increased, indicating that in the case of volatile storage, the



(a) Processes of 10 Parallel Tasks with Persistent Storage



(b) Processes of 10 Parallel Tasks with Volatile Storage

Figure 15. Scalable Navigation: Average Throughput of the System Using an Increasingly Large Number of Parallel Navigators

performance of the replicated navigator did not saturate. Although the absolute throughput reached about 350 tasks per second, this value was also obtained with 12 navigators, because the centralized task-execution scheduler is the limiting factor of this configuration.

Figure 16 shows the system response time with up to 2,048 concurrent processes of 10 sequential tasks run in the same settings as Figure 15. As the number of navigators for small workloads increased, the batch-execution time approximated the time necessary to run only one process, which was close to 10 or 100 seconds, depending on the duration of the tasks. For larger workloads, the response time still grew linearly with the workload size, but the rate of increase could be controlled by changing the number of navigators. Using volatile storage, the system scaled well up to 20 navigators. Although the absolute response time was twice as high, the penalty for adding persistent stor-

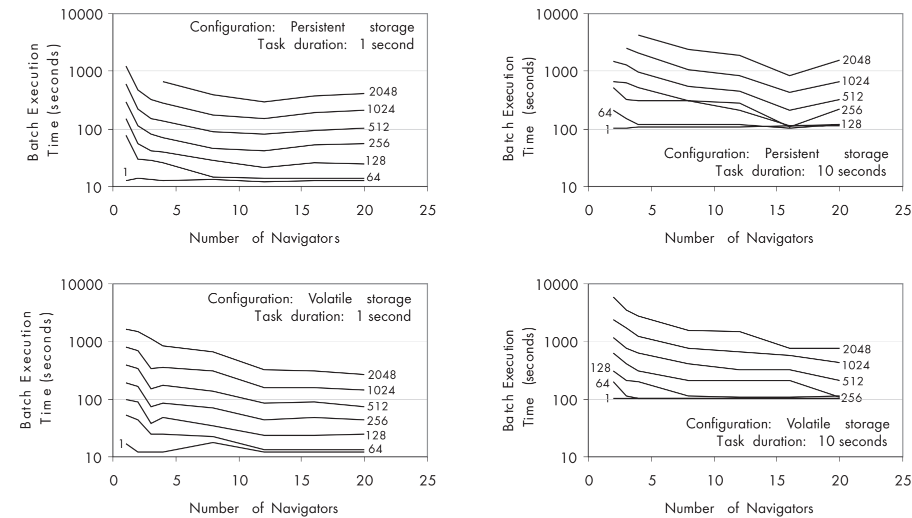


Figure 16. Scalable Navigation: Batch Execution Time of Processes Having 10 Sequential Tasks Depending on the Number of Navigators, Task Duration, and Storage Configuration

age was acceptable, for it showed good scalability up to 16 navigators accessing the same centralized data repository.

Related Work

The idea of developing large-scale applications by composing coarse-grained, reusable component modules was pioneered by Wiederhold, Wegner, and Ceri [47]. Cox and Song have developed a formal model for software based on traditional CORBA, EJB, and COM components, while Muench and Schuerr provide an overview of established component-based visual languages [14, 32]. Gelernter and Carriero present a good argument on the need for a composition "glue" language that is different from traditional programming languages [17].

Browne, Hyder, Dongarra, Moore, and Newton propose a similar, two-step approach in the parallel computing domain [9]. In this case, sequential procedures are first written in Fortran or C and then are composed into a parallel structure using a control-flow-based graphical notation, where the data flow is derived implicitly by matching parameter names [8].

Many researchers have worked on the problem of extending data flow languages with iteration constructs. A survey of their contributions will be found in the paper by Mosconi and Porta [31]. Auguston and Delgado give an example of iteration through vector operators and conditional switches [3].

There are many different commercial products and research projects dedicated to process-management systems, especially in the area of process-modeling languages, with emphasis on flexibility and transactional properties [18, 37, 42].

Many different graphical formalisms have been used as modeling tools in the workflow community. Examples include state charts, used in the Mentor project, or Petri nets and such variations as Object Coordination Nets (OCoN) [40, 48, 49]. However, there is still no well-established visual standard for process modeling.

There has been relatively less research in the area of distributed architectures for scalable process execution. More specifically, scalability has been a common goal to be achieved through different means: replication at the database layer, distribution in the process-execution engine, and decoupled communication through events notification. Only rarely have all of these approaches been followed within the same project.

The idea of building a distributed workflow-enactment system based on event communication and event-condition-action rules has also been proposed in the EVE project [19]. The exchange of event notifications plays an important role in the approach proposed in this paper, whereas the ECA rules are only an intermediate representation that bridges the gap between graph-based models and the corresponding executable code to make them more readily understandable to the user designing the process.

The theme of enhancing the system's fault tolerance and scalability through replication at the database layer was pioneered by Kamath, Alonso, Guenthoer, and Mohan [28]. A scalable strategy for distributing process data among separate databases was proposed in the MOBILE project as a way to replicate the process execution layer [21, 38]. Although the experiment described in this paper compared the performance of a centralized, persistent repository with a distributed, volatile implementation, the subject of replicated storage was not explored any further.

Decentralization is pursued by the MENTOR project, which analyzes process definitions and automatically partitions them among distributed execution sites in order to avoid the bottleneck of a centralized engine [33, 49]. This approach fits well with the requirements of workflows spanning multiple organizations. However, one execution site can become a hot spot when it is involved in the execution of a large number of processes.

Once a distributed process architecture has been designed, load balancing, network congestion, and QoS guarantees become interesting options. Jin, Casati, Sayal, and Shan present a cluster-based workflow-management system focused on quantitative comparison of two different load-balancing strategies [27]. Bauer and Dadam use simulations, in the context of several distributed architectures, to study how different workloads influence the load of the network and thus the scalability of the workflow engines [5]. Gu, Nahrstedt, Chang, and Ward use extensive simulations to validate a composition model with QoS guarantees based on service-overlay networks [20].

The BPEL4WS specification is currently supported by two implementations [25]. Both execution engines are meant to be deployed inside an application server. The Collaxa BPEL server is the most advanced of the two implementations, because it comes with a graphical process designer and debugger [13]. The visual notation employed has a very close mapping to the underlying BPEL document. This is advantageous, because it means that a BPEL document does not need to be edited at the XML level. On the other hand, unlike

the JVCL language, the notation is not abstract enough to be applied to other process-modeling paradigms. The other implementation is the Business Process Execution Language for Web Services Java Run Time (BPWS4J) from IBM, which also includes an editor with minimal visual support [23].

Conclusion

E-commerce applications composed of Web services are one of the most complex distributed applications that can be built today. This is so because they potentially involve interactions and exchanges of information between heterogeneous services distributed across the Internet. With the JOpera system, it is possible to program such applications by simply drawing a graph.

This paper has presented the JOpera Visual Composition Language, a visual programming language for Web service composition. With a simple syntax, the language offers the following features: conditional execution, failure handling, optional type safety, implicit (list-based) and explicit iteration, nesting, and recursion, as well as the visual specification of late binding and scheduling constraints. The JOpera development environment supports the rapid building of processes from a library of existing component services and monitoring of their execution. In addition to developing an integrated set of tools for component library management, automatic layout of graphs, static type checking, process compilation, execution profiling, analysis, and optimization, the authors have successfully tried the system with computer science students developing small application integration projects.

This paper describes the visual composition language and presents a novel architecture for a process-support system kernel that can be used to execute processes written in it. The main innovation of this architecture consists of the ability to transparently tailor the system's performance to different QoS guarantees. By switching between different implementations of basic services, such as data storage, event communication, and job execution, it is possible to deploy the same algorithm for process navigation in a variety of configurations, each with its own performance, scalability, and reliability properties. In particular, the cost of reliability is determined by comparing navigation performed over volatile and persistent states. In this setting, the effect of caching is also studied. The discussion shows that system throughput can be set to the desired level by performing navigation in parallel, when the kernel is replicated across multiple machines.

In order to leverage this extensive set of process-management tools, the authors are in the process of completing the integration of the new kernel in the existing BioOpera API [7]. They are also evaluating other possible implementations of the data-storage layer, such as using JDBC with a relational database, and they are planning to develop a mapping of the event-communication system to the Java Message Service (JMS) specification. More long-term plans include adding automatic support for dynamic system reconfiguration in response to variations in the system's workload and investigating the feasibility of using peer-to-peer technologies for distributed storage and event propagation.

REFERENCES

1. Agrawal, R.; Somani, A.; and Xu, Y. Storage and querying of e-commerce data. In P.M.G. Apers et al. (eds.), *Proceedings of the 27th International Conference on Very Large Data Bases.* San Francisco: Morgan Kaufmann, 2001, pp. 149–158.

2. Alonso, G.; Casati, F.; Kuno, H.; and Machiraju, V. *Web Services: Concepts, Architectures and Applications.* Heidelberg: Springer, 2003.

3. Auguston, M., and Delgado, A. Iterative constructs in the visual data flow language. In *Proceedings of the 1997 IEEE Symposium on Visual Languages.* Los Alamitos, CA: IEEE Computer Society, 1997, pp. 152–159.

4. Baeyens, T. Java business process management. www.jbpm.org.

5. Bauer, T., and Dadam, P. A distributed execution environment for large-scale workflow management systems with subnets and server migration. In A.L.P. Chen, W. Klas, and M.P. Singh (eds.), *Proceedings of the 2nd IFCIS International Conference on Cooperative Information Systems*). Los Alamitos, CA: IEEE Computer Society, 1997, pp. 99–108.

6. Bausch, W. OPERA-G: A microkernel for computational grids. PhD dissertation (ETH Zurich ETH Nr. 15395), Swiss Federal Institute of Technology, 2003.

7. Bausch, W.; Pautasso, C.; and Alonso, G. Programming for dependability in a service based grid. In S. Lee, S. Sekiguschi, S. Matsuoka, and M. Sato (eds.), *Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid.* Los Alamitos, CA: IEEE Computer Society, 2003, pp. 164–171.

8. Beguelin, A.; Dongarra, J.J.; Geist, A.; Manchek, R.; Moore, K.; Wade, R.; and Sunderam, V.S. HeNCE: Graphical development tools for network-based concurrent computing. In J. Saltz (ed.), *Proceedings of the 1992 Scalable High Performance Computing Conference.* Los Alamitos, CA: IEEE Computer Society Press, 1992, pp. 129–136.

9. Browne, J.C.; Hyder, S.I.; Dongarra, J.; Moore, K.; and Newton, P. Visual programming and debugging for parallel computing. *IEEE Parallel and Distributed Technology: Systems and Applications, 3*, 1 (spring 1995), 75–83.

10. Bussler, C. *B2B Integration: Concepts and Architecture*. Heidelberg: Springer, 2002.

11. Carriero, N., and Gelernter, D. *How to Write Parallel Programs*. Cambridge, MA: MIT Press, 1990.

12. Casati, F., and Shan, M.-C. Dynamic and adaptive composition of e-services. *Information Systems, 26* (2001), 143–163.

13. Collaxa. BPEL server and designer. www.collaxa.com.

14. Cox, P.T., and Song, B. A formal model for component-based software. In S. Levialdi (ed.), *Proceedings of the 2001 IEEE International Symposium on Human-Centric Computing Languages and Environments.* Los Alamitos, CA: IEEE Computer Society, 2001, pp. 304–311.

15. ebXML. ebXML business process specification schema (BPSS) 1.01, (2001). www.ebxml.org/specs/ebBPSS.pdf.

16. Freeman, E.; Hupfer, S.; and Arnold, K. *JavaSpaces: Principles, Patterns and Practice.* Reading, MA: Addison-Wesley, 1999.

17. Gelernter, D., and Carriero, N. Coordination languages and their significance. *Communications of the ACM, 35,* 2, (February 1992), 97–107.
18. Georgakopoulos, D.; Hornick, M.F.; and Sheth, A.P. An overview of workflow management: From process modelling to workflow automation infrastructure. *Distributed and Parallel Databases, 3*, 2 (April 1995), 119–153.
19. Geppert, A., and Tombros, D. Event-based distributed workflow execution with EVE. Technical Report 96.05. University of Zurich, Department of Computer Science, 1998.
20. Gu, X.; Nahrstedt, K.; Chang, R.N.; and Ward, C. QoS-assured service composition in managed service overlay networks. In F.M. Titsworth (ed.), *Proceedings of the 23rd International Conference on Distributed Computing Systems.* Los Alamitos, CA: IEEE Computer Society, 2003, pp. 194–201.
21. Heinl, P., and Schuster, H. Towards a highly scaleable architecture for workflow management systems. In R.R. Wagner and H. Thoma (eds.), *Proceedings of the 7th International Workshop on Database and Expert Systems Applications.* Los Alamitos, CA: IEEE Computer Society, 1996, pp. 439–444.
22. Hils, D.D. Visual languages and computing survey: Data flow visual programming languages. *Journal of Visual Languages and Computing, 3*, 1 (1992), 69–101.
23. IBM. BPEL4WS Java runtime. www.alphaworks.ibm.com/tech/bpws4j/.
24. IBM. TSpaces. www.almaden.ibm.com/cs/Tspaces/.
25. IBM, Microsoft, and BEA Systems. Business Process Execution Language for Web services (BPEL4WS) 1.0 (2002). www.ibm.com/developerworks/library/ws-bpel.
26. IBM, Microsoft, and BEA Systems. Web services coordination (WS-Coordination) (2002). www.ibm.com/developerworks/library/ws-coor/.
27. Jin, L.; Casati, F.; Sayal, M.; and Shan, M.-C. Load balancing in distributed workflow management system. In G.B. Lamont (ed.), *Proceedings of the ACM Symposium on Applied Computing.* Las Vegas: ACM Press, 2001, pp. 522–530.
28. Kamath, M.; Alonso, G.; Guenthoer, R.; and Mohan, C. Providing high availability in very large workflow management systems. In P.M.G. Apers, M. Bouzeghoub, and G. Gardarin (eds.), *Proceedings of the 5th International Conference on Advances in Database Technology.* Avignon, France: Springer, 1996, pp. 427–442.
29. Lehman, T.J.; Cozzi, A.; Xiong, Y.; Gottschalk, J.; Vasudevan, V.; Landis, S.; Davis, P.; Khavar, B.; and Bowman, P. Hitting the distributed computing sweet spot with TSpaces. *Computer Networks, 35*, 4, (March 2001), 457–472.
30. Leymann, F. Web services flow language (WSFL 1.0). IBM (2001). www-3.ibm.com/software/solutions/webservices/pdf/WSFL.pdf.
31. Mosconi, M., and Porta, M. Iteration constructs in data-flow visual programming languages. *Computer Languages, 26*, 2–4 (July 2000), 67–104.
32. Muench, M., and Schuerr, A. Leaving the visual language ghetto. In D.C. Martin (ed.), *Proceedings of the IEEE Symposium on Visual Languages.* Los Alamitos, CA: IEEE Computer Society, 1999, pp. 148–155.
33. Muth, P.; Wodtke, D.; Weissenfels, J.; Dittrich, A.; and Weikum, G. From centralized workflow specification to distributed workflow execution.

Journal of Intelligent Information Systems, 10, 2 (1998), 159–184.
34. Oasis. Universal Description, Discovery and Integration of Web services (UDDI) Version 3.0, (2002). http://uddi.org/pubs/uddi v3.htm.
35. Pautasso, C. JOpera: Process support for more than Web services. www.iks.ethz.ch/jopera/.
36. Pautasso, C., and Alonso, G. Visual composition of Web services. In P. Cox and J. Kosking (eds.), *Proceedings of the 2003 IEEE International Symposium on Human-Centric Computing Languages and Environments.* Los Alamitos, CA: IEEE Computer Society, 2003, pp. 92–99.
37. Schuldt, H.; Alonso, G.; Beeri, C.; and Schek, H.-J. Atomicity and isolation for transactional processes. *ACM Transactions on Database Systems, 27*, 1 (2002), 63–116.
38. Schuster, H., and Heinl, P. A workflow data distribution strategy for scalable workflow management systems. In B. Bryant, J. Carroll, J. Hightower, and K.M. George (eds.), *Proceedings of the 1997 ACM Symposium on Applied Computing.* San Jose, CA: ACM Press, 1997, pp. 174–176.
39. Thatte, S. XLANG: Web services for business process design. Microsoft (2001). www.gotdotnet.com/team/xml_wsspecs/xlang-c/.
40. van der Aalst, W.M.P. The application of Petri nets to workflow management. *Journal of Circuits, Systems and Computers, 8,* 1 (1998), 21–66.
41. van der Aalst, W.M.P. Don't go with the flow: Web services composition standards exposed. *IEEE Intelligent Systems, 18,* 1 (2003), 72–85.
42. van der Aalst, W.M.P., and Berens, P.J.S. Beyond workflow management: product driven case handling. In C. Ellis, and I. Zigurs (eds.), *Proceedings of the 2001 International ACM SIGGROUP Conference on Supporting Group Work.* Boulder, CO: ACM Press, 2001, pp. 42–51.
43. W3C. Simple Object Access Protocol (SOAP) 1.1 (2000). www.w3.org/TR/SOAP/.
44. W3C. Web Services Definition Language (WSDL) 1.1, (2001). www.w3.org/TR/wsdl/.
45. W3C. Web Services Choreography Interface (WSCI) 1.0, (2002) www.w3.org/TR/wsci/.
46. W3C. Web Services Conversation Language (WSCL) 1.0, (2002). www.w3.org/TR/wsc110/.
47. Wiederhold, G.; Wegner, P.; and Ceri, S. Towards megaprogramming: A paradigm for component-based programming. *Communications of the ACM, 35,* 11 (1992), 89–99.
48. Wirtz, G.; Weske, M.; and Giese, H. Extending UML with workflow modeling capabilities. In O. Etzion and P. Scheuermann (eds.), *7th International Conference on Cooperative Information Systems.* Eilat, Israel: Springer, 2000, pp. 30–41.
49. Wodtke, D.; Weissenfels, J.; Weikum, G.; and Kotz-Dittrich, A. The Mentor project: Steps toward enterprise-wide workflow management. In S.Y.W. Su (ed.), *Proceedings of the 12th International Conference on Data Engineering.* Los Alamitos, CA: IEEE Computer Society, 1996, pp. 556–565.

CESARE PAUTASSO (pautasso@inf.ethz.ch) received the computer science engineering degree from the Politecnico di Milano in 2000 and the Ph.D. degree from the Swiss

Federal Institute of Technology (ETH Zurich) in 2004. He is currently doing his post-doctoral research at ETH in the areas of process-support system, autonomic computing, and visual programming languages applied to Web service composition. He is a student member of the IEEE and the IEEE Computer Society.

GUSTAVO ALONSO (alonso@inf.ethz.ch) is a professor of computer science at the Swiss Federal Institute of Technology (ETH Zurich). He has an engineering degree in telecommunications from Madrid Polytechnic University (1989), and M.S. (1992) and Ph.D. (1994) degrees in computer science from the University of California at Santa Barbara. Before joining ETH, he worked at the IBM Almaden Research Center in San Jose, California. His general research interests include databases, distributed applications, mobile and pervasive computing, dynamic software adaptation, and proactive computing. He is a member of the ACM and the IEEE Computer Society.